dispRity manual

Thomas Guillerme

March 15, 2016

dispRity is a package for calculating disparity in R. It allows to summarise ordinated matrices (e.g. MDS, PCA, PCO, PCoA) into single values.

Contents

1 Before starting

1.1 Glossary

Because this package is aimed to be multidisciplinary, many names or terms used in this tutorial might be non-familiar to certain fields. Here is a list of what are the exact meaning of these term:

- **Ordinated space**: it designates the mathematical multidimensional object studied here. In morphometrics, this one is often referred as being the morphospace. However it can also be the the cladisto-space for cladistic data or the eco-space in ecology, etc. In practice, this term designates an ordinated matrix where the columns represent the dimensions of the ordinated space (often – but not necessarily – > 3!) and the rows represent the elements within this space.

- **Elements**: it designates the rows of the ordinated space, elements can be either taxa, field sites, countries, etc...
- **Dimensions**: it designates the columns of the ordinated space. The dimensions can also be referred to as axis or principal components.
- **Series**: it designates sub-samples of the ordinated space. Basically a series contain the same number of dimensions as the morphospace but may contain a smaller number of elements. For example, if our ordinated space is composed of birds and mammals (i.e. the elements) and 50 dimensions, we can create two series of just mammals or birds as elements (but with the same 50 dimensions) to look at the difference in disparity between the two clades.

1.2 Installation

You can install this package easily if you use the latest version of `R` and `devtools`.

```
if(!require(devtools)) install.packages("devtools")
install_github("TGuillerme/dispRity", ref = "release")
library(dispRity)
```

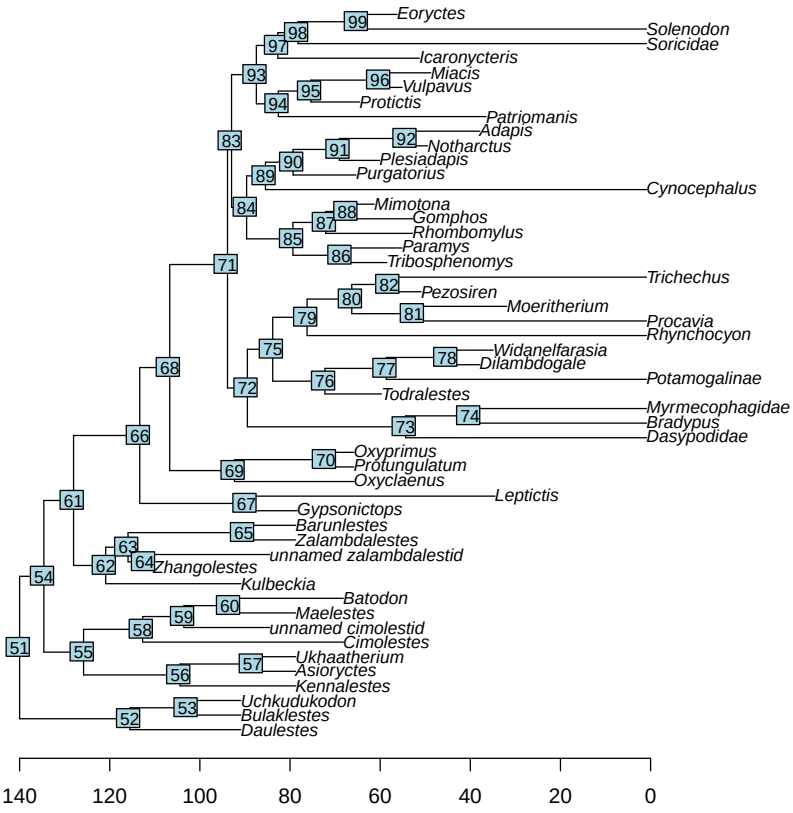
Note that we use the `release` branch here which is version 0.2. For the piping-hot (but potentially full of bugs) version, you can change the argument `ref = "release"` to `ref = "master"`. This package depends mainly on the `ape` package and uses functions of several other (`ade4`, `geometry`, `grDevices`, `hypervolume`, `paleotree` and `snow`).

1.3 Data

In this tutorial we are going to use a subset of the ordinated cladistic data from Beck and Lee (2014) that contains 50 taxa (elements) ordinated using their cladistic distance (i.e. the distance between their discrete morphological characters). Note that this data is more oriented towards palaebiology analysis but that it can apply to other disciplines. Please refer to the GitHub page: github.com/TGuillerme/dispRity for other vignettes covering some specific example.

```
## Loading the package
library(dispRity)

## Warning in .doLoadActions(where, attach): trying to execute load actions without 'methods'
package

## Setting the random seed for repeatability
set.seed(123)

## Loading the ordinated matrix containing 50 taxa
data(BeckLee_mat50)
dim(BeckLee_mat50)

## [1] 50 48

head(BeckLee_mat50[,1:5])

##                    [,1]          [,2]        [,3]       [,4]      [,5]
## Cimolestes   -0.5319679  0.1117759259  0.09865194 -0.1933148 0.2035833
## Maelestes    -0.4087147  0.0139690317  0.26268300  0.2297096 0.1310953
## Batodon      -0.6923194  0.3308625215 -0.10175223 -0.1899656 0.1003108
## Bulaklestes  -0.6802291 -0.0134872777  0.11018009 -0.4103588 0.4326298
## Daulestes    -0.7386111  0.0009001369  0.12006449 -0.4978191 0.4741342
## Uchkudukodon -0.5105254 -0.2420633915  0.44170317 -0.1172972 0.3602273
```

```r
## Loading another ordinated matrix containing 50 tips + 49 nodes
data(BeckLee_mat99)
dim(BeckLee_mat99)

## [1] 99 97

head(BeckLee_mat99[,1:5], 2)

##                  [,1]       [,2]       [,3]       [,4]        [,5]
## Cimolestes -0.6082437 -0.0323683 0.08458885 -0.4338448 -0.30536875
## Maelestes  -0.5730206 -0.2840361 0.01308847 -0.1258848  0.06123611

tail(BeckLee_mat99[,1:5], 2)

##            [,1]      [,2]       [,3]       [,4]       [,5]
## n48 -0.05529018 0.4799330 0.04118477 0.04944912 -0.3558830
## n49 -0.13067785 0.4478168 0.11956268 0.13800340 -0.3222785

## Loading a list of first and last occurrence data
data(BeckLee_ages)
head(BeckLee_ages)

##             FAD  LAD
## Adapis     37.2 36.8
## Asioryctes 83.6 72.1
## Leptictis  33.9 33.3
## Miacis     49.0 46.7
## Mimotona   61.6 59.2
## Notharctus 50.2 47.0

## Loading the phylogeny
data(BeckLee_tree)
plot(BeckLee_tree, cex = 0.8) ; nodelabels(cex = 0.8) ; axisPhylo(root = 140)
```

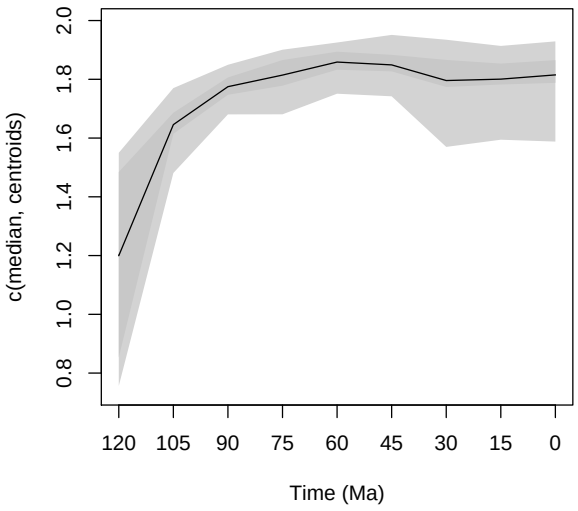


1.4 A quick go through

Here is a really crude and quick analysis to go through the package, showing some of its features. Note that all these features will be discussed in more details below.

```
## Splitting the data
sliced_data <- time.series(BeckLee_mat99, BeckLee_tree, method = "continuous",
    model = "acctran", time = rev(seq(from=0, to=130, by=15)),
    FADLAD = BeckLee_ages)

## Bootstrapping the data
bootstrapped_data <- boot.matrix(sliced_data, 100)

## Calculating disparity
median_centroids <- dispRity(bootstrapped_data, metric = c(median, centroids))

## Summarising the results
summary(median_centroids)

##   series  n observed  mean  2.5%   25%   75% 97.5%
## 1    120  5    1.494 1.200 0.756 0.854 1.484 1.550
## 2    105 11    1.725 1.646 1.481 1.614 1.686 1.770
```

```
## 3     90 18    1.820 1.775 1.681 1.747 1.806 1.849
## 4     75 19    1.923 1.814 1.681 1.778 1.865 1.901
## 5     60 20    1.946 1.859 1.751 1.833 1.894 1.925
## 6     45 14    1.930 1.849 1.742 1.827 1.883 1.951
## 7     30 10    1.910 1.796 1.570 1.774 1.866 1.935
## 8     15 10    1.910 1.801 1.594 1.782 1.853 1.914
## 9      0 10    1.910 1.815 1.588 1.788 1.865 1.929

plot(median_centroids)

## Testing the effect of time on disparity
summary(test.dispRity(median_centroids, test = aov, comparisons = "all"))

##              Df Sum Sq Mean Sq F value Pr(>F)
## series        8  34.47   4.309   279.5 <2e-16 ***
## Residuals   891  13.74   0.015
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```



2 Package specificities

2.1 The `dispRity` objects

Disparity analysis can involve a lot of shuffling around with many matrices (especially when bootstrapping the data) which can be impractical to visualise and will quickly jam your `R` console. For example, we can have a look at the structure of the object created in the quick example:

```
str(median_centroids)
## That's a more than 4000 lines of output!
```

Therefore this package proposes a specific class of object called `dispRity` objects. These objects allow to easily use S3 method functions such as `summary.dispRity` (just called as `summary`; see section 3.5) or `plot.dispRity` (just called as `plot`; see section 3.6). But also, this allows to use the S3 method for printing

dispRity objects via print.dispRity that allows to summarise the content of the objects similar to the phylo class objects (see ape::print.phylo).

```
## What is the class of the median_centroids object?
class(median_centroids)

## [1] "dispRity"

## What is in the object?
names(median_centroids)

## [1] "data"       "disparity" "elements"  "series"    "call"

## Summarising it using the S3 method print.dispRity:
median_centroids

## Disparity measurements across 9 series for 99 elements
## Series:
## 120, 105, 90, 75, 60, 45 ...
## Disparity calculated as: c(median, centroids) for 97 dimensions.
## Data was split using continuous method.
## Data was bootstrapped 100 times, using the full bootstrap method.

## This displays some basic information about the object content
```

Note however, that it is always possible to recall the full object using the argument all=TRUE:

```
## Displaying the full object
print(median_centroids, all = TRUE)
## But that's 155091 lines!
```

Finally, some utility functions such as get.dispRity or extract.dispRity allows to access to some specific content of the object:

```
## Extracting some specific series from the disparity object
series_1_and_4 <- get.dispRity(median_centroids, what = c(1,4))
series_1_and_4

## Disparity measurements across 2 series for 24 elements
## Series:
## 120, 75.
## Disparity calculated as: c(median, centroids) for 97 dimensions.
## Data was split using continuous method.
## Data was bootstrapped 100 times, using the full bootstrap method.

## The observed disparity
extract.dispRity(median_centroids)

##      120      105       90       75       60       45       30       15        0
## 1.493570 1.725280 1.820292 1.922531 1.945952 1.930306 1.910203 1.910203 1.910203

## The distribution of the bootstrapped disparity scores for the first series
str(extract.dispRity(get.dispRity(median_centroids, what = 1),
    observed = FALSE))

## List of 1
##  $ 120: num [1:100] 1.52 0.786 1.555 1.288 0.891 ...
```

2.2 Modular functions

This package aims to be a modular package where users can personalise some aspects of the package fairly easily. The modular aspect allows the package users to use several implemented tools to help them creating personalised functions. For example, the `dispRity` function intake a `metric` argument designating how disparity should be calculated. This argument can take some already implemented functions such as `mean` but also some completely new ones such as the sum divided by length:

```
## Disparity measured as the mean value of the ordinated matrix
summary(dispRity(sliced_data, metric = mean))

##   series  n observed
## 1    120  5   -0.005
## 2    105 11   -0.010
## 3     90 18   -0.010
## 4     75 19   -0.001
## 5     60 20    0.005
## 6     45 14    0.013
## 7     30 10    0.017
## 8     15 10    0.017
## 9      0 10    0.017

## A function for measuring the sum divided by the length (the mean!)
sum.divide.by.length <- function(X) sum(X)/length(X)

## Disparity measured as the mean divided by the length
summary(dispRity(sliced_data, metric = sum.divide.by.length))

##   series  n observed
## 1    120  5   -0.005
## 2    105 11   -0.010
## 3     90 18   -0.010
## 4     75 19   -0.001
## 5     60 20    0.005
## 6     45 14    0.013
## 7     30 10    0.017
## 8     15 10    0.017
## 9      0 10    0.017

## Giving the exact same results!
```

For more information, please refer to the GitHub page: github.com/TGuillerme/dispRity for the vignette about the metric implementation in `dispRity` .

3 Functions

In the following section, different functionalities are detailed. The functions are presented in a "classic" pipelined study fashion for convenience (i.e. splitting the data, bootstrapping it, calculating disparity, testing differences). Note however that most of these functions can be used in any order and that some steps can of course be skipped.

One of the first steps in disparity analysis is to split the ordinated space into sub-samples of this space (hereafter *series*). These series correspond to regions of the ordinated space that we want to compare to each other.

3.1 cust.series

The cust.series function allows to create these series according to factors determined by the user. In this example, we can split the matrix based on the phylogeny and separate to different groups of taxa: the crown mammals (all the living mammals and their direct ancestor) and the stem mammals (all the mammals that have no direct offspring).

```
## We want to separate the species around the node 71 (see phylogeny above).
## All the descendant of these node are crown and all the ancestors are stem.
crown <- extract.clade(BeckLee_tree, node = 71)$tip.label
stem <- drop.tip(BeckLee_tree, tip = crown)$tip.label
## We then have to feed this information in a data frame with one column
factors <- as.data.frame(
    matrix(data = c(rep("crown", length(crown)), rep("stem", length(stem))),
    ncol = 1, dimnames = list(c(crown, stem), "Group")))

## We then can use the customised series function to create the two series
## i.e. the two regions of the ordinated space
crown_stem <- cust.series(BeckLee_mat50, factors)

## This created a dispRity object containing two series: crown and stem
class(crown_stem)

## [1] "dispRity"

crown_stem

## 2 custom series for 50 elements
## Series:
## Group.crown, Group.stem.

## This object contains three elements
names(crown_stem)

## [1] "data"     "elements" "series"

## With "data" being the list sub-matrices
str(crown_stem$data)

## List of 2
##  $ Group.crown: num [1:30, 1:48] 0.3079 0.6531 0.5089 -0.1652 -0.0419 ...
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : chr [1:30] "Dasypodidae" "Bradypus" "Myrmecophagidae" "Todralestes" ...
##   .. ..$ : NULL
##  $ Group.stem : num [1:20, 1:48] -0.739 -0.68 -0.511 -0.477 -0.473 ...
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : chr [1:20] "Daulestes" "Bulaklestes" "Uchkudukodon" "Kennalestes" ...
##   .. ..$ : NULL

## "elements" being the list of taxa in the original ordinated matrix
str(crown_stem$elements)

##  chr [1:50] "Cimolestes" "Maelestes" "Batodon" "Bulaklestes" "Daulestes" ...

## and finally, "series" containing information on the series type (custom)
## and names (crown and stem)
crown_stem$series

## [1] "custom"      "Group.crown" "Group.stem"
```

Note that here we separating the taxa into two exclusive groups (i.e. crown mammals can be crown only, same for stem mammals). It is however possible to add more factors with overlapping groups:

```
## Adding all mammals to the a new series (along with crown and stem as before)
factors2 <- as.data.frame(
    matrix(data = c(rep("crown", length(crown)), rep("stem", length(stem)),
    rep("Mammalia", Ntip(BeckLee_tree))), ncol = 2,
    dimnames = list(c(crown, stem), c("Group", "Class")), byrow = FALSE))

## Creating the three series: Group.stem/crown and Class.Mammalia
cust.series(BeckLee_mat50, factors2)

## 3 custom series for 50 elements
## Series:
## Group.crown, Group.stem, Class.Mammalia.
```

3.2 `time.series`

Another way to split the ordinated space (maybe more relevant to palaeobiologists) is to do it according to time. The `time.series` function allows to create series that contain all the elements present at specific points or during specific periods in time. This functions needs as input an ordinated space and a matching phylogenetic tree. Two types of time series can be performed by using the `method` option:

1. discrete time series (or time-binning) using `method = "discrete"`;
2. continuous time series (or time-slicing) using `method = "continuous"`.

For the time-slicing method details see Guillerme and Cooper (in prep.). For both methods, the function intakes the `time` argument which can be a vector of `numeric` values for:

1. defining the boundaries of the time bins (when `method = "discrete"`);
2. defining the time slices (when `method = "continuous"`).

Otherwise, the `time` argument can be set as a single `numeric` value for automatically generating a given number of equidistant time-bins/slices. Additionally, it is also possible to input a data frame containing the first and last occurrence data (`FAD/LAD`) for taxa that span over a longer time than the tips/nodes age.

Here is an example for `method = "discrete"`:

```
## Generating three time bins containing the data present every 40 Ma
time.series(data = BeckLee_mat50, tree = BeckLee_tree, method = "discrete",
    time = c(120, 80, 40, 0))

## 3 discrete series for 50 elements
## Series:
## 120-80, 80-40, 40-0.
```

In this example, the taxa where split inside each timebin according to their age. Note however that some taxa might span between some time bins and should be included in more than one. This is possible by providing a table containing the first and last occurrence data:

```
## Generating three time bins containing the data present every 40 Ma, including
## taxa that might span between time bins.
time_bins <- time.series(data = BeckLee_mat50, tree = BeckLee_tree,
    method = "discrete", time = c(120, 80, 40, 0), FADLAD = BeckLee_ages)
time_bins
```

```
## 3 discrete series for 50 elements
## Series:
## 120-80, 80-40, 40-0.
```

This generated indeed a list of 3 sub matrices. Note that we can also generate equivalent results by just telling the function that we want three time-bins (series) as follow:

```
## Automatically generate three equal length bins:
time.series(data = BeckLee_mat50, tree = BeckLee_tree, method = "discrete",
    time = 3)

## 3 discrete series for 50 elements
## Series:
## 133.51104-89.00736, 89.00736-44.50368, 44.50368-0.
```

We now have three time bins of 44.50368 million years each.

When using this method, the oldest boundary of the first bin (or the first slice, see below) is automatically generated as being the root age + 1% of the tree length as long as at least three elements are present at that point in time. The algorithm adds an extra 1% tree length until reaching the required minimum of three elements. It is also possible to include nodes in each bin by using `inc.nodes = TRUE` and providing a matrix that contains the ordinated distance between tips *and* nodes.

For the time-slicing method (`method = "continuous"`), the idea is really similar. This option intakes a matrix that contains the ordinated distance between taxa *and* nodes and an assumed evolutionary model via the `model` argument:

1. `"acctran"` where the data chosen on each time slice is always the one of the offspring
2. `"deltran"` where the data chosen on each time slice is always the one of the descendant
3. `"punctuated"` where the data chosen on each time slice is randomly chosen between the offspring or the descendant
4. `"gradual"` where the data chosen on each time slice is either the offspring or the descendant depending on branch length

```
## Generating four time slices every 40 million years assuming a gradual
## evolution model
time_slices <- time.series(data = BeckLee_mat99, tree = BeckLee_tree,
    method = "continuous", model = "gradual", time = c(120, 80, 40, 0),
    FADLAD = BeckLee_ages)
time_slices

## 4 continuous series for 99 elements
## Series:
## 120, 80, 40, 0.

## Note that in the same way as for the discrete method, we can also
## automatically generate the slices
time.series(data = BeckLee_mat99, tree = BeckLee_tree, method = "continuous",
    model = "gradual", time = 4)

## 4 continuous series for 99 elements
## Series:
## 133.51104, 89.00736, 44.50368, 0.
```

3.3 boot.matrix

Once we obtain our different series, we might want to bootstrap and rarefy it (i.e. pseudo-replicating the data). The bootstrap will allow us to make each series more robust to outliers and the rarefaction will allow us to compare slices with the same number of elements to get rid of eventual sampling problems. The `boot.matrix` allows to bootstraps and rarefy ordinated matrices in a fast and easy way. The default options will bootstrap the matrix 1000 times without rarefaction. The number of bootstrap pseudo-replicates can be defined using the `bootstraps` option (see below).

```
## Bootstrapping a matrix
boot.matrix(data = BeckLee_mat50)

## Bootstrapped ordinated matrix with 50 elements
## 1
## Data was bootstrapped 1000 times, using the full bootstrap method.

## As we can see, the output is also a dispRity object that is summarised
## automatically, and gives information on the data as well as the number of
## bootstraps and the bootstraps methods.
```

This function allows to control the bootstrap algorithm through the `boot.type` argument. Currently two algorithms are implemented:

1. `"full"` where the bootstrapping is entirely stochastic (all the data is bootstrapped)
2. `"single"` where only one random elements is replaced by one other random elements each pseudo-replication

This function also allows to rarefy the data using the `rarefaction` argument. The default argument is `FALSE` but it can be set to `TRUE` to fully rarefy the data (i.e. remove n elements for the number of pseudo-replicates, where n varies from the maximum number of elements present in the dataset to a minimum of 3 elements). It can also be set to a fix `numeric` value (or a set of `numeric` values). Finally, one last argument, `rm.last.axis` allows to remove a certain amount of dimensions (or axis) for the ordinated space. This can be logical argument where `FALSE` (default) will not remove any dimension and `TRUE` will remove the last dimensions that bear up to 5% of the total ordinated space's variance.

```
## Bootstrapping with the single bootstrap method
boot.matrix(BeckLee_mat50, boot.type = "single")

## Bootstrapped ordinated matrix with 50 elements
## 1
## Data was bootstrapped 1000 times, using the single bootstrap method.

## Bootstrapping with the full rarefaction
boot.matrix(BeckLee_mat50, bootstraps = 20, rarefaction = TRUE)

## Bootstrapped ordinated matrix with 50 elements
## 1
## Data was bootstrapped 20 times, using the full bootstrap method.
## Data was fully rarefied (down to 3 elements).

## Or with a set number of rarefaction levels
boot.matrix(BeckLee_mat50, bootstraps = 20, rarefaction = c(6:8,3))

## Bootstrapped ordinated matrix with 50 elements
## 1
## Data was bootstrapped 20 times, using the full bootstrap method.
## Data was rarefied with a maximum of 6, 7, 8 and 3 elements
```

```
## And removing the last axis (default)
boot.matrix(BeckLee_mat50, rm.last.axis = TRUE)

## Bootstrapped ordinated matrix with 50 elements
## 1
## Data was bootstrapped 1000 times, using the full bootstrap method.
## The 6 last axis were removed from the original ordinated data.

## Or with a fix value (50%)
boot.matrix(BeckLee_mat50, rm.last.axis = 0.5)

## Bootstrapped ordinated matrix with 50 elements
## 1
## Data was bootstrapped 1000 times, using the full bootstrap method.
## The 35 last axis were removed from the original ordinated data.
```

Of course, one could be interested in directly supplying the sub-sampled matrices generated above directly to this function. In fact, it can also deal with a list of matrices or with a `dispRity` object output from the `cust.series` or `time.series` functions.

```
## Bootstrap and full rarefaction on the crown/stem series
crown_stemBS <- boot.matrix(crown_stem, bootstraps = 100, rarefaction = TRUE)

## Bootstrap on the time binning/slicing series
time_binsBS <- boot.matrix(time_bins, bootstraps = 100)
time_slicesBS <- boot.matrix(time_slices, bootstraps = 100)

## Note that all these objects are of class dispRity
crown_stemBS

## Bootstrapped ordinated matrix with 50 elements
## Series:
## Group.crown, Group.stem.
## Data was split using custom method.
## Data was bootstrapped 100 times, using the full bootstrap method.
## Data was fully rarefied (down to 3 elements).

time_binsBS

## Bootstrapped ordinated matrix with 50 elements
## Series:
## 120-80, 80-40, 40-0.
## Data was split using discrete method.
## Data was bootstrapped 100 times, using the full bootstrap method.

time_slicesBS

## Bootstrapped ordinated matrix with 99 elements
## Series:
## 120, 80, 40, 0.
## Data was split using continuous method.
## Data was bootstrapped 100 times, using the full bootstrap method.
```

3.4 dispRity

This function is a modular function that allows to simply (and quickly!) calculate disparity from a matrix.

Because disparity can be measured in many ways, this function is a tool to measure disparity *as defined by the user*. In fact, the `dispRity` function intakes two main arguments: the data and the disparity metric. The disparity metric is a function or a set of functions that summarises the ordinated matrix to a single value that represents, in our example, the diversity of morphologies.

The `dispRity` algorithm decomposes the metrics functions into three levels that correspond to the dimensions of the output of each metric function. For more details on this algorithm please refer to the metric vignette.

In practice, the `dispRity` function intakes one or more functions as a definition of disparity. Several of these functions will be already implemented in other packages (such as `stats::median`, `base::sum`, etc.); some others are implemented in this package (listed in `?dispRity.metric`) and finally some others will be defined by the users. The `make.metric` function is designed to help users create and test their own disparity metric functions. In practice, the use of these metrics in the `dispRity` function, is pretty easy:

```
dispRity(BeckLee_mat50, metric = mean)

## Disparity measurements across 1 series for 50 elements
## 1
## Disparity calculated as: mean for 48 dimensions.

## This defines disparity as the mean value of the morphospace
## (using base::mean).

## It is also possible to combine multiple functions:
dispRity(BeckLee_mat50, metric = c(sum, variances))

## Disparity measurements across 1 series for 50 elements
## 1
## Disparity calculated as: c(sum, variances) for 48 dimensions.

## Defining disparity as the sum (base::sum) of the variances
## (dispRity::variances) of each dimension of the morphospace.

dispRity(BeckLee_mat50, metric = c(prod, centroids))

## Disparity measurements across 1 series for 50 elements
## 1
## Disparity calculated as: c(prod, centroids) for 48 dimensions.

## For the product (base::prod) of the distances between each elements and the
## centroid of the morphospace (dispRIty::centroids).

## Or user defined ones:
total.range <- function(X) abs(range(X)[1]-range(X)[2])
dispRity(BeckLee_mat50, metric = c(total.range, centroids))

## Disparity measurements across 1 series for 50 elements
## 1
## Disparity calculated as: c(total.range, centroids) for 48 dimensions.

## For the range (total.range, user defined) of the distances between each
## elements and the centroid of the morphospace (dispRity::centroids).

## Or more complex ones:
```

```
vars1 <- dispRity(BeckLee_mat50, metric = c(sd, variances, var))
## For the standard deviation (stats::sd) of the variance (dispRity::variances)
## of each column of the variance/covariance matrix (stats::var).

## Note that the order of the function is not important since the levels of
## each function are automatically detected by the dispRity algorithm.
vars2 <- dispRity(BeckLee_mat50, metric = c(variances, var, sd))
all(summary(vars1) == summary(vars2))

## [1] TRUE
```

Note that these functions do not directly output the disparity values but only the summary of the `dispRity` objects. To display the results, see the section 3.5.

In these examples we used only simple ordinated matrix but of course, it might be more interesting to directly use the `dispRity` objects we generated in the steps above. Thus we can calculate the bootstrapped and rarefied disparity in each sub-sample of the morphospace. In this example we are going to define disparity as being the median distance between each element and each dimensions centroids.

```
## Disparity in crown and stem mammals:
disp_crown_stemBS <- dispRity(crown_stemBS, metric = c(median, centroids))

## Disparity through time:
disp_time_binsBS <- dispRity(time_binsBS, metric = c(median, centroids))
disp_time_slicesBS <- dispRity(time_slicesBS, metric = c(median, centroids))
```

Note that the computational time was longer for the rarefied data (`crown_stemBS`) since it had to calculate the disparity on all the rarefied matrix, each bootstrapped 100 times. For both the `boot.matrix` and the `dispRity` functions, it is possible to increase computational speed in some cases using the `parallel` option (see each function's manuals).

3.5 summary

This function is a S3 function (`summary.dispRity`) allowing to summarize the content of `dispRity` objects that contain disparity calculations. This function intakes a `dispRity` object plus various options namely the `quantile` values for the confidence intervals levels; the `cent.tend` for the central tendency to use for summarising the results and two *visual* options which are whether to recall the `dispRity` options and how much digits are wanted in the results.

```
summary(disp_time_binsBS)

##   series  n observed  mean  2.5%   25%   75% 97.5%
## 1 120-80  8    1.203 1.103 0.895 1.057 1.178 1.236
## 2  80-40 27    1.344 1.322 1.275 1.306 1.339 1.366
## 3   40-0 16    1.353 1.323 1.265 1.304 1.346 1.368

## By default, the 50 and 95 quantiles and the mean are calculated.
## Note that the function also displays the observed disparity
## (non-bootstrapped).

## These arguments can be changed easily as follow
summary(disp_time_binsBS, quantile = 88, cent.tend = sd, rounding = 1)

##   series  n observed  sd  6% 94%
## 1 120-80  8      1.2 0.1 1.0 1.2
## 2  80-40 27      1.3 0.0 1.3 1.4
## 3   40-0 16      1.4 0.0 1.3 1.4
```

```
## If on happens to forgot what was calculated in the summarised object,
## it is possible to recall the different steps details using recall:
summary(disp_time_binsBS, recall = TRUE)

## Disparity calculated as: c(median, centroids) for 48 dimensions.
## Data was split using discrete method.
## Data was bootstrapped 100 times, using the full bootstrap method.
##   series  n observed  mean  2.5%   25%   75% 97.5%
## 1 120-80  8    1.203 1.103 0.895 1.057 1.178 1.236
## 2  80-40 27    1.344 1.322 1.275 1.306 1.339 1.366
## 3   40-0 16    1.353 1.323 1.265 1.304 1.346 1.368

## Note that the information with each different number of elements is
## displayed for rarefied data:
tail(summary(disp_crown_stemBS))

##        series  n observed  mean  2.5%   25%   75% 97.5%
## 41 Group.stem 15       NA 1.229 1.146 1.210 1.253 1.282
## 42 Group.stem 16       NA 1.232 1.154 1.217 1.250 1.279
## 43 Group.stem 17       NA 1.233 1.174 1.218 1.251 1.281
## 44 Group.stem 18       NA 1.236 1.175 1.219 1.255 1.282
## 45 Group.stem 19       NA 1.239 1.173 1.226 1.256 1.286
## 46 Group.stem 20     1.27 1.240 1.185 1.225 1.258 1.280

## The observed disparity is NA for rarefied data.
```

3.6 `plot`

An alternative way to summarise the data is to plot the results using the S3 method `plot.dispRity`. This function intakes the same options as `summary.dispRity` along side with various graphical options described in the function manual (see `?plot.dispRity`).

Two main categories of plots can be displayed: continuous plots with the argument `type = "continuous"` (see below) and discrete plots (default is `type = "box"`. Note that this argument can be left missing. In this case, the algorithm will automatically detect the type of series from the `dispRity` object.

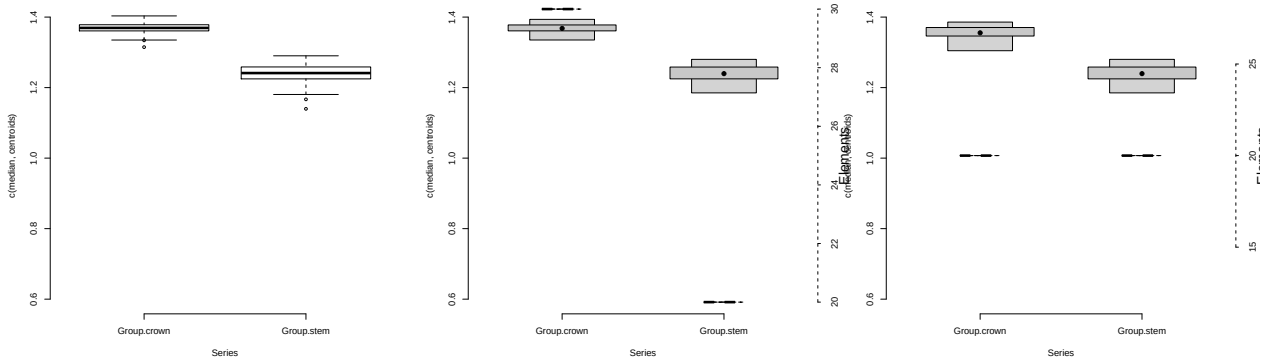
The discrete plot arguments can be either:

- `type = "box"` for plotting boxplots.
- `type = "polygon"` for plotting boxes that represent the different levels of confidence intervals (with a dot for the central tendency).
- `type = "lines"` which is similar to `type = "polygon"` but plots lines rather than boxes for the confidence intervals levels.

It is also possible to display the number of elements per series (has a horizontal doted line) using the option `elements = TRUE`. Additionally, when the data is rarefied (see section 3.3), one can also indicate which level of rarefaction to display (i.e. display the results for a certain number of elements) by using the `rarefaction` argument.

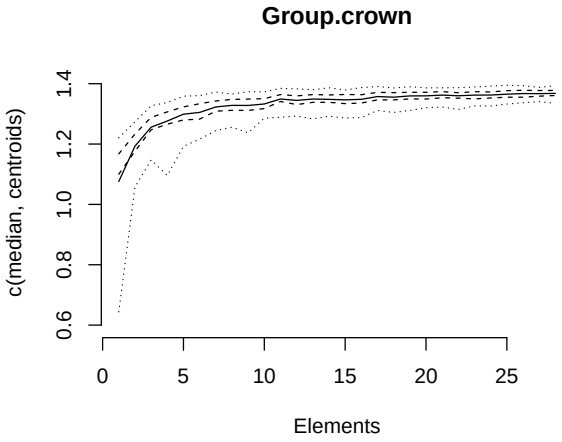
```
## Graphical options
quartz(width = 15, height = 5) ; par(mfrow = (c(1,3)), bty = "n")

## Plotting the disparity for both groups
plot(disp_crown_stemBS, type = "box")
```

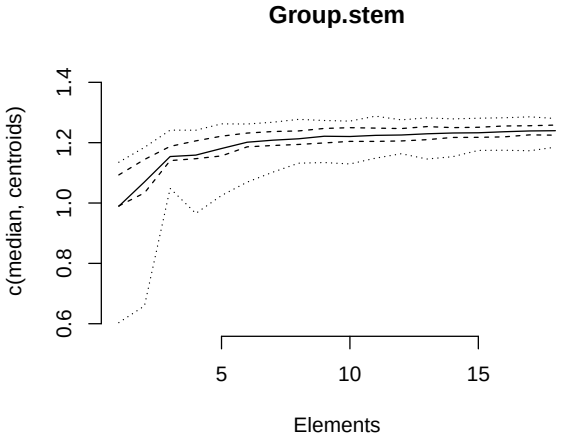
```
## Same plot but with the number of elements on the right and using the
## "polygon" plot type.
plot(disp_crown_stemBS, type = "polygon", elements = TRUE)
## Rarefied version of the same plot (20 elements per series)
plot(disp_crown_stemBS, type = "polygon", elements = TRUE, rarefaction = 20)
```



It is also possible to easily look at the behaviour of the disparity metric with the rarefied data (i.e. how does the disparity score change when the number of elements goes down) by using the `rarefaction = "plot"` argument:

```
## Graphical options
par(bty = "n")
## The rarefaction curves
plot(disp_crown_stemBS, elements = TRUE, rarefaction = "plot")
```

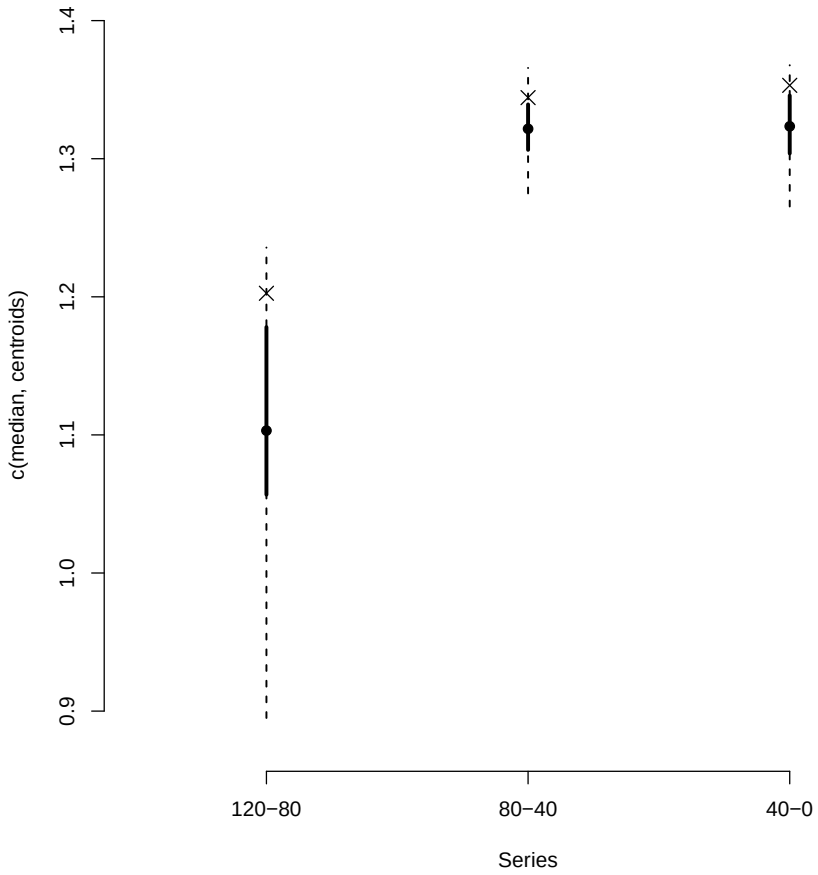




The different dashed lines represent the different confidence intervals around each rarefaction curve. We can see that the disparity metric becomes stable after more than 10 elements.

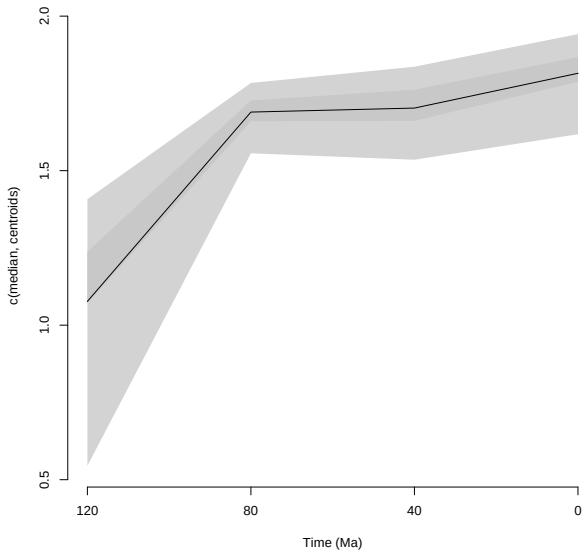
It is also possible, when using the `type = "lines"` argument, to plot lines rather than boxes. This option can be useful when displaying many series in a same window. Additionally, it is possible to plot the observed data when existing (i.e. not when the data is rarefied) by using the option `observed = TRUE`.

```
## Graphical options
par(bty = "n")
## The disparity-through-time data in a time-binned way (with lines rather than
## boxes) with the observed data as crosses
plot(disp_time_binsBS, type = "lines", observed = TRUE)
```



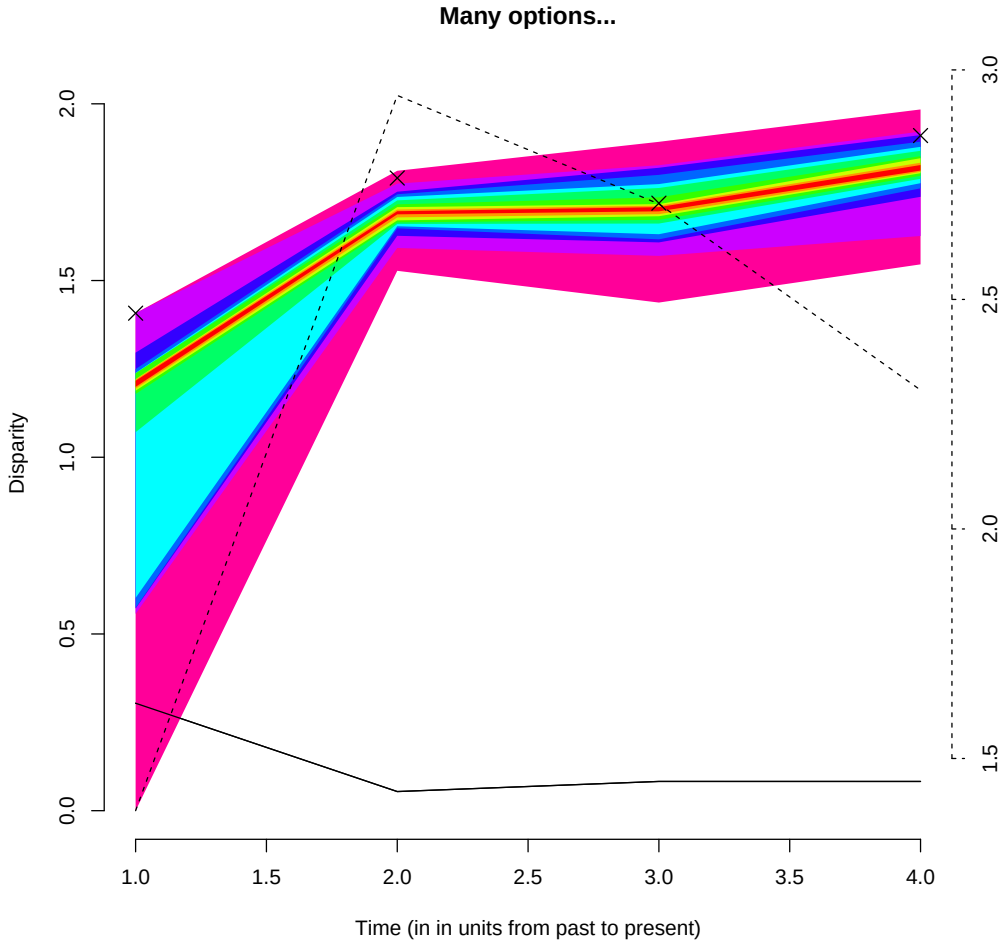
The `type = "continuous"` argument plots superimposed continuous (along the x axis) polygons that represent the confidence intervals and a continuous line that represents the central tendency of the data. Note that, as a S3 method function, `plot.dispRity` can take the classic graphical options from the `plot` function or even to add other plots to the previous ones.

```
## Graphical options
par(bty = "n")

## First, plotting the continuous data with a y axis limit
plot(disp_time_slicesBS, type = "continuous")
## Note that the type = "continuous" argument is not mandatory here.
```



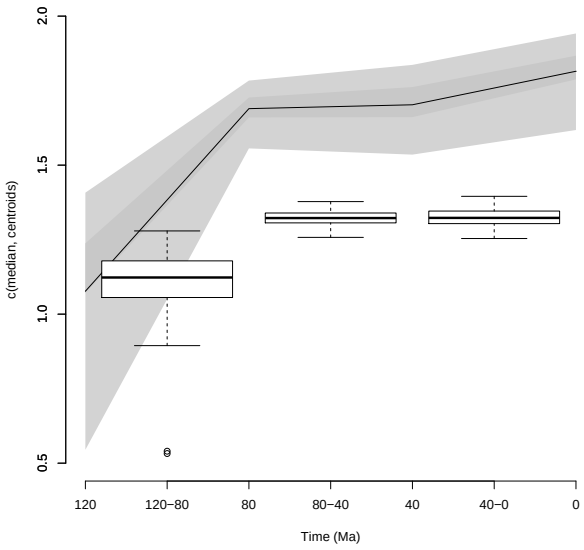
I encourage you to play with the graphical options to have some prettier results. Note that most of the options from `plot` can be passed to `plot.dispRity` via `...`.

```
## Graphical options
par(bty = "n")
## A plot with many options!
plot(disp_time_slicesBS, quantile = c(seq(from=10, to=100, by=10)),
    cent.tend = sd, type = "continuous", elements = "log",
    col = c("black", rainbow(10)), ylab = c("Disparity", "log(Diversity)"),
    xlab = "Time (in in units from past to present)", time.series = FALSE,
    observed = TRUE, main = "Many options...")
```



Finally, it is possible to add multiple plots on top of each other:

```
## Graphical options
par(bty = "n")

## First, plotting the continuous data with a y axis limit
plot(disp_time_slicesBS, ylim = c(0.5,2))

## Second, adding the discrete data
par(new = TRUE)
plot(disp_time_binsBS, discrete_type = "line", ylim=c(0.5,2), xlab="", ylab="")
```



3.7 `test.dispRity`

Finally, the `dispRity` package allows to apply some tests to the data in order to properly test hypothesis. The function `test.dispRity` works in a similar way as the `dispRity` function: it intakes a `dispRity` object (containing disparity measurements), a `test` and a `comparisons` argument. The `test` argument can be any statistical or non-statistical test to apply to the disparity object. It can be common test function (e.g. `stats::t.test`) or some functions implemented in `dispRity` (see `dispRity.test`) or even user defined functions. The `comparisons` argument must indicate the way the test should be applied to the data:

- `"pairwise"` (default): to compare each series pairwise
- `"referential"`: to compare each series to the first one
- `"sequential"`: to compare each series to the following one
- `"all"`: to compare all the series together (like in analysis of variance)

It is also possible to input a list of pairs of `numeric` values or `characters` matching the series names to create personalised test. Some other specific tests implemented in `dispRity` such as the `dispRity::null.test` and the `dispRity::sequential.test` have a specific way to be applied to the data and therefore ignore the `comparisons` argument.

This function also allows to correct for type I error inflation when using multiple comparisons via the `correction` argument. This argument can be empty (no correction applied) or can contain one of the corrections from the `stats::p.adjust` function (see `?p.adjust`).

Note that the `test.dispRity` algorithm deals with some classic tests outputs (`h.test`, `lm` and `numeric` vector) and summarises the test output. It is however possible to get the full detailed output by using the options `details = TRUE`.

```
## Performing a pairwise t-test to the test a difference in disparity between
## crown and stem mammals (note that "pairwise" is default)
test.dispRity(disp_crown_stemBS, test = t.test)

##                                t       df      p.value mean of x mean of y
## Group.crown - Group.stem 42.25372 164.8723 2.303646e-90   1.36798  1.239652
```

```
## The same test but with the detailed output
test.dispRity(disp_crown_stemBS, test = t.test, details = TRUE)
```

```
## $`Group.crown - Group.stem`
##
##  Welch Two Sample t-test
##
## data:  data[[list_of_comp[[1]]]] and data[[list_of_comp[[2]]]]
## t = 42.254, df = 164.87, p-value < 2.2e-16
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  0.1223319 0.1343251
## sample estimates:
## mean of x mean of y
##  1.367980  1.239652
```

```
## A wilcoxon rank test applied to sliced disparity the data sequentially with
## the Bonferroni correction.
test.dispRity(disp_time_slicesBS, test = wilcox.test,
    comparisons = "sequential", correction = "bonferroni")
```

```
##             W      p.value
## 120 - 80    0 7.457311e-34
## 80 - 40  4448 5.334298e-01
## 40 - 0   1600 2.961960e-16
```

```
## An analysis of variance (aov) applied to the same data
test.dispRity(disp_time_slicesBS, test = aov, comparisons = "all")
```

```
## Call:
##    test(formula = data ~ series, data = data)
##
## Terms:
##                   series Residuals
## Sum of Squares  33.50243  10.79609
## Deg. of Freedom        3       396
##
## Residual standard error: 0.1651147
## Estimated effects may be unbalanced
```

```
## Note that in this case, the output is the regular aov output:
slice_aov <- test.dispRity(disp_time_slicesBS, test = aov, comparisons = "all")
class(slice_aov) ; summary(slice_aov)
```

```
## [1] "aov" "lm"
##              Df Sum Sq Mean Sq F value Pr(>F)
## series        3   33.5  11.167   409.6 <2e-16 ***
## Residuals   396   10.8   0.027
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
## Measuring the overlap between distributions in the time bins (using the
## implemented Bhattacharyya Coefficient function - see ?bhatt.coeff) with a
## personalised series comparisons and some additional arguments to pass to the
## test function (bw = bw.nrd)
test.dispRity(disp_time_binsBS, test = bhatt.coeff, bw = bw.nrd,
    comparisons = list(c(1,2), c(1,3), c(2,1), c(3,1)))
```

```
##                bhatt.coeff
## 120-80 - 80-40  0.04472136
## 120-80 - 40-0   0.03741657
## 80-40 - 120-80  0.04472136
## 40-0 - 120-80   0.03741657
```

3.8 utilities

`tree.age`

This function allows to calculate the age of each individual nodes and tips in a tree. It can either use the root age of the tree (if present as `$root.time`) or else calculate the age using a user defined root age via the `age` argument. Also, it is possible to decide whether the time is calculated towards the past (e.g. million years ago) or towards the present (e.g. in time since the origin).

```
## This tree has a root age
BeckLee_tree$root.time

## [1] 139.074

## So we can get the age of each tips and nodes directly
head(tree.age(BeckLee_tree), 5)

##    ages     elements
## 1 90.00    Daulestes
## 2 90.00  Bulaklestes
## 3 90.00 Uchkudukodon
## 4 77.85  Kennalestes
## 5 77.85   Asioryctes

## But we can also decide to make the age relative (between 1 and 0)
head(tree.age(BeckLee_tree, age = 1), 5)

##    ages     elements
## 1 0.647    Daulestes
## 2 0.647  Bulaklestes
## 3 0.647 Uchkudukodon
## 4 0.560  Kennalestes
## 5 0.560   Asioryctes

## Or even relative, but from the root (i.e. how far are the nodes/tips
## from the root)
head(tree.age(BeckLee_tree, age = 1, order = "present"), 5)

##        ages     elements
## 1 0.3528637    Daulestes
## 2 0.3528637  Bulaklestes
## 3 0.3528637 Uchkudukodon
## 4 0.4402271  Kennalestes
## 5 0.4402271   Asioryctes
```

4 Developments

As stated at the start of the demo, this version 0.2 is still in development and many parts are missing. Here are the new functionalities that will be implemented in further versions.

4.1 More user defined functions

I intend also develop functions to help users to develop their own algorithms for the bootstrap method (via `make.boot`) or the evolutionary models (via `make.model`). Both functions will provide similar testing as the `make.metric` function.

4.2 Suggestions?

If you have any extra suggestions or comments on what has already been developed or will be developed, please send me an email (guillert@tcd.ie) or if you are a GitHub user, directly create an issue on the GitHub page (github.com/TGuillerme/dispRity).

References

Beck, R. M. and M. S. Lee. 2014. Ancient dates or accelerated rates? Morphological clocks and the antiquity of placental mammals. Proceedings of the Royal Society B: Biological Sciences 281:1–10.

Guillerme, T. and N. Cooper. in prep. Mammalian morphological diversity does not increase in response to the cretaceous-paleogene mass extinction and the extinction of the (non-avian) dinosaurs. .