dispRity manual

Thomas Guillerme

November 25, 2015

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, or the cladisto-space for cladisticts or the eco-space for ecology, etc. In practice though, this term designates an ordinated matrix where the columns represent the dimensions of the ordinated space (often > 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.
- **Series**: it designates sub-samples of the ordinated space. Basically a series contain the same number of dimensions as the morphospace but might 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 the same 50 dimensions) to look at the difference in disparity between both groups.

1.2 Installation

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

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

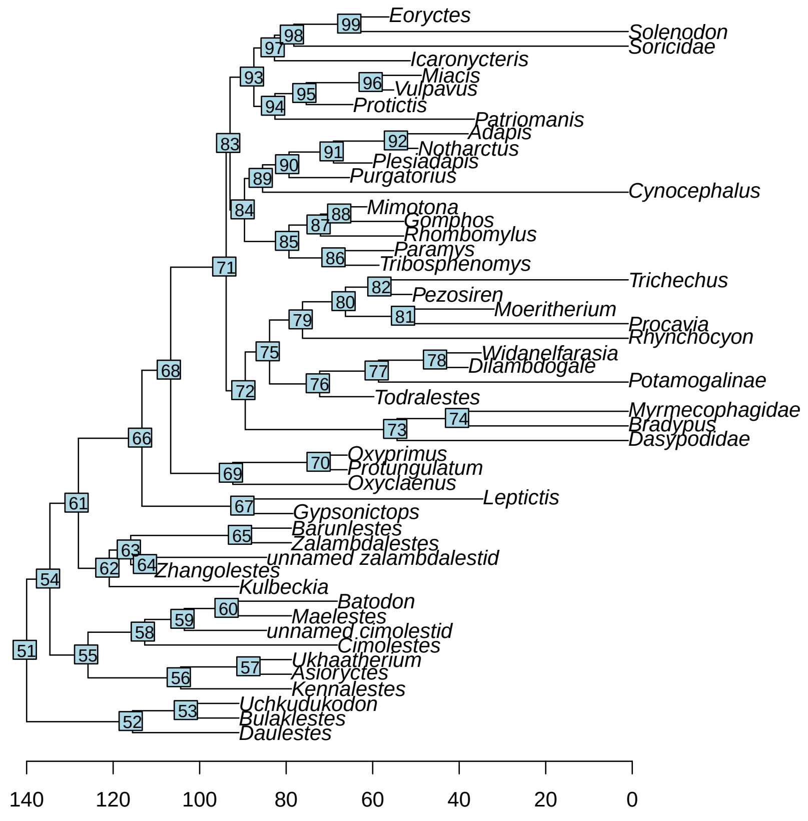
Note that we use the `release` branch here which is version 0.1.2. For the piping-hot (but potentially full of bugs) version, one can change the argument `ref = "release"` to `ref = "master"`. This package depends mainly on the `ape` package and the `timeSliceTree::paleotree` function.

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 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 of the package with ecological data.

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

## Loading required package:  paleotree

## 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

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

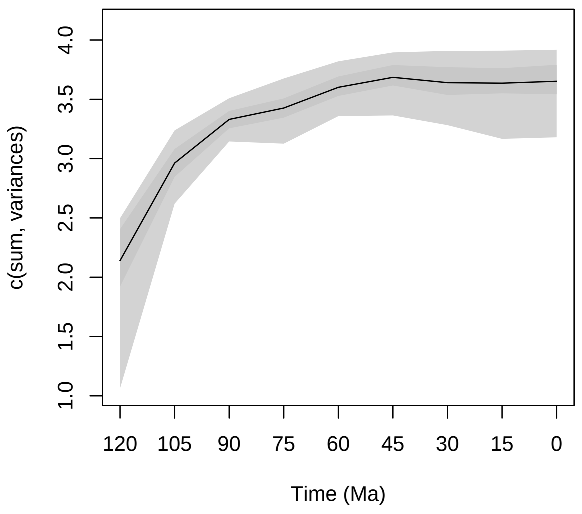
```
## [1] 99 97

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) ; nodelabels(cex=0.8) ; axisPhylo(root=140)
```



1.4 A quick go through

Here is a really crude and quick go through the package to show some of the features of this package. 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)

## Some tips have no FAD/LAD and are assumed to be single points in time.

## Bootstrapping the data
bootstrapped_data <- boot.matrix(sliced_data, 100)
## Calculating disparity
sum_of_variances <- dispRity(bootstrapped_data, metric = c(sum, variances))
## Summarising the results
summary(sum_of_variances)

##   series  n observed  mean  2.5%   25%   75% 97.5%
## 1    120  5    2.699 2.140 1.063 1.921 2.406 2.498
## 2    105 11    3.275 2.962 2.620 2.846 3.081 3.238
## 3     90 18    3.534 3.331 3.145 3.256 3.402 3.509
## 4     75 19    3.660 3.427 3.126 3.346 3.508 3.676
## 5     60 20    3.805 3.601 3.358 3.529 3.692 3.820
## 6     45 14    3.956 3.685 3.364 3.617 3.789 3.896
## 7     30 10    4.055 3.640 3.282 3.537 3.771 3.909
## 8     15 10    4.055 3.636 3.167 3.551 3.762 3.910
## 9      0 10    4.055 3.652 3.180 3.542 3.791 3.918

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

##              Df Sum Sq Mean Sq F value Pr(>F)
## series        8 205.28  25.660   688.9 <2e-16 ***
## Residuals   891  33.19   0.037
## ---
## 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 a bit impractical to visualise and quickly jam your `R` console. For example, we can have a look at the structure of the object created in the quick example:

```
str(sum_of_variances)
## That's a more than 4500 lines of output!
```

Therefore this package proposes a new class of object called `dispRity` objects. These objects allow to easily use a 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 `print.phylo::ape`).

```
## Which class is the sum_of_variances object?
class(sum_of_variances)

## [1] "dispRity"

## What's in the object
names(sum_of_variances)

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

## We can summarise it using the S3 method print.dispRity
sum_of_variances

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

## This displays some information about what's in the object
```

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

```
## Displaying the full object
print(sum_of_variances, all=TRUE)
```

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(sum_of_variances, what=c(1,4))
series_1_and_4

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

## The observed disparity
extract.dispRity(sum_of_variances)

##      120      105       90       75       60       45       30       15        0
## 2.698895 3.274613 3.533596 3.660471 3.804800 3.955998 4.054575 4.054575 4.054575

## The list of bootstrapped scores of disparity
str(extract.dispRity(sum_of_variances, observed = FALSE))

## List of 9
##  $ 120: num [1:100] 1.98 2.4 2.46 2.4 2.15 ...
##  $ 105: num [1:100] 3.17 3 2.8 2.64 2.97 ...
##  $ 90 : num [1:100] 3.47 3.5 3.42 3.43 2.98 ...
##  $ 75 : num [1:100] 3.26 3.44 3.35 3.25 3.68 ...
##  $ 60 : num [1:100] 3.08 3.36 3.74 3.63 3.59 ...
##  $ 45 : num [1:100] 3.81 3.54 3.49 3.77 3.89 ...
##  $ 30 : num [1:100] 3.43 3.67 3.62 3.54 3.54 ...
##  $ 15 : num [1:100] 3.4 3.56 3.69 3.6 3.66 ...
##  $ 0  : num [1:100] 3.24 3.71 3.65 3.52 3.77 ...
```

2.2 Modular functions

This package aims to be a modular package where users can personalise some aspects of the package fairly easily. In this version 0.1.2 only one function is fully modular (`dispRity` ; see section 3.4) but more will be hopefully available in the near future (see section 4).

The modular idea is that users can use implemented tools to help facilitating their own personalised function. 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
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.by.length <- function(X)  sum(X)/length(X)
## Disparity measured as the mean divided by the length
summary(dispRity(sliced_data, metric = sum.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 concerning this modularity, please refer to the GitHub page: github.com/TGuillerme/dispRity for the vignette about the metric implementation in `dispRity` .

3 Functions

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 at 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

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

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

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

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 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-slicing places (when using `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/Last Occurrence Data (`FADLAD`) 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_bins <- time.series(data = BeckLee_mat50, tree = BeckLee_tree,
    method = "discrete", time = c(120, 80, 40, 0))

## No FAD/LAD table has been provided.
## Every tips are assumed to be single points in time.

## Note that the function provides a warning saying that tips where single
## points in time (no FAD/LAD information). We can fix that by adding the
## age data for the taxa that have some longer occurrence spans.
time_bins <- time.series(data = BeckLee_mat50, tree = BeckLee_tree,
    method = "discrete", time = c(120, 80, 40, 0), FADLAD = BeckLee_ages)

## Some tips have no FAD/LAD and are assumed to be single points in time.

## To entirely avoid the warning we could collect the occurrence span data
## for all the taxa but that's not necessary. The function automatically
## assumes no occurrence span time (i.e. single points in time) for all taxa
## by default.
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)

## No FAD/LAD table has been provided.
## Every tips are assumed to be single points in time.

## 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 Ma 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 (an extra 1% is added until reaching the required minimum of three elements). We can also ask 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 Ma 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)

## Some tips have no FAD/LAD and are assumed to be single points in time.

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)

## No FAD/LAD table has been provided.
## Every tips are assumed to be single points in time.

## 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 or/and rarefy it (i.e. pseudo-replicating the data). The bootstrap will allow us to make each subsample 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) .

```
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.
```

Additionally, 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 also 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 subsampled 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. let's bootstrap our previously generate series:

```
## Bootstrap and full rarefaction on the crown/stem series
crown_stemBS <- boot.matrix(crown_stem, rarefaction = TRUE)
## Bootstrap on the time binning/slicing series
time_binsBS <- boot.matrix(time_bins)
time_slicesBS <- boot.matrix(time_slices)
## 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 1000 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 1000 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 1000 times, using the full bootstrap method.
```

3.4 `dispRity`

Now we're going to see the functionalities of the core function of this package: the `dispRity` function. This function is a modulable function that allow 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* (and here's where the modulable part comes in). In fact, the `dispRity` function intakes two main arguments: the data and the disparity metric. The disparity metric boils down to a single value that describes the matrix. Some are classic like the sum or the product of the ranges or the variances of the matrix columns (i.e. the any-o-space axis/dimensions) Wills et al. (1994). However, many more exist like the median distances between the taxa and the any-o-space centroid Guillerme and Cooper (in prep.).

One can usually decompose the disparity metrics into two elements:

1. the **class metric** that is a descriptor of the matrix. For example describing the ranges of each column in the matrix or the euclidean distances between each row and the centroid of the matrix.

2. the **summary metric** that is a summary of the class metric values. For example, the sum of the ranges or the median of the euclidean distances.

Basically the combination can be infinite between the class and summary metrics. For example, people might want to measure the median variances of the axis or the product of the distances from the centroid. However, it is probable that some metrics are better to reflect some biological aspects of the any-o-space than others (see section **??**)...

In practice, the `dispRity` function intakes a pair of class and summary metrics as a definition of disparity. Several of these metrics are implemented in other packages (like `stats::median`, `base::sum`, etc.) and this package proposes several metrics listed in `dispRity.metric` (see `?dispRity.metric`). It is even possible to use your very own class and summary metrics! This will be actually heavily encouraged and facilitate with the `make.metric` function in a future version (see **??**).

To use these metrics pairs in the `dispRity` function, it's pretty easy:

```r
dispRity(BeckLee_mat50, metric = c(sum, ranges))

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

## Yep, that's it. The order of the metrics does not matter (the function detects
## automatically which one is the class and the summary metric) so the combinations
## are numerous!
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.

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.

dispRity(BeckLee_mat50, metric = c(mode.val, ranges))

## Disparity measurements across 1 series for 50 elements
## 1
## Disparity calculated as: c(mode.val, ranges) for 48 dimensions.

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

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

## That would be disparity as the range of the axis ranges!
```

Of course, this function can take a simple ordinated matrix but, more interesting it can also deal with a list of matrices (from the `cust.series` or `time.series` functions for example) or a bootstrapped or/and rarefied matrix (from the `boot.matrix` function for example).

```
## For example, let's calculate the sum of variances from our crown/stem mammals
dispRity(crown_stem, metric = c(sum, variances))

## Disparity measurements across 2 series for 50 elements
## Series:
## Group.crown, Group.stem.
## Disparity calculated as: c(sum, variances) for 48 dimensions.
## Data was split using custom method.

## Or even better, from the bootstrapped data
disp_crown_stemBS <- dispRity(crown_stemBS, metric = c(sum, variances))
## Note that the bootstrapped + rarefied data takes some time to run (it's
## calculating disparity from 46 matrices!)

## Let's calculate a different metric for the disparity-through-time analysis
disp_time_binsBS <- dispRity(time_binsBS, metric = c(median, centroids))
disp_time_slicesBS <- dispRity(time_slicesBS, metric = c(median, centroids))
```

Again, note that the results are `dispRity` objects and can displayed through the S3 `print.dispRity` method to remind users what are in these objects. Also note that an even faster parallel version will be developed in the future (see section **??**).

3.5 `summary`

As shown in these examples above however, even though the disparity has been calculated in many way, the function doesn't directly outputs the results, rather just a summary of the analysis. To really display the results we can use the S3 `summary.dispRity` and `plot.dispRity` functions.

`summary` function

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.

```
## Let's have a look at the disparity-through-time using in the time-binned data
summary(disp_time_binsBS)

##   series  n observed  mean  2.5%   25%   75% 97.5%
## 1 120-80  8    1.203 1.113 0.914 1.068 1.171 1.244
## 2  80-40 27    1.344 1.322 1.273 1.308 1.339 1.359
## 3   40-0 16    1.353 1.322 1.254 1.304 1.344 1.372

## The 50 and 95 quantiles are calculated and the central tendency is the mean by default.
## Note that the function also displays the observed disparity (non-bootstrapped).
## We can change that easily by specifying different values in the options
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
## Also, if we happen to have forgotten what was in the disp_time_binsBS object (as in
## which options where used to modify the original matrix) we can use the recall option:
summary(disp_time_binsBS, recall = TRUE)

## Disparity calculated as: c(median, centroids) for 48 dimensions.
## Data was split using discrete method.
## Data was bootstrapped 1000 times, using the full bootstrap method.
##   series  n observed  mean  2.5%   25%   75% 97.5%
## 1 120-80  8    1.203 1.113 0.914 1.068 1.171 1.244
## 2  80-40 27    1.344 1.322 1.273 1.308 1.339 1.359
## 3   40-0 16    1.353 1.322 1.254 1.304 1.344 1.372

## Note that the recall just prints text. The output of summary.dispRity is always a
## data.frame
summary_table <- summary(disp_time_binsBS, recall = TRUE)

## Disparity calculated as: c(median, centroids) for 48 dimensions.
## Data was split using discrete method.
## Data was bootstrapped 1000 times, using the full bootstrap method.

class(summary_table)

## [1] "data.frame"

## Finally we can see the results of the rarefaction analysis for the crown/stem data
head(summary(disp_crown_stemBS))

##        series n observed  mean  2.5%   25%   75% 97.5%
## 1 Group.crown 3       NA 1.940 1.272 1.901 2.071 2.199
## 2 Group.crown 4       NA 1.932 1.529 1.875 2.046 2.136
## 3 Group.crown 5       NA 1.931 1.607 1.852 2.025 2.108
## 4 Group.crown 6       NA 1.923 1.653 1.858 2.009 2.080
## 5 Group.crown 7       NA 1.931 1.731 1.882 1.997 2.071
## 6 Group.crown 8       NA 1.922 1.729 1.876 1.984 2.051

## This outputs a longer table with all the variations of the number of crown/stem taxa
## from the maximum (30 and 20) to the minimum (3). Note that the observed disparity
## is NA for rarefied data.
```

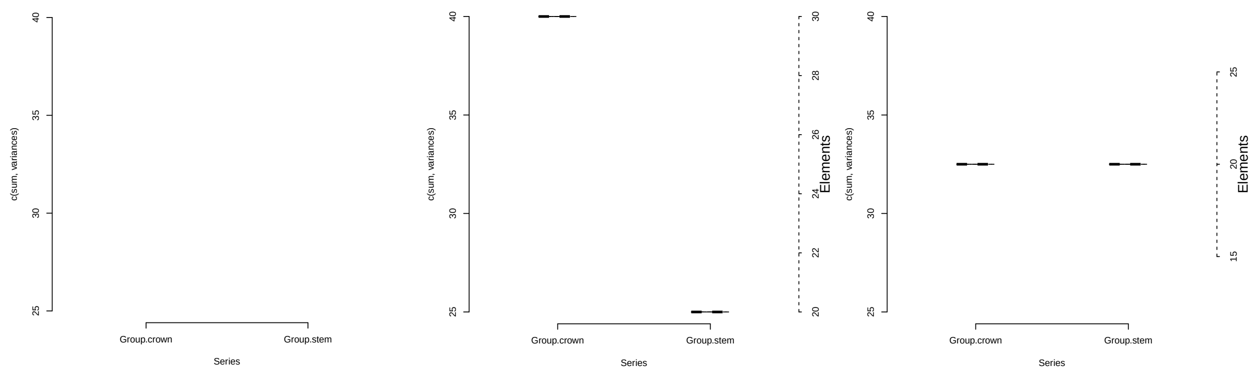
3.6 `plot`

This final function allows to visualise the disparity results in a often nicer fashion than just a table (even though the exact same data is displayed). The `plot.dispRity` option intakes the same options as `summary.dispRity` along side with various graphical options described in the function manual. Here, let's just have a look a few of these options.

Let's start with plotting the difference in disparity between the crown and the stem mammals. For this we are going to simply specify the type of plots using the `type` option that allows to choose between the `"continuous"` or `"discrete"` method. In our case, we are interested in looking at the results in a discrete way. Note that the plot `type` can be left missing. The function will then use `"discrete"` as default or `"continuous"` if the input data came from the `time.series` function with `method = "continuous"`.

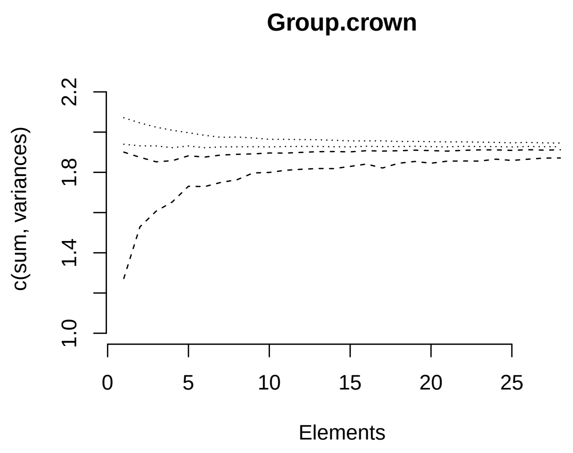
The grey squares represent the confidence intervals (50 in dark grey and 90 in light grey) and the dot represents the mean. However, it might be interesting to see how many taxa are present in each group. This can be done via the `elements` option. A third interesting option is to plot the rarefied data if a rarefaction was applied on the data with the `boot.matrix` function. For displaying the rarefied data we can use the `rarefaction` option. This option can be either `FALSE` (default) for always plotting the maximum number of taxa per series or `TRUE` to always plot the minimum. Additionally, it can be a specific number to plot a specific number of taxa. Let say we want both groups to have 20 taxa maximum. We can see now that the confidence intervals overlap more than previously

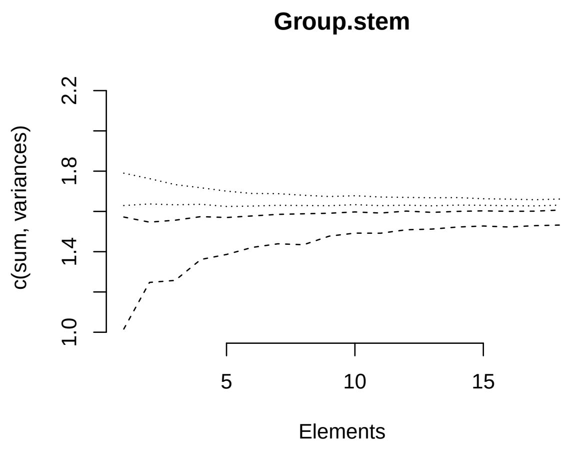
```r
## Graphical options
quartz(width = 15, height = 5) ; par(mfrow = (c(1,3)), bty = "n")
## Plotting the score for both groups
plot(disp_crown_stemBS, ylim = c(25,40))
## Same one with the number of taxa for each data set
plot(disp_crown_stemBS, elements = TRUE, ylim = c(25,40))
## Same one with rarefied data (note how that affects the elements data as well!)
plot(disp_crown_stemBS, elements = TRUE, rarefaction = 20, ylim = c(25,40))
```



To understand how the number of taxa affects each series, we can use the `rarefaction = "plot"` function to plot the rarefaction curves.

```r
## Graphical options
par(bty = "n")
## The rarefaction curves
plot(disp_crown_stemBS, elements = TRUE, rarefaction = "plot")
## Note that the function automatically draws the curves for each series and splits
## the plot screen accordingly.
```

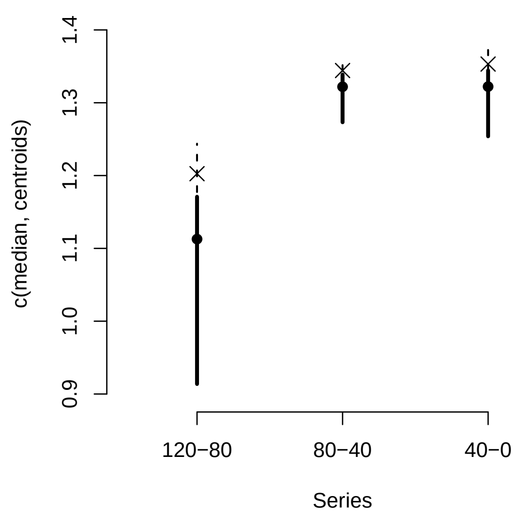


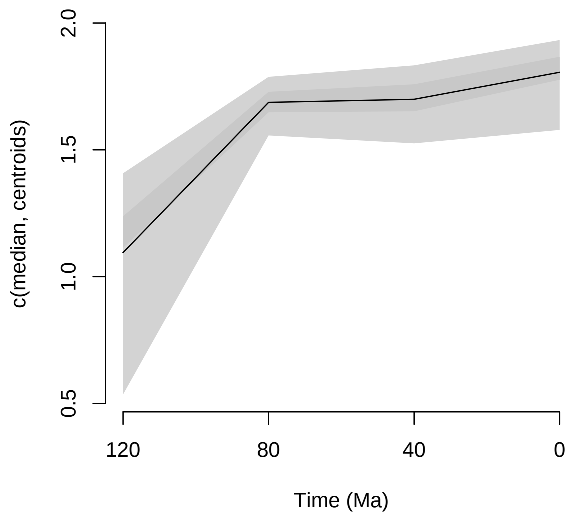


The different dashed lines represent the different confidence intervals around each rarefaction curve.

Additionally, for the `type = "discrete"` option, we might also decide to plot many boxplots (which can be a bit messy). The `discrete_type` option allows to switch between `"box"` or `"line"` for showing the results in one or the other format. We can also plot the observed data when existing (i.e. not with some rarefaction levels) by using `observed = TRUE`. Let's look at that with the time binned data (even though there is only three time bins). Finally, we can use the `type = "continuous"` option that comes in handy for plotting continuous data like in the time sliced analysis (obviously...).

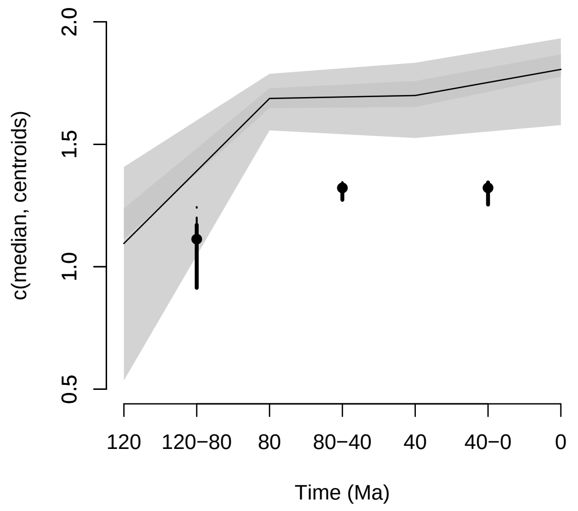
```r
## Graphical options
quartz(width = 10, height = 5) ; par(mfrow = (c(1,2)), bty = "n")
## The disparity-through-time data in a time-binned way (with lines rather than
## boxes) with the observed data as crosses
plot(disp_time_binsBS, discrete_type = "line", observed = TRUE)
## The disparity-through-time data in a time-sliced ways
plot(disp_time_slicesBS)
```





We can even plot both results (discrete and continuous) on the same graph:

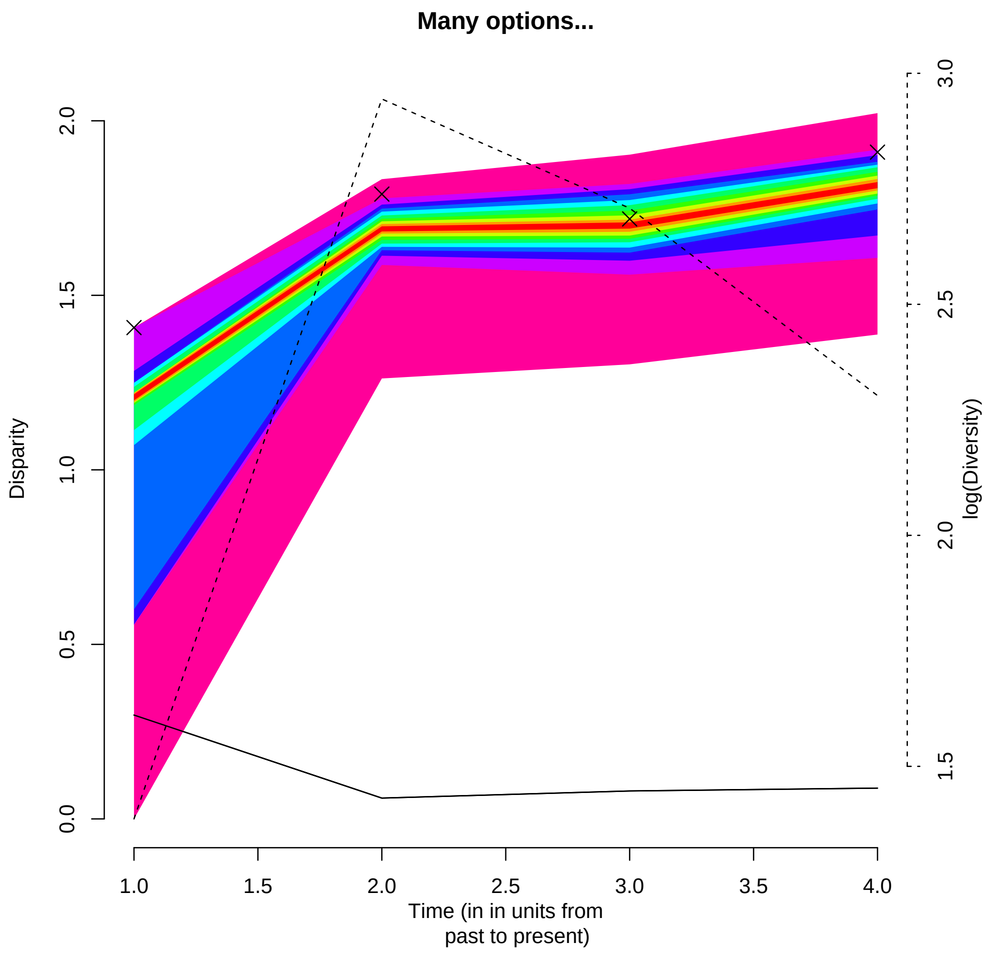
```
## Graphical options
par(bty = "n")
## First, plotting the continuous data
plot(disp_time_slicesBS, ylim=c(0.5,2))
## Second, adding the discrete data
par(new = TRUE)
plot(disp_time_binsBS, discrete_type = "line", ylim=c(0.5,2), xlab="", ylab="")
```



Note that the results vary in values but not in pattern. The change in values might be due to the fact that nodes data are not included in the discrete analysis.

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...")
```



3.7 test.dispRity

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 0.1.2 is still in development and many parts are missing. Here are the new functionalities that will be implemented before the proper release version (v.1).

4.1 More user defined stuff

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 Faster!

Finally, for long analysis, I intend to develop a parallel running version of the package. In fact, most of the internal functions are base on `lapply` functions that can be easily passed to `snow::parLapply` or similar parallel functions.

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. .

Wills, M. A., D. E. G. Briggs, and R. A. Fortey. 1994. Disparity as an evolutionary index: A comparison of Cambrian and recent arthropods. Paleobiology 20:93–130.