# dispRity manual

Thomas Guillerme (guillert@tcd.ie), Mark Puttick (marknputtick@gmail.com) and Natalie Coope

2021-04-15

# Contents

# Chapter 1

# dispRity

This is a package for measuring disparity (aka multidimensional space occupancy) in R. It allows users to summarise matrices as representations as multidimensional spaces into a single value or distribution describing a specific aspect of this multidimensional space (the disparity). Multidimensional spaces can be ordinated matrices from MDS, PCA, PCO, PCoA but the package is *not* restricted to any type of matrices! This manual is based on the version 1.6.

## 1.1 What is dispRity?

This is a modular package for measuring disparity in R. It allows users to summarise ordinated matrices (e.g. MDS, PCA, PCO, PCoA) to perform some multidimensional analysis. Typically, these analysis are used in palaeobiology and evolutionary biology to study the changes in morphology through time. However, there are many more applications in ecology, evolution and beyond.

### 1.1.1 Modular?

Because their exist a multitude of ways to measure disparity, each adapted to every specific question, this package uses an easy to modify modular architecture. In coding, each module is simply a function or a modification of a function that can be passed to the main functions of the package to tweak it to your proper needs! In practice, you will notice throughout this manual that some function can take other functions as arguments: the modular architecture of this package allows you to use any function for these arguments (with some restrictions explained for each specific cases). This will allow you to finely tune your multidimensional analysis to the needs of your specific question!

## 1.2 Installing and running the package

You can install this package easily, directly from the CRAN:

```
install.packages("dispRity")
```

Alternatively, for the most up to data version and some functionalities not compatible with the CRAN, you can use the package through GitHub using `devtool` (see to CRAN or not to CRAN? for more details):

```
## Checking if devtools is already installed
if(!require(devtools)) install.packages("devtools")

## Installing the latest released version directly from GitHub
install_github("TGuillerme/dispRity", ref = "release")
```

Note this uses the `release` branch (1.6). For the piping-hot (but potentially unstable) version, you can change the argument `ref = release` to `ref = master`. `dispRity` depends mainly on the `ape` package and uses functions from several other packages (`ade4`, `geometry`, `grDevices`, `hypervolume`, `paleotree`, `snow`, `Claddis`, `geomorph` and `RCurl`).

## 1.3 Which version do I choose?

There are always three version of the package available:

- The CRAN one
- The GitHub `release` one
- The GitHub `master` one

The differences between the CRAN one and the GitHub `release` or `master` ones is explained just above. For the the GitHub version, the differences are that the `release` one is more stable (i.e. more rarely modified) and the `master` one is more live one (i.e. bug fixes and new functionalities are added as they come).

If you want the latest-latest version of the package I suggest using the GitHub `master` one, especially if you recently emailed me reporting a minor bug or wanting a new functionality! Note however that *it can happen* that the `master` version can sometimes be bugged (especially when there are major `R` and `R` packages updates), however, the status of the package state on both the `release` and the `master` version is constantly displayed on the `README` page of the package with the nice badges displaying these different (and constantly tested) information.

## 1.4 dispRity is always changing, how do I know it's not broken?

This is a really common a legitimate question in software development. Like R itself:

> dispRity is free software and comes with ABSOLUTELY NO WARRANTY.

So you are using it at your own risk.

**HOWEVER**, there are two points that can be used as objective-ish markers on why it's OK to use dispRity.

First, the package has been use in a number of peer reviewed publications (the majority of them independently) which could be taken as warranty.

Second, I spend a lot of time and attention in making sure that every function in every version actually does what I think it is supposed to do. This is done through CI; continuous integration development, the CRAN check, and unit testing. The two first checks (CRAN and CI) ensure that the version you are using is not bugged (the CRAN check if you are using the CRAN version and the Travis CI if you are using a GitHub version). The third check, unit testing, is checking that every function is doing what it is supposed to do. For a real basic example, it is testing that the following expression should always return the same thing no matter what changes in the package.

```
> mean(c(1,2,3))
[1] 2
```

Or, more formally:

```
testthat::expect_equal(object = mean(c(1,2,3)),
                       expected = 2)
```

You can always access what is actually tested in the test/testthat sub-folder. For example here is how the core function dispRity is tested (through $> 500$ tests!). All these tests are run every time a change is made to the package and you can always see for yourself how much a single function is covered (i.e. what percentage of the function is actually covered by at least one test). You can always see the global coverage here or the specific coverage for each function here.

Finally, this package is build on the shoulders of the whole open science philosophy so when bugs do occur and are caught by myself or the package users, they are quickly fixed and notified in the NEWS.md file. And all the changes to the package are public and annotated so there's that too...

## 1.5 Help

If you need help with the package, hopefully the following manual will be useful. However, parts of this package are still in development and some other parts are probably not covered. Thus if you have suggestions or comments on 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.

## 1.6 Citations

To cite the package, this manual or some specific functionalities, you can use the following references:

The package main paper:

> Guillerme T. dispRity: A modular R package for measuring disparity. Methods Ecol Evol. 2018;9:1755–1763. doi.org/10.1111/2041-210X.13022.

The package manual (regularly updated!):

> Guillerme, T. & Cooper, N. (2018): dispRity manual. figshare. Preprint. 10.6084/m9.figshare.6187337.v1.

The time-slicing method implemented in `chrono.subsets` (unfortunately not Open Access, but you can still get a free copy from here):

> Guillerme, T. and Cooper, N. (2018), Time for a rethink: time subsampling methods in disparity-through-time analyses. Palaeontology, 61: 481-493. doi:10.1111/pala.12364.

Furthermore, don't forget to cite `R`:

> R Core Team (2020). R: A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria. URL https://www.R-project.org/.

Bonus: you can also cite `ape` since the `dispRity` package heavily relies on it:

> Paradis E. & Schliep K. 2019. ape 5.0: an environment for modern phylogenetics and evolutionary analyses in R. Bioinformatics 35: 526-528.

### 1.6.1 Why is it important to cite us?

Aside from how science works (if you're using a method from a specific paper, cite that specific paper to refer to that specific method), why is it important to also cite the package and the manual?

All the people involve in making the `dispRity` package happened to do it enthusiastically, freely and most amazingly without asking anything in return! I created the package with this idea in mind and I am still sticking to it. However, academia (the institutions and people producing science around the globe) is unfortunately not optimal at many level (some might even say "broken"): high impact papers attract big grants that attract high impact papers and big grants again, all this along with livelihood, permanent position and job security. Unfortunately however, method development has a hard time to catch up with the current publish or perish system: constantly updating the `dispRity` package and this manual is hugely time consuming (but really fun!) and that is not even taking into account maintenance and helping users. Although I do truly believe that this time spent doing these things modestly help the scientific endeavour, it does not contribute to our paper list!

Therefore, by citing the package and this manual, you help provide visibility to other workers and you might help them in their work! And you directly contribute in making this project fun for all the people involved and most of all, free, updated and independent from the publish and perish system!

Thank you!

# Chapter 2

# Glossary

- **Multidimensional space** (or just space). The mathematical multidimensional object that will be analysed with this package. In morphometrics, this is often referred to as the morphospace. However it may also be referred to as the cladisto-space for cladistic data or the eco-space for ecological data etc. In practice, this term designates a matrix where the columns represent the dimensions of the space (often – but not necessarily - > 3!) and the rows represent the elements within this space.

- **Elements**. The rows of the multidimensional space matrix. Elements can be taxa, field sites, countries etc.

- **Dimensions**. The columns of the multidimensional space matrix. The dimensions can be referred to as axes of variation, or principal components, for ordinated spaces obtained from a PCA for example.

- **Subsets**. Subsets of the multidimensional space. A subset (or subsets) contains the same number of dimensions as the space but may contain a smaller subset of elements. For example, if our space is composed of birds and mammals (the elements) and 50 principal components of variation (the dimensions), we can create two subsets containing just mammals or birds, but with the same 50 dimensions, to compare disparity in the two clades.

- **Disparity**. A metric expressing the similarities/dissimilarities of the elements within the space or a summarising the space dimensions. For example the pairwise distances between elements or the range of each dimensions.

## 2.1 Glossary equivalences in palaeobiology and ecology

| In this manual | In `dispRity` | E.g. in palaeobiology | E.g. in ecology |
|---|---|---|---|
| the multidimensional space | a `matrix` object ($n \times d$) | a morphospace | a function-space |
| elements | rows ($n$) | taxa | field experiments |
| dimensions | columns ($d$) | morphological characters | communities' compositions |
| subsets | a `matrix` ($m \times d$, with $m \leq n$) | time series | experimental treatments |
| disparity | a `function` | sum of variances | ellipsoid volume |

# Chapter 3

# Getting started with `dispRity`

## 3.1 What sort of data does `dispRity` work with?

**Any matrix object in `R`.** Disparity can be estimated from pretty much any matrix as long as rows represent the elements and columns the dimensions. These matrices can be observations, pairwise differences between elements, ordinations, etc...

Since version `1.4` it is also possible to include a `"list"` containing matrices. These matrices need to have the same dimensions and rownames but can contain different values. This is especially useful for modelling uncertainty (see here for more details).

## 3.2 Ordinated matrices

Classically, when a high number of variables is used, disparity is calculated from ordinated matrices. These can be any type of ordinations (PCO, PCA, PCoA, MDS, etc.) as long as elements are the rows (taxa, countries, field experiments) and the dimensions are the columns. However, note that this is not required from any of the functions in this package. You can also use distance matrices or any other matrix type that suits your question and your analysis!

### 3.2.1 Ordination matrices from `geomorph`

You can also easily use data from `geomorph` using the `geomorph.ordination` function. This function simply takes Procrustes aligned data and performs an

ordination:

```
require(geomorph)

## Loading the plethodon dataset
data(plethodon)

## Performing a Procrustes transform on the landmarks
procrustes <- gpagen(plethodon$land, PrinAxes = FALSE,
                     print.progress = FALSE)

## Ordinating this data
geomorph.ordination(procrustes)[1:5,1:5]
```

```
##               PC1        PC2           PC3          PC4          PC5
## [1,] -0.0369930887 0.05118246 -0.0016971586 -0.003128881 -0.010935739
## [2,] -0.0007493689 0.05942083  0.0001371682 -0.002768621 -0.008117767
## [3,]  0.0056004751 0.07419599 -0.0052612189 -0.005034502 -0.002747104
## [4,] -0.0134808326 0.06463958 -0.0458436274 -0.007887336  0.009817034
## [5,] -0.0334696064 0.06863518  0.0136292227  0.007359383  0.022347215
```

Options for the ordination (from `?prcomp`) can be directly passed to this function to perform customised ordinations. Additionally you can give the function a `geomorph.data.frame` object. If the latter contains sorting information (i.e. factors), they can be directly used to make a customised `dispRity` object customised `dispRity` object!

```
## Using a geomorph.data.frame
geomorph_df <- geomorph.data.frame(procrustes,
     species = plethodon$species, site = plethodon$site)

## Ordinating this data and making a dispRity object
geomorph.ordination(geomorph_df)
```

```
##  ---- dispRity object ----
## 4 customised subsets for 40 elements in one matrix:
##     species.Jord, species.Teyah, site.Allo, site.Symp.
```

More about these `dispRity` objects below!

### 3.2.2 Ordination matrices from `Claddis`

`dispRity` package can also easily take data from the `Claddis` package using the `Claddis.ordination` function. For this, simply input a matrix in the `Claddis` format to the function and it will automatically calculate and ordinate the distances among taxa:

```
require(Claddis)

## Ordinating the example data from Claddis
Claddis.ordination(michaux_1989)
```

```
##                       [,1]          [,2]       [,3]
## Ancilla       0.000000e+00  4.154578e-01  0.2534942
## Turrancilla -5.106645e-01 -1.304614e-16 -0.2534942
## Ancillista   5.106645e-01 -1.630768e-17 -0.2534942
## Amalda       1.603581e-16 -4.154578e-01  0.2534942
```

Note that several options are available, namely which type of distance should be computed. See more info in the function manual (`?Claddis.ordination`). Alternatively, it is of course also possible to manual calculate the ordination matrix using the functions `Claddis::calculate_morphological_distances` and `stats::cmdscale`.

### 3.2.3 Other kinds of ordination matrices

If you are not using the packages mentioned above (`Claddis` and `geomorph`) you can easily make your own ordination matrices by using the following functions from the `stats` package. Here is how to do it for the following types of matrices:

- Multivariate matrices (principal components analysis; PCA)

```
## A multivariate matrix
head(USArrests)
```

```
##            Murder Assault UrbanPop Rape
## Alabama      13.2     236       58 21.2
## Alaska       10.0     263       48 44.5
## Arizona       8.1     294       80 31.0
## Arkansas      8.8     190       50 19.5
## California    9.0     276       91 40.6
## Colorado      7.9     204       78 38.7
```

```
## Ordinating the matrix using `prcomp`
ordination <- prcomp(USArrests)

## Selecting the ordinated matrix
ordinated_matrix <- ordination$x
head(ordinated_matrix)
```

```
##                 PC1        PC2        PC3        PC4
## Alabama     64.80216 -11.448007 -2.4949328 -2.4079009
## Alaska      92.82745 -17.982943 20.1265749  4.0940470
## Arizona    124.06822   8.830403 -1.6874484  4.3536852
## Arkansas    18.34004 -16.703911  0.2101894  0.5209936
```

```
## California 107.42295  22.520070  6.7458730  2.8118259
## Colorado    34.97599  13.719584 12.2793628  1.7214637
```

This results in a ordinated matrix with US states as elements and four dimensions (PC 1 to 4). For an alternative method, see the `?princomp` function.

- Distance matrices (classical multidimensional scaling; MDS)

```
## A matrix of distances between cities
str(eurodist)
```

```
##  'dist' num [1:210] 3313 2963 3175 3339 2762 ...
##  - attr(*, "Size")= num 21
##  - attr(*, "Labels")= chr [1:21] "Athens" "Barcelona" "Brussels" "Calais" ...
```

```
## Ordinating the matrix using cmdscale() with k = 5 dimensions
ordinated_matrix <- cmdscale(eurodist, k = 5)
head(ordinated_matrix)
```

```
##                 [,1]      [,2]       [,3]       [,4]       [,5]
## Athens    2290.27468 1798.8029   53.79314 -103.82696 -156.95511
## Barcelona -825.38279  546.8115 -113.85842   84.58583  291.44076
## Brussels    59.18334 -367.0814  177.55291   38.79751  -95.62045
## Calais     -82.84597 -429.9147  300.19274  106.35369 -180.44614
## Cherbourg -352.49943 -290.9084  457.35294  111.44915 -417.49668
## Cologne    293.68963 -405.3119  360.09323 -636.20238  159.39266
```

This results in a ordinated matrix with European cities as elements and five dimensions.

Of course any other method for creating the ordination matrix is totally valid, you can also not use any ordination at all! The only requirements for the `dispRity` functions is that the input is a matrix with elements as rows and dimensions as columns.

## 3.3 Performing a simple dispRity analysis

Two `dispRity` functions allow users to run an analysis pipeline simply by inputting an ordination matrix. These functions allow users to either calculate the disparity through time (`dispRity.through.time`) or the disparity of user-defined groups (`dispRity.per.group`).

**IMPORTANT**

Note that `disparity.through.time` and `disparity.per.group` are wrapper functions (i.e. they incorporate lots of other functions) that allow users to run a basic disparity-through-time, or disparity among groups, analysis without too much effort. As such they use a lot of default options. These are described in the help files for the functions that are used to make the wrapper functions, and not described in the help files for `disparity.through.time`

and `disparity.per.group`. These defaults are good enough for **data exploration**, but for a proper analysis you should consider the **best parameters for your question and data**. For example, which metric should you use? How many bootstraps do you require? What model of evolution is most appropriate if you are time slicing? Should you rarefy the data? See `chrono.subsets`, `custom.subsets`, `boot.matrix` and `dispRity.metric` for more details of the defaults used in each of these functions. Note that any of these default arguments can be changed within the `disparity.through.time` or `disparity.per.group` functions.

### 3.3.1 Example data

To illustrate these functions, we will use data from Beck and Lee [2014]. This dataset contains an ordinated matrix of 50 discrete characters from mammals (`BeckLee_mat50`), another matrix of the same 50 mammals and the estimated discrete data characters of their descendants (thus 50 + 49 rows, `BeckLee_mat99`), a dataframe containing the ages of each taxon in the dataset (`BeckLee_ages`) and finally a phylogenetic tree with the relationships among the 50 mammals (`BeckLee_tree`).

```
## Loading the ordinated matrices
data(BeckLee_mat50)
data(BeckLee_mat99)

## The first five taxa and dimensions of the 50 taxa matrix
head(BeckLee_mat50[, 1:5])
```

```
##                 [,1]       [,2]       [,3]       [,4]       [,5]
## Cimolestes  -0.5613001  0.06006259  0.08414761 -0.2313084 -0.18825039
## Maelestes   -0.4186019 -0.12186005  0.25556379  0.2737995 -0.28510479
## Batodon     -0.8337640  0.28718501 -0.10594610 -0.2381511 -0.07132646
## Bulaklestes -0.7708261 -0.07629583  0.04549285 -0.4951160 -0.39962626
## Daulestes   -0.8320466 -0.09559563  0.04336661 -0.5792351 -0.37385914
## Uchkudukodon -0.5074468 -0.34273248  0.40410310 -0.1223782 -0.34857351
```

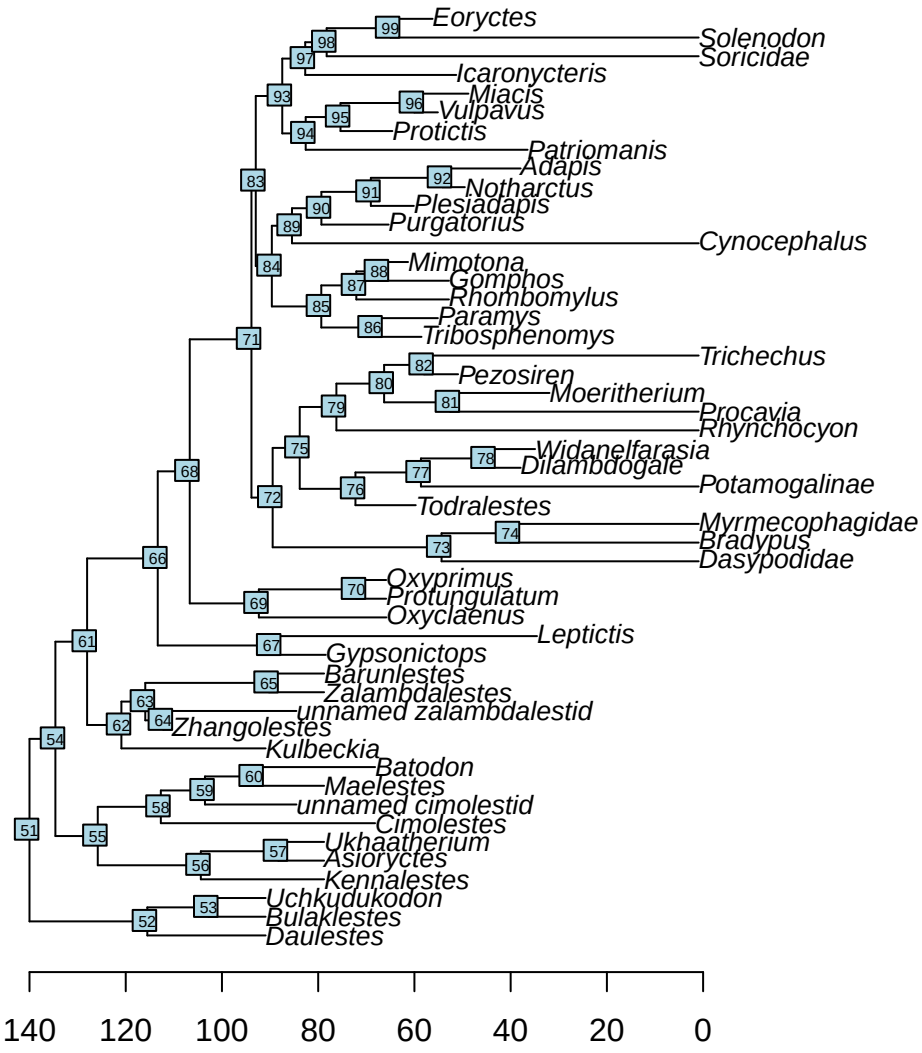
```
## The first five taxa and dimensions of the 99 taxa + ancestors matrix
BeckLee_mat99[c(1, 2, 98, 99), 1:5]
```

```
##                [,1]       [,2]       [,3]       [,4]       [,5]
## Cimolestes -0.6794737 0.15658591  0.04918307  0.22509831 -0.38139436
## Maelestes  -0.5797289 0.04223105 -0.20329542 -0.15453876 -0.06993258
## n48         0.2614394 0.01712426  0.21997583 -0.05383777  0.07919679
## n49         0.3881123 0.13771446  0.11966941  0.01856597 -0.15263921
```

```
## Loading a list of first and last occurrence dates for the fossils
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 and plotting the phylogeny
data(BeckLee_tree)
plot(BeckLee_tree, cex = 0.8)
axisPhylo(root = 140)
nodelabels(cex = 0.5)
```



Of course you can use your own data as detailed in the previous section.

### 3.3.2 Disparity through time

The `dispRity.through.time` function calculates disparity through time, a common analysis in palaeontology. This function (and the following one) uses an analysis pipeline with a lot of default parameters to make the analysis as simple as possible. Of course all the defaults can be changed if required, more on this later.

For a disparity through time analysis, you will need:

- An ordinated matrix (we covered that above)
- A phylogenetic tree: this must be a `phylo` object (from the `ape` package) and needs a `root.time` element. To give your tree a root time (i.e. an age for the root), you can simply do\ `my_tree$root.time <- my_age`.
- The required number of time subsets (here `time = 3`)
- Your favourite disparity metric (here the sum of variances)

Using the Beck and Lee (2014) data described above:

```
## Measuring disparity through time
disparity_data <- dispRity.through.time(BeckLee_mat50, BeckLee_tree,
                                        metric = c(sum, variances),
                                        time = 3)
```

This generates a `dispRity` object (see here for technical details). When displayed, these `dispRity` objects provide us with information on the operations done to the matrix:

```
## Print the disparity_data object
disparity_data
```

```
##  ---- dispRity object ----
## 3 discrete time subsets for 50 elements in one matrix with 48 dimensions with 1 phylogenetic tree
##     133.51 - 89.01, 89.01 - 44.5, 44.5 - 0.
## Data was bootstrapped 100 times (method:"full").
## Disparity was calculated as: metric.
```

We asked for three subsets (evenly spread across the age of the tree), the data was bootstrapped 100 times (default) and the metric used was the sum of variances.

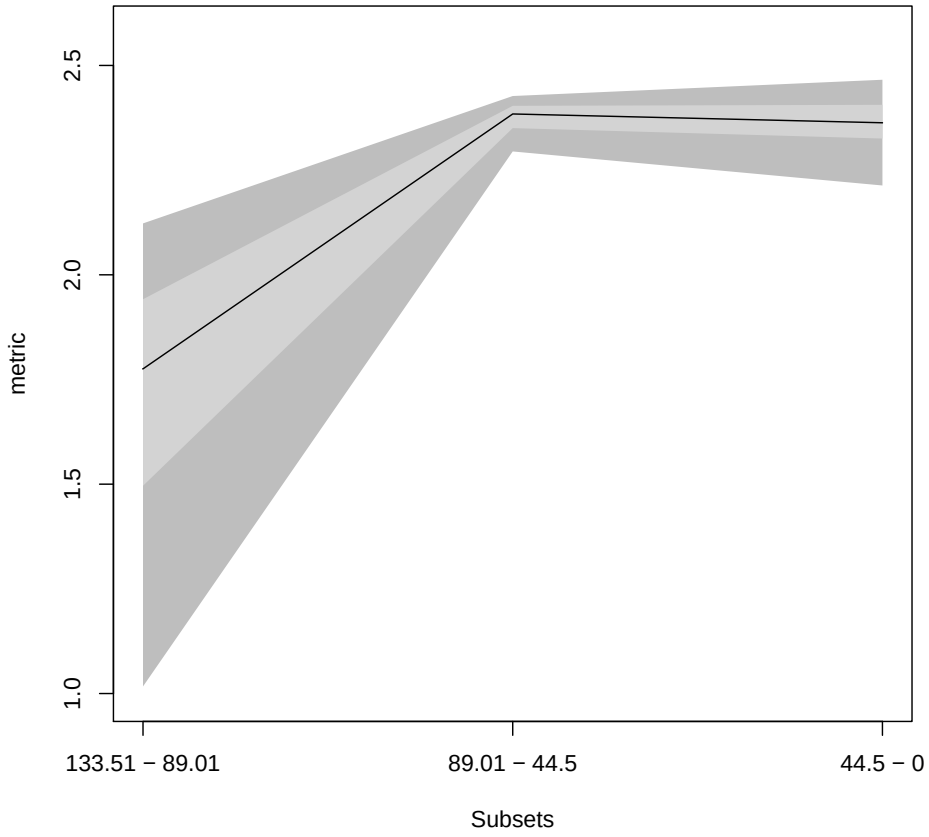
We can now summarise or plot the `disparity_data` object, or perform statistical tests on it (e.g. a simple `lm`):

```
## Summarising disparity through time
summary(disparity_data)
```

```
##          subsets  n   obs bs.median  2.5%   25%   75% 97.5%
```

```
## 1 133.51 - 89.01  5 2.123     1.775 1.017 1.496 1.942 2.123
## 2   89.01 - 44.5 29 2.456     2.384 2.295 2.350 2.404 2.427
## 3       44.5 - 0 16 2.528     2.363 2.213 2.325 2.406 2.466
```

```
## Plotting the results
plot(disparity_data, type = "continuous")
```




```
## Testing for an difference among the time bins
disp_lm <- test.dispRity(disparity_data, test = lm,
                         comparisons = "all")
summary(disp_lm)
```

```
##
## Call:
## test(formula = data ~ subsets, data = data)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -0.87430 -0.04100  0.01456  0.05318  0.41059
##
```

```
## Coefficients:
##                      Estimate Std. Error t value Pr(>|t|)
## (Intercept)           1.71217    0.01703  100.55   <2e-16 ***
## subsets44.5 - 0       0.64824    0.02408   26.92   <2e-16 ***
## subsets89.01 - 44.5   0.66298    0.02408   27.53   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.1703 on 297 degrees of freedom
## Multiple R-squared:  0.769,  Adjusted R-squared:  0.7674
## F-statistic: 494.3 on 2 and 297 DF,  p-value: < 2.2e-16
```

Please refer to the specific tutorials for (much!) more information on the nuts and bolts of the package. You can also directly explore the specific function help files within R and navigate to related functions.

### 3.3.3 Disparity among groups

The `dispRity.per.group` function is used if you are interested in looking at disparity among groups rather than through time. For example, you could ask if there is a difference in disparity between two groups?

To perform such an analysis, you will need:

- An matrix with rows as elements and columns as dimensions (always!)
- A list of group members: this list should be a list of numeric vectors or names corresponding to the row names in the matrix. For example `list("A" = c(1,2), "B" = c(3,4))` will create a group $A$ containing elements 1 and 2 from the matrix and a group $B$ containing elements 3 and 4. Note that elements can be present in multiple groups at once.
- Your favourite disparity metric (here the sum of variances)

Using the Beck and Lee [2014] data described above:

```
## Creating the two groups (crown versus stem) as a list
mammal_groups <- crown.stem(BeckLee_tree, inc.nodes = FALSE)

## Measuring disparity for each group
disparity_data <- dispRity.per.group(BeckLee_mat50,
                                     group = mammal_groups,
                                     metric = c(sum, variances))
```

We can display the disparity of both groups by simply looking at the output variable (`disparity_data`) and then summarising the `disparity_data` object and plotting it, and/or by performing a statistical test to compare disparity across the groups (here a Wilcoxon test).

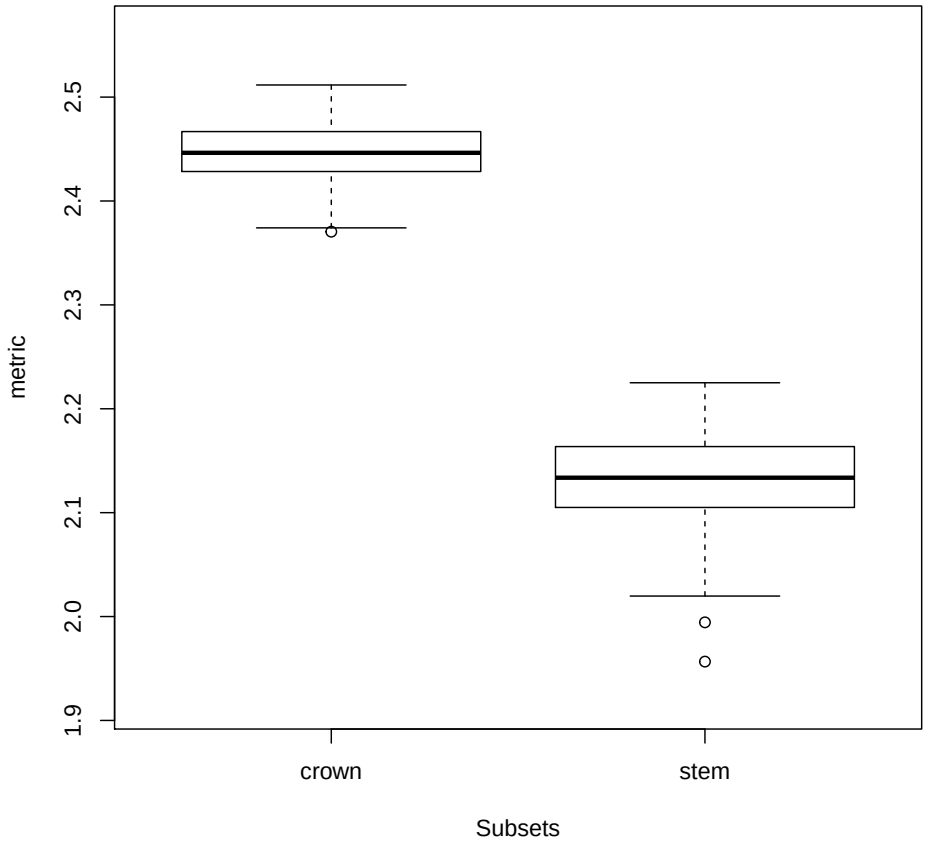
```
## Print the disparity_data object
disparity_data
```

```
##  ---- dispRity object ----
## 2 customised subsets for 50 elements in one matrix with 48 dimensions:
##     crown, stem.
## Data was bootstrapped 100 times (method:"full").
## Disparity was calculated as: metric.
```

```
## Summarising disparity in the different groups
summary(disparity_data)
```

```
##   subsets  n   obs bs.median  2.5%   25%   75% 97.5%
## 1   crown 30 2.526     2.446 2.380 2.429 2.467 2.498
## 2    stem 20 2.244     2.134 2.025 2.105 2.164 2.208
```

```
## Plotting the results
plot(disparity_data)
```



```
## Testing for a difference between the groups
test.dispRity(disparity_data, test = wilcox.test, details = TRUE)
```

```
## $`crown : stem`
## $`crown : stem`[[1]]
##
##  Wilcoxon rank sum test with continuity correction
##
## data:  dots[[1L]][[1L]] and dots[[2L]][[1L]]
## W = 10000, p-value < 2.2e-16
## alternative hypothesis: true location shift is not equal to 0
```

# Chapter 4

# Details of specific functions

The following section contains information specific to some functions. If any of your questions are not covered in these sections, please refer to the function help files in `R`, send me an email (guillert@tcd.ie), or raise an issue on GitHub. The several tutorials below describe specific functionalities of certain functions; please always refer to the function help files for the full function documentation!

Before each section, make sure you loaded the Beck and Lee [2014] data (see example data for more details).

```
## Loading the data
data(BeckLee_mat50)
data(BeckLee_mat99)
data(BeckLee_tree)
data(BeckLee_ages)
```

## 4.1 Time slicing

The function `chrono.subsets` allows users to divide the matrix into different time subsets or slices given a dated phylogeny that contains all the elements (i.e. taxa) from the matrix. Each subset generated by this function will then contain all the elements present at a specific point in time or during a specific period in time.

Two types of time subsets can be performed by using the `method` option:

- Discrete time subsets (or time-binning) using `method = discrete`
- Continuous time subsets (or time-slicing) using `method = continuous`

For the time-slicing method details see Guillerme and Cooper [2018]. For both methods, the function takes the `time` argument which can be a vector of `numeric` values for:

- Defining the boundaries of the time bins (when `method = discrete`)
- 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 dataframe containing the first and last occurrence data (FAD/LAD) for taxa that span over a longer time than the given tips/nodes age, so taxa can appear in more than one time bin/slice.

### 4.1.1 Time-binning

Here is an example for the time binning method (`method = discrete`):

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

```
##  ---- dispRity object ----
## 3 discrete time subsets for 50 elements in one matrix with 1 phylogenetic tree
##     120 - 80, 80 - 40, 40 - 0.
```

Note that we can also generate equivalent results by just telling the function that we want three time-bins as follow:

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

```
##  ---- dispRity object ----
## 3 discrete time subsets for 50 elements in one matrix with 1 phylogenetic tree
##     133.51 - 89.01, 89.01 - 44.5, 44.5 - 0.
```

In this example, the taxa were split inside each time-bin according to their age. However, the taxa here are considered as single points in time. It is totally possible that some taxa could have had longer longevity and that they exist in multiple time bins. In this case, it is possible to include them in more than one bin by providing a table of first and last occurrence dates (FAD/LAD). This table should have the taxa names as row names and two columns for respectively the first and last occurrence age:

```
## Displaying the table of first and last occurrence dates
## for each taxa
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
```

```
## Generating time bins including taxa that might span between them
chrono.subsets(data = BeckLee_mat50, tree = BeckLee_tree,
               method = "discrete",
               time = c(120, 80, 40, 0), FADLAD = BeckLee_ages)
```

```
##  ---- dispRity object ----
## 3 discrete time subsets for 50 elements in one matrix with 1 phylogenetic tree
##     120 - 80, 80 - 40, 40 - 0.
```

When using this method, the oldest boundary of the first bin (or the first slice, see below) is automatically generated as the root age plus 1% of the tree length, as long as at least three elements/taxa 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 among tips *and* nodes.

If you want to generate time subsets based on stratigraphy, the package proposes a useful functions to do it for you: `get.bin.ages` (check out the function's manual in `R`)!

### 4.1.2 Time-slicing

For the time-slicing method (`method = continuous`), the idea is fairly similar. This option, however, requires a matrix that contains the ordinated distance among taxa *and* nodes and an extra argument describing the assumed evolutionary model (via the `model` argument). This model argument is used when the time slice occurs along a branch of the tree rather than on a tip or a node, meaning that a decision must be made about what the value for the branch should be. The model can be one of the following:

- **Punctuated models**
  - `acctran` where the data chosen along the branch is always the one of the descendant
  - `deltran` where the data chosen along the branch is always the one of the ancestor
  - `random` where the data chosen along the branch is randomly chosen between the descendant or the ancestor
  - `proximity` where the data chosen along the branch is either the descendant or the ancestor depending on branch length
- **Gradual models**

  - `equal.split` where the data chosen along the branch is both the descendant and the ancestor with an even probability
  - `gradual.split` where the data chosen along the branch is both the descendant and the ancestor with a probability depending on branch length

  Note that the four first models are a proxy for punctuated evolution: the selected data is always either the one of the descendant or the ancestor. In other words, changes along the branches always occur at either ends of it. The two last models are a proxy for gradual evolution: the data from both the descendant and the ancestor is used with an associate probability. These later models perform better when bootstrapped, effectively approximating the "intermediate" state between and the ancestor and the descendants.

More details about the differences between these methods can be found in Guillerme and Cooper [2018].

```
## Generating four time slices every 40 million years
## under a model of proximity evolution
chrono.subsets(data = BeckLee_mat99, tree = BeckLee_tree,
               method = "continuous", model = "proximity",
               time = c(120, 80, 40, 0),
               FADLAD = BeckLee_ages)
```

```
##  ---- dispRity object ----
## 4 continuous (proximity) time subsets for 99 elements in one matrix with 1 phylogenetic t
##     120, 80, 40, 0.
```

```
## Generating four time slices automatically
chrono.subsets(data = BeckLee_mat99, tree = BeckLee_tree,
               method = "continuous", model = "proximity",
               time = 4, FADLAD = BeckLee_ages)
```

```
##  ---- dispRity object ----
## 4 continuous (proximity) time subsets for 99 elements in one matrix with 1 phylogenetic t
##     133.51, 89.01, 44.5, 0.
```

## 4.2 Customised subsets

Another way of separating elements into different categories is to use customised subsets as briefly explained above. This function simply takes the list of elements to put in each group (whether they are the actual element names or their position in the matrix).

```
## Creating the two groups (crown and stems)
mammal_groups <- crown.stem(BeckLee_tree, inc.nodes = FALSE)
```

```
## Separating the dataset into two different groups
custom.subsets(BeckLee_mat50, group = mammal_groups)
```

```
##  ---- dispRity object ----
## 2 customised subsets for 50 elements in one matrix:
##     crown, stem.
```

Like in this example, you can use the utility function `crown.stem` that allows to automatically separate the crown and stems taxa given a phylogenetic tree. Also, elements can easily be assigned to different groups if necessary!

```
## Creating the three groups as a list
weird_groups <- list("even" = seq(from = 1, to = 49, by = 2),
                     "odd" = seq(from = 2, to = 50, by = 2),
                     "all" = c(1:50))
```

The `custom.subsets` function can also take a phylogeny (as a `phylo` object) as an argument to create groups as clades:

```
## Creating groups as clades
custom.subsets(BeckLee_mat50, group = BeckLee_tree)
```

This automatically creates 49 (the number of nodes) groups containing between two and 50 (the number of tips) elements.

## 4.3 Bootstraps and rarefactions

One important step in analysing ordinated matrices is to pseudo-replicate the data to see how robust the results are, and how sensitive they are to outliers in the dataset. This can be achieved using the function `boot.matrix` to bootstrap and/or rarefy the data. The default options will bootstrap the matrix 100 times without rarefaction using the "full" bootstrap method (see below):

```
## Default bootstrapping
boot.matrix(data = BeckLee_mat50)
```

```
##  ---- dispRity object ----
## 50 elements in one matrix with 48 dimensions.
## Data was bootstrapped 100 times (method:"full").
```

The number of bootstrap replicates can be defined using the `bootstraps` option. The method can be modified by controlling which bootstrap algorithm to use through the `boot.type` argument. Currently two algorithms are implemented:

- `full` where the bootstrapping is entirely stochastic ($n$ elements are replaced by any $m$ elements drawn from the data)
- `single` where only one random element is replaced by one other random element for each pseudo-replicate

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

```
##  ---- dispRity object ----
## 50 elements in one matrix with 48 dimensions.
## Data was bootstrapped 100 times (method:"single").
```

This function also allows users to rarefy the data using the `rarefaction` argument. Rarefaction allows users to limit the number of elements to be drawn at each bootstrap replication. This is useful if, for example, one is interested in looking at the effect of reducing the number of elements on the results of an analysis.

This can be achieved by using the `rarefaction` option that draws only $n$-$x$ at each bootstrap replicate (where $x$ is the number of elements not sampled). The default argument is `FALSE` but it can be set to `TRUE` to fully rarefy the data (i.e. remove $x$ elements for the number of pseudo-replicates, where $x$ varies from the maximum number of elements present in each subset to a minimum of three elements). It can also be set to one or more `numeric` values to only rarefy to the corresponding number of elements.

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

```
##  ---- dispRity object ----
## 50 elements in one matrix with 48 dimensions.
## Data was bootstrapped 20 times (method:"full") and fully rarefied.
```

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

```
##  ---- dispRity object ----
## 50 elements in one matrix with 48 dimensions.
## Data was bootstrapped 20 times (method:"full") and rarefied to 6, 7, 8, 3 elements.
```

> Note that using the `rarefaction` argument also bootstraps the data. In these examples, the function bootstraps the data (without rarefaction) AND also bootstraps the data with the different rarefaction levels.

One other argument is `dimensions` that specifies how many dimensions from the matrix should be used for further analysis. When missing, all dimensions from the ordinated matrix are used.

```
## Using the first 50% of the dimensions
boot.matrix(BeckLee_mat50, dimensions = 0.5)
```

```
##  ---- dispRity object ----
## 50 elements in one matrix with 24 dimensions.
```

```
## Data was bootstrapped 100 times (method:"full").
```

```
## Using the first 10 dimensions
boot.matrix(BeckLee_mat50, dimensions = 10)
```

```
##  ---- dispRity object ----
## 50 elements in one matrix with 1 dimensions.
## Data was bootstrapped 100 times (method:"full").
```

It is also possible to specify the sampling probability in the bootstrap for each elements. This can be useful for weighting analysis for example (i.e. giving more importance to specific elements). These probabilities can be passed to the `prob` argument individually with a vector with the elements names or with a matrix with the rownames as elements names. The elements with no specified probability will be assigned a probability of 1 (or 1/maximum weight if the argument is weights rather than probabilities).

```
## Attributing a weight of 0 to Cimolestes and 10 to Maelestes
boot.matrix(BeckLee_mat50,
            prob = c("Cimolestes" = 0, "Maelestes" = 10))
```

```
##  ---- dispRity object ----
## 50 elements in one matrix with 48 dimensions.
## Data was bootstrapped 100 times (method:"full").
```

Of course, one could directly supply the subsets generated above (using `chrono.subsets` or `custom.subsets`) to this function.

```
## Creating subsets of crown and stem mammals
crown_stem <- custom.subsets(BeckLee_mat50,
                             group = crown.stem(BeckLee_tree,
                                                inc.nodes = FALSE))
## Bootstrapping and rarefying these groups
boot.matrix(crown_stem, bootstraps = 200, rarefaction = TRUE)
```

```
##  ---- dispRity object ----
## 2 customised subsets for 50 elements in one matrix with 48 dimensions:
##     crown, stem.
## Data was bootstrapped 200 times (method:"full") and fully rarefied.
```

```
## Creating time slice subsets
time_slices <- chrono.subsets(data = BeckLee_mat99,
                              tree = BeckLee_tree,
                              method = "continuous",
                              model = "proximity",
                              time = c(120, 80, 40, 0),
                              FADLAD = BeckLee_ages)

## Bootstrapping the time slice subsets
boot.matrix(time_slices, bootstraps = 100)
```

```
##  ---- dispRity object ----
## 4 continuous (proximity) time subsets for 99 elements in one matrix with 97 dimensions wi
##     120, 80, 40, 0.
## Data was bootstrapped 100 times (method:"full").
```

## 4.4 Disparity metrics

There are many ways of measuring disparity! In brief, disparity is a summary metric that will represent an aspect of an ordinated space (e.g. a MDS, PCA, PCO, PCoA). For example, one can look at ellipsoid hyper-volume of the ordinated space (Donohue *et al.* 2013), the sum and the product of the ranges and variances (Wills *et al.* 1994) or the median position of the elements relative to their centroid (Wills *et al.* 1994). Of course, there are many more examples of metrics one can use for describing some aspect of the ordinated space, with some performing better than other ones at particular descriptive tasks, and some being more generalist. Check out this pre-print on selecting the best metric for your specific question on biorXiv. You can also use the `moms` shiny app to test which metric captures which aspect of traitspace occupancy regarding your specific space and your specific question.

Regardless, and because of this great diversity of metrics, the package `dispRity` does not have one way to measure disparity but rather proposes to facilitate users in defining their own disparity metric that will best suit their particular analysis. In fact, the core function of the package, `dispRity`, allows the user to define any metric with the `metric` argument. However the `metric` argument has to follow certain rules:

1. It must be composed from one to three `function` objects;
2. The function(s) must take as a first argument a `matrix` or a `vector`;
3. The function(s) must be of one of the three dimension-levels described below;
4. At least one of the functions must be of dimension-level 1 or 2 (see below).

### 4.4.1 The function dimension-levels

The metric function dimension-levels determine the "dimensionality of decomposition" of the input matrix. In other words, each dimension-level designates the dimensions of the output, i.e. either three (a `matrix`); two (a `vector`); or one (a single `numeric` value) dimension.

#### 4.4.1.1 Dimension-level 1 functions

A dimension-level 1 function will decompose a `matrix` or a `vector` into a single value:

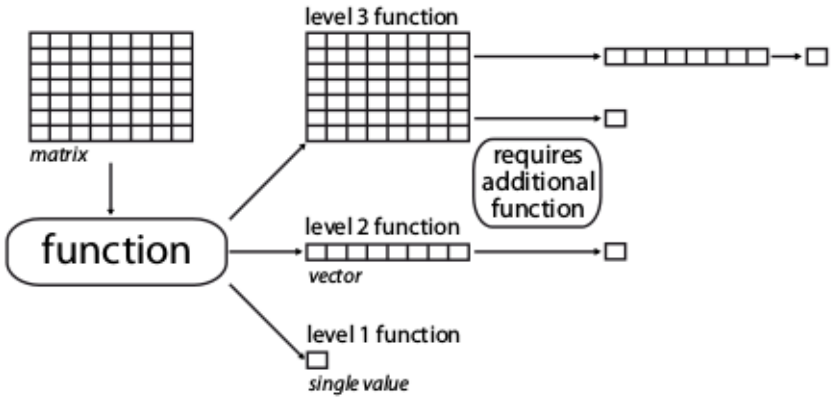



Figure 4.1: Illustration of the different dimension-levels of functions with an input `matrix`

```
## Creating a dummy matrix
dummy_matrix <- matrix(rnorm(12), 4, 3)

## Example of dimension-level 1 functions
mean(dummy_matrix)
```

```
## [1] 0.1012674
```

```
median(dummy_matrix)
```

```
## [1] 0.3345108
```

Any summary metric such as mean or median are good examples of dimension-level 1 functions as they reduce the matrix to a single dimension (i.e. one value).

#### 4.4.1.2 Dimension-level 2 functions

A dimension-level 2 function will decompose a `matrix` into a `vector`.

```
## Defining the function as the product of rows
prod.rows <- function(matrix) apply(matrix, 1, prod)

## A dimension-level 2 metric
prod.rows(dummy_matrix)
```

```
## [1]  0.72217818  2.48612354 -0.08986575  0.58266449
```

Several dimension-level 2 functions are implemented in `dispRity` (see `?dispRity.metric`) such as the `variances` or `ranges` functions that calculate the variance or the range of each dimension of the ordinated matrix respectively.

#### 4.4.1.3 Dimension-level 3 functions

Finally a dimension-level 3 function will transform the matrix into another matrix. Note that the dimension of the output matrix doesn't need to match the the input matrix:

```
## A dimension-level 3 metric
var(dummy_matrix)
```

```
##            [,1]       [,2]       [,3]
## [1,]  1.8570383  0.7417569 -0.5131686
## [2,]  0.7417569  1.3194330 -1.5344429
## [3,] -0.5131686 -1.5344429  2.8070556
```

```
## A dimension-level 3 metric with a forced matrix output
as.matrix(dist(dummy_matrix))
```

```
##          1        2        3        4
## 1 0.000000 4.794738 3.382990 3.297110
## 2 4.794738 0.000000 2.400321 3.993864
## 3 3.382990 2.400321 0.000000 2.187412
## 4 3.297110 3.993864 2.187412 0.000000
```

### 4.4.2 Between groups metrics

One specific category of metrics in the `dispRity` package is the between groups metrics. As the name suggest, these metrics can be used to calculate the disparity between groups rather than within the groups. These metrics follow the same classifications as the "normal" (within group) metrics with dimension-level 1, 2 and 3 between groups metrics. However, at the difference of the "normal" metrics, their input arguments must be `matrix` and `matrix2` (and of course any other additional arguments). For example, this metric measures the difference in mean between two matrices:

```
## A simple example
mean.difference <- function(matrix, matrix2) {
  mean(matrix) - mean(matrix2)
}
```

You can find the list of implemented between groups metric here or design them yourself for your specific needs (potentially using `make.metric` for help).

The function works by simply using the two available matrices, with no restriction in terms of dimensions (although you'd probably want both matrices to have the same number of dimensions)

```
## A second matrix
dummy_matrix2 <- matrix(runif(12), 4, 3)
```

```
## The difference between groups
mean.difference(dummy_matrix, dummy_matrix2)
```

```
## [1] -0.3194556
```

Beyond this super simple example, it might probably be interesting to use this metric on `dispRity` objects, especially the ones from `custom.subsets` and `chrono.subsets`. In fact, the `dispRity` function allows to apply the between groups metric directly to the `dispRity` objects using the `between.groups = TRUE` option. For example:

```
## Combining both matrices
big_matrix <- rbind(dummy_matrix, dummy_matrix2)
rownames(big_matrix) <- 1:8

## Making a dispRity object with both groups
grouped_matrix <- custom.subsets(big_matrix,
                                 group = c(list(1:4), list(1:4)))

## Calculating the mean difference between groups
(mean_differences <- dispRity(grouped_matrix,
                              metric = mean.difference,
                              between.groups = TRUE))
```

```
##  ---- dispRity object ----
## 2 customised subsets for 8 elements in one matrix with 3 dimensions:
##     1, 2.
## Disparity was calculated as: mean.difference between groups.
```

```
## Summarising the object
summary(mean_differences)
```

```
##   subsets n_1 n_2 obs
## 1     1:2   4   4   0
```

```
## Note how the summary table now indicates
## the number of elements for each group
```

For `dispRity` objects generated by `custom.subsets`, the `dispRity` function will by default apply the metric on the groups in a pairwise fashion. For example, if the object contains multiple groups, all groups will be compared to each other:

```
## A dispRity object with multiple groups
grouped_matrix <- custom.subsets(big_matrix,
                                 group = c("A" = list(1:4),
                                           "B" = list(1:4),
                                           "C" = list(2:6),
                                           "D" = list(1:8)))
```

```
## Measuring disparity between all groups
summary(dispRity(grouped_matrix, metric = mean.difference,
                 between.groups = TRUE))
```

```
##   subsets n_1 n_2    obs
## 1     A:B   4   4  0.000
## 2     A:C   4   5 -0.172
## 3     A:D   4   8 -0.160
## 4     B:C   4   5 -0.172
## 5     B:D   4   8 -0.160
## 6     C:D   5   8  0.012
```

For `dispRity` objects generated by `chrono.subsets` (not shown here), the `dispRity` function will by default apply the metric on the groups in a serial way (group 1 vs. group 2, group 2 vs. group 3, group 3 vs. group 4, etc…). However, in both cases (for objects from `custom.subsets` or `chrono.subsets`) it is possible to manually specific the list of pairs of comparisons through their ID numbers:

```
## Measuring disparity between specific groups
summary(dispRity(grouped_matrix, metric = mean.difference,
                 between.groups = list(c(1,3), c(3,1), c(4,1))))
```

```
##   subsets n_1 n_2    obs
## 1     A:C   4   5 -0.172
## 2     C:A   5   4  0.172
## 3     D:A   8   4  0.160
```

Note that in any case, the order of the comparison can matter. In our example, it is obvious that `mean(matrix) - mean(matrix2)` is not the same as `mean(matrix2) - mean(matrix)`.

### 4.4.3 `make.metric`

Of course, functions can be more complex and involve multiple operations such as the `centroids` function (see `?dispRity.metric`) that calculates the Euclidean distance between each element and the centroid of the ordinated space. The `make.metric` function implemented in `dispRity` is designed to help test and find the dimension-level of the functions. This function tests:

1. If your function can deal with a `matrix` or a `vector` as an input;
2. Your function's dimension-level according to its output (dimension-level 1, 2 or 3, see above);
3. Whether the function can be implemented in the `dispRity` function (the function is fed into a `lapply` loop).

For example, let's see if the functions described above are the right dimension-levels:

```
## Which dimension-level is the mean function?
## And can it be used in dispRity?
make.metric(mean)
```

```
## mean outputs a single value.
## mean is detected as being a dimension-level 1 function.
```

```
## Which dimension-level is the prod.rows function?
## And can it be used in dispRity?
make.metric(prod.rows)
```

```
## prod.rows outputs a matrix object.
## prod.rows is detected as being a dimension-level 2 function.
```

```
## Which dimension-level is the var function?
## And can it be used in dispRity?
make.metric(var)
```

```
## var outputs a matrix object.
## var is detected as being a dimension-level 3 function.
## Additional dimension-level 2 and/or 1 function(s) will be needed.
```

A non verbose version of the function is also available. This can be done using the option `silent = TRUE` and will simply output the dimension-level of the metric.

```
## Testing whether mean is dimension-level 1
if(make.metric(mean, silent = TRUE) != "level1") {
    message("The metric is not dimension-level 1.")
}
```

```
## Warning in if (make.metric(mean, silent = TRUE) != "level1") {: the condition
## has length > 1 and only the first element will be used
```

```
## Testing whether var is dimension-level 1
if(make.metric(var, silent = TRUE) != "level1") {
    message("The metric is not dimension-level 1.")
}
```

```
## Warning in if (make.metric(var, silent = TRUE) != "level1") {: the condition has
## length > 1 and only the first element will be used
```

```
## The metric is not dimension-level 1.
```

### 4.4.4 Metrics in the `dispRity` function

Using this metric structure, we can easily use any disparity metric in the `dispRity` function as follows:

```
## Measuring disparity as the standard deviation
## of all the values of the
## ordinated matrix (dimension-level 1 function).
summary(dispRity(BeckLee_mat50, metric = sd))
```

```
##   subsets  n   obs
## 1       1 50 0.227
```

```
## Measuring disparity as the standard deviation
## of the variance of each axis of
## the ordinated matrix (dimension-level 1 and 2 functions).
summary(dispRity(BeckLee_mat50, metric = c(sd, variances)))
```

```
##   subsets  n   obs
## 1       1 50 0.032
```

```
## Measuring disparity as the standard deviation
## of the variance of each axis of
## the variance covariance matrix (dimension-level 1, 2 and 3 functions).
summary(dispRity(BeckLee_mat50, metric = c(sd, variances, var)), round = 10)
```

```
##   subsets  n obs
## 1       1 50   0
```

Note that the order of each function in the metric argument does not matter, the `dispRity` function will automatically detect the function dimension-levels (using `make.metric`) and apply them to the data in decreasing order (dimension-level $3 > 2 > 1$).

```
## Disparity as the standard deviation of the variance of each axis of the
## variance covariance matrix:
disparity1 <- summary(dispRity(BeckLee_mat50,
                               metric = c(sd, variances, var)),
                      round = 10)

## Same as above but using a different function order for the metric argument
disparity2 <- summary(dispRity(BeckLee_mat50,
                               metric = c(variances, sd, var)),
                      round = 10)

## Both ways output the same disparity values:
disparity1 == disparity2
```

```
##      subsets    n  obs
## [1,]    TRUE TRUE TRUE
```

In these examples, we considered disparity to be a single value. For example, in the previous example, we defined disparity as the standard deviation of the variances of each column of the variance/covariance matrix (`metric =`

`c(variances, sd, var)`). It is, however, possible to calculate disparity as a distribution.

### 4.4.5 Metrics implemented in `dispRity`

Several disparity metrics are implemented in the `dispRity` package. The detailed list can be found in `?dispRity.metric` along with some description of each metric.

| Level | Name | Description | Source |
|---|---|---|---|
| 2 | `ancestral.dist` | The distance between an element and its ancestor | `dispRity` |
| 2 | `angles` | The angle of main variation of each dimensions | `dispRity` |
| 2 | `centroids` | The distance between each element and the centroid of the ordinated space | `dispRity` |
| 1 | `convhull.surface` | The surface of the convex hull formed by all the elements | `geometry::convhulln$area` |
| 1 | `convhull.volume` | The volume of the convex hull formed by all the elements | `geometry::convhulln$vol` |
| 2 | `deviations` | The minimal distance between each element and a hyperplane | `dispRity` |
| 1 | `diagonal` | The longest distance in the ordinated space (like the diagonal in two dimensions) | `dispRity` |
| 2 | `displacements` | The ratio between the distance from a reference and the distance from the centroid | `dispRity` |
| 1 | `edge.length.tree` | The edge lengths of the elements on a tree | `ape` |
| 1 | `ellipse.volume` | The volume of the ellipsoid of the space | Donohue *et al.* (2013) |
| 1 | `func.div` | The functional divergence (the ratio of deviation from the centroid) | `dispRity` (similar to `FD::dbFD$FDiv` but without abundance) |
| 1 | `func.eve` | The functional evenness (the minimal spanning tree distances evenness) | `dispRity` (similar to `FD::dbFD$FEve` but without abundance) |
| 1 | `group.dist` | The distance between two groups | `dispRity` |
| 1 | `mode.val` | The modal value | `dispRity` |
| 1 | `n.ball.volume` | The hyper-spherical ($n$-ball) volume | `dispRity` |

| Level | Name | Description | Source |
|---|---|---|---|
| 2 | `neighbours` | The distance to specific neighbours (e.g. the nearest neighbours - by default) | `dispRity` |
| 2 | `pairwise.dist` | The pairwise distances between elements | `vegan::vegist` |
| 2 | `point.dist` | The distance between one group and the point of another group | `dispRity` |
| 2 | `projections` | The projections metric but where the vector can be based on a tree | `dispRity` |
| 2 | `projections` | The distance *on* (projection) or *from* (rejection) an arbitrary vector | `dispRity` |
| 2 | `quantiles` | The $n$th quantile range per axis | `dispRity` |
| 2 | `radius` | The radius of each dimensions | `dispRity` |
| 2 | `ranges` | The range of each dimension | `dispRity` |
| 2 | `span.tree.length` | The minimal spanning tree length | `ape` (but see also `vegan::spantree`) |
| 2 | `variances` | The variance of each dimension | `dispRity` |

1: Note that by default, the centroid is the centroid of the elements. It can, however, be fixed to a different value by using the `centroid` argument `centroids(space, centroid = rep(0, ncol(space)))`, for example the origin of the ordinated space.

2: This function uses an estimation of the eigenvalue that only works for MDS or PCoA ordinations (*not* PCA).

You can find more informations on the vast variety of metrics that you can use in your analysis in this preprint.

### 4.4.6 Equations and implementations

Some of the functions described below are implemented in the `dispRity` package and do not require any other packages to calculate (see implementation here).

$$ancestral.dist = \sqrt{\sum_{i=1}^{n} (d_n - Ancestor_n)^2} \tag{4.1}$$

$$centroids = \sqrt{\sum_{i=1}^{n} (d_n - Centroid_d)^2} \tag{4.2}$$

$$diagonal = \sqrt{\sum_{i=1}^{d} |max(d_i) - min(k_i)|} \tag{4.3}$$

$$deviations = \frac{|Ax + By + ... + Nm + Intercept|}{\sqrt{A^2 + B^2 + ... + N^2}} \tag{4.4}$$

$$displacements = \frac{\sqrt{\sum_{i=1}^{n} (d_n - Reference_d)^2}}{\sqrt{\sum_{i=1}^{n} (d_n - Centroid_k)^2}} \tag{4.5}$$

$$ellipse.volume = \frac{\pi^{d/2}}{\Gamma(\frac{d}{2} + 1)} \prod_{i=1}^{d} (\lambda_i^{0.5}) \tag{4.6}$$

$$n.ball.volume = \frac{\pi^{d/2}}{\Gamma(\frac{d}{2} + 1)} \prod_{i=1}^{d} R \tag{4.7}$$

$$projection_{on} = \|\vec{i} \cdot \vec{b}\| \tag{4.8}$$

$$projection_{from} = \|\vec{i} - \vec{i} \cdot \vec{b}\| \tag{4.9}$$

$$radius = |\frac{\sum_{i=1}^{n} d_i}{n} - f(\mathbf{v}d)| \tag{4.10}$$

$$ranges = |max(d_i) - min(d_i)| \tag{4.11}$$

$$variances = \sigma^2 d_i \tag{4.12}$$

$$span.tree.length = \text{branch length} \tag{4.13}$$

Where $d$ is the number of dimensions, $n$ the number of elements, $\Gamma$ is the Gamma distribution, $\lambda_i$ is the eigenvalue of each dimensions, $\sigma^2$ is their variance and $Centroid_k$ is their mean, $Ancestor_n$ is the coordinates of the ancestor of element $n$, $f(\mathbf{v}k)$ is function to select one value from the vector $\mathbf{v}$ of the dimension $k$ (e.g. it's maximum, minimum, mean, etc.), $R$ is the radius of the sphere or the product of the radii of each dimensions ($\prod_{i=1}^{k} R_i$ - for a hyper-ellipsoid), $Reference_k$ is an arbitrary point's coordinates (usually 0), $\vec{b}$ is the vector defined by (`(point1, point2)`), and $\vec{i}$ is the vector defined by (`(point1, i)` where `i` is any row of the matrix).

### 4.4.7 Using the different disparity metrics

Here is a brief demonstration of the main metrics implemented in `dispRity`. First, we will create a dummy/simulated ordinated space using the `space.maker` utility function (more about that here:

```
## Creating a 10*5 normal space
set.seed(1)
dummy_space <- space.maker(10, 5, rnorm)
rownames(dummy_space) <- 1:10
```

We will use this simulated space to demonstrate the different metrics.

#### 4.4.7.1 Volumes and surface metrics

The functions `ellipse.volume`, `convhull.surface`, `convhull.volume` and `n.ball.volume` all measure the surface or the volume of the ordinated space occupied:

Because there is only one subset (i.e. one matrix) in the dispRity object, the operations below are the equivalent of `metric(dummy_space)` (with rounding).

```
## Calculating the ellipsoid volume
summary(dispRity(dummy_space, metric = ellipse.volume))
```

```
##   subsets  n   obs
## 1       1 10 1.061
```

> WARNING: in such dummy space, this gives the estimation of the ellipsoid volume, not the real ellipsoid volume! See the cautionary note in `?ellipse.volume`.

```
## Calculating the convex hull surface
summary(dispRity(dummy_space, metric = convhull.surface))
```

```
##   subsets  n   obs
## 1       1 10 11.91
```

```
## Calculating the convex hull volume
summary(dispRity(dummy_space, metric = convhull.volume))
```

```
##   subsets  n   obs
## 1       1 10 1.031
```

```
## Calculating the convex hull volume
summary(dispRity(dummy_space, metric = n.ball.volume))
```

```
##   subsets  n  obs
## 1       1 10 4.43
```

The convex hull based functions are a call to the `geometry::convhulln` function with the `"FA"` option (computes total area and volume). Also note that they are really sensitive to the size of the dataset.

> Cautionary note: measuring volumes in a high number of dimensions can be strongly affected by the curse of dimensionality that often results in near 0 disparity values. I strongly recommend reading this really intuitive explanation from Toph Tucker.

#### 4.4.7.2 Ranges, variances, quantiles, radius, pairwise distance, neighbours, modal value and diagonal

The functions `ranges`, `variances radius`, `pairwise.dist`, `mode.val` and `diagonal` all measure properties of the ordinated space based on its dimensional properties (they are also less affected by the "curse of dimensionality"):

`ranges`, `variances quantiles` and `radius` work on the same principle and measure the range/variance/radius of each dimension:

```
## Calculating the ranges of each dimension in the ordinated space
ranges(dummy_space)
```

```
## [1] 2.430909 3.726481 2.908329 2.735739 1.588603
```

```
## Calculating disparity as the distribution of these ranges
summary(dispRity(dummy_space, metric = ranges))
```

```
##   subsets  n obs.median  2.5%   25%   75% 97.5%
## 1       1 10      2.736 1.673 2.431 2.908 3.645
```

```
## Calculating disparity as the sum and the product of these ranges
summary(dispRity(dummy_space, metric = c(sum, ranges)))
```

```
##   subsets  n   obs
## 1       1 10 13.39
```

```
summary(dispRity(dummy_space, metric = c(prod, ranges)))
```

```
##   subsets  n   obs
## 1       1 10 114.5
```

```
## Calculating the variances of each dimension in the
## ordinated space
variances(dummy_space)
```

```
## [1] 0.6093144 1.1438620 0.9131859 0.6537768 0.3549372
```

```
## Calculating disparity as the distribution of these variances
summary(dispRity(dummy_space, metric = variances))
```

```
##   subsets  n obs.median 2.5%   25%   75% 97.5%
```

```
## 1       1 10      0.654 0.38 0.609 0.913 1.121
```

```
## Calculating disparity as the sum and
## the product of these variances
summary(dispRity(dummy_space, metric = c(sum, variances)))
```

```
##   subsets  n   obs
## 1       1 10 3.675
```

```
summary(dispRity(dummy_space, metric = c(prod, variances)))
```

```
##   subsets  n   obs
## 1       1 10 0.148
```

```
## Calculating the quantiles of each dimension
## in the ordinated space
quantiles(dummy_space)
```

```
## [1] 2.234683 3.280911 2.760855 2.461077 1.559057
```

```
## Calculating disparity as the distribution of these variances
summary(dispRity(dummy_space, metric = quantiles))
```

```
##   subsets  n obs.median  2.5%   25%   75% 97.5%
## 1       1 10      2.461 1.627 2.235 2.761 3.229
```

```
## By default, the quantile calculated is the 95%
## (i.e. 95% of the data on each axis)
## this can be changed using the option quantile:
summary(dispRity(dummy_space, metric = quantiles, quantile = 50))
```

```
##   subsets  n obs.median  2.5%   25%   75% 97.5%
## 1       1 10      0.967 0.899 0.951 0.991 1.089
```

```
## Calculating the radius of each dimension in the ordinated space
radius(dummy_space)
```

```
## [1] 1.4630780 2.4635449 1.8556785 1.4977898 0.8416318
```

```
## By default the radius is the maximum distance from the centre of
## the dimension. It can however be changed to any function:
radius(dummy_space, type = min)
```

```
## [1] 0.05144054 0.14099827 0.02212226 0.17453525 0.23044528
```

```
radius(dummy_space, type = mean)
```

```
## [1] 0.6233501 0.7784888 0.7118713 0.6253263 0.5194332
```

```
## Calculating disparity as the mean average radius
summary(dispRity(dummy_space,
                 metric = c(mean, radius),
                 type = mean))
```

```
##   subsets  n   obs
## 1       1 10 0.652
```

The pairwise distances and the neighbours distances uses the function `vegan::vegdist` and can take the normal `vegdist` options:

```
## The average pairwise euclidean distance
summary(dispRity(dummy_space, metric = c(mean, pairwise.dist)))
```

```
##   subsets  n   obs
## 1       1 10 2.539
```

```
## The distribution of the Manhattan distances
summary(dispRity(dummy_space, metric = pairwise.dist,
                 method = "manhattan"))
```

```
##   subsets  n obs.median  2.5%   25%   75% 97.5%
## 1       1 10      4.427 2.566 3.335 5.672  9.63
```

```
## The average nearest neighbour distances
summary(dispRity(dummy_space, metric = neighbours))
```

```
##   subsets  n obs.median  2.5%   25%   75% 97.5%
## 1       1 10      1.517 1.266 1.432 1.646 2.787
```

```
## The average furthest neighbour manhattan distances
summary(dispRity(dummy_space, metric = neighbours,
                 which = max, method = "manhattan"))
```

```
##   subsets  n obs.median 2.5%   25%   75% 97.5%
## 1       1 10      7.895 6.15 6.852 9.402 10.99
```

Note that this function is a direct call to `vegan::vegdist(matrix, method = method, diag = FALSE, upper = FALSE, ...)`.

The `diagonal` function measures the multidimensional diagonal of the whole space (i.e. in our case the longest Euclidean distance in our five dimensional space). The `mode.val` function measures the modal value of the matrix:

```
## Calculating the ordinated space's diagonal
summary(dispRity(dummy_space, metric = diagonal))
```

```
##   subsets  n   obs
## 1       1 10 3.659
```

```
## Calculating the modal value of the matrix
summary(dispRity(dummy_space, metric = mode.val))
```

```
##   subsets  n   obs
## 1       1 10 -2.21
```

> This metric is only a Euclidean diagonal (mathematically valid) if the dimensions within the space are all orthogonal!

#### 4.4.7.3 Centroids, displacements and ancestral distances metrics

The `centroids` metric allows users to measure the position of the different elements compared to a fixed point in the ordinated space. By default, this function measures the distance between each element and their centroid (centre point):

```
## The distribution of the distances between each element and their centroid
summary(dispRity(dummy_space, metric = centroids))
```

```
##   subsets  n obs.median  2.5%   25%   75% 97.5%
## 1       1 10      1.435 0.788 1.267 1.993 3.167
```

```
## Disparity as the median value of these distances
summary(dispRity(dummy_space, metric = c(median, centroids)))
```

```
##   subsets  n   obs
## 1       1 10 1.435
```

It is however possible to fix the coordinates of the centroid to a specific point in the ordinated space, as long as it has the correct number of dimensions:

```
## The distance between each element and the origin
## of the ordinated space
summary(dispRity(dummy_space, metric = centroids, centroid = 0))
```

```
##   subsets  n obs.median  2.5% 25%   75% 97.5%
## 1       1 10      1.487 0.785 1.2 2.044 3.176
```

```
## Disparity as the distance between each element
## and a specific point in space
summary(dispRity(dummy_space, metric = centroids,
                 centroid = c(0,1,2,3,4)))
```

```
##   subsets  n obs.median  2.5%   25%   75% 97.5%
## 1       1 10      5.489 4.293 5.032 6.155 6.957
```

If you have subsets in your `dispRity` object, you can also use the `matrix.dispRity` (see utilities) and `colMeans` to get the centre of a specific subgroup. For example

```
## Create a custom subsets object
dummy_groups <- custom.subsets(dummy_space,
                               group = list("group1" = 1:5,
                                            "group2" = 6:10))
summary(dispRity(dummy_groups, metric = centroids,
    centroid = colMeans(matrix.dispRity(dummy_groups, "group1"))))
```

```
##   subsets n obs.median  2.5%   25%   75% 97.5%
## 1  group1 5      2.011 0.902 1.389 2.284 3.320
## 2  group2 5      1.362 0.760 1.296 1.505 1.985
```

The `displacements` distance is the ratio between the `centroids` distance and the `centroids` distance with `centroid = 0`. Note that it is possible to measure a ratio from another point than `0` using the `reference` argument. It gives indication of the relative displacement of elements in the multidimensional space: a score >1 signifies a displacement *away* from the reference. A score of >1 signifies a displacement *towards* the reference.

```
## The relative displacement of the group in space to the centre
summary(dispRity(dummy_space, metric = displacements))
```

```
##   subsets  n obs.median  2.5%   25% 75% 97.5%
## 1       1 10      1.014 0.841 0.925 1.1 1.205
```

```
## The relative displacement of the group to an arbitrary point
summary(dispRity(dummy_space, metric = displacements,
                 reference = c(0,1,2,3,4)))
```

```
##   subsets  n obs.median  2.5%  25%   75% 97.5%
## 1       1 10      3.368 2.066 3.19 4.358 7.166
```

The `ancestral.dist` metric works on a similar principle as the `centroids` function but changes the centroid to be the coordinates of each element's ancestor (if `to.root = FALSE`; default) or to the root of the tree (`to.root = TRUE`). Therefore this function needs a matrix that contains tips and nodes and a tree as additional argument.

```
## A generating a random tree with node labels
my_tree <- makeNodeLabel(rtree(5), prefix = "n")
## Adding the tip and node names to the matrix
dummy_space2 <- dummy_space[-1,]
rownames(dummy_space2) <- c(my_tree$tip.label,
                            my_tree$node.label)

## Calculating the distances from the ancestral nodes
ancestral_dist <- dispRity(dummy_space2, metric = ancestral.dist,
                           tree = my_tree)

## The ancestral distances distributions
summary(ancestral_dist)
```

```
##   subsets n obs.median  2.5%   25%   75% 97.5%
## 1       1 9      1.729 0.286 1.653 1.843 3.981
```

```
## Calculating disparity as the sum of the distances from all the ancestral nodes
summary(dispRity(ancestral_dist, metric = sum))
```

```
##   subsets n   obs
## 1       1 9 17.28
```

#### 4.4.7.4 Minimal spanning tree length

The `span.tree.length` uses the `vegan::spantree` function to heuristically calculate the minimum spanning tree (the shortest multidimensional tree connecting each elements) and calculates its length as the sum of every branch lengths.

```
## The length of the minimal spanning tree
summary(dispRity(dummy_space, metric = c(sum, span.tree.length)))
```

```
##   subsets  n  obs
## 1       1 10 15.4
```

Note that because the solution is heuristic, this metric can take a long time to compute for big matrices.

#### 4.4.7.5 Functional divergence and evenness

The `func.div` and `func.eve` functions are based on the `FD::dpFD` package. They are the equivalent to `FD::dpFD(matrix)$FDiv` and `FD::dpFD(matrix)$FEve` but a bit faster (since they don't deal with abundance data). They are pretty straightforward to use:

```
## The ratio of deviation from the centroid
summary(dispRity(dummy_space, metric = func.div))
```

```
##   subsets  n   obs
## 1       1 10 0.747
```

```
## The minimal spanning tree distances evenness
summary(dispRity(dummy_space, metric = func.eve))
```

```
##   subsets  n   obs
## 1       1 10 0.898
```

```
## The minimal spanning tree manhanttan distances evenness
summary(dispRity(dummy_space, metric = func.eve,
                 method = "manhattan"))
```

```
##   subsets  n   obs
## 1       1 10 0.913
```

#### 4.4.7.6 Orientation: angles and deviations

The `angles` performs a least square regression (via the `lm` function) and returns slope of the main axis of variation for each dimension. This slope can be converted into different units, `"slope"`, `"degree"` (the default) and `"radian"`. This can be changed through the `unit` argument. By default, the angle is measured from the slope 0 (the horizontal line in a 2D plot) but this can be changed through the `base` argument (using the defined `unit`):

```
## The distribution of each angles in degrees for each
## main axis in the matrix
summary(dispRity(dummy_space, metric = angles))
```

```
##   subsets  n obs.median  2.5%   25%   75% 97.5%
## 1       1 10      21.26 -39.8 3.723 39.47    56
```

```
## The distribution of slopes deviating from the 1:1 slope:
summary(dispRity(dummy_space, metric = angles, unit = "slope",
                 base = 1))
```

```
##   subsets  n obs.median  2.5%   25%   75% 97.5%
## 1       1 10      1.389 0.118 1.065 1.823 2.514
```

The `deviations` function is based on a similar algorithm as above but measures the deviation from the main axis (or hyperplane) of variation. In other words, it finds the least square line (for a 2D dataset), plane (for a 3D dataset) or hyperplane (for a >3D dataset) and measures the shortest distances between every points and the line/plane/hyperplane. By default, the hyperplane is fitted using the least square algorithm from `stats::glm`:

```
## The distribution of the deviation of each point
## from the least square hyperplane
summary(dispRity(dummy_space, metric = deviations))
```

```
##   subsets  n obs.median 2.5%   25%   75% 97.5%
## 1       1 10      0.274 0.02 0.236 0.453 0.776
```

It is also possible to specify the hyperplane equation through the `hyperplane` equation. The equation must contain the intercept first and then all the slopes and is interpreted as $intercept + Ax + By + ... + Nd = 0$. For example, a 2 line defined as beta + intercept (e.g. $y = 2x + 1$) should be defined as `hyperplane = c(1, 2, 1)` ($2x - y + 1 = 0$).

```
## The distribution of the deviation of each point
## from a slope (with only the two first dimensions)
summary(dispRity(dummy_space[, c(1:2)], metric = deviations,
                 hyperplane = c(1, 2, -1)))
```

```
##   subsets  n obs.median  2.5%   25%   75% 97.5%
## 1       1 10      0.516 0.038 0.246 0.763  2.42
```

Since both the functions `angles` and `deviations` effectively run a `lm` or `glm` to estimate slopes or hyperplanes, it is possible to use the option `significant = TRUE` to only consider slopes or intercepts that have a slope significantly different than zero using an `aov` with a significant threshold of $p = 0.05$. Note that depending on your dataset, using and `aov` could be completely inappropriate! In doubt, it's probably better to enter your `base` (for `angles`) or your `hyperplane` (for `deviations`) manually so you're sure you know what the function is measuring.

#### 4.4.7.7 Projections and phylo projections: elaboration and exploration

The `projections` metric calculates the geometric projection and corresponding rejection of all the rows in a matrix on an arbitrary vector (respectively the distance *on* and the distance *from* that vector). The function is based on Aguilera and Pérez-Aguila [2004]'s n-dimensional rotation algorithm to use linear algebra in mutidimensional spaces. The projection or rejection can be seen as respectively the elaboration and exploration scores on a trajectory (*sensu* Endler et al. [2005]).

By default, the vector (e.g. a trajectory, an axis), on which the data is projected is the one going from the centre of the space (coordinates 0,0, …) and the centroid of the matrix. However, we advice you do define this axis to something more meaningful using the `point1` and `point2` options, to create the vector (the vector's norm will be `dist(point1, point2)` and its direction will be from `point1` towards `point2`).

```
## The elaboration on the axis defined by the first and
## second row in the dummy_space
summary(dispRity(dummy_space, metric = projections,
                              point1 = dummy_space[1,],
                              point2 = dummy_space[2,]))
```

```
##   subsets  n obs.median  2.5%   25%   75% 97.5%
## 1       1 10      0.998 0.118 0.651 1.238 1.885
```

```
## The exploration on the same axis
summary(dispRity(dummy_space, metric = projections,
                              point1 = dummy_space[1,],
                              point2 = dummy_space[2,],
                              measure = "distance"))
```

```
##   subsets  n obs.median 2.5%   25%   75% 97.5%
## 1       1 10      0.719    0 0.568 0.912  1.65
```

By default, the vector `(point1, point2)` is used as unit vector of the projections (i.e. the Euclidean distance between `(point1, point2)` is set to 1) meaning that a projection value (`"distance"` or `"position"`) of $X$ means $X$ times the distance between `point1` and `point2`. If you want use the unit vector of the input matrix or are using a space where Euclidean distances are non-sensical, you can remove this option using `scale = FALSE`:

```
## The elaboration on the same axis using the dummy_space's
## unit vector
summary(dispRity(dummy_space, metric = projections,
                              point1 = dummy_space[1,],
                              point2 = dummy_space[2,],
                              scale = FALSE))
```

```
##   subsets  n obs.median  2.5%   25%  75% 97.5%
## 1       1 10      4.068 0.481 2.655 5.05 7.685
```

The `projections.tree` is the same as the `projections` metric but allows to determine the vector (`(point1, point2)`) using a tree rather than manually entering these points. The function intakes the exact same options as the `projections` function described above at the exception of `point1` and `point2`. Instead it takes a the argument `type` that designates the type of vector to draw from the data based on a phylogenetic tree `phy`. The argument `type` can be a pair of any of the following inputs:

- `"root"`: to automatically use the coordinates of the root of the tree (the first element in `phy$node.label`);
- `"ancestor"`: to automatically use the coordinates of the elements' (i.e. any row in the matrix) most recent ancestor;
- `"tips"`: to automatically use the coordinates from the centroid of all tips;
- `"nodes"`: to automatically use the coordinates from the centroid of all nodes;
- `"livings"`: to automatically use the coordinates from the centroid of all "living" tips (i.e. the tips that are the furthest away from the root);
- `"fossils"`: to automatically use the coordinates from the centroid of all "fossil" tips and nodes (i.e. not the "living" ones);
- any numeric values that can be interpreted as `point1` and `point2` in `projections` (e.g. `0`, `c(0, 1.2, 3/4)`, etc.);
- or a user defined function that with the inputs `matrix` and `phy` and `row` (the element's ID, i.e. the row number in `matrix`).

For example, if you want to measure the projection of each element in the matrix (tips and nodes) on the axis from the root of the tree to each element's most recent ancestor, you can define the vector as `type = c("root", "ancestor")`.

```
## Adding a extra row to dummy matrix (to match dummy_tree)
tree_space <- rbind(dummy_space, root = rnorm(5))
## Creating a random dummy tree (with labels matching the ones from tree_space)
dummy_tree <- rtree(6)
dummy_tree$tip.label <- rownames(tree_space)[1:6]
dummy_tree$node.label <- rownames(tree_space)[rev(7:11)]

## Measuring the disparity as the projection of each element
## on its root-ancestor vector
summary(dispRity(tree_space, metric = projections.tree,
                             tree   = dummy_tree,
                             type   = c("root", "ancestor")))
```

```
## Warning in max(nchar(round(column)), na.rm = TRUE): no non-missing arguments to
## max; returning -Inf

## Warning in max(nchar(round(column)), na.rm = TRUE): no non-missing arguments to
```

```
## max; returning -Inf
```

```
##   subsets  n obs.median  2.5%   25%   75% 97.5%
## 1       1 11         NA 0.229 0.416 0.712 1.016
```

Of course you can also use any other options from the projections function:

```
## A user defined function that's returns the centroid of
## the first three nodes
fun.root <- function(matrix, tree, row = NULL) {
  return(colMeans(matrix[tree$node.label[1:3], ]))
}
## Measuring the unscaled rejection from the vector from the
## centroid of the three first nodes
## to the coordinates of the first tip
summary(dispRity(tree_space, metric  = projections.tree,
                             tree    = dummy_tree,
                             measure = "distance",
                             type    = list(fun.root,
                                            tree_space[1, ])))
```

```
##   subsets  n obs.median  2.5%   25%   75% 97.5%
## 1       1 11      0.606 0.064 0.462 0.733 0.999
```

#### 4.4.7.8 Between group metrics

You can find detailed explanation on how between group metrics work here.

##### 4.4.7.8.1 `group.dist`

The `group.dist` metric allows to measure the distance between two groups in the multidimensional space. This function needs to intake several groups and use the option `between.groups = TRUE` in the `dispRity` function. It calculates the vector normal distance (euclidean) between two groups and returns 0 if that distance is negative. Note that it is possible to set up which quantiles to consider for calculating the distances between groups. For example, one might be interested in only considering the 95% CI for each group. This can be done through the option `probs = c(0.025, 0.975)` that is passed to the `quantile` function. It is also possible to use this function to measure the distance between the groups centroids by calculating the 50% quantile (`probs = c(0.5)`).

```
## Creating a dispRity object with two groups
grouped_space <- custom.subsets(dummy_space,
                      group = list(c(1:5), c(6:10)))

## Measuring the minimum distance between both groups
summary(dispRity(grouped_space, metric = group.dist,
                 between.groups = TRUE))
```

```
##   subsets n_1 n_2 obs
## 1     1:2   5   5   0
```

```
## Measuring the centroid distance between both groups
summary(dispRity(grouped_space, metric = group.dist,
                 between.groups = TRUE, probs = 0.5))
```

```
##   subsets n_1 n_2   obs
## 1     1:2   5   5 0.708
```

```
## Measuring the distance between both group's 75% CI
summary(dispRity(grouped_space, metric = group.dist,
                 between.groups = TRUE, probs = c(0.25, 0.75)))
```

```
##   subsets n_1 n_2   obs
## 1     1:2   5   5 0.059
```

#### 4.4.7.8.2 point.dist

The metric measures the distance between the elements in one group (`matrix`) and a point calculated from a second group (`matrix2`). By default this point is the centroid but can be any point defined by a function passed to the `point` argument. For example, the centroid of `matrix2` is the mean of each column of that matrix so `point = colMeans` (default). This function also takes the `method` argument like previous one described above to measure either the `"euclidean"` (default) or the `"manhattan"` distances:

```
## Measuring the distance between the elements of the first group
## and the centroid of the second group
summary(dispRity(grouped_space, metric = point.dist,
                 between.groups = TRUE))
```

```
##   subsets n_1 n_2 obs.median  2.5%   25%   75% 97.5%
## 1     1:2   5   5      2.182 1.304 1.592 2.191 3.355
```

```
## Measuring the distance between the elements of the second group
## and the centroid of the first group
summary(dispRity(grouped_space, metric = point.dist,
                 between.groups = list(c(2,1))))
```

```
##   subsets n_1 n_2 obs.median 2.5%   25%   75% 97.5%
## 1     2:1   5   5      1.362 0.76 1.296 1.505 1.985
```

```
## Measuring the distance between the elements of the first group
## a point defined as the standard deviation of each column
## in the second group
sd.point <- function(matrix2) {apply(matrix2, 2, sd)}
summary(dispRity(grouped_space, metric = point.dist,
```

```
                point = sd.point, method = "manhattan",
                between.groups = TRUE))
```

```
##   subsets n_1 n_2 obs.median  2.5%   25%   75% 97.5%
## 1     1:2   5   5      4.043 2.467 3.567 4.501 6.884
```

### 4.4.8 Which disparity metric to choose?

The disparity metric that gives the most consistent results is the following one:

```
best.metric <- function() return(42)
```

Joke aside, this is a legitimate question that has no simple answer: **it depends on the dataset and question at hand**. Thoughts on which metric to choose can be find in Guillerme et al. [2020b] and Guillerme et al. [2020a] but again, will ultimately depend on the question and dataset. The question should help figuring out which type of metric is desired: for example, in the question "does the extinction released niches for mammals to evolve", the metric in interest should probably pick up a change in size in the trait space (the release could result in some expansion of the mammalian morphospace); or if the question is "does group X compete with group Y", maybe the metric of interested should pick up changes in position (group X can be displaced by group Y).

In order to visualise what signal different disparity metrics are picking, you can use the `moms` that come with a detailed manual on how to use it.

Alternatively, you can use the `test.metric` function:

#### 4.4.8.1 `test.metric`

This function allows to test whether a metric picks different changes in disparity. It intakes the space on which to test the metric, the disparity metric and the type of changes to apply gradually to the space. Basically this is a type of biased data rarefaction (or non-biased for `"random"`) to see how the metric reacts to specific changes in trait space.

```
# Creating a 2D uniform space
example_space <- space.maker(300, 2, runif)

## Testing the product of ranges metric on the example space
example_test <- test.metric(example_space, metric = c(prod, ranges),
                            shifts = c("random", "size"))
```

By default, the test runs three replicates of space reduction as described in Guillerme et al. [2020b] by gradually removing 10% of the data points following the different algorithms from Guillerme et al. [2020b] (here the `"random"` reduction and the `"size"`) reduction, resulting in a `dispRity` object that can

be summarised or plotted. The number of replicates can be changed using the `replicates` option. Still by default, the function then runs a linear model on the simulated data to measure some potential trend in the changes in disparity. The model can be changed using the `model` option. Finally, the function runs 10 reductions by default from keeping 10% of the data (removing 90%) and way up to keeping 100% of the data (removing 0%). This can be changed using the `steps` option. A good disparity metric for your dataset will typically have no trend in the `"random"` reduction (the metric is ideally not affected by sample size) but should have a trend for the reduction of interest.

```
## The results as a dispRity object
example_test
```

```
## Metric testing:
## The following metric was tested: c(prod, ranges).
## The test was run on the random, size shifts for 3 replicates using the following model:
## lm(disparity ~ reduction, data = data)
## Use summary(x) or plot(x) for more details.
```

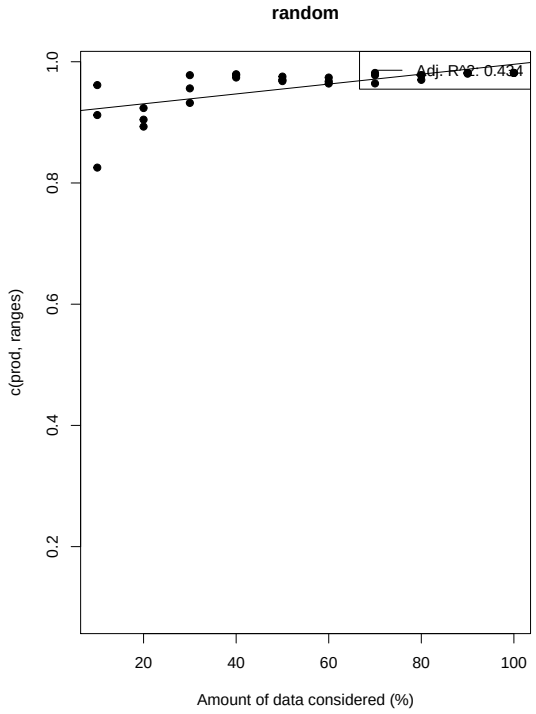
```
## Summarising these results
summary(example_test)
```

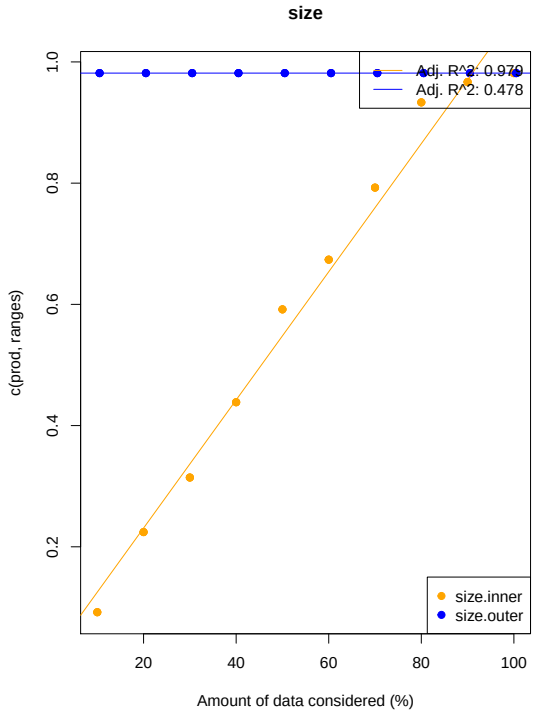
```
## Warning in summary.lm(model): essentially perfect fit: summary may be unreliable

## Warning in summary.lm(model): essentially perfect fit: summary may be unreliable

## Warning in summary.lm(model): essentially perfect fit: summary may be unreliable
```

```
##          10%  20%  30%  40%  50%  60%  70%  80%  90% 100%       slope
## random    0.91 0.90 0.96 0.98 0.97 0.97 0.98 0.98 0.98 0.98  8.149128e-04
## size.inner 0.09 0.22 0.31 0.44 0.59 0.67 0.79 0.93 0.97 0.98  1.057142e-02
## size.outer 0.98 0.98 0.98 0.98 0.98 0.98 0.98 0.98 0.98 0.98 -4.422503e-18
##                  p_value  R^2(adj)
## random     4.530491e-05 0.4340055
## size.inner 2.353746e-25 0.9793579
## size.outer 1.188240e-01 0.4780089
```

```
## Or visualising them
plot(example_test)
```

```
## Warning in summary.lm(model): essentially perfect fit: summary may be unreliable

## Warning in summary.lm(model): essentially perfect fit: summary may be unreliable
```


random
Adj. R^2: 0.434
c(prod, ranges)
1.0
0.8
0.6
0.4
0.2
20
40
60
80
100
Amount of data considered (%)


size
Adj. R^2: 0.979
Adj. R^2: 0.478
size.inner
size.outer
c(prod, ranges)
1.0
0.8
0.6
0.4
0.2
20
40
60
80
100
Amount of data considered (%)

## 4.5 Summarising dispRity data (plots)

Because of its architecture, printing `dispRity` objects only summarises their content but does not print the disparity value measured or associated analysis (more about this here). To actually see what is in a dispRity object, one can either use the `summary` function for visualising the data in a table or `plot` to have a graphical representation of the results.

### 4.5.1 Summarising `dispRity` data

This function is an S3 function (`summary.dispRity`) allowing users to summarise the content of `dispRity` objects that contain disparity calculations.

```
## Example data from previous sections
crown_stem <- custom.subsets(BeckLee_mat50,
                             group = crown.stem(BeckLee_tree,
                             inc.nodes = FALSE))
## Bootstrapping and rarefying these groups
boot_crown_stem <- boot.matrix(crown_stem, bootstraps = 100,
                               rarefaction = TRUE)
## Calculate disparity
disparity_crown_stem <- dispRity(boot_crown_stem,
                                 metric = c(sum, variances))

## Creating time slice subsets
time_slices <- chrono.subsets(data = BeckLee_mat99,
                              tree = BeckLee_tree,
                              method = "continuous",
                              model = "proximity",
                              time = c(120, 80, 40, 0),
                              FADLAD = BeckLee_ages)
## Bootstrapping the time slice subsets
boot_time_slices <- boot.matrix(time_slices, bootstraps = 100)
## Calculate disparity
disparity_time_slices <- dispRity(boot_time_slices,
                                  metric = c(sum, variances))

## Creating time bin subsets
time_bins <- chrono.subsets(data = BeckLee_mat99,
                            tree = BeckLee_tree,
                            method = "discrete",
                            time = c(120, 80, 40, 0),
                            FADLAD = BeckLee_ages,
                            inc.nodes = TRUE)
## Bootstrapping the time bin subsets
```

```
boot_time_bins <- boot.matrix(time_bins, bootstraps = 100)
## Calculate disparity
disparity_time_bins <- dispRity(boot_time_bins,
                                metric = c(sum, variances))
```

These objects are easy to summarise as follows:

```
## Default summary
summary(disparity_time_slices)
```

```
##   subsets  n   obs bs.median  2.5%   25%   75% 97.5%
## 1     120  5 3.258     2.666 1.800 2.447 2.893 3.075
## 2      80 19 3.491     3.314 3.162 3.263 3.369 3.439
## 3      40 15 3.677     3.433 3.171 3.328 3.512 3.672
## 4       0 10 4.092     3.710 3.203 3.560 3.846 4.014
```

Information about the number of elements in each subset and the observed (i.e. non-bootstrapped) disparity are also calculated. This is specifically handy when rarefying the data for example:

```
head(summary(disparity_crown_stem))
```

```
##   subsets  n   obs bs.median  2.5%   25%   75% 97.5%
## 1   crown 30 2.526     2.442 2.374 2.418 2.462 2.488
## 2   crown 29    NA     2.444 2.357 2.420 2.468 2.497
## 3   crown 28    NA     2.443 2.381 2.420 2.463 2.493
## 4   crown 27    NA     2.444 2.365 2.419 2.467 2.495
## 5   crown 26    NA     2.441 2.369 2.418 2.469 2.506
## 6   crown 25    NA     2.442 2.357 2.414 2.461 2.496
```

The summary functions can also take various options such as:

- `quantiles` values for the confidence interval levels (by default, the 50 and 95 quantiles are calculated)
- `cent.tend` for the central tendency to use for summarising the results (default is `median`)
- digits`option corresponding to the number of decimal places to print (default is`2‘)
- `recall` option for printing the call of the `dispRity` object as well (default is `FALSE`)

These options can easily be changed from the defaults as follows:

```
## Same as above but using the 88th quantile and the standard deviation as the summary
summary(disparity_time_slices, quantiles = 88, cent.tend = sd)
```

```
##   subsets  n   obs bs.sd    6%   94%
## 1     120  5 3.258 0.381 1.852 3.000
## 2      80 19 3.491 0.074 3.195 3.422
## 3      40 15 3.677 0.133 3.223 3.647
```

```
## 4       0 10 4.092 0.215 3.318 3.974
```

```
## Printing the details of the object and digits the values to the 5th decimal place
summary(disparity_time_slices, recall = TRUE, digits = 5)
```

```
##  ---- dispRity object ----
## 4 continuous (proximity) time subsets for 99 elements in one matrix with 97 dimensions with 1 phylo
##     120, 80, 40, 0.
## Data was bootstrapped 100 times (method:"full").
## Disparity was calculated as: c(sum, variances).
```

```
##   subsets  n     obs bs.median    2.5%     25%     75%   97.5%
## 1     120  5 3.25815   2.66615 1.79981 2.44681 2.89284 3.07467
## 2      80 19 3.49145   3.31422 3.16207 3.26324 3.36872 3.43851
## 3      40 15 3.67702   3.43316 3.17105 3.32817 3.51160 3.67212
## 4       0 10 4.09234   3.70971 3.20321 3.56007 3.84566 4.01394
```

Note that the summary table is a `data.frame`, hence it is as easy to modify as any dataframe using `dplyr`. You can also export it in `csv` format using `write.csv` or `write_csv` or even directly export into `LaTeX` format using the following;

```
## Loading the xtable package
require(xtable)
## Converting the table in LaTeX
xtable(summary(disparity_time_slices))
```

### 4.5.2 Plotting `dispRity` data

An alternative (and more fun!) way to display the calculated disparity is to plot the results using the S3 method `plot.dispRity`. This function takes the same options as `summary.dispRity` along with various graphical options described in the function help files (see `?plot.dispRity`).

The plots can be of five different types:

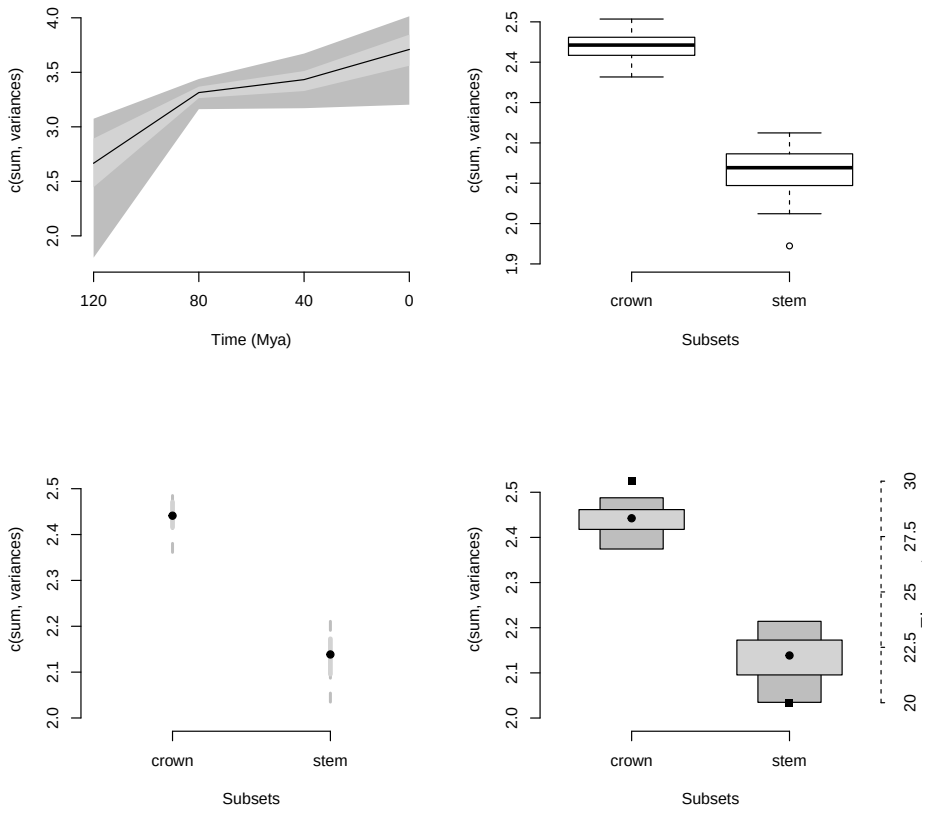
- `preview` for a 2d preview of the trait-space.
- `continuous` for displaying continuous disparity curves
- `box`, `lines`, and `polygons` to display discrete disparity results in respectively a boxplot, confidence interval lines, and confidence interval polygons.

  This argument can be left empty. In this case, the algorithm will automatically detect the type of subsets from the `dispRity` object and plot accordingly.

It is also possible to display the number of elements in each subset (as a horizontal dotted line) using the option `elements = TRUE`. Additionally, when the data is rarefied, one can indicate which level of rarefaction to display (i.e. only

display the results for a certain number of elements) by using the `rarefaction` argument.

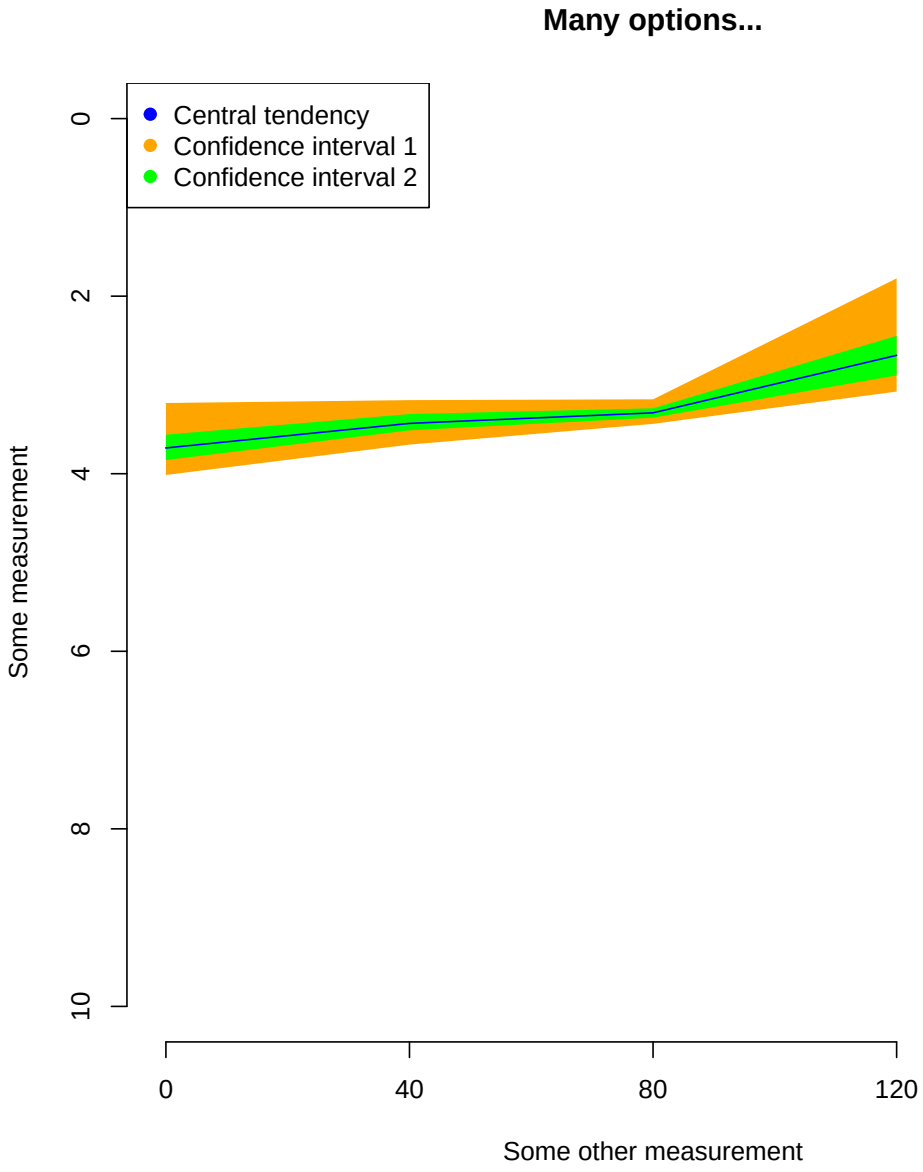
```
## Graphical parameters
op <- par(mfrow = c(2, 2), bty = "n")

## Plotting continuous disparity results
plot(disparity_time_slices, type = "continuous")

## Plotting discrete disparity results
plot(disparity_crown_stem, type = "box")

## As above but using lines for the rarefaction level of 20 elements only
plot(disparity_crown_stem, type = "line", rarefaction = 20)

## As above but using polygons while also displaying the number of elements
plot(disparity_crown_stem, type = "polygon", elements = TRUE)
```




```
## Resetting graphical parameters
par(op)
```

Since `plot.dispRity` uses the arguments from the generic `plot` method, it is of course possible to change pretty much everything using the regular plot arguments:

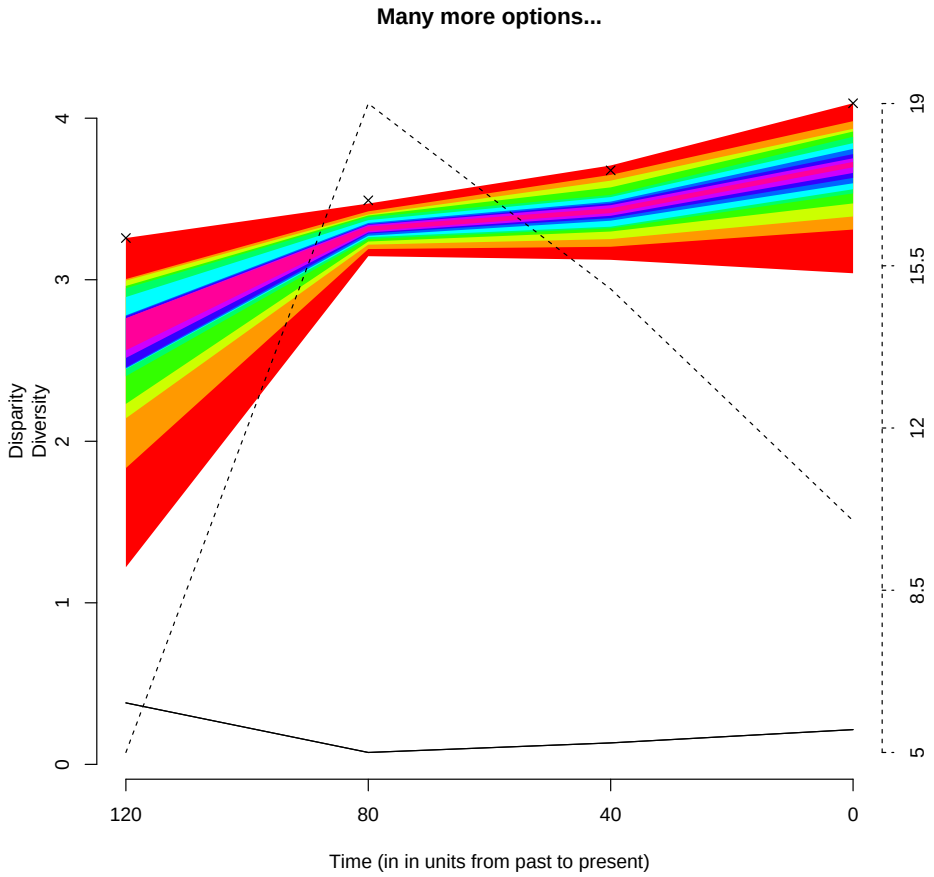
```
## Graphical options
op <- par(bty = "n")

## Plotting the results with some classic options from plot
plot(disparity_time_slices, col = c("blue", "orange", "green"),
    ylab = c("Some measurement"), xlab = "Some other measurement",
    main = "Many options...", ylim = c(10, 0), xlim = c(4, 0))

## Adding a legend
legend("topleft", legend = c("Central tendency",
                             "Confidence interval 1",
                             "Confidence interval 2"),
      col = c("blue", "orange", "green"), pch = 19)
```




```
## Resetting graphical parameters
par(op)
```

In addition to the classic `plot` arguments, the function can also take arguments that are specific to `plot.dispRity` like adding the number of elements or rarefaction level (as described above), and also changing the values of the quantiles to plot as well as the central tendency.

```
## Graphical options
op <- par(bty = "n")

## Plotting the results with some plot.dispRity arguments
plot(disparity_time_slices,
    quantiles = c(seq(from = 10, to = 100, by = 10)),
    cent.tend = sd, type = "c", elements = TRUE,
    col = c("black", rainbow(10)),
    ylab = c("Disparity", "Diversity"),
    xlab = "Time (in in units from past to present)",
    observed = TRUE,
    main = "Many more options...")
```



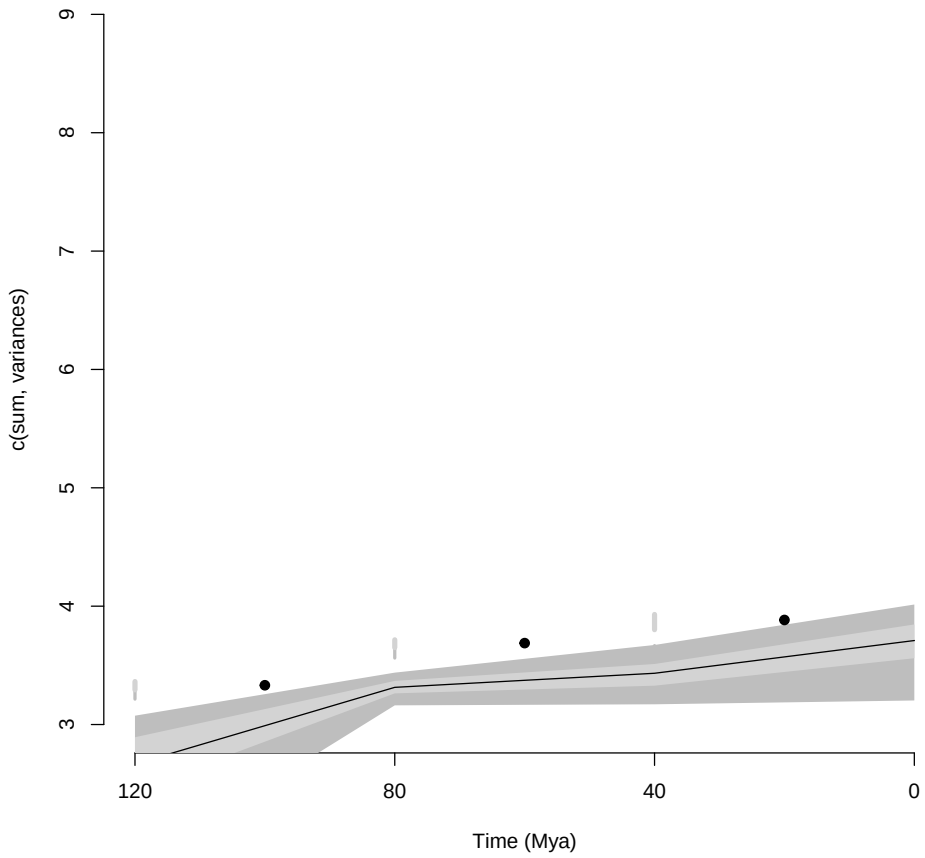

```
## Resetting graphical parameters
par(op)
```

Note that the argument `observed = TRUE` allows to plot the disparity values calculated from the non-bootstrapped data as crosses on the

plot.

For comparing results, it is also possible to add a plot to the existent plot by using `add = TRUE`:

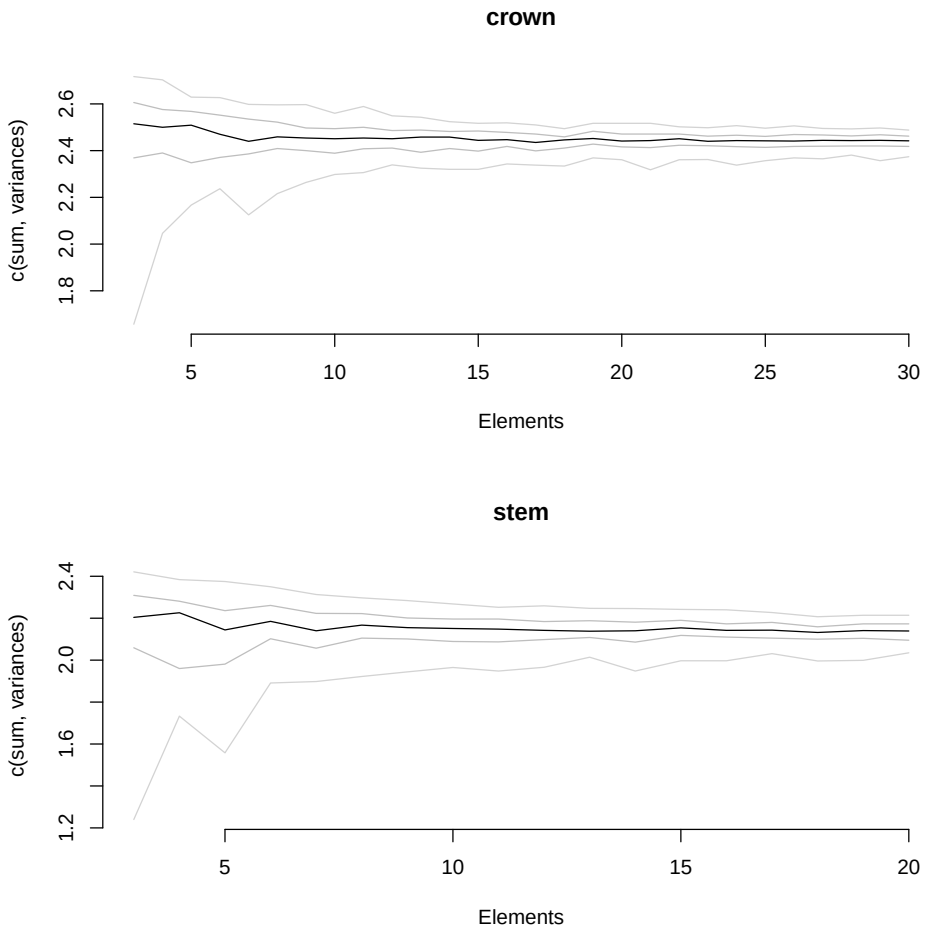
```
## Graphical options
op <- par(bty = "n")

## Plotting the continuous disparity with a fixed y axis
plot(disparity_time_slices, ylim = c(3, 9))
## Adding the discrete data
plot(disparity_time_bins, type = "line", ylim = c(3, 9),
     xlab = "", ylab = "", add = TRUE)
```




```
## Resetting graphical parameters
par(op)
```

Finally, if your data has been fully rarefied, it is also possible to easily look at rarefaction curves by using the `rarefaction = TRUE` argument:

```
## Graphical options
op <- par(bty = "n")

## Plotting the rarefaction curves
plot(disparity_crown_stem, rarefaction = TRUE)
```




```
## Resetting graphical parameters
par(op)
```

### 4.5.3 type = preview

Note that all the options above are plotting disparity objects for which a disparity metric *has been calculated*. This makes totally sense for `dispRity` objects but sometimes it might be interesting to look at what the trait-space looks like before measuring the disparity. This can be done by plotting `dispRity` objects

with no calculated disparity!

For example, we might be interested in looking at how the distribution of elements change as a function of the distributions of different sub-settings. For example custom subsets *vs.* time subsets:

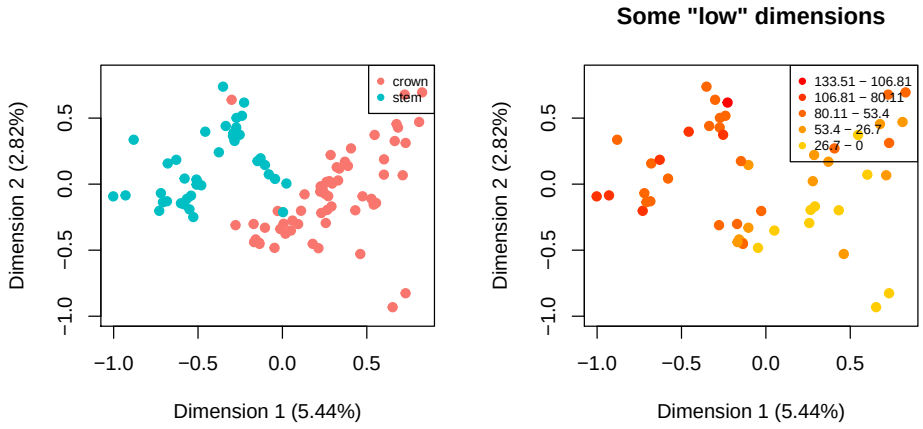
```
## Making the different subsets
cust_subsets <- custom.subsets(BeckLee_mat99,
                               crown.stem(BeckLee_tree,
                                          inc.nodes = TRUE))
time_subsets <- chrono.subsets(BeckLee_mat99,
                               tree = BeckLee_tree,
                               method = "discrete",
                               time = 5)

## Note that no disparity has been calculated here:
is.null(cust_subsets$disparity)
```

```
## [1] TRUE
```

```
is.null(time_subsets$disparity)
```

```
## [1] TRUE
```

```
## But we can still plot both spaces by using the default plot functions
par(mfrow = c(1,2))
## Default plotting
plot(cust_subsets)
## Plotting with more arguments
plot(time_subsets, specific.args = list(dimensions = c(1,2)),
     main = "Some \"low\" dimensions")
```

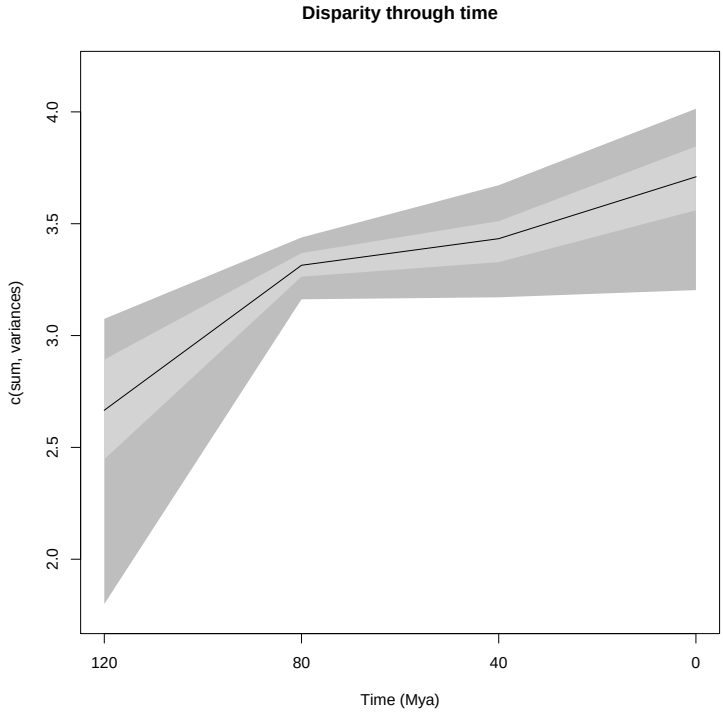


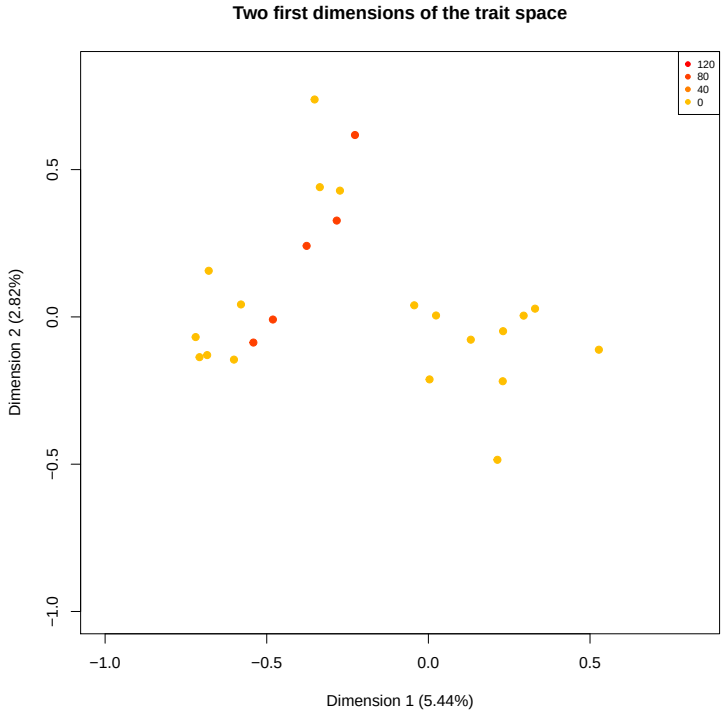

DISCLAIMER: This functionality can be handy for exploring the data (e.g. to visually check whether the subset attribution worked) but it might be misleading on how the data is *actually* distributed

in the multidimensional space! Groups that don't overlap on two set dimensions can totally overlap in all other dimensions!

For `dispRity` objects that do contain disparity data, the default option is to plot your disparity data. However you can always force the `preview` option using the following:

```
par(mfrow = c(2,1))
## Default plotting
plot(disparity_time_slices, main = "Disparity through time")
## Plotting with more arguments
plot(disparity_time_slices, type = "preview",
     main = "Two first dimensions of the trait space")
```


Disparity through time
c(sum, variances)
4.0
3.5
3.0
2.5
2.0
120
80
40
0
Time (Mya)


Two first dimensions of the trait space
120
80
40
0
Dimension 2 (2.82%)
0.5
0.0
−0.5
−1.0
−1.0
−0.5
0.0
0.5
Dimension 1 (5.44%)

## 4.6 Testing disparity hypotheses

The `dispRity` package allows users to apply statistical tests to the calculated disparity to test various hypotheses. The function `test.dispRity` works in a similar way to the `dispRity` function: it takes a `dispRity` object, a `test` and a `comparisons` argument.

The `comparisons` argument indicates the way the test should be applied to the data:

- `pairwise` (default): to compare each subset in a pairwise manner
- `referential`: to compare each subset to the first subset
- `sequential`: to compare each subset to the following subset
- `all`: to compare all the subsets together (like in analysis of variance)

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

The `test` argument can be any statistical or non-statistical test to apply to the disparity object. It can be a common statistical test function (e.g. `stats::t.test`), a function implemented in `dispRity` (e.g. see `?null.test`) or any function defined by the user.

This function also allows users to correct for Type I error inflation (false positives) 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 classical test 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`.

Here we are using the variables generated in the section above:

```
## T-test to test for a difference in disparity between crown and stem mammals
test.dispRity(disparity_crown_stem, test = t.test)
```

```
## [[1]]
##              statistic: t
## crown : stem     50.69093
##
## [[2]]
##              parameter: df
## crown : stem      159.3761
##
## [[3]]
##                    p.value
## crown : stem 3.250849e-100
```

```
##
## [[4]]
##                     stderr
## crown : stem 0.006027599
```

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

```
## $`crown : stem`
## $`crown : stem`[[1]]
##
##  Welch Two Sample t-test
##
## data:  dots[[1L]][[1L]] and dots[[2L]][[1L]]
## t = 50.691, df = 159.38, p-value < 2.2e-16
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  0.2936403 0.3174489
## sample estimates:
## mean of x mean of y
##  2.438778  2.133233
```

```
## Wilcoxon test applied to time sliced disparity with sequential comparisons,
## with Bonferroni correction
test.dispRity(disparity_time_slices, test = wilcox.test,
              comparisons = "sequential", correction = "bonferroni")
```

```
## [[1]]
##          statistic: W
## 120 : 80           21
## 80 : 40          2305
## 40 : 0           1500
##
## [[2]]
##                p.value
## 120 : 80 1.427461e-33
## 80 : 40  1.376543e-10
## 40 : 0   3.672152e-17
```

```
## Measuring the overlap between distributions in the time bins (using the
## implemented Bhattacharyya Coefficient function - see ?bhatt.coeff)
test.dispRity(disparity_time_bins, test = bhatt.coeff)
```

```
## Warning in test.dispRity(disparity_time_bins, test = bhatt.coeff): Multiple p-values wi
## This can inflate Type I error!
```

```
##                     bhatt.coeff
## 120 - 80 : 80 - 40  0.01414214
## 120 - 80 : 40 - 0   0.01414214
```

```
## 80 - 40 : 40 - 0    0.36695541
```

Because of the modular design of the package, tests can always be made by the user (the same way disparity metrics can be user made). The only condition is that the test can be applied to at least two distributions. In practice, the `test.dispRity` function will pass the calculated disparity data (distributions) to the provided function in either pairs of distributions (if the `comparisons` argument is set to `pairwise`, `referential` or `sequential`) or a table containing all the distributions (`comparisons = all`; this should be in the same format as data passed to `lm`-type functions for example).

### 4.6.1 NPMANOVA in `dispRity`

One often useful test to apply to multidimensional data is the permutational multivariate analysis of variance based on distance matrices `vegan::adonis`. This can be done on `dispRity` objects using the `adonis.dispRity` wrapper function. Basically, this function takes the exact same arguments as `adonis` and a `dispRity` object for data and performs a PERMANOVA based on the distance matrix of the multidimensional space (unless the multidimensional space was already defined as a distance matrix). The `adonis.dispRity` function uses the information from the `dispRity` object to generate default formulas:

- If the object contains customised subsets, it applies the default formula `matrix ~ group` testing the effect of `group` as a predictor on `matrix` (called from the `dispRity` object as `data$matrix` see `dispRitu` object details)
- If the object contains time subsets, it applies the default formula `matrix ~ time` testing the effect of `time` as a predictor (were the different levels of `time` are the different time slices/bins)

```
set.seed(1)
## Generating a random character matrix
character_matrix <- sim.morpho(rtree(20), 50,
                               rates = c(rnorm, 1, 0))

## Calculating the distance matrix
distance_matrix <- as.matrix(dist(character_matrix))

## Creating two groups
random_groups <- list("group1" = 1:10, "group2" = 11:20)

## Generating a dispRity object
random_disparity <- custom.subsets(distance_matrix, random_groups)
```

```
## Warning: custom.subsets is applied on what seems to be a distance matrix.
## The resulting matrices won't be distance matrices anymore!
```

```
## Running a default NPMANOVA
adonis.dispRity(random_disparity)
```

```
##
## Call:
## vegan::adonis(formula = matrix ~ group, data = random_disparity,     method = "euclidean
##
## Permutation: free
## Number of permutations: 999
##
## Terms added sequentially (first to last)
##
##           Df SumsOfSqs MeanSqs F.Model      R2 Pr(>F)
## group      1      14.2  14.200  1.2396 0.06443  0.166
## Residuals 18     206.2  11.456         0.93557
## Total     19     220.4                 1.00000
```

Of course, it is possible to pass customised formulas if the disparity object contains more more groups. In that case the predictors must correspond to the names of the groups explained data must be set as `matrix`:

```
## Creating two groups with two states each
groups <- as.data.frame(matrix(data = c(rep(1,10),
                                        rep(2,10),
                                        rep(c(1,2), 10)),
                               nrow = 20, ncol = 2,
                               dimnames = list(paste0("t", 1:20),
                                        c("g1", "g2"))))

## Creating the dispRity object
multi_groups <- custom.subsets(distance_matrix, groups)
```

```
## Warning: custom.subsets is applied on what seems to be a distance matrix.
## The resulting matrices won't be distance matrices anymore!
```

```
## Running the NPMANOVA
adonis.dispRity(multi_groups, matrix ~ g1 + g2)
```

```
##
## Call:
## vegan::adonis(formula = matrix ~ g1 + g2, data = multi_groups,     method = "euclidean")
##
## Permutation: free
## Number of permutations: 999
##
## Terms added sequentially (first to last)
##
##           Df SumsOfSqs MeanSqs F.Model      R2 Pr(>F)
```

```
## g1         1      14.2  14.200 1.22042 0.06443  0.174
## g2         1       8.4   8.400 0.72194 0.03811  0.884
## Residuals 17     197.8  11.635         0.89746
## Total     19     220.4                 1.00000
```

Finally, it is possible to use objects generated by `chrono.subsets`. In this case, `adonis.dispRity` will applied the `matrix ~ time` formula by default:

```
## Creating time series
time_subsets <- chrono.subsets(BeckLee_mat50, BeckLee_tree,
                               method = "discrete",
                               inc.nodes = FALSE,
                               time = c(100, 85, 65, 0),
                               FADLAD = BeckLee_ages)

## Running the NPMANOVA with time as a predictor
adonis.dispRity(time_subsets)
```

```
## Warning in adonis.dispRity(time_subsets): The input data for adonis.dispRity was not a distance ma
## The results are thus based on the distance matrix for the input data (i.e. dist(data$matrix[[1]])).
## Make sure that this is the desired methodological approach!

##
## Call:
## vegan::adonis(formula = dist(matrix) ~ time, data = time_subsets,    method = "euclidean")
##
## Permutation: free
## Number of permutations: 999
##
## Terms added sequentially (first to last)
##
##           Df SumsOfSqs MeanSqs F.Model      R2 Pr(>F)
## time       2     9.593  4.7966  1.9796 0.07769  0.001 ***
## Residuals 47   113.884  2.4231         0.92231
## Total     49   123.477                 1.00000
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Note that the function warns you that the input data was transformed into a distance matrix. This is reflected in the Call part of the output (`formula = dist(matrix) ~ time`).

To use each time subset as a separate predictor, you can use the `matrix ~ chrono.subsets` formula; this is equivalent to `matrix ~ first_time_subset + second_time_subset + ...`:

```
## Running the NPMANOVA with each time bin as a predictor
adonis.dispRity(time_subsets, matrix ~ chrono.subsets)
```

```
## Warning in adonis.dispRity(time_subsets, matrix ~ chrono.subsets): The input data for adonis.displ
```

```
## The results are thus based on the distance matrix for the input data (i.e. dist(data$mat
## Make sure that this is the desired methodological approach!

##
## Call:
## vegan::adonis(formula = dist(matrix) ~ chrono.subsets, data = time_subsets,    method =
##
## Permutation: free
## Number of permutations: 999
##
## Terms added sequentially (first to last)
##
##            Df SumsOfSqs MeanSqs F.Model      R2 Pr(>F)
## t100to85    1     3.714  3.7144  1.5329 0.03008  0.006 **
## t85to65     1     5.879  5.8788  2.4262 0.04761  0.001 ***
## Residuals 47   113.884  2.4231         0.92231
## Total     49   123.477                 1.00000
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

### 4.6.2 `geiger::dtt` model fitting in `dispRity`

The `dtt` function from the `geiger` package is also often used to compare a trait's disparity observed in living taxa to the disparity of a simulated trait based on a given phylogeny. The `dispRity` package proposes a wrapper function for `geiger::dtt`, `dtt.dispRity` that allows the use of any disparity metric. Unfortunately, this implementation is slower that `geiger::dtt` (so if you're using the metrics implemented in `geiger` prefer the original version) and, as the original function, is limited to ultrametric trees (only living taxa!)...

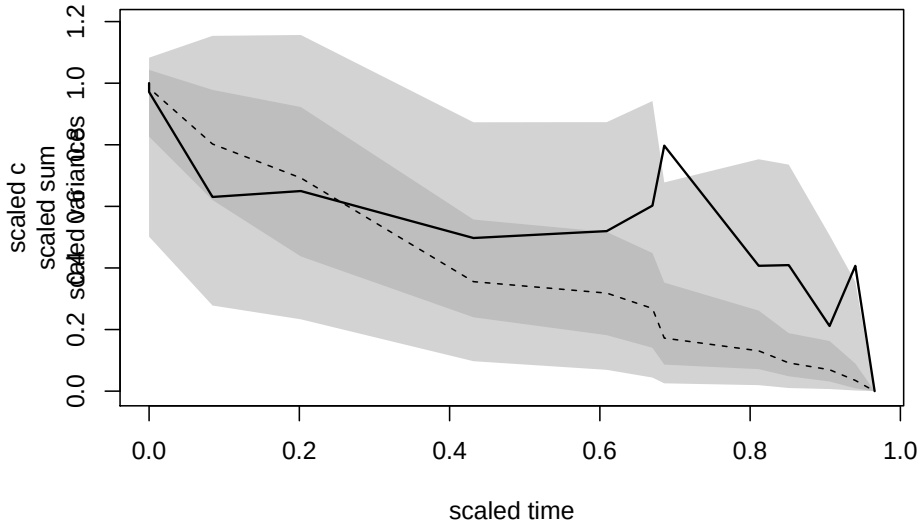
```
require(geiger)
```

```
## Loading required package: geiger
```

```
geiger_data <- get(data(geospiza))

## Calculate the disparity of the dataset using the sum of variance
dispRity_dtt <- dtt.dispRity(data = geiger_data$dat,
                             metric = c(sum, variances),
                             tree = geiger_data$phy,
                             nsim = 100)
```

```
## Warning in dtt.dispRity(data = geiger_data$dat, metric = c(sum, variances), :
## The following tip(s) was not present in the data: olivacea.
```

```
## Plotting the results
plot(dispRity_dtt)
```




Note that, like in the original `dtt` function, it is possible to change the evolutionary model (see `?geiger::sim.char` documentation).

### 4.6.3 null morphospace testing with `null.test`

This test is equivalent to the test performed in Díaz et al. [2016]. It compares the disparity measured in the observed space to the disparity measured in a set of simulated spaces. These simulated spaces can be built with based on the hypothesis assumptions: for example, we can test whether our space is normal.

```
set.seed(123)
## A "normal" multidimensional space with 50 dimensions and 10 elements
normal_space <- matrix(rnorm(1000), ncol = 50)

## Calculating the disparity as the average pairwise distances
obs_disparity <- dispRity(normal_space,
                          metric = c(mean, pairwise.dist))
```

```
## Warning in check.dispRity.data(data): Row names have been automatically added to
## data.
```

```
## Testing against 100 randomly generated normal spaces
(results <- null.test(obs_disparity, replicates = 100,
                      null.distrib = rnorm))
```

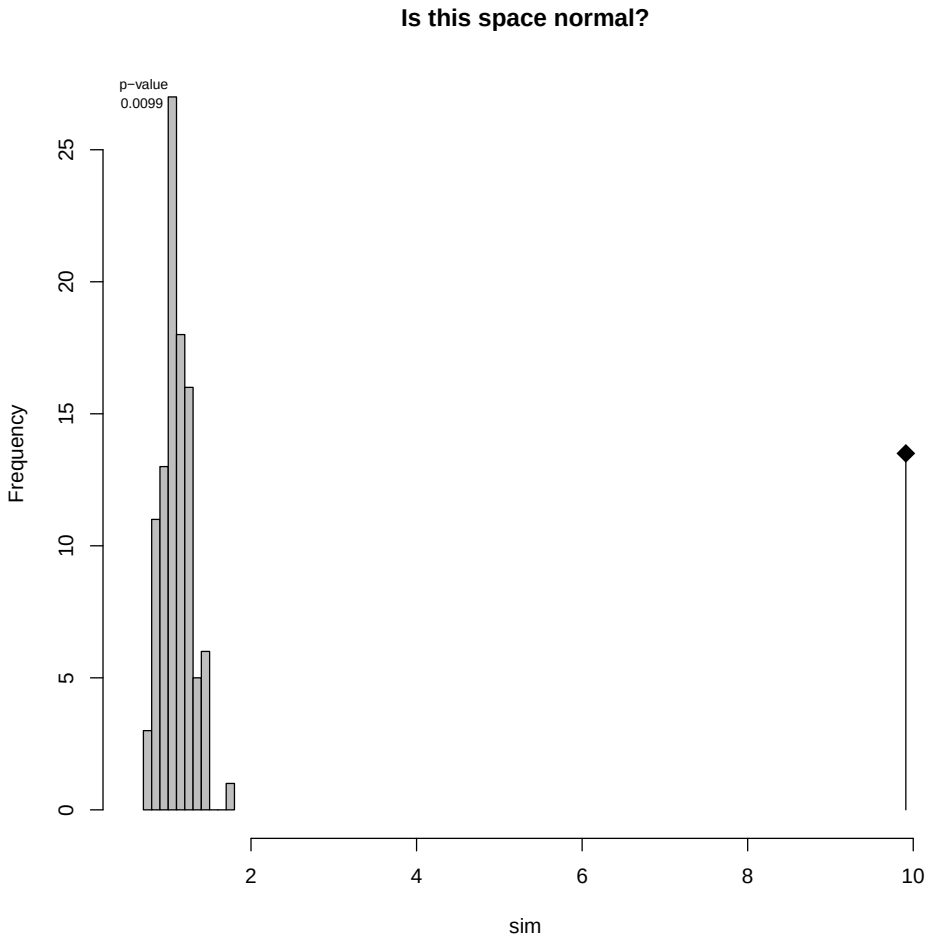
```
## Monte-Carlo test
## Call: [1] "dispRity::null.test"
##
## Observation: 9.910536
##
```

```
## Based on 100 replicates
## Simulated p-value: 0.00990099
## Alternative hypothesis: two-sided
##
##     Std.Obs Expectation    Variance
## 49.07674865  1.10038000  0.03222668
```

Here the results show that disparity measured in our observed space is not significantly different than the one measured in a normal space. We can then propose that our observed space is normal!

These results have an attributed `dispRity` and `randtest` class and can be plotted as `randtest` objects using the `dispRity` S3 `plot` method:

```
## Plotting the results
plot(results, main = "Is this space normal?")
```




For more details on generating spaces see the `space.maker` function tutorial.

## 4.7 Fitting modes of evolution to disparity data

The code used for these models is based on those developed by Gene Hunt [Hunt, 2006, 2012, Hunt et al., 2015]. So we acknowledge and thank Gene Hunt for developing these models and writing the original R code that served as inspiration for these models.

> DISCLAIMER: this method of analysing disparity has not been published yet and has not been peer reviewed. Caution should be used in interpreting these results: it is unclear what "a disparity curve fitting a Brownian motion" actually means biologically.

As Malcolm said in Jurassic Park: "although the examples within this chapter all work and produce solid tested results (from an algorithm point of view), that doesn't mean you should use it" (or something along those lines).

### 4.7.1 Simple modes of disparity change through time

#### 4.7.1.1 `model.test`

Changes in disparity-through-time can follow a range of models, such as random walks, stasis, constrained evolution, trends, or an early burst model of evolution. We will start with by fitting the simplest modes of evolution to our data. For example we may have a null expectation of time-invariant change in disparity in which values fluctuate with a variance around the mean - this would be best describe by a Stasis model:

```
## Loading premade disparity data
data(BeckLee_disparity)
disp_time <- model.test(data = BeckLee_disparity, model = "Stasis")
```

```
## Evidence of equal variance (Bartlett's test of equal variances p = 0).
## Variance is not pooled.
## Running Stasis model...Done. Log-likelihood = -18.694
```

We can see the standard output from `model.test`. The first output message tells us it has tested for equal variances in each sample. The model uses Bartlett's test of equal variances to assess if variances are equal, so if $p > 0.05$ then variance is treated as the same for all samples, but if ($p < 0.05$) then each bin variance is unique. Here we have $p < 0.05$, so variance is not pooled between samples.

By default `model.test` will use Bartlett's test to assess for homogeneity of variances, and then use this to decide to pool variances or not. This is ignored if the argument `pool.variance` in `model.test` is changed from the default `NULL` to `TRUE` or `FALSE`. For example, to ignore Bartlett's test and pool variances manually we would do the following:

```
disp_time_pooled <- model.test(data = BeckLee_disparity,
                               model = "Stasis",
                               pool.variance = TRUE)
```

```
## Running Stasis model...Done. Log-likelihood = -16.884
```

However, unless you have good reason to choose otherwise it is recommended to use the default of `pool.variance = NULL`:

```
disp_time <- model.test(data = BeckLee_disparity,
                        model = "Stasis",
                        pool.variance = NULL)
```

```
## Evidence of equal variance (Bartlett's test of equal variances p = 0).
## Variance is not pooled.
## Running Stasis model...Done. Log-likelihood = -18.694
```

```
disp_time
```

```
## Disparity evolution model fitting:
## Call: model.test(data = BeckLee_disparity, model = "Stasis", pool.variance = NULL)
##
##            aicc delta_aicc weight_aicc
## Stasis 41.48967          0           1
##
## Use x$full.details for displaying the models details
## or summary(x) for summarising them.
```

The remaining output gives us the log-likelihood of the Stasis model of -18.7 (you may notice this change when we pooled variances above). The output also gives us the small sample Akaike Information Criterion (AICc), the delta AICc (the distance from the best fitting model), and the AICc weights (~the relative support of this model compared to all models, scaled to one).

These are all metrics of relative fit, so when we test a single model they are not useful. By using the function summary in `dispRity` we can see the maximum likelihood estimates of the model parameters:

```
summary(disp_time)
```

```
##        aicc delta_aicc weight_aicc log.lik param theta.1 omega
## Stasis 41.5          0           1   -18.7     2     3.6   0.1
```

So we again see the AICc, delta AICc, AICc weight, and the log-likelihood we saw previously. We now also see the number of parameters from the model (2: theta and omega), and their estimates so the variance (omega = 0.1) and the mean (theta.1 = 3.6).

The `model.test` function is designed to test relative model fit, so we need to test more than one model to make relative comparisons. So let's compare to the fit of the Stasis model to another model with two parameters: the Brownian

motion. Brownian motion assumes a constant mean that is equal to the ancestral estimate of the sequence, and the variance around this mean increases linearly with time. The easier way to compare these models is to simply add "BM" to the models vector argument:

```
disp_time <- model.test(data = BeckLee_disparity,
                        model = c("Stasis", "BM"))
```

```
## Evidence of equal variance (Bartlett's test of equal variances p = 0).
## Variance is not pooled.
## Running Stasis model...Done. Log-likelihood = -18.694
## Running BM model...Done. Log-likelihood = 149.289
```

```
disp_time
```

```
## Disparity evolution model fitting:
## Call: model.test(data = BeckLee_disparity, model = c("Stasis", "BM"))
##
##              aicc delta_aicc  weight_aicc
## Stasis   41.48967   335.9656 1.111708e-73
## BM     -294.47595     0.0000 1.000000e+00
##
## Use x$full.details for displaying the models details
## or summary(x) for summarising them.
```

Et voilà! Here we can see by the log-likelihood, AICc, delta AICc, and AICc weight Brownian motion has a much better relative fit to these data than the Stasis model. Brownian motion has a relative AICc fit336 units better than Stasis, and has a AICc weight of 1.

We can also all the information about the relative fit of models alongside the maximum likelihood estimates of model parameters using the summary function

```
summary(disp_time)
```

```
##      aicc delta_aicc weight_aicc log.lik param theta.1 omega ancestral state
## Stasis  41      336        0  -18.7    2  3.629 0.074          NA
## BM    -294        0        1  149.3    2     NA    NA       3.267
##        sigma squared
## Stasis            NA
## BM             0.001
```

Not that because the parameters per models differ, the summary includes NA for inapplicable parameters per models (e.g. the theta and omega parameters from the Stasis models are inapplicable for a Brownian motion model).

We can plot the relative fit of our models using the plot function

```
plot(disp_time)
```

Here we see and overwhelming support for the Brownian motion model.

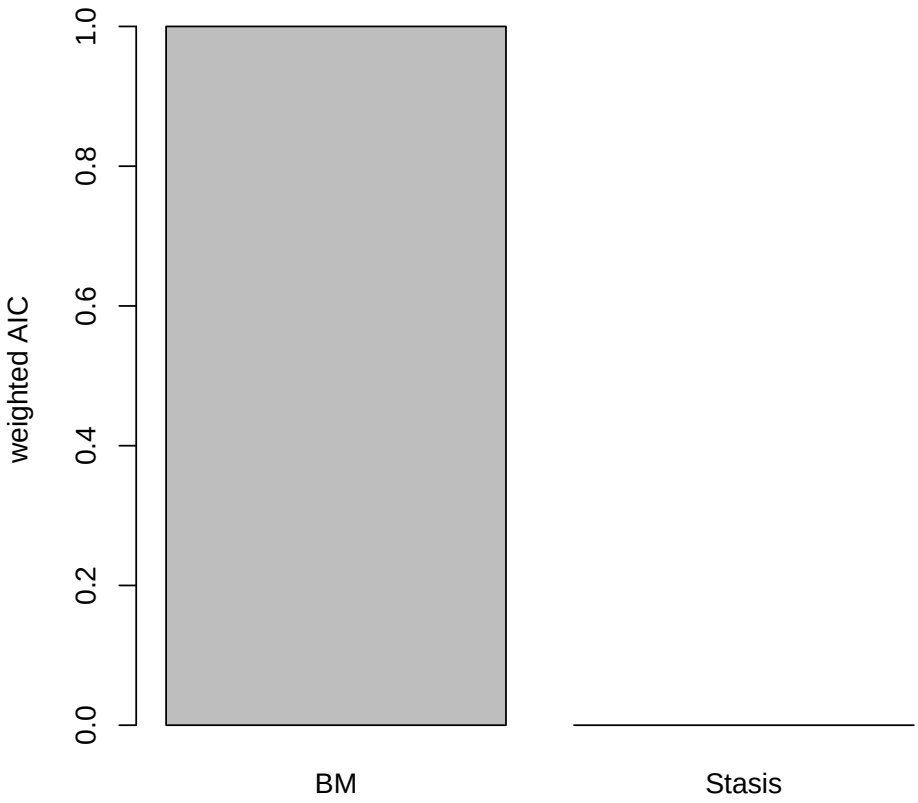



Figure 4.2: relative fit (AICc weight) of Stasis and Brownian models of disparity through time

Alternatively, we could test all available models single modes: Stasis, Brownian motion, Ornstein-Uhlenbeck (evolution constrained to an optima), Trend (increasing or decreasing mean through time), and Early Burst (exponentially decreasing rate through time)

```
disp_time <- model.test(data = BeckLee_disparity,
                  model = c("Stasis", "BM", "OU", "Trend", "EB"))
```

```
## Evidence of equal variance (Bartlett's test of equal variances p = 0).
## Variance is not pooled.
## Running Stasis model...Done. Log-likelihood = -18.694
## Running BM model...Done. Log-likelihood = 149.289
## Running OU model...Done. Log-likelihood = 152.119
## Running Trend model...Done. Log-likelihood = 152.116
## Running EB model...Done. Log-likelihood = 126.268
```

```
summary(disp_time)
```

```
##      aicc delta_aicc weight_aicc log.lik param theta.1 omega ancestral state
## Stasis  41    339.5     0.000  -18.7    2  3.629 0.074          NA
## BM    -294      3.6     0.112  149.3    2    NA   NA        3.267
## OU    -296      2.1     0.227  152.1    4    NA   NA        3.254
## Trend -298      0.0     0.661  152.1    3    NA   NA        3.255
## EB    -246     51.7     0.000  126.3    3    NA   NA        4.092
##        sigma squared alpha optima.1 trend     eb
## Stasis            NA    NA       NA    NA     NA
## BM             0.001    NA       NA    NA     NA
## OU             0.001 0.001    12.88    NA     NA
## Trend          0.001    NA       NA 0.007     NA
## EB             0.000    NA       NA    NA -0.032
```

These models indicate support for a Trend model, and we can plot the relative support of all model AICc weights.

```
plot(disp_time)
```

> Note that although AIC values are indicator of model best fit, it is also important to look at the parameters themselves. For example OU can be really well supported but with an alpha parameter really close to 0, making it effectively a BM model [Cooper et al., 2016].

Is this a trend of increasing or decreasing disparity through time? One way to find out is to look at the summary function for the Trend model:

```
summary(disp_time)["Trend",]
```

```
##        aicc    delta_aicc    weight_aicc      log.lik        param
##    -298.000         0.000         0.661       152.100         3.000
##     theta.1       omega ancestral state sigma squared       alpha
##          NA          NA         3.255         0.001          NA
```

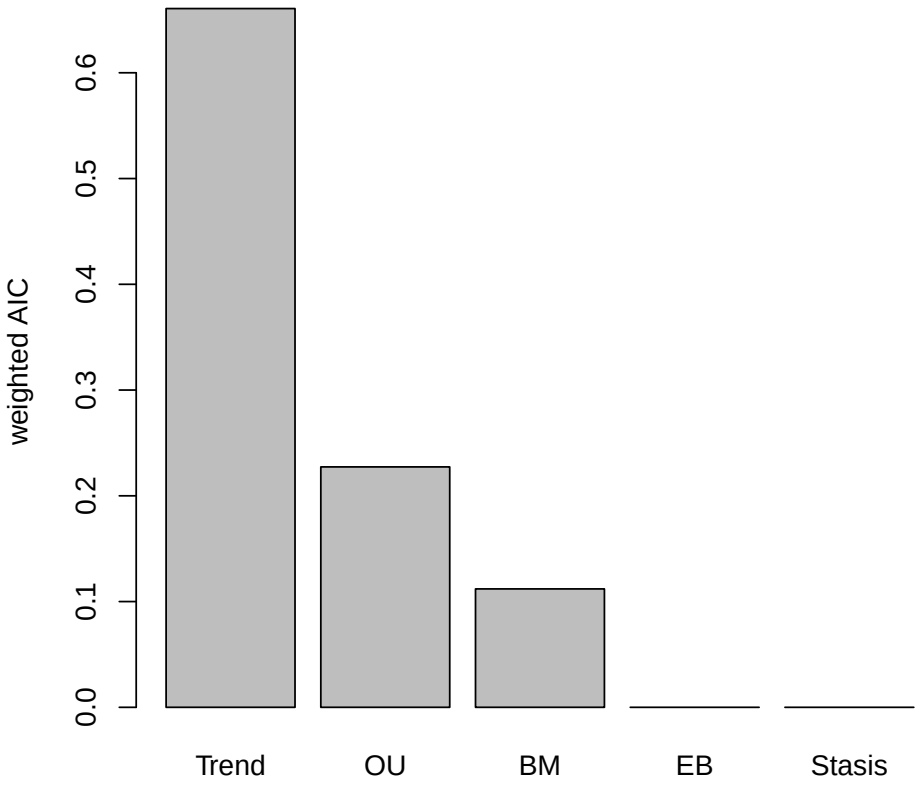



Figure 4.3: relative fit (AICc weight) of various modes of evolution

```
##        optima.1           trend              eb
##              NA           0.007              NA
```

This show a positive trend (0.007) of increasing disparity through time.

## 4.7.2 Plot and run simulation tests in a single step

### 4.7.2.1 `model.test.wrapper`

Patterns of evolution can be fit using `model.test`, but the `model.test.wrapper` fits the same models as `model.test` as well as running predictive tests and plots.

The predictive tests use the maximum likelihood estimates of model parameters to simulate a number of datasets (default = 1000), and analyse whether this is significantly different to the empirical input data using the Rank Envelope test [Murrell, 2018]. Finally we can plot the empirical data, simulated data, and the Rank Envelope test p values. This can all be done using the function `model.test.wrapper`, and we will set the argument `show.p = TRUE` so $p$ values from the Rank Envelope test are printed on the plot:

```
disp_time <- model.test.wrapper(data = BeckLee_disparity,
                    model = c("Stasis", "BM", "OU", "Trend", "EB"),
                                show.p = TRUE)
```

```
## Evidence of equal variance (Bartlett's test of equal variances p = 0).
## Variance is not pooled.
## Running Stasis model...Done. Log-likelihood = -18.694
## Running BM model...Done. Log-likelihood = 149.289
## Running OU model...Done. Log-likelihood = 152.119
## Running Trend model...Done. Log-likelihood = 152.116
## Running EB model...Done. Log-likelihood = 126.268
```

```
disp_time
```

```
##      aicc delta_aicc weight_aicc log.lik param theta.1 omega ancestral state
## Trend -298      0.0     0.661   152.1    3     NA   NA        3.255
## OU    -296      2.1     0.227   152.1    4     NA   NA        3.254
## BM    -294      3.6     0.112   149.3    2     NA   NA        3.267
## EB    -246     51.7     0.000   126.3    3     NA   NA        4.092
## Stasis  41    339.5     0.000   -18.7    2  3.629 0.074          NA
##      sigma squared alpha optima.1 trend    eb median p value lower p value
## Trend       0.001   NA     NA 0.007    NA    0.97752248   0.977022977
## OU          0.001 0.001   12.88   NA    NA    0.97802198   0.978021978
## BM          0.001   NA     NA   NA    NA    0.16283716   0.137862138
## EB          0.000   NA     NA   NA -0.032    0.06893107   0.000999001
## Stasis         NA   NA     NA   NA    NA    1.00000000   1.000000000
##        upper p value
## Trend      0.9780220
```

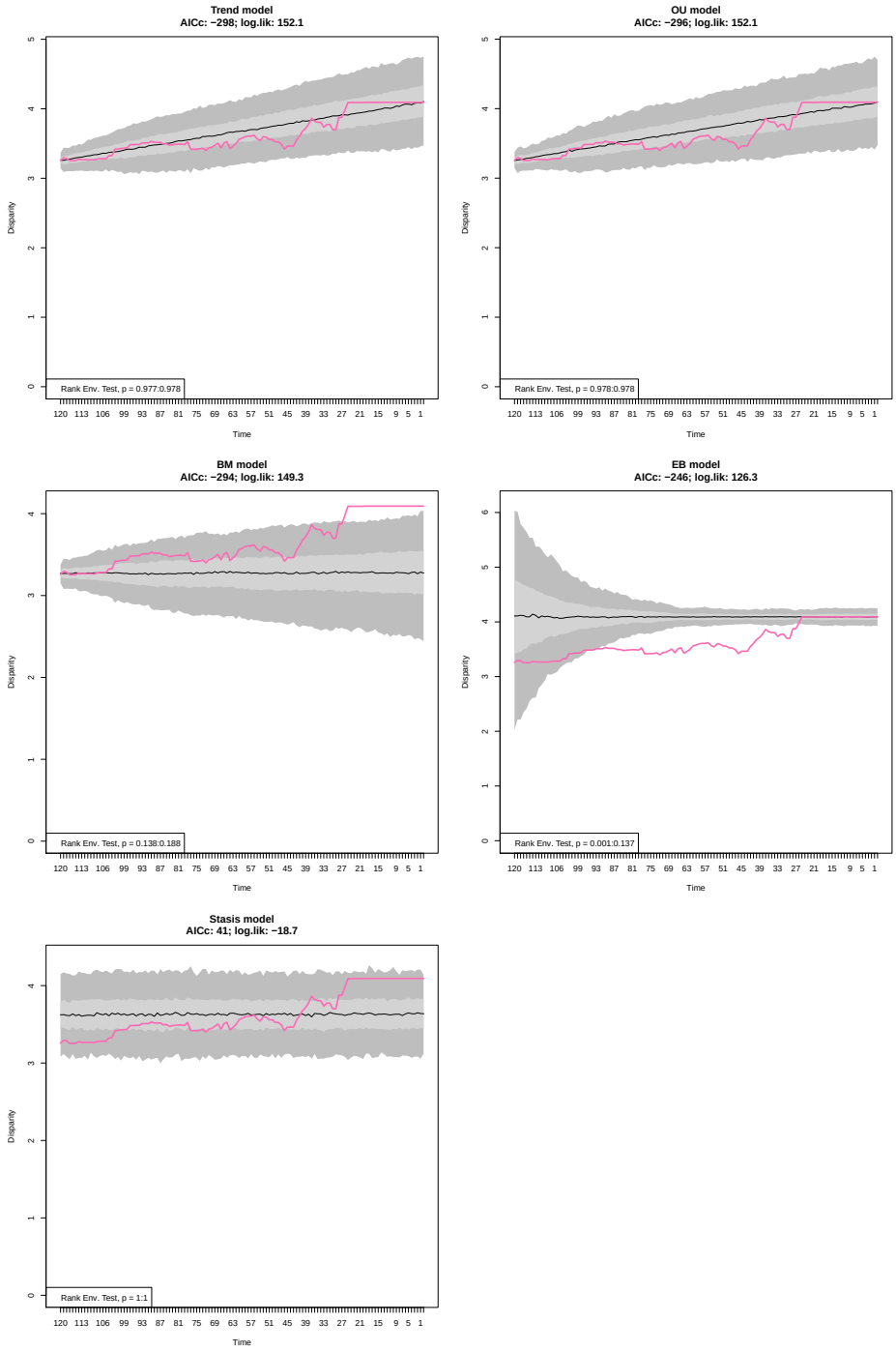



Figure 4.4: Empirical disparity through time (pink), simulate data based on estimated model parameters (grey), delta AICc, and range of p values from the Rank Envelope test for Trend, OU, BM, EB, and Stasis models

```
## OU          0.9780220
## BM          0.1878122
## EB          0.1368631
## Stasis      1.0000000
```

From this plot we can see the empirical estimates of disparity through time (pink) compared to the predictive data based upon the simulations using the estimated parameters from each model. There is no significant differences between the empirical data and simulated data, except for the Early Burst model.

Trend is the best-fitting model but the plot suggests the OU model also follows a trend-like pattern. This is because the optima for the OU model (12.88) is different to the ancestral state (3.254) and outside the observed value. This is potentially unrealistic, and one way to alleviate this issue is to set the optima of the OU model to equal the ancestral estimate - this is the normal practice for OU models in comparative phylogenetics. To set the optima to the ancestral value we change the argument `fixed.optima = TRUE`:

```
disp_time <- model.test.wrapper(data = BeckLee_disparity,
                    model = c("Stasis", "BM", "OU", "Trend", "EB"),
                                show.p = TRUE, fixed.optima = TRUE)
```

```
## Evidence of equal variance (Bartlett's test of equal variances p = 0).
## Variance is not pooled.
## Running Stasis model...Done. Log-likelihood = -18.694
## Running BM model...Done. Log-likelihood = 149.289
## Running OU model...Done. Log-likelihood = 149.289
## Running Trend model...Done. Log-likelihood = 152.116
## Running EB model...Done. Log-likelihood = 126.268
```

```
disp_time
```

```
##      aicc delta_aicc weight_aicc log.lik param theta.1 omega ancestral state
## Trend -298      0.0     0.814  152.1    3     NA   NA        3.255
## BM    -294      3.6     0.138  149.3    2     NA   NA        3.267
## OU    -292      5.7     0.048  149.3    3     NA   NA        3.267
## EB    -246     51.7     0.000  126.3    3     NA   NA        4.092
## Stasis  41    339.5     0.000  -18.7    2  3.629 0.074          NA
##       sigma squared alpha trend    eb median p value lower p value
## Trend         0.001    NA 0.007    NA     0.98351648   0.983016983
## BM            0.001    NA    NA    NA     0.26473526   0.249750250
## OU            0.001     0    NA    NA     0.30469530   0.292707293
## EB            0.000    NA    NA -0.032    0.06943057   0.000999001
## Stasis           NA    NA    NA    NA     0.99900100   0.999000999
##        upper p value
## Trend      0.9840160
## BM         0.2797203
## OU         0.3166833
```

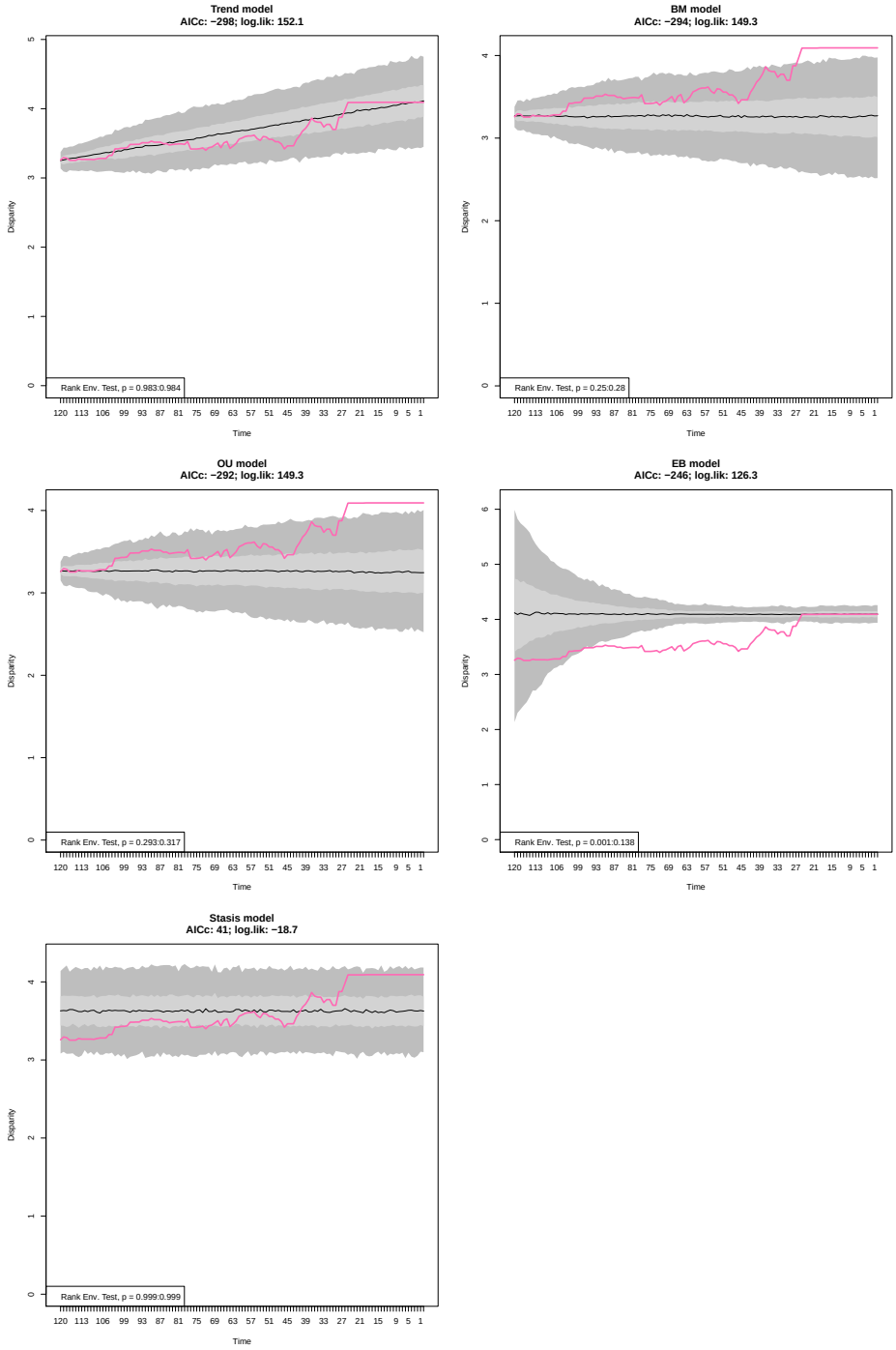



Figure 4.5: Empirical disparity through time (pink), simulate data based on estimated model parameters (grey), delta AICc, and range of p values from the Rank Envelope test for Trend, OU, BM, EB, and Stasis models with the optima of the OU model set to equal the ancestral value

```
## EB          0.1378621
## Stasis      0.9990010
```

The relative fit of the OU model is decreased by constraining the fit of the optima to equal the ancestral state value. In fact as the OU attraction parameter (alpha) is zero, the model is equal to a Brownian motion model but is penalised by having an extra parameter. Note that indeed, the plots of the BM model and the OU model look nearly identical.

### 4.7.3 Multiple modes of evolution (time shifts)

As well as fitting a single model to a sequence of disparity values we can also allow for the mode of evolution to shift at a single or multiple points in time. The timing of a shift in mode can be based on an a prior expectation, such as a mass extinction event, or the model can test multiple points to allow to find time shift point with the highest likelihood.

Models can be fit using `model.test` but it can be more convenient to use `model.test.wrapper`. Here we will compare the relative fit of Brownian motion, Trend, Ornstein-Uhlenbeck and a multi-mode Ornstein Uhlenbck model in which the optima changes at 66 million years ago, the Cretaceous-Palaeogene boundary.

For example, we could be testing the hypothesis that the extinction of non-avian dinosaurs allowed mammals to go from scurrying in the undergrowth (low optima/low disparity) to dominating all habitats (high optima/high disparity). We will constrain the optima of OU model in the first time begin (i.e, pre-66 Mya) to equal the ancestral value:

```
disp_time <- model.test.wrapper(data = BeckLee_disparity,
                        model = c("BM", "Trend", "OU", "multi.OU"),
                                time.split = 66,
                                pool.variance = NULL,
                                show.p = TRUE,
                                fixed.optima = TRUE)
```

```
## Evidence of equal variance (Bartlett's test of equal variances p = 0).
## Variance is not pooled.
## Running BM model...Done. Log-likelihood = 149.289
## Running Trend model...Done. Log-likelihood = 152.116
## Running OU model...Done. Log-likelihood = 149.289
## Running multi.OU model...Done. Log-likelihood = 152.116
```

```
disp_time
```

```
##          aicc delta_aicc weight_aicc log.lik param ancestral state
## Trend    -298      0.000       0.636   152.1     3          3.255
## multi.OU -296      2.139       0.218   152.1     4          3.253
```

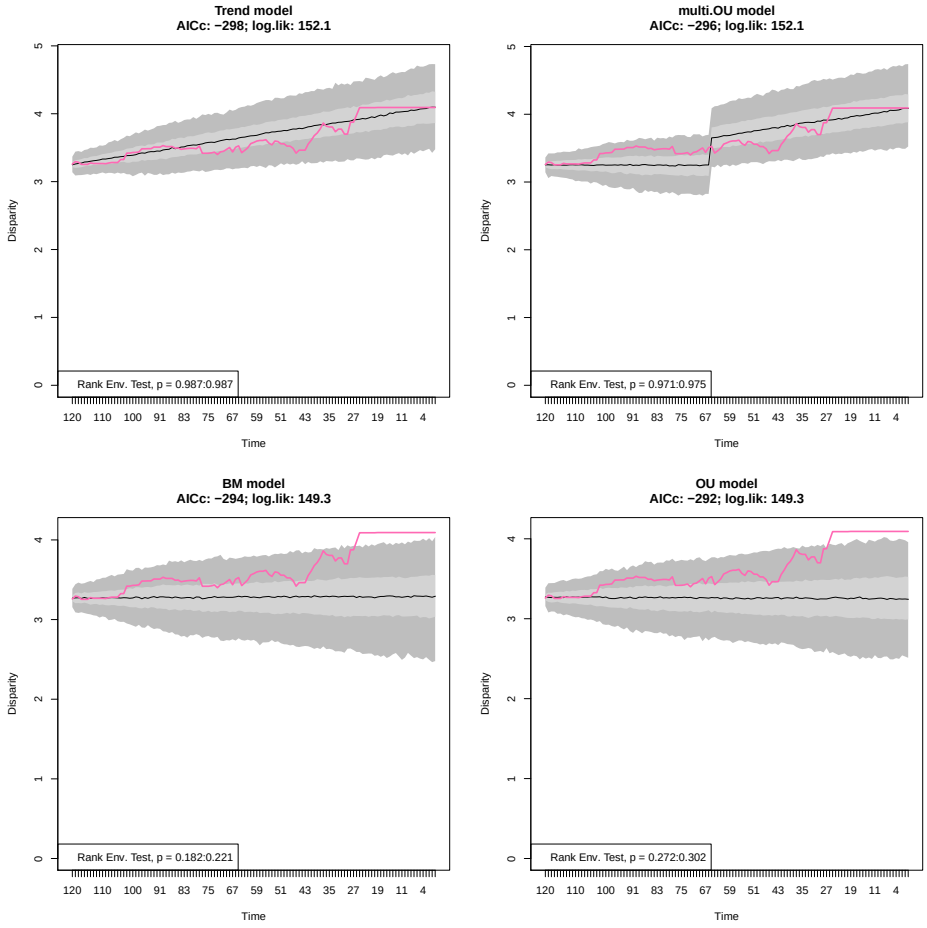



Figure 4.6: Empirical disparity through time (pink), simulate data based on estimated model parameters (grey), delta AICc, and range of p values from the Rank Envelope test for BM, Trend, OU, and multi OU models with a shift in optima allowed at 66 Ma

```
## BM       -294      3.550       0.108   149.3     2          3.267
## OU       -292      5.654       0.038   149.3     3          3.267
##       sigma squared trend alpha optima.2 median p value lower p value
## Trend         0.001 0.007    NA      NA     0.9870130    0.9870130
## multi.OU      0.001    NA 0.002   7.899     0.9730270    0.9710290
## BM            0.001    NA    NA      NA     0.2012987    0.1818182
## OU            0.001    NA 0.000      NA     0.2867133    0.2717283
##          upper p value
## Trend        0.9870130
## multi.OU     0.9750250
## BM           0.2207792
## OU           0.3016983
```

The multi-OU model shows an increase an optima at the Cretaceous-Palaeogene boundary, indicating a shift in disparity. However, this model does not fit as well as a model in which there is an increasing trend through time. We can also fit a model in which the we specify a heterogeneous model but we do not give a `time.split`. In this instance the model will test all splits that have at least 10 time slices on either side of the split. That's 102 potential time shifts in this example dataset so be warned, the following code will estimate 105 models!

```
## An example of a time split model in which all potential splits are tested
## WARNING: this will take between 20 minutes and half and hour to run!
disp_time <- model.test.wrapper(data = BeckLee_disparity,
                        model = c("BM", "Trend", "OU", "multi.OU"),
                                show.p = TRUE, fixed.optima = TRUE)
```

As well as specifying a multi-OU model we can run any combination of models. For example we could fit a model at the Cretaceous-Palaeogene boundary that goes from an OU to a BM model, a Trend to an OU model, a Stasis to a Trend model or any combination you want to use. The only model that can't be used in combination is a multi-OU model.

These can be introduced by changing the input for the models into a list, and supplying a vector with the two models. This is easier to see with an example:

```
## The models to test
my_models <- list(c("BM", "OU"),
                  c("Stasis", "OU"),
                  c("BM", "Stasis"),
                  c("OU", "Trend"),
                  c("Stasis", "BM"))

## Testing the models
disp_time <- model.test.wrapper(data = BeckLee_disparity,
                                model = my_models, time.split = 66,
                                show.p = TRUE, fixed.optima = TRUE)
```

```
## Evidence of equal variance (Bartlett's test of equal variances p = 0).
## Variance is not pooled.
## Running BM:OU model...Done. Log-likelihood = 144.102
## Running Stasis:OU model...Done. Log-likelihood = 125.066
## Running BM:Stasis model...Done. Log-likelihood = 69.265
## Running OU:Trend model...Done. Log-likelihood = 147.839
## Running Stasis:BM model...Done. Log-likelihood = 125.066
```

```
disp_time
```

```
##         aicc delta_aicc weight_aicc log.lik param ancestral state
## OU:Trend  -287       0.0       0.977   147.8     4          3.352
## BM:OU     -280       7.5       0.023   144.1     4          3.350
## Stasis:BM -244      43.4       0.000   125.1     3             NA
## Stasis:OU -240      47.7       0.000   125.1     5             NA
## BM:Stasis -130     157.1       0.000    69.3     4          3.268
##        sigma squared alpha optima.1 theta.1 omega trend median p value
## OU:Trend       0.001 0.041      NA     NA    NA 0.011     0.3246753
## BM:OU          0.001 0.000   4.092     NA    NA    NA     0.5009990
## Stasis:BM      0.002    NA      NA  3.390 0.004    NA     0.9970030
## Stasis:OU      0.002 0.000   4.092  3.390 0.004    NA     1.0000000
## BM:Stasis      0.000    NA      NA  3.806 0.058    NA     1.0000000
##           lower p value upper p value
## OU:Trend      0.2957043     0.3536464
## BM:OU         0.4885115     0.5134865
## Stasis:BM     0.9970030     0.9970030
## Stasis:OU     1.0000000     1.0000000
## BM:Stasis     1.0000000     1.0000000
```

### 4.7.4 `model.test.sim`

Note that all the models above where run using the `model.test.wrapper` function that is a… wrapping function! In practice, this function runs two main functions from the `dispRity` package and then plots the results:

- `model.test` and
- `model.test.sim`

The `model.test.sim` allows to simulate disparity evolution given a `dispRity` object input (as in `model.test.wrapper`) or given a model and its specification. For example, it is possible to simulate a simple Brownian motion model (or any of the other models or models combination described above):

```
## A simple BM model
model_simulation <- model.test.sim(sim = 1000, model = "BM",
                                   time.span = 50, variance = 0.1,
                                   sample.size = 100,
```

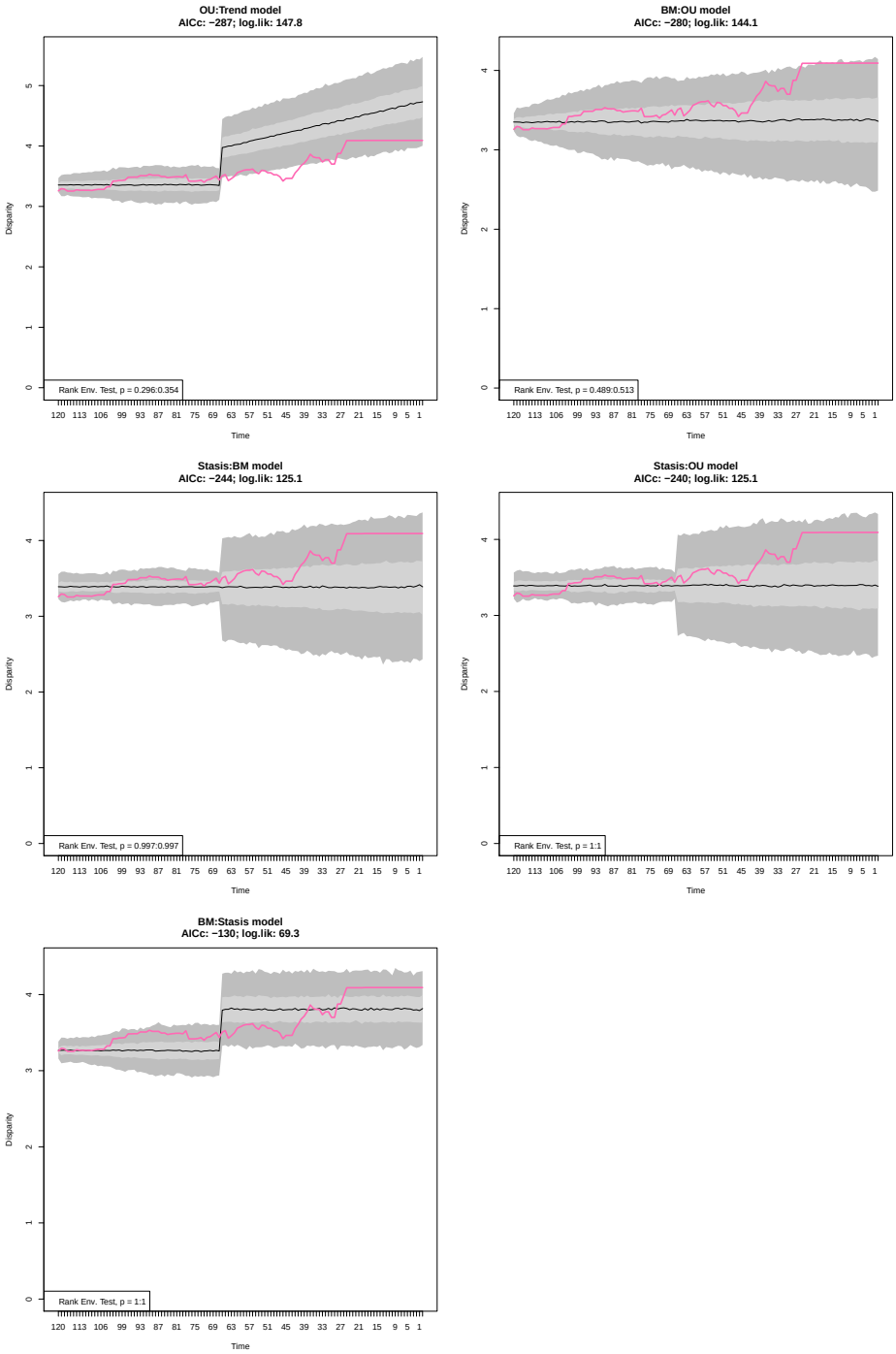



Figure 4.7: Empirical disparity through time (pink), simulate data based on estimated model parameters (grey), delta AICc, and range of p values from the Rank Envelope test for a variety of models with a shift in optima allowed at 66 Ma

```
                                    parameters = list(ancestral.state = 0))
model_simulation
```

```
## Disparity evolution model simulation:
## Call: model.test.sim(sim = 1000, model = "BM", time.span = 50, variance = 0.1, sample.si
##
## Model simulated (1000 times):
## [1] "BM"
```

This will simulate 1000 Brownian motions for 50 units of time with 100 sampled elements, a variance of 0.1 and an ancestral state of 0. We can also pass multiple models in the same way we did it for `model.test` This model can then be summarised and plotted as most `dispRity` objects:

```
## Displaying the 5 first rows of the summary
head(summary(model_simulation))
```

```
##   subsets   n var     median     2.5%       25%      75%    97.5%
## 1      50 100 0.1 -0.06195918 -1.963569 -0.7361336 0.5556715 1.806730
## 2      49 100 0.1 -0.09905061 -2.799025 -1.0670018 0.8836605 2.693583
## 3      48 100 0.1 -0.06215828 -3.594213 -1.3070097 1.1349712 3.272569
## 4      47 100 0.1 -0.10602238 -3.949521 -1.4363010 1.2234625 3.931000
## 5      46 100 0.1 -0.09016928 -4.277897 -1.5791755 1.3889584 4.507491
## 6      45 100 0.1 -0.13183180 -5.115647 -1.7791878 1.6270527 5.144023
```

```
## Plotting the simulations
plot(model_simulation)
```

Note that these functions can take all the arguments that can be passed to `plot`, `summary`, `plot.dispRity` and `summary.dispRity`.

#### 4.7.4.1 Simulating tested models

Maybe more interestingly though, it is possible to pass the output of `model.test` directly to `model.test.sim` to simulate the models that fits the data the best and calculate the Rank Envelope test $p$ value. Let's see that using the simple example from the start:

```
## Fitting multiple models on the data set
disp_time <- model.test(data = BeckLee_disparity,
                        model = c("Stasis", "BM", "OU", "Trend", "EB"))
```

```
## Evidence of equal variance (Bartlett's test of equal variances p = 0).
## Variance is not pooled.
## Running Stasis model...Done. Log-likelihood = -18.694
## Running BM model...Done. Log-likelihood = 149.289
## Running OU model...Done. Log-likelihood = 152.119
## Running Trend model...Done. Log-likelihood = 152.116
```

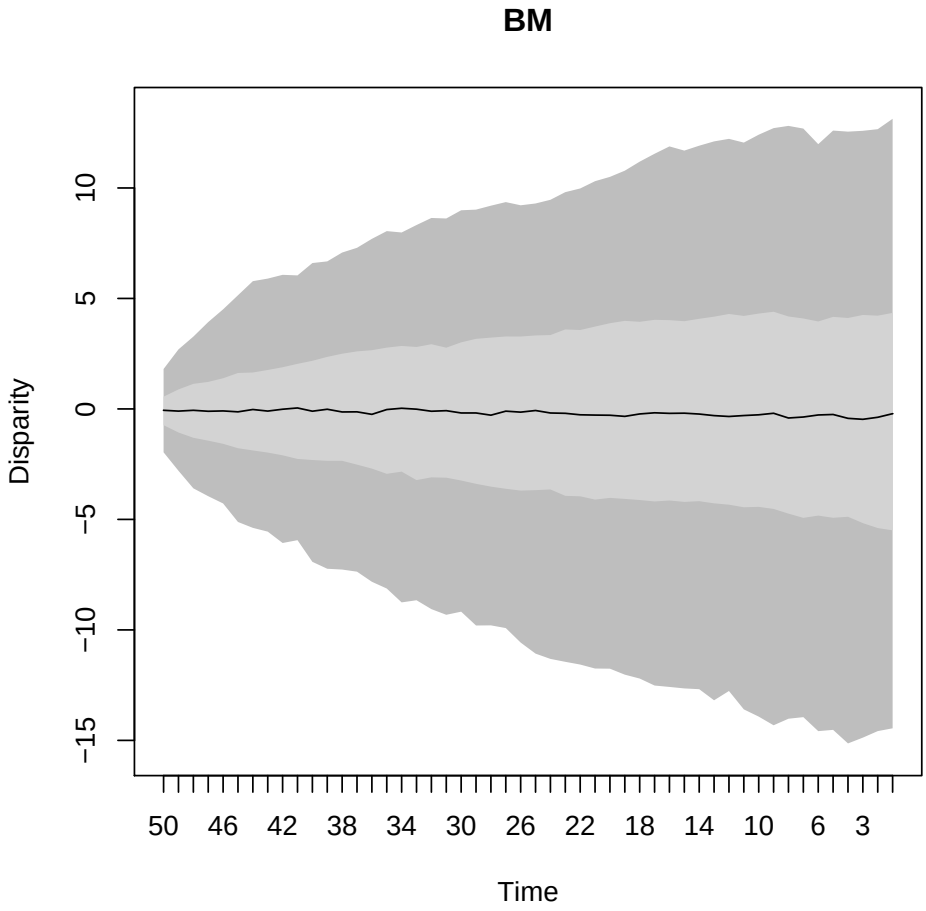



Figure 4.8: A simulated Brownian motion

```
## Running EB model...Done. Log-likelihood = 126.268
```

```
summary(disp_time)
```

```
##      aicc delta_aicc weight_aicc log.lik param theta.1 omega ancestral state
## Stasis  41    339.5     0.000  -18.7    2  3.629 0.074          NA
## BM    -294      3.6     0.112  149.3    2    NA   NA        3.267
## OU    -296      2.1     0.227  152.1    4    NA   NA        3.254
## Trend -298      0.0     0.661  152.1    3    NA   NA        3.255
## EB    -246     51.7     0.000  126.3    3    NA   NA        4.092
##        sigma squared alpha optima.1 trend     eb
## Stasis            NA    NA       NA    NA     NA
## BM             0.001    NA       NA    NA     NA
## OU             0.001 0.001    12.88    NA     NA
## Trend          0.001    NA       NA 0.007     NA
## EB             0.000    NA       NA    NA -0.032
```

As seen before, the Trend model fitted this dataset the best. To simulate what 1000 Trend models would look like using the same parameters as the ones estimated with `model.test` (here the ancestral state being 3.255, the sigma squared being 0.001 and the trend of 0.007), we can simply pass this model to `model.test.sim`:

```
## Simulating 1000 Trend model with the observed parameters
sim_trend <- model.test.sim(sim = 1000, model = disp_time)
sim_trend
```

```
## Disparity evolution model simulation:
## Call: model.test.sim(sim = 1000, model = disp_time)
##
## Model simulated (1000 times):
##       aicc log.lik param ancestral state sigma squared trend
## Trend -298   152.1     3           3.255         0.001 0.007
##
## Rank envelope test
##  p-value of the test: 0.99001 (ties method: midrank)
##  p-interval         : (0.99001, 0.99001)
```

By default, the model simulated is the one with the lowest AICc (`model.rank = 1`) but it is possible to choose any ranked model, for example, the OU (second one):

```
## Simulating 1000 OU model with the observed parameters
sim_OU <- model.test.sim(sim = 1000, model = disp_time,
                         model.rank = 2)
sim_OU
```

```
## Disparity evolution model simulation:
## Call: model.test.sim(sim = 1000, model = disp_time, model.rank = 2)
```

```
##
## Model simulated (1000 times):
##   aicc log.lik param ancestral state sigma squared alpha optima.1
## OU -296   152.1     4           3.254         0.001 0.001    12.88
##
## Rank envelope test
##  p-value of the test: 0.9915085 (ties method: midrank)
##  p-interval         : (0.991009, 0.992008)
```

And as the example above, the simulated data can be plotted or summarised:

```
head(summary(sim_trend))
```

```
##   subsets n       var   median     2.5%      25%      75%    97.5%
## 1    120 5 0.01723152 3.255121 3.135057 3.219150 3.293407 3.375118
## 2    119 5 0.03555816 3.265538 3.093355 3.200493 3.323520 3.440795
## 3    118 6 0.03833089 3.269497 3.090438 3.212015 3.329629 3.443074
## 4    117 7 0.03264826 3.279180 3.112205 3.224810 3.336801 3.447997
## 5    116 7 0.03264826 3.284500 3.114788 3.223247 3.347970 3.463631
## 6    115 7 0.03264826 3.293918 3.101298 3.231659 3.354321 3.474645
```

```
head(summary(sim_OU))
```

```
##   subsets n       var   median     2.5%      25%      75%    97.5%
## 1    120 5 0.01723152 3.253446 3.141550 3.212259 3.293839 3.371701
## 2    119 5 0.03555816 3.263230 3.083542 3.197505 3.324500 3.440508
## 3    118 6 0.03833089 3.262999 3.101401 3.203909 3.332642 3.440208
## 4    117 7 0.03264826 3.272600 3.104511 3.214542 3.330617 3.442819
## 5    116 7 0.03264826 3.280440 3.100239 3.219782 3.342742 3.475893
## 6    115 7 0.03264826 3.287360 3.094703 3.222526 3.355281 3.477519
```

```
## The trend model with some graphical options
plot(sim_trend, xlab = "Time (Mya)", ylab = "sum of variances",
    col = c("#F65205", "#F38336", "#F7B27E"))

## Adding the observed disparity through time
plot(BeckLee_disparity, add = TRUE, col = c("#3E9CBA", "#98D4CF90", "#BFE4E390"))
```

## 4.8 Disparity as a distribution

Disparity is often regarded as a summary value of the position of the all elements in the ordinated space. For example, the sum of variances, the product of ranges or the median distance between the elements and their centroid will summarise disparity as a single value. This value can be pseudo-replicated (bootstrapped) to obtain a distribution of the summary metric with estimated error. However, another way to perform disparity analysis is to use the *whole distribution* rather than just a summary metric (e.g. the variances or the ranges).

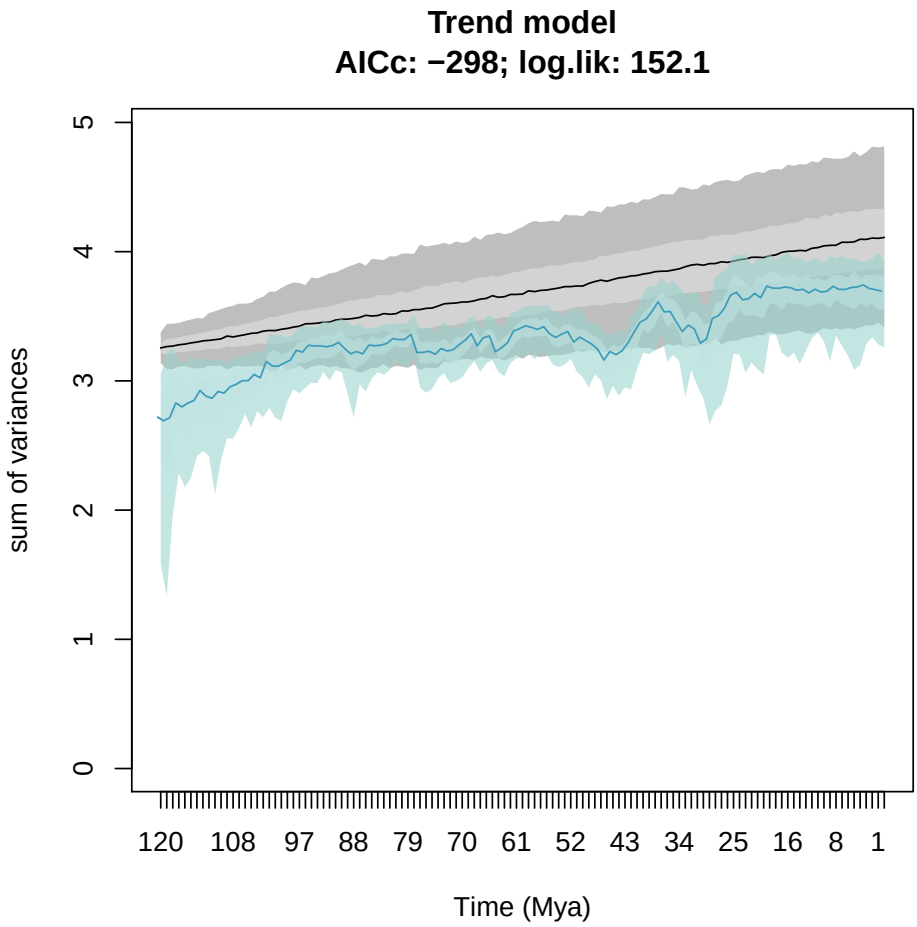



Figure 4.9: The best fitted model (Trend) and the observed disparity through time

This is possible in the `dispRity` package by calculating disparity as a dimension-level 2 metric only! Let's have a look using our previous example of bootstrapped time slices but by measuring the distances between each taxon and their centroid as disparity.

```
## Measuring disparity as a whole distribution
disparity_centroids <- dispRity(boot_time_slices,
                                metric = centroids)
```

The resulting disparity object is of dimension-level 2, so it can easily be transformed into a dimension-level 1 object by, for example, measuring the median distance of all these distributions:

```
## Measuring median disparity in each time slice
disparity_centroids_median <- dispRity(disparity_centroids,
                                       metric = median)
```

And we can now compare the differences between these methods:

```
## Summarising both disparity measurements:
## The distributions:
summary(disparity_centroids)
```

```
##   subsets  n obs.median bs.median  2.5%   25%   75% 97.5%
## 1     120  5      1.605     1.369 0.849 1.208 1.662 1.915
## 2      80 19      1.834     1.776 1.527 1.687 1.848 1.967
## 3      40 15      1.804     1.782 1.403 1.682 1.895 2.096
## 4       0 10      1.911     1.823 1.376 1.711 1.965 2.104
```

```
## The summary of the distributions (as median)
summary(disparity_centroids_median)
```

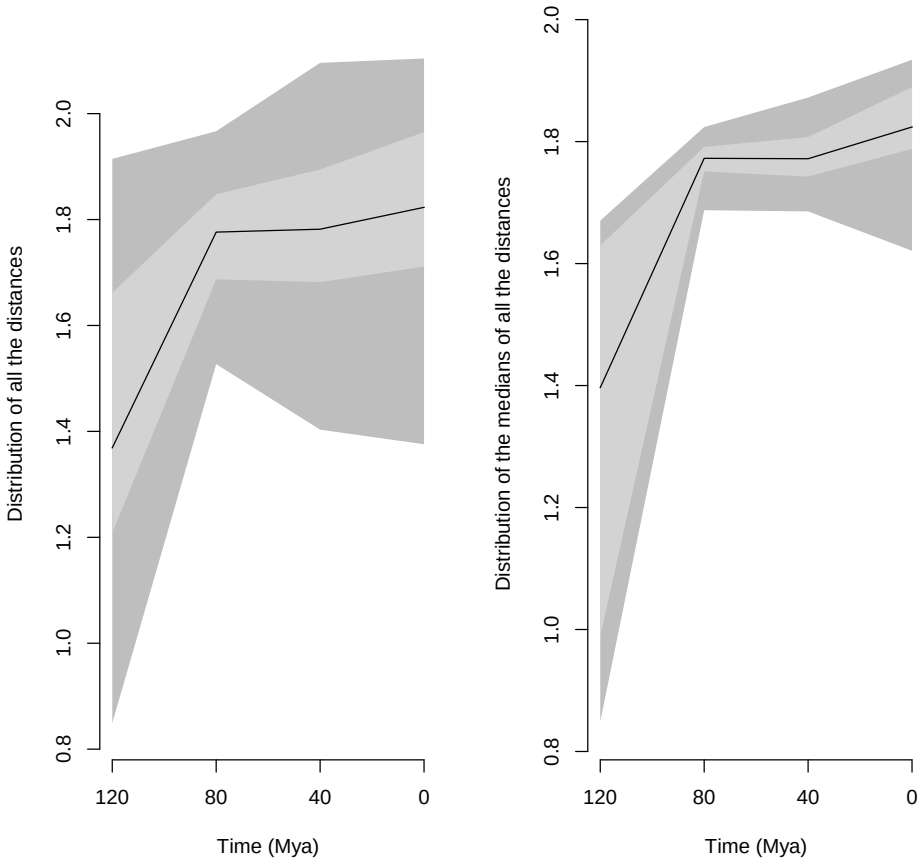
```
##   subsets  n   obs bs.median  2.5%   25%   75% 97.5%
## 1     120  5 1.605     1.396 0.849 0.989 1.630 1.670
## 2      80 19 1.834     1.773 1.688 1.751 1.791 1.824
## 3      40 15 1.804     1.772 1.685 1.743 1.807 1.872
## 4       0 10 1.911     1.824 1.621 1.788 1.889 1.934
```

We can see that the summary message for the distribution is slightly different than before. Here `summary` also displays the observed central tendency (i.e. the central tendency of the measured distributions). Note that, as expected, this central tendency is the same in both metrics!

Another, maybe more intuitive way, to compare both approaches for measuring disparity is to plot the distributions:

```
## Graphical parameters
op <- par(bty = "n", mfrow = c(1, 2))

## Plotting both disparity measurements
plot(disparity_centroids,
```

```
     ylab = "Distribution of all the distances")
plot(disparity_centroids_median,
     ylab = "Distribution of the medians of all the distances")
```




```
par(op)
```

We can then test for differences in the resulting distributions using `test.dispRity` and the `bhatt.coeff` test as described above.

```
## Probability of overlap in the distribution of medians
test.dispRity(disparity_centroids_median, test = bhatt.coeff)
```

```
## Warning in test.dispRity(disparity_centroids_median, test = bhatt.coeff): Multiple p-v
## This can inflate Type I error!
```

```
##          bhatt.coeff
## 120 : 80   0.1643168
## 120 : 40   0.1519868
## 120 : 0    0.2271532
## 80 : 40    0.8809326
```

```
## 80 : 0     0.5698764
## 40 : 0     0.7767260
```

In this case, we are looking at the probability of overlap of the distribution of median distances from centroids among each pair of time slices. In other words, we are measuring whether the medians from each bootstrap pseudo-replicate for each time slice overlap. But of course, we might be interested in the actual distribution of the distances from the centroid rather than simply their central tendencies. This can be problematic depending on the research question asked since we are effectively comparing non-independent medians distributions (because of the pseudo-replication).

One solution, therefore, is to look at the full distribution:

```
## Probability of overlap for the full distributions
test.dispRity(disparity_centroids, test = bhatt.coeff)
```

```
## Warning in test.dispRity(disparity_centroids, test = bhatt.coeff): Multiple p-values will be calcu
## This can inflate Type I error!
```

```
##          bhatt.coeff
## 120 : 80   0.5899902
## 120 : 40   0.6436707
## 120 : 0    0.6271424
## 80 : 40    0.9119738
## 80 : 0     0.8444125
## 40 : 0     0.9531318
```

These results show the actual overlap among all the measured distances from centroids concatenated across all the bootstraps. For example, when comparing the slices 120 and 80, we are effectively comparing the $5 \times 100$ distances (the distances of the five elements in slice 120 bootstrapped 100 times) to the $19 \times 100$ distances from slice 80. However, this can also be problematic for some specific tests since the $n \times 100$ distances are also pseudo-replicates and thus are still not independent.

A second solution is to compare the distributions to each other *for each replicate*:

```
## Boostrapped probability of overlap for the full distributions
test.dispRity(disparity_centroids, test = bhatt.coeff,
              concatenate = FALSE)
```

```
## Warning in test.dispRity(disparity_centroids, test = bhatt.coeff, concatenate = FALSE): Multiple p
## This can inflate Type I error!
```

```
##          bhatt.coeff      2.5%       25%       75%     97.5%
## 120 : 80   0.2582035 0.0000000 0.1695523 0.3745110 0.5652651
## 120 : 40   0.2985743 0.0000000 0.2000000 0.4499599 0.6444474
## 120 : 0    0.2527329 0.0000000 0.1414214 0.4000000 0.6007397
## 80 : 40    0.5762757 0.2036442 0.4400379 0.6924779 0.8713920
## 80 : 0     0.4579673 0.1227841 0.3487172 0.6082889 0.7583736
```

```
## 40 : 0     0.5616576 0.2449490 0.4420687 0.6838864 0.8757140
```

These results show the median overlap among pairs of distributions in the first column (`bhatt.coeff`) and then the distribution of these overlaps among each pair of bootstraps. In other words, when two distributions are compared, they are now compared for each bootstrap pseudo-replicate, thus effectively creating a distribution of probabilities of overlap. For example, when comparing the slices 120 and 80, we have a mean probability of overlap of 0.28 and a probability between 0.18 and 0.43 in 50% of the pseudo-replicates. Note that the quantiles and central tendencies can be modified via the `conc.quantiles` option.

## 4.9 Disparity from other matrices

In the example so far, disparity was measured from an ordinated multidimensional space (i.e. a PCO of the distances between taxa based on discrete morphological characters). This is a common approach in palaeobiology, morphometrics or ecology but ordinated matrices are not mandatory for the `dispRity` package! It is totally possible to perform the same analysis detailed above using other types of matrices as long as your elements are rows in your matrix.

For example, we can use the data set `eurodist`, an `R` inbuilt dataset that contains the distances (in km) between European cities. We can check for example, if Northern European cities are closer to each other than Southern ones:

```
## Making the eurodist data set into a matrix (rather than "dist" object)
eurodist <- as.matrix(eurodist)
eurodist[1:5, 1:5]
```

```
##           Athens Barcelona Brussels Calais Cherbourg
## Athens         0      3313     2963   3175      3339
## Barcelona   3313         0     1318   1326      1294
## Brussels    2963      1318        0    204       583
## Calais      3175      1326      204      0       460
## Cherbourg   3339      1294      583    460         0
```

```
## The two groups of cities
Northern <- c("Brussels", "Calais", "Cherbourg", "Cologne", "Copenhagen",
              "Hamburg", "Hook of Holland", "Paris", "Stockholm")
Southern <- c("Athens", "Barcelona", "Geneva", "Gibraltar", "Lisbon", "Lyons",
              "Madrid", "Marseilles", "Milan", "Munich", "Rome", "Vienna")

## Creating the subset dispRity object
eurodist_subsets <- custom.subsets(eurodist, group = list("Northern" = Northern,
                                                          "Southern" = Southern))
```

```
## Warning: custom.subsets is applied on what seems to be a distance matrix.
## The resulting matrices won't be distance matrices anymore!
```

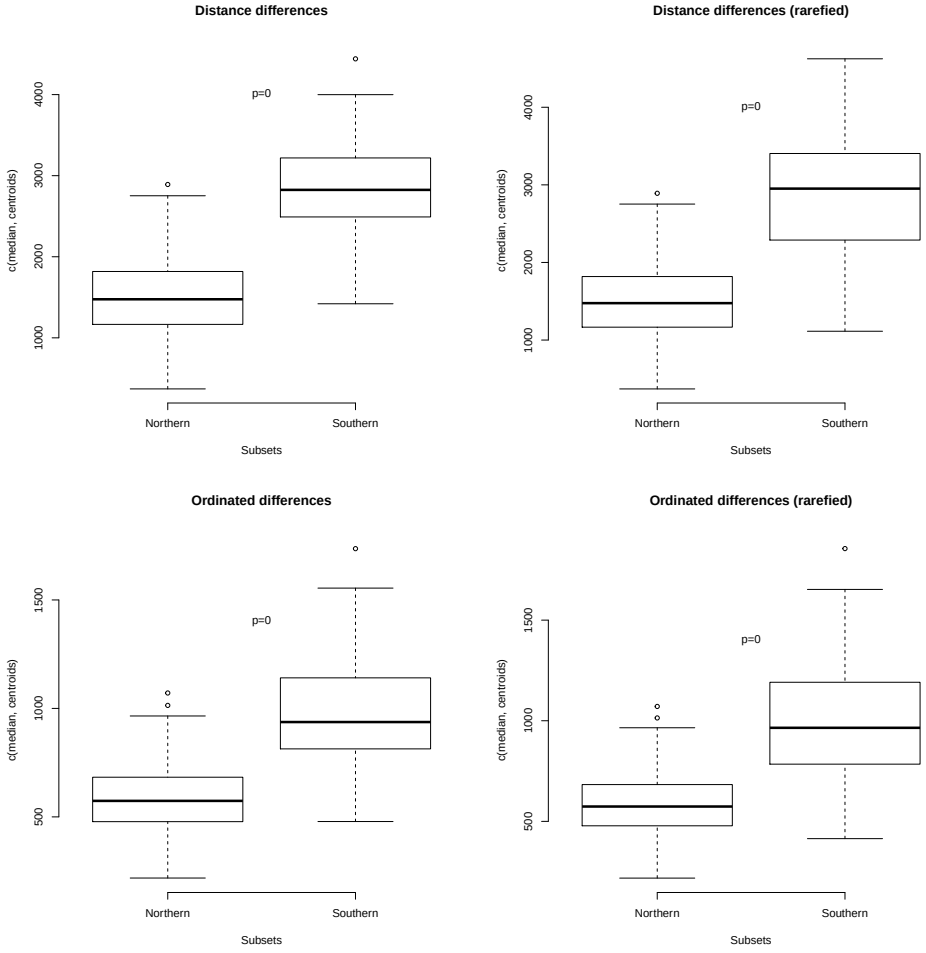
```
## Bootstrapping and rarefying to 9 elements (the number of Northern cities)
eurodist_bs <- boot.matrix(eurodist_subsets, rarefaction = 9)

## Measuring disparity as the median distance from group's centroid
euro_disp <- dispRity(eurodist_bs, metric = c(median, centroids))

## Testing the differences using a simple wilcox.test
euro_diff <- test.dispRity(euro_disp, test = wilcox.test)
euro_diff_rar <- test.dispRity(euro_disp, test = wilcox.test, rarefaction = 9)
```

We can compare this approach to an ordination one:

```
## Ordinating the eurodist matrix (with 11 dimensions)
euro_ord <- cmdscale(eurodist, k = 11)

## Calculating disparity on the bootstrapped and rarefied subset data
euro_ord_disp <- dispRity(boot.matrix(custom.subsets(euro_ord, group =
        list("Northern" = Northern, "Southern" = Southern)), rarefaction = 9),
        metric = c(median, centroids))

## Testing the differences using a simple wilcox.test
euro_ord_diff <- test.dispRity(euro_ord_disp, test = wilcox.test)
euro_ord_diff_rar <- test.dispRity(euro_ord_disp, test = wilcox.test, rarefaction = 9)
```

And visualise the differences:

```
## Plotting the differences
par(mfrow = c(2,2), bty = "n")
## Plotting the normal disparity
plot(euro_disp, main = "Distance differences")
## Adding the p-value
text(1.5, 4000, paste0("p=",round(euro_diff[[2]][[1]], digit = 5)))
## Plotting the rarefied disparity
plot(euro_disp, rarefaction = 9, main = "Distance differences (rarefied)")
## Adding the p-value
text(1.5, 4000, paste0("p=",round(euro_diff_rar[[2]][[1]], digit = 5)))

## Plotting the ordinated disparity
plot(euro_ord_disp, main = "Ordinated differences")
## Adding the p-value
text(1.5, 1400, paste0("p=",round(euro_ord_diff[[2]][[1]], digit = 5) ))
## Plotting the rarefied disparity
plot(euro_ord_disp, rarefaction = 9, main = "Ordinated differences (rarefied)")
## Adding the p-value
text(1.5, 1400, paste0("p=",round(euro_ord_diff_rar[[2]][[1]], digit = 5) ))
```




As expected, the results are pretty similar in pattern but different in terms of scale. The median centroids distance is expressed in km in the "Distance differences" plots and in Euclidean units of variation in the "Ordinated differences" plots.

## 4.10 Disparity from multiple matrices (and multiple trees!)

Since the version `1.4` of this package, it is possible to use multiple trees and multiple matrices in `dispRity` objects. To use multiple matrices, this is rather easy: just supply a list of matrices to any of the `dispRity` functions and, as long as they have the same size and the same rownames they will be handled as a distribution of matrices.

```
set.seed(1)
## Creating 3 matrices with 4 dimensions and 10 elements each (called t1, t2, t3, etc...)
matrix_list <- replicate(3, matrix(rnorm(40), 10, 4, dimnames = list(paste0("t", 1:10))),
                         simplify = FALSE)
class(matrix_list) # This is a list of matrices
```

```
## [1] "list"
```

```
## Measuring some disparity metric on one of the matrices
summary(dispRity(matrix_list[[1]], metric = c(sum, variances)))
```

```
##   subsets  n  obs
## 1       1 10 3.32
```

```
## Measuring the same disparity metric on the three matrices
summary(dispRity(matrix_list, metric = c(sum, variances)))
```

```
##   subsets  n obs.median  2.5%   25%   75% 97.5%
## 1       1 10       3.32 3.044 3.175 3.381 3.435
```

As you can see, when measuring the sum of variances on multiple matrices, we now have a distribution of sum of variances rather than a single observed value.

Similarly as running disparity analysis using multiple matrices, you can run the `chrono.subsets` function using multiple trees. This can be useful if you want to use a tree posterior distribution rather than a single consensus tree. These trees can be passed to `chrono.subsets` as a `"multiPhylo"` object (with the same node and tip labels in each tree). First let's define a function to generate multiple trees with the same labels and root ages:

```
set.seed(1)
## Matches the trees and the matrices
## A bunch of trees
make.tree <- function(n, fun = rtree) {
    ## Make the tree
    tree <- fun(n)
    tree <- chronos(tree, quiet = TRUE,
                    calibration = makeChronosCalib(tree, age.min = 10, age.max = 10))
    class(tree) <- "phylo"
    ## Add the node labels
    tree$node.label <- paste0("n", 1:Nnode(tree))
    ## Add the root time
    tree$root.time <- max(tree.age(tree)$ages)
    return(tree)
}
trees <- replicate(3, make.tree(10), simplify = FALSE)
```

```
## Warning: false convergence (8)
```

```
class(trees) <- "multiPhylo"
trees
```

```
## 3 phylogenetic trees
```

We can now simulate some ancestral states for the matrices in the example above to have multiple matrices associated with the multiple trees.

```
## A function for running the ancestral states estimations
do.ace <- function(tree, matrix) {
    ## Run one ace
    fun.ace <- function(character, tree) {
        results <- ace(character, phy = tree)$ace
        names(results) <- paste0("n", 1:Nnode(tree))
        return(results)
    }
    ## Run all ace
    return(rbind(matrix, apply(matrix, 2, fun.ace, tree = tree)))
}

## All matrices
matrices <- mapply(do.ace, trees, matrix_list, SIMPLIFY = FALSE)
```

Let's first see an example of time-slicing with one matrix and multiple trees. This assumes that your tip values (observed) and node values (estimated) are fixed with no error on them. It also assumes that the nodes in the matrix always corresponds to the node in the trees (in other words, the tree topologies are fixed):

```
## Making three "proximity" time slices across one tree
one_tree <- chrono.subsets(matrices[[1]], trees[[1]],
                           method = "continuous",
                           model = "proximity", time = 3)
## Making three "proximity" time slices across the three trees
three_tree <- chrono.subsets(matrices[[1]], trees,
                             method = "continuous",
                             model = "proximity", time = 3)
## Measuring disparity as the sum of variances and summarising it
summary(dispRity(one_tree, metric = c(sum, variances)))
```

```
##   subsets  n   obs
## 1     8.3  3 0.080
## 2    4.15  5 2.903
## 3       0 10 3.320
```

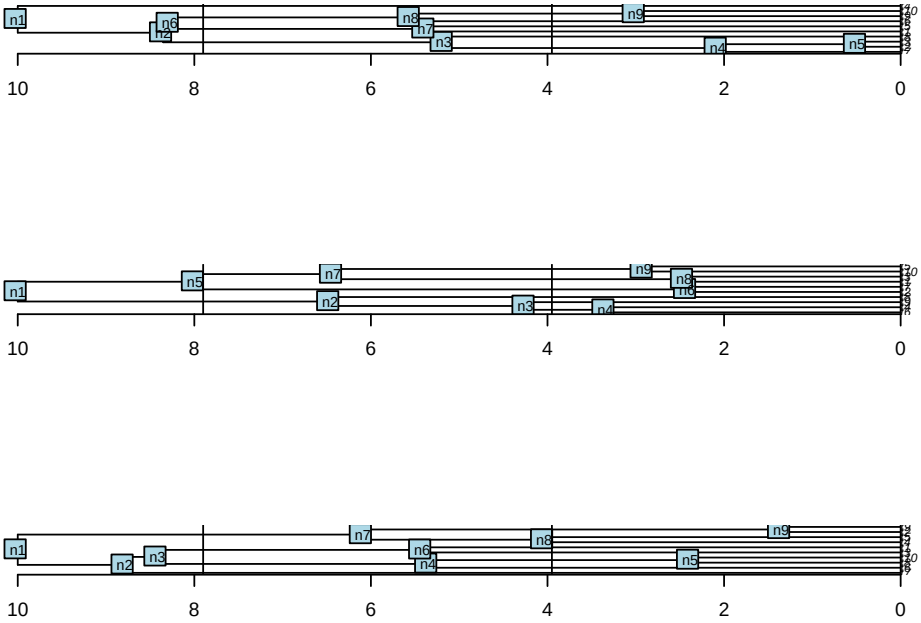
```
summary(dispRity(three_tree, metric = c(sum, variances)))
```

```
##   subsets  n obs.median  2.5%   25%   75% 97.5%
## 1     7.9  3      0.250 0.088 0.165 0.307 0.360
```

```
## 2    3.95  5      0.256 0.133 0.191 1.579 2.771
## 3       0 10      3.320 3.320 3.320 3.320 3.320
```

This results show the effect of considering a tree distribution: in the first case (`one_tree`) the time slice at 3.95 Mya has a sum of variances of 2.9 but this values goes down to 0.256 in the second case (`three_tree`) which is due to the differences in branch lengths distributions:

```
par(mfrow = c(3,1))
slices <- c(7.9, 3.95, 0)
fun.plot <- function(tree) {
  plot(tree)
  nodelabels(tree$node.label, cex = 0.8)
  axisPhylo()
  abline(v = tree$root.time - slices)
}
silent <- lapply(trees, fun.plot)
```




Note that in this example, the nodes are actually even different in each tree! The node `n4` for example, is not direct descendent of `t4` and `t6` in all trees! To fix that, it is possible to input a list of trees and a list of matrices that correspond to each tree in `chrono.subsets` by using the `bind.data = TRUE` option. In this case, the matrices need to all have the same row names and the trees all need the same labels as before:

```
## Making three "proximity" time slices across three trees and three bound matrices
bound_data <- chrono.subsets(matrices, trees,
                             method = "continuous",
```

```
                                model = "proximity",
                                time = 3,
                                bind.data = TRUE)
## Making three "proximity" time slices across three trees and three matrices
unbound_data <- chrono.subsets(matrices, trees,
                                  method = "continuous",
                                  model = "proximity",
                                  time = 3,
                                  bind.data = FALSE)

## Measuring disparity as the sum of variances and summarising it
summary(dispRity(bound_data, metric = c(sum, variances)))
```

```
##   subsets  n obs.median  2.5%   25%   75% 97.5%
## 1     7.9  3       0.08 0.076 0.078 0.273 0.447
## 2    3.95  5       1.79 0.353 1.034 2.347 2.848
## 3       0 10       3.32 3.044 3.175 3.381 3.435
```

```
summary(dispRity(unbound_data, metric = c(sum, variances)))
```

```
##   subsets  n obs.median 2.5%  25%  75% 97.5%
## 1     7.9  3       0.79 0.48 0.63 0.83  0.85
## 2    3.95  5       3.25 1.36 2.25 3.94  4.56
## 3       0 10       9.79 9.79 9.79 9.79  9.79
```

Note here that the results are again rather different: with the bound data, the slices are done across the three trees and each of their corresponding matrix (resulting in three observation) which is more accurate than the previous results from `three_trees` above. With the unbound data, the slices are done across the three trees and applied to the three matrices (resulting in 9 observations). As we've seen before, this is incorrect in this case since the trees don't have the same topology (so the nodes selected by a slice through the second tree are not equivalent to the nodes in the first matrix) but it can be useful if the topology is fixed to integrate both uncertainty in branch length (slicing through different trees) and uncertainty from, say, ancestral states estimations (applying the slices on different matrices).

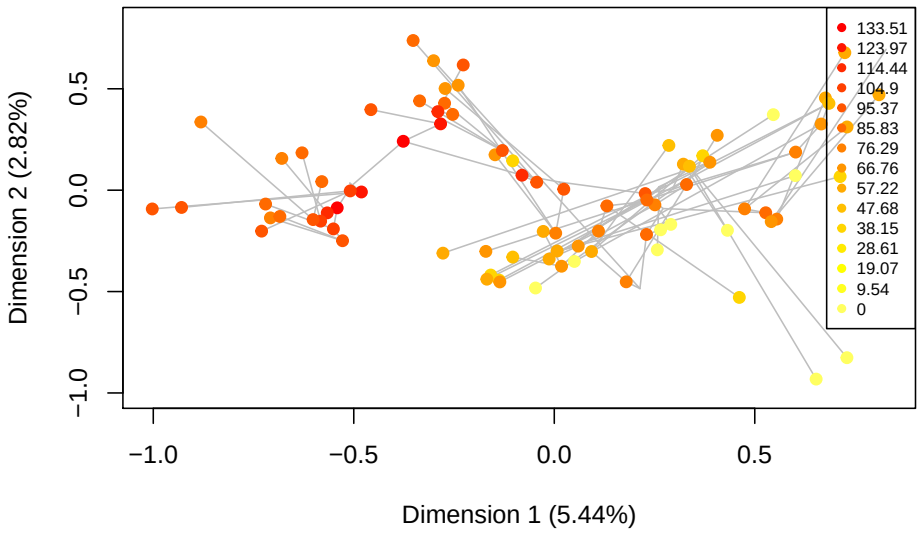
## 4.11 Disparity with trees: *dispRitree!*

Since the package's version 1.5.10, trees can be directly attached to `dispRity` objects. This allows any function in the package that has an input argument called "`tree`" to automatically intake the tree from the `dispRity` object. This is especially useful for disparity metrics that requires calculations based on a phylogenetic tree (e.g. `ancestral.dist` or `projections.tree`) and if phylogeny (or phylogenie*s*) are going to be an important part of your analyses.

Trees are attached to `dispRity` object as soon as they are called in any function of the package (e.g. as an argument in `chrono.subsets` or in `dispRity`) and are stored in `my_dispRity_object$tree`. You can always manually attach, detach or modify the tree parts of a `dispRity` object using the utility functions `get.tree` (to access the trees), `remove.tree` (to remove it) and `add.tree` (to… add trees!). The only requirement for this to work is that the labels in the tree must match the ones in the data. If the tree has node labels, their node labels must also match the data. Similarly if the data has entries for node labels, they must be present in the tree.

Here is a quick demo on how attaching trees to `dispRity` objects can work and make your life easy: for example here we will measure how the sum of branch length changes through time when time slicing through some demo data with a acctran split time slice model (see more info here).

```
## Loading some demo data:
## An ordinated matrix with node and tip labels
data(BeckLee_mat99)
## The corresponding tree with tip and node labels
data(BeckLee_tree)
## A list of tips ages for the fossil data
data(BeckLee_ages)

## Time slicing through the tree using the equal split algorithm
time_slices <- chrono.subsets(data   = BeckLee_mat99,
                              tree   = BeckLee_tree,
                              FADLAD = BeckLee_ages,
                              method = "continuous",
                              model  = "acctran",
                              time   = 15)

## We can visualise the resulting trait space with the phylogeny
## (using the specific argument as follows)
plot(time_slices, type = "preview",
     specific.args = list(tree = TRUE))
```



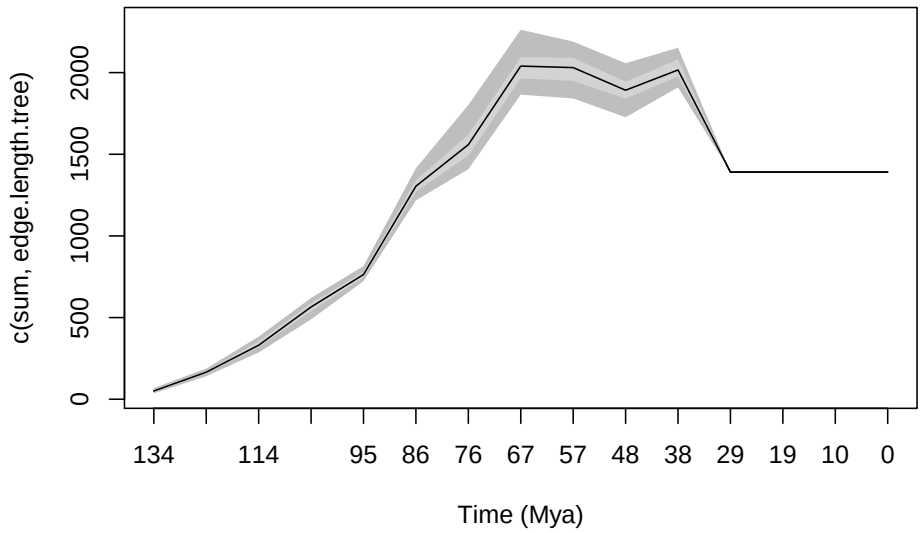

```
## Note that some nodes are never selected thus explaining the branches not reaching t
```

And we can then measure disparity as the sum of the edge length at each time slice on the bootstrapped data:

```
## Measuring the sum of the edge length per slice
sum_edge_length <- dispRity(boot.matrix(time_slices), metric = c(sum, edge.length.tree
## Summarising and plotting
summary(sum_edge_length)
```

```
##    subsets  n  obs bs.median 2.5%  25%  75% 97.5%
## 1   133.51  3   51        51   36   40   61    69
## 2   123.97  6  163       166  141  158  172   188
## 3   114.44  9  332       331  287  317  354   383
## 4    104.9 12  558       565  489  540  587   620
## 5    95.37 15  762       763  723  745  782   815
## 6    85.83 20 1303      1305 1218 1271 1342  1415
## 7    76.29 19 1565      1559 1408 1491 1620  1802
## 8    66.76 23 2055      2040 1865 1965 2095  2262
## 9    57.22 20 2029      2031 1842 1949 2091  2190
## 10   47.68 16 1908      1892 1727 1840 1945  2057
## 11   38.15 16 2017      2016 1910 1975 2081  2152
## 12   28.61 10 1391      1391 1391 1391 1391  1391
## 13   19.07 10 1391      1391 1391 1391 1391  1391
## 14    9.54 10 1391      1391 1391 1391 1391  1391
## 15       0 10 1391      1391 1391 1391 1391  1391
```

```
plot(sum_edge_length)
```




Of course this can be done with multiple trees and be combined with an approach using multiple matrices (see here)!

# Chapter 5

# Making stuff up!

The `dispRity` package also offers some advanced data simulation features to allow to test hypothesis, explore ordinate-spaces or metrics properties or simply playing around with data! All the following functions are based on the same modular architecture of the package and therefore can be used with most of the functions of the package.

## 5.1 Simulating discrete morphological data

The function `sim.morpho` allows to simulate discrete morphological data matrices (sometimes referred to as "cladistic" matrices). It allows to evolve multiple discrete characters on a given phylogenetic trees, given different models, rates, and states. It even allows to include "proper" inapplicable data to make datasets as messy as in real life!

In brief, the function `sim.morpho` takes a phylogenetic tree, the number of required characters, the evolutionary model, and a function from which to draw the rates. The package also contains a function for quickly checking the matrix's phylogenetic signal (as defined in systematics not phylogenetic comparative methods) using parsimony. The methods are described in details below

```
set.seed(3)
## Simulating a starting tree with 15 taxa as a random coalescent tree
my_tree <- rcoal(15)

## Generating a matrix with 100 characters (85% binary and 15% three state) and
## an equal rates model with a gamma rate distribution (0.5, 1) with no
## invariant characters.
my_matrix <- sim.morpho(tree = my_tree, characters = 100, states = c(0.85,
    0.15), rates = c(rgamma, 0.5, 1), invariant = FALSE)
```

```
## The first few lines of the matrix
my_matrix[1:5, 1:10]
```

```
##     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## t10 "1"  "0"  "1"  "0"  "1"  "0"  "0"  "1"  "0"  "0"
## t1  "0"  "0"  "1"  "0"  "0"  "0"  "0"  "1"  "0"  "1"
## t9  "0"  "0"  "1"  "0"  "0"  "0"  "0"  "1"  "0"  "1"
## t14 "1"  "0"  "1"  "0"  "0"  "0"  "0"  "1"  "0"  "1"
## t13 "1"  "0"  "1"  "0"  "0"  "0"  "0"  "1"  "0"  "1"
```

```
## Checking the matrix properties with a quick Maximum Parsimony tree search
check.morpho(my_matrix, my_tree)
```

```
##
## Maximum parsimony        144.000000
## Consistency index          0.750000
## Retention index            0.918552
## Robinson-Foulds distance   4.000000
```

Note that this example produces a tree with a great consistency index and an identical topology to the random coalescent tree! Nearly too good to be true…

### 5.1.1 A more detailed description

The protocol implemented here to generate discrete morphological matrices is based on the ones developed in [Guillerme and Cooper, 2016, O'Reilly et al., 2016, Puttick et al., 2017, E. et al.].

- The first `tree` argument will be the tree on which to "evolve" the characters and therefore requires branch length. You can generate quick and easy random Yule trees using `ape::rtree(number_of_taxa)` but I would advise to use more realistic trees for more realistic simulations based on more realistic models (really realistic then) using the function `tree.bd` from the `diversitree` package [FitzJohn, 2012].
- The second argument, `character` is the number of characters. Pretty straight forward.
- The third, `states` is the proportion of characters states above two (yes, the minimum number of states is two). This argument intakes the proportion of $n$-states characters, for example `states = c(0.5,0.3,0.2)` will generate 50% of binary-state characters, 30% of three-state characters and 20% of four-state characters. There is no limit in the number of state characters proportion as long as the total makes up 100%.
- The forth, `model` is the evolutionary model for generating the character(s). More about this below.
- The fifth and sixth, `rates` and `substitution` are the model parameters described below as well.

- Finally, the two logical arguments, are self explanatory: `invariant` whether to allow invariant characters (i.e. characters that don't change) and `verbose` whether to print the simulation progress on your console.

#### 5.1.1.1 Available evolutionary models

There are currently three evolutionary models implemented in `sim.morpho` but more will come in the future. Note also that they allow fine tuning parameters making them pretty plastic!

- `"ER"`: this model allows any number of character states and is based on the Mk model [Lewis, 2001]. It assumes a unique overall evolutionary rate equal substitution rate between character states. This model is based on the `ape::rTraitDisc` function.
- `"HKY"`: this is binary state character model based on the molecular HKY model [Hasegawa et al., 1985]. It uses the four molecular states (A,C,G,T) with a unique overall evolutionary rate and a biased substitution rate towards transitions (A <-> G or C <-> T) against transvertions (A <-> C and G <-> T). After evolving the nucleotide, this model transforms them into binary states by converting the purines (A and G) into state 0 and the pyrimidines (C and T) into state 1. This method is based on the `phyclust::seq.gen.HKY` function and was first proposed by O'Reilly et al. [2016].
- `"MIXED"`: this model uses a random (uniform) mix between both the `"ER"` and the `"HKY"` models.

The models can take the following parameters: (1) `rates` is the evolutionary rate (i.e. the rate of changes along a branch: the evolutionary speed) and (2) `substitution` is the frequency of changes between one state or another. For example if a character can have high probability of changing (the *evolutionary* rate) with, each time a change occurs a probability of changing from state $X$ to state $Y$ (the *substitution* rate). Note that in the `"ER"` model, the substitution rate is ignore because… by definition this (substitution) rate is equal!

The parameters arguments `rates` and `substitution` takes a distributions from which to draw the parameters values for each character. For example, if you want an `"HKY"` model with an evolutionary rate (i.e. speed) drawn from a uniform distribution bounded between 0.001 and 0.005, you can define it as `rates = c(runif, min = 0.001, max = 0.005)`, `runif` being the function for random draws from a uniform distribution and `max` and `min` being the distribution parameters. These distributions should always be passed in the format `c(random_distribution_function, distribution_parameters)` with the names of the distribution parameters arguments.

#### 5.1.1.2 Checking the results

An additional function, `check.morpho` runs a quick Maximum Parsimony tree search using the `phangorn` parsimony algorithm. It quickly calculates the parsimony score, the consistency and retention indices and, if a tree is provided (e.g. the tree used to generate the matrix) it calculates the Robinson-Foulds distance between the most parsimonious tree and the provided tree to determine how different they are.

#### 5.1.1.3 Adding inapplicable characters

Once a matrix is generated, it is possible to apply inapplicable characters to it for increasing realism! Inapplicable characters are commonly designated as `NA` or simply `-`. They differ from missing characters `?` in their nature by being inapplicable rather than unknown[see Brazeau et al., 2018, for more details]. For example, considering a binary character defined as "colour of the tail" with the following states "blue" and "red"; on a taxa with no tail, the character should be coded as inapplicable ("`-`") since the state of the character "colour of tail" is *known*: it's neither "blue" or "red", it's just not there! It contrasts with coding it as missing ("`?`" - also called as ambiguous) where the state is *unknown*, for example, the taxon of interest is a fossil where the tail has no colour preserved or is not present at all due to bad conservation!

This type of characters can be added to the simulated matrices using the `apply.NA` function/ It takes, as arguments, the `matrix`, the source of inapplicability (`NAs` - more below), the `tree` used to generate the matrix and the two same `invariant` and `verbose` arguments as defined above. The `NAs` argument allows two types of sources of inapplicability:

- `"character"` where the inapplicability is due to the character (e.g. coding a character tail for species with no tail). In practice, the algorithm chooses a character $X$ as the underlying character (e.g. "presence and absence of tail"), arbitrarily chooses one of the states as "absent" (e.g. 0 = absent) and changes in the next character $Y$ any state next to character $X$ state 0 into an inapplicable token ("`-`"). This simulates the inapplicability induced by coding the characters (i.e. not always biological).
- `"clade"` where the inapplicability is due to evolutionary history (e.g. a clade loosing its tail). In practice, the algorithm chooses a random clade in the tree and a random character $Z$ and replaces the state of the taxa present in the clade by the inapplicable token ("`-`"). This simulates the inapplicability induced by evolutionary biology (e.g. the lose of a feature in a clade).

To apply these sources of inapplicability, simply repeat the number of inapplicable sources for the desired number of characters with inapplicable data.

```
## Generating 5 "character" NAs and 10 "clade" NAs
my_matrix_NA <- apply.NA(my_matrix, tree = my_tree,
                         NAs = c(rep("character", 5),
                                 rep("clade", 10)))

## The first few lines of the resulting matrix
my_matrix_NA[1:10, 90:100]
```

```
##     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11]
## t10 "1"  "0"  "1"  "1"  "2"  "0"  "0"  "0"  "1"  "0"   "1"
## t1  "1"  "0"  "0"  "0"  "2"  "0"  "0"  "0"  "0"  "0"   "1"
## t9  "1"  "0"  "1"  "0"  "2"  "0"  "0"  "0"  "0"  "0"   "1"
## t14 "1"  "0"  "0"  "0"  "2"  "0"  "0"  "0"  "0"  "0"   "0"
## t13 "1"  "0"  "0"  "0"  "2"  "0"  "0"  "0"  "0"  "0"   "0"
## t5  "1"  "0"  "0"  "0"  "2"  "0"  "0"  "0"  "0"  "0"   "0"
## t2  "1"  "0"  "0"  "0"  "2"  "0"  "0"  "0"  "0"  "0"   "0"
## t8  "1"  "0"  "0"  "0"  "2"  "0"  "0"  "0"  "0"  "0"   "0"
## t6  "0"  "1"  "1"  "0"  "0"  "1"  "1"  "2"  "0"  "1"   "1"
## t15 "0"  "1"  "1"  "0"  "0"  "1"  "1"  "2"  "0"  "1"   "1"
```

### 5.1.2 Parameters for a realistic(ish) matrix

There are many parameters that can create a "realistic" matrix (i.e. not too different from the input tree with a consistency and retention index close to what is seen in the literature) but because of the randomness of the matrix generation not all parameters combination end up creating "good" matrices. The following parameters however, seem to generate fairly "realist" matrices with a starting coalescent tree, equal rates model with 0.85 binary characters and 0.15 three state characters, a gamma distribution with a shape parameter ($\alpha$) of 5 and no scaling ($\beta = 1$) with a rate of 100.

```
set.seed(0)
## tree
my_tree <- rcoal(15)
## matrix
morpho_mat <- sim.morpho(my_tree,
                         characters = 100,
                         model = "ER",
                         rates = c(rgamma, rate = 100, shape = 5),
                         invariant = FALSE)
check.morpho(morpho_mat, my_tree)
```

```
##
## Maximum parsimony        103.0000000
## Consistency index          0.9708738
## Retention index            0.9919571
```

```
## Robinson-Foulds distance   4.0000000
```

## 5.2 Simulating multidimensional spaces

Another way to simulate data is to directly simulate an ordinated space with the `space.maker` function. This function allows users to simulate multidimensional spaces with a certain number of properties. For example, it is possible to design a multidimensional space with a specific distribution on each axis, a correlation between the axes and a specific cumulative variance per axis. This can be useful for creating ordinated spaces for null hypothesis, for example if you're using the function `null.test` [Díaz et al., 2016].

This function takes as arguments the number of elements (data points - `elements` argument) and dimensions (`dimensions` argument) to create the space and the distribution functions to be used for each axis. The distributions are passed through the `distribution` argument as… modular functions! You can either pass a single distribution function for all the axes (for example `distribution = runif` for all the axis being uniform) or a specific distribution function for each specific axis (for example `distribution = c(runif, rnorm, rgamma))` for the first axis being uniform, the second normal and the third gamma). You can of course use your very own functions or use the ones implemented in `dispRity` for more complex ones (see below). Specific optional arguments for each of these distributions can be passed as a list via the `arguments` argument.

Furthermore, it is possible to add a correlation matrix to add a correlation between the axis via the `cor.matrix` argument or even a vector of proportion of variance to be bear by each axis via the `scree` argument to simulate realistic ordinated spaces.

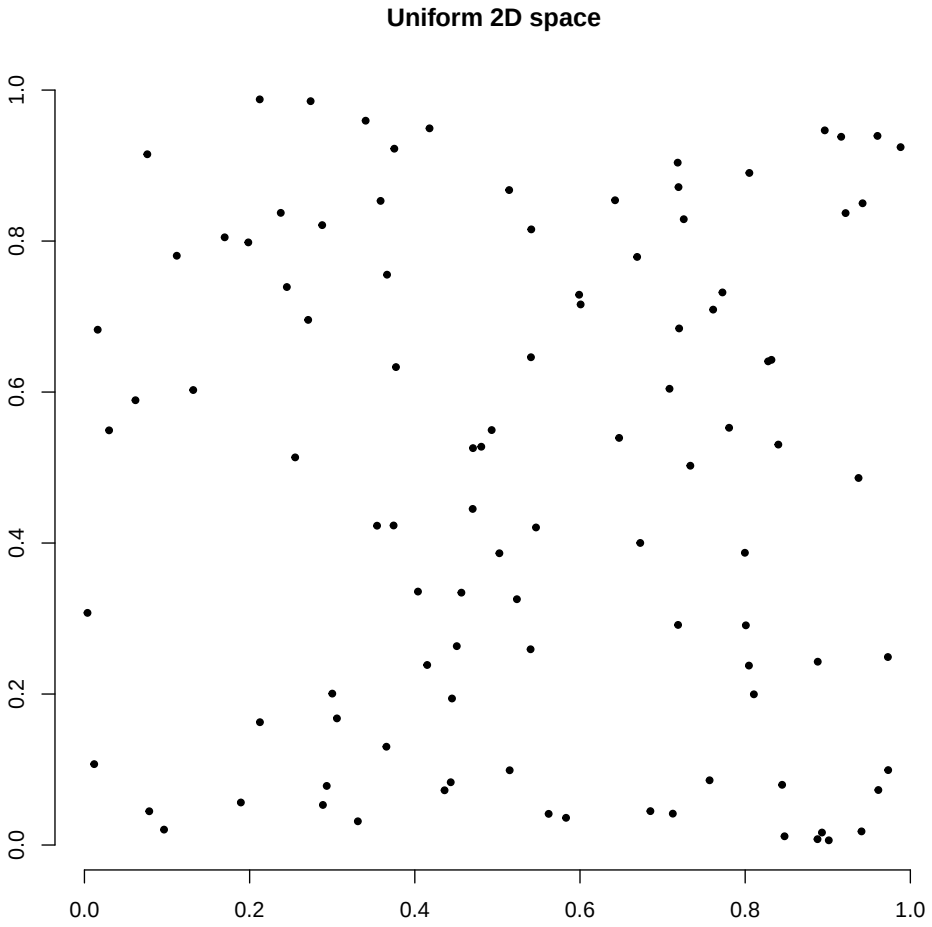
Here is a simple two dimensional example:

```
## Graphical options
op <- par(bty = "n")

## A square space
square_space <- space.maker(100, 2, runif)

## The resulting 2D matrix
head(square_space)
```

```
##           [,1]       [,2]
## [1,] 0.2878797 0.82110157
## [2,] 0.5989886 0.72890558
## [3,] 0.8401571 0.53042419
## [4,] 0.3663870 0.75545936
## [5,] 0.2122375 0.98768804
```

```
## [6,] 0.9612441 0.07285561
```

```
## Visualising the space
plot(square_space, pch = 20, xlab = "", ylab = "",
     main = "Uniform 2D space")
```




Of course, more complex spaces can be created by changing the distributions, their arguments or adding a correlation matrix or a cumulative variance vector:

```
## A plane space: uniform with one dimensions equal to 0
plane_space <- space.maker(2500, 3, c(runif, runif, runif),
                           arguments = list(list(min = 0, max = 0),
                           NULL, NULL))

## Correlation matrix for a 3D space
(cor_matrix <- matrix(cbind(1, 0.8, 0.2, 0.8, 1, 0.7, 0.2, 0.7, 1), nrow = 3))
```

```
##      [,1] [,2] [,3]
```

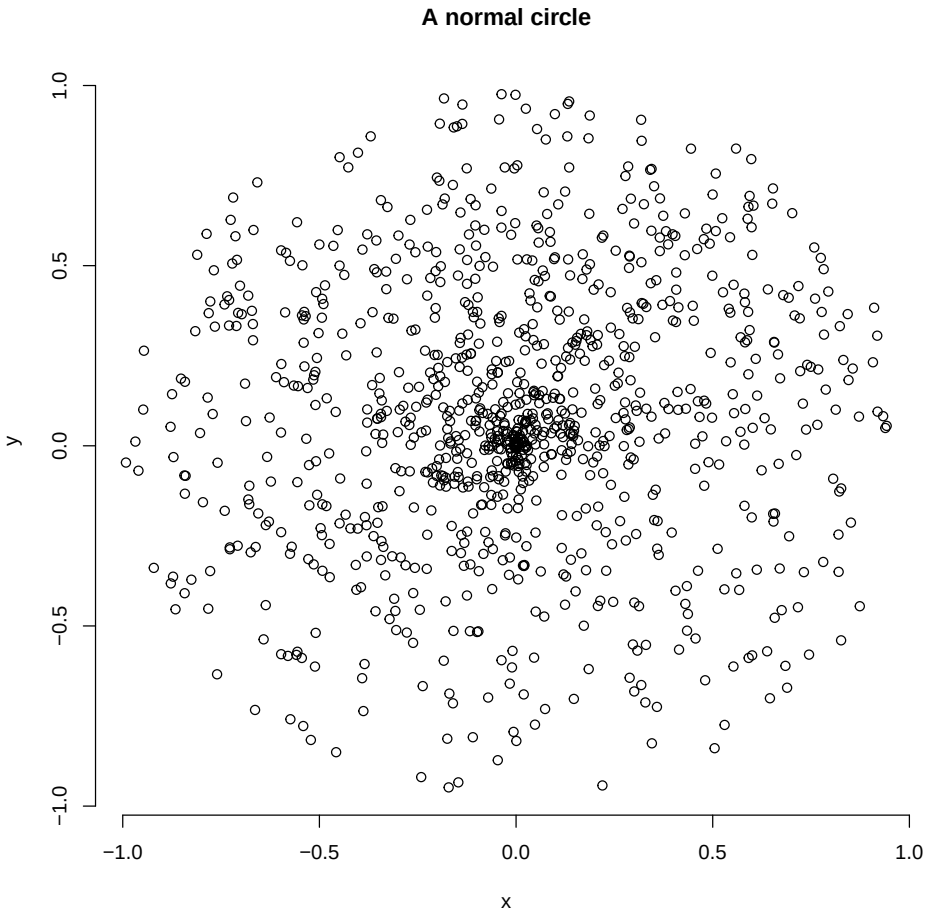
```
## [1,]  1.0  0.8  0.2
## [2,]  0.8  1.0  0.7
## [3,]  0.2  0.7  1.0
```

```
## An ellipsoid space (normal space with correlation)
ellipse_space <- space.maker(2500, 3, rnorm,
                             cor.matrix = cor_matrix)

## A cylindrical space with decreasing axes variance
cylindrical_space <- space.maker(2500, 3, c(rnorm, rnorm, runif),
                                 scree = c(0.7, 0.2, 0.1))
```

### 5.2.1 Personalised dimensions distributions

Following the modular architecture of the package, it is of course possible to pass home made distribution functions to the `distribution` argument. For example, the `random.circle` function is a personalised one implemented in `dispRity`. This function allows to create circles based on basic trigonometry allowing to axis to covary to produce circle coordinates. By default, this function generates two sets of coordinates with a `distribution` argument and a minimum and maximum boundary (`inner` and `outer` respectively) to create nice sharp edges to the circle. The maximum boundary is equivalent to the radius of the circle (it removes coordinates beyond the circle radius) and the minimum is equivalent to the radius of a smaller circle with no data (it removes coordinates below this inner circle radius).

```
## Graphical options
op <- par(bty = "n")

## Generating coordinates for a normal circle with a upper boundary of 1
circle <- random.circle(1000, rnorm, inner = 0, outer = 1)

## Plotting the circle
plot(circle, xlab = "x", ylab = "y", main = "A normal circle")
```




```
## Creating doughnut space (a spherical space with a hole)
doughnut_space <- space.maker(5000, 3, c(rnorm, random.circle),
     arguments = list(list(mean = 0),
                      list(runif, inner = 0.5, outer = 1)))
```

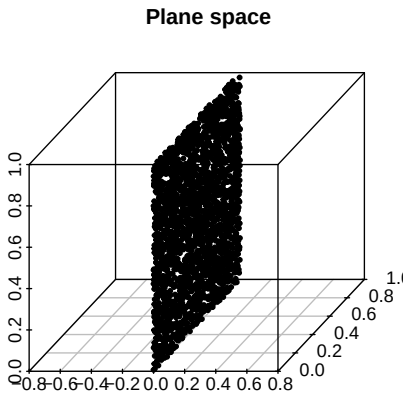
### 5.2.2 Visualising the space

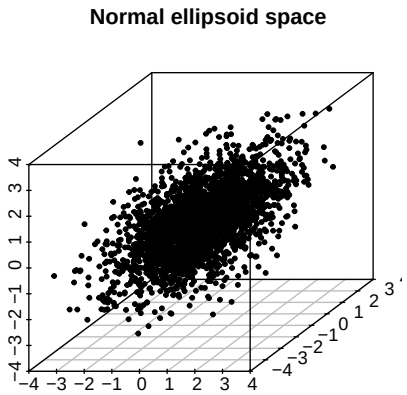
I suggest using the excellent `scatterplot3d` package to play around and visualise the simulated spaces:

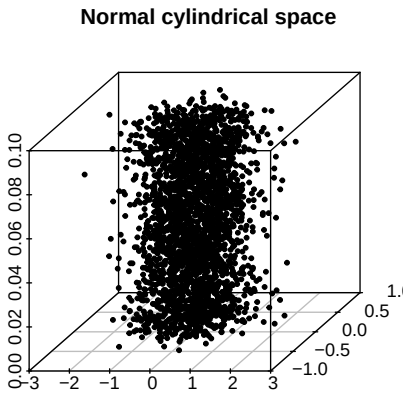
```
## Graphical options
op <- par(mfrow = (c(2, 2)), bty = "n")
## Visualising 3D spaces
require(scatterplot3d)
```

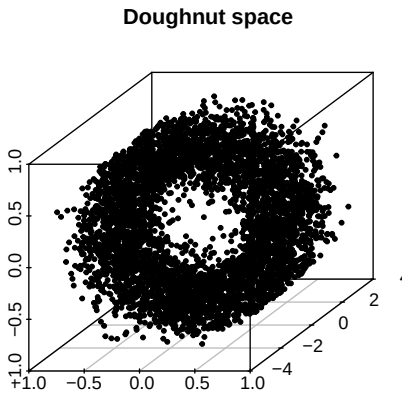
```
## Loading required package: scatterplot3d
```

```
## The plane space
scatterplot3d(plane_space, pch = 20, xlab = "", ylab = "", zlab = "",
              xlim = c(-0.5, 0.5), main = "Plane space")

## The ellipsoid space
scatterplot3d(ellipse_space, pch = 20, xlab = "", ylab = "", zlab = "",
              main = "Normal ellipsoid space")

## A cylindrical space with a decreasing variance per axis
scatterplot3d(cylindrical_space, pch = 20, xlab = "", ylab = "", zlab = "",
              main = "Normal cylindrical space")
## Axes have different orders of magnitude

## Plotting the doughnut space
scatterplot3d(doughnut_space[,c(2,1,3)], pch = 20, xlab = "", ylab = "",
              zlab = "", main = "Doughnut space")
```



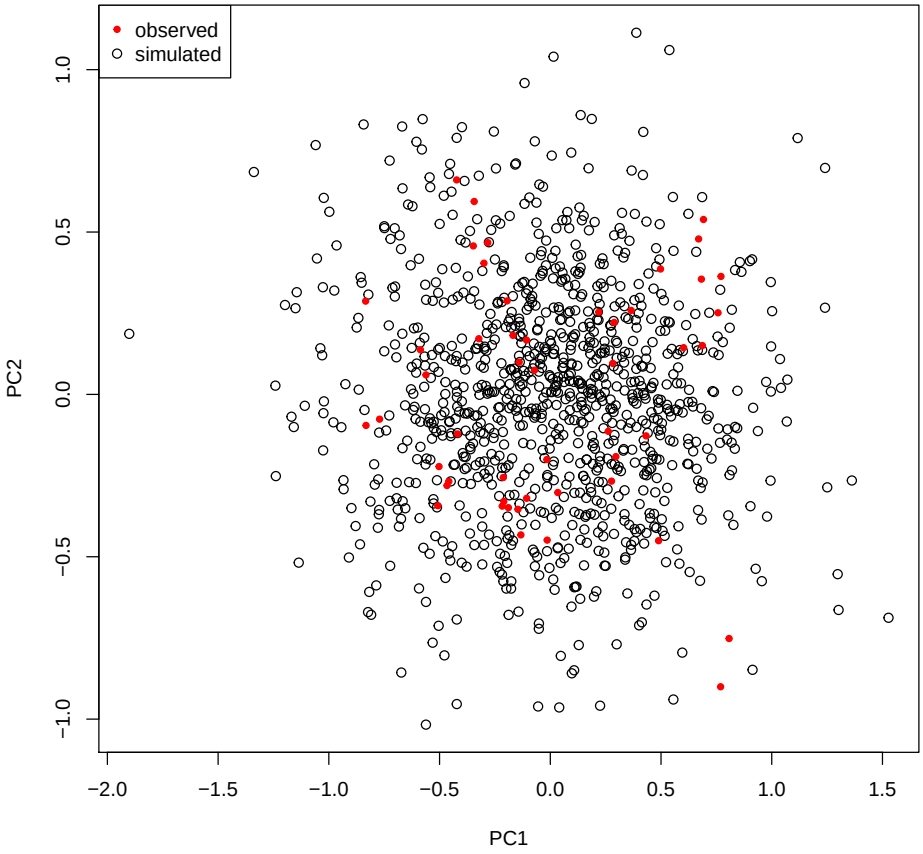



```
par(op)
```

### 5.2.3 Generating realistic spaces

It is possible to generate "realistic" spaces by simply extracting the parameters of an existing space and scaling it up to the simulated space. For example, we can extract the parameters of the `BeckLee_mat50` ordinated space and simulate a similar space.

```
## Loading the data
data(BeckLee_mat50)

## Number of dimensions
obs_dim <- ncol(BeckLee_mat50)

## Observed correlation between the dimensions
obs_correlations <- cor(BeckLee_mat50)

## Observed mean and standard deviation per axis
obs_mu_sd_axis <- mapply(function(x,y) list("mean" = x, "sd" = y),
                         as.list(apply(BeckLee_mat50, 2, mean)),
                         as.list(apply(BeckLee_mat50, 2, sd)), SIMPLIFY = FALSE)

## Observed overall mean and standard deviation
obs_mu_sd_glob <- list("mean" = mean(BeckLee_mat50), "sd" = sd(BeckLee_mat50))

## Scaled observed variance per axis (scree plot)
obs_scree <- variances(BeckLee_mat50)/sum(variances(BeckLee_mat50))

## Generating our simulated space
simulated_space <- space.maker(1000, dimensions = obs_dim,
                               distribution = rep(list(rnorm), obs_dim),
                               arguments = obs_mu_sd_axis,
                               cor.matrix = obs_correlations)

## Visualising the fit of our data in the space (in the two first dimensions)
plot(simulated_space[,1:2], xlab = "PC1", ylab = "PC2")
points(BeckLee_mat50[,1:2], col = "red", pch = 20)
legend("topleft", legend = c("observed", "simulated"),
       pch = c(20,21), col = c("red", "black"))
```

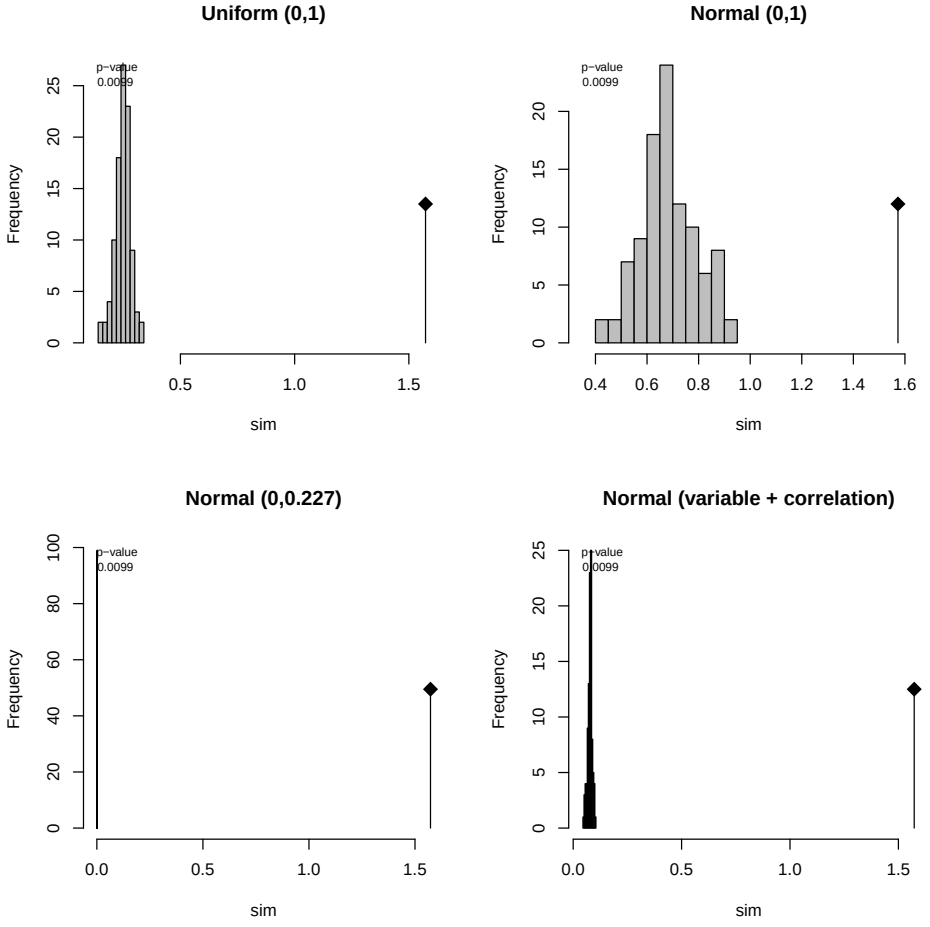



It is now possible to simulate a space using these observed arguments to test several hypothesis:

- Is the space uniform or normal?
- If the space is normal, is the mean and variance global or specific for each axis?

```
## Measuring disparity as the sum of variance
observed_disp <- dispRity(BeckLee_mat50, metric = c(median, centroids))

## Is the space uniform?
test_unif <- null.test(observed_disp, null.distrib = runif)

## Is the space normal with a mean of 0 and a sd of 1?
test_norm1 <- null.test(observed_disp, null.distrib = rnorm)

## Is the space normal with the observed mean and sd and cumulative variance
test_norm2 <- null.test(observed_disp, null.distrib = rep(list(rnorm), obs_dim),
                        null.args = rep(list(obs_mu_sd_glob), obs_dim),
                        null.scree = obs_scree)
```

```
## Is the space multiple normal with multiple means and sds and a correlation?
test_norm3 <- null.test(observed_disp, null.distrib = rep(list(rnorm), obs_dim),
                        null.args = obs_mu_sd_axis, null.cor = obs_correlations)

## Graphical options
op <- par(mfrow = (c(2, 2)), bty = "n")
## Plotting the results
plot(test_unif, main = "Uniform (0,1)")
plot(test_norm1, main = "Normal (0,1)")
plot(test_norm2, main = paste0("Normal (", round(obs_mu_sd_glob[[1]], digit = 3),
                               ",", round(obs_mu_sd_glob[[2]], digit = 3), ")"))
plot(test_norm3, main = "Normal (variable + correlation)")
```




If we measure disparity as the median distance from the morphospace centroid, we can explain the distribution of the data as normal with the variable observed mean and standard deviation and with a correlation between the dimensions.

# Chapter 6

# Other functionalities

The `dispRity` package also contains several other functions that are not specific to multidimensional analysis but that are often used by `dispRity` internal functions. However, we decided to make these functions also available at a user level since they can be handy for certain specific operations! You'll find a brief description of each of them (alphabetically) here:

## 6.1 `char.diff`

This is yet another function for calculating distance matrices. There are many functions for calculating pairwise distance matrices in `R` (`stats::dist`, `vegan::vegdist`, `cluster::daisy` or `Claddis::calculate_morphological_distances`) but this one is the `dispRity` one. It is slightly different to the ones mentioned above (though not that dissimilar from `Claddis::calculate_morphological_distances`) in the fact that it focuses on comparing discrete morphological characters and tries to solve all the problems linked to these kind of matrices (especially dealing with special tokens).

The function intakes a matrix with either `numeric` or `integer` (`NA` included) or matrices with `character` that are indeed `integers` (e.g.`"0"` and `"1"`). It then uses a bitwise operations architecture implemented in `C` that renders the function pretty fast and pretty modular. This bitwise operations translates the character states into binary values. This way, `0` becomes `1`, `1` becomes `2`, `2` becomes `4`, `3` becomes `8`, etc… Specifically it can handle any rules specific to special tokens (i.e. symbols) for discrete morphological characters. For example, should you treat missing values `"?"` as `NA` (ignoring them) or as any possible character state (e.g. `c("0", "1")`?)? And how to treat characters with a ampersand (`"&"`)? `char.diff` can answer to all these questions!

Let's start by a basic binary matrix 4*3 with random `integer`:

```
## A random binary matrix
matrix_binary <- matrix(sample(c(0,1), 12, replace = TRUE), ncol = 4,
                        dimnames = list(letters[1:3], LETTERS[1:4]))
```

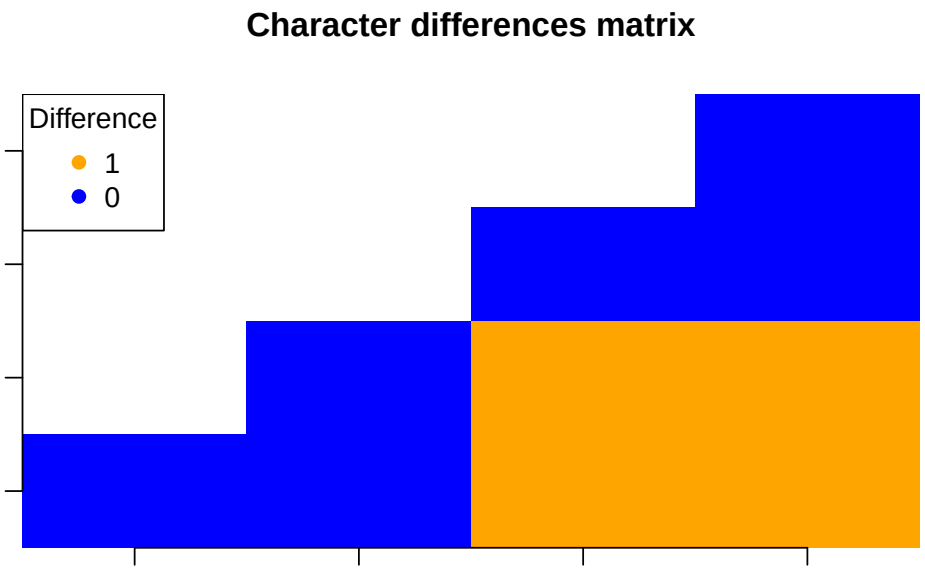
By default, `char.diff` measures the hamming distance between characters:

```
## The hamming distance between characters
(differences <- char.diff(matrix_binary))
```

```
##   A B C D
## A 0 0 1 1
## B 0 0 1 1
## C 1 1 0 0
## D 1 1 0 0
## attr(,"class")
## [1] "matrix"    "char.diff"
```

Note that the results is just a pairwise distance (dissimilarity) matrix with some special dual class `matrix` and `char.diff`. This means it can easily be plotted via the disparity package:

```
## Visualising the matrix
plot(differences)
```




You can check all the numerous plotting options in the `?plot.char.diff` manual (it won't be developed here).

The `char.diff` function has much more options however (see all of them in the `?char.diff` manual) for example to measure different differences (via `method`) or making the comparison work per row (for a distance matrix between the

rows):

```
## Euclidean distance between rows
char.diff(matrix_binary, by.col = FALSE, method = "euclidean")
```

```
##          a        b        c
## a 0.000000 1.414214 1.414214
## b 1.414214 0.000000 0.000000
## c 1.414214 0.000000 0.000000
## attr(,"class")
## [1] "matrix"    "char.diff"
```

We can however make it more interesting by playing with the different rules to play with different tokens. First let's create a matrix with morphological characters as numeric `characters`:

```
## A random character matrix
(matrix_character <- matrix(sample(c("0","1","2"), 30, replace = TRUE), ncol = 5,
                            dimnames = list(letters[1:6], LETTERS[1:5])))
```

```
##   A   B   C   D   E
## a "1" "1" "1" "1" "0"
## b "0" "2" "0" "2" "0"
## c "2" "2" "1" "2" "0"
## d "1" "2" "0" "0" "1"
## e "2" "2" "1" "1" "2"
## f "0" "2" "0" "2" "0"
```

```
## The hamming difference between columns
char.diff(matrix_character)
```

```
##     A   B   C   D   E
## A 0.0 0.6 0.6 0.6 0.8
## B 0.6 0.0 0.4 0.4 0.8
## C 0.6 0.4 0.0 0.4 0.6
## D 0.6 0.4 0.4 0.0 1.0
## E 0.8 0.8 0.6 1.0 0.0
## attr(,"class")
## [1] "matrix"    "char.diff"
```

Here the characters are automatically converted into bitwise integers to be compared efficiently. We can now add some more special tokens like `"?"` or `"0/1"` for uncertainties between state `"0"` and `"1"` but not `"2"`:

```
## Adding uncertain characters
matrix_character[sample(1:30, 8)] <- "0/1"

## Adding missing data
(matrix_character[sample(1:30, 5)] <- "?")
```

```
## [1] "?"
```

```
## The hamming difference between columns including the special characters
char.diff(matrix_character)
```

```
##           A         B    C    D         E
## A 0.0000000 0.6666667 1.00 0.50 0.6666667
## B 0.6666667 0.0000000 1.00 1.00 0.7500000
## C 1.0000000 1.0000000 0.00 0.00 0.2500000
## D 0.5000000 1.0000000 0.00 0.00 0.2500000
## E 0.6666667 0.7500000 0.25 0.25 0.0000000
## attr(,"class")
## [1] "matrix"    "char.diff"
```

Note here that it detected the default behaviours for the special tokens `"?"` and `"/"`: `"?"` are treated as `NA` (not compared) and `"/"` are treated as both states (e.g. `"0/1"` is treated as `"0"` and as `"1"`). We can specify both the special tokens and the special behaviours to consider via `special.tokens` and `special.behaviours`. The `special.tokens` are `missing = "?"`, `inapplicable = "-"`, `uncertainty = "\"` and `polymorphism = "&"` meaning we don't have to modify them for now. However, say we want to change the behaviour for `"?"` and treat them as all possible characters and treat `"/"` as only the character `"0"` (as an integer) we can specify them giving a behaviour function:

```
## Specifying some special behaviours
my_special_behaviours <- list(missing = function(x,y) return(y),
                              uncertainty = function(x,y) return(as.integer(0)))

## Passing these special behaviours to the char.diff function
char.diff(matrix_character, special.behaviour = my_special_behaviours)
```

```
##     A   B   C   D   E
## A 0.0 0.6 0.6 0.6 0.6
## B 0.6 0.0 0.8 0.8 0.8
## C 0.6 0.8 0.0 0.4 0.6
## D 0.6 0.8 0.4 0.0 1.0
## E 0.6 0.8 0.6 1.0 0.0
## attr(,"class")
## [1] "matrix"    "char.diff"
```

The results are quiet different as before! Note that you can also specify some really specific behaviours for any type of special token.

```
## Adding weird tokens to the matrix
matrix_character[sample(1:30, 8)] <- "%"

## Specify the new token and the new behaviour
char.diff(matrix_character, special.tokens = c(weird_one = "%"),
```

```
                            special.behaviours = list(
                                weird_one = function(x,y) return(as.integer(42)))
                            )
```

```
##     A   B C D   E
## A   0   1 1 0 NaN
## B   1   0 1 1 NaN
## C   1   1 0 0   0
## D   0   1 0 0   0
## E NaN NaN 0 0   0
## attr(,"class")
## [1] "matrix"    "char.diff"
```

Of course the results can be quiet surprising then… But that's the essence of the modularity. You can see more options in the function manual `?char.diff`!

## 6.2 clean.data

This is a rather useful function that allows matching a `matrix` or a `data.frame` to a tree (`phylo`) or a distribution of trees (`multiPhylo`). This function outputs the cleaned data and trees (if cleaning was needed) and a list of dropped rows and tips.

```
## Generating a trees with labels from a to e
dummy_tree <- rtree(5, tip.label = LETTERS[1:5])

## Generating a matrix with rows from b to f
dummy_data <- matrix(1, 5, 2, dimnames = list(LETTERS[2:6], c("var1", "var2")))

##Cleaning the trees and the data
(cleaned <- clean.data(data = dummy_data, tree = dummy_tree))
```

```
## $tree
##
## Phylogenetic tree with 4 tips and 3 internal nodes.
##
## Tip labels:
##   D, B, E, C
##
## Rooted; includes branch lengths.
##
## $data
##   var1 var2
## B    1    1
## C    1    1
## D    1    1
```

```
## E    1    1
##
## $dropped_tips
## [1] "A"
##
## $dropped_rows
## [1] "F"
```

## 6.3 crown.stem

This function quiet handily separates tips from a phylogeny between crown members (the living taxa and their descendants) and their stem members (the fossil taxa without any living relatives).

```
data(BeckLee_tree)
## Diving both crow and stem species
(crown.stem(BeckLee_tree, inc.nodes = FALSE))
```

```
## $crown
##  [1] "Dasypodidae"    "Bradypus"      "Myrmecophagidae" "Todralestes"
##  [5] "Potamogalinae"  "Dilambdogale"   "Widanelfarasia"  "Rhynchocyon"
##  [9] "Procavia"      "Moeritherium"   "Pezosiren"     "Trichechus"
## [13] "Tribosphenomys" "Paramys"       "Rhombomylus"    "Gomphos"
## [17] "Mimotona"      "Cynocephalus"   "Purgatorius"    "Plesiadapis"
## [21] "Notharctus"     "Adapis"       "Patriomanis"    "Protictis"
## [25] "Vulpavus"      "Miacis"       "Icaronycteris"  "Soricidae"
## [29] "Solenodon"       "Eoryctes"
##
## $stem
##  [1] "Daulestes"           "Bulaklestes"          "Uchkudukodon"
##  [4] "Kennalestes"          "Asioryctes"          "Ukhaatherium"
##  [7] "Cimolestes"            "unnamed_cimolestid"    "Maelestes"
## [10] "Batodon"              "Kulbeckia"           "Zhangolestes"
## [13] "unnamed_zalambdalestid" "Zalambdalestes"      "Barunlestes"
## [16] "Gypsonictops"          "Leptictis"           "Oxyclaenus"
## [19] "Protungulatum"           "Oxyprimus"
```

Note that it is possible to include or exclude nodes from the output. To see a more applied example: this function is used in chapter 03: specific tutorials.

## 6.4 get.bin.ages

This function is similar than the `crown.stem` one as it is based on a tree but this one outputs the stratigraphic bins ages that the tree is covering. This can

be useful to generate precise bin ages for the `chrono.subsets` function:

```
get.bin.ages(BeckLee_tree)
```

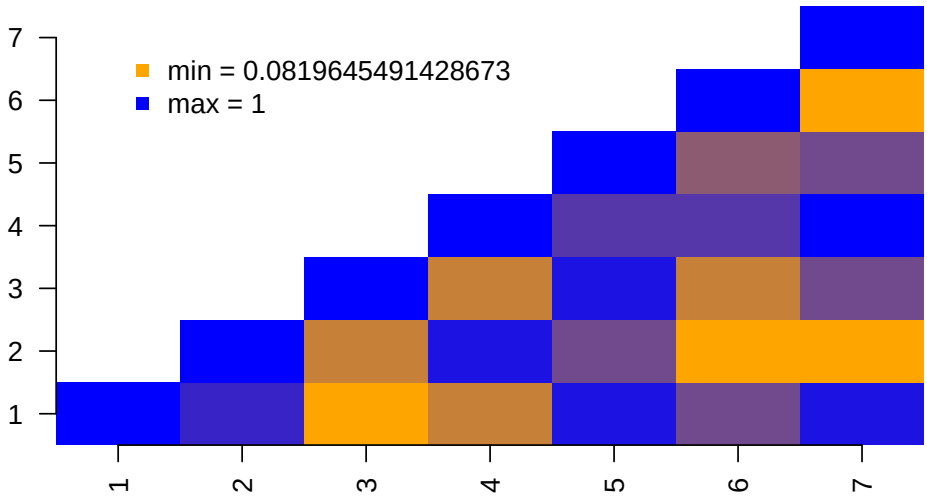
```
##  [1] 132.9000 129.4000 125.0000 113.0000 100.5000  93.9000  89.8000  86.3000
##  [9]  83.6000  72.1000  66.0000  61.6000  59.2000  56.0000  47.8000  41.2000
## [17]  37.8000  33.9000  28.1000  23.0300  20.4400  15.9700  13.8200  11.6300
## [25]   7.2460   5.3330   3.6000   2.5800   1.8000   0.7810   0.1260   0.0117
## [33]   0.0000
```

Note that this function outputs the stratigraphic age limits by default but this can be customisable by specifying the `type` of data (e.g. `type = "Eon"` for eons). The function also intakes several optional arguments such as whether to output the startm end, range or midpoint of the stratigraphy or the year of reference of the International Commission of Stratigraphy. To see a more applied example: this function is used in chapter 03: specific tutorials.

## 6.5 pair.plot

This utility function allows to plot a matrix image of pairwise comparisons. This can be useful when getting pairwise comparisons and if you'd like to see at a glance which pairs of comparisons have high or low values.

```
## Random data
data <- matrix(data = runif(42), ncol = 2)

## Plotting the first column as a pairwise comparisons
pair.plot(data, what = 1, col = c("orange", "blue"), legend = TRUE, diag = 1)
```

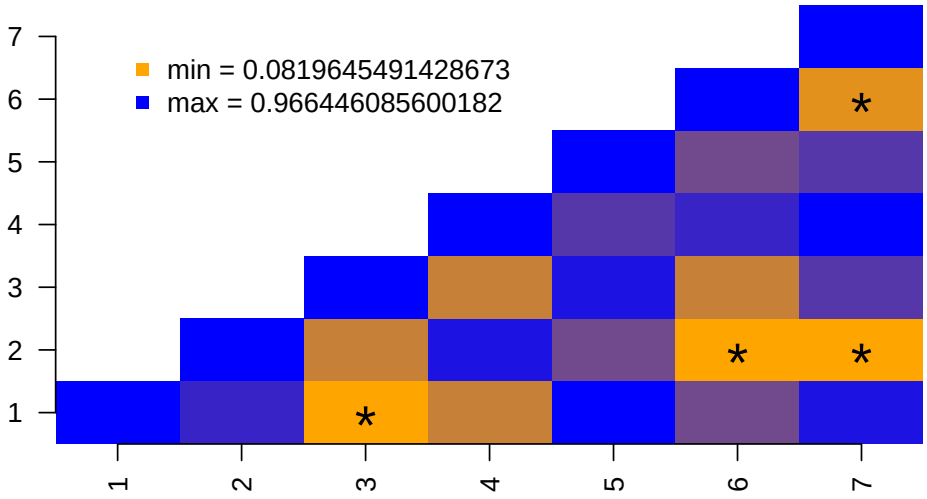



Here blue squares are ones that have a high value and orange ones the ones that have low values. Note that the values plotted correspond the first column of the

data as designated by `what = 1`.

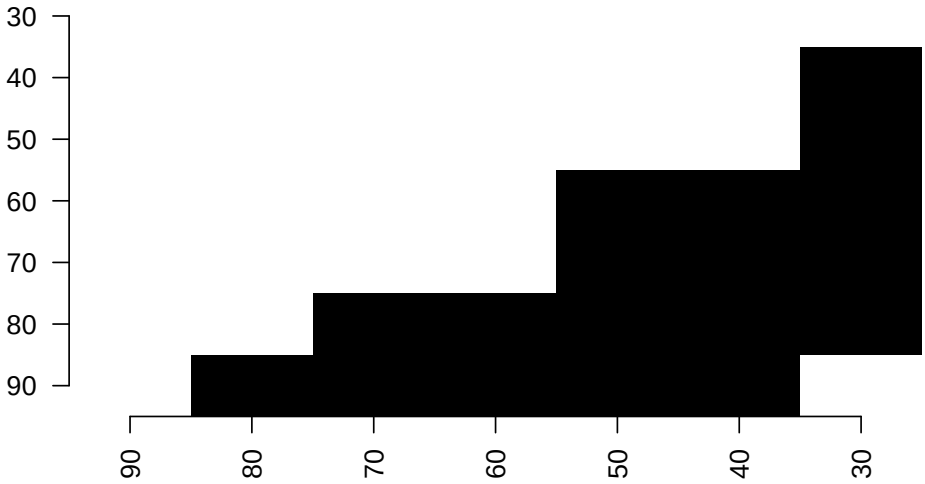
It is also possible to add some tokens or symbols to quickly highlight to specific cells, for example which elements in the data are below a certain value:

```
## The same plot as before without the diagonal being the maximal observed value
pair.plot(data, what = 1, col = c("orange", "blue"), legend = TRUE, diag = "max")
## Highlighting with an asterisk which squares have a value below 0.2
pair.plot(data, what = 1, binary = 0.2, add = "*", cex = 2)
```




This function can also be used as a binary display when running a series of pairwise t-tests. For example, the following script runs a wilcoxon test between the time-slices from the `disparity` example dataset and displays in black which pairs of slices have a p-value below 0.05:

```
## Loading disparity data
data(disparity)

## Testing the pairwise difference between slices
tests <- test.dispRity(disparity, test = wilcox.test, correction = "bonferroni")

## Plotting the significance
pair.plot(as.data.frame(tests), what = "p.value", binary = 0.05)
```

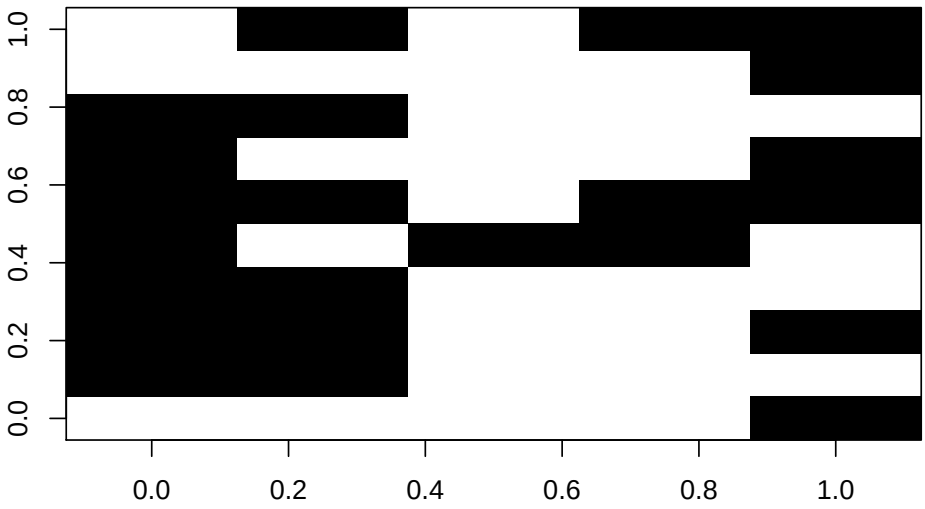



## 6.6 reduce.matrix

This function allows to reduce columns or rows of a matrix to make sure that there is enough overlap for further analysis. This is particularly useful if you are going to use distance matrices since it uses the `vegan::vegdist` function to test whether distances can be calculated or not.

For example, if we have a patchy matrix like so (where the black squares represent available data):

```
set.seed(1)
## A 10*5 matrix
na_matrix <- matrix(rnorm(50), 10, 5)
## Making sure some rows don't overlap
na_matrix[1, 1:2] <- NA
na_matrix[2, 3:5] <- NA
## Adding 50% NAs
na_matrix[sample(1:50, 25)] <- NA
## Illustrating the gappy matrix
image(t(na_matrix), col = "black")
```



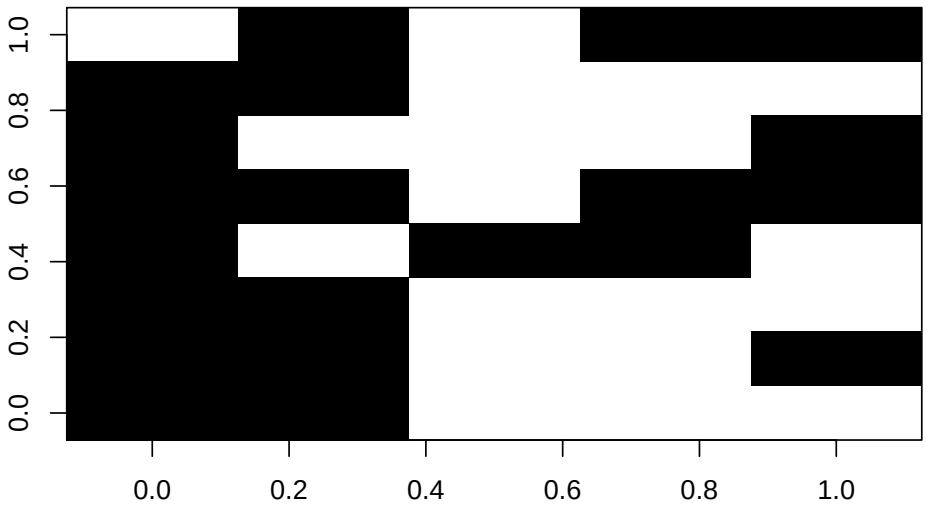

We can use the `reduce.matrix` to double check whether any rows cannot be compared. The functions needs as an input the type of distance that will be used, say a `"gower"` distance:

```
## Reducing the matrix by row
(reduction <- reduce.matrix(na_matrix, distance = "gower"))
```

```
## $rows.to.remove
## [1] "9" "1"
##
## $cols.to.remove
## NULL
```

We can not remove the rows 1 and 9 and see if that improved the overlap:

```
image(t(na_matrix[-as.numeric(reduction$rows.to.remove), ]), col = "black")
```



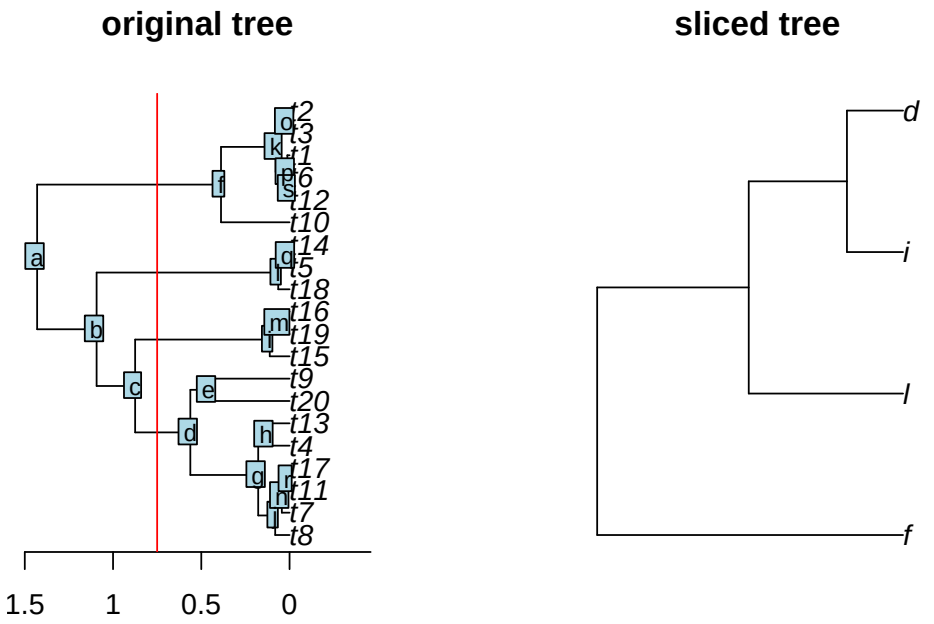

## 6.7 slice.tree

This function is a modification of the `paleotree::timeSliceTree` function that allows to make slices through a phylogenetic tree. Compared to the `paleotree::timeSliceTree`, this function allows a model to decide which tip or node to use when slicing through a branch (whereas `paleotree::timeSliceTree` always choose the first available tip alphabetically). The models for choosing which tip or node are the same as the ones used in the `chrono.subsets` and are described in chapter 03: specific tutorials.

The function works by using at least a tree, a slice age and a model:

```
set.seed(1)
## Generate a random ultrametric tree
tree <- rcoal(20)
## Add some node labels
tree$node.label <- letters[1:19]
## Add its root time
tree$root.time <- max(tree.age(tree)$ages)

## Slicing the tree at age 0.75
tree_75 <- slice.tree(tree, age = 0.75, "acctran")

## Showing both trees
par(mfrow = c(1,2))
plot(tree, main = "original tree")
axisPhylo() ; nodelabels(tree$node.label, cex = 0.8)
abline(v = (max(tree.age(tree)$ages) - 0.75), col = "red")
```

```
plot(tree_75, main = "sliced tree")
```



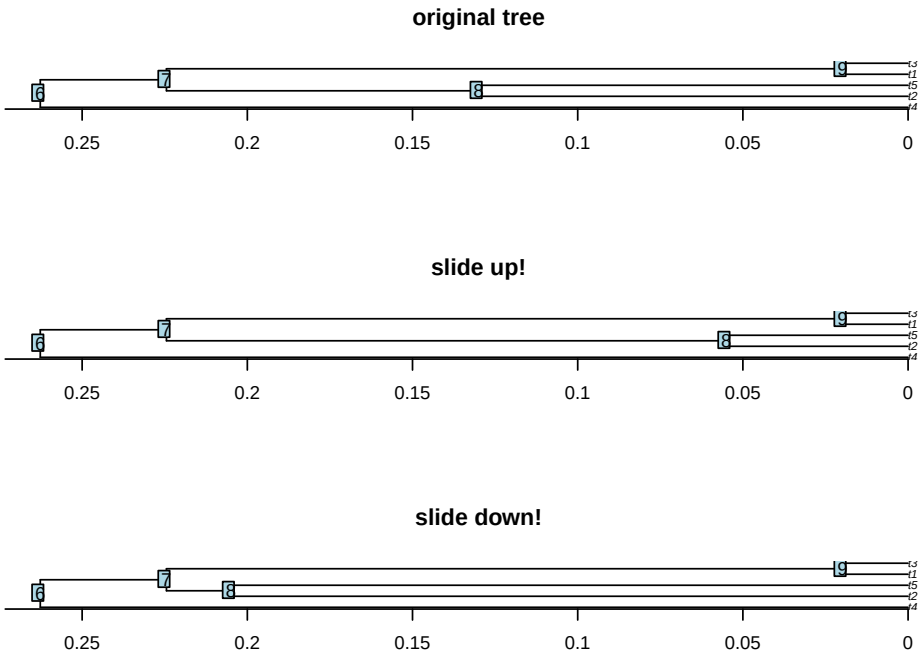

## 6.8 `slide.nodes` and `remove.zero.brlen`

This function allows to slide nodes along a tree! In other words it allows to change the branch length leading to a node without modifying the overall tree shape. This can be useful to add some value to 0 branch lengths for example.

The function works by taking a node (or a list of nodes), a tree and a sliding value. The node will be moved "up" (towards the tips) for the given sliding value. You can move the node "down" (towards the roots) using a negative value.

```
set.seed(42)
## Generating simple coalescent tree
tree <- rcoal(5)

## Sliding node 8 up and doum
tree_slide_up <- slide.nodes(8, tree, slide = 0.075)
tree_slide_down <- slide.nodes(8, tree, slide = -0.075)

## Display the results
par(mfrow = c(3,1))
plot(tree, main = "original tree") ; axisPhylo() ; nodelabels()
plot(tree_slide_up, main = "slide up!") ; axisPhylo() ; nodelabels()
```

```
plot(tree_slide_down, main = "slide down!") ; axisPhylo() ; nodelabels()
```




The `remove.zero.brlen` is a "clever" wrapping function that uses the `slide.nodes` function to stochastically remove zero branch lengths across a whole tree. This function will slide nodes up or down in successive postorder traversals (i.e. going down the tree clade by clade) in order to minimise the number of nodes to slide while making sure there are no silly negative branch lengths produced! By default it is trying to slide the nodes using 1% of the minimum branch length to avoid changing the topology too much.

```
set.seed(42)
## Generating a tree
tree <- rtree(20)

## Adding some zero branch lengths (5)
tree$edge.length[sample(1:Nedge(tree), 5)] <- 0

## And now removing these zero branch lengths!
tree_no_zero <- remove.zero.brlen(tree)

## Exaggerating the removal (to make it visible)
tree_exaggerated <- remove.zero.brlen(tree, slide = 1)

## Check the differences
any(tree$edge.length == 0)
```

```
## [1] TRUE
```

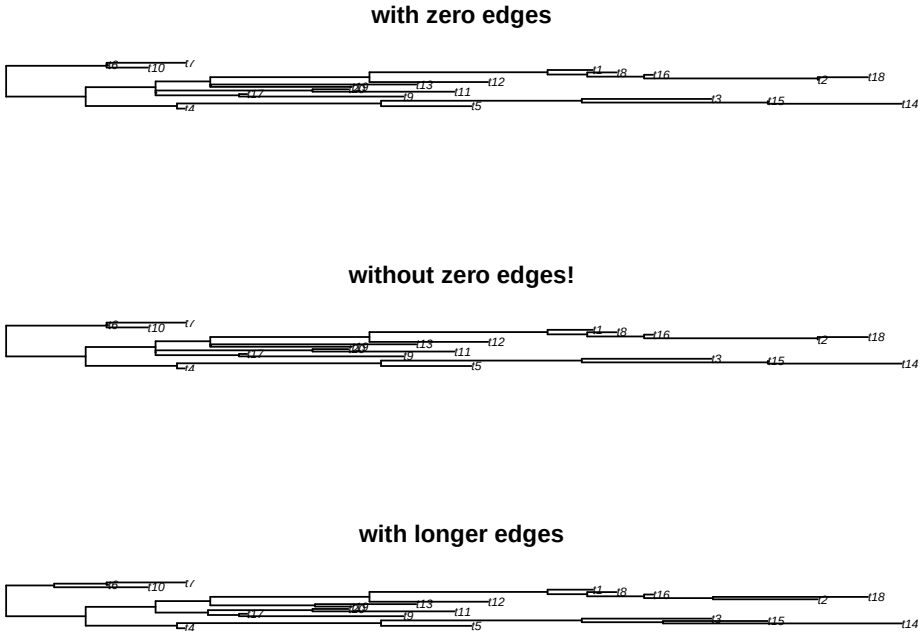
```
any(tree_no_zero$edge.length == 0)
```

```
## [1] FALSE
```

```
any(tree_exaggerated$edge.length == 0)
```

```
## [1] FALSE
```

```
## Display the results
par(mfrow = c(3,1))
plot(tree, main = "with zero edges")
plot(tree_no_zero, main = "without zero edges!")
plot(tree_exaggerated, main = "with longer edges")
```




## 6.9 tree.age

This function allows to quickly calculate the ages of each tips and nodes present in a tree.

```
set.seed(1)
tree <- rtree(10)
## The tree age from a 10 tip tree
tree.age(tree)
```

```
##     ages elements
```

```
## 1  0.707       t7
## 2  0.142       t2
## 3  0.000       t3
## 4  1.467       t8
## 5  1.366       t1
## 6  1.895       t5
## 7  1.536       t6
## 8  1.456       t9
## 9  0.815      t10
## 10 2.343       t4
## 11 3.011       11
## 12 2.631       12
## 13 1.854       13
## 14 0.919       14
## 15 0.267       15
## 16 2.618       16
## 17 2.235       17
## 18 2.136       18
## 19 1.642       19
```

It also allows to set the age of the root of the tree:

```
## The ages starting from -100 units
tree.age(tree, age = 100)
```

```
##       ages elements
## 1   23.472       t7
## 2    4.705       t2
## 3    0.000       t3
## 4   48.736       t8
## 5   45.352       t1
## 6   62.931       t5
## 7   51.012       t6
## 8   48.349       t9
## 9   27.055      t10
## 10  77.800       t4
## 11 100.000       11
## 12  87.379       12
## 13  61.559       13
## 14  30.517       14
## 15   8.875       15
## 16  86.934       16
## 17  74.235       17
## 18  70.924       18
## 19  54.533       19
```

Usually tree age is calculated from the present to the past (e.g. in million years ago) but it is possible to reverse it using the `order = present` option:

```r
## The ages in terms of tip/node height
tree.age(tree, order = "present")
```

```
##     ages elements
## 1  2.304       t7
## 2  2.869       t2
## 3  3.011       t3
## 4  1.544       t8
## 5  1.646       t1
## 6  1.116       t5
## 7  1.475       t6
## 8  1.555       t9
## 9  2.196      t10
## 10 0.668       t4
## 11 0.000       11
## 12 0.380       12
## 13 1.157       13
## 14 2.092       14
## 15 2.744       15
## 16 0.393       16
## 17 0.776       17
## 18 0.876       18
## 19 1.369       19
```

# Chapter 7

# The guts of the dispRity package

## 7.1 Manipulating dispRity objects

Disparity analysis involves a lot of manipulation of many matrices (especially when bootstrapping) which can be impractical to visualise and will quickly overwhelm your R console. Even the simple Beck and Lee 2014 example above produces an object with $> 72$ lines of lists of lists of matrices!

Therefore dispRity uses a specific class of object called a dispRity object. These objects allow users to use S3 method functions such as summary.dispRity, plot.dispRity and print.dispRity. dispRity also contains various utility functions that manipulate the dispRity object (e.g. sort.dispRity, extract.dispRity see the full list in the next section). These functions modify the dispRity object without having to delve into its complex structure! The full structure of a dispRity object is detailed here.

```
## Loading the example data
data(disparity)

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

```
## [1] "dispRity"
```

```
## What does the object contain?
names(disparity)
```

```
## [1] "matrix"    "tree"      "call"      "subsets"   "disparity"
```

```
## Summarising it using the S3 method print.dispRity
disparity
```

```
##  ---- dispRity object ----
## 7 continuous (acctran) time subsets for 99 elements in one matrix with 97 dimensions witl
##      90, 80, 70, 60, 50 ...
## Data was bootstrapped 100 times (method:"full") and rarefied to 20, 15, 10, 5 elements.
## Disparity was calculated as: c(median, centroids).
```

Note that it is always possible to recall the full object using the argument `all = TRUE` in `print.dispRity`:

```
## Display the full object
print(disparity, all = TRUE)
## This is more nearly ~ 5000 lines on my 13 inch laptop screen!
```

## 7.2 dispRity utilities

The package also provides some utility functions to facilitate multidimensional analysis.

### 7.2.1 dispRity object utilities

The first set of utilities are functions for manipulating `dispRity` objects:

#### 7.2.1.1 make.dispRity

This function creates empty `dispRity` objects.

```
## Creating an empty dispRity object
make.dispRity()
```

```
## Empty dispRity object.
```

```
## Creating an "empty" dispRity object with a matrix
(disparity_obj <- make.dispRity(matrix(rnorm(20), 5, 4)))
```

```
##  ---- dispRity object ----
## Contains a matrix 5x4.
```

#### 7.2.1.2 fill.dispRity

This function initialises a `dispRity` object and generates its call properties.

```
## The dispRity object's call is indeed empty
disparity_obj$call
```

```
## list()
```

```
## Filling an empty disparity object (that needs to contain at least a matrix)
(disparity_obj <- fill.dispRity(disparity_obj))
```

```
## Warning in check.dispRity.data(data$matrix): Row names have been automatically
## added to data$matrix.
```

```
##  ---- dispRity object ----
## 5 elements in one matrix with 4 dimensions.
```

```
## The dipRity object has now the correct minimal attributes
disparity_obj$call
```

```
## $dimensions
## [1] 1 2 3 4
```

#### 7.2.1.3 matrix.dispRity

This function extracts a specific matrix from a disparity object. The matrix can be one of the bootstrapped matrices or/and a rarefied matrix.

```
## Extracting the matrix containing the coordinates of the elements at time 50
str(matrix.dispRity(disparity, "50"))
```

```
##  num [1:18, 1:97] -0.1036 0.4318 0.3371 0.0501 0.685 ...
##  - attr(*, "dimnames")=List of 2
##  ..$ : chr [1:18] "Leptictis" "Dasypodidae" "n24" "Potamogalinae" ...
##   ..$ : NULL
```

```
## Extracting the 3rd bootstrapped matrix with the 2nd rarefaction level
## (15 elements) from the second group (80 Mya)
str(matrix.dispRity(disparity, subsets = 1, bootstrap = 3, rarefaction = 2))
```

```
##  num [1:15, 1:97] -0.12948 -0.57973 0.00361 0.27123 0.27123 ...
##  - attr(*, "dimnames")=List of 2
##   ..$ : chr [1:15] "n15" "Maelestes" "n20" "n34" ...
##   ..$ : NULL
```

#### 7.2.1.4 n.subsets

This function simply counts the number of subsets in a dispRity object.

```
## How many subsets are in this object?
n.subsets(disparity)
```

```
## [1] 7
```

#### 7.2.1.5 size.subsets

This function tells the number of elements in each subsets of a dispRity object.

```
## How many elements are there in each subset?
size.subsets(disparity)
```

```
## 90 80 70 60 50 40 30
## 18 22 23 21 18 15 10
```

#### 7.2.1.6 get.subsets

This function creates a dispRity object that contains only elements from one specific subsets.

```
## Extracting all the data for the crown mammals
(crown_mammals <- get.subsets(disp_crown_stemBS, "Group.crown"))

## The object keeps the properties of the parent object but is composed of only one su
length(crown_mammals$subsets)
```

#### 7.2.1.7 combine.subsets

This function allows to merge different subsets.

```
## Combine the two first subsets in the dispRity data example
combine.subsets(disparity, c(1,2))
```

Note that the computed values (bootstrapped data + disparity metric) are not merge.

#### 7.2.1.8 extract.dispRity

This function extracts the calculated disparity values of a specific matrix.

```
## Extracting the observed disparity (default)
extract.dispRity(disparity)

## Extracting the disparity from the bootstrapped values from the
## 10th rarefaction level from the second subsets (80 Mya)
extract.dispRity(disparity, observed = FALSE, subsets = 2, rarefaction = 10)
```

#### 7.2.1.9 rescale.dispRity

This is the modified S3 method for `scale` (scaling and/or centring) that can be applied to the disparity data of a `dispRity` object and can take optional arguments (for example the rescaling by dividing by a maximum value).

```
## Getting the disparity values of the time subsets
head(summary(disparity))

## Scaling the same disparity values
head(summary(rescale.dispRity(disparity, scale = TRUE)))

## Scaling and centering:
head(summary(rescale.dispRity(disparity, scale = TRUE, center = TRUE)))

## Rescaling the value by dividing by a maximum value
head(summary(rescale.dispRity(disparity, max = 10)))
```

#### 7.2.1.10 sort.dispRity

This is the S3 method of `sort` for sorting the subsets alphabetically (default) or following a specific pattern.

```
## Sorting the disparity subsets in inverse alphabetic order
head(summary(sort(disparity, decreasing = TRUE)))

## Customised sorting
head(summary(sort(disparity, sort = c(7, 1, 3, 4, 5, 2, 6))))
```

#### 7.2.1.11 get.tree add.tree and remove.tree

These functions allow to manipulate the potential tree components of `dispRity` objects.

```
## Getting the tree component of a dispRity object
get.tree(disparity)

## Removing the tree
remove.tree(disparity)

## Adding a tree
add.tree(disparity, tree = BeckLee_tree)
```

## 7.3 The dispRity object content

The functions above are utilities to easily and safely access different elements in the `dispRity` object. Alternatively, of course, each elements can be accessed manually. Here is an explanation on how it works. The `dispRity` object is a `list` of two to four elements, each of which are detailed below:

- `$matrix`: an object of class `list` that contains at least one object of class `matrix`: the full multidimensional space.
- `$call`: an object of class `list` containing information on the `dispRity` object content.
- `$subsets`: an object of class `list` containing the subsets of the multidimensional space.
- `$disparity`: an object of class `list` containing the disparity values.

The `dispRity` object is loosely based on `C` structure objects. In fact, it is composed of one unique instance of a matrix (the multidimensional space) upon which the metric function is called via "pointers" to only a certain number of elements and/or dimensions of this matrix. This allows for: (1) faster and easily tractable execution time: the metric functions are called through apply family function and can be parallelised; and (2) a really low memory footprint: at any time, only one matrix (or list of matrices) is present in the `R` environment rather than multiple copies of it for each subset.

### 7.3.1 $matrix

This is the multidimensional space, stored in the `R` environment as a `list` object containing one or more `matrix` objects. Each `matrix` requires row names but not column names (optional). By default, if the row names are missing, `dispRity` function will arbitrarily generate them in numeric order (i.e. `rownames(matrix) <- 1:nrow(matrix)`). This element of the `dispRity` object is never modified.

### 7.3.2 $call

This element contains the information on the `dispRity` object content. It is a `list` that can contain the following:

- `$call$subsets`: a vector of `character` with information on the subsets type (either `"continuous"`, `"discrete"` or `"custom"`), their eventual model (`"acctran"`, `"deltran"`, `"random"`, `"proximity"`, `"equal.split"`, `"gradual.split"`) and eventual information about the trees and matrices used through `chrono.subsets`. This element generated only once via `chrono.subsets()` and `custom.subsets()`.
- `$call$dimensions`: either a single `numeric` value indicating how many dimensions to use or a vector of `numeric` values indicating which specific

dimensions to use. This element is by default the number of columns in `$matrix` but can be modified through `boot.matrix()` or `dispRity()`.

- `$call$bootstrap`: this is a `list` containing three elements:
  - `[[1]]`: the number of bootstrap replicates (`numeric`)
  - `[[2]]`: the bootstrap method (`character`)
  - `[[3]]`: the rarefaction levels (`numeric` vector)
- `$call$disparity`: this is a `list` containing one element, `$metric`, that is a `list` containing the different functions passed to the `metric` argument in `dispRity`. These are `call` elements and get modified each time the `dispRity` function is used (the first element is the first metric(s), the second, the second metric(s), etc.).

### 7.3.3 $subsets

This element contain the eventual subsets of the multidimensional space. It is a `list` of subset names. Each subset name is in turn a `list` of at least one element called `elements` which is in turn a `matrix`. This `elements` matrix is the raw (observed) elements in the subsets. The `elements` matrix is composed of `numeric` values in one column and $n$ rows (the number of elements in the subset). Each of these values are a "pointer" (`C` inspired) to the element of the `$matrix`. For example, lets assume a `dispRity` object called `disparity`, composed of at least one subsets called `sub1`:

```
disparity$subsets$sub1$elements
     [,1]
[1,]    5
[2,]    4
[3,]    6
[4,]    7
```

The values in the matrix "point" to the elements in `$matrix`: here, the multidimensional space with only the 4th, 5th, 6th and 7th elements. The following elements in `diparity$subsets$sub1` will correspond to the same "pointers" but drawn from the bootstrap replicates. The columns will correspond to different bootstrap replicates. For example:

```
disparity$subsets$sub1[[2]]
     [,1] [,2] [,3] [,4]
[1,]   57   43   70    4
[2,]   43   44    4    4
[3,]   42   84   44    1
[4,]   84    7    2   10
```

This signifies that we have four bootstrap pseudo-replicates pointing each time to four elements in `$matrix`. The next element (`[[3]]`) will be the same for the eventual first rarefaction level (i.e. the resulting bootstrap matrix will have $m$ rows where $m$ is the number of elements for this rarefaction level). The next

element after that (`[[4]]`) will be the same for with an other rarefaction level and so forth…

When a probabilistic model was used to select the elements (models that have the `"split"` suffix, e.g. `chrono.subsets(..., model = "gradual.split")`), the `$elements` is a matrix containing a pair of elements of the matrix and a probability for sampling the first element in that list:

```
disparity$subsets$sub1$elements
     [,1] [,2]       [,3]
[1,]   73   36 0.01871893
[2,]   74   37 0.02555876
[3,]   33   38 0.85679821
```

In this example, you can read the table row by row as: "there is a probability of `0.018` for sampling element `73` and a probability of `0.82` (`1-0.018`) of sampling element `36`".

### 7.3.4 $disparity

The `$disparity` element is identical to the `$subsets` element structure (a list of list(s) containing matrices) but the matrices don't contain "pointers" to `$matrix` but the disparity result of the disparity metric applied to the "pointers". For example, in our first example (`$elements`) from above, if the disparity metric is of dimensions level 1, we would have:

```
disparity$disparity$sub1$elements
     [,1]
[1,]    1.82
```

This is the observed disparity (1.82) for the subset called `sub1`. If the disparity metric is of dimension level 2 (say the function `range` that outputs two values), we would have:

```
disparity$disparity$sub1$elements
     [,1]
[1,]    0.82
[2,]    2.82
```

The following elements in the list follow the same logic as before: rows are disparity values (one row for a dimension level 1 metric, multiple for a dimensions level 2 metric) and columns are the bootstrap replicates (the bootstrap with all elements followed by the eventual rarefaction levels). For example for the bootstrap without rarefaction (second element of the list):

```
disparity$disparity$sub1[[2]]
        [,1]     [,2]     [,3]     [,4]
[1,] 1.744668 1.777418 1.781624 1.739679
```

# Bibliography


Antonio Aguilera and Ricardo Pérez-Aguila. General n-dimensional rotations. 2004. URL http://wscg.zcu.cz/wscg2004/Papers_2004_Short/N29.pdf.

Robin M Beck and Michael S Lee. Ancient dates or accelerated rates? Morphological clocks and the antiquity of placental mammals. *Proceedings of the Royal Society B: Biological Sciences*, 281(20141278):1–10, 2014. doi: 10.1098/rspb.2014.1278. URL http://dx.doi.org/10.1098/rspb.2014.1278.

Martin D Brazeau, Thomas Guillerme, and Martin R Smith. An algorithm for Morphological Phylogenetic Analysis with Inapplicable Data. *Systematic Biology*, 68(4):619–631, 12 2018. ISSN 1063-5157. doi: 10.1093/sysbio/syy083. URL https://doi.org/10.1093/sysbio/syy083.

Natalie Cooper, Gavin H. Thomas, Chris Venditti, Andrew Meade, and Rob P. Freckleton. A cautionary note on the use of ornstein uhlenbeck models in macroevolutionary studies. *Biological Journal of the Linnean Society*, 118 (1):64–77, 2016. doi: 10.1111/bij.12701. URL http://dx.doi.org/10.1111/bij.12701.

Sandra Díaz, Jens Kattge, Johannes HC Cornelissen, Ian J Wright, Sandra Lavorel, Stéphane Dray, Björn Reu, Michael Kleyer, Christian Wirth, I Colin Prentice, et al. The global spectrum of plant form and function. *Nature*, 529 (7585):167, 2016. URL http://dx.doi.org/10.1038/nature16489.

O'Reilly Joseph E., Puttick Mark N., Pisani Davide, and Donoghue Philip C. J. Probabilistic methods surpass parsimony when assessing clade support in phylogenetic analyses of discrete morphological data. *Palaeontology*, 61 (1):105–118. doi: 10.1111/pala.12330. URL http://dx.doi.org/10.1111/pala.12330.

John A Endler, David A Westcott, Joah R Madden, and Tim Robson. Animal visual systems and the evolution of color patterns: sensory processing illuminates signal evolution. *Evolution*, 59(8):1795–1818, 2005.

Richard G. FitzJohn. Diversitree: comparative phylogenetic analyses of diversification in R. *Methods in Ecology and Evolution*, 3(6):1084–1092, 2012. ISSN 2041-210X. doi: 10.1111/j.2041-210X.2012.00234.x. URL http://dx.doi.org/10.1111/j.2041-210X.2012.00234.x.

T. Guillerme and N. Cooper. Time for a rethink: time sub-sampling methods in disparity-through-time analyses. *Palaeontology*, 61(4):481–493, 2018. doi: 10.1111/pala.12364. URL http://dx.doi.org/10.1111/pala.12364.

Thomas Guillerme and Natalie Cooper. Effects of missing data on topological inference using a Total Evidence approach. *Molecular Phylogenetics and Evolution*, 94, Part A:146–158, 2016. ISSN 1055-7903. doi: http://dx.doi.org/10.1016/j.ympev.2015.08.023. URL http://dx.doi.org/10.1016/j.ympev.2015.08.023.

Thomas Guillerme, Natalie Cooper, Stephen L. Brusatte, Katie E. Davis, Andrew L. Jackson, Sylvain Gerber, Anjali Goswami, Kevin Healy, Melanie J. Hopkins, Marc E. H. Jones, Graeme T. Lloyd, Joseph E. O'Reilly, Abi Pate, Mark N. Puttick, Emily J. Rayfield, Erin E. Saupe, Emma Sherratt, Graham J. Slater, Vera Weisbecker, Gavin H. Thomas, and Philip C. J. Donoghue. Disparities in the analysis of morphological disparity. *Biology Letters*, 16(7):20200199, 2020a. doi: 10.1098/rsbl.2020.0199. URL https://royalsocietypublishing.org/doi/abs/10.1098/rsbl.2020.0199.

Thomas Guillerme, Mark N Puttick, Ariel E Marcy, and Vera Weisbecker. Shifting spaces: Which disparity or dissimilarity measurement best summarize occupancy in multidimensional spaces? *Ecology and Evolution*, 2020b.

M. Hasegawa, H. Kishino, and T. A. Yano. Dating of the human ape splitting by a molecular clock of mitochondrial-DNA. *Journal of Molecular Evolution*, 22(2):160–174, 1985.

Gene Hunt. Fitting and comparing models of phyletic evolution: random walks and beyond. *Paleobiology*, 32(4):578–601, 2006. URL https://doi.org/10.1666/05070.1.

Gene Hunt. Measuring rates of phenotypic evolution and the inseparability of tempo and mode. *Paleobiology*, 38(3):351–373, 2012. URL https://doi.org/10.1666/11047.1.

Gene Hunt, Melanie J Hopkins, and Scott Lidgard. Simple versus complex models of trait evolution and stasis as a response to environmental change. *Proceedings of the National Academy of Sciences*, page 201403662, 2015. URL https://doi.org/10.1073/pnas.1403662111.

P. Lewis. A likelihood approach to estimating phylogeny from discrete morphological character data. *Systematic Biology*, 50(6):913–925, 2001. doi: 10.1080/106351501753462876. URL http://dx.doi.org/10.1080/106351501753462876.

David J Murrell. A global envelope test to detect non-random bursts of trait evolution. *Methods in Ecology and Evolution*, 9(7):1739–1748, 2018. URL https://doi.org/10.1111/2041-210X.13006.

Joseph E. O'Reilly, Mark N. Puttick, Luke Parry, Alastair R. Tanner, James E. Tarver, James Fleming, Davide Pisani, and Philip C. J. Donoghue. Bayesian methods outperform parsimony but at the expense of precision in the es-

timation of phylogeny from discrete morphological data. *Biology Letters*, 12(4), 2016. ISSN 1744-9561. doi: 10.1098/rsbl.2016.0081. URL http://dx.doi.org/10.1098/rsbl.2016.0081.

Mark N Puttick, Joseph E O'Reilly, Alastair R Tanner, James F Fleming, James Clark, Lucy Holloway, Jesus Lozano-Fernandez, Luke A Parry, James E Tarver, Davide Pisani, et al. Uncertain-tree: discriminating among competing approaches to the phylogenetic analysis of phenotype data. *Proceedings of the Royal Society B*, 284(1846):20162290, 2017. URL http://dx.doi.org/10.1098/rspb.2016.2290.