dispRity manual

Thomas Guillerme (guillert@tcd.ie) and Natalie Cooper (natalie.cooper@nhm.ac.uk)

2017-11-15

Contents

Chapter 1

dispRity

This is a package for measuring disparity in R. It allows users to summarise matrices as representations as multidimensional spaces (namely from ordinated matrices from MDS, PCA, PCO, PCoA) into a single value or distribution describing a specific aspect of this multidimensional space (the disparity). This manual is based on the version 0.4.

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 if you use the latest version of R (> 3.4.0) and devtools.

```
## 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")

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

Note this uses the `release` branch (version`0.4`). 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 Why not CRAN?

This package is not available on CRAN. This is mainly because some parts are still in development and that the reactivity of GitHub is better for implementing new suggestions from users. However, the package follows the strict (and useful!) CRAN standards via Travis.

1.4 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.

1.5 Citations

You can cite both the package or this manual with the following citation:

> Guillerme, T. (2016). dispRity: a package for measuring disparity in R. Zenodo. 10.5281/zenodo.55646

Note that this citation is only temporary (but can still be used!). A proper description of the package is currently in review in Methods in Ecology and Evolution and should be re-submitted this winter. Upon acceptance (hopefully!), a proper DOI will be released for this manual and the package description.

Chapter 2

Glossary

- **Multidimensional 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.

Chapter 3

Getting started with dispRity

3.1 What sort of data does dispRity work with?

Disparity can be estimated from pretty much any matrix. Classically, however, it is measured from ordinated matrices. These matrices can be from any type of ordination (PCO, PCA, PCoA, MDS, etc.) as long as they have your element names as rows (taxa, experiments, countries etc.) and your ordination axes as columns (the dimensions of your dataset).

3.1.1 Ordination matrices from Claddis

dispRity package can easily take data from Claddis 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(Michaux1989)
```

```
##                       [,1]          [,2]       [,3]
## Ancilla       7.252259e-17  4.154578e-01  0.2534942
## Turrancilla -5.106645e-01 -4.566150e-16 -0.2534942
## Ancillista   5.106645e-01 -8.153839e-16 -0.2534942
## Amalda      -3.207162e-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::MorphDistMatrix and stats::cmdscale.

3.1.2 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,] -0.0369931363 0.05118247 -0.0016971082 -0.003128809 -0.010936371
## [2,] -0.0007493738 0.05942082  0.0001371715 -0.002768680 -0.008117383
## [3,]  0.0056004654 0.07419599 -0.0052612103 -0.005034566 -0.002746592
## [4,] -0.0134808572 0.06463959 -0.0458436015 -0.007887369  0.009816827
## [5,] -0.0334696244 0.06863518  0.0136292041  0.007359409  0.022347225
```

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:
##     species.Jord, species.Teyah, site.Allo, site.Symp.
```

More about these `dispRity` objects below!

3.1.3 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)
```

```
## Class 'dist'  atomic [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.

3.2 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 `time.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.

3.2.1 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   -0.5319679  0.1117759259  0.09865194 -0.1933148 0.2035833
## Maelestes    -0.4087147  0.0139690317  0.26268300  0.2297096 0.1310953
## Batodon      -0.6923194  0.3308625215 -0.10175223 -0.1899656 0.1003108
## Bulaklestes  -0.6802291 -0.0134872777  0.11018009 -0.4103588 0.4326298
## Daulestes    -0.7386111  0.0009001369  0.12006449 -0.4978191 0.4741342
## Uchkudukodon -0.5105254 -0.2420633915  0.44170317 -0.1172972 0.3602273
```

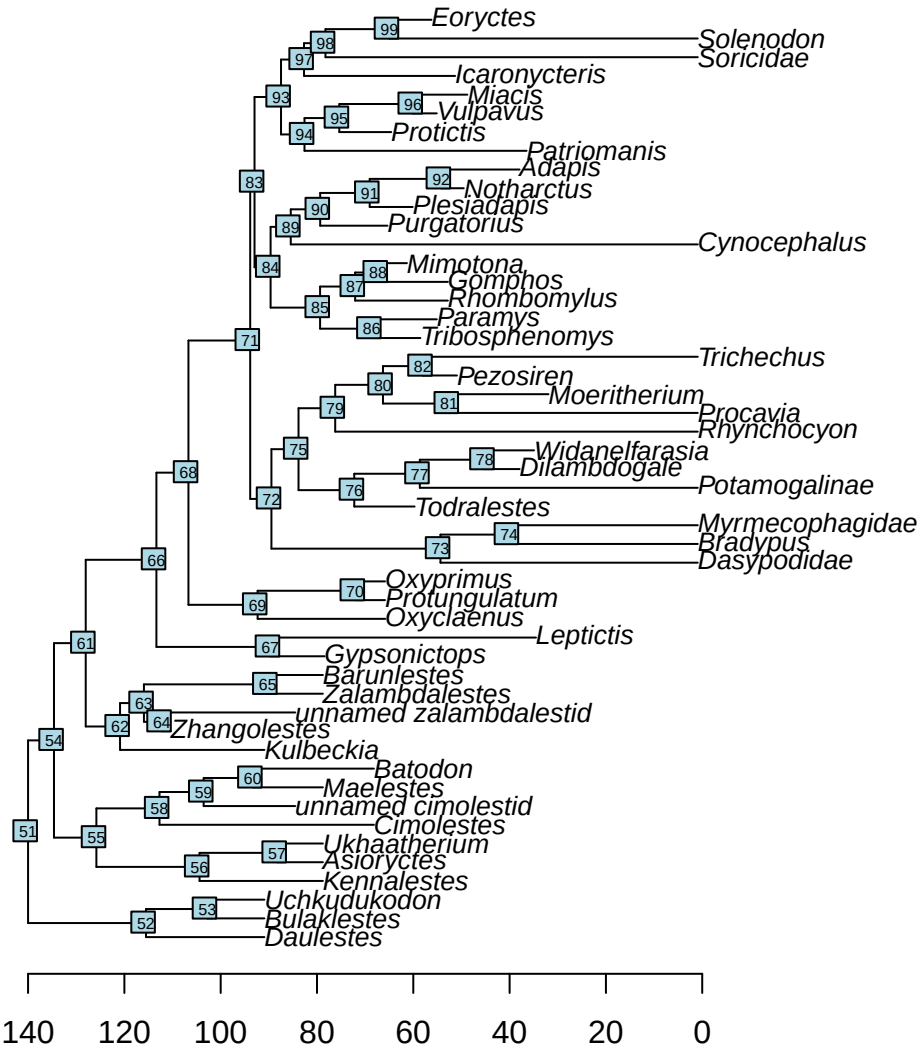
```
## The first five taxa and dimensions of the 99 taxa + ancestors matrix
BeckLee_mat99[c(1, 2, 98, 99), 1:5]
```

```
##                   [,1]       [,2]       [,3]        [,4]        [,5]
## Cimolestes -0.60824375 -0.0323683 0.08458885 -0.43384481 -0.30536875
## Maelestes  -0.57302058 -0.2840361 0.01308847 -0.12588477  0.06123611
## n48        -0.05529018  0.4799330 0.04118477  0.04944912 -0.35588301
## n49        -0.13067785  0.4478168 0.11956268  0.13800340 -0.32227852
```

```
## 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 chapter.

3.2.2 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,
                                        time = 3, metric = c(sum, variances))
```

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 with 48 dimensions:
##     133.51104 - 89.00736, 89.00736 - 44.50368, 44.50368 - 0.
## Data was bootstrapped 100 times (method:"full").
## Disparity was calculated as: metric.
```

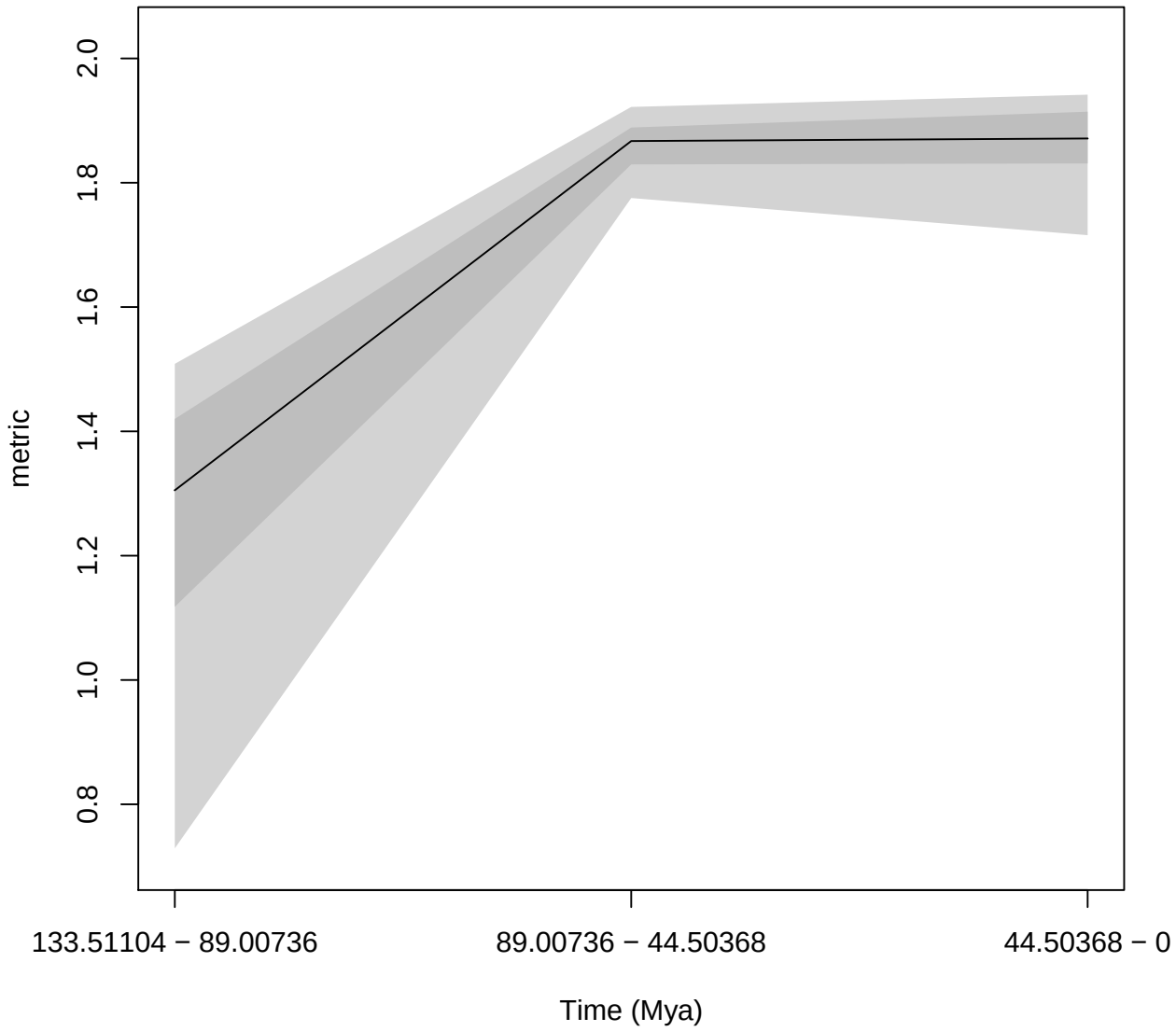
We asked for three subsets (evenly spread across the age of the tree), the data was bootstrapped 100 times (default) and the metric used was the sum of variances.

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%
## 1 133.51104 - 89.00736  5 1.575     1.305 0.729 1.118 1.420 1.509
## 2  89.00736 - 44.50368 29 1.922     1.867 1.775 1.830 1.889 1.922
## 3         44.50368 - 0 16 1.990     1.871 1.716 1.831 1.914 1.942
```

```
## 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")
```

```
## Warning in test.dispRity(disparity_data, test = lm, comparisons = "all"): Multiple p-values will be calcu
## This will inflate Type I error!
```

```
summary(disp_lm)
```

```
##
## Call:
## test(formula = data ~ subsets, data = data)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -0.56623 -0.04160  0.01049  0.05507  0.31886
##
## Coefficients:
##                            Estimate Std. Error t value Pr(>|t|)
## (Intercept)                 1.25647    0.01270   98.97   <2e-16 ***
## subsets44.50368 - 0         0.60863    0.01795   33.90   <2e-16 ***
```

```
## subsets89.00736 - 44.50368  0.60169    0.01795   33.51   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.127 on 297 degrees of freedom
## Multiple R-squared:  0.8361, Adjusted R-squared:  0.835
## F-statistic: 757.5 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.

3.2.3 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 <- list("crown" = c(16, 19:41, 45:50),
                      "stem" = c(1:15, 17:18, 42:44))

## 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).

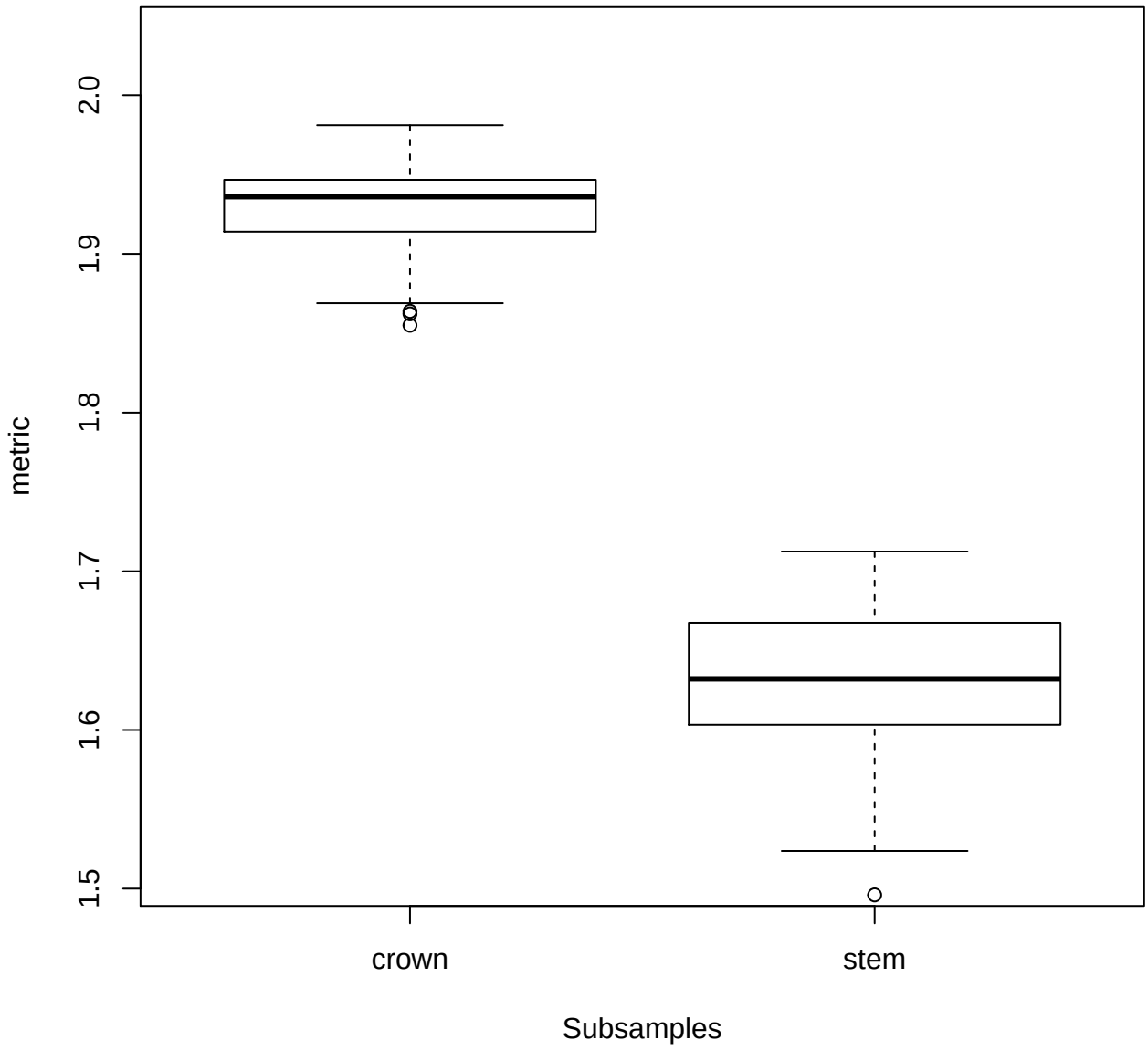
```
## Print the disparity_data object
disparity_data
```

```
##  ---- dispRity object ----
## 2 customised subsets for 50 elements 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 1.995     1.936 1.866 1.914 1.947 1.971
## 2    stem 20 1.715     1.632 1.541 1.604 1.667 1.695
```

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




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

```
## Warning in test.dispRity(disparity_data, test = wilcox.test, details = TRUE): Multiple p-values will be c
## This will inflate Type I error!
## $`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 `time.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 Cooper and Guillerme (in prep.). 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.

Here is an example for `method = discrete`:

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

```
##  ---- dispRity object ----
## 3 discrete time subsets for 50 elements:
```

```
##     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:
time.subsets(data = BeckLee_mat50, tree = BeckLee_tree, method = "discrete",
                time = 3)
```

```
##  ---- dispRity object ----
## 3 discrete time subsets for 50 elements:
##     133.51104 - 89.00736, 89.00736 - 44.50368, 44.50368 - 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
time.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:
##     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.

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:

- `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
- `punctuated` where the data chosen along the branch is both the descendant and the ancestor with an even probability
- `gradual` 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 "fixed": the selected data is always either the one of the descendant or the ancestor. The two last models are "probabilistic": 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.

```
## Generating four time slices every 40 million years under a model of proximity evolution
time.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:
##     120, 80, 40, 0.
```

```
## Generating four time slices automatically
time.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:
##     133.51104, 89.00736, 44.50368, 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 as a list
mammal_groups <- list("crown" = c(16, 19:41, 45:50),
                      "stem" = c(1:15, 17:18, 42:44))

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

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

Elements can easily be assigned to different groups if necessary!

```
## Creating the three groups as a list
mammal_groups <- list("crown" = c(16, 19:41, 45:50),
                      "stem" = c(1:15, 17:18, 42:44).
                      "all" = c(1:50))
```

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 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 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 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 with 48 dimensions.
## Data was bootstrapped 20 times (method:"full") and rarefied to 6, 7, 8, 3 elements.
```

One additional important 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 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 with 10 dimensions.
## Data was bootstrapped 100 times (method:"full").
```

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

```
## Creating subsets of crown and stem mammals
crown_stem <- custom.subsets(BeckLee_mat50,
                             group = list("crown" = c(16, 19:41, 45:50),
                                          "stem" = c(1:15, 17:18, 42:44)))
## Bootstrapping and rarefying these groups
boot.matrix(crown_stem, bootstraps = 200, rarefaction = TRUE)
```

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

```
## Creating time slice subsets
time_slices <- time.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 with 97 dimensions:
##     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.

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).

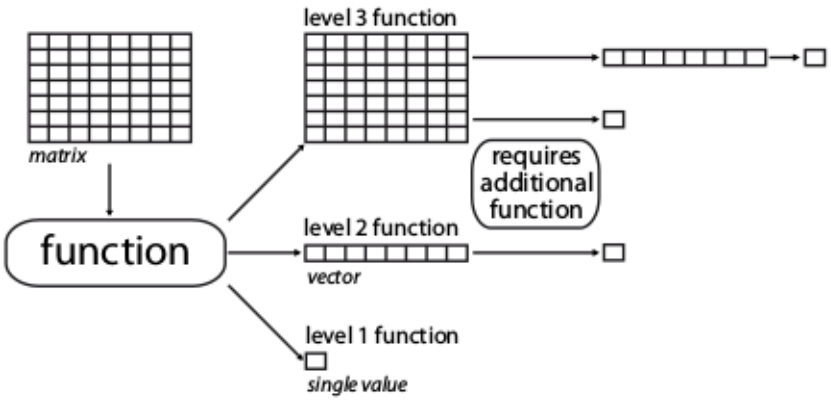


Figure 4.1: Illustration of the different dimension-levels of functions with an input `matrix`

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:

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

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

```
## [1] 0.3873238
```

```
median(dummy_matrix)
```

```
## [1] 0.8189937
```

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] -8.0595618 -0.5106728 -1.3381023  0.1982694
```

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,]  0.7722329 -1.232395  0.5044684
## [2,] -1.2323952  5.258134 -2.5675598
## [3,]  0.5044684 -2.567560  1.8322537
```

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

```
##          1        2        3        4
## 1 0.000000 5.096201 5.831904 4.944653
## 2 5.096201 0.000000 2.058924 1.534373
## 3 5.831904 2.058924 0.000000 1.823741
## 4 4.944653 1.534373 1.823741 0.000000
```

4.4.2 `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.")
}
## Testing whether var is dimension-level 1
if(make.metric(var, silent = TRUE) != "level1") {
    message("The metric is not dimension-level 1.")
}
```

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

4.4.3 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 0.201
```

```
## 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 0.028
```

```
## 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 0.0001025857
```

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.

4.4.4 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
1	`ellipse.volume`	The volume of the ellipsoid of the space	Donohue *et al.* (2013)
1	`convhull.surface`	The surface of the convex hull formed by all the elements	`geometry::convhulln`
1	`convhull.volume`	The volume of the convex hull formed by all the elements	`geometry::convhulln`

1 | `diagonal` | The longest distance in the ordinated space (like the diagonal in two dimensions) | `dispRity` | 2 | `ranges` | The range of each dimension | `dispRity` | 2 | `variances` | The variance of each dimension | `dispRity` | 2 | `centroids`2 | The distance between each element and the centroid of the ordinated space | `dispRity` | 1 | `mode.val` | The modal value | `dispRity` |

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

2: 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.

4.4.5 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).

$$ellipse.volume = \frac{\pi^{k/2}}{\Gamma(\frac{k}{2}+1)} \prod_{i=1}^{k} (\lambda_i^{0.5}) \tag{4.1}$$

Where k is the number of dimensions, and λ_i is the eigenvalue of each dimension.

$$diagonal = \sqrt{\sum_{i=1}^{k} |max(k_i) - min(k_i)|} \tag{4.2}$$

Where k is the number of dimensions.

$$ranges = |max(k_i) - min(k_i)| \tag{4.3}$$

Where k is the number of dimensions.

$$variances = \sigma^2 k_i \tag{4.4}$$

Where k is the number of dimensions, and σ^2 is their variance.

$$centroids = \sqrt{\sum_{i=1}^{n} (k_n - Centroid_k)^2} \tag{4.5}$$

Where n is each element in the ordinated space, k is the number of dimensions, and $Centroid_k$ is their mean (or can be set to another value).

4.4.6 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)
```

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

4.4.6.1 Volumes and surface metrics

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

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

```
##   subsets  n   obs
## 1       1 10 257.8
```

> Because there is only one subset (i.e. one matrix) in the dispRity object, this operation is the equivalent of `ellipse.volume(dummy_space)` (with rounding).

```
## 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
```

The convex hull functions make a (good) estimation of the multidimensional properties of the ordinated space.

> 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.

4.4.6.2 Ranges, variances and diagonal

The functions `ranges`, `variances` 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` and `variances` both work on the same principle and measure the range/variance 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
```

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):

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

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

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

4.4.6.3 Centroids metric

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 = c(0,0,0,0,0)))
```

```
##   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
```

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 = list("crown" = c(16, 19:41, 45:50),
                                             "stem" = c(1:15, 17:18, 42:44)))
## 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 <- time.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 <- time.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  6 2.899     2.399 1.749 2.202 2.642 2.794
## 2      80 22 3.558     3.414 3.127 3.314 3.463 3.557
## 3      40 15 4.032     3.764 3.569 3.694 3.842 3.954
## 4       0 10 4.055     3.661 3.282 3.564 3.771 3.927
```

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 1.995     1.930 1.866 1.909 1.950 1.970
## 2   crown 29    NA     1.932 1.871 1.915 1.953 1.974
## 3   crown 28    NA     1.926 1.852 1.903 1.942 1.978
## 4   crown 27    NA     1.935 1.860 1.908 1.954 1.982
## 5   crown 26    NA     1.931 1.859 1.908 1.953 1.977
## 6   crown 25    NA     1.933 1.855 1.909 1.954 1.978
```

The summary functions can also take various options such as:

- `quantile` 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`)
- rounding`option corresponding to the number of decimal places to print (default is2‘)`
- `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, quantile = 88, cent.tend = sd)
```

```
##   subsets  n   obs bs.sd    6%   94%
## 1     120  6 2.899 0.293 1.917 2.762
## 2      80 22 3.558 0.113 3.208 3.542
## 3      40 15 4.032 0.109 3.612 3.918
## 4       0 10 4.055 0.169 3.387 3.882
```

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

```
##  ---- dispRity object ----
## 4 continuous (proximity) time subsets for 99 elements with 97 dimensions:
##     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  6 2.89926   2.39864 1.74855 2.20230 2.64153 2.79428
## 2      80 22 3.55782   3.41432 3.12737 3.31353 3.46313 3.55661
## 3      40 15 4.03191   3.76393 3.56890 3.69427 3.84171 3.95368
## 4       0 10 4.05457   3.66131 3.28211 3.56431 3.77130 3.92706
```

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`).

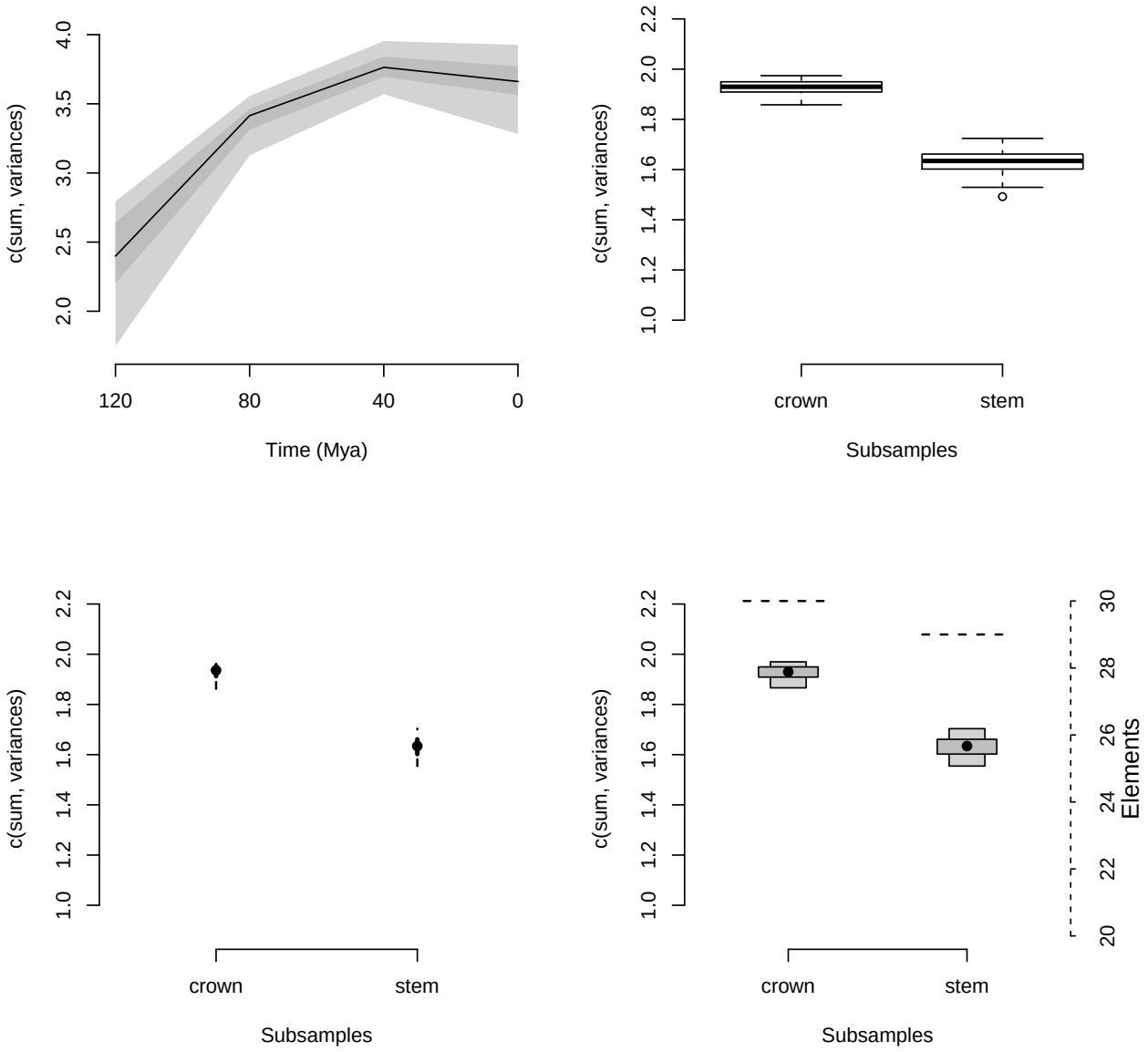
The plots can be of four different types:

- `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)
```

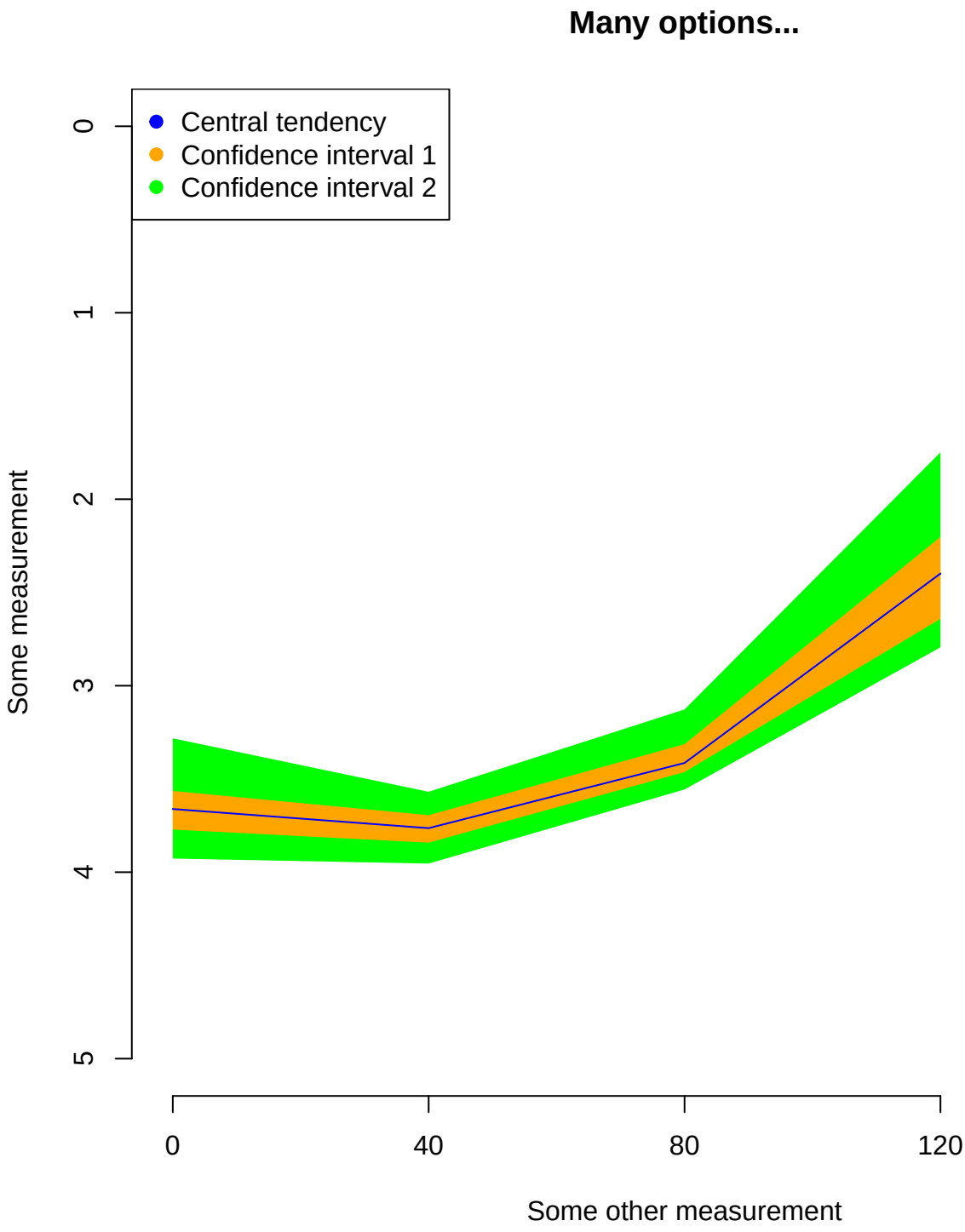



```
## Resetting graphical parameters
par(op)
```

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(5, 0), xlim = c(4, 0))

## Adding a legend
legend("topleft", legend = c("Central tendency",
                             "Confidence interval 1",
                            "Confidence interval 2"),
```

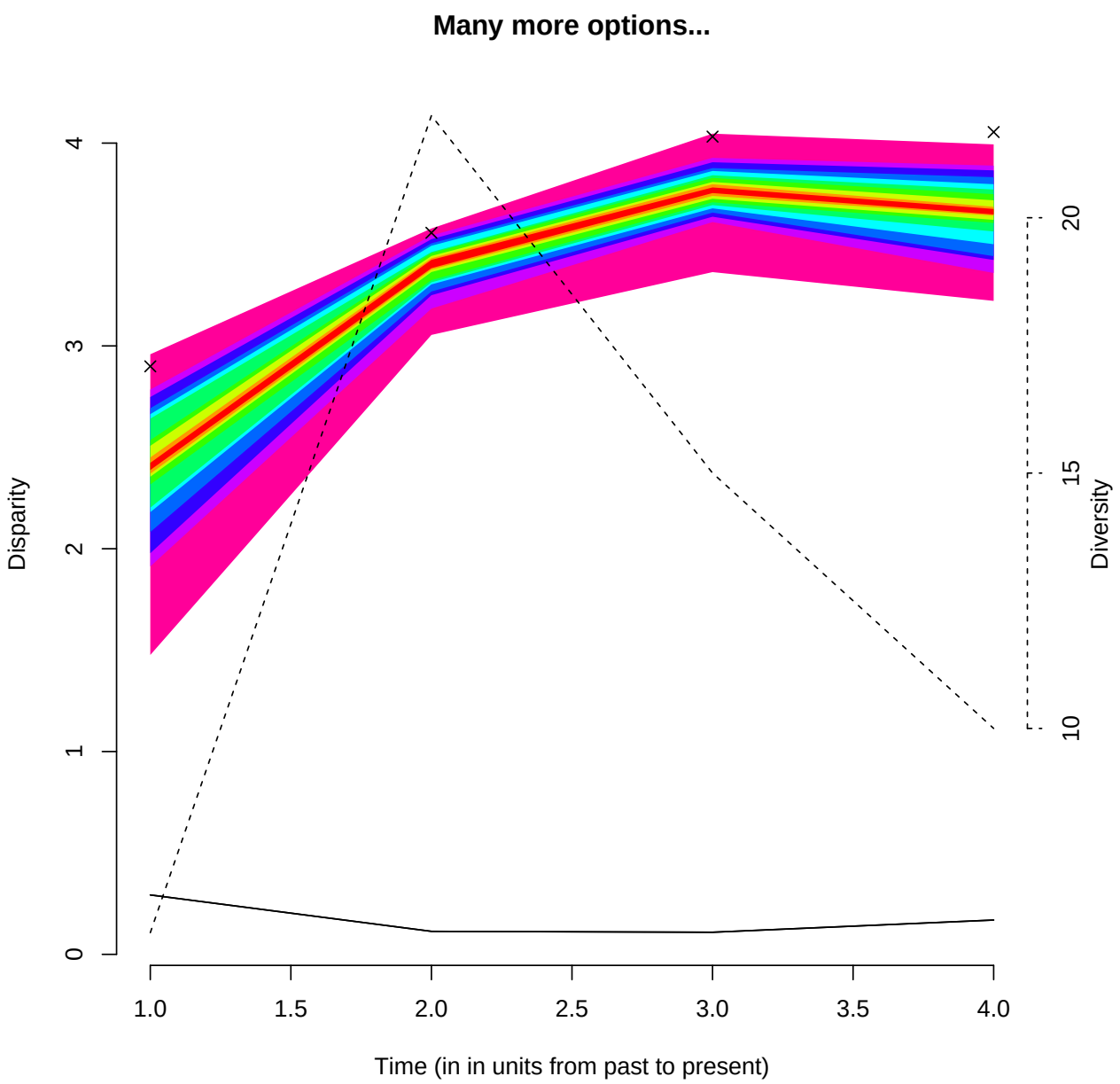
```
      col = c("blue", "orange", "green"), pch = 19)
```




```
## 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, quantile = c(seq(from = 10, to = 100, by = 10)),
    cent.tend = sd, type = "c", elements = TRUE, col = c("black", rainbow(10)),
    ylab = c("Disparity", "Diversity"), time.subsets = FALSE,
    xlab = "Time (in in units from past to present)", observed = TRUE,
    main = "Many more options...")
```

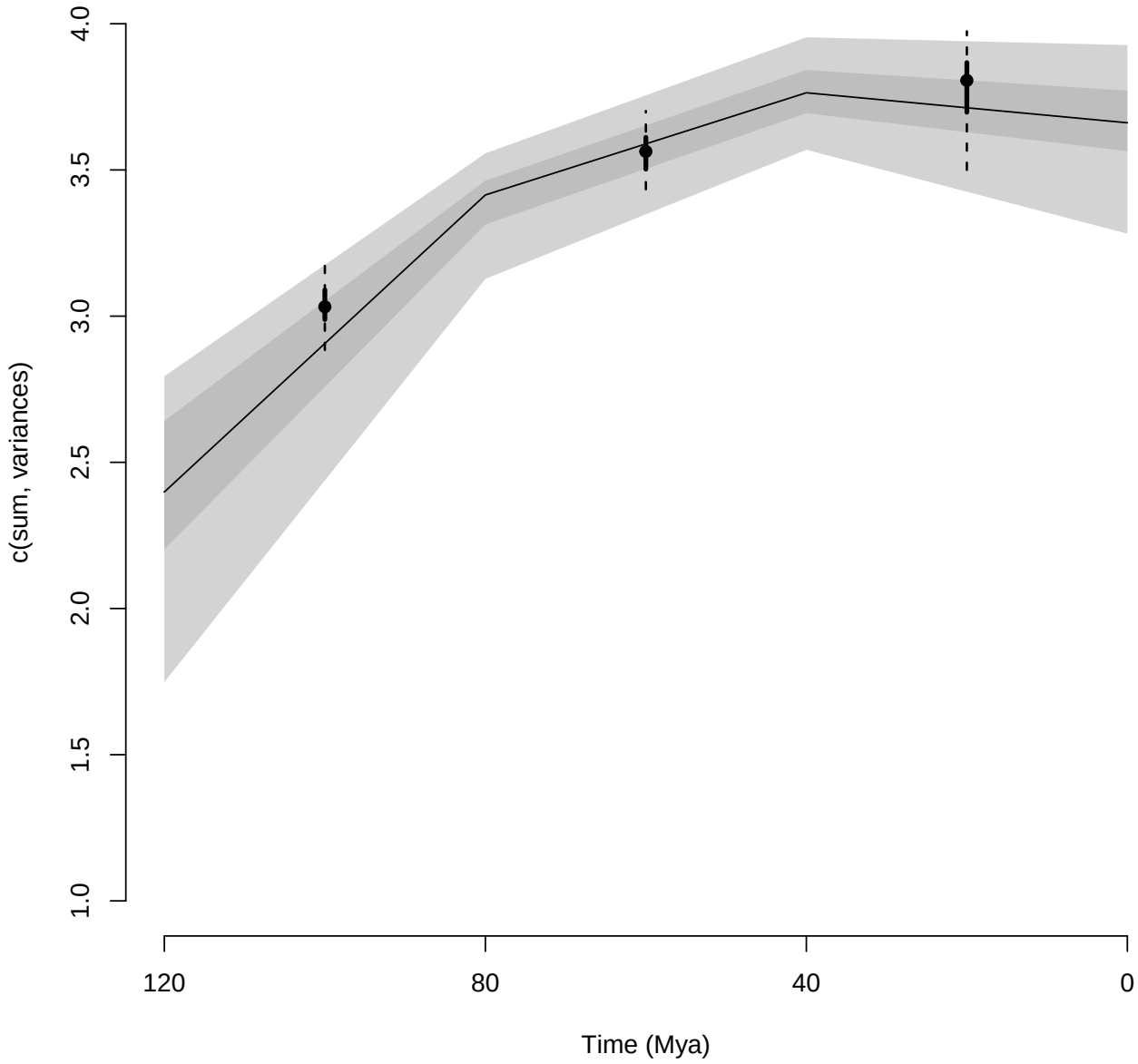



```
## Resetting graphical parameters
par(op)
```

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(1, 4))
## Adding the discrete data
plot(disparity_time_bins, type = "line", ylim = c(1, 4), xlab = "", ylab = "",
    add = TRUE)
```

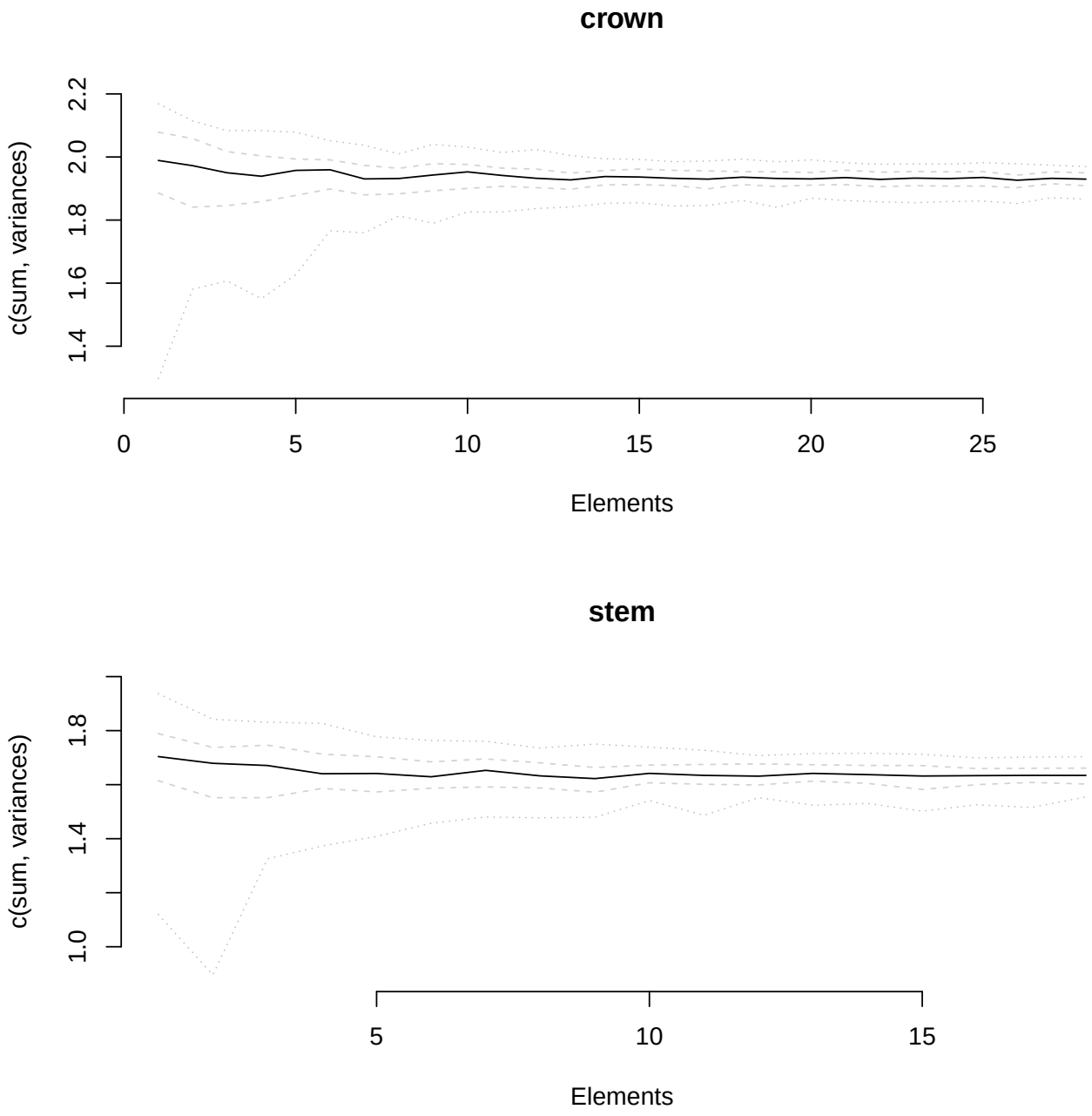



```
## Resetting graphical parameters
par(op)
```

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)
```

4.6 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)
```

```
## Warning in test.dispRity(disparity_crown_stem, test = t.test): Multiple p-values will be calculated with
## This will inflate Type I error!
```

```
## [[1]]
##              statistic
## crown : stem  57.80615
##
## [[2]]
##              parameter
## crown : stem  168.7267
##
## [[3]]
##                     p.value
## crown : stem 3.927248e-113
```

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

```
## Warning in test.dispRity(disparity_crown_stem, test = t.test, details = TRUE): Multiple p-values will be
## This will inflate Type I error!
```

```
## $`crown : stem`
## $`crown : stem`[[1]]
##
##  Welch Two Sample t-test
##
## data:  dots[[1L]][[1L]] and dots[[2L]][[1L]]
## t = 57.806, df = 168.73, p-value < 2.2e-16
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  0.2843010 0.3044058
## sample estimates:
```

```
## mean of x mean of y
##  1.926709  1.632356
```

```
## 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
## 120 : 80         0
## 80 : 40         74
## 40 : 0        7076
##
## [[2]]
##               p.value
## 120 : 80 7.673885e-34
## 80 : 40  6.986470e-33
## 40 : 0   1.185611e-06
```

```
## 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 will be calculated
## This will inflate Type I error!
```

```
##                     bhatt.coeff
## 120 - 80 : 80 - 40   0.0000000
## 120 - 80 : 40 - 0    0.0000000
## 80 - 40 : 40 - 0     0.4800631
```

It is also possible to apply some more *complex* tests that have their own output classes (like `stats::lm`).

The results can then be analysed as usual using the associated `summary` S3 method:

```
## Performing and linear model applied to the same data
(slice_lm <- test.dispRity(disparity_time_slices, test = lm,
                           comparisons = "all"))
```

```
## Warning in test.dispRity(disparity_time_slices, test = lm, comparisons = "all"): Multiple p-values will b
## This will inflate Type I error!
```

```
##
## Call:
## test(formula = data ~ subsets, data = data)
##
## Coefficients:
## (Intercept)   subsets120    subsets40    subsets80
##      3.6517      -1.2627       0.1145      -0.2634
```

```
## The output is a regular `lm` output
class(slice_lm)
```

```
## [1] "lm"
```

```
## This output can be summarised using summary
summary(slice_lm)
```

```
##
## Call:
```

```
## test(formula = data ~ subsets, data = data)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -0.91282 -0.08285  0.00993  0.10745  0.56989
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  3.65169    0.01865 195.810  < 2e-16 ***
## subsets120  -1.26273    0.02637 -47.878  < 2e-16 ***
## subsets40    0.11452    0.02637   4.342 1.79e-05 ***
## subsets80   -0.26339    0.02637  -9.987  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.1865 on 396 degrees of freedom
## Multiple R-squared:  0.8954, Adjusted R-squared:  0.8946
## F-statistic:  1130 on 3 and 396 DF,  p-value: < 2.2e-16
```

Of course, due to 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` for example).

4.7 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).

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  6      1.536     1.375 0.864 1.187 1.611 1.886
```

```
## 2      80 22      1.871     1.807 1.481 1.681 1.904 2.075
## 3      40 15      1.943     1.885 1.529 1.777 1.989 2.097
## 4       0 10      1.910     1.802 1.463 1.687 1.960 2.095
```

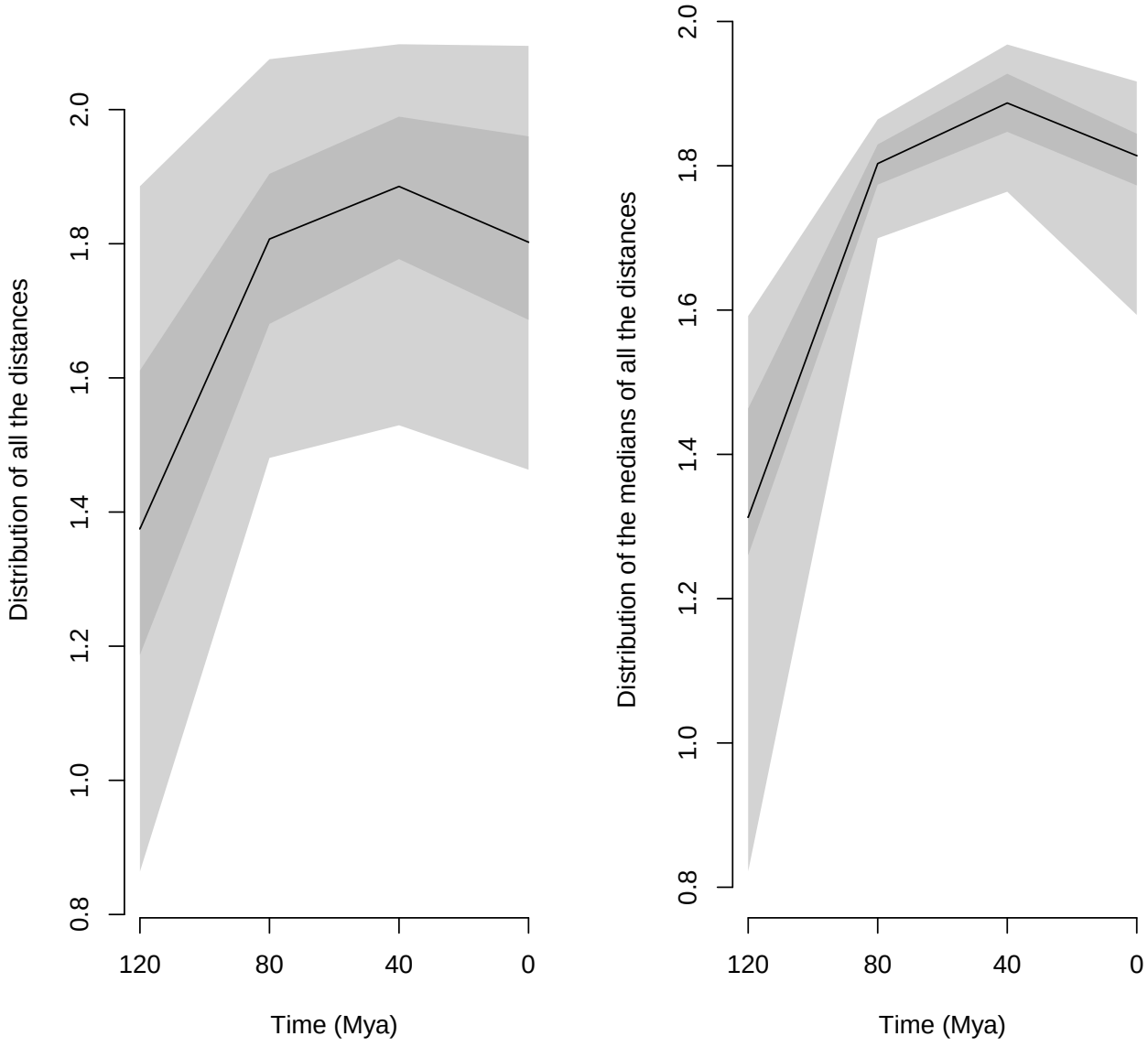
```
## The summary of the distributions (as median)
summary(disparity_centroids_median)
```

```
##   subsets  n   obs bs.median  2.5%   25%   75% 97.5%
## 1     120  6 1.536     1.313 0.822 1.260 1.464 1.592
## 2      80 22 1.871     1.803 1.700 1.774 1.830 1.864
## 3      40 15 1.943     1.887 1.764 1.847 1.928 1.968
## 4       0 10 1.910     1.814 1.593 1.773 1.844 1.917
```

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-values will be calc
## This will inflate Type I error!
```

```
##          bhatt.coeff
## 120 : 80  0.00000000
## 120 : 40  0.05537319
## 120 : 0   0.12649111
## 80 : 40   0.55897558
## 80 : 0    0.87452867
## 40 : 0    0.73959979
```

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 calculated
## This will inflate Type I error!
```

```
##          bhatt.coeff
## 120 : 80   0.5876931
## 120 : 40   0.4929308
## 120 : 0    0.6023832
## 80 : 40    0.9466307
## 80 : 0     0.9461665
## 40 : 0     0.9418949
```

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 5×100 distances (the distances of the five elements in slice 120 bootstrapped 100 times) to the 19×100 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-value
## This will inflate Type I error!
```

```
##          bhatt.coeff      2.5%       25%       75%     97.5%
## 120 : 80   0.2744178 0.0000000 0.1507557 0.3810388 0.5463043
## 120 : 40   0.1687997 0.0000000 0.0000000 0.2981424 0.4854356
## 120 : 0    0.2717451 0.0000000 0.1825742 0.3872983 0.6652158
## 80 : 40    0.6113891 0.2752409 0.5316275 0.7309501 0.8414439
## 80 : 0     0.5707989 0.2452170 0.4876771 0.6732360 0.8162527
## 40 : 0     0.5599584 0.1807321 0.3641941 0.7392429 0.9509206
```

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.

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]
## t15 "1"  "1"  "0"  "1"  "1"  "2"  "1"  "1"  "0"  "0"
## t12 "1"  "1"  "0"  "1"  "1"  "2"  "1"  "1"  "0"  "0"
## t14 "1"  "1"  "0"  "1"  "1"  "2"  "1"  "1"  "0"  "0"
## t6  "1"  "1"  "0"  "1"  "1"  "2"  "1"  "1"  "0"  "0"
## t3  "0"  "0"  "1"  "0"  "1"  "0"  "0"  "2"  "1"  "0"
```

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

```
##
## Maximum parsimony        139.0000000
## Consistency index          0.7625899
## Retention index            0.8881356
## Robinson-Foulds distance   0.0000000
```

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];OReilly2017].

- 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. 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]
## t15 "0"  "1"  "0"  "1"  "0"  "0"  "0"  "1"  "0"  "1"   "0"
```

```
## t12 "1"  "1"  "0"  "1"  "0"  "0"  "0"  "1"  "0"  "1"   "0"
## t14 "0"  "0"  "0"  "1"  "0"  "0"  "0"  "1"  "0"  "1"   "0"
## t6  "0"  "0"  "0"  "1"  "0"  "0"  "0"  "1"  "0"  "1"   "0"
## t3  "0"  "0"  "1"  "0"  "-"  "0"  "0"  "0"  "-"  "0"   "1"
## t2  "0"  "0"  "1"  "0"  "-"  "0"  "0"  "0"  "-"  "0"   "1"
## t11 "0"  "0"  "1"  "0"  "-"  "0"  "0"  "0"  "-"  "0"   "1"
## t7  "0"  "0"  "1"  "0"  "-"  "0"  "0"  "0"  "-"  "0"   "1"
## t1  "0"  "0"  "0"  "0"  "-"  "0"  "0"  "0"  "-"  "2"   "1"
## t5  "0"  "0"  "0"  "0"  "-"  "0"  "0"  "0"  "-"  "0"   "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 (α) 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        104.0000000
## Consistency index          0.0000000
## Retention index            0.7886179
## Robinson-Foulds distance   0.0000000
```

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.

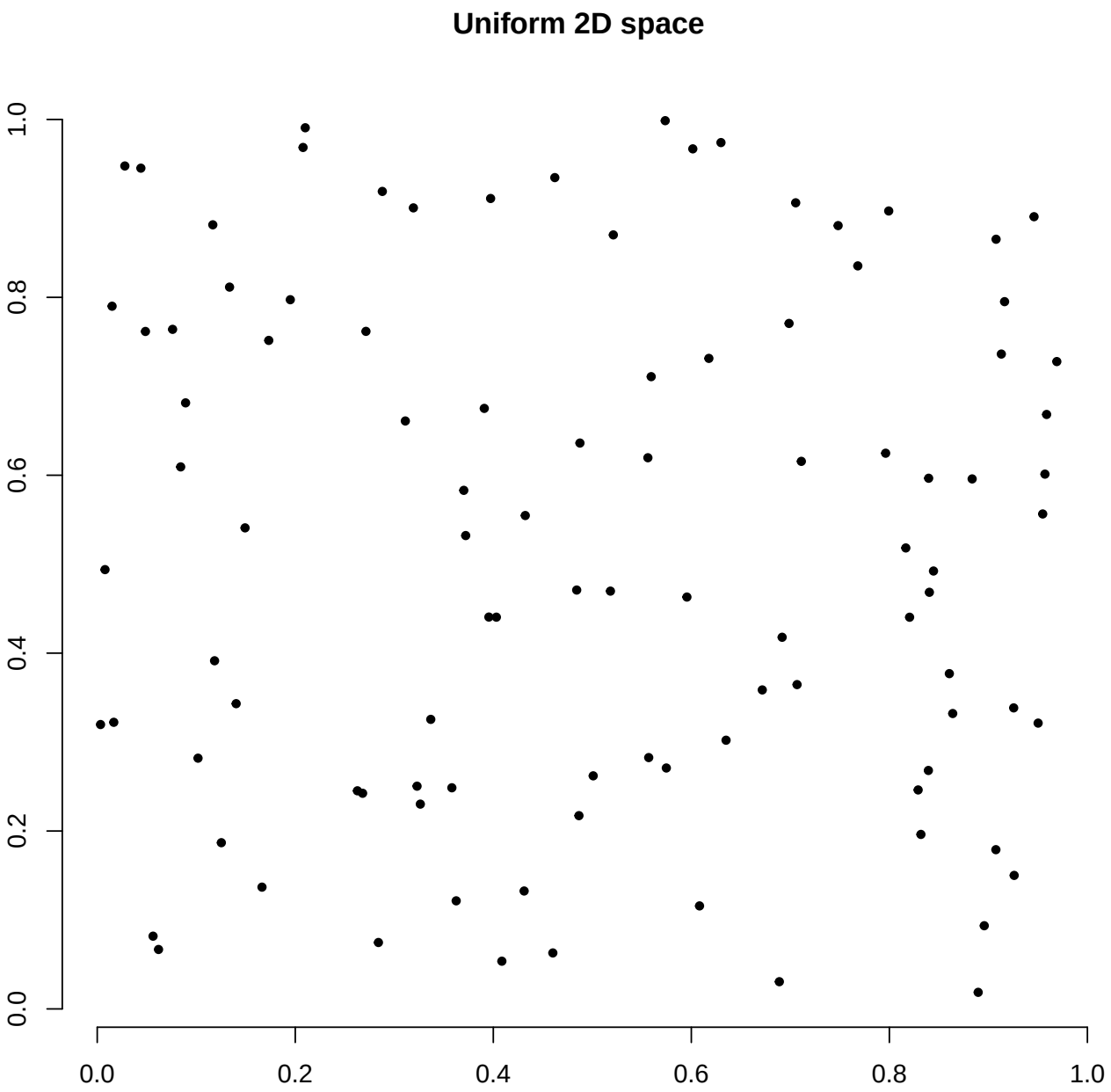
Furthermore, it is possible to add a correlation matrix to add a correlation between the axis via the `cor.matrix` argument or even a vector of proportion of variance to be bear by each axis via the `scree` argument to simulate realistic ordinated spaces.

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,] 0.9548679 0.55645395
## [2,] 0.3721235 0.53218069
## [3,] 0.3229877 0.25041834
## [4,] 0.8404244 0.46840450
## [5,] 0.2839796 0.07466592
## [6,] 0.2627652 0.24523019
```

```
## 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))
```

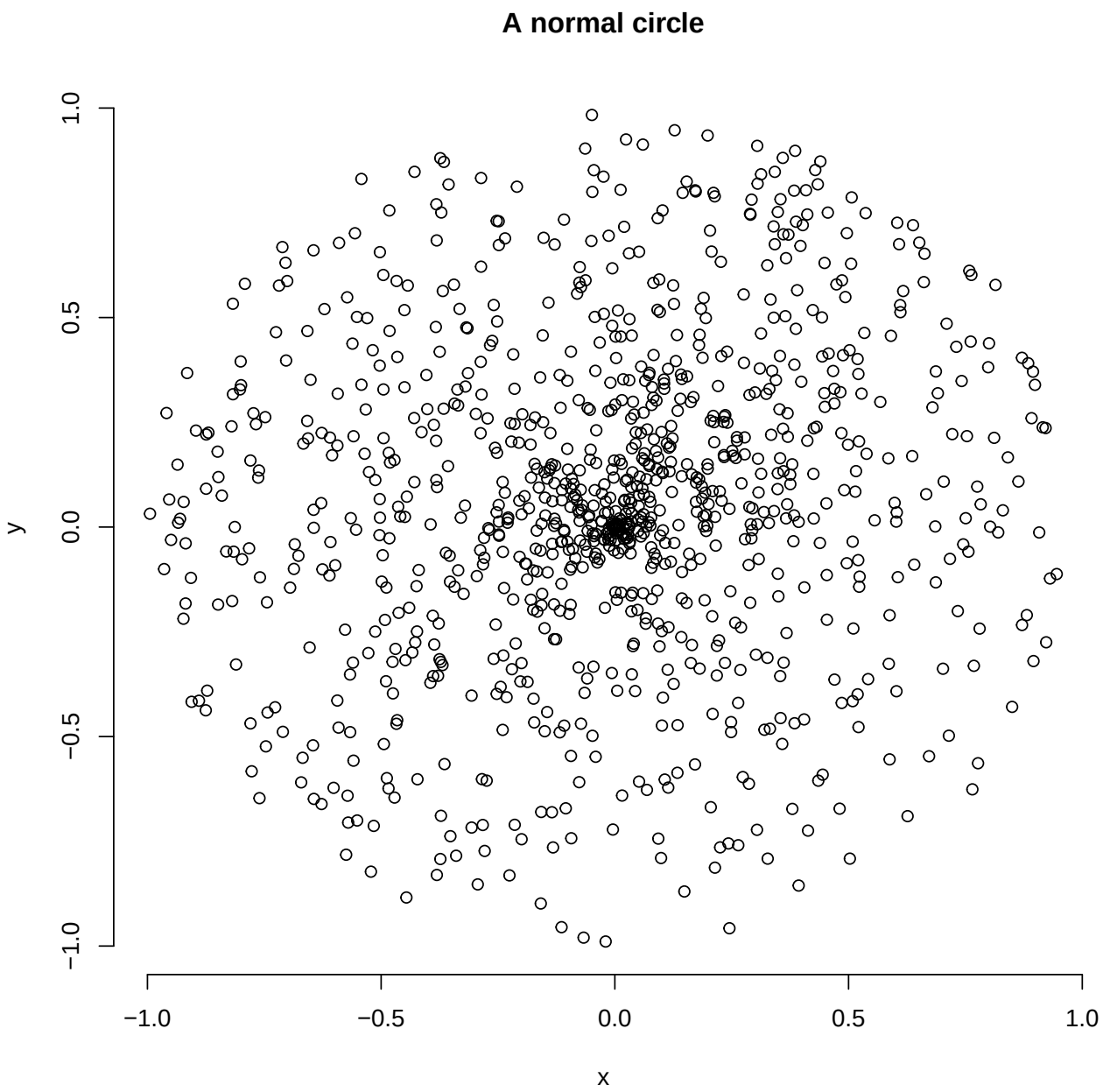
```
##      [,1] [,2] [,3]
## [1,]  1.0  0.8  0.2
## [2,]  0.8  1.0  0.7
## [3,]  0.2  0.7  1.0
```

```
## 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)))
```

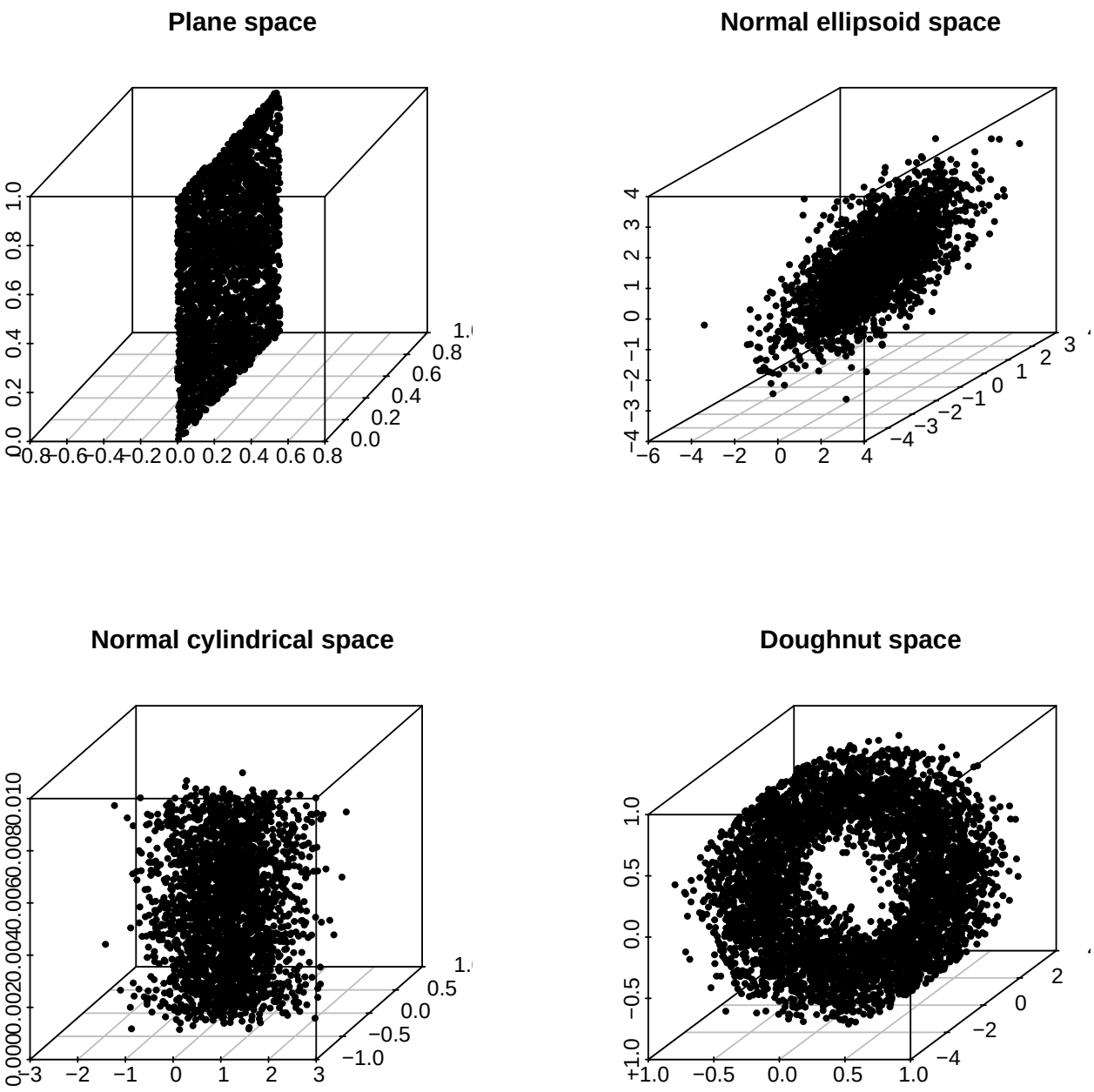
5.2.2 Visualising the space

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")
```



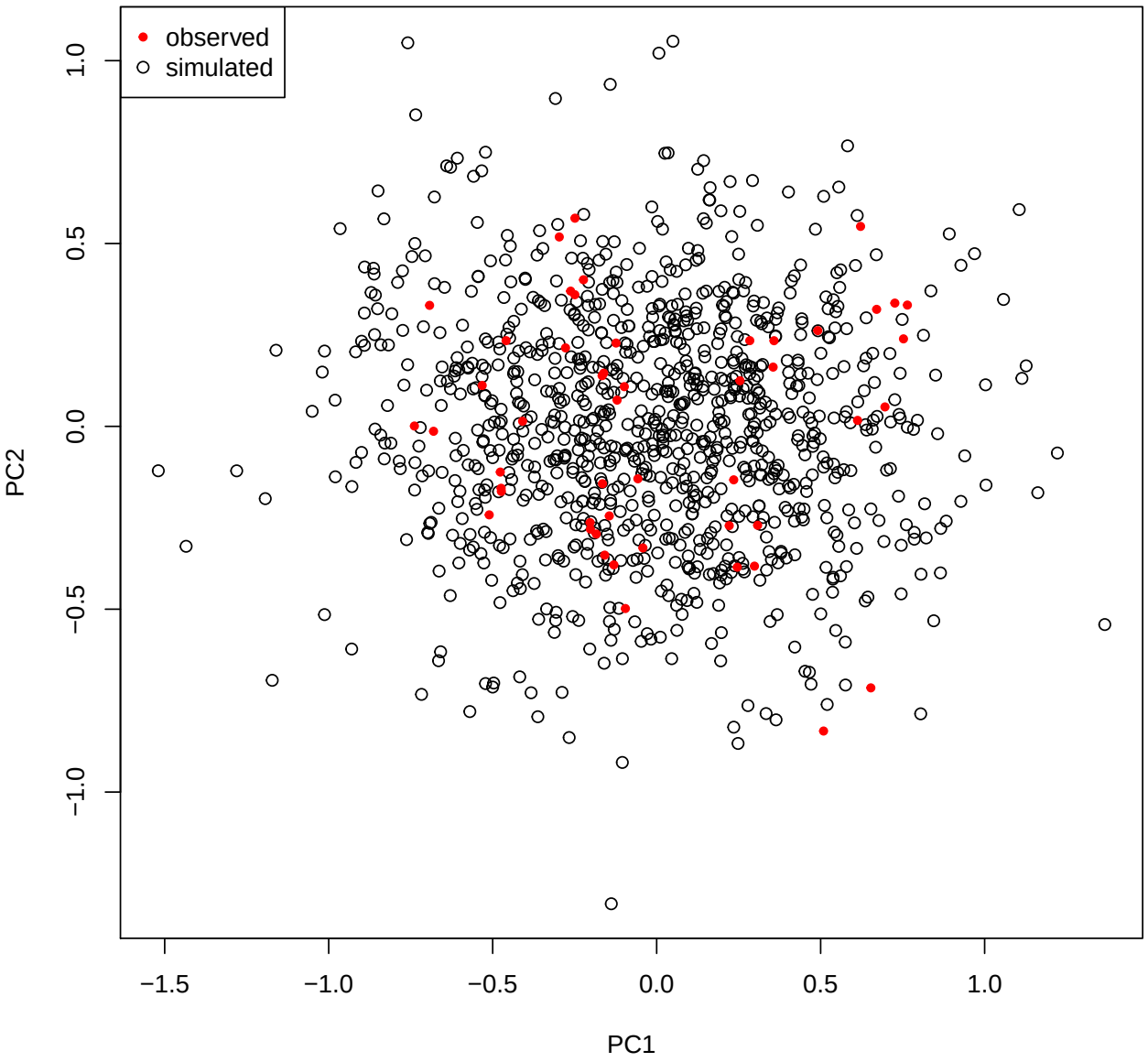

```
par(op)
```

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)
```

```r
## 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"))
```

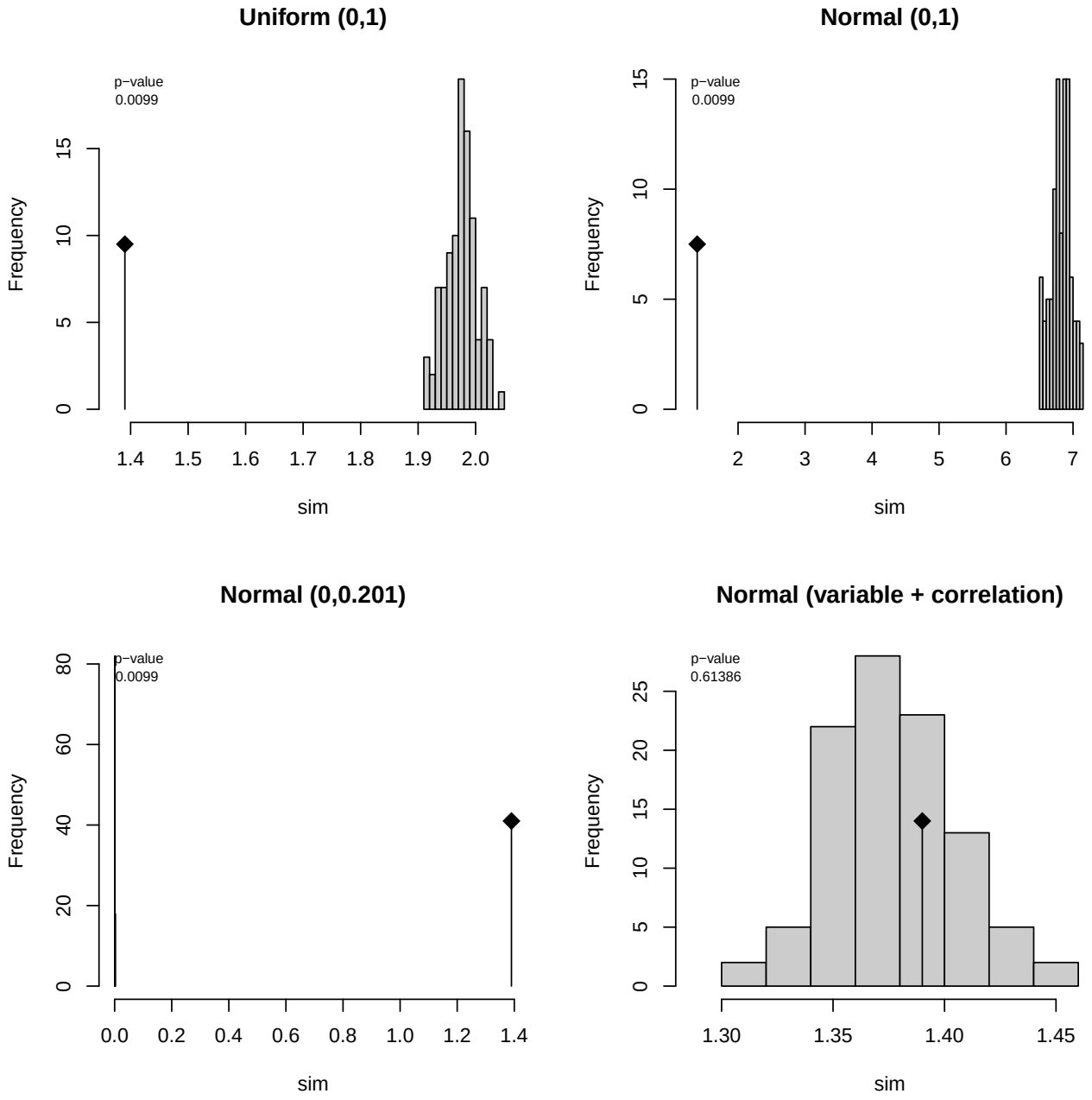


It is now possible to simulate a space using these observed arguments to test several hypothesis:

- 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

The guts of the dispRity package

6.1 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"    "call"      "subsets"   "disparity"
```

```
## Summarising it using the S3 method print.dispRity
disparity
```

```
##  ---- dispRity object ----
## 7 continuous (acctran) time subsets for 99 elements with 97 dimensions:
##      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!
```

6.2 dispRity utilities

The package also provides some utility functions to facilitate multidimensional analysis.

6.2.1 dispRity object utilities

The first set of utilities are functions for manipulating dispRity objects:

6.2.1.1 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 only a matrix 5x4.
```

6.2.1.2 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))
```

```
##  ---- dispRity object ----
## 5 elements with 4 dimensions.
```

```
## The dipRity object has now the correct minimal attributes
disparity_obj$call
```

```
## $dimensions
## [1] 4
```

6.2.1.3 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] -0.1038 0.2844 0.2848 0.0927 0.1619 ...
##  - 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] -0.7161 0.3496 -0.573 -0.0445 -0.1427 ...
##  - attr(*, "dimnames")=List of 2
##   ..$ : chr [1:15] "n7" "n34" "Maelestes" "n20" ...
##   ..$ : NULL
```

6.2.1.4 get.subsets.dispRity

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.dispRity(disp_crown_stemBS, "Group.crown"))

## The object keeps the properties of the parent object but is composed of only one subsets
length(crown_mammals$subsets)
```

6.2.1.5 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)
```

6.2.1.6 scale.dispRity

This is the S3 method of `scale` (scaling and/or centring) that can be applied to the disparity data of a `dispRity` object.

```
## Getting the disparity values of the time subsets
head(summary(disparity))

## Scaling the same disparity values
head(summary(scale(disparity, scale = TRUE)))

## Scaling and centering:
head(summary(scale(disparity, scale = TRUE, center = TRUE)))
```

6.2.1.7 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))))
```

6.3 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:

- `$matrix`: an object of class `matrix`, the full multidimensional space.
- `$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 is present in the `R` environment rather than multiple copies of it for each subset.

6.3.1 $matrix

This is the multidimensional space, stored in the `R` environment as a `matrix` object. It requires row names but not column names. 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.

6.3.2 $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"`) and their eventual model (`"acctran"`, `"deltran"`, `"random"`, `"proximity"`, `"punctuated"`, `"gradual"`). This element generated only once via `time.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.).

6.3.3 $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…

6.3.4 $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
```

Chapter 7

Palaeobiology demo: disparity-through-time and within groups

This demo aims to give quick overview of the `dispRity` package (v.0.4) for palaeobiology analyses of disparity, including disparity through time analyses.

This demo showcases a typical disparity-through-time analysis: we are going to test whether the disparity changed through time in a subset of eutherian mammals from the last 100 million years using a dataset from Beck and Lee (2014).

7.1 Before starting

7.1.1 The morphospace

In this example, we are going to use a subset of the data from Beck and Lee (2014). See the example data description for more details. Briefly, 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`). The ordinated matrix will represent our full morphospace, i.e. all the mammalian morphologies that ever existed through time (for this dataset).

```
## Loading demo and the package data
library(dispRity)

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

## Loading the ordinated matrix/morphospace:
data(BeckLee_mat50)
head(BeckLee_mat50[,1:5])
```

```
##                  [,1]          [,2]        [,3]       [,4]      [,5]
## Cimolestes   -0.5319679  0.1117759259  0.09865194 -0.1933148 0.2035833
## Maelestes    -0.4087147  0.0139690317  0.26268300  0.2297096 0.1310953
## Batodon      -0.6923194  0.3308625215 -0.10175223 -0.1899656 0.1003108
```

```
## Bulaklestes  -0.6802291 -0.0134872777  0.11018009 -0.4103588 0.4326298
## Daulestes    -0.7386111  0.0009001369  0.12006449 -0.4978191 0.4741342
## Uchkudukodon -0.5105254 -0.2420633915  0.44170317 -0.1172972 0.3602273
```

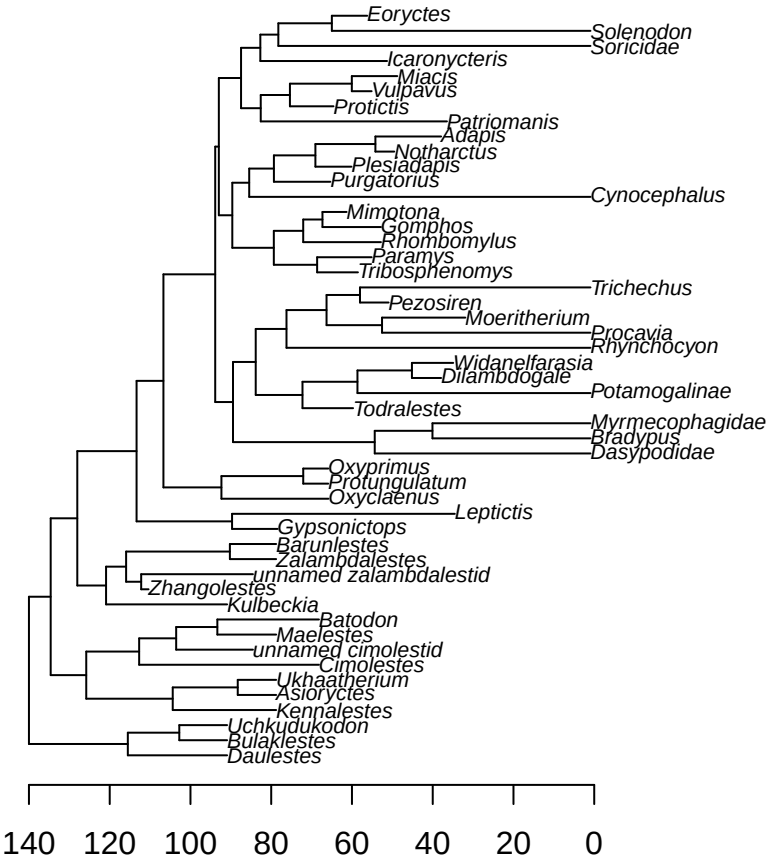
```
dim(BeckLee_mat50)
```

```
## [1] 50 48
```

```
## The morphospace contains 50 taxa and has 48 dimensions (or axes)

## Showing a list of first and last occurrences data for some fossils
data(BeckLee_ages)
head(BeckLee_ages)
```

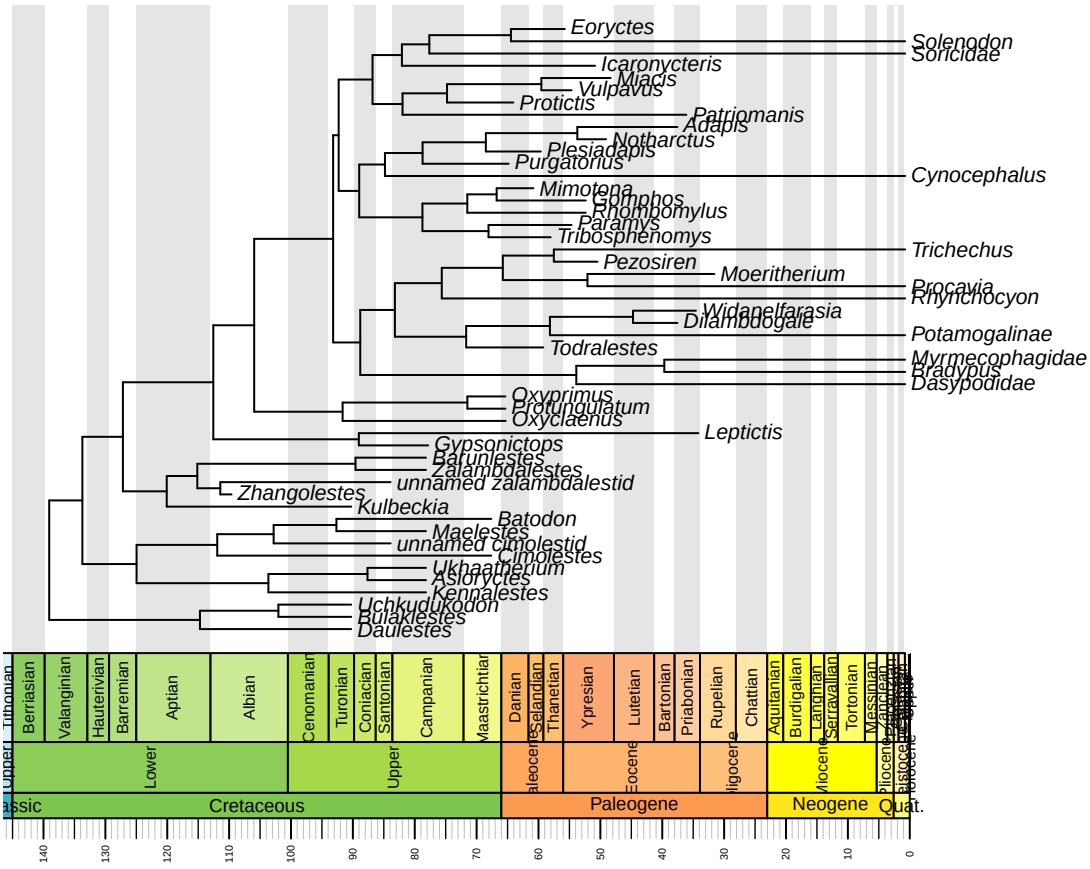
```
##             FAD  LAD
## Adapis     37.2 36.8
## Asioryctes 83.6 72.1
## Leptictis  33.9 33.3
## Miacis     49.0 46.7
## Mimotona   61.6 59.2
## Notharctus 50.2 47.0
```

```
## Plotting a phylogeny
data(BeckLee_tree)
plot(BeckLee_tree, cex = 0.7)
axisPhylo(root = 140)
```



You can have an even nicer looking tree if you use the `strap` package!

```
if(!require(strap)) install.packages("strap")
strap::geoscalePhylo(BeckLee_tree, cex.tip = 0.7, cex.ts = 0.6)
```



7.2 A disparity-through-time analysis

7.2.1 Splitting the morphospace through time

One of the crucial steps in disparity-through-time analysis is to split the full morphospace into smaller time subsets that contain the total number of morphologies at certain points in time (time-slicing) or during certain periods in time (time-binning). Basically, the full morphospace represents the total number of morphologies across all time and will be greater than any of the time subsets of the morphospace.

The `dispRity` package provides a `time.subsets` function that allows users to split the morphospace into time slices (using `method = continuous`) or into time bins (using `method = discrete`). In this example, we are going to split the morphospace into five equal time bins of 20 million years long from 100 million years ago to the present. We will also provide to the function a table containing the first and last occurrences dates for some fossils to take into account that some fossils might occur in several of our different time bins.

```
## Creating the vector of time bins ages
(time_bins <- rev(seq(from = 0, to = 100, by = 20)))
```

```
## [1] 100  80  60  40  20   0
```

```
## Splitting the morphospace using the time.subsets function
(binned_morphospace <- time.subsets(data = BeckLee_mat50, tree = BeckLee_tree,
```

```
    method = "discrete", time = time_bins, inc.nodes = FALSE,
    FADLAD = BeckLee_ages))
```

```
##  ---- dispRity object ----
## 5 discrete time subsets for 50 elements:
##     100 - 80, 80 - 60, 60 - 40, 40 - 20, 20 - 0.
```

The output object is a `dispRity` object (see more about that here. In brief, however, `dispRity` objects are lists of different elements (i.e. disparity results, morphospace time subsets, morphospace attributes, etc.) that display only a summary of the object when calling the object to avoiding filling the `R` console with superfluous output.

```
## Printing the class of the object
class(binned_morphospace)
```

```
## [1] "dispRity"
```

```
## Printing the content of the object
str(binned_morphospace)
```

```
## List of 3
##  $ matrix : num [1:50, 1:48] -0.532 -0.409 -0.692 -0.68 -0.739 ...
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : chr [1:50] "Cimolestes" "Maelestes" "Batodon" "Bulaklestes" ...
##   .. ..$ : NULL
##  $ call   :List of 1
##   ..$ subsets: chr "discrete"
##  $ subsets:List of 5
##   ..$ 100 - 80:List of 1
##   .. ..$ elements: int [1:8, 1] 5 4 6 8 43 10 11 42
##   ..$ 80 - 60 :List of 1
##   .. ..$ elements: int [1:15, 1] 7 8 9 1 2 3 12 13 14 44 ...
##   ..$ 60 - 40 :List of 1
##   .. ..$ elements: int [1:13, 1] 41 49 24 25 26 27 28 21 22 19 ...
##   ..$ 40 - 20 :List of 1
##   .. ..$ elements: int [1:6, 1] 15 39 40 35 23 47
##   ..$ 20 - 0  :List of 1
##   .. ..$ elements: int [1:10, 1] 36 37 38 32 33 34 50 48 29 30
##  - attr(*, "class")= chr "dispRity"
```

```
names(binned_morphospace)
```

```
## [1] "matrix"  "call"    "subsets"
```

```
## Printing the object as a dispRity class
binned_morphospace
```

```
##  ---- dispRity object ----
## 5 discrete time subsets for 50 elements:
##     100 - 80, 80 - 60, 60 - 40, 40 - 20, 20 - 0.
```

> These objects will gradually contain more information when completing the following steps in the disparity-through-time analysis.

7.2.2 Bootstrapping the data

Once we obtain our different time subsets, we can bootstrap and rarefy them (i.e. pseudo-replicating the data). The bootstrapping allows us to make each subset more robust to outliers and the rarefaction allows

us to compare subsets with the same number of taxa to remove sampling biases (i.e. more taxa in one subset than the others). The `boot.matrix` function bootstraps the `dispRity` object and the `rarefaction` option within performs rarefaction.

```
## Bootstrapping each time subset 100 times (default)
(boot_bin_morphospace <- boot.matrix(binned_morphospace))
```

```
##  ---- dispRity object ----
## 5 discrete time subsets for 50 elements with 48 dimensions:
##     100 - 80, 80 - 60, 60 - 40, 40 - 20, 20 - 0.
## Data was bootstrapped 100 times (method:"full").
```

```
## Getting the minimum number of rows (i.e. taxa) in the time subsets
min(size.subsets(boot_bin_morphospace))
```

```
## [1] 6
```

```
## Bootstrapping each time subset 100 times and rarefying them
(rare_bin_morphospace <- boot.matrix(binned_morphospace, bootstraps = 100,
    rarefaction = 6))
```

```
##  ---- dispRity object ----
## 5 discrete time subsets for 50 elements with 48 dimensions:
##     100 - 80, 80 - 60, 60 - 40, 40 - 20, 20 - 0.
## Data was bootstrapped 100 times (method:"full") and rarefied to 6 elements.
```

7.2.3 Calculating disparity

We can now calculate the disparity within each time subsets along with some confidence intervals generated by the pseudoreplication step above (bootstraps/rarefaction). Disparity can be calculated in many ways and this package allows users to come up with their own disparity metrics. For more details, please refer to the `dispRity` metric section.

In this example, we are going to calculate the spread of the data in each time subset by calculating disparity as the sum of the variance of each dimension of the morphospace in each time subset using the `dispRity` function. Thus, in this example, disparity is defined by the multi-dimensional variance of each time subset (i.e. the spread of the taxa within the morphospace). Note that this metric comes with a caveat (not solved here) since it ignores covariances among the dimensions of the morphospace. We use this here because it is a standard metric used in disparity-through-time analysis (Wills *et al.* 1994).

```
## Calculating disparity for the bootstrapped data
(boot_disparity <- dispRity(boot_bin_morphospace, metric = c(sum, variances)))
```

```
##  ---- dispRity object ----
## 5 discrete time subsets for 50 elements with 48 dimensions:
##     100 - 80, 80 - 60, 60 - 40, 40 - 20, 20 - 0.
## Data was bootstrapped 100 times (method:"full").
## Disparity was calculated as: c(sum, variances).
```

```
## Calculating disparity for the rarefied data
(rare_disparity <- dispRity(rare_bin_morphospace, metric = c(sum, variances)))
```

```
##  ---- dispRity object ----
## 5 discrete time subsets for 50 elements with 48 dimensions:
##     100 - 80, 80 - 60, 60 - 40, 40 - 20, 20 - 0.
## Data was bootstrapped 100 times (method:"full") and rarefied to 6 elements.
## Disparity was calculated as: c(sum, variances).
```

The `dispRity` function does not actually display the calculated disparity values but rather only the properties of the disparity object (size, subsets, metric, etc.). To display the actual calculated scores, we need to summarise the disparity object using the S3 method `summary` that is applied to a `dispRity` object (see `?summary.dispRity` for more details).

As for any `R` package, you can refer to the help files for each individual function for more details.

```
## Summarising the disparity results
summary(boot_disparity)
```

```
##    subsets  n   obs bs.median  2.5%   25%   75% 97.5%
## 1 100 - 80  8 1.675     1.488 1.087 1.389 1.568 1.648
## 2  80 - 60 15 1.782     1.679 1.538 1.631 1.728 1.792
## 3  60 - 40 13 1.913     1.772 1.607 1.734 1.826 1.886
## 4  40 - 20  6 2.022     1.707 1.212 1.537 1.822 1.942
## 5   20 - 0 10 1.971     1.794 1.598 1.716 1.842 1.890
```

```
summary(rare_disparity)
```

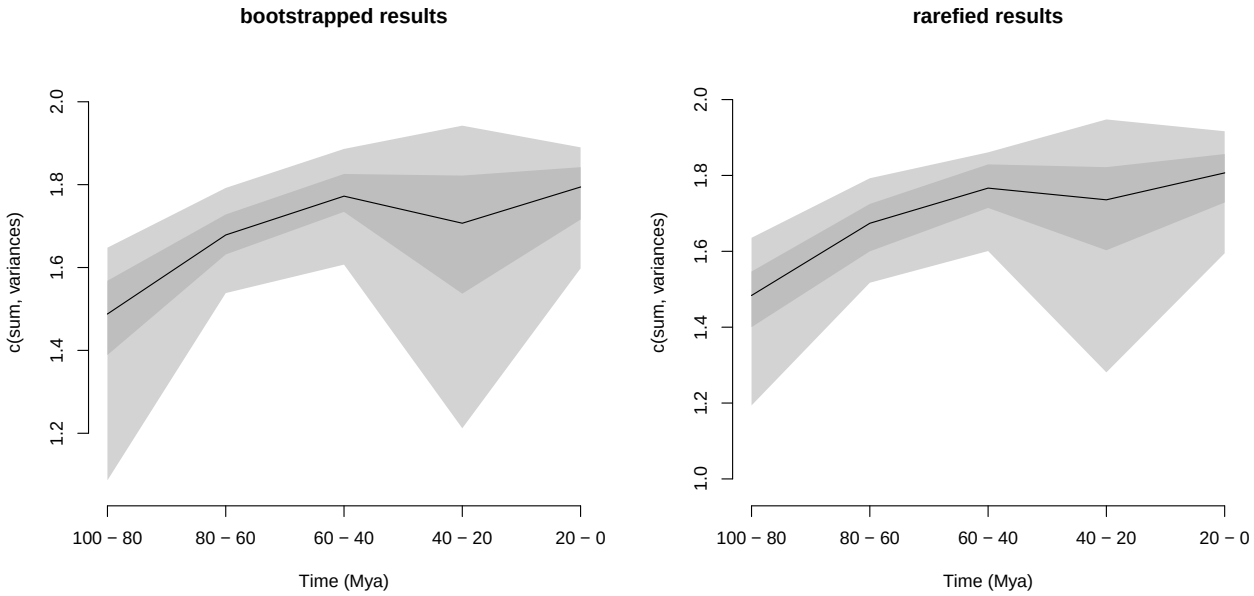
```
##    subsets  n   obs bs.median  2.5%   25%   75% 97.5%
## 1 100 - 80  8 1.675     1.484 1.194 1.400 1.547 1.636
## 2 100 - 80  6    NA     1.477 0.993 1.361 1.569 1.698
## 3  80 - 60 15 1.782     1.674 1.517 1.600 1.725 1.793
## 4  80 - 60  6    NA     1.655 1.299 1.532 1.754 1.882
## 5  60 - 40 13 1.913     1.767 1.601 1.714 1.829 1.861
## 6  60 - 40  6    NA     1.787 1.314 1.672 1.879 1.984
## 7  40 - 20  6 2.022     1.736 1.281 1.603 1.822 1.948
## 8   20 - 0 10 1.971     1.807 1.595 1.729 1.856 1.917
## 9   20 - 0  6    NA     1.790 1.435 1.718 1.873 1.995
```

> The summary.dispRity function comes with many options on which values to calculate (central tendency and quantiles) and on how many digits to display. Refer to the function's manual for more details.

7.2.4 Plotting the results

It is sometimes easier to visualise the results in a plot than in a table. For that we can use the `plot` S3 function to plot the `dispRity` objects (see `?plot.dispRity` for more details).

```
## Graphical options
quartz(width = 10, height = 5) ; par(mfrow = (c(1,2)), bty = "n")

## Plotting the bootstrapped and rarefied results
plot(boot_disparity, type = "continuous", main = "bootstrapped results")
plot(rare_disparity, type = "continuous", main = "rarefied results")
```



7.3 Testing differences

Finally, to draw some valid conclusions from these results, we can apply some statistical tests. We can test, for example, if mammalian disparity changed significantly through time over the last 100 million years. To do so, we can compare the means of each time-bin in a sequential manner to see whether the disparity in bin *n* is equal to the disparity in bin *n+1*, and whether this is in turn equal to the disparity in bin *n+2*, etc. Because our data is temporally autocorrelated (i.e. what happens in bin *n+1* depends on what happened in bin *n*) and pseudoreplicated (i.e. each bootstrap draw creates non-independent time subsets because they are all based on the same time subsets), we apply a non-parametric mean comparison: the `wilcox.test`. Also, we need to apply a p-value correction (e.g. Bonferroni correction) to correct for multiple testing (see `?p.adjust` for more details).

```
## Testing the differences between bins in the bootstrapped dataset.
test.dispRity(boot_disparity, test = wilcox.test, comparison = "sequential",
    correction = "bonferroni")
```

```
## [[1]]
##                     statistic
## 100 - 80 : 80 - 60       471
## 80 - 60 : 60 - 40       1562
## 60 - 40 : 40 - 20       6250
## 40 - 20 : 20 - 0        3725
##
## [[2]]
##                          p.value
## 100 - 80 : 80 - 60 7.427563e-28
## 80 - 60 : 60 - 40  1.798899e-16
## 60 - 40 : 40 - 20  9.061511e-03
## 40 - 20 : 20 - 0   7.379715e-03
```

```
## Testing the differences between bins in the rarefied dataset.
test.dispRity(rare_disparity, test = wilcox.test, comparison = "sequential",
    correction = "bonferroni")
```

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

```
##                     statistic
## 100 - 80 : 80 - 60       662
## 80 - 60 : 60 - 40       1814
## 60 - 40 : 40 - 20       5752
## 40 - 20 : 20 - 0        3621
##
## [[2]]
##                          p.value
## 100 - 80 : 80 - 60 1.214988e-25
## 80 - 60 : 60 - 40  2.823697e-14
## 60 - 40 : 40 - 20  2.653018e-01
## 40 - 20 : 20 - 0   3.026079e-03
```

Here our results show significant changes in disparity through time between all time bins (all p-values $<$ 0.05). However, when looking at the rarefied results, there is no significant difference between the time bins in the Palaeogene (60-40 to 40-20 Mya), suggesting that the differences detected in the first test might just be due to the differences in number of taxa sampled (13 or 6 taxa) in each time bin.

Chapter 8

Ecology demo

This is an example of typical disparity analysis that can be performed in ecology.

8.1 Data

For this example, we will use the famous `iris` inbuilt data set

```
data(iris)
```

This data contains petal and sepal length for 150 individual plants sorted into three species.

```
## Separating the species
species <- iris[,5]
## Which species?
unique(species)
```

```
## [1] setosa     versicolor virginica
## Levels: setosa versicolor virginica
```

```
## Separating the petal/sepal length
measurements <- iris[,1:4]
head(measurements)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width
## 1          5.1         3.5          1.4         0.2
## 2          4.9         3.0          1.4         0.2
## 3          4.7         3.2          1.3         0.2
## 4          4.6         3.1          1.5         0.2
## 5          5.0         3.6          1.4         0.2
## 6          5.4         3.9          1.7         0.4
```

We can then ordinate the data using a PCA (`prcomp` function) thus defining our four dimensional space as the poetically named petal-space.

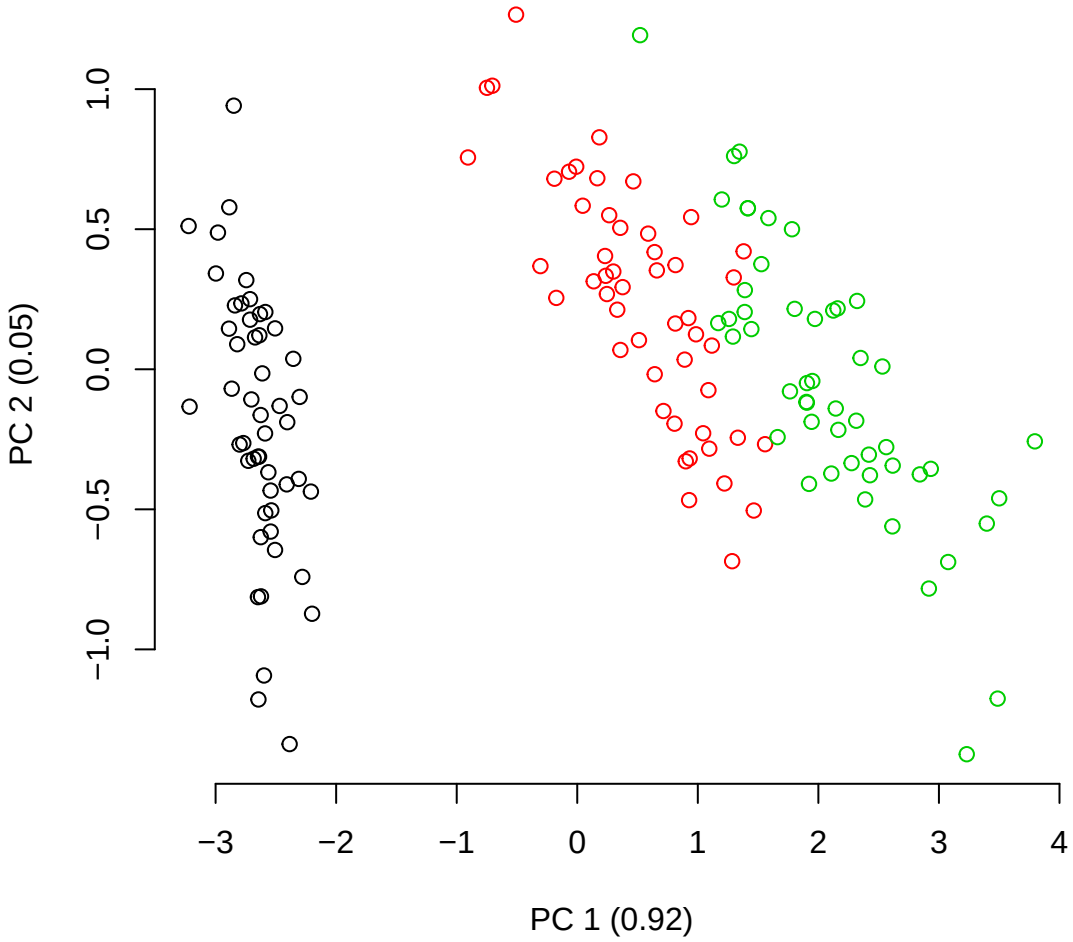
```
## Ordinating the data
ordination <- prcomp(measurements)

## The petal-space
petal_space <- ordination$x
```

```
## Adding the elements names to the petal-space (the individuals IDs)
rownames(petal_space) <- 1:nrow(petal_space)
```

8.2 Classic analysis

A classical way to represent this ordinated data would be to use two dimensional plots to look at how the different species are distributed in the petal-space.

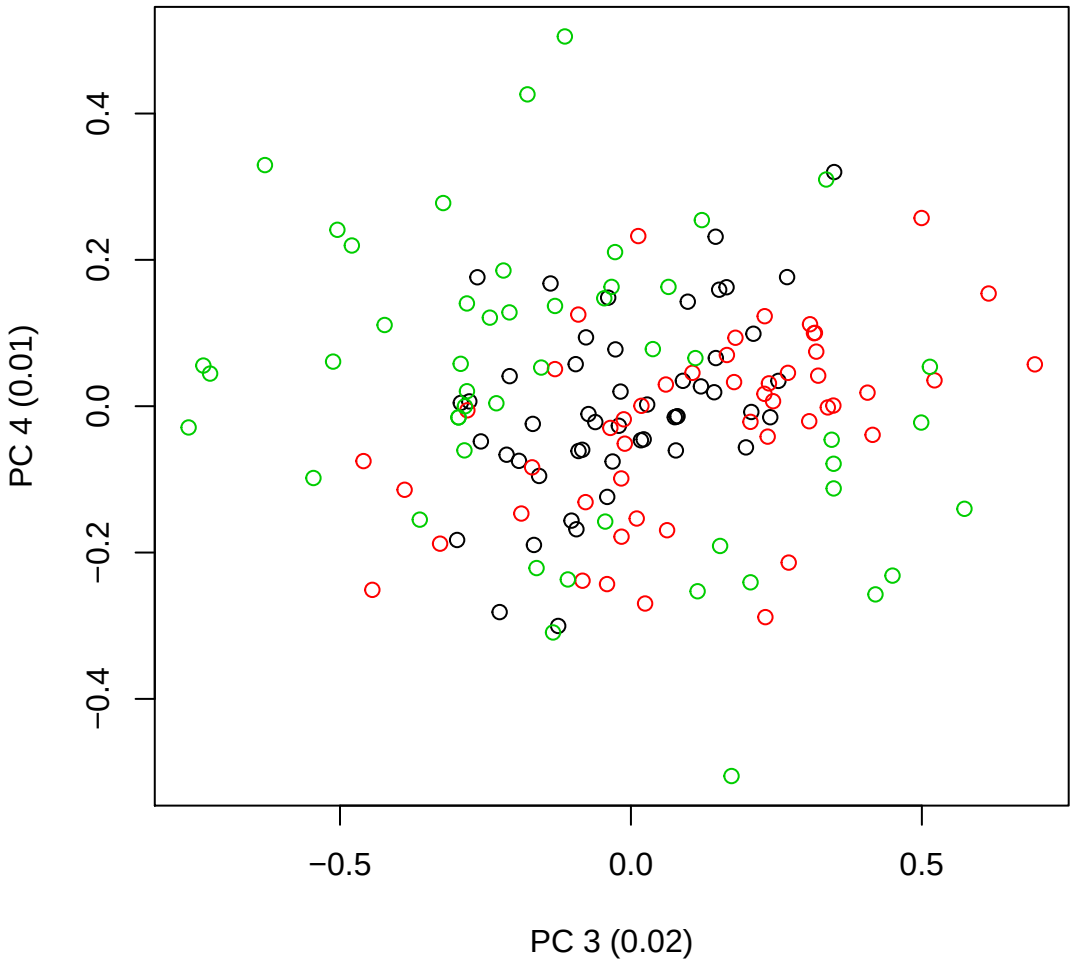
```
## Measuring the variance on each axis
variances <- apply(petal_space, 2, var)
variances <- variances/sum(variances)

## Graphical option
par(bty = "n")

## A classic 2D ordination plot
plot(petal_space[, 1], petal_space[, 2], col = species,
    xlab = paste0("PC 1 (", round(variances[1], 2), ")"),
    ylab = paste0("PC 2 (", round(variances[2], 2), ")"))
```



This shows the distribution of the different species in the petal-space along the two first axis of variation. This is a pretty standard way to visualise the multidimensional space and further analysis might be necessary to test wether the groups are different such as a linear discriminant analysis (LDA). However, in this case

we are ignoring the two other dimensions of the ordination! If we look at the two other axis we see a totally different result:

```
## Plotting the two second axis of the petal-space
plot(petal_space[, 3], petal_space[, 4], col = species,
    xlab = paste0("PC 3 (", round(variances[3], 2), ")"),
    ylab = paste0("PC 4 (", round(variances[4], 2), ")"))
```



Additionally, these two represented dimensions do not represent a biological reality *per se*; i.e. the values on the first dimension do not represent a continuous trait (e.g. petal length), instead they just represent the ordinations of correlations between the data and some factors.

Therefore, we might want to approach this problem without getting stuck in only two dimensions and consider the whole dataset as a n-dimensional object.

8.3 A multidimensional approach with dispRity

The first step is to create different subsets that represent subsets of the ordinated space (i.e. sub-regions within the n-dimensional object). Each of these subsets will contain only the individuals of a specific species.

```
## Creating the table that contain the elements and their attributes
petal_subsets <- custom.subsets(petal_space, group = list(
                                "setosa" = which(species == "setosa"),
                                "versicolor" = which(species == "versicolor"),
```

```
                                "virginica" = which(species == "virginica")))

## Visualising the dispRity object content
petal_subsets
```

```
##  ---- dispRity object ----
## 3 customised subsets for 150 elements:
##     setosa, versicolor, virginica.
```

This created a `dispRity` object (more about that here) with three subsets corresponding to each subspecies.

8.3.1 Bootstrapping the data

We can the bootstrap the subsets to be able test the robustness of the measured disparity to outliers. We can do that using the default options of `boot.matrix` (more about that here):

```
## Bootstrapping the data
(petal_bootstrapped <- boot.matrix(petal_subsets))
```

```
##  ---- dispRity object ----
## 3 customised subsets for 150 elements with 4 dimensions:
##     setosa, versicolor, virginica.
## Data was bootstrapped 100 times (method:"full").
```

8.3.2 Calculating disparity

Disparity can be calculated in many ways, therefore the `dispRity` function allows users to define their own measure of disparity. For more details on measuring disparity, see the dispRity metrics section.

In this example, we are going to define disparity as the median distance between the different individuals and the centroid of the ordinated space. High values of disparity will indicate a generally high spread of points from this centroid (i.e. on average, the individuals are far apart in the ordinated space). We can define the metrics easily in the `dispRity` function by feeding them to the `metric` argument. Here we are going to feed the functions `stats::median` and `dispRity::centroids` which calculates distances between elements and their centroid.

```
## Calculating disparity as the median distance between each elements and
## the centroid of the petal-space
(petal_disparity <- dispRity(petal_bootstrapped, metric = c(median, centroids)))
```

```
##  ---- dispRity object ----
## 3 customised subsets for 150 elements with 4 dimensions:
##     setosa, versicolor, virginica.
## Data was bootstrapped 100 times (method:"full").
## Disparity was calculated as: c(median, centroids).
```

8.3.3 Summarising the results (plot)

Similarly to the `custom.subsets` and `boot.matrix` function, `dispRity` displays a `dispRity` object. But we are definitely more interested in actually look at the calculated values.

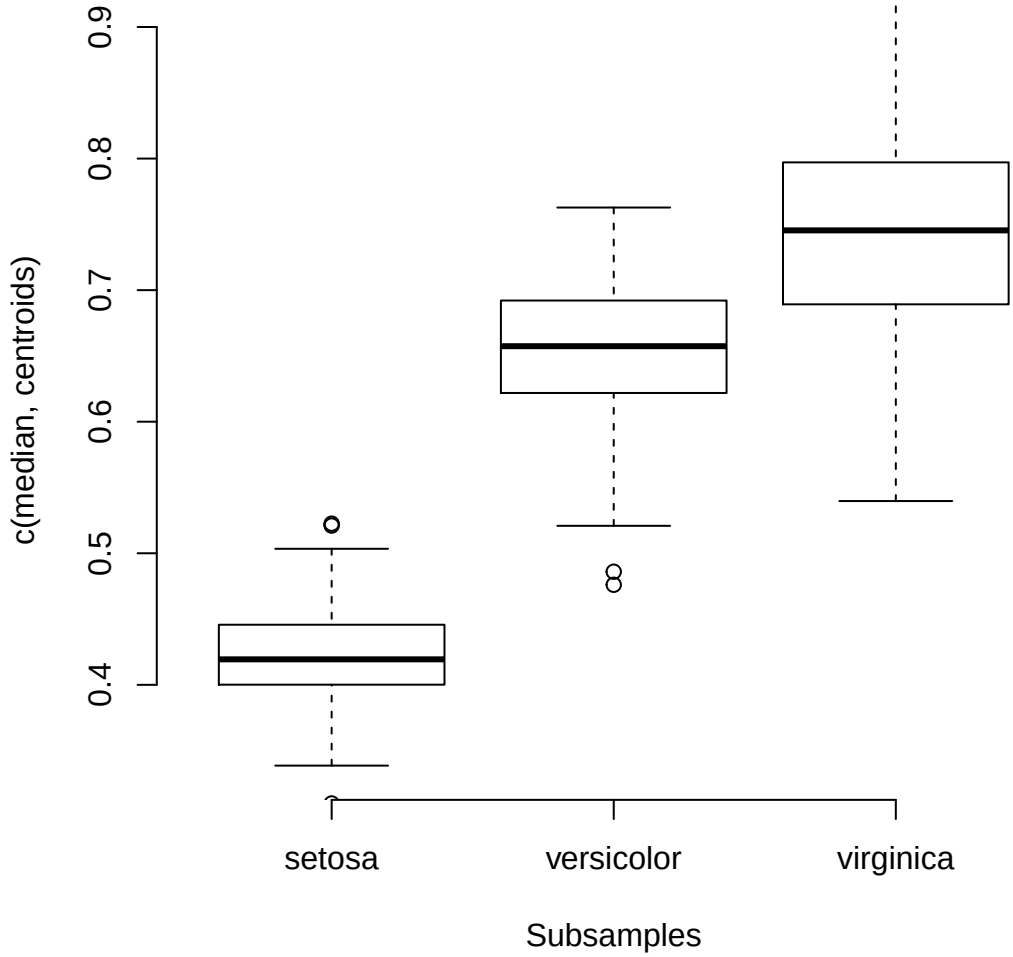
First we can summarise the data in a table by simply using `summary`:

```
## Displaying the summary of the calculated disparity
summary(petal_disparity)
```

```
##      subsets  n   obs bs.median  2.5%   25%   75% 97.5%
## 1     setosa 50 0.421     0.419 0.342 0.401 0.445 0.503
## 2 versicolor 50 0.693     0.657 0.530 0.622 0.692 0.733
## 3  virginica 50 0.785     0.745 0.577 0.690 0.797 0.880
```

We can also plot the results in a similar way:

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

## Plotting the disparity in the petal_space
plot(petal_disparity)
```



Now contrary to simply plotting the two first axis of the PCA where we saw that the species have a different position in the two first petal-space, we can now also see that they occupy this space clearly differently!

8.3.4 Testing hypothesis

Finally we can test our hypothesis that we guessed from the disparity plot (that some groups occupy different volume of the petal-space) by using the `test.dispRity` option.

```
## Fitting a linear model to our data
disparity_lm <- test.dispRity(petal_disparity, test = lm, comparisons = "all")
```

```
## Warning in test.dispRity(petal_disparity, test = lm, comparisons = "all"): Multiple p-values will be calc
```

```
## This will inflate Type I error!
## Testing whether there is a difference in disparity between the different species
summary(aov(disparity_lm))
```

```
##              Df Sum Sq Mean Sq F value Pr(>F)
## subsets       2  5.387  2.6935   705.5 <2e-16 ***
## Residuals   297  1.134  0.0038
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
## Post-hoc testing of the differences between species (corrected for multiple tests)
test.dispRity(petal_disparity, test = t.test, correction = "bonferroni")
```

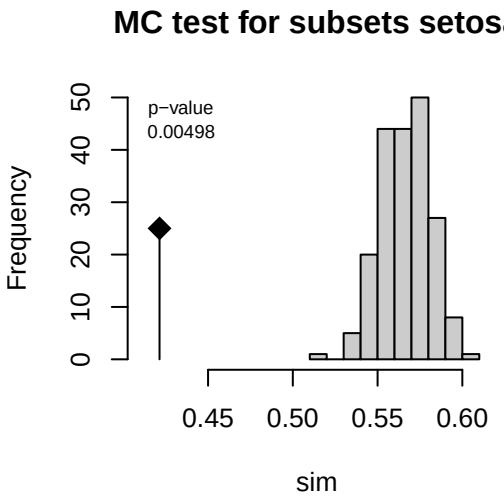
```
## [[1]]
##                           statistic
## setosa : versicolor    -33.714480
## setosa : virginica     -34.257797
## versicolor : virginica  -8.595829
##
## [[2]]
##                          parameter
## setosa : versicolor     189.0217
## setosa : virginica      150.1371
## versicolor : virginica  171.2387
##
## [[3]]
##                               p.value
## setosa : versicolor    2.107278e-81
## setosa : virginica     2.252270e-72
## versicolor : virginica 1.506037e-14
```

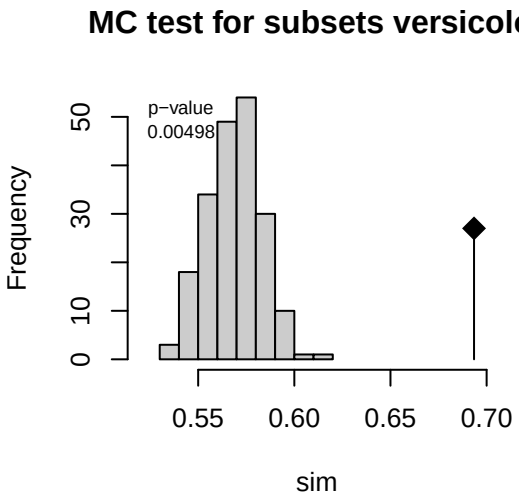
We can now see that there is a significant difference in petal-space occupancy between all species of iris.

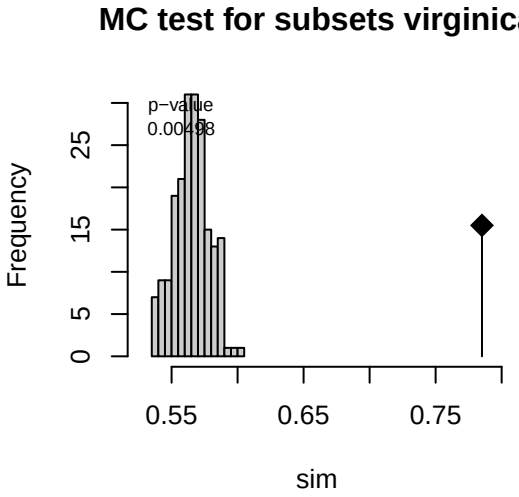
8.3.4.1 Setting up a multidimensional null-hypothesis

One other series of test can be done on the shape of the petal-space. Using a MCMC permutation test we can simulate a petal-space with specific properties and see if our observed petal-space matches these properties (similarly to Diaz *et al.* 2016):

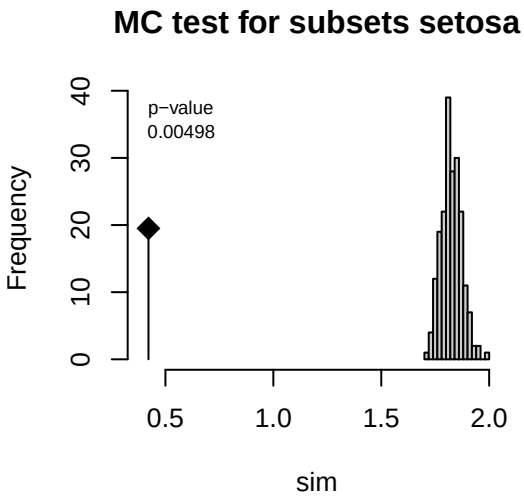
```
## Testing against a uniform distribution
disparity_uniform <- null.test(petal_disparity, replicates = 200,
    null.distrib = runif, scale = FALSE)
plot(disparity_uniform)
```

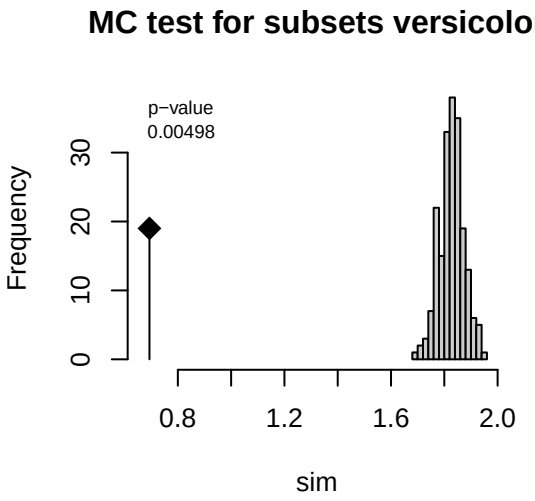


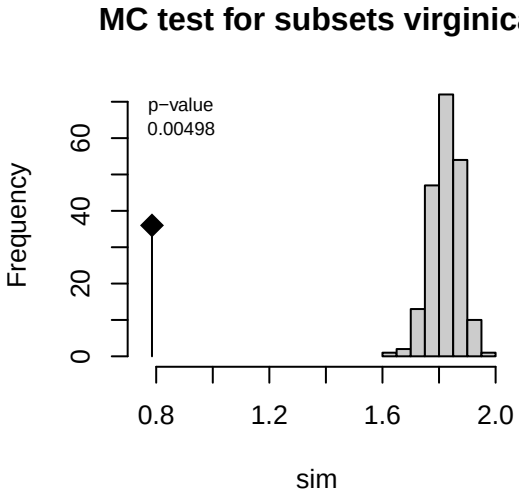





```
## Testing against a normal distribution
disparity_normal <- null.test(petal_disparity, replicates = 200,
    null.distrib = rnorm, scale = TRUE)
plot(disparity_normal)
```







In both cases we can see that our petal-space is not entirely normal or uniform. This is expected because of the simplicity of these parameters.

Chapter 9

Future directions

9.1 More tests!

Some more tests are being developed such as a `sequential.test` to run sequential linear models for testing hypothesis in disparity through time or a `model.test` developed in collaboration with Mark Puttick to fit modes of evolution to disparity curves. Stay tuned!

9.2 Faster disparity calculations

I am slowly implementing parallel disparity calculation as well as C implementations of some disparity metrics to increase significantly improve the speed of the `dispRity` function.

9.3 More modularity

I am equally slowly developing functions to allow more of the options in the package to be modular (in the same way as the `metric` argument in `dispRity`). The next arguments to benefit this increased modularity will be `time.subsets`'s `model` argument and `boot.matrix`'s `type` argument.

Chapter 10

References

- Beck, R. M., & Lee, M. S. (2014). Ancient dates or accelerated rates? Morphological clocks and the antiquity of placental mammals. Proceedings of the Royal Society of London B: Biological Sciences, 281(1793), 20141278.
- Cooper, N., & Guillerme, T. (in prep.). Coming soonish!.
- Diaz, S., Kattge, J., Cornelissen, J.H., Wright, I.J., Lavorel, S., Dray, S., Reu, B., Kleyer, M., Wirth, C., Prentice, I.C. and Garnier, E. (2016). The global spectrum of plant form and function. Nature, 529(7585), 167.
- Donohue, I., Petchey, O.L., Montoya, J.M., Jackson, A.L., McNally, L., Viana, M., Healy, K., Lurgi, M., O'Connor, N.E. and Emmerson, M.C, (2013). On the dimensionality of ecological stability. Ecology letters, 16(4), 421-429.
- Wills, M. A., Briggs, D. E., & Fortey, R. A. (1994). Disparity as an evolutionary index: a comparison of Cambrian and Recent arthropods. Paleobiology, 20(2), 93-130.

Bibliography

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.

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.

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://www.sciencedirect.com/science/article/pii/S1055790315002547.

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.

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.

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 estimation of phylogeny from discrete morphological data. *Biology Letters*, 12(4), 2016. ISSN 1744-9561. doi: 10.1098/rsbl.2016.0081. URL http://rsbl.royalsocietypublishing.org/content/12/4/20160081.

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.