Drawing Phylogenies in R: Basic and Advanced Features With ape

Emmanuel Paradis

December 10, 2024

Contents

---

1 Introduction

Graphical functions have been present in `ape` since its first version (0.1, released in August 2002). Over the years, these tools have been improved to become quite sophisticated although complicated to use efficiently. This document gives an overview of these functionalities. Section 2 explains the basic concepts and tools behind graphics in `ape`. A figure made with `ape` usually starts by calling the function `plot.phylo` which is detailed in Section 3, and further graphical annotations can be done with functions covered in Section 4.

Section 5 shows some specialized functions available in ape, and finally, Sections 6 and 7 give an overview of some ideas to help making complicated figures.

2 Basic Concepts

The core of ape's graphical tools is the `plot` method for the class `"phylo"`, the function `plot.phylo`. This function is studied in details in Section 3, but first we see the basic ideas behind it and other functions mentioned in this document.

2.1 Graphical Model

The graphical functions in ape use the package graphics. Overall, the conventions of this package are followed quite closely (see Murrell's book [3]), so users familiar with graphics in R are expected to find their way relatively easily when plotting phylogenies with ape.

ape has several functions to perform computations before drawing a tree, so that they may be used to implement the same graphical functionalities with other graphical engines such as the grid package. These functions are detailed in the next section.

To start simply, we build a small tree with three genera of primates which we will use in several examples in this document:

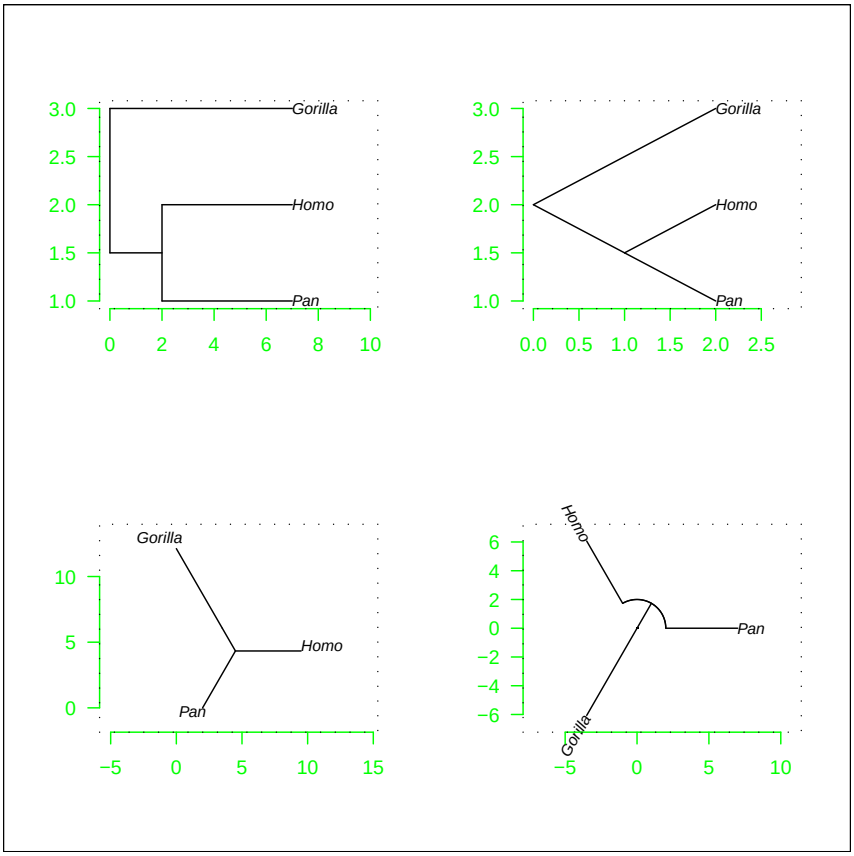
```
> library(ape)
> mytr <- read.tree(text = "((Pan:5,Homo:5):2,Gorilla:7);")
```

Now let's build a small function to show the frame around the plot with dots, and the x- and y-axes in green:

```
> foo <- function() {
+     col <- "green"
+     for (i in 1:2)
+         axis(i, col = col, col.ticks = col, col.axis = col, las = 1)
+     box(lty = "19")
+ }
```

We then plot the tree in four different ways (see below for explanations about the options) and call for each of them the previous small function:

```
> layout(matrix(1:4, 2, 2, byrow = TRUE))
> plot(mytr); foo()
> plot(mytr, "c", FALSE); foo()
> plot(mytr, "u"); foo()
> par(xpd = TRUE)
> plot(mytr, "f"); foo()
> box("outer")
```



The last command (`box("outer")`) makes visible the most outer frame of the figure showing more clearly the margins around each tree (more on this in Sect. 6). We note also the command `par(xpd = TRUE)`: by default this parameter is `FALSE` so that graphical elements (points, lines, text, ...) outside the plotting region (i.e., in the margins or beyond) are cut (clipped).[1] These small figures illustrate the way trees are drawn with `ape`. This can be summarised with the following (pseudo-)algorithm:

---

1. Compute the node coordinates depending on the type of tree plot, the branch lengths, and other parameters.
2. Evaluate the space required for printing the tip labels.
3. Depending on the options, do some rotations and/or translations.
4. Set the limits of the x- and y-axes.
5. Open a graphical device (or reset it if already open) and draw an empty plot with the limits found at the previous step.
6. Call `segments()` to draw the branches.
7. Call `text()` to draw the labels.

There are a lot of ways to control these steps. The main variations along these steps are given below.

[1] `par(xpd = TRUE)` is used in several examples in this document mainly because of the small size of the trees drawn here. However, in practice, this is rarely needed.

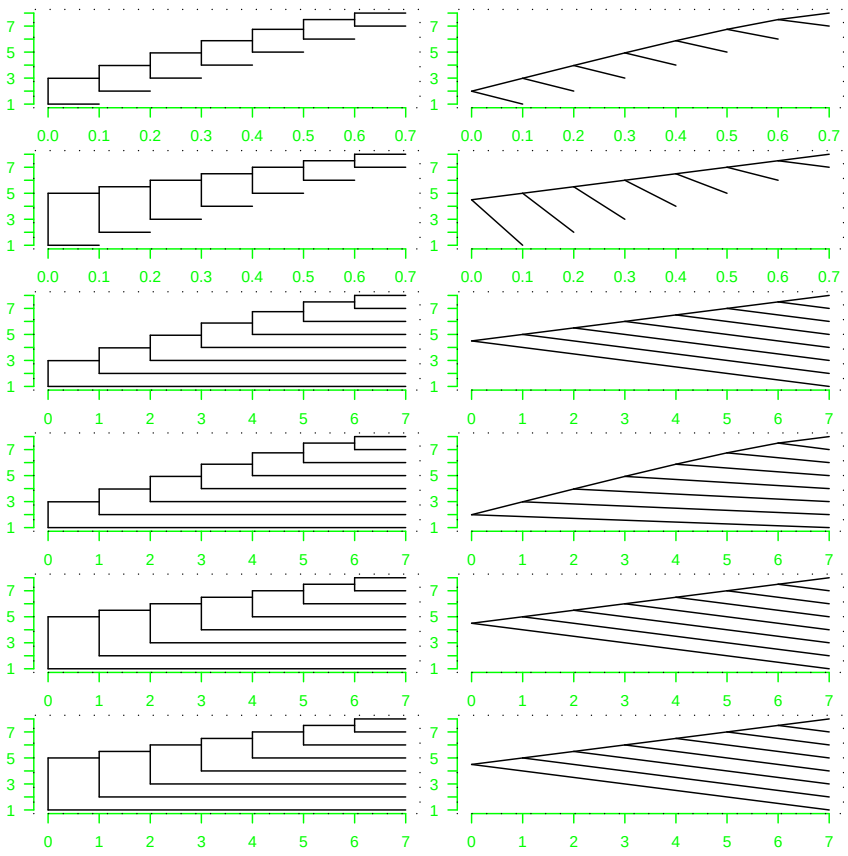
Step 1. The option `type` specifies the shape of the tree plot: five values are possible, `"phylogram"`, `"cladogram"`, `"fan"`, `"unrooted"`, and `"radial"` (the last one is not considered in this document). The first three types are valid representations for rooted trees, while the fourth one should be selected for unrooted trees.

The node coordinates depend also on whether the tree has branch lengths or not, and on the options `node.pos` and `node.depth`. This is illustrated below using a tree with eight tips and all branch length equal to one (these options have little effect if the tree has only three tips):

```
> tr <- compute.brlen(stree(8, "l"), 0.1)
> tr$tip.label[] <- ""
```

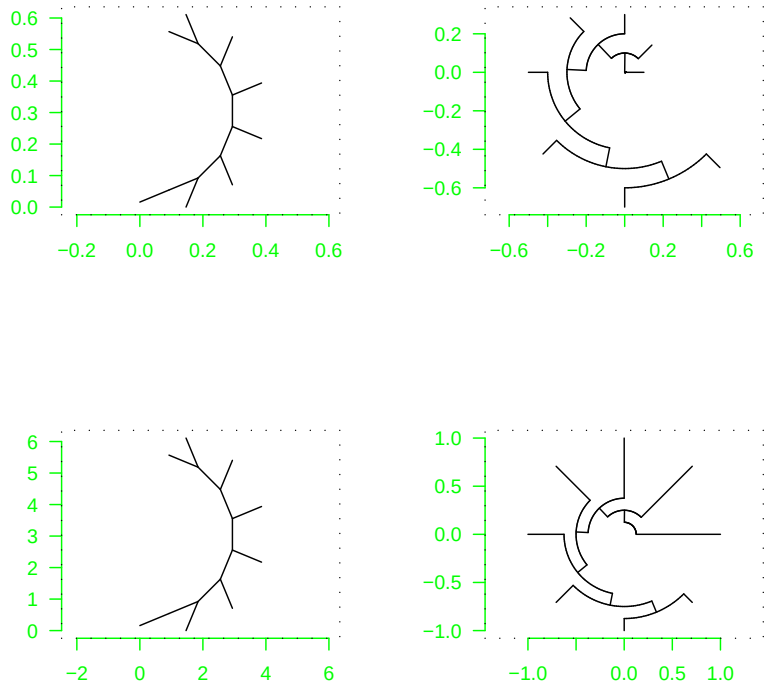
We now draw this tree using the option `type = "phylogram"` (first column of plots) or `type = "cladogram"` (second column) and different options:

```
> layout(matrix(1:12, 6, 2))
> par(mar = c(2, 2, 0.3, 0))
> for (type in c("p", "c")) {
+     plot(tr, type); foo()
+     plot(tr, type, node.pos = 2); foo()
+     plot(tr, type, FALSE); foo()
+     plot(tr, type, FALSE, node.pos = 1, node.depth = 2); foo()
+     plot(tr, type, FALSE, node.pos = 2); foo()
+     plot(tr, type, FALSE, node.pos = 2, node.depth = 2); foo()
+ }
```



Some combinations of options may result in the same tree shape as shown by the last two rows of trees. For unrooted and circular trees, only the option `use.edge.length` has an effect on the layout and/or the scales of the axes:

```
> layout(matrix(1:4, 2, 2))
> par(las = 1)
> plot(tr, "u"); foo()
> plot(tr, "u", FALSE); foo()
> plot(tr, "f"); foo()
> plot(tr, "f", FALSE); foo()
```



Step 2. In the graphics package, text are printed with a fixed size, which means that whether you draw a small tree or a large tree, on a small or large device, the labels will have the same size. However, before anything is plotted or drawn on the device it is difficult to find the correspondence between this size (in inches) and the user coordinates used for the node coordinates. Therefore, the following steps are implemented to determine the limits on the x-axis:

1. Find the width of the device in inches (see Sect. 7.2).
2. Find the widths of all labels in inches: if at least one of them is wider than the device, assign two thirds of the device for the branches and one third to the tip labels. (This makes sure that by default the tree is visible in the case there are very long tip labels.)
3. Otherwise, the space allocated to the tip labels is increased incrementally until all labels are visible on the device.

The limits on the y-axis are easier to determine since it depends only on the number of branches in the tree. The limits on both axes can be changed manually with the options `x.lim` and `y.lim` which take one or two values: if only one value is given this will set the rightmost or uppermost limit, respectively; if two values are given these will set both limits on the respecive axis.[2]

By default, there is no space between the tip labels and the tips of the terminal branches; however, text strings are printed with a bounding box around them making sure there is actually a small space (besides, the default font is italics making this space more visible). The option `label.offset` (which is 0 by default) makes possible to add an explicit space between them (this must be in user coordinates).

Step 3. For rooted trees, only 90° rotations are supported using the option `direction`.[3] For unrooted (`type = "u"`) and circular (`type = "fan"`) trees, full rotation is supported with the option `rotate.tree`. If these options are used, the tip labels are not rotated. Label rotation is controlled by other options: `srt`[4] for all trees, and `lab4ut` for unrooted trees.

Step 4. These can be fully controlled with the options `x.lim` and `y.lim`. Note that the options `xlim` and `ylim` *cannot* be used from `plot.phylo`.

Step 5. If the options `plot = FALSE` is used, then steps 6 and 7 are not performed.

2.2 Computations

As we can see from the previous section, a lot of computations are done before a tree is plotted. Some of these computations are performed by special functions accessible to all users, particularly the three functions used to calculate the node coordinates. First, two functions calculate "node depths" which are the coordinates of the nodes on the x-axis for rooted trees:

```
> args(node.depth.edgelength)
function (phy)
NULL

> args(node.depth)
function (phy, method = 1)
NULL
```

Here, `phy` is an object of class `"phylo"`. The first function uses edge lengths to calculate these coordinates, while the second one calculates these coordinates proportional to the number of tips descending from each node (if `method = 1`), or evenly spaced (if `method = 2`).

The third function is `node.height` and is used to calculate "node heights", the coordinates of the nodes on the y-axis:

```
> args(node.height)
function (phy, clado.style = FALSE)
NULL
```

If `clado.style = TRUE`, the node heights are calculated for a "triangular cladogram" (see figure above). Otherwise, by default they are calculated to fall in the middle of the vertical

[2] These two options differ from their standard counterparts `xlim` and `ylim` which always require two values.

[3] To have full control of the tree rotation, the option 'rotate' in LaTeX does the job very well.

[4] `srt` is for *string rotation*, not to be confused with the function `str` to print the *structure* of an object.

segments with the default `type = "phylogram"`.[5]

For unrooted trees, the node coordinates are calculated with the "equal angle" algorithm described by Felsenstein [2]. This is done by an internal function which arguments are:

```
> args(unrooted.xy)
function (Ntip, Nnode, edge, edge.length, nb.sp, rotate.tree)
NULL
```

There are three other internal functions used to plot the segments of the tree after the above calculations have been performed (steps 1–4 in the previous section):

```
> args(phylogram.plot)
function (edge, Ntip, Nnode, xx, yy, horizontal, edge.color = NULL,
    edge.width = NULL, edge.lty = NULL, node.color = NULL, node.width = NULL,
    node.lty = NULL)
NULL


> args(cladogram.plot)
function (edge, xx, yy, edge.color, edge.width, edge.lty)
NULL


> args(circular.plot)
function (edge, Ntip, Nnode, xx, yy, theta, r, edge.color, edge.width,
    edge.lty)
NULL
```

Although these four functions are not formally documented, they are anyway exported because they are used by several packages outside ape.

3 The `plot.phylo` Function

The `plot` method for `"phylo"` objects follows quite closely the R standard practice. It has a relatively large number of arguments: the first one (`x`) is mandatory and is the tree to be drawn. It is thus not required to name it, so in practice the tree `tr` can be plotted with the command `plot(tr)`. All other arguments have default values:

```
> args(plot.phylo)
function (x, type = "phylogram", use.edge.length = TRUE, node.pos = NULL,
    show.tip.label = TRUE, show.node.label = FALSE, edge.color = NULL,
    edge.width = NULL, edge.lty = NULL, node.color = NULL, node.width = NULL,
    node.lty = NULL, font = 3, cex = par("cex"), adj = NULL,
    srt = 0, no.margin = FALSE, root.edge = FALSE, label.offset = 0,
    underscore = FALSE, x.lim = NULL, y.lim = NULL, direction = "rightwards",
    lab4ut = NULL, tip.color = par("col"), plot = TRUE, rotate.tree = 0,
    open.angle = 0, node.depth = 1, align.tip.label = FALSE,
    ...)
NULL
```

[5] It may be good to remind here than these segments, vertical since `direction = "rightwards"` is the default, are not part of the edges of the tree.

The second and third arguments are the two commonly used in practice, so they can be modified without explicitly naming them like in the above examples. Besides, `"cladogram"` can be abbreviated with `"c"`, `"unrooted"` with `"u"`, and so on. For the other arguments, it is better to name them if they are used or modified (e.g., `lab4ut = "a"`).

3.1 Overview on the Options

The logic of this long list of options is double: the user can modify the aspect of the tree plot, and/or use some of these options to display some data in association with the tree. Therefore, the table below group these options into three categories. The following two sections show how data can be displayed in connection to the tips or to the branches of the tree.

Aspect of the tree	Attributes of the labels	Attributes of the edges
`type`	`show.tip.label`	`edge.color`
`use.edge.length`	`show.node.label`	`edge.width`
`node.pos`	`font`	`edge.lty`
`x.lim`	`tip.color`	
`y.lim`	`cex`	
`direction`	`adj`	
`no.margin`	`underscore`	
`root.edge`	`srt`	
`rotate.tree`	`lab4ut`	
`open.angle`	`label.offset`	
`node.depth`	`align.tip.label`	

3.2 Connecting Data to the Tips

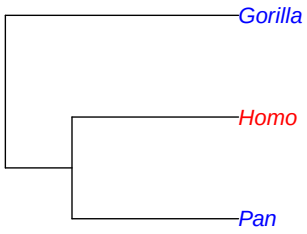
It is common that some data are associated with the tips of a tree: body mass, population, treatment, ... The options `font`, `tip.color`, and `cex` make possible to show this kind of information by changing the font (normal, bold, italics, or bold-italics), the colour, or the size of the tip labels, or any combination of these. These three arguments work in the usual R way: they can a vector of any length whose values are eventually recycled if this length is less than the number of tips. This makes possible to change all tips if a single value is given.

For instance, consider the small primate tree where we want to show the geographic distributions stored in a factor:

```
> geo <- factor(c("Africa", "World", "Africa"))
```

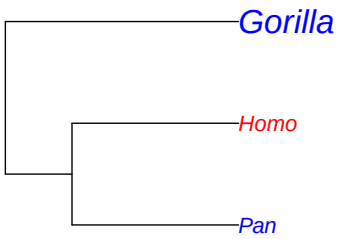
We can define a color for each region and use the above factor as a numeric index vector and pass it to `tip.color`:

```
> (mycol <- c("blue", "red")[geo])
[1] "blue" "red"  "blue"

> plot(mytr, tip.color = mycol)
```



The values must be in the same order than in the vector of tip labels, here `mytr$tip.label`. Reordering can be done in the usual R way (e.g., with `names` or with `row.names` if the data are in a data frame). This can be combined with another argument, for instance to show (relative) body size:

```
> par(xpd = TRUE)
> plot(mytr, tip.color = mycol, cex = c(1, 1, 1.5))
```



The function `def` gives another way to define the above arguments given a vector of labels (`x`):

```
> args(def)
function (x, ..., default = NULL, regexp = FALSE)
NULL
```

The '`...`' are arguments separated by commas of the form *`<label>`* `=` *`<value>`* with *`<label>`* being one of the labels in `x` and *`<value>`* the value which will be given to this element, whereas the value `default` will be given to the others. This default value is either 1 if *`<value>`* is numeric, or `"black"` if *`<value>`* is character. The above set of colours could have thus be defined with:

```
> mycol2 <- def(mytr$tip.label, Homo = "red", default = "blue")
> identical(mycol, mycol2)
[1] TRUE
```

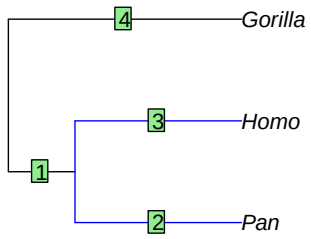
The function `tiplabels`, presented below, gives more possibilities to display data on the tips of a tree.

3.3 Connecting Data to the Branches

The three options in the third column of the above table control the aspect of the branches of the tree. Like the options for the tips, the arguments are recycled. The values given as argument(s) must be in the same order than in the `edge` matrix of the tree. There are several ways to find these numbers.

The function `which.edge` returns the row indices of the `edge` matrix given a list of tip labels so that the edges define the clade of the tips. The returned indices can then be used to change the aspect of these branches:

```
> (i <- which.edge(mytr, c("Homo", "Pan")))
[1] 2 3


> co <- rep("black", Nedge(mytr))
> co[i] <- "blue"
> par(xpd = TRUE)
> plot(mytr, edge.col = co)
> edgelabels()
```



These indices can be displayed on the tree simply by calling `edgelabels()` without option as illustrated on the previous plot (this function is further explained in the next section).

It is possible for a function to return data or values that are directly indexed with respect to the rows of the `edge` matrix. One example is `chronos()` which returns an ultrametric tree with an additional numeric vector giving the estimated substitution rate for each branch of the chronogram. This vector can be used directly to define the aspect of the branches when plotting the chronogram.

4 Annotating Trees

Once a tree has been plotted, it is possible to add graphical elements with the low-level plotting functions[6] in R: `text`, `segments`, `points`, `arrows`, `rect`, and `polygon` (all in the package graphics). These functions require to give the coordinates where to draw these elements which may be a bit difficult when plotting a tree since the axes are not drawn by default. In practice, the function `locator` may be helpful here.

ape has five specialized functions that also facilitate this task: three of them add elements on the tree, and the two others do so beside the tree. Additionally, the functions `axisPhylo` and `add.scale.bar` display scale information of the tree branches.

4.1 Scales

The function `axisPhylo` draws a time axis assuming the tips represent the present time and the time scales backward towards the root. For non-ultrametric trees, the most distant tip from the root is taken to represent present time. There are a few options:

```
> args(axisPhylo)
function (side = NULL, root.time = NULL, backward = TRUE, ...)
NULL
```

The option `backward` and `root.time` change the settings of the scale, `side` has the same meaning than in `axis()`, and the extra arguments ('`...`') are passed to `text()`.

`add.scale.bar` draws a scale bar of the branch length, by default on the lower left corner of the plot. Several options control the position and/or the formatting:

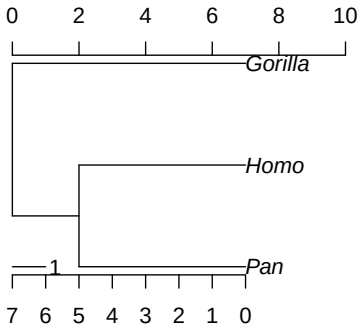
```
> args(add.scale.bar)
function (x, y, length = NULL, ask = FALSE, lwd = 1, lcol = "black",
    ...)
NULL
```

If `ask = TRUE`, the function calls `locator()` and the user is asked to click on the plot to specify where to draw the scale bar.

The example below illustrates both functions and draws the standard x-axis above the tree:

```
> plot(mytr)
> add.scale.bar()
> axisPhylo()
> axis(3)
```

[6] These functions add graphical elements to an existing plot, by contrast to high-level plotting functions which create a new plot.



4.2 Tips, Nodes, and Edges

The three functions `nodelabels`, `tiplabels`, and `edgelabels` have almost identical options to add annotations on the nodes, tips, or edges of a plotted tree:

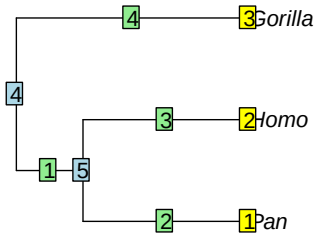
```
> args(nodelabels)
function (text, node, adj = c(0.5, 0.5), frame = "rect", pch = NULL,
    thermo = NULL, pie = NULL, piecol = NULL, col = "black",
    bg = "lightblue", horiz = FALSE, width = NULL, height = NULL,
    ...)
NULL


> args(tiplabels)
function (text, tip, adj = c(0.5, 0.5), frame = "rect", pch = NULL,
    thermo = NULL, pie = NULL, piecol = NULL, col = "black",
    bg = "yellow", horiz = FALSE, width = NULL, height = NULL,
    offset = 0, ...)
NULL


> args(edgelabels)
function (text, edge, adj = c(0.5, 0.5), frame = "rect", pch = NULL,
    thermo = NULL, pie = NULL, piecol = NULL, col = "black",
    bg = "lightgreen", horiz = FALSE, width = NULL, height = NULL,
    date = NULL, ...)
NULL
```

These functions differ in the second argument which specifies the positions where the annotations should be printed or drawn. If these functions are used without any argument, they print the numbers of the nodes, tips, or edges, respectively, used in the `"phylo"` object:[7]

```
> par(xpd = TRUE)
> plot(mytr)
> nodelabels()
```

[7]The structure of objects of class `"phylo"` is described on `ape`'s web site: `https://emmanuelparadis.github.io/ape_development.html`

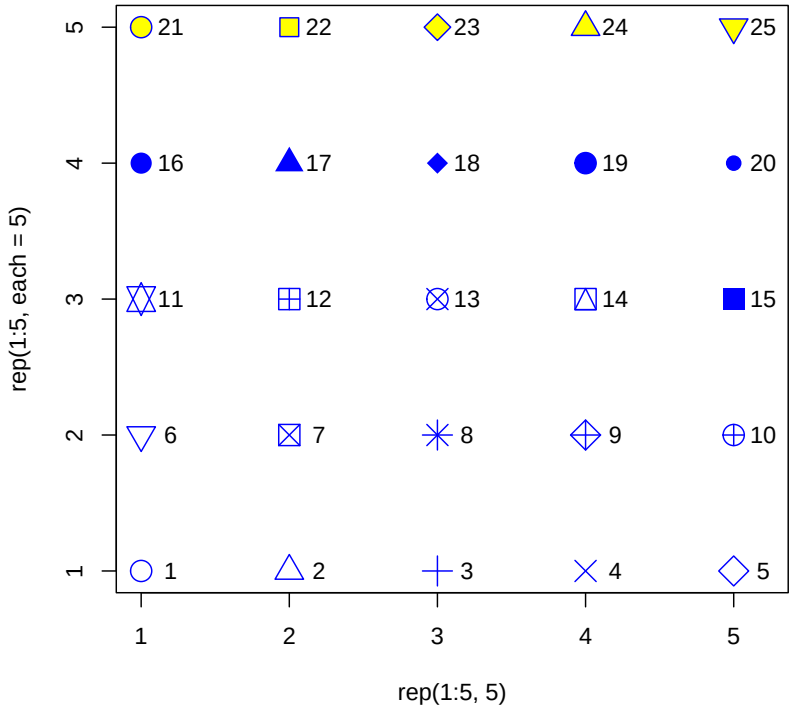
```
> tiplabels()
> edgelabels()
```



There are a lot of possibilities offered by these functions: a range of examples are given in the help page which can be displayed with `example(nodelabels)` and are not repeated here. A common application of these function, and in particular `nodelabels`, is to diplay pie charts of ancestral values. Several functions in `ape` (e.g., `ace`), or in other packages, return their ancestral reconstructions indexed along the nodes of the tree so that using `nodelabels` is usually straightforward.

Like for `plot.phylo`, the options can be used for either showing some data, or formatting the annotations. The option `text` works in the same way than the standard function of the same name; formatting of the character strings is done with the option `col` and others (e.g., `font` or `cex` can be passed with '`...`'). A frame is drawn around the character strings unless `frame = "none"`. The option `pch` is used to plot standard plotting symbols, this can take the values from 1 to 25, as shown below:

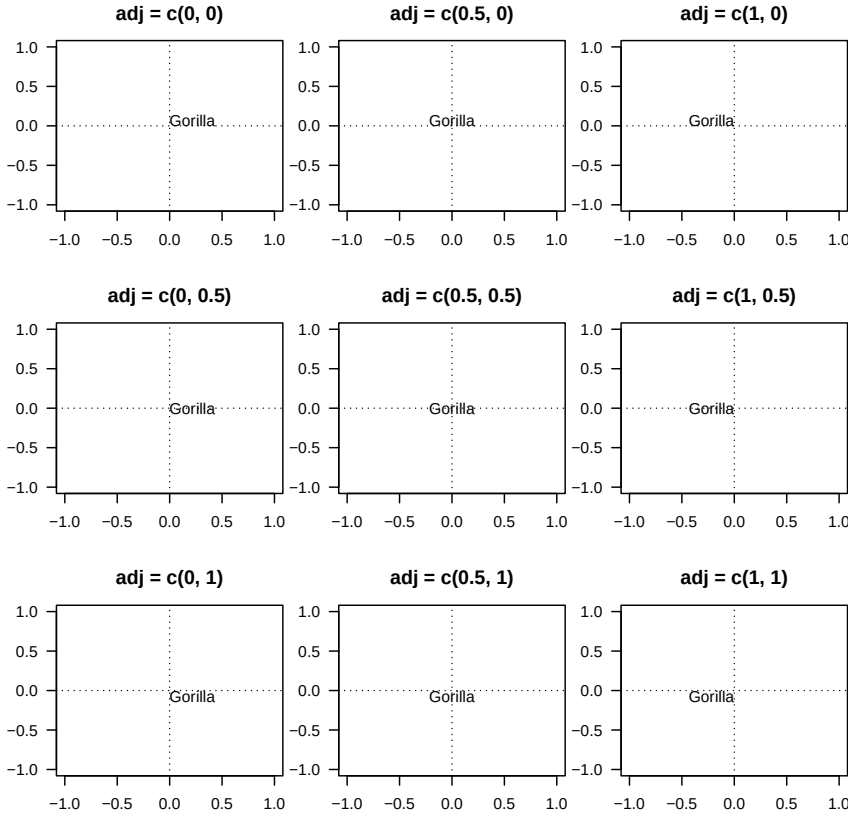
```
> plot(rep(1:5, 5), rep(1:5, each = 5), pch = 1:25, xlim = c(1, 5.2),
+      col = "blue", bg = "yellow", cex = 2)
> text(rep(1:5, 5) + 0.2, rep(1:5, each = 5), 1:25)
```



It should be kept in mind that these three functions are low-level plotting commands, so they can be called as many times as the user wishes with possibly different formatting options which could be easier than using a single call (see the first example in `?nodelabels`).

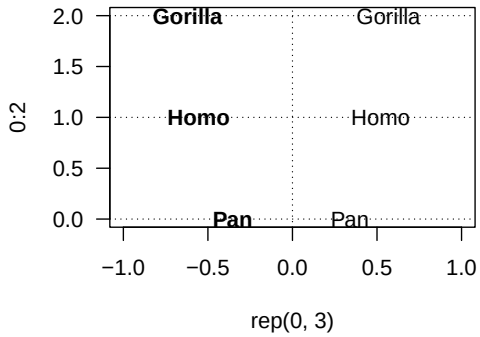
The option `adj` is used to set the adjustment of the character strings passed to `text` (and also the plotting symbols drawn with `pch`, unlike the function `points`). This can used to print several series of numbers around nodes (see `?nodelabels`). The adjustment is made with respect to the size of each string, so useful values are 0 (left- or bottom-aligned), 0.5 (centred), and 1 (right- or top-aligned):

```
> v <- c(0, 0.5, 1)
> layout(matrix(1:9, 3, 3))
> par(mar = c(3, 3, 3, 0), las = 1)
> for (i in v) {
+     for (j in v) {
+         plot(0, 0, "n", main = paste0("adj = c(", i, ", ", j, ")"))
+         abline(v = 0, h = 0, lty = 3)
+         text(0, 0, "Gorilla", adj = c(i, j))
+     }
+ }
```



Other values are accepted but they will likely result in strings badly aligned if they have different (printed) widths:

```
> plot(rep(0, 3), 0:2, "n", las = 1)
> abline(v = 0, h = 0:2, lty = 3)
> text(0, 0:2, mytr$tip.label, adj = -1)
> text(0, 0:2, mytr$tip.label, adj = 2, font = 2)
```



The options `pie` and `thermo`, by contrast to the previous ones, take a matrix whose values are taken as proportions which are displayed as piecharts or as thermometres. The colours of the "slices" of the pies, or the "levels" of the thermometers, are specified with `piecol` (by

default, the function `rainbow` is used to define these colours). The options `horiz`, `width`, and `height` are used to format the aspect of the thermometres.

4.3 Data Plots Besides Trees

The two following functions add a graph beside and in connection to a plotted tree (or around it if it is circular):

```
> args(phydataplot)
function (x, phy, style = "bars", offset = 1, scaling = 1, continuous = FALSE,
    width = NULL, legend = "below", funcol = rainbow, ...)
NULL

> args(ring)
function (x, phy, style = "ring", offset = 1, ...)
NULL
```

Their usage is a bit more complicated than the previous functions because it is usually needed to leave some space when plotting the tree using `x.lim` and/or `y.lim` (see above). It may not be straightforward to find how much space must be left: in practice the simplest solution is to try different values, possibly using `axis` as shown above, or using the (invisibly) returned value of `plot.phylo`. Once some values of these limits are found for a given data set, they are expected to work similarly accross different graphical devices (`pdf()`, `png()`, ...) For more sophisticated or automated solutions, it is possible to calculate these limits a priori (see below).

These two functions use the labels of the data (given by their `names` if the data are in a vector, their `rownames` if in a matrix, or their `row.names` if in a data frame) to match with the tip labels of the tree (argument `phy`). Thus, it is not needed to reorder the data if they have such (row)names.

The option `style` can take seven different values giving a lot of possibilities. Furthermore, further arguments can be given which are then passed to a function of the graphics package giving flexibility to customize the appearance of the plot. The table below gives the correspondence between the values of `style` and the graphics functions called by `phydataplot`:[8]

`style`	Extra arguments ('...') passed to:
`"bars"`	`barplot`
`"segments"`	`segments`
`"image"`	`image`
`"arrows"`	`fancyarrows`
`"boxplot"`	`bxp`
`"dotchart"`	`points`
`"mosaic"`	`rect`

A simple example is given here with the primate tree to illustrate what can be done with `type = "mosaic"`. We build a matrix with ten columns filled with values 1 to 6, and give names to its rows and columns:

```
> x <- matrix(1:6, 3, 10)
> dimnames(x) <- list(c("Homo", "Gorilla", "Pan"), LETTERS[1:ncol(x)])
```

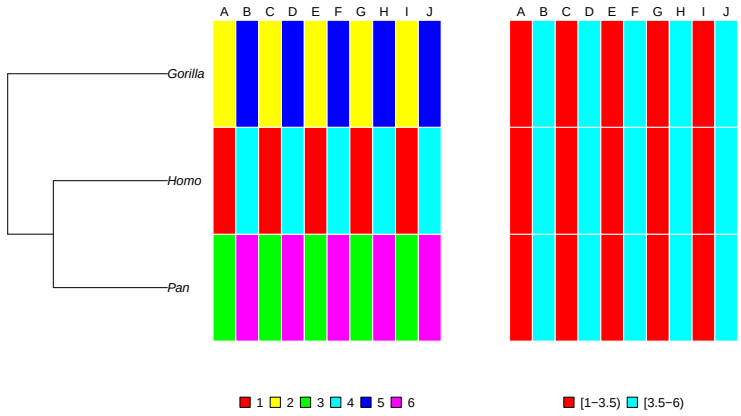
[8] `fancyarrows()` is in `ape`: it has the same arguments than `arrows()` plus the option `type` which can be either `"triangle"` (the default) or `"harpoon"`.

Note that these data are not ordered like the tip labels of the tree:

```
> x
        A B C D E F G H I J
Homo    1 4 1 4 1 4 1 4 1 4
Gorilla 2 5 2 5 2 5 2 5 2 5
Pan     3 6 3 6 3 6 3 6 3 6

> mytr$tip.label
[1] "Pan"     "Homo"    "Gorilla"
```

It is possible to treat these values as categories or as continuous variables. With `phydataplot`, this is controlled by the option `continuous` which is either a logical value (`FALSE` by default, so the variables are considered as categorical), or an integer specifying the number of classes used to represent the data (if `continuous = TRUE`, then ten classes are used). Because we call `phydataplot` twice, we use the option `offset` which sets the space between the tree and the graph:

```
> par(mar = c(10, 2, 5, 5))
> plot(mytr, x.lim = 30)
> phydataplot(x, mytr, "m", border = "white", offset = 2, width = 1)
> phydataplot(x, mytr, "m", border = "white", offset = 15,
+             width = 1, continuous = 2)
>
```

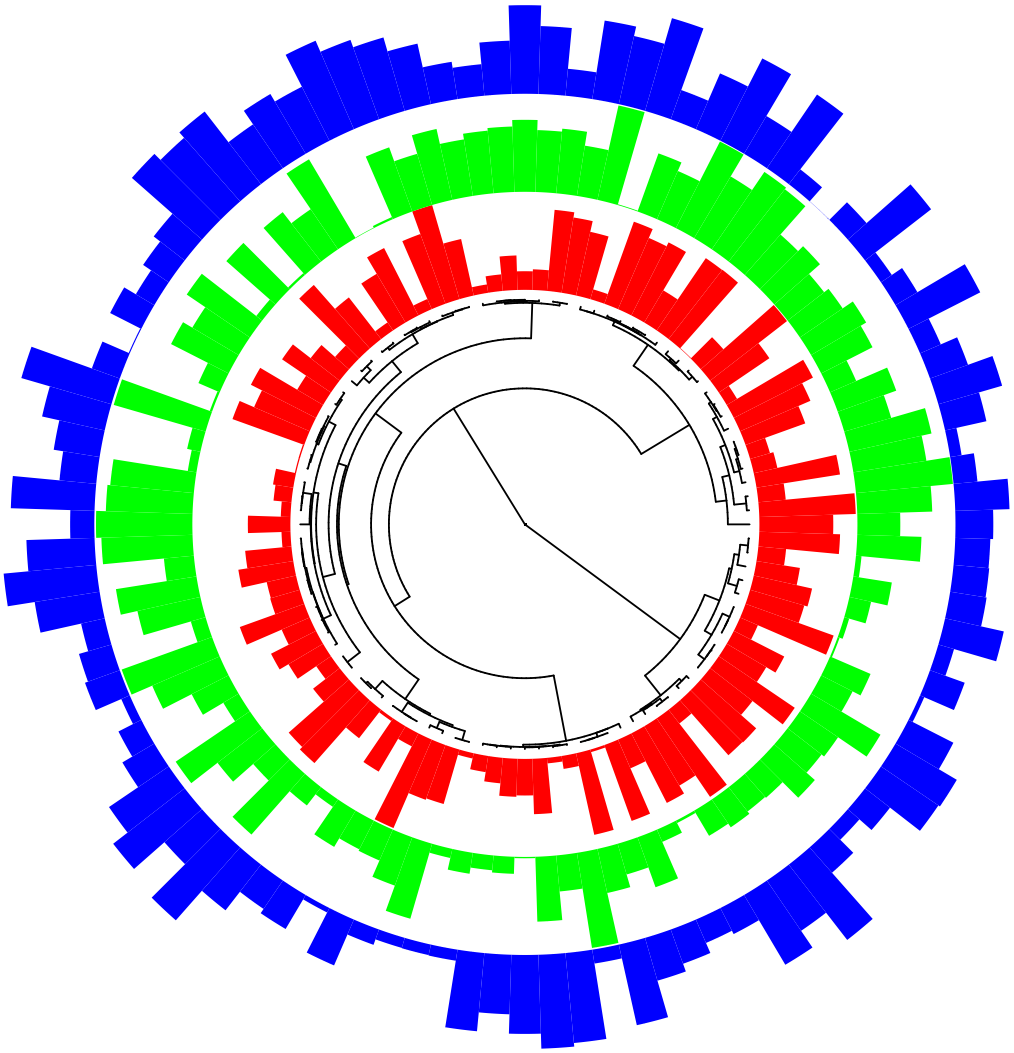


The options `border = "white"` and `width = 1` help to have a better looking figure. The colour scheme used for the first mosaic makes easy to check that the data were correctly matched between the rows of the matrix and tips of the tree.

For circular trees, `offset` is the key to combine different 'rings' of data, like in the next example with a random coalescent tree with 100 tips and three variables from a uniform distribution:

```
> n <- 100
> p <- 3
> set.seed(3)
> tr <- rcoal(n)
> x <- matrix(runif(p * n), n, p)
```

```
> rownames(x) <- tr$tip.label
> COL <- c("red", "green", "blue")
> par(xpd = TRUE)
> plot(tr, "f", x.lim = c(-5, 5), show.tip.label = FALSE, no.margin = TRUE)
> for (j in 1:p) ring(x[, j], tr, offset = j - 1 + 0.1, col = COL[j])
```



Like for the functions discussed in the previous section, `phydataplot` and `ring` offer a very wide range of possibilities which are (partially) illustrated on their help page (`example(phydataplot)`) and are not repeated here.

5 Specialized Functions

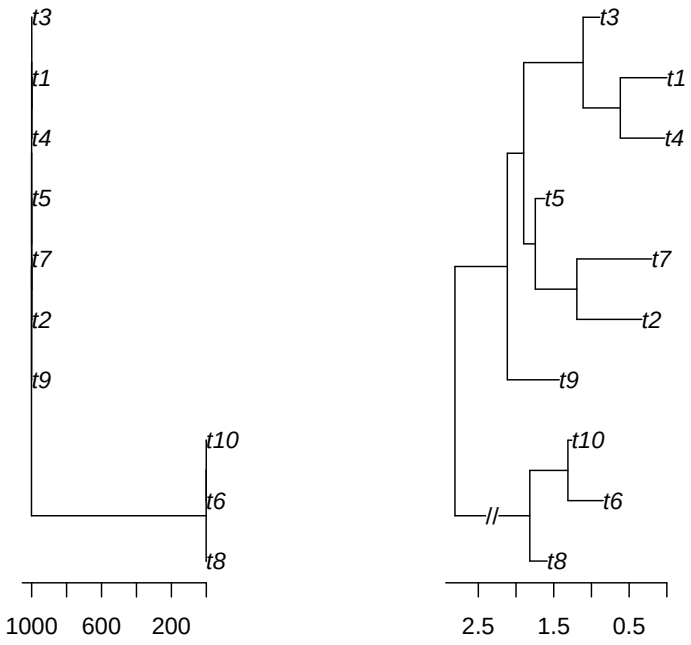
We see in this section several functions, built from `plot.phylo`, that each have a specific purpose.

5.1 Function `plotBreakLongEdges`

It happens sometimes that, in a tree, one (or several) branch is much longer than the others, for instance the branch between the ingroup and a very distant outgroup. When

plotting such a tree, the contrast between the shortest branches will likely not be visible. `plotBreakLongEdges()` shortens the longest branch of the tree and represents it with a broken segment. We illustrate this with a random tree where the first branch is given a length of 1000 (all others have random lengths between 0 and 1):

```
> tree <- rtree(10)
> tree$edge.length[1] <- 1000
> layout(matrix(1:2, 1))
> plot(tree); axisPhylo()
> plotBreakLongEdges(tree); axisPhylo()
```



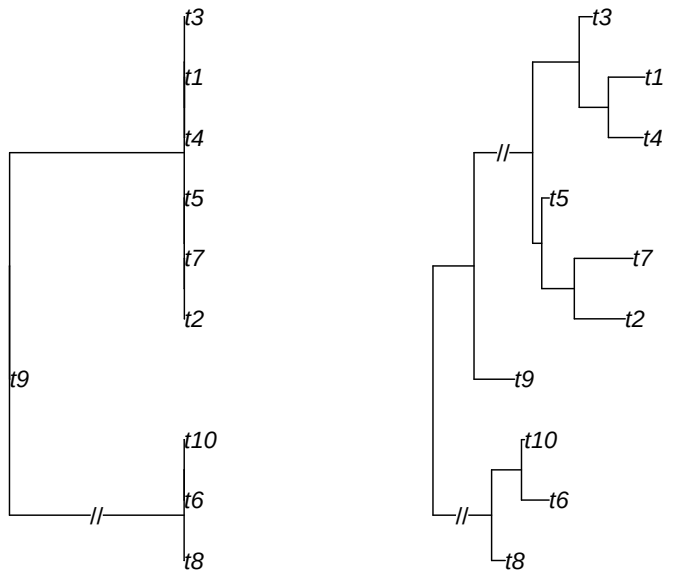
This function has the option `n = 1` which specified the number of branches to be broken. This can be automated in some way, for instance if we want to find how many branches in the above tree are longer than the mean plus twice the standard-deviation:

```
> el <- tree$edge.length
> sum(el > mean(el) + 2 * sd(el))
[1] 1
```

We test with approach after changing another branch length:

```
> tree$edge.length[8] <- 1000
> el <- tree$edge.length
> sum(el > mean(el) + 2 * sd(el))
[1] 2
```

```
> layout(matrix(1:2, 1))
> plotBreakLongEdges(tree)
> plotBreakLongEdges(tree, n = 2)
```



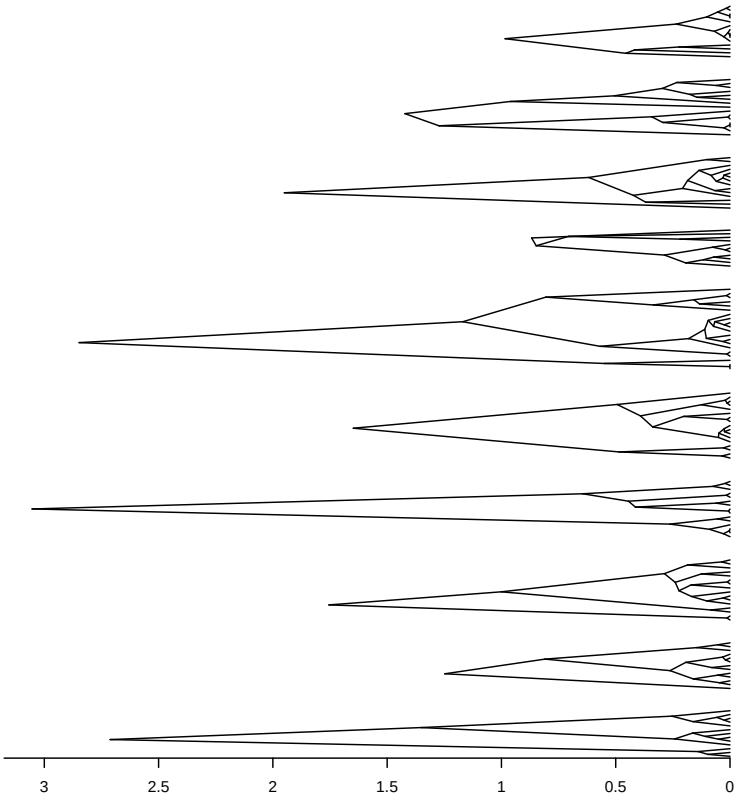
5.2 Function `drawSupportOnEdges`

The phylogenetic bootstrap assessed the uncertainty of a phylogenetic tree inferred from a method (parsimony, maximum likehood, neighbour-joining, ...) If the inference method outputs an unrooted tree (e.g., ML, NJ), then the bootstrap values relate to the internal branches of the tree which define bipartitions (also known as splits). Following the review by Czech et al. [1], two modifications were done in `ape`. First, the option `edgelabel` was added to the function `root` so that bootstrap values, normally attached to the nodes (see `?boot.phylo`) are considered attached to the edges. Second, the function `drawSupportOnEdges`, derived from `edgelabels()`, was written so that bootstrap values are correctly printed on the internal branches of a tree.

5.3 Function `kronoviz`

`kronoviz()` plots a series of ultrametric trees so that all tips are aligned on the same line and the scale is the same for all trees:

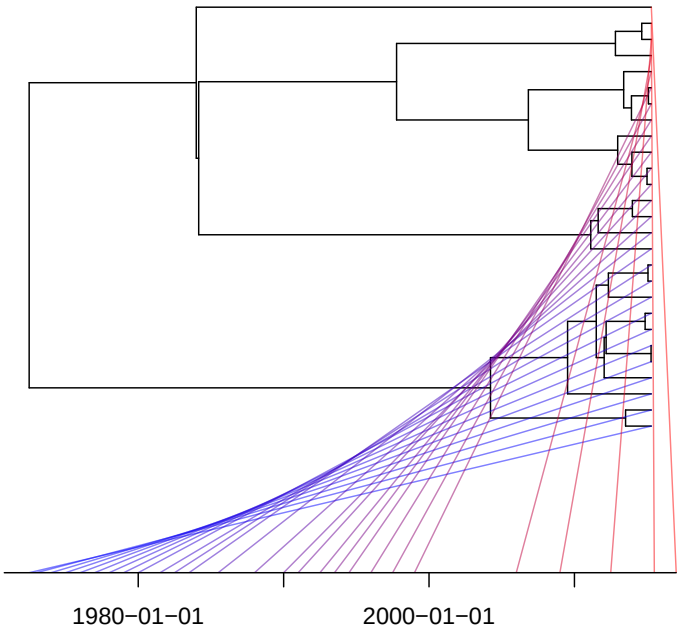
```
> TR <- replicate(10, rcoal(sample(11:20, size = 1)), simplify = FALSE)
> kronoviz(TR, type = "c", show.tip.label = FALSE)
```



5.4 Function `plotTreeTime`

If some dates are associated with the tips of a tree (e.g., from an epidemiological study), the function `plotTreeTime` can plot both information where the branch lengths are in their own units and the x-axis is drawn as a time axis. For example, with a simulated random coalescent tree and dates taken from the leap seconds in R:

```
> dates <- as.Date(.leap.seconds)
> tr <- rcoal(length(dates))
> plotTreeTime(tr, dates)
```

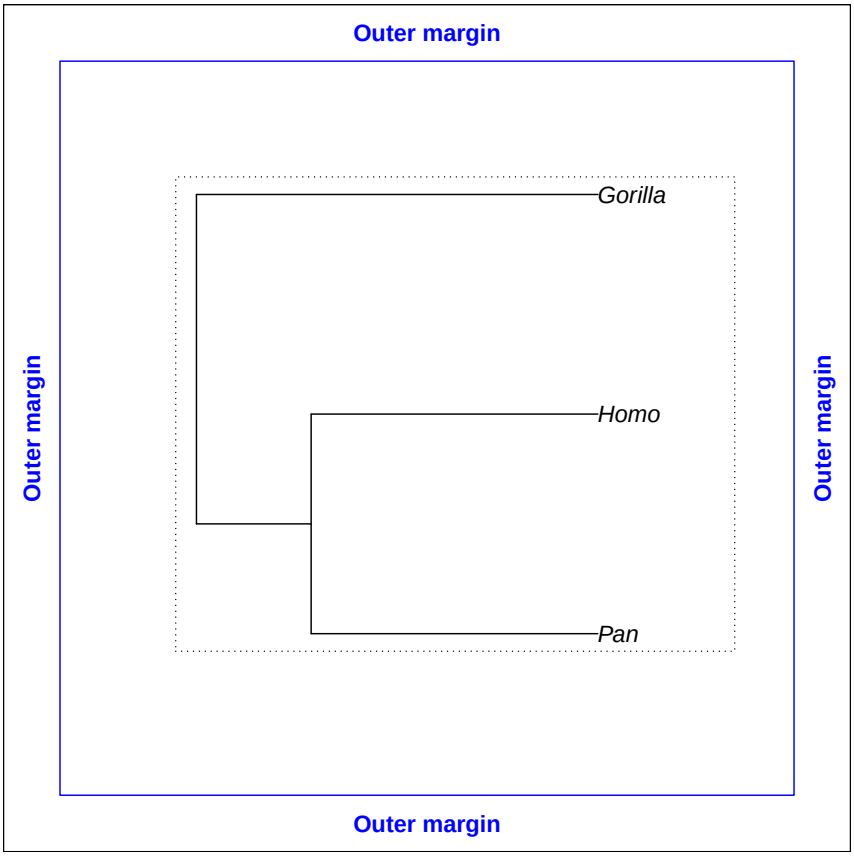


6 Geometry and Composite Figures

6.1 Single Plotting Region

We now see some details of the geometry of a figure when plotting a tree. We first set the outer margins (graphical parameter `oma`) which are by default zero and then display the inner and outer margins by calling `box()` three times with the appropriate options:

```
> par(oma = rep(2, 4))
> plot(mytr)
> box(lty = 3)
> box("figure", col = "blue")
> for (i in 1:4)
+     mtext("Outer margin", i, 0.5, outer = TRUE, font = 2, col = "blue")
> box("outer")
```

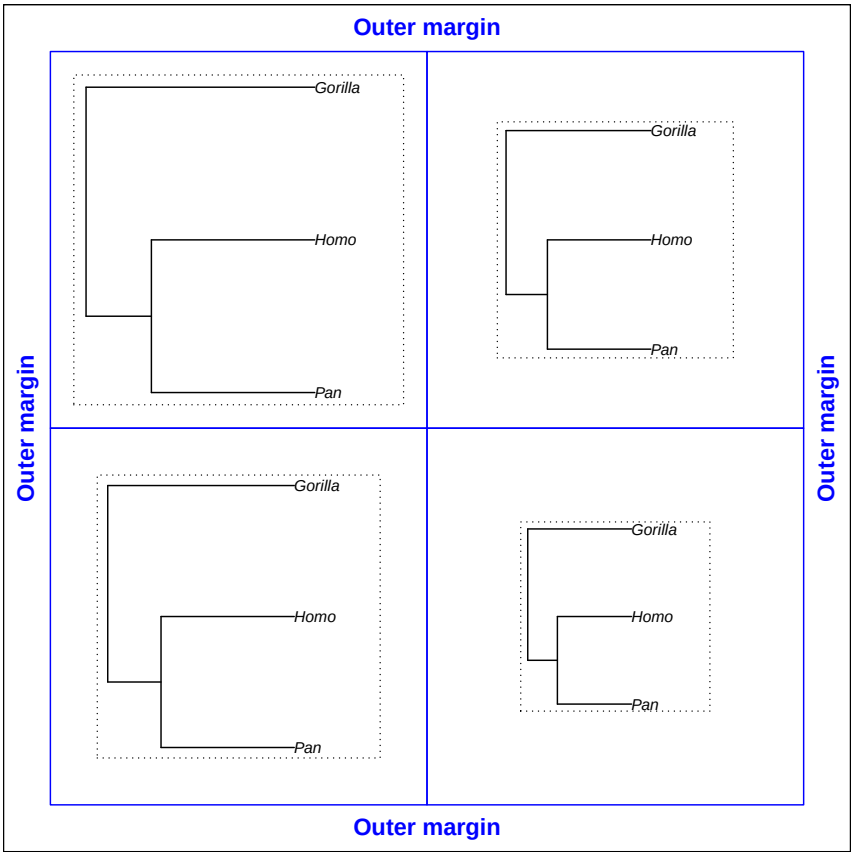


It is, of course, also possible to changer the inner margins with `par(mar = ....)`. Note that doing `plot(mytr, no.margin = TRUE)` is identical to calling `par(mar = rep(0, 4))` before doing `plot(mytr)`.

6.2 Multiple Layout

It is possible to draw several trees on the same figure by first splitting the graphical device with `layout`. In that case, each plot will have its own set of inner margins (which can be different like in the next example) whereas the outer margins are the "global" margins of the figure:

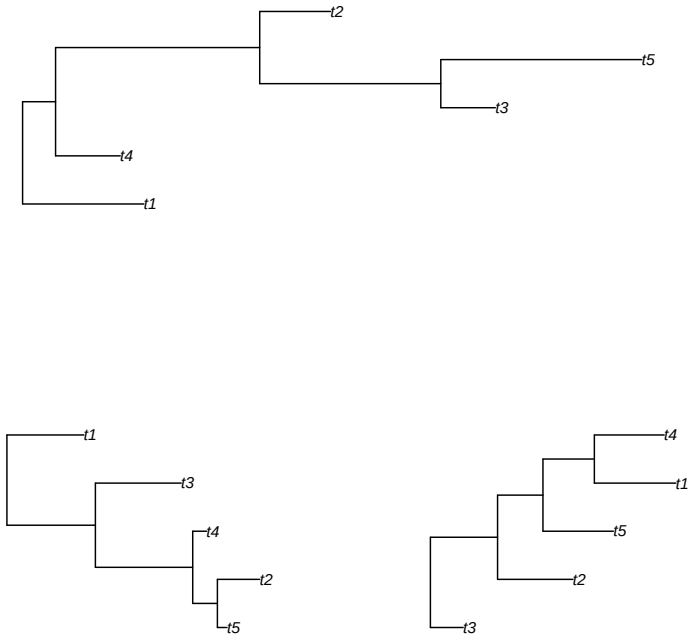
```
> layout(matrix(1:4, 2, 2))
> par(oma = rep(2, 4))
> for (i in 1:4) {
+     par(mar = rep(i, 4))
+     plot(mytr)
+     box(lty = 3)
+     box("figure", col = "blue")
+ }
> for (i in 1:4)
+     mtext("Outer margin", i, 0.5, outer = TRUE, font = 2, col = "blue")
> box("outer")
```



An alternative to split a graphical device is to call `par(mfcol = c(2, 2))`. However, I recommend to use `layout` which is much more flexible and powerful:

- `layout` has the options `width` and `height` which give the (relative) sizes of the rows and columns of plots.
- The matrix given as main argument to `layout` is a symbolic representation of the plot layout: a single plot can span on several cells of the matrix; for instance, `layout(matrix(c(1, 2, 1, 3), 2, 2))` will specify three plots on the device with the first one spanning on the first row, and the second row split into two plots:

```
> layout(matrix(c(1, 2, 1, 3), 2, 2))
> for (i in 1:3) plot(rtree(5))
```



7 Building Your Own Code and Functions

In this section, we examine two ways to build code or functions using ape's graphical functionalities.

7.1 Getting the Recorded Parameters

Every time `plot.phylo` is called, some data and parameters are saved in a specific R environment[9] which are then used by functions such as `nodelabels`. This environment is called `.PlotPhyloEnv` and is accessible to all users so that the list `last_plot.phylo` can be obtained with:

```
> (pp <- get("last_plot.phylo", envir = .PlotPhyloEnv))
$type
[1] "phylogram"

$use.edge.length
[1] TRUE

$node.pos
[1] 1

$node.depth
[1] 1
```

[9]Here "R environment" means a part of the memory of the computer used by R with a set of objects.

```
$show.tip.label
[1] TRUE

$show.node.label
[1] FALSE

$font
[1] 3

$cex
[1] 0.83

$adj
[1] 0

$srt
[1] 0

$no.margin
[1] FALSE

$label.offset
[1] 0

$x.lim
[1] 0.00000 3.11948

$y.lim
[1] 1 5

$direction
[1] "rightwards"

$tip.color
[1] "black"

$Ntip
[1] 5

$Nnode
[1] 4

$root.time
NULL

$align.tip.label
[1] FALSE

$edge
     [,1] [,2]
[1,]    6    1
[2,]    6    7
```

```
[3,]    7    2
[4,]    7    8
[5,]    8    3
[6,]    8    9
[7,]    9    4
[8,]    9    5

$xx
[1] 0.3967457 1.7260289 2.2158455 2.9728029 2.8310192 0.0000000 0.8134181
[8] 1.3631198 1.9857132

$yy
[1] 1.0000 2.0000 3.0000 4.0000 5.0000 1.9375 2.8750 3.7500 4.5000
```

Most of the recorded parameters correspond to the arguments of `plot.phylo`. The last two vectors stored in this list, `xx` and `yy`, are the coordinates of the tips and nodes of the tree.

7.2 Overlaying Layouts

The graphics package does not have an easy way to interact with the plots on a multiple layout. For instance, after calling `layout` and after the second plot has been made, it not possible to add elements (points, arrows, ...) to the first plot which limits the possibility of connecting these plots in a graphical way. However, the command `par(new = TRUE)` offers a way to do that as described here.

The idea is to keep track of the coordinates of the nodes (and other graphical elements if needed) on the whole graphical device. However, because the plots are very likely to use different coordinate systems (e.g., due to different scales of the branch lengths) these coordinates must be converted into physical units on the graphical device (usually inches). This is possible thanks to `par()` which returns, among many parameters, the physical dimensions of the plotting region, the margins, and the device in inches with the parameters `"pin"`, `"mai"`, and `"din"`, respectively (it also has `"fin"` for the size of the figure, which excludes the outer margins, and will be needed if these outer margins are not zero). An additional graphical parameter that we will use here is given by `par("usr")` giving the extremes of the coordinates of the plotting region.

The procedure follows these steps:

1. Rescale the node coordinates of each tree plot so they are between 0 and 1.
2. Convert these coordinates in inches by multiplying them with the relevant value in `"fin"` and adding the size (also in inches) of the relevant margin. This gives the coordinates in inches in each figure (i.e., including the margins).
3. Use the geometry eventually defined by `layout` to add to the coordinates output at the previous step.

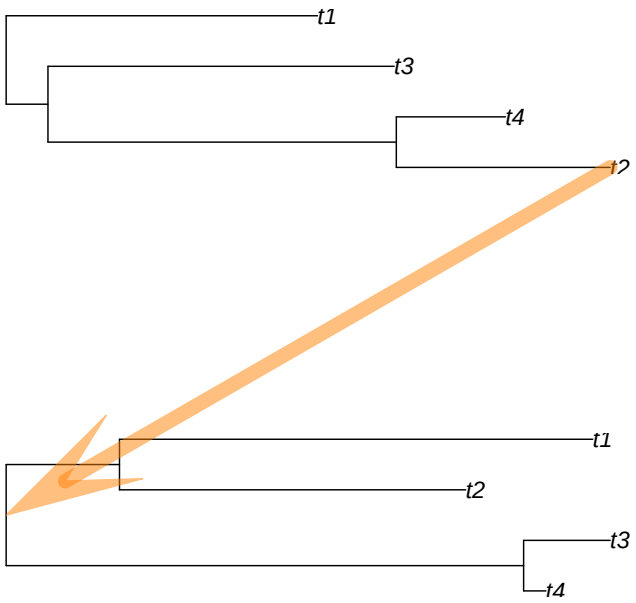
After these three steps have been performed, we obtain the coordinates of the nodes in inches over the whole device independently of the various scales used in each plot. Then, the call to `par(new = TRUE)` makes like nothing has been plotted and R will not delete what is already drawn on the device. We then call `plot(NA)` with the options `type = "n", ann = FALSE, axes = FALSE` which results in a "bare" plot. The important options here are `[x|y]lim = 0:1` which set the scale on each axis, and `[x|y]axs = "i"` which set the axes to extend exactly to the data range or the specified limits.[10]

[10] By default, `xaxs` and `yaxs` are equal to `"r"` which adds 4% on each side of the data range or limits.

Unfortunately, in a vignette like the present document, each call to `plot` opens a new device so that it is not possible to execute the above steps interactively. Instead, the next bloc of commands includes some commentaries and prints some intermediate results. In this example, two random trees are plotted and an arrow connects the first tip of the first tree to the root of the second tree:

```
> ## split the device into 2:
> layout(matrix(1:2, 2))
> ## plot the 1st tree
> plot(rtree(4))
> ## get the parameters of this first plot:
> pp <- get("last_plot.phylo", envir = .PlotPhyloEnv)
> ## keep the coordinates of the first tip:
> x <- pp$xx[1]
> y <- pp$yy[1]
> ## extremes of the coordinates on both axes:
> (pu <- par("usr"))
[1] -0.06357898  1.65305358  0.88000000  4.12000000


> ## fraction of the plot region:
> (x - pu[1]) / (pu[2] - pu[1])
[1] 0.9359244


> (y - pu[3]) / (pu[4] - pu[3])
[1] 0.03703704


> ## get the dimensions of plotting region and margins in inches:
> pi <- par("pin")
> mi <- par("mai")
> ## convert the coordinates into inches:
> xi1 <- (x - pu[1]) / (pu[2] - pu[1]) * pi[1] + mi[2]
> yi1 <- (y - pu[3]) / (pu[4] - pu[3]) * pi[2] + mi[1]
> ## xi1 and yi1 are the final coordinates of this tip in inches
>
> ## plot the 2nd tree:
> plot(rtree(4))
> ## same as above:
> pp <- get("last_plot.phylo", envir = .PlotPhyloEnv)
> ## ... except we take the coordinates of the root:
> x <- pp$xx[5]
> y <- pp$yy[5]
> pu <- par("usr")
> pi <- par("pin")
> mi <- par("mai")
> xi2 <- (x - pu[1]) / (pu[2] - pu[1]) * pi[1] + mi[2]
> yi2 <- (y - pu[3]) / (pu[4] - pu[3]) * pi[2] + mi[1]
> ## xi2 and yi2 are the final coordinates of this root in inches
>
> ## we add the height of this second plot to the 'y' coordinate
> ## of the first tip of the first tree which is above the second
> ## one according to layout()
```

```
> yi1 <- yi1 + par("fin")[2]
> ## => this operation depends on the specific layout of plots
>
> ## get the dimension of the device in inches:
> di <- par("din")
> ## reset the layout
> layout(1)
> ## set new = TRUE and the margins to zero:
> par(new = TRUE, mai = rep(0, 4))
> ## set the scales to be [0,1] on both axes (in user coordinates):
> plot(NA, type = "n", ann = FALSE, axes = FALSE, xlim = 0:1,
+      ylim = 0:1, xaxs = "i", yaxs = "i")
> ## graphical elements can now be added:
> fancyarrows(xi1/di[1], yi1/di[2], xi2/di[1], yi2/di[2], 1,
+             lwd = 10, col = rgb(1, .5, 0, .5), type = "h")
```



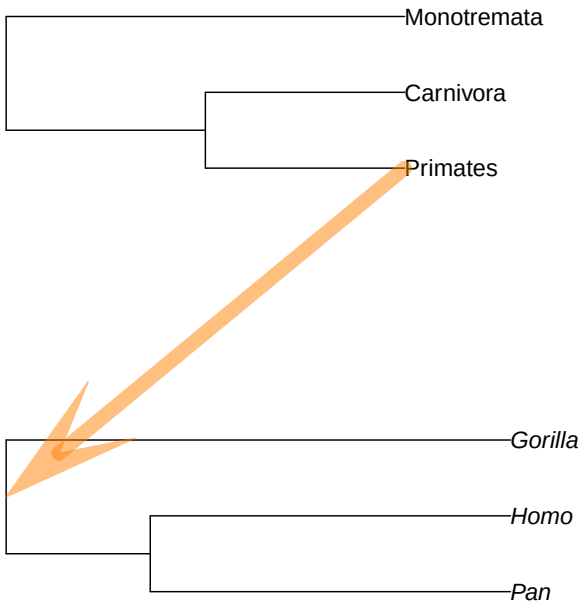
This code must be adapted to the exact layout in order to shift appropriately the coordinates in the x- and/or y-direction(s). The next bloc defines a function which uses the above code to plot two trees and add an arrow from the tip specified with the argument `from` of the first tree to the root of the second tree:

```
> foo <- function(phy1, phy2, from)
+ {
+     layout(matrix(1:2, 2))
+     plot(phy1, font = 1)
+     pp <- get("last_plot.phylo", envir = .PlotPhyloEnv)
```

```
+     from <- which(phy1$tip.label == from)
+     x <- pp$xx[from]
+     y <- pp$yy[from]
+     ## fraction of the plot region:
+     pu <- par("usr")
+     ## convert into inches:
+     pi <- par("pin")
+     mi <- par("mai")
+     xi1 <- (x - pu[1]) / (pu[2] - pu[1]) * pi[1] + mi[2]
+     yi1 <- (y - pu[3]) / (pu[4] - pu[3]) * pi[2] + mi[1]
+     plot(phy2)
+     pp <- get("last_plot.phylo", envir = .PlotPhyloEnv)
+     to <- Ntip(phy2) + 1
+     x <- pp$xx[to]
+     y <- pp$yy[to]
+     ## same as above:
+     pu <- par("usr")
+     pi <- par("pin")
+     mi <- par("mai")
+     xi2 <- (x - pu[1]) / (pu[2] - pu[1]) * pi[1] + mi[2]
+     yi2 <- (y - pu[3]) / (pu[4] - pu[3]) * pi[2] + mi[1]
+     yi1 <- yi1 + par("fin")[2]
+     di <- par("din")
+     layout(1)
+     par(new = TRUE, mai = rep(0, 4))
+     plot(NA, type = "n", ann = FALSE, axes = FALSE, xlim = 0:1,
+          ylim = 0:1, xaxs = "i", yaxs = "i")
+     fancyarrows(xi1/di[1], yi1/di[2], xi2/di[1], yi2/di[2], 1,
+                 lwd = 10, col = rgb(1, .5, 0, .5), type = "h")
+ }
```

We try this function with a tree of mammalian orders that we want to connect with our small tree `mytr`:

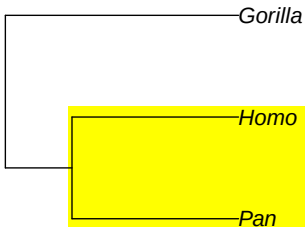
```
> trb <- read.tree(text = "((Primates,Carnivora),Monotremata);")
> par(xpd = TRUE)
> foo(trb, mytr, "Primates")
```



Note that the first tree has no branch length, so both trees have obviously different scales.

Another use for these functionalities is to draw coloured rectangles to delimit clades. Again we can use `par(new = TRUE)` this time with the option `plot = FALSE` of `plot.phylo` which opens and set the graphical device with the coordinates taken from the tree as detailed in Section 2.2, but nothing is drawn. The user can then call any low-level plotting command, then use `plot()` after `par(new = TRUE)`. For instance, if we want to draw a rectangle to show the clade made by humans and chimpanzees:

```
> plot(mytr, plot = FALSE)
> pp <- get("last_plot.phylo", envir = .PlotPhyloEnv)
> rect(pp$xx[5] - 0.1, pp$yy[1] - 0.1, pp$xx[1] + 2, pp$yy[2] + 0.1,
+      col = "yellow", border = NA)
> par(new = TRUE)
> plot(mytr)
```



There are several ways to find the limits of the rectangle: trying different values empirically, using `locator()` on a first draft plot with the tree, or with calculations similar to the previous example. The examples in `?phydataplot` gives other examples of using `par(new = TRUE)`.

References

[1] L. Czech, J. Huerta-Cepas, and A. Stamatakis. A critical review on the use of support values in tree viewers and bioinformatics toolkits. *Molecular Biology and Evolution*, 34:1535–1542, 2017.

[2] J. Felsenstein. *Inferring Phylogenies*. Sinauer Associates, Sunderland, MA, 2004.

[3] P. Murrell. *R Graphics*. Chapman & Hall/CRC, Boca Raton, FL, 2006.