treats workshop

Thomas Guillerme

2024-04-25

Pre-intro

Most of the information about the workshop comes from the `treats` manual. You can also always find more documentation about the functions used here using the `R` inbuilt manual by typing `?function.name`.

`R` level

In this workshop I will assume you are already familiar with basic `R`. The basic notions that I'll assume you know are:

- What is a package (e.g. `ape` or `dispRity`)
- What is an object (e.g. `this_object <- 1`)
- What is an object's class (e.g. the class `"matrix"` or `"phylo"`)
- What is a function (e.g. the function `mean(c(1,2))`)
- How to access function manuals (e.g. `?mean`)

Let's get into it.

The workshop markdown file: https://github.com/TGuillerme/treats/blob/master/inst/vignettes/treats_workshop.Rmd

The same file for direct download (right click + save as): https://raw.githubusercontent.com/TGuillerme/treats/master/inst/vignettes/treats_workshop.Rmd

The same file but in html (for visualising the fancy plots): https://cdn.githubraw.com/TGuillerme/treats/master/inst/vignettes/treats_workshop.html

First we'll want to download and install the package:

```r
install.packages("treats")
library(treats)
```

Acknowledgments

Note that I am the only author on the paper associated with this software suggesting I work alone and deserve all credits. This is of course not the case and a lot of input on this project was provided by many patient people. Many thanks to Mark Puttick that inspired the idea of the package; Andrew Beckerman, Natalie Cooper and Gavin Thomas for support during the development stage; Ian Brennan, Gustavo Burin, Christopher Cooney, Alex Cranston, William Gearty, Jasmine Hardie, Tom Lansley, Clement Prieul, Tiago Quental, Joe Rees, Louie Rombaut, James Rule and Sophie Ryan for comments and suggestions on late development stages; and Alain Danet, Tom Lansley, Rob McDonald and Ankita Sinha for testing this workshop

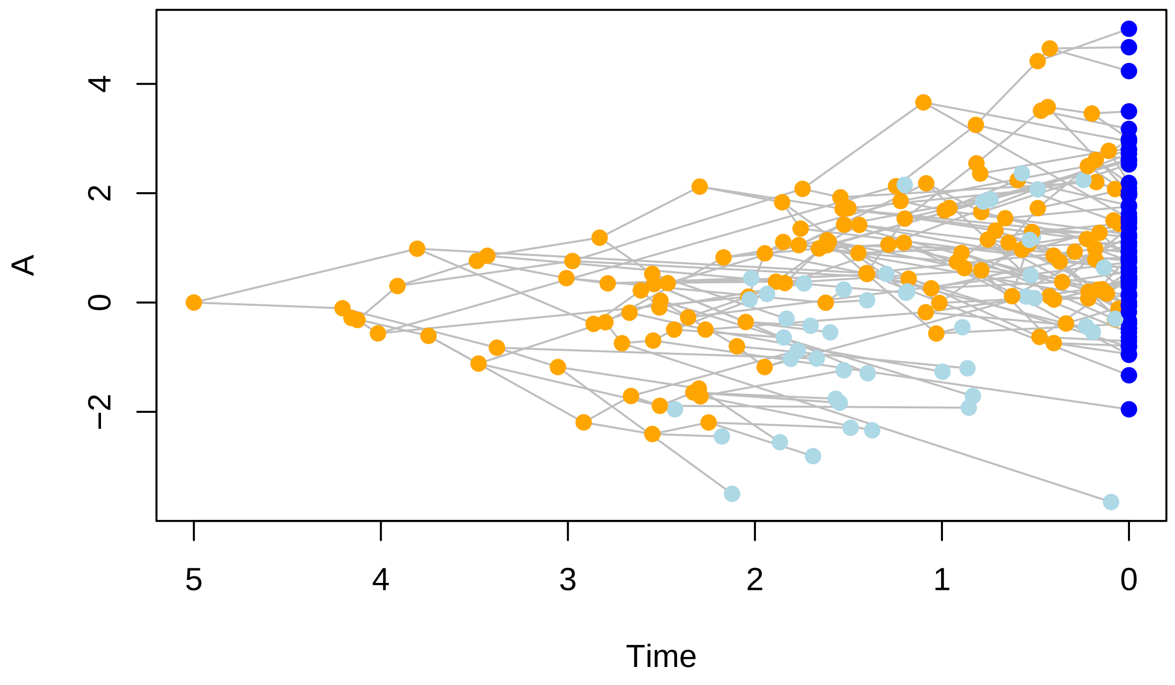
Intro

What ca we do with `treats`?

- Simulate trees and traits for illustration (toy model, teaching);
- Simulate trees and traits as a null model for analysis (we'll see an example of that later);
- Whatever you want: this is because of the modular architecture of the package allowing you to modify each steps of the simulations easily (we'll see this throughout the workshop);

Here are a couple of examples of things that we can do with `treats`:

Simulating a tree with increased extinction rate for species with negative trait values after some



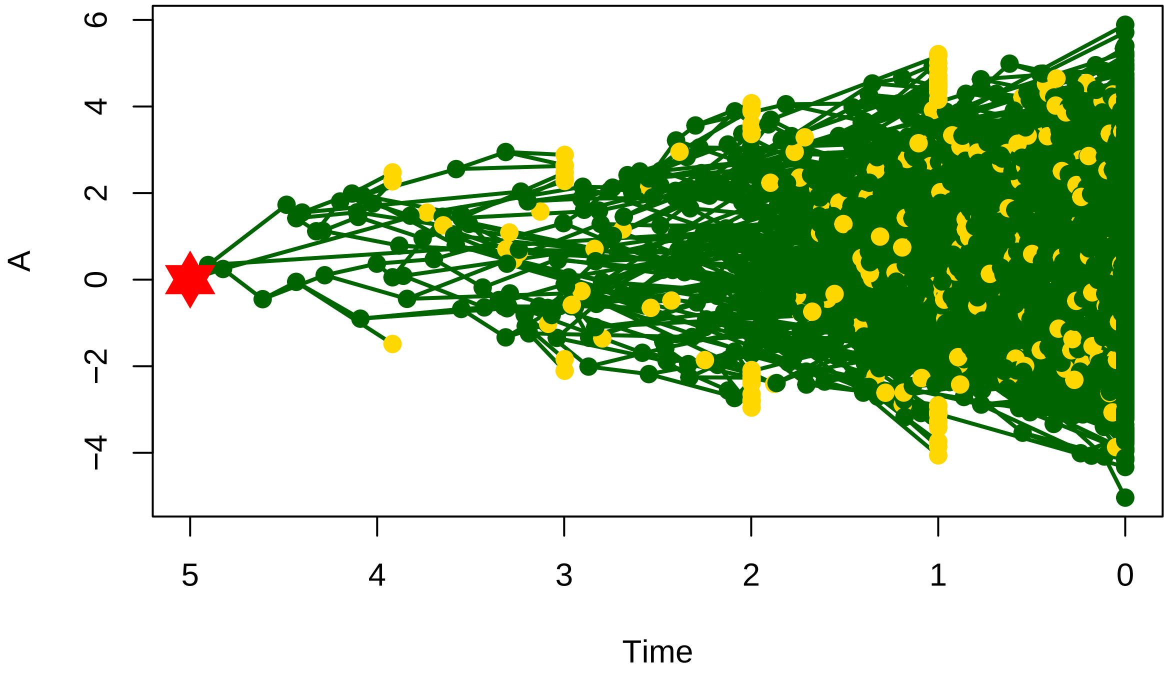
time:

Simulating a tree with a 2D trait with changing correlation:

```
## Warning in snapshot3d(scene = x, width = width, height = height): webshot =
## TRUE requires the webshot2 package and Chrome browser; using rgl.snapshot()
## instead

## Warning in rgl.snapshot(filename, fmt, top): this build of rgl does not support
## snapshots
```

[redacted]

Simulating a tree with sequential extinction events affect species on the edges of the trait distri-



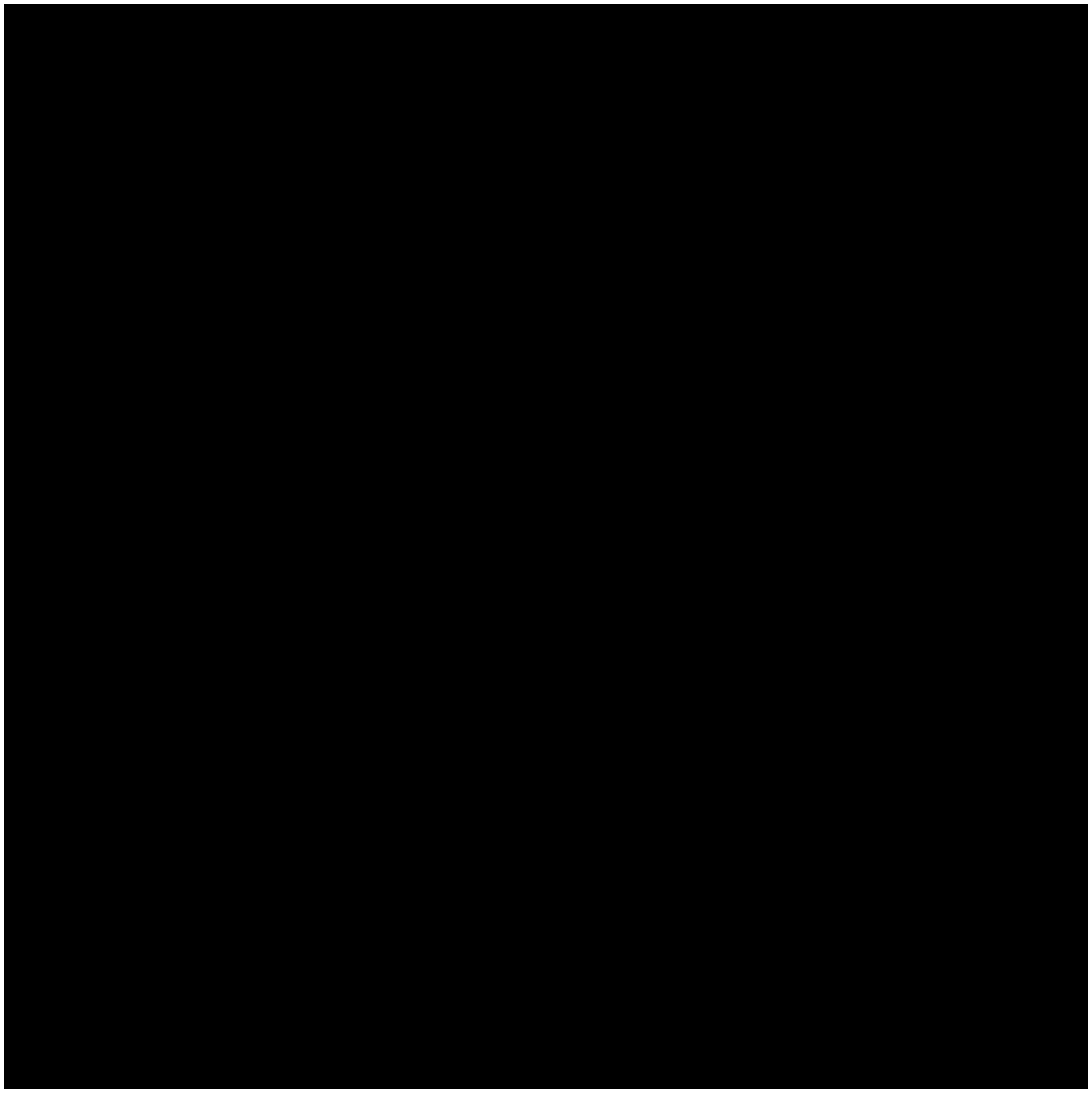
bution each time:

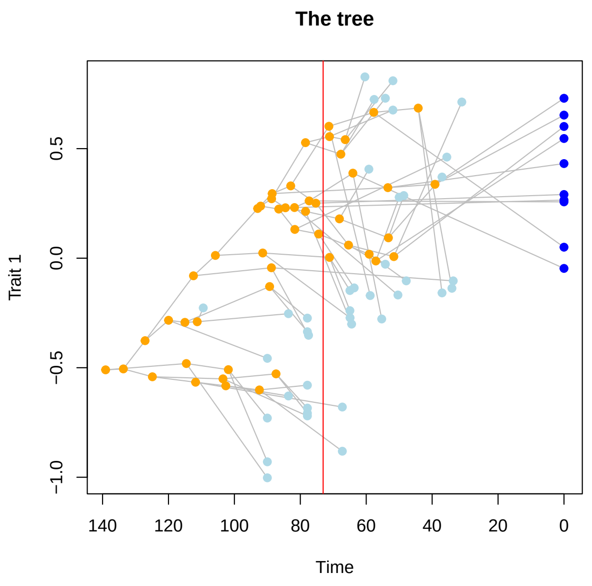
An example of using some null modeling

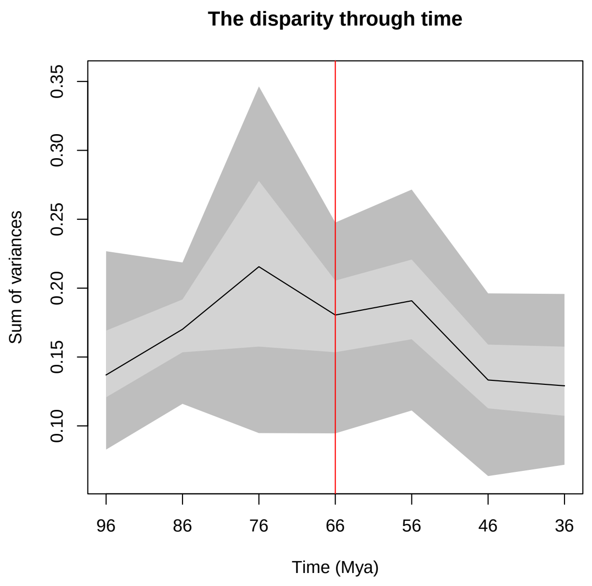
Say we're looking at an empirical dataset showing trait evolution through time. Here we have (part of) a mammalian dataset from Beck and Lee 2014 showing mammalian traits evolution around the K-Pg extinction 66 million years ago (the one that wiped out dinosaurs). We could be interested in the question of whether the mass extinction event freed ecological niches and was responsible for mammalian diversification in terms of traits. To do this we can simply look at the dataset an measure changes in trait diversity through time (disparity). For ecologists: this is very similar to functional diversity or dissimilarity analyses (I would even argue it's identical).

```
## Warning in snapshot3d(scene = x, width = width, height = height): webshot =
## TRUE requires the webshot2 package and Chrome browser; using rgl.snapshot()
## instead
```

```
## Warning in rgl.snapshot(filename, fmt, top): this build of rgl does not support
## snapshots
```







Here we can see that, although disparity (trait diversity) changes through time, it doesn't seem that affected by the K-Pg extinction event. This can be due to many causes, namely because of the quality of this dataset (that was not designed for this). But a fair question could be whether mass extinction events in general *CAN* actually result in changes of disparity through time, regardless of the clade and the data! In other words, could our tools pick up something here?

Simulating data to test our hypothesis

To test our hypothesis **whether mass extinction could liberate niches for trait diversification to occur**, we could simulate some neutral scenarios that would show us what disparity through time (trait diversification) would look like under two neutral scenarios:

- What happens to disparity when a random mass extinction occurs?
- What happens to disparity when a selective mass extinction occurs?

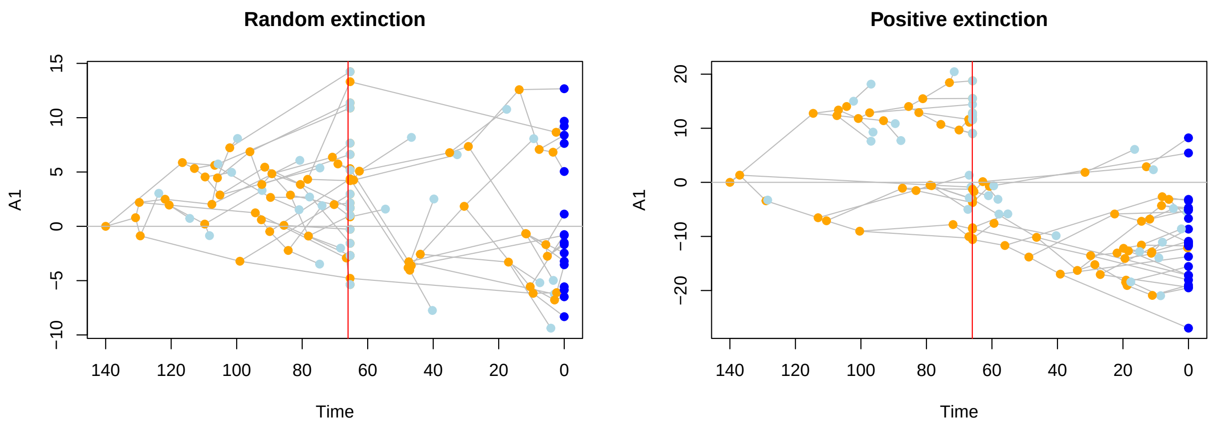
To do so, we would need to simulate a decent amount of datasets under these two specific scenarios and measure disparity from them and then compare it to our observed disparity.

This can be done relatively easily in `treats` by design very specific extinction scenarios. For example here we can simulate some similar datasets as the empirical ones with the two extinction scenarios:

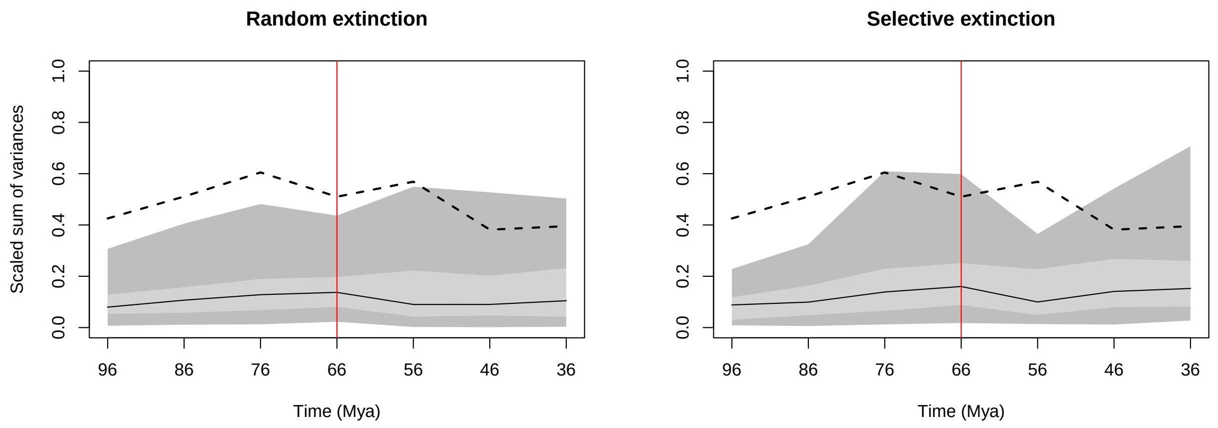
```
## Warning in snapshot3d(scene = x, width = width, height = height): webshot =
## TRUE requires the webshot2 package and Chrome browser; using rgl.snapshot()
## instead
```

```
## Warning in rgl.snapshot(filename, fmt, top): this build of rgl does not support
## snapshots
```

```
## Warning in snapshot3d(scene = x, width = width, height = height): webshot =
## TRUE requires the webshot2 package and Chrome browser; using rgl.snapshot()
## instead

## Warning in rgl.snapshot(filename, fmt, top): this build of rgl does not support
## snapshots
```



We can then compare the measured disparity in these scenarios to the observed one in the empirical data:



So here this example is just to highlight the need for a modular package to simulate trees and traits using very specific simulation scenarios.

> Note that here I didn't explain what's going on on the coding level. But you can check out a more detailed version of this case example at the end of this workshop.

The basics: copy-pasting starts here!

```r
library(treats)
```

Growing a tree

For the very basics here we're going to see how to grow a tree with the minimum number of options. `treats` is designed to grow trees easily but not with to many default options this is so to encourage users to know what they're doing *at least a minimum.*

`stop.rule`

So what do we need? To grow a tree you need to fix a rule of when the simulation should stop: This is a hard coded rule for stopping the growth of a tree, this can be the number of species (living or fossil) or a certain time of the simulation. Here when I mean time, I mean how long the simulation runs for in arbitrary time units (e.g. million of years, etc...).

This is passed to the `stop.rule` argument that is a named list containing integers or numeric values. For example:

```r
## Stopping when reaching 200 species
stop_200_sp <- list(max.taxa = 200)
## Stopping when reaching 4 time units
stop_4_time <- list(max.time = 4)
```

You can then feed these stop rules to the `treats` function and let the magic happen:

```r
## Create a tree with 200 taxa
tree_200_sp <- treats(stop.rule = stop_200_sp)
## Creating a tree with 4 time units
tree_4_time <- treats(stop.rule = stop_4_time)
```

This results in two different trees (`"phylo"` objects) that you can easily visualise with the excellent `"ape"` package:

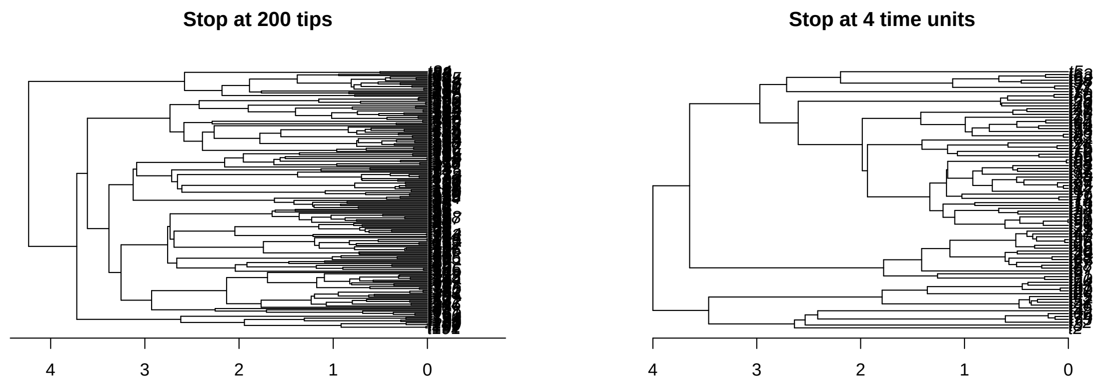
```
## What's in these objects?
tree_200_sp
```

```
##
## Phylogenetic tree with 200 tips and 199 internal nodes.
##
## Tip labels:
##   t1, t2, t3, t4, t5, t6, ...
## Node labels:
##   n1, n2, n3, n4, n5, n6, ...
##
## Rooted; includes branch lengths.
```

```
tree_4_time
```

```
##
## Phylogenetic tree with 91 tips and 90 internal nodes.
##
## Tip labels:
##   t1, t2, t3, t4, t5, t6, ...
## Node labels:
##   n1, n2, n3, n4, n5, n6, ...
##
## Rooted; includes branch lengths.
```

```
## Displaying the trees
op <- par(mfrow = c(1,2))
plot(tree_200_sp, main = "Stop at 200 tips") ; axisPhylo()
plot(tree_4_time, main = "Stop at 4 time units") ; axisPhylo()
```

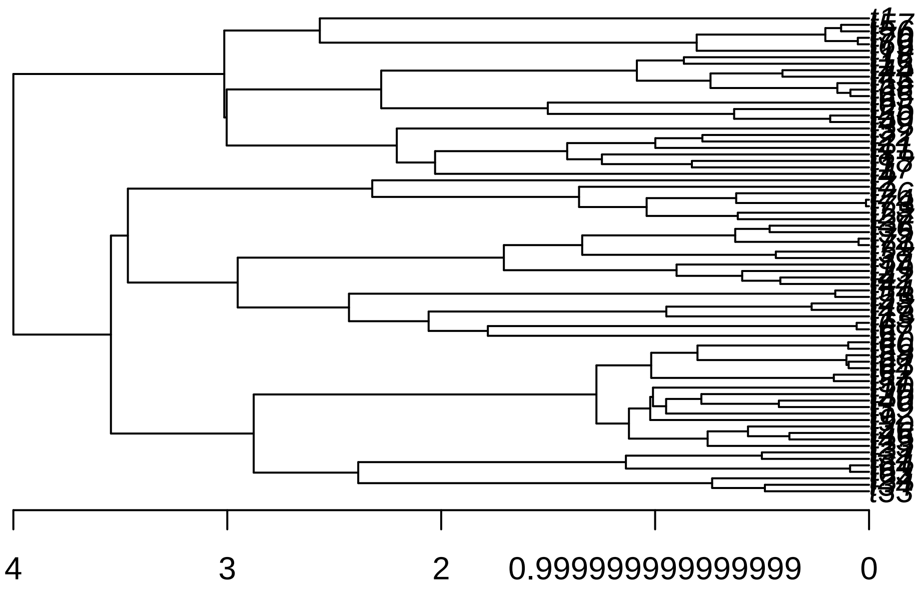


```
par(op)
```

So the trees here have roughly the same age but a very different shape, we'll get back to that in a moment. One other thing to note for this stopping rule, is that you can stack them: for example, you can ask the simulation to stop when it reaches 200 species *or* 4 time units.

```
## Multiple rules
my_stop_rules <- list(max.taxa = 200, max.time = 4)
## Simulating one tree with these rules
my_tree <- treats(stop.rule = my_stop_rules)
## Displaying the resulting tree
```

```r
plot(my_tree) ; axisPhylo()
```

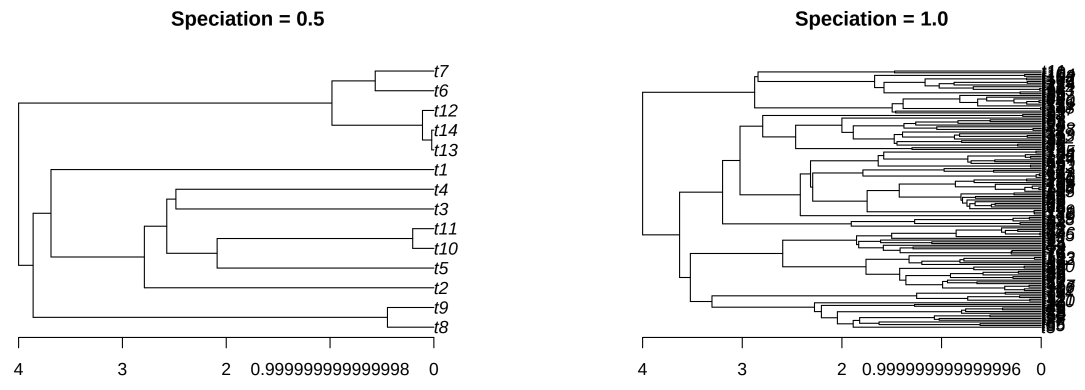


bd.params

As mentioned above, we can have vastly different trees depending on the simulation random seed. You can of course always modify that by modifying the seed. But a more interesting method is to try to control the actual speciation parameter. This is also called the birth or λ parameter. You can find details on how the speciation and the extinction parameters are used in the birth-death equations in the `treats` manual (or anywhere else where there's info about birth-death algorithms) so I'm not gonna go in the details here. Basically, the bigger the speciation parameter, the shorter the tree branches (the bushier the tree).

You can set this parameter using the `make.bd.params` function:

```r
set.seed(1)
## Creating a small birth parameter
speciation_0.5 <- make.bd.params(speciation = 0.5)
## Creating a big birth parameter (1 is the default by the way)
speciation_1.0 <- make.bd.params(speciation = 1.0)

## Simulating the trees
my_tree_small <- treats(stop.rule = my_stop_rules,
                        bd.params = speciation_0.5)
my_tree_big   <- treats(stop.rule = my_stop_rules,
                        bd.params = speciation_1.0)
## Displaying the trees
op <- par(mfrow = c(1,2))
plot(my_tree_small, main = "Speciation = 0.5") ; axisPhylo()
plot(my_tree_big, main = "Speciation = 1.0") ; axisPhylo()
```



```r
par(op)
```

By the way, throughout the pipeline in the package you can check what's in the object you create. For example, `speciation_0.5` is a `"treats"` and `"bd.params"` object that contains info about the parameters:

```r
## What is this?
class(speciation_0.5)
```

```
## [1] "treats"    "bd.params"
```

```r
## What's in it?
speciation_0.5
```

```
##  ---- treats birth-death parameters object ----
## speciation: 0.5.
## extinction: 0.
```

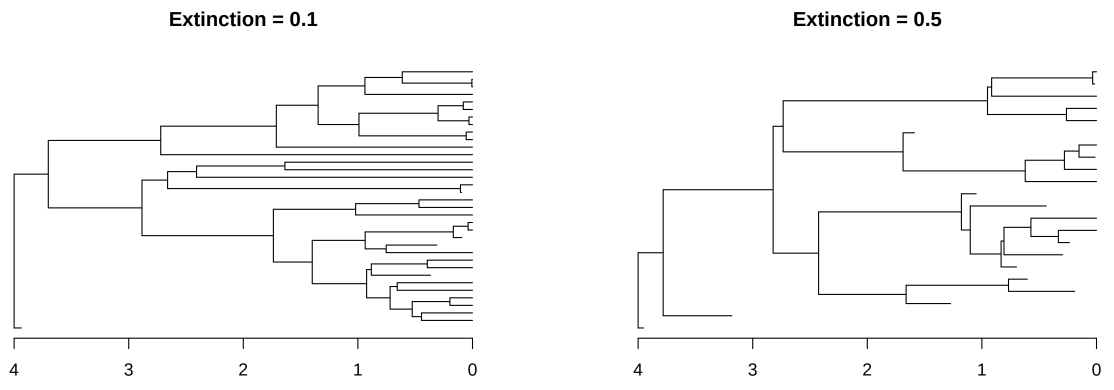
This allows you to always double check what's in the objects you create!

The death parameter!

Note that this object contained also an extinction parameter (set to 0 by default). This allows to introduce extinction events in your trees! The bigger this parameter, the more fossils in your tree.

```r
## Creating a small death parameter
extinction_0.1 <- make.bd.params(extinction = 0.1,
                                 speciation = 1)
## Creating a big death parameter
## (note speciation = 1 is the default)
extinction_0.5 <- make.bd.params(extinction = 0.5)

## Simulating the trees
set.seed(1) ## Putting the seed here ensures both trees grow the same way
few_fossils  <- treats(stop.rule = my_stop_rules,
                       bd.params = extinction_0.1)
set.seed(1)
many_fossils <- treats(stop.rule = my_stop_rules,
                       bd.params = extinction_0.5)

## Displaying the trees
```

```r
op <- par(mfrow = c(1,2))
plot(few_fossils,  main = "Extinction = 0.1", show.tip.label = FALSE) ; axisPhylo()
plot(many_fossils, main = "Extinction = 0.5", show.tip.label = FALSE) ; axisPhylo()
```



```r
par(op)
```

OK but what's actually happening in the algorithm here? Again there's many ways to illustrate or define the birth-death process (e.g. wikipedia) but we can have a look at a simple illustration of each step here.

> Note that speciation and extinction are fixed values here. But they can also be distributions! For example `make.bd.params(speciation = runif)` creates a `bd.params` object with a speciation always randomly drawn between 0 and 1. You can also link this parameter to the extinction parameter (for example for keep speciation always greater than extinction). Have a look at `?make.bd.params` for more details

More simulation options for `treats`

Just before we get in the more interesting bits, I want to highlight some useful options to deal with stochasticity in `treats`. In fact, sometimes (often?) your trees can "die" before reaching the stop value, especially if the speciation and extinction parameters are close to each other numerically

```r
## Creating some dangerous parameters
dangerous_params <- make.bd.params(speciation = 1,
                                   extinction = 0.9)
```

```r
## Simulating a tree that "dies" (returns an error)
set.seed(2)
dead_tree <- treats(stop.rule = my_stop_rules,
                    bd.params = dangerous_params)
```

This outputs an error message! Aaaah! But fear not, you can handle this error message differently depending on what you need:

- you can either set it to null rather than and error (giving you `NULL` rather than an error) using the `null.error = 100` option:

```r
## Simulating a tree that "dies" (returns NULL)
set.seed(2)
dead_tree <- treats(stop.rule  = my_stop_rules,
                    bd.params  = dangerous_params,
                    null.error = TRUE)
dead_tree # is NULL
```

```
## NULL
```

- or you can “brute force” it by keeping trying to generate a tree until it works (or you loose) patience using the `null.error = n` option (where *n* is the number of trials you’re happy to wait for:

```
## Simulating a tree that "dies" (try and try again!)
set.seed(2)
dead_tree <- treats(stop.rule  = my_stop_rules,
                    bd.params  = dangerous_params,
                    null.error = 100,  ## Try up to 100 times!
                    verbose    = TRUE) ## Visualise the trials
```

```
## Building the tree:....Done.
```

```
dead_tree # is a tree!
```

```
##
## Phylogenetic tree with 50 tips and 49 internal nodes.
##
## Tip labels:
##   t1, t2, t3, t4, t5, t6, ...
## Node labels:
##   n1, n2, n3, n4, n5, n6, ...
##
## Rooted; includes branch lengths.
```

Finally, you can also create tree distributions by using the option `replicates = n` that will generate *n* trees.

Adding a trait

Now this is all nice a well but not very different from other great `R` packages that allow you to grow a birth-death tree. If anything it’s maybe even just more cumbersome! But of course, in `treats` you can also do the traits simulation part! To do that we’re going to use the `make.traits` function that will create a `"treats"` and `"traits"` object.

First we need to choose a trait generation process. This process should be a `function` that intakes the arguments `x0` (the value of the trait in at it’s last node) and `edge.length` (the amount of evolutionary time since last node).

For example we can create a Brownian Motion (BM) that is process that draws a random value from a normal distribution centered on some value (`x0`) and with a standard deviation that’s relative to the evolutionary time spend (`edge.length`). By default it’s good practice to set the `edge.length` argument to a specific value (say 1).

```
## Creating a simple BM process
my.BM.process <- function(x0, edge.length = 1) {
    return(rnorm(n = 1, mean = x0, sd = sqrt(edge.length)))
}
```

You can then feed this process to `make.traits` and set some specific parameters (e.g. the starting value) or check if it works and visualise it:

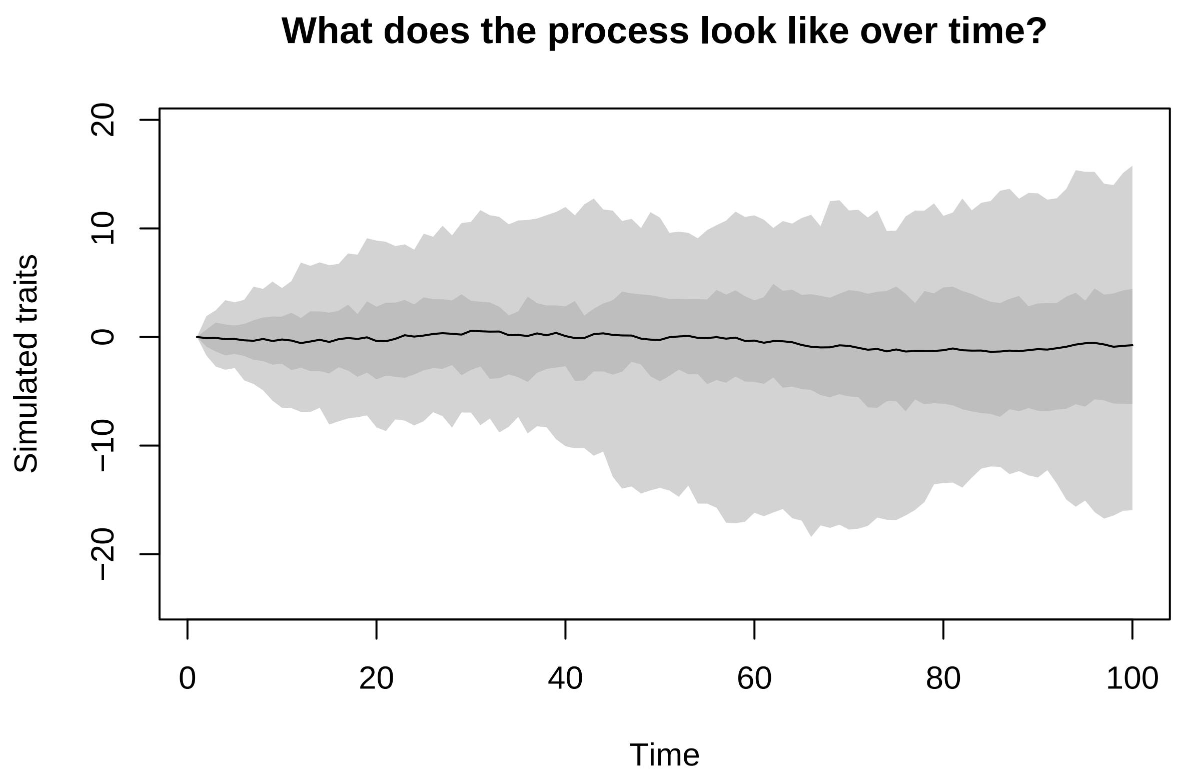
```
## Creating the trait object with the BM process
## and a starting value of 0
my_trait <- make.traits(process = my.BM.process,
                        start   = 0)
## What is the trait?
class(my_trait)
```

```
## [1] "treats" "traits"
```

```r
## What's in it?
my_trait ## Note the process name is A (you can change that)
```

```
##  ---- treats traits object ----
## 1 trait for 1 process (A) with one starting value (0).
```

```r
## Add a name to the trait object
my_trait <- make.traits(process = my.BM.process,
                        start   = 0,
                        trait.name = "BM")

## Visualising what the process would look over time
plot(my_trait, main = "What does the process look like over time?")
```



Note that you can of course use much more complex processes (e.g. multidimensional, with correlation, etc…). You can find some inbuilt processes by looking at the documentation for `?trait.process`.

Importantly, the function `make.traits` also checks if you have configured your process function correctly and will ping you an error message if not:

```r
## This doesn't work!
badly_formated_trait <- make.traits(process = rnorm)
```

Once you're happy with your process you can simply feed it to `treats` with the other arguments we've covered before:

```r
## Generating a tree and a trait
tree_and_trait <- treats(stop.rule  = my_stop_rules,
                         bd.params  = extinction_0.5,
                         traits     = my_trait,
                         null.error = 100,
                         verbose    = TRUE)
```

```
## Building the tree:..Done.
```

The resulting object is now a `"treats"` object (not a `"phylo"` as previously) that contains two main elements: the tree (a `"phylo"` object) and the data (a code `"matrix"`).

```r
## What is it?
class(tree_and_trait)
```

```
## [1] "treats"
```

```r
## What's in it?
tree_and_trait
```

```
##  ---- treats object ----
## Simulated phylogenetic tree (x$tree):
##
## Phylogenetic tree with 52 tips and 51 internal nodes.
##
## Tip labels:
##   t1, t2, t3, t4, t5, t6, ...
## Node labels:
##   n1, n2, n3, n4, n5, n6, ...
##
## Rooted; includes branch lengths.
##
## Simulated trait data (x$data):
## 1 trait for 1 process (BM) with one starting value (0).
```

```r
## The tree:
tree_and_trait$tree
```

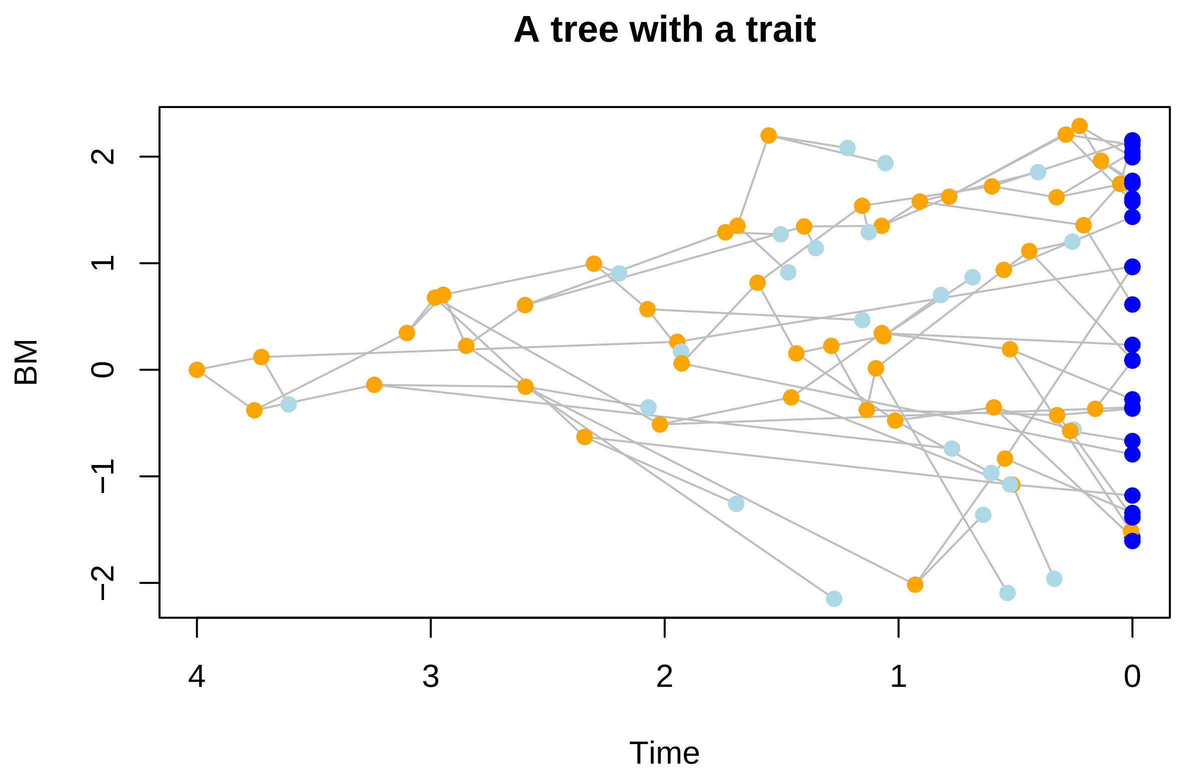
```
##
## Phylogenetic tree with 52 tips and 51 internal nodes.
##
## Tip labels:
##   t1, t2, t3, t4, t5, t6, ...
## Node labels:
##   n1, n2, n3, n4, n5, n6, ...
##
## Rooted; includes branch lengths.
```

```r
## The data
head(tree_and_trait$data)
```

```
##            BM
## n1  0.0000000
## n2  0.1197709
## n3 -0.3800877
## n4  0.3451465
## n5 -0.1409610
## t1 -0.3241549
```

You can again also visualise both together using our faithful `plot` function:

```r
## Wow!
plot(tree_and_trait, main = "A tree with a trait")
```



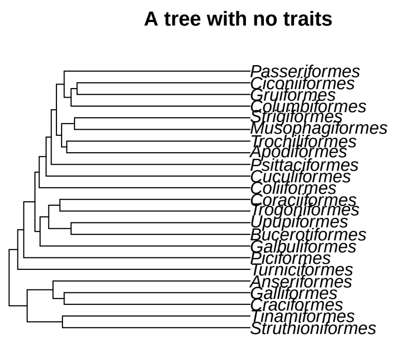
And that's it! We've simulated a tree and a trait at the same time. The whole point of this package is of course the modularity where you can play around and choose different tree parameters and different traits.

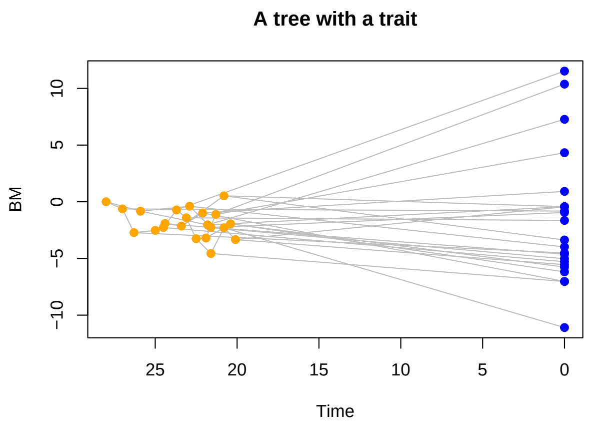
In the next part we'll even see how we can complexify all that by making all the birth-death process and the traits interact. Because yes, in this example so far the trait is just "mapped" on the tree: in other words, although both are simulated at the same time, the traits here are not influencing the tree generation.

Bonus: map.traits

This means that you can also just "map" any trait on a tree that you already have. For example, here we're going to map this BM trait on the bird orders example tree from `ape` using the `map.traits` function:

```r
## Loading the bird.orders tree
data(bird.orders)

## map the trait on the tree
## (i.e. simulate the trait using the tree topology)
my_bird_BM <- map.traits(traits = my_trait, tree = bird.orders)

## Show the results
op <- par(mfrow = c(1,2))
plot(bird.orders, main = "A tree with no traits")
plot(my_bird_BM, main = "A tree with a trait")
```





```
par(op)
```

The resulting object is a `"treats"` one so you can also just quickly generate just the data using `map.traits(traits = my_trait, tree = bird.orders)$data`.

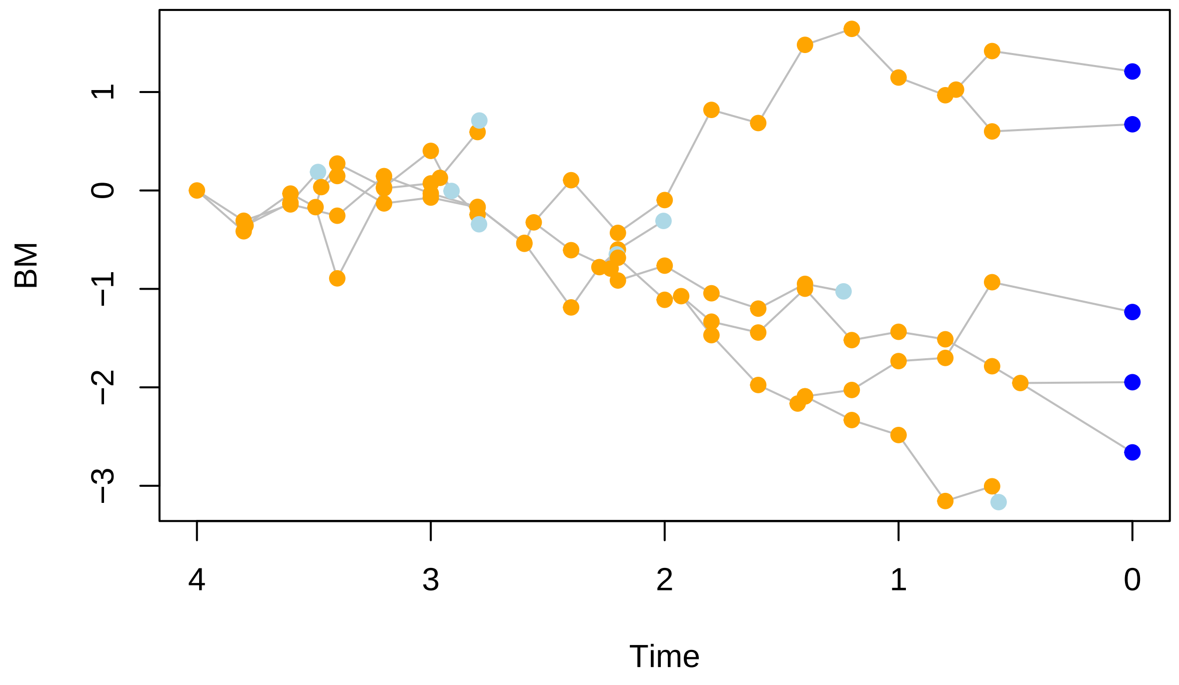
Bonus: save.steps

Note that here we've been generating trait values only at nodes and tips. In some cases, you might be interested to see you trait values generated constantly through time rather than just during speciation and extinction events. For example simulating trait values every x amount of time. You can do this using the `save.steps` in `treats` for example to generate the data every 0.2 time units:

```
## Generating a tree and a trait every 0.2 time steps
continuous_trait <- treats(stop.rule   = my_stop_rules,
                           bd.params   = extinction_0.5,
                           traits      = my_trait,
                           save.steps = 0.2,
                           null.error = 100,
                           verbose     = TRUE)
```

```
## Building the tree:.Done.
```

```
## Visualise the tree and trait
plot(continuous_trait)
```



Of course though, the bigger the tree and the traits, the slower the algorithm (and the more RAM eaten)!

Adding modifiers!

But let's get back to fun modular stuff: one of the advantages of the `treats` package is that you can modify *all* steps of the process. The birth-death process is effectively separated into three independent parts:

1. The selection of the lineage of interest.
2. The generation of branch length.
3. The decision whether that lineage goes extinct or speciates.

These three steps are encoded by default in the `treats` birth-death algorithm to be:

1. The selection of the lineage of interest: `sample(1, n)`
2. The generation of branch length: `rexp(1, rate = n*(speciation + extinction))`
3. The decision whether that lineage speciates (`TRUE`) or goes extinct (`FALSE`): `runif(1) < speciation/(speciation + extinction)`

You can see in more details what's happening by clicking on the modifier at the bottom of the pages here.

So what's the point of all that and of the modifiers?

Modifying the speciation process

This is the default speciation process that's using the `bd.params` object:

```
## The default speciation process (if TRUE, do speciate)
bd.speciating <- function(bd.params) {
    return(runif(1) <= (bd.params$speciation/(bd.params$speciation + bd.params$extinction)))
}
```

Here the speciation process is just depending on the birth-death parameters. But we can be interested in making a simulating a scenario approximating environements limiting capacities: for example, extinction never occurs until there is at least ten species, then it becomes more and more likely the more species in the environement. We can create a modifier that scales the speciation/turnover ratio by the number of species present in the simulation:

```r
## The default speciation process (if TRUE, do speciate)
density.dependent.sp <- function(bd.params, lineage) {
    ## Calculate the speciation/turnover ratio (as previously)
    ratio <- bd.params$speciation/(bd.params$speciation + bd.params$extinction)
    ## Scale in with the 10s of species
    return(runif(1) <= ratio*(10/lineage$n))
}
```

> Note here that `lineage` is a new argument introduced here. You can always check what arguments are allowed for internal functions and what internal parts they modify by checking the manual "Quick overview" sections. In the case the argument `lineage` points to an internal structure of the `treats` package (no need to worry what's in there for now - but if you're interested, it's all detailed here).

Similarly as for `make.traits` you can create a `"modifiers"` object using `make.modifiers` (again, testing for the validity of your inputs):

```r
## A modifier that makes extinction density dependent
dd.extinction <- make.modifiers(speciation = density.dependent.sp)
## What is it?
class(dd.extinction)
```

```
## [1] "treats"    "modifiers"
```

```r
## What's in it?
dd.extinction
```

```
##  ---- treats modifiers object ----
## Default branch length process.
## Default selection process.
## Speciation process is set to density.dependent.sp.
```

And then feed it to the `treats` function:

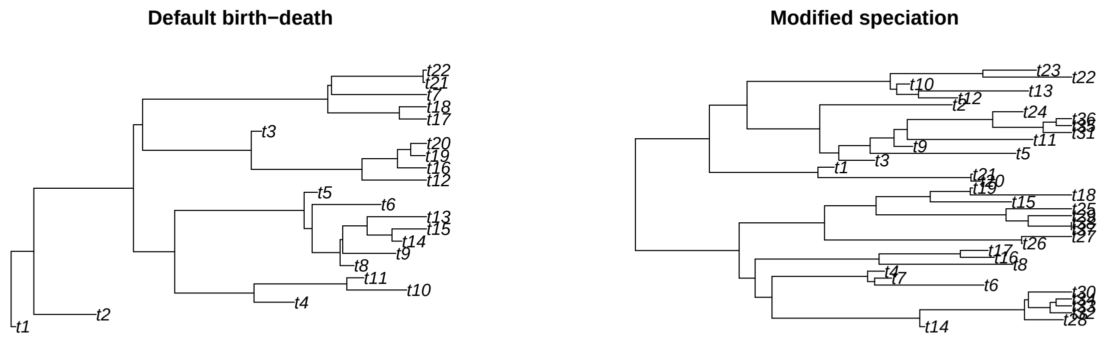
```r
set.seed(1)
## Generating a tree and a trait (no modifier)
no_modifier <- treats(stop.rule  = my_stop_rules,
                      bd.params  = extinction_0.5,
                      null.error = 100,
                      verbose    = TRUE)
```

```
## Building the tree:.Done.
```

```r
set.seed(1)
## Generating a tree and a trait (modified sepciation)
modified_speciation <- treats(
                      stop.rule  = my_stop_rules,
                      bd.params  = extinction_0.5,
                      modifiers  = dd.extinction,
                      null.error = 100,
                      verbose    = TRUE)
```

```
## Building the tree:..Done.
```

```r
## Plotting the results
op <- par(mfrow = c(1,2))
plot(no_modifier, main = "Default birth-death")
plot(modified_speciation, main = "Modified speciation")
```



```r
par(op)
```

Adding events!

OK, and finally, the last class of `"treats"` objects are `"events"` created via `make.events`. These work the same way as for `"traits"` and `"modifiers"` but affect the `treats` simulation at a higher level. For example you can use them to simulate mass extinctions after a certain amount of time. Or you can do much more complex things due to the package modular structure: for example, you can use an `"event"` that changes a `"modifier"` depending on some trait values… Let's look at a quick example here.

To create an extinction event, you need to set up three aspects:

- *What the event is going to affect?* This can be each individual object passed to the simulation (i.e. the `modifiers`, `traits` or `bd.params`) or internal objects like, in our case the we can set to target to `target = "taxa"` (that will affect the internal `"lineage"` object);
- *When should the event happen?* This should be a function that is triggered following any specific condition (e.g. some species has a specific trait value, some number of species are present, etc.). In our case we are going to use a pre-implemented function called `age.condition` that generates the function to trigger the event at a specific age:

```r
## Making a function that triggers and event after 2 time units
age.condition(x = 2, condition = `>=`)
```

```
## function (bd.params, lineage, trait.values, time)
## {
##     condition(time, x)
## }
## <bytecode: 0x6512497b6b90>
## <environment: 0x6512561b4570>
```

```r
## This is effectively testing whether time is greater or equal than x
x = 2
time <- 1
time >= x
```

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

```r
## Or, in a more classic function format
`>=`(time, x)
```

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

Note that for `age.condition`: the age is in units from the past, i.e. since the start of the simulation, not units from the present (i.e. units of time ago). You can always change that by defining it as `age.condition(x = time_in_the_present - time)`

- *What the event is going to modify?* This should be also a function that, once the event is triggered, modifies the target of the simulation (e.g. modify the `bd.params` or the `traits` object). In our case, we will use the pre-implemented function `trait.extinction`:

```r
## Making a function that removes all species with trait values >= 0
trait.extinction(x = 0, condition = `>=`)
```

```
## function (bd.params, lineage, trait.values)
## {
##     parent_traits <- parent.traits(trait.values, lineage, current = FALSE)
##     selected_nodes <- as.numeric(names(which(condition(parent_traits[,
##         trait], x))))
##     extinct <- which(lineage$parents %in% selected_nodes)
##     lineage$livings <- lineage$livings[!lineage$livings %in%
##         extinct]
##     lineage$n <- length(lineage$livings)
##     return(lineage)
## }
## <bytecode: 0x65124b55a318>
## <environment: 0x6512548ce3a8>
```

We can then create these events the same way as for `make.modifiers` or `make.traits`:

```r
## Creating an extinction that removes species with positive trait values
positive_extinction <- make.events(
    target = "taxa",
    condition = age.condition(3),
    modification = trait.extinction(x = 0, condition = `>=`))

## What's it?
class(positive_extinction)
```

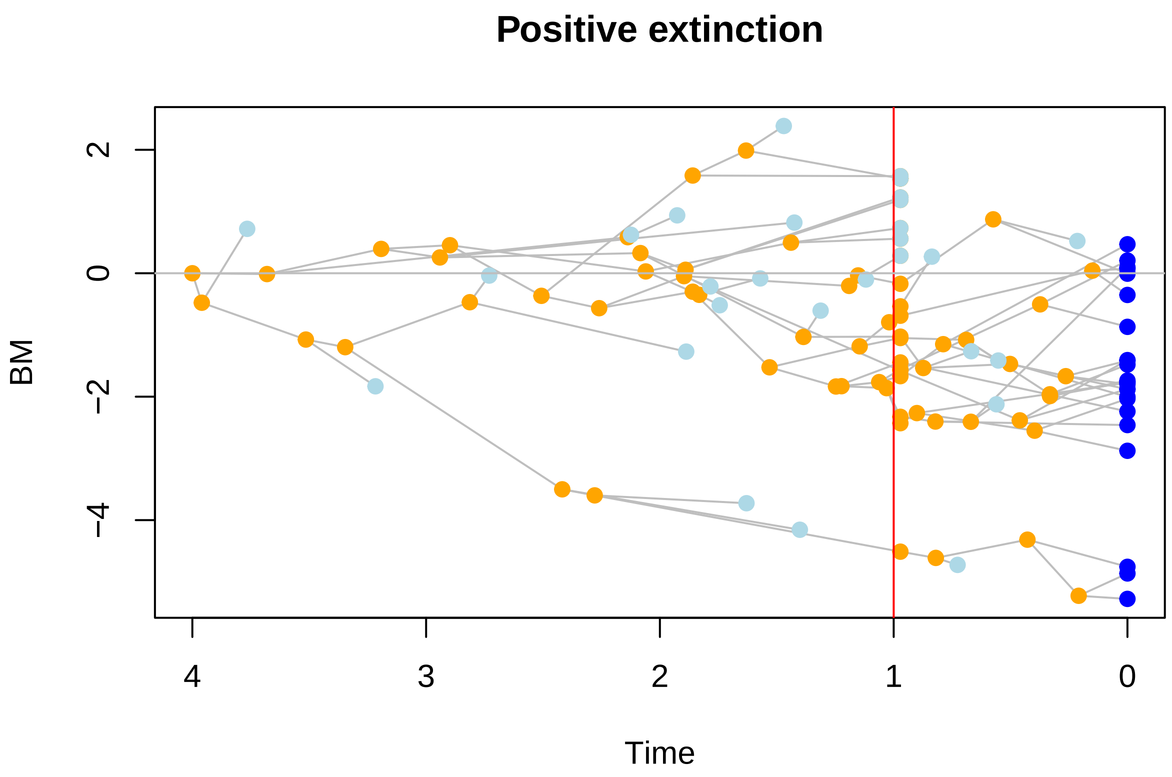
```
## [1] "treats" "events"
```

```r
## What's in it?
positive_extinction
```

```
##  ---- treats events object ----
## Event targeting "taxa" to be triggered 1 time.
## The condition function is: age.condition
## The modification function is: trait.extinction
```

And then simulate a tree and trait with this extinction event:

```r
set.seed(7)
## Simulate the tree and traits with a selective extinction event
sim_trait_extinction <- treats(
                   traits     = my_trait,
                   bd.params  = extinction_0.5,
                     stop.rule  = my_stop_rules,
                     events     = positive_extinction,
                     null.error = 100)

## Selective extinction
plot(sim_trait_extinction, main = "Positive extinction")
abline(v = 1, col = "red")
abline(h = 0, col = "grey")
```



Bonus: pre-made evolutionary scenarios

On the GitHub page of the package you can find a series of simulation templates: these are simplified complex simulation scenarios that you could find useful for your own specific questions. The idea here is to have a library of such templates so if you create one, please share it to the community by following this template.

Mega bonus: comparing simulated extinction scenarios to observed ones

Here is the detailed script for the simulating extinction scenarios from the start of the workshop.

Mammalian evolution after the K-Pg extinction

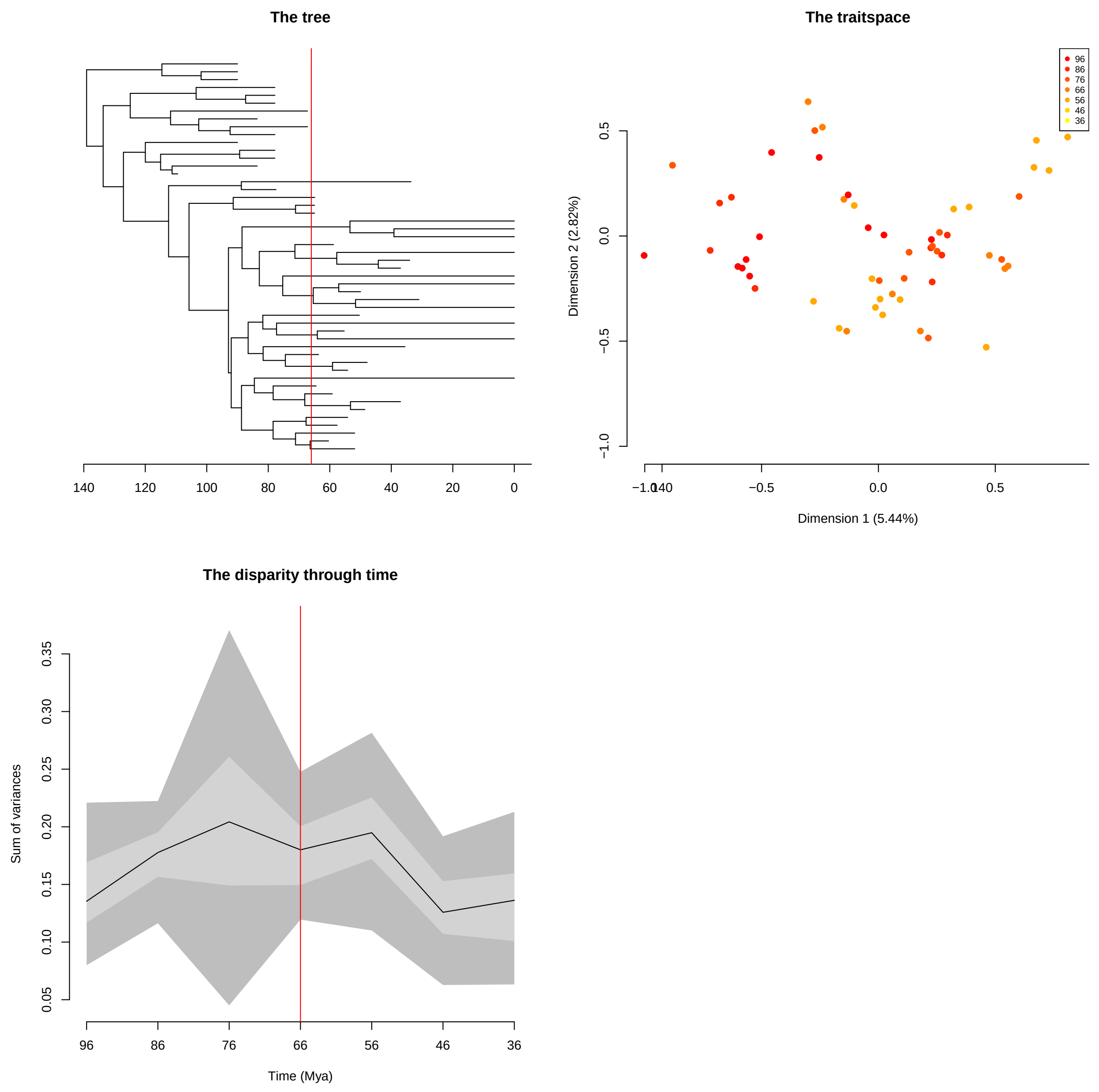
One toy example could be to test how a mass extinction event affect species traits evolution. Two potential scenarios could be tested depending on the type of extinction:

- The mass extinction is random
- The mass extinction targets specific trait values

We could simulate both scenarios and compare them to an observed one to see if the trait evolution matches with either of the simulated scenarios. If they do, we could suggest that one of the extinction scenario occurred.

Let's first look at the observed example data from the `dispRity` package (from Beck and Lee 2014). We will do a simple disparity through time analyses. Very briefly: we can build a **traitspace** (a multidimensional space that contains all the possible trait combinations for a group of species) and look how species occupied this space through time by looking at traitspace occupancy (**disparity**). This is a common type of analyses in palaebiology but I would argue that it's identical to functional diversity (dissimilarity) analyses in ecology.

```r
## Loading the package and data
library(dispRity)
data(BeckLee_tree)
data(BeckLee_mat99)

## Creating the time slices
time_slices <- chrono.subsets(BeckLee_mat99, BeckLee_tree,
                              method = "continuous",
                              model  = "proximity",
                              time   = seq(from = 96, to = 36, by = -10))

## Calculating disparity on the two first dimensions only
observed_disparity <- dispRity(boot.matrix(time_slices),
                               metric = c(sum, variances),
                               dimensions = c(1,2))

## Plotting the tree and the disparity through time
par(mfrow = c(2,2), bty = "null")
## Thre tree
plot(ladderize(BeckLee_tree), show.tip.label = FALSE,
     main = "The tree")
axisPhylo()
abline(v = BeckLee_tree$root.time - 66, col = "red")

## The Traitspace
plot(time_slices, main = "The traitspace")
axisPhylo()
abline(v = BeckLee_tree$root.time - 66, col = "red")

## The disparity
plot(observed_disparity, ylab = "Sum of variances",
     main = "The disparity through time")
abline(v = 4, col = "red")
```



Simulating similar data

To simulate the data and the extinction we will need the following:

1. A stopping rule
2. A set of birth death parameters
3. Some traits
4. Some extinction events

1. The stopping rule

Stopping when reaching 140 time units:

```r
## Setting the stopping rule (stop after 140 time units)
stop_rule <- list(max.time = 140)
```

2. **The birth death parameters**

```
## Using the birth-death parameters from the observed tree
my_bd_params <- make.bd.params(speciation = 0.035, extinction = 0.02)
```

3. **The traits**

```
## A 2D Brownian motion
my_traits <- make.traits(process = BM.process, n = 2)
```

4. **The extinction events**

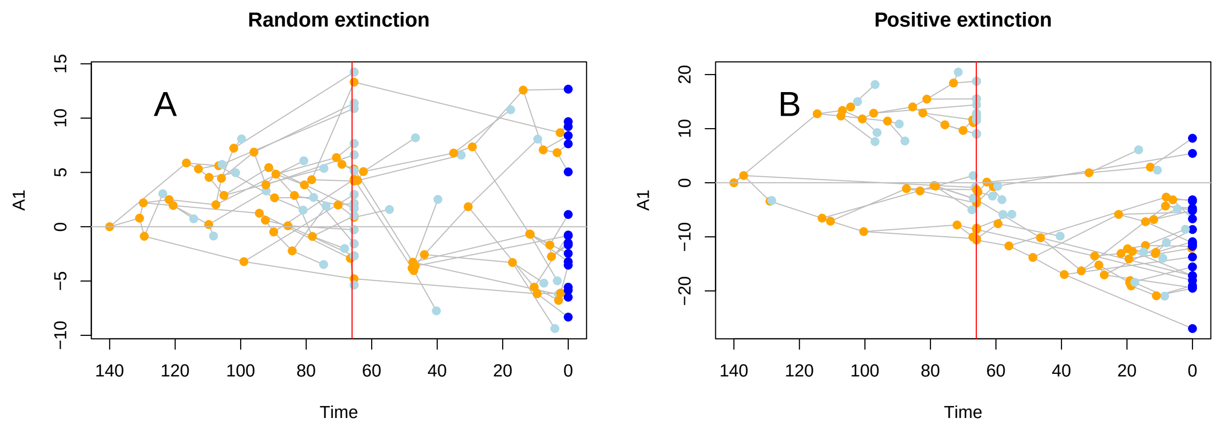
```
## Creating a random mass extinction
random_extinction <- make.events(
    target       = "taxa",
    condition    = age.condition(140-66),
    modification = random.extinction(0.75))

## Creating an extinction that removes species with positive trait values
positive_extinction <- make.events(
    target = "taxa",
    condition = age.condition(140-66),
    modification = trait.extinction(x = 0, condition = `>=`))
```

Simulating the data

We can then feed all that to our two simulations:

```
set.seed(123)
## Simulate the tree and traits with a random extinction event
sim_rand_extinction <- treats(
                   traits     = my_traits,
                   bd.params  = my_bd_params,
                   stop.rule  = stop_rule,
                   events     = random_extinction,
                   null.error = 100,
                   replicates = 50)

## Simulate the tree and traits with a selective extinction event
sim_trait_extinction <- treats(
                   traits     = my_traits,
                   bd.params  = my_bd_params,
                   stop.rule  = stop_rule,
                   events     = positive_extinction,
                   null.error = 100,
                   replicates = 50)

par(mfrow = c(1,2))
## Visualising the difference between both scenarios
## Random extinction
plot(sim_rand_extinction[[42]], main = "Random extinction")
abline(v = 66, col = "red")
abline(h = 0, col = "grey")
legend("topleft", pch = NULL, legend = "A", bty = "n", cex = 2)
```

```
## Selective extinction
plot(sim_trait_extinction[[50]], main = "Positive extinction")
abline(v = 66, col = "red")
abline(h = 0, col = "grey")
legend("topleft", pch = NULL, legend = "B", bty = "n", cex = 2)
```



Tada!

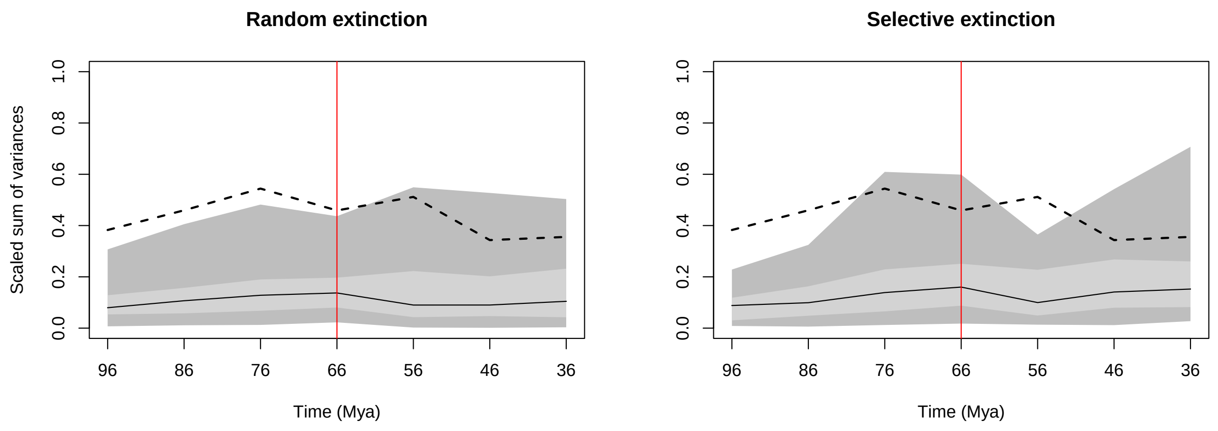
> Note that because of the stochastic nature of these simulations we will simulate a series of trees and traits, not just one. This can easily be done in `treats` using the `replicates` option. Here we will do 50 replicates and focus on just 2 traits.

Comparing both simualted and empirical scenarios

And from there, we can compare the simulated data to the empirical on to see if they match or not. In our specific question we can use the `dispRitreats` function that's a handy interface between the `dispRity` package and `treats` and use that to measure disparity in our simulated scenarios. Again, I'm not going into the details there because this kind of comparisons requires much more work and thoughts but it's here just used as a dummy example on how you can use `treats` for your empirical questions!

```
## Remove single nodes simulated at the extinction
sim_rand_extinction <- drop.singles(sim_rand_extinction)
sim_trait_extinction <- drop.singles(sim_trait_extinction)

## Calculate the dispRity for all the simulations
random_extinction_disparity <- dispRitreats(sim_rand_extinction,
                                            method = "continuous",
                                            model  = "proximity",
                                            time   = seq(from = 96, to = 36, by = -10),
                                            metric = c(sum, variances),
                                            scale.trees = FALSE)
selective_extinction_disparity <- dispRitreats(sim_trait_extinction,
                                            method = "continuous",
                                            model  = "proximity",
                                            time   = seq(from = 96, to = 36, by = -10),
                                            metric = c(sum, variances),
                                            scale.trees = FALSE)

## Scale the disparity results (to compare to the observed ones)
random_extinction_disparity <- scale.dispRity(random_extinction_disparity)
```

```r
selective_extinction_disparity <- scale.dispRity(selective_extinction_disparity)
observed_disparity <- unlist(get.disparity(scale.dispRity(observed_disparity)))

## Plotting the results with the observed disparity
par(mfrow = c(1,2))
plot(random_extinction_disparity, main = "Random extinction",
     ylab = "Scaled sum of variances", ylim = c(0, 1))
abline(v = 4, col = "red")
lines(x = 1:7, y = observed_disparity, lty = 2, lwd = 2)
plot(selective_extinction_disparity, main = "Selective extinction",
     ylab = "", ylim = c(0, 1))
abline(v = 4, col = "red")
lines(x = 1:7, y = observed_disparity, lty = 2, lwd = 2)
```


Random extinction
Selective extinction
Scaled sum of variances
Time (Mya)