Specific examples

Thomas Guillerme (guillert@tcd.ie)

2023-03-27

Contents

Chapter 1

`dads`: disparity and diversity simulations.

Allowing to simulate disparity and diversity at the same time with interaction between both.

The core of this package is based on the `diversitree` birth-death algorithm.

1.1 What is `dads`?

There are some very good packages out there to simulate birth death trees (like TreeSim) or even packages simulating traits (disparity) and diversity jointly (like RPANDA or PETER). We strongly advice you have a look at these packages first as they might be more appropriate for your tasks.

`dads` aims to be a highly modular and friendly version of all these packages: it allows to simulate disparity and diversity jointly with a vast array of options that can be easily modified by users. For example, you can easily generate any type of process to generate a trait (BM, OU, something else, etc…) in multiple dependent or independent dimensions through `"traits"` objects. You can then specify how the traits should affect disparity through `"modifiers"` objects. And finally you can create events (like mass extinctions) through `"events"` objects. These objects and how to modify them will be detailed throughout this manual. Finally we are putting an emphasise in the development of this package on the speed and reliability of the functions.

1.1.1 Modular?

Because their is an infinite way you might want to generate disparity and diversity (different traits, different modifiers and different events), the `dads` package

is designed to make all these parts easy to code separately and integrate them easily in the `dads` core functions. This allows you to simulate finely tuned multidimensional data for your specific project!

1.2 Who is this manual for?

This manual explains how `dads` work in as much details as possible and aims to give you the keys to design your own diversity and disparity simulations.

The first section `Getting started` is aimed for people with a `R` beginner level: * you know what a package is - and how to install it; * you know what a function is - and what arguments are; * you have already (vaguely) heard that they are different types of objects in R - like matrices (`"matrix"`) or trees (`"phylo"`); * and you have already looked for documentation and bugs online.

The sections after that are a bit more advanced and requires the following level: * you have already created functions before; * you know how to subset elements of a list - i.e. you understand the `list$element` syntax; * you are aware that both `TRUE == 1` and `FALSE == 0` are `TRUE`; * you have heard that the objects in R have classes and that it sometimes matters - for example you know that R can differentiate between `as.numeric(1)` and `as.integer(1)`; * you owe part of your life to Stack Overflow.

1.3 Installing and running the package

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

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

## Installing the latest version directly from GitHub
devtools::install_github("TGuillerme/dads")
```

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. Doing so will not only hopefully help you but also other users since it will help improve this manual!

1.5 How does dads work?

Basically, the dads function intakes your personalised traits, modifiers and events to generate your disparity and diversity. You will find more details about how these objects (traits, modifiers and events) work in the rest of the tutorial but here is a graphical representation of how dads work:

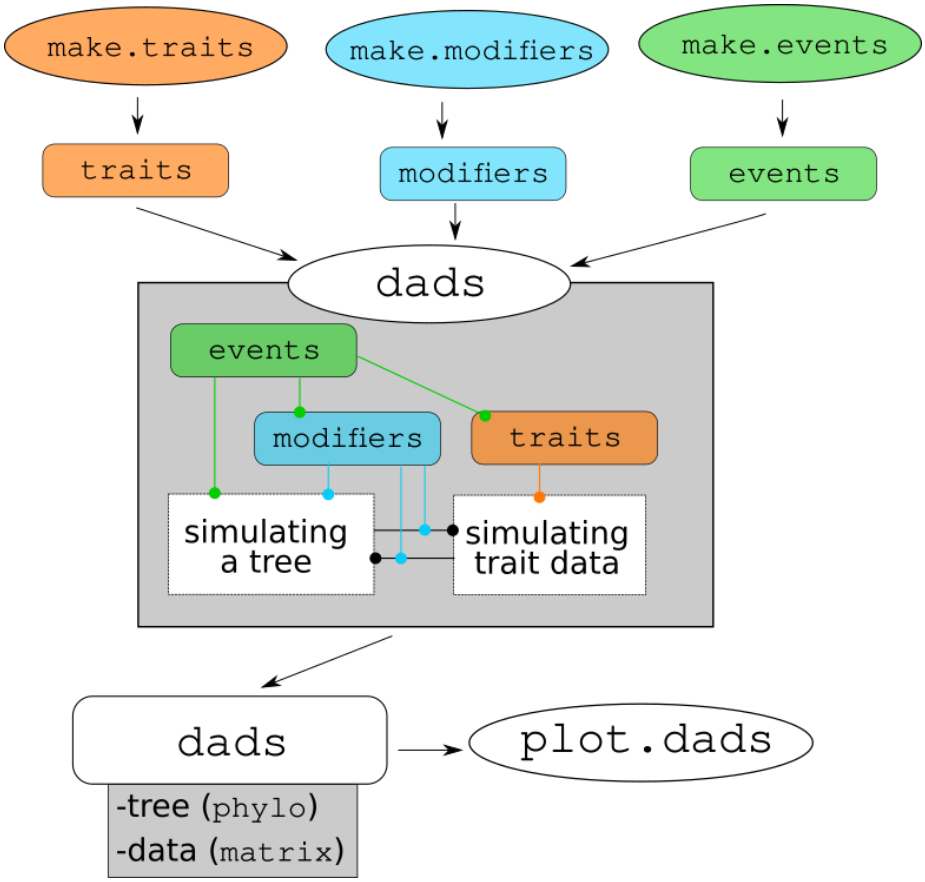


Figure 1.1: Schematised summary of the dads package architecture

1.6 The birth-death algorithm

> If you want to get your hands dirty, you can go straight to the getting start section, this following section just describes the process in pseudo-code.

The dads algorithm is based on a modular version of the birth-death process algorithm. The birth-death model is a continuous Markov process (i.e. a contin-

uous random process where the state will change each step are linked through time and change according to some random exponential variables) which is well defined mathematically and commonly used in biology but also in many other fields.

The algorithm used in `dads` allows modularity of this process and is based on the following steps (the text in `courier font` is for the name of the process in the algorithm):

0. **Starting the process**: this step is actually non modular and creates a random tree with one tip, one node and one branch length connecting both. This step is used to optimise the rest of the algorithm in terms of speed and memory management. The node, tip and branch resulting from this step are discarded at the end of the simulations.
1. **Selecting a lineage** (`selecting`): this step selects a tip that is currently not extinct. In a standard birth-death process this is just done randomly, however in `dads` this can be modified based on the birth-death parameters, the currently available lineages and potential trait values. *For example, it is possible to put a higher probability in selecting a lineage that is next to a lineage that recently went extinct and has a positive trait value.* Then go to step 2.
2. **Growing the tree** (`waiting`): this step grows the tree by a certain amount. It does so by adding the same amount of branch length to all the non-extinct lineages. In an exact birth death process, this is done by drawing a random value from an exponential distribution with a rate of

   $$\text{number of living lineages} \times (\text{speciation} + \text{extinction parameters})$$

   . In `dads`, this can be modified based on the birth-death parameters, the currently available lineages and potential trait values. *For example, it is possible to increase branch length by the ratio of living/extinct fossils and a random number drawn from the range of current trait values.* If the total tree length is less than the required tree length, go to step 3. Else go to step 6.
3. **Simulating traits** [optional](`traits`): this step allows to simulate a trait value for the selected lineage from step 1. This is typically not part of a standard birth-death process and is handled via the `traits` option in `dads` (see the `make.traits` chapter for more details). Then go to step 4.
4. **Speciating** (`speciating`): in this step, the selected lineage from step 1 has the option of speciating or going extinct. In a standard birth-death process, this happens by randomly drawing a value between 0 and 1 as a way to trigger speciation relatively to the birth-death parameters:

```
if
    randomly drawn number is smaller or equal to speciation/(speciation + extinction)
then
    do speciate
else
```

```
    go extinct
```

Again, in `dads`, this process can be modified based on the birth-death parameters, the currently available lineages and potential trait values. *For example, the lineage can only speciate if it's trait value is positive, regardless of whether speciation or extinction has been triggered.* Then go to step 5. 5. If the number of lineages is less than required number of lineages, then go to step 1. Else go to step 6. 6. The simulation has stopped because it reached the required amount of lineages or time.

```
## Loading required package: dispRity
```

Chapter 2

Getting started

2.1 The simplest of all analysis: simulating diversity only

One of the simplest things to do with the `dads` package is just to simulate a birth death tree. For that you can use the function `dads` and specify your stopping rule. The stopping rule simply tells the birth death process to step whenever it reaches one of these three conditions:

- `"max.taxa"` `= n` stop when `n` taxa are generated;
- `"max.living"` `= n` stop when there is `n` co-occuring taxa of the same age (i.e. "living" taxa);
- `"max.time"` `= n` stop when the simulated tree is `n` units of age old (these units are arbitrary);

For example, we might want to generate a birth-death tree with 20 taxa:

```
## Setting a stopping rule to reach a maximum of 20 taxa
my_stop_rule <- list(max.taxa = 20)
```

We can now run the simulations using:

```
## Running the birth death simulation
my_tree <- dads(stop.rule = my_stop_rule)
```

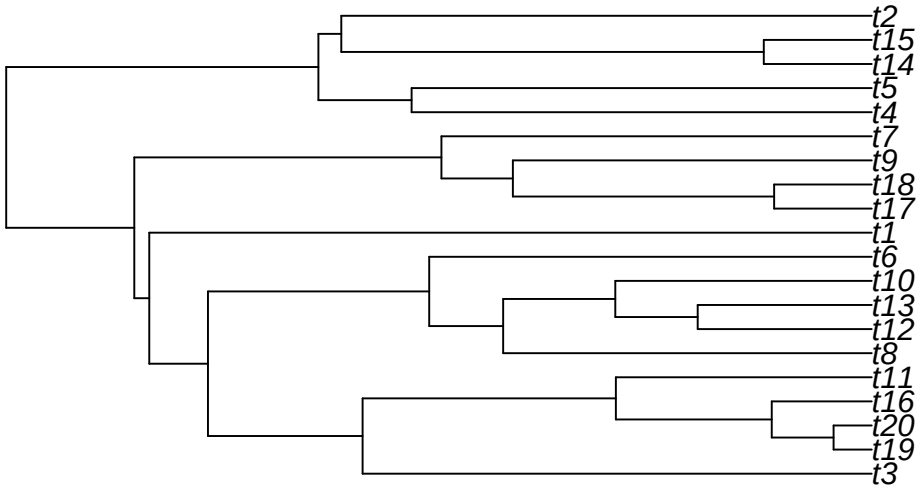
> Note that here we could have specified more than one stopping rule, for example, we might want to run a simulation and stop it if it either reaches 10 taxa or the age 2 using `stop.rule = list(max.time = 2, max.taxa = 10)`. The simulation will then stop when either of these conditions are met.

The resulting object is a classic `"phylo"` object that you can simply plot or visualise like so:

```
## The tree object
my_tree
```

```
##
## Phylogenetic tree with 20 tips and 19 internal nodes.
##
## Tip labels:
##   t1, t2, t3, t4, t5, t6, ...
## Node labels:
##   n1, n2, n3, n4, n5, n6, ...
##
## Rooted; includes branch lengths.
```

```
## Plotting it
plot(my_tree)
```



2.1.1 Changing the birth-death parameters

People familiar with the birth-death models might have noticed that we did not specify two important things here: the speciation parameter (sometimes called "lambda" or "birth") and the extinction parameter (sometimes called "mu", "death" or "background extinction"). By default `dads` runs a pure birth model (the speciation is set to 1 and the extinction to 0). However, you can easily change that by specifying your new birth death parameters:

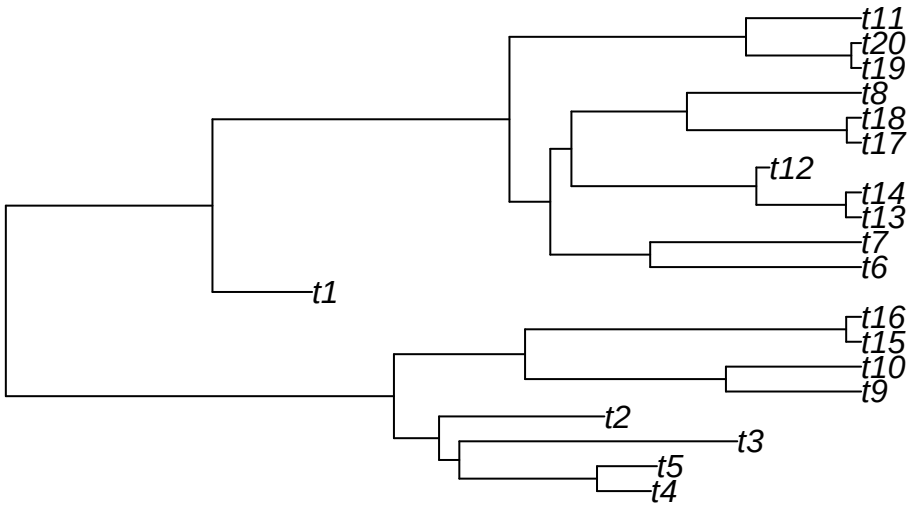
```
## my birth death parameters
my_params <- list(speciation = 1,
                  extinction = 1/3)
```

> Note that here it is not necessary to specify `speciation = 1` since this is the default option, you can always just change the parameter

of interest (e.g. changing `extinction = 0` to `extinction = 1/3`). However, we think it's good practice to attribute both parameters specifically to avoid any confusion. You can find more information about setting up more complex birth death parameters in this section.

You can then run the same birth death tree with extinction:

```
## Generating a birth death tree with extinctions:
my_tree <- dads(bd.params = my_params, stop.rule = my_stop_rule)
## Visualising the new tree
plot(my_tree)
```



2.2 Slightly more complex: simulating disparity and diversity

Chances are that you want to also simulate traits (disparity) along with your diversity (otherwise, we suggest using the `TreeSim` package that provides many more birth death models). Simulating traits is not much more complicated in `dads`: you'll simply need to create a `"traits"` object using the `make.traits` function. These objects can have increasing complexity (see the rest of this tutorial) but we will keep it simple here.

`"traits"` objects contain one or more processes which are the ways to generate the trait. The most common of these processes is the Brownian Motion. This is used by default with the `make.traits` function:

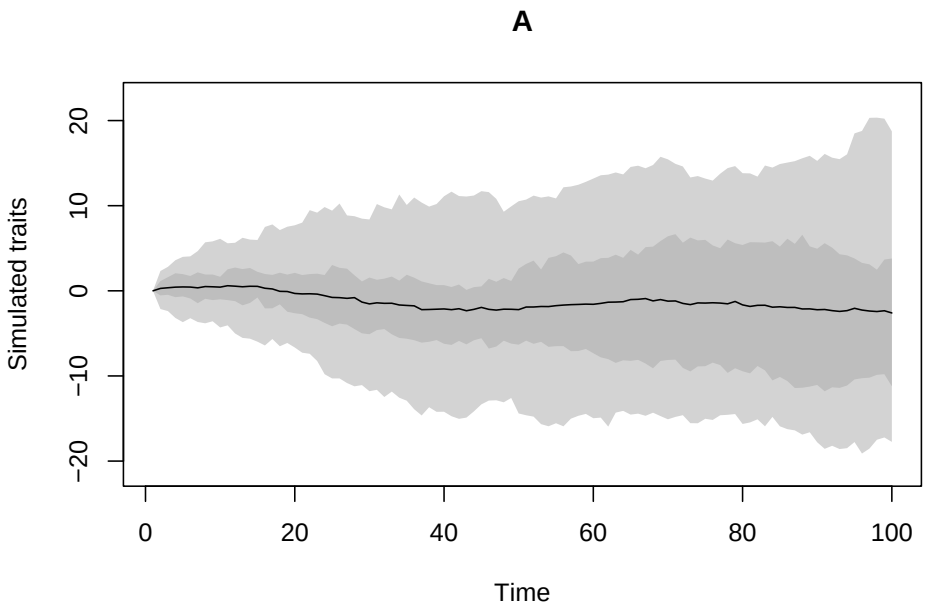
```
## Creating the traits object
my_trait <- make.traits()
```

This trait object can be simply printed (to see what's in it) or plotted (to see what the process looks like in the absence of a phylogeny):

```
## Which process is in here?
my_trait
```

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

```
## What does it look like?
plot(my_trait)
```



By default, this trait is called "A". This is not a really good name but you'll see more about specifying trait names later on. If this is what the process should look like (theoretically) you can then add its `"traits"` object to our previous `dads` function to generate the tree and the traits:

```
## Simulate disparity and diversity
my_data <- dads(bd.params = my_params,
                stop.rule = my_stop_rule,
                traits    = my_trait)
```

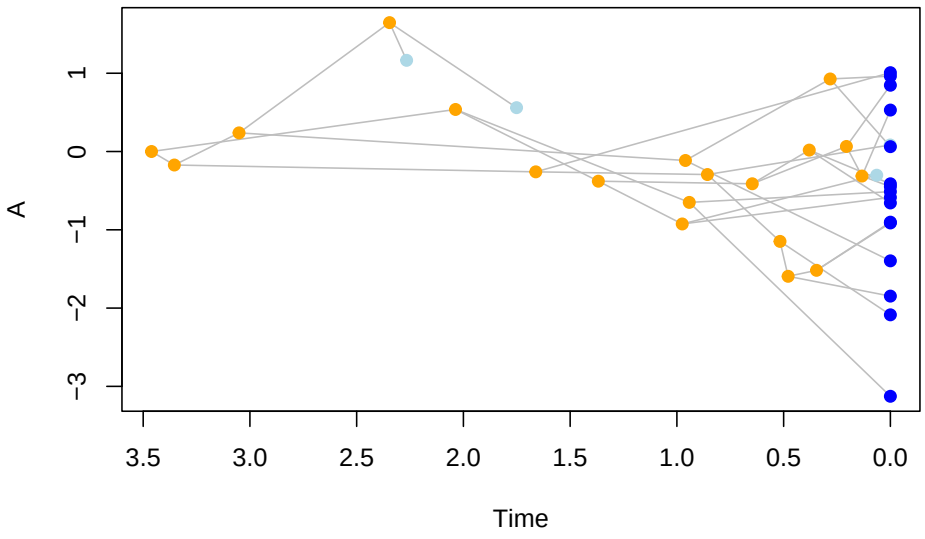
Et voilà! We now have a simple disparity and diversity simulation. We can see what's in the results by simply printing it or plotting it:

```
## What's in there
my_data
```

```
##  ---- dads object ----
```

```
## Simulated diversity data (x$tree):
##
## Phylogenetic tree with 20 tips and 19 internal nodes.
##
## Tip labels:
##   t1, t2, t3, t4, t5, t6, ...
## Node labels:
##   n1, n2, n3, n4, n5, n6, ...
##
## Rooted; includes branch lengths.
##
## Simulated disparity data (x$data):
## 1 trait for 1 process (A) with one starting value (0).
```

```
## Plotting the disparity and diversity
plot(my_data)
```



You can then extract the components you need for your specific analysis like so:

```
## Extracting the tree (a "phylo" object)
the_generated_tree <- my_data$tree
class(the_generated_tree)
```

```
## [1] "phylo"
```

```
## Extracting the data (a "matrix")
the_generated_data <- my_data$data
class(the_generated_data)
```

```
## [1] "matrix" "array"
```

You can find much more on how to design trait objects in the `make.traits` section.

2.3 Slightly more more complex: simulating linked disparity and diversity

The example above is also still pretty simple and easily done through a variety of `R` packages: here the trait and the tree are simulated at the same time but only the tree is simulating the trait (i.e. the trait value at a tip is affected by it's ancestor and the branch length leading to it) but not the other way around (the trait value does not affect the tree). It is possible to add this aspect using `"modifiers"` objects. `"modifiers"` are similar to `"traits"` in that you specify what should go in there and then feed it to your simulation.

`"modifiers"` affect two key steps of the birth-death process: the calculation of the waiting time (i.e. the component generating branch lengths) and the triggering of speciation or extinction events. These events can be modified using some `condition` and `modify` function. In other words, when reaching a certain condition specified by a `condition` function, the birth-death process will modify either the branch length or the speciation (or extinction) probability by applying a `modify` function.

You can use the function `make.modifiers` to design a specific `"modifiers"` object. By default, this function generates a `"modifiers"` object that affects branch length and speciation in the following way:

- branch length is a randomly drawn number from an exponential distribution with a rate equal to the current number of taxa multiplied by the sum of the speciation and extinction rates.
- speciation is triggered if a randomly drawn number (from a (0,1) uniform distribution) is smaller than the ratio between the speciation rate and the sum of the speciation and extinction rates. If that random number is greater, the lineage goes extinct.

Note that these are default for a birth death tree and were actually already applied in the examples before (without specifying a modifier):

```
## Make a default modifiers
default_modifiers <- make.modifiers()
## What's in it?
default_modifiers
```

```
##  ---- dads modifiers object ----
## No modifiers applied to the branch length, selection and speciation processes (defau
```

This will not do much to our simulations compared to the previous trait and tree simulation but we can provide our modifiers object to the `dads` function:

```
## Setting the simulation parameters
extinction_02 <- list(extinction = 0.2)
living_20     <- list(max.living = 20)
BM_trait      <- make.traits()

set.seed(1)
## Simulate disparity and diversity
default_data <- dads(bd.params = extinction_02,
                     stop.rule = living_20,
                     traits    = BM_trait,
                     modifiers = default_modifiers)
default_data
```

```
##  ---- dads object ----
## Birth death process with modifiers:
## speciation: 1.
## extinction: 0.2.
## No modifiers applied to the branch length, selection and speciation processes (default).
##
## Simulated diversity data (x$tree):
##
## Phylogenetic tree with 24 tips and 23 internal nodes.
##
## Tip labels:
##   t1, t2, t3, t4, t5, t6, ...
## Node labels:
##   n1, n2, n3, n4, n5, n6, ...
##
## Rooted; includes branch lengths.
##
## Simulated disparity data (x$data):
## 1 trait for 1 process (A) with one starting value (0).
```

Note however that the printing information is now updated to state that you've added a modifier (even though it's a default one).

For more interesting simulations however, you can provide modifiers that actually modify the birth death process. We can create one for example that makes species go extinct if their ancestor have a negative trait value. For that we need to create a modifiers object that modifies the `speciation` process with a specific condition and a specific modification when that condition is met. For a speciation to occur (and a species to not go extinct), the algorithm draws a random value x between 0 and 1 and if the value is smaller than the speciation parameter divided by the speciation + extinction parameter, the lineage speciates ($x < (\lambda\ (\lambda + \mu))$), else it goes extinct.

Here we can create a modifier forcing the lineage to go extinct every time a

specific condition is met (say the lineage trait value is negative). In other words, if the lineages trait value is negative, make sure that x is equal to 1 (so that the speciation condition $x < (\lambda\ (\lambda + \mu))$ is never met).

First we need to create a condition function to trigger this (non) speciation event. We can do that by specifying our `condition` function (when the ancestor is trait is negative) apply our modification (here the `staying.alive` function). For that we can use the `parent.traits` utility function that is optimised for accessing traits in the birth death process (but you can of course write your own). This function intakes the `trait.values` and `parent.lineage` arguments, two arguments that you can leave named as they are to facilitate `dads`' understanding of what you want to assess:

```
## Triggering a modification only if the ancestor trait is negative
negative.ancestor <- function(trait.values, lineage) {
    return(all(parent.traits(trait.values, lineage) < 0))
}
```

> Note that we use the function `all` here to evaluate all traits: i.e. if the data has more than one trait we trigger the modification only if all the trait values are negative.

The we are going to create the modification function. This function must intake the argument `x` and, in our case, returns the same value no matter what: `1` so that speciation never happens when the `negative.ancestor` function is triggered.

```
## Always go extinct
going.extinct <- function(x) return(1)
```

We can then provide these two functions (the condition `negative.ancestor` and how to modify the speciation event when this condition is met `staying.alive`). If you are an advances `dads` user, you can design your own `speciation` function but if you just want to use a normal `speciation` function, you can use the default one from `dads` called… `speciation`.

```
## Making a modifier for species to go extinct if
## their ancestor's trait value is (or are) negative
negatives_extinct <- make.modifiers(
            ## If the following condition is met...
            condition = negative.ancestor, # Does the lineage has a negative ancestor?
            ## Then apply the following modifier...
            modify = going.extinct, # The species goes extinct
            ## To the the speciation process.
            speciation = speciation) # Here the speciation() process is default

## What's in it?
negatives_extinct
```

```
##  ---- dads modifiers object ----
## Default branch length process.
## Default selection process.
## Speciation process is set to speciation with a condition (negative.ancestor) and a modifier (g
```

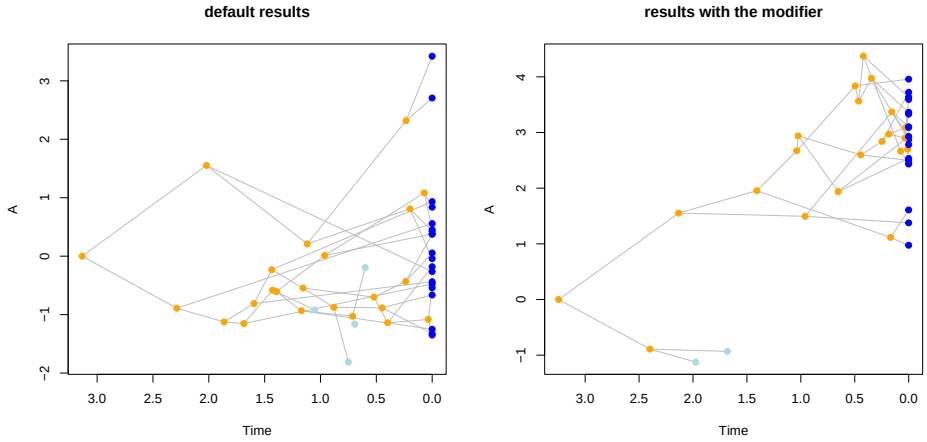
Note that the `make.modifiers` function tests whether the input is compatible with `dads` by default so unless you have an error message, your `modifiers` will work! We can now simulate our tree and traits with our modifier: species will go extinct if their ancestor have a negative trait value:

```
set.seed(1)
## Simulate disparity and diversity
biased_data <- dads(bd.params = extinction_02,
                    stop.rule = living_20,
                    traits    = BM_trait,
                    modifiers = negatives_extinct)
biased_data
```

```
##  ---- dads object ----
## Birth death process with modifiers:
## speciation: 1.
## extinction: 0.2.
## Default branch length process.
## Default selection process.
## Speciation process is set to speciation with a condition (negative.ancestor) and a modifier (g
##
## Simulated diversity data (x$tree):
##
## Phylogenetic tree with 22 tips and 21 internal nodes.
##
## Tip labels:
##   t1, t2, t3, t4, t5, t6, ...
## Node labels:
##   n1, n2, n3, n4, n5, n6, ...
##
## Rooted; includes branch lengths.
##
## Simulated disparity data (x$data):
## 1 trait for 1 process (A) with one starting value (0).
```

We can now compare the two trees and their trait values. Note that we've used the same starting seed for both trees so the only thing differing between them is the `"modifier"` object which leads to very different trees!

```
par(mfrow = c(1,2))
plot(default_data, main = "default results")
plot(biased_data, main = "results with the modifier")
```



You can find much more how to design modifiers in the `make.modifiers` section.

Chapter 3

Simulating traits

In `dads`, traits are simulating by providing a `traits` object to the `traits` argument. This object is generated using the `make.traits`.

> **What is a trait in biology?** This is one of these epistemological question like what *is* a species that is not covered by all means in this package. For the purpose of *this manual*, **biological trait** can be any coherent process of any number of dimensions which can or cannot be independent. Basically anything can be a trait as well as it's defined by the user (e.g. one trait can be a 20 dimensional Brownian process that's correlated to a 1D OU process - which is another trait). For the purpose of the syntax in *this package* though, a **trait in `dads`** is a column in the output trait matrix. For example, one might be interested in a biological trait that is "beak shape" which is defined as one complex biological trait that is the Procrustes superimposition of 4 3D landmarks and 300 3D semi-landmarks. This single "beak shape" **biological trait** is then composed of 912 **traits in `dads`** (i.e. a matrix of $(4 + 300) \times 3$ columns).

3.1 The process (`process`)

The function `make.traits` allows you to design the process of a trait or a set of traits. Here, the process of a trait designates the rules to generate the trait through time while simulating a phylogeny. This process can depend on the previous state in the tree (i.e. the trait of the ancestor) and the branch length to the descendant. One classic example is the Brownian motion process (or Weiner process). Note that it *can* depend on both the ancestor and the branch length but does *not necessary needs* (i.e. the process can be only based on the previous state or only on branch length or on neither).

3.1.1 The syntax (how to code a process?)

Trait processes in `dads` are functions that must always intake the following arguments by default.

- `x0`: the previous trait value(s)
- `edge.length`: the branch length value
- `...`: a placeholder for any extra arguments

For example, the following function would be a valid process that always generate the *true* trait value: 42!. In this example, the process is not dependent on neither the previous state (`x0`) and the branch length (`edge.length`).

```
## A valid (but useless?) process
valid.process <- function(x0 = 0, edge.length = 1, ...) {
    return(42)
}
```

> Note that in this function definition the arguments `x0` and `edge.length` have a default value set to `0` and `1` respectively. In practice, these arguments are effectively set to the correct values in the `dads` internal function (i.e. whatever `x0` and `edge.length` are at that specific time of the process) but providing a default can help speed up the algorithms (specifically all the internal checks).

On the other hand, the following process (a unidimensional Brownian motion) is incorrect (it's missing `edge.length` and `...`):

```
## A wrongly formated process
invalid.process <- function(x0 = 0) {
    return(rnorm(1, mean = x0))
}
```

3.1.2 Using a "process" in dads

You can design your own process as a function (as long as it has a valid syntax). Alternatively, the `dads` package proposes inbuilt processes, namely a multidimensional Brownian motion (`BM.process`) or a a multidimensional Ornstein-Uhlenbeck process (`OU.process`). You can find the list of implemented process by looking at the `?trait.process` manual page in `R`.

Once a process is chosen, you can feed it to the `make.traits` function:

```
## Creating a trait object
my_trait_object <- make.traits(process = BM.process)
```

This creates `"dads" "traits"` objects that you can print, and visualise using the `plot` function:

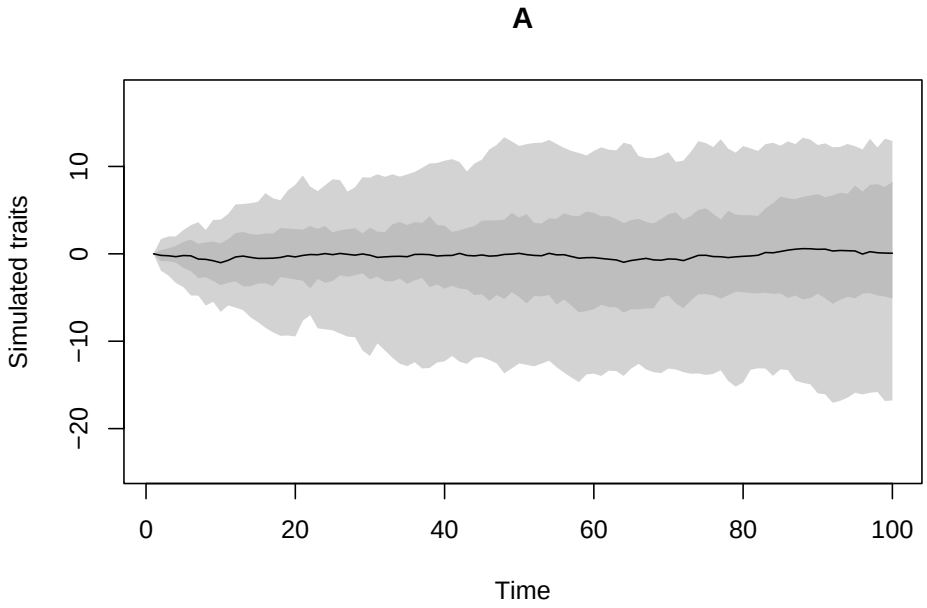
```
## The class of the object
class(my_trait_object)
```

```
## [1] "dads"   "traits"
```

```
## What's in it?
my_trait_object
```

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

```
## What does the process looks like
plot(my_trait_object)
```



Note that you can see the multiple options for plotting the trait process by looking at `?plot.dads` manual. Furthermore, you can look at what's actually in the object using this specific syntax (this applies to every object handled by the `dads` package):

```
## What's actually in that object?
print.dads(my_trait_object, all = TRUE)
```

```
## $main
## $main$A
## $main$A$process
## function (x0 = 0, edge.length = 1, Sigma = diag(length(x0)),
##     ...)
## {
##     return(t(MASS::mvrnorm(n = 1, mu = x0, Sigma = sqrt(Sigma^2 *
##         edge.length), ...)))
```

```
## }
## <bytecode: 0x55e278dc9cd8>
## <environment: namespace:dads>
##
## $main$A$start
## [1] 0
##
## $main$A$trait_id
## [1] 1
##
##
##
## $background
## NULL
```

As traits can get more and more complex, the automatic printing of its summary allows for a easier display of what's in the traits object.

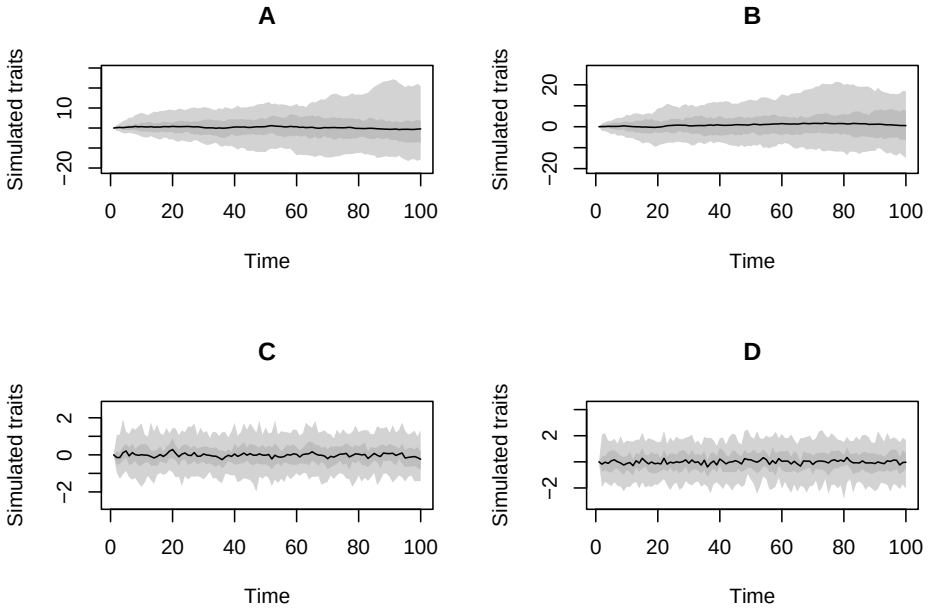
Note that it is possible to make `"traits"` objects with multiple processes (that can be the same):

```
## 4 traits: two BM, one OU and one normal non process
four_traits <- make.traits(process = c(BM.process,
                                       BM.process,
                                       OU.process,
                                       no.process))
four_traits
```

```
##  ---- dads traits object ----
## 4 traits for 4 processes (A, B, C, D) with one starting value (0).
```

You can visualise them individually using the `trait` argument in `plot.dads`:

```
## Plot options (4 plots in one window)
par(mfrow = c(2,2))
plot(four_traits, trait = 1)
plot(four_traits, trait = 2)
plot(four_traits, trait = 3)
plot(four_traits, trait = 4)
```



3.2 The number of traits `n` and the starting values `start`

Two further important arguments are `n` the number of traits per process and `start` the starting values for all traits. By default they are set to `n = 1` and `start = 0`. This means that `make.traits` will assume that your processes are always unidimensional by default and that they always start with the value `0`. It is however possible to change these values.

For example you can use the following to create a three dimensional Brownian motion with each dimensions starting with the value `1`:

```
## Multidimensional Brownian motion
make.traits(BM.process, n = 3, start = 1)
```

```
##  ---- dads traits object ----
## 3 traits for 1 process (A:3) with one starting value (1).
```

Or the following with each dimensions starting with different values (respectively `1`, `2` and `3`):

```
## Multidimensional Brownian motion
make.traits(BM.process, n = 3, start = c(1,2,3))
```

```
##  ---- dads traits object ----
## 3 traits for 1 process (A:3) with different starting values (1,2,3).
```

Note that the number of traits are distributed per processes. If the traits contains multiple process, the number of traits are distributed per processes:

```
## two 3D processes (BM and OU)
make.traits(c(BM.process, OU.process), n = 3)
```

```
##  ---- dads traits object ----
## 6 traits for 2 processes (A:3, B:3) with one starting value (0).
```

```
## one 1D processes (BM) and one 4D process (OU)
make.traits(c(BM.process, OU.process), n = c(1, 4))
```

```
##  ---- dads traits object ----
## 5 traits for 2 processes (A:1, B:4) with one starting value (0).
```

And starting values are distributed for all the traits or for the traits one by one:

```
## two 3D processes (BM and OU) starting with 1
make.traits(c(BM.process, OU.process), n = 3, start = 1)
```

```
##  ---- dads traits object ----
## 6 traits for 2 processes (A:3, B:3) with one starting value (1).
```

```
## two 3D processes (BM and OU) starting with values 1 to 6
make.traits(c(BM.process, OU.process), n = 3, start = 1:6)
```

```
##  ---- dads traits object ----
## 6 traits for 2 processes (A:3, B:3) with different starting values (1,2,3,4,5,6).
```

```
## two 3D processes (BM and OU) with the two first ones starting
## with 1 and the 4 other ones with the default (0)
make.traits(c(BM.process, OU.process), n = 3, start = c(1,1))
```

```
## Warning in make.traits(c(BM.process, OU.process), n = 3, start = c(1, 1)): Only
## the first 2 starting values were supplied for a required 6 traits. The missing
## start values are set to 0.
```

```
##  ---- dads traits object ----
## 6 traits for 2 processes (A:3, B:3) with different starting values (1,1,0,0,0,0).
```

3.2.1 What even is a trait?

The definition of what a trait is can vary quiet a lot depending on the context of an analysis and the field. Because it would be impossible to accommodate all definitions in `dads` we had to go with an arbitrary one: a trait is whatever you define as a trait! A trait can be uni-dimensional as the measurement of a feature of an organism (e.g. leaf surface, femur length, etc…) but can also be multi-dimensional description of a feature, for example in 3D geometric morphometric, a trait could be defined as "position of landmark X" (which will be a trait with three dimensions, x, y and z) or in ecology, the location of a plant can

be a single 2D trait (expressed as latitude and longitude). In dads, the **process** corresponds to this trait definition (e.g. a process can be of n-dimensions and represents one organisms feature) and the **traits** represents the number of dimensions in total. So in the examples above, this is how the following traits are interpreted by dads:

```
## Three traits with one process:
make.traits(BM.process, n = 3, start = c(1,2,3))
## Six traits with two processes:
make.traits(c(BM.process, OU.process), n = 3)
## Five traits with two processes
make.traits(c(BM.process, OU.process), n = c(1, 4))
```

3.3 Extra argument for the processes with process.args

You can also feed extra arguments to your process(es) functions. For example, the inbuilt process no.process (that is just a number generator not based on the previous value x0 or the branch length) can intake a specific random number generator as a function:

```
## no process trait using the normal distribution (default)
make.traits(no.process, process.args = list(fun = rnorm))
```

```
##  ---- dads traits object ----
## 1 trait for 1 process (A) with one starting value (0).
## process A uses the following extra argument: fun;
```

```
## no process trait using the uniform distribution
## bounded between 1 and 100
make.traits(no.process, process.args = list(fun = runif, min = 1, max = 100))
```

```
##  ---- dads traits object ----
## 1 trait for 1 process (A) with one starting value (0).
## process A uses the following extra arguments: fun,min,max;
```

You can also add multiple extra arguments for multiple processes giving them as a list.

```
## Two traits with no process:one normal and one uniform (1,100)
make.traits(process      = c(no.process, no.process),
            process.args = list(list(fun = rnorm),
                                list(fun = runif, min = 1, max = 100)))
```

```
##  ---- dads traits object ----
## 2 traits for 2 processes (A, B) with one starting value (0).
## process A uses the following extra argument: fun;
## process B uses the following extra arguments: fun,min,max;
```

If one process do not need extra argument you must still give it and extra `NULL` process argument:

```
## Three traits with no process:
## one default, one lognormal and one uniform (1,100)
make.traits(process      = c(no.process, no.process, no.process),
            process.args = list(## Extra arguments for the first process (none)
                                list(NULL),
                                ## Extra arguments for the second process
                                list(fun = rlnorm),
                                ## Extra arguments for the third process
                                list(fun = runif, min = 1, max = 100)))
```

```
##  ---- dads traits object ----
## 3 traits for 3 processes (A, B, C) with one starting value (0).
## process B uses the following extra argument: fun;
## process C uses the following extra arguments: fun,min,max;
```

3.4 Naming the traits with `trait.names`

As traits become more and more complex, it can be useful to give clearer names to each process. This is easily done using the `trait.names` argument that attributes one name per process:

```
## A simple trait with a proper name
simple_trait <- make.traits(trait.names = "1D Brownian Motion")
simple_trait
```

```
##  ---- dads traits object ----
## 1 trait for 1 process (1D Brownian Motion) with one starting value (0).
```

This becomes more useful if we use the complex example above:

```
## Three named traits with no process:
## one default, one lognormal and one uniform (1,100)
make.traits(process      = c(no.process, no.process, no.process),
            process.args = list(## Extra arguments for the first process (none)
                                list(NULL),
                                ## Extra arguments for the second process
                                list(fun = rlnorm),
                                ## Extra arguments for the third process
                                list(fun = runif, min = 1, max = 100)),
            ## Naming each trait
            trait.names  = c("Normal", "LogNormal", "Uniform(1,100)"))
```

```
##  ---- dads traits object ----
## 3 traits for 3 processes (Normal, LogNormal, Uniform(1,100)) with one starting value
## process LogNormal uses the following extra argument: fun;
```

```
## process Uniform(1,100) uses the following extra arguments: fun,min,max;
```

3.5 Combining multiple traits with add

You can also add traits to already existing trait objects using the simple `add` option. This option just intakes a `"dads" "traits"` object and the additional process(es) will be added to it. For example:

```
## Creating on simple default Brownian motion
one_process <- make.traits(trait.names = "BM")

## Creating a new trait (a 3D OU.process)
## and adding the previous one
two_processes <- make.traits(OU.process, n = 3, add = one_process,
                             trait.names = "3D OU")

## Only one process
one_process
```

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

```
## The two processes
two_processes
```

```
##  ---- dads traits object ----
## 4 traits for 2 processes (BM:1, 3D OU:3) with one starting value (0).
```

3.6 Using a background trait

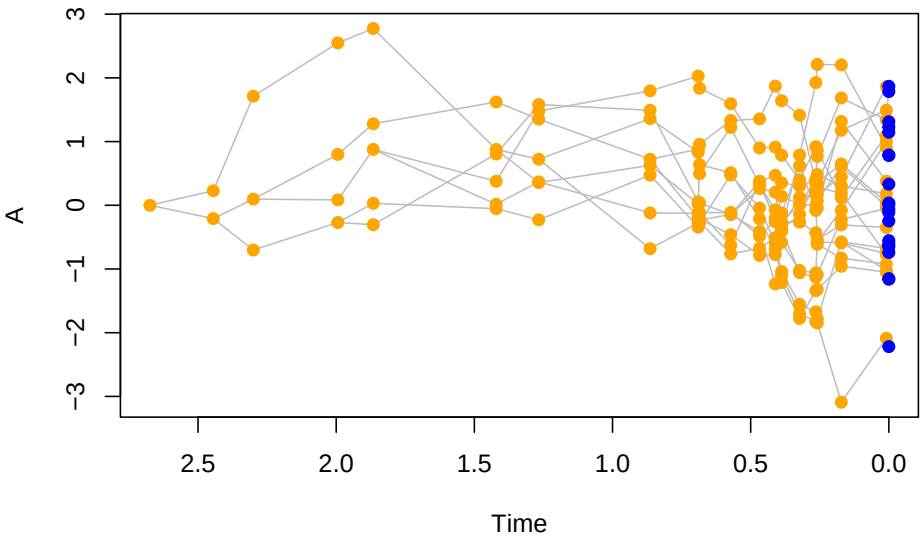
`traits` objects also allow a background trait to be used when traits are simulated (step 3 here). This basically allows traits to be simulating for *all* tips whenever a trait is generated for one tip. This can be useful for keeping track of trait values along the simulation (*cf* just at bifurcating nodes). The `background` argument takes any output from the `make.traits` function in a nested way:

```
## Generating a default BM trait:
BM_trait <- make.traits()
## Generating an OU trait with a background BM trait
my_trait <- make.traits(process = OU.process, background = BM_trait)
```

Note that technically you can nest as many background traits as you want (e.g. `make.traits(background = make.traits(background = make.traits(...)))` is valid). However, you should always make sure that the background trait has the same dimensions as the normal (main) trait. When using a trait with background, your tree will have internal singleton

nodes (i.e. nodes linking to one ancestor and only one descendant). You can remove these nodes using the `drop.things` function.

```
set.seed(1)
## Generating a pure birth tree with the background trait
tree_bkg <- dads(stop.rule = list(max.taxa = 20),
                 traits    = my_trait)
## This tree has many internal singleton nodes
plot(tree_bkg)
```

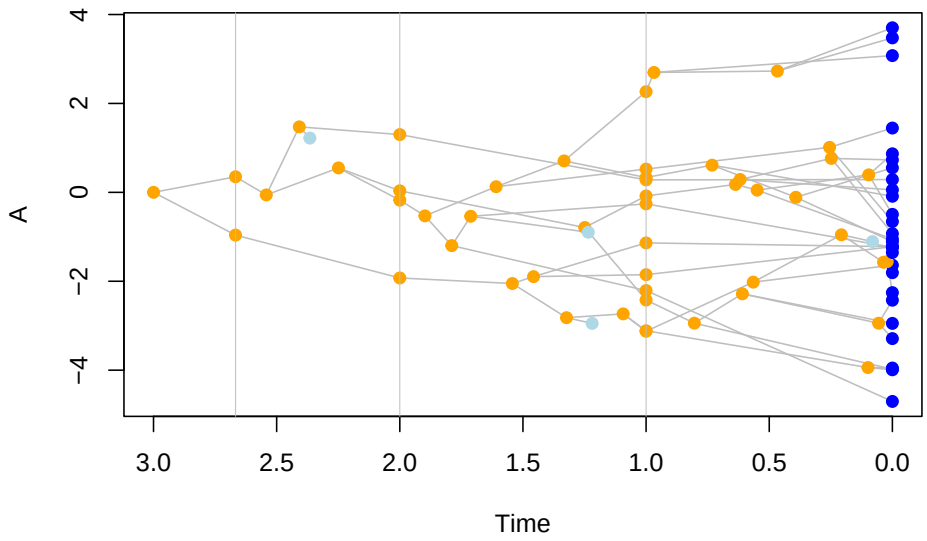


3.7 Saving trait values at different time steps

Alternatively to having a background trait, you can also simulate a tree by generating traits at specific time steps with the `save.steps` option in `dads`. This will apply the `traits` object to all lineages currently alive at the required time steps. These time steps can be either regular by providing a single numeric value; e.g. `save.steps = 0.1` will get a snapshot of the trait values every 0.1 units of time. Or specific, by providing a specific set of values; e.g. `save.steps = c(1, 1.2, 3)` will make a snapshot of the trait values at the required time steps.

```
set.seed(123)
## Generating a birth-death tree with a BM trait and saving steps at specific times
tree_steps <- dads(stop.rule  = list(max.time = 3),
                   bd.params  = list(speciation = 1, extinction = 0.1),
                   traits     = make.traits(),
                   save.steps = c(1/3, 1, 2))
## This also creates internal singleton nodes
plot(tree_steps)
```

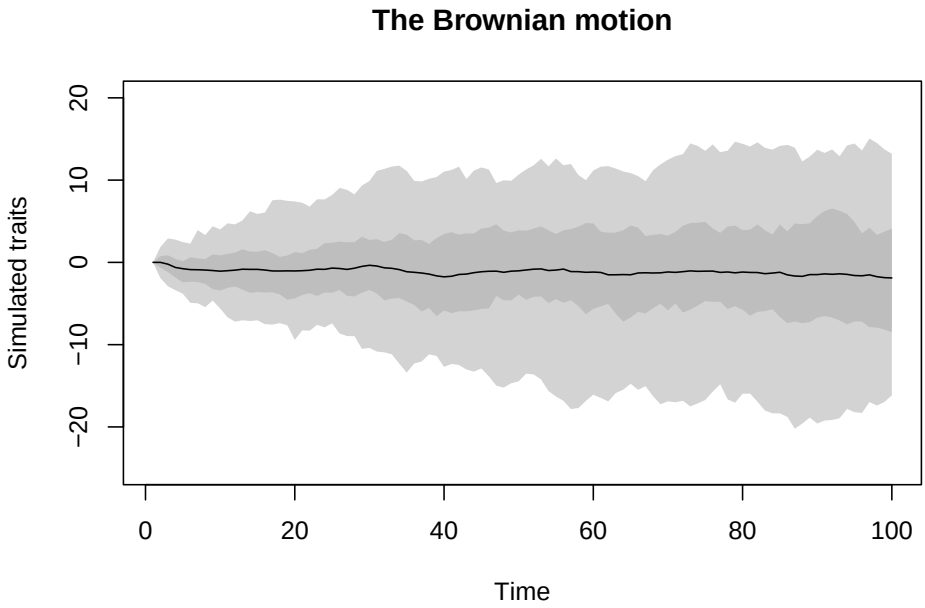
```
abline(v = 3 - c(1/3, 1, 2), lwd = 0.5, col = "grey")
```



3.8 Traits implemented in dads

If you don't want to design your own trait process, you can use one of the following trait processes that are currently implemented in `dads`. You can find more information about their many options using their specific manuals in R or the generic `?trait.process`:

- `BM.process`: this is the well famous Brownian motion process.

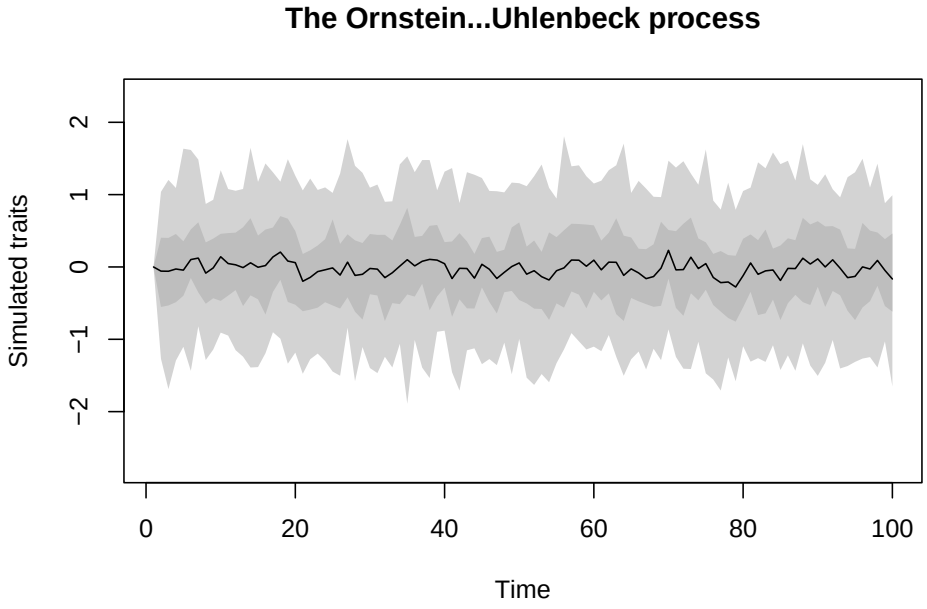


- `OU.process`: this is the equally famous Ornstein–Uhlenbeck process.

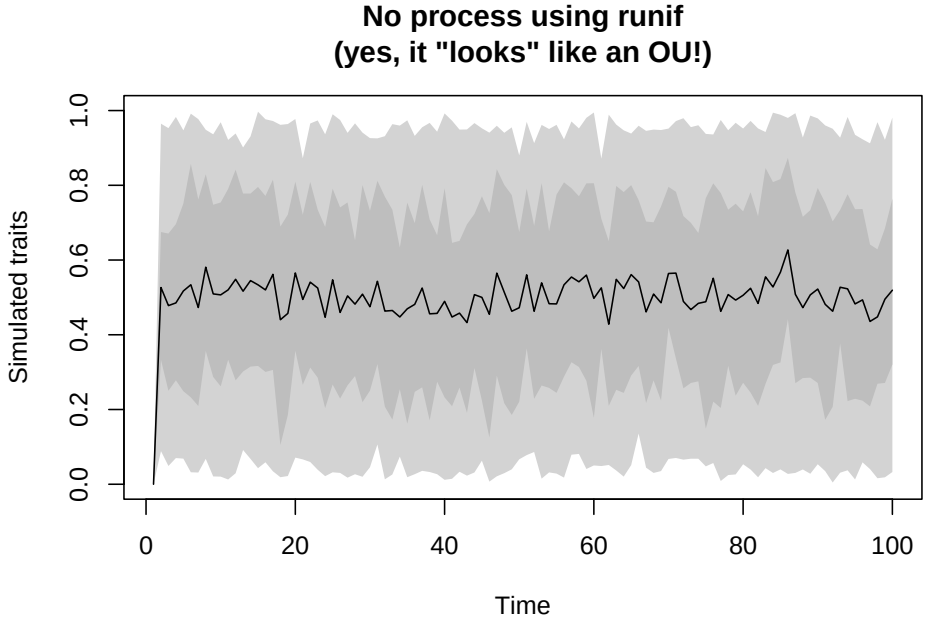
```
## Warning in title(...): conversion failure on 'The Ornstein-
Uhlenbeck process'
## in 'mbcsToSbcs': dot substituted for <e2>
```

```
## Warning in title(...): conversion failure on 'The Ornstein-
Uhlenbeck process'
## in 'mbcsToSbcs': dot substituted for <80>
```

```
## Warning in title(...): conversion failure on 'The Ornstein-
Uhlenbeck process'
## in 'mbcsToSbcs': dot substituted for <93>
```

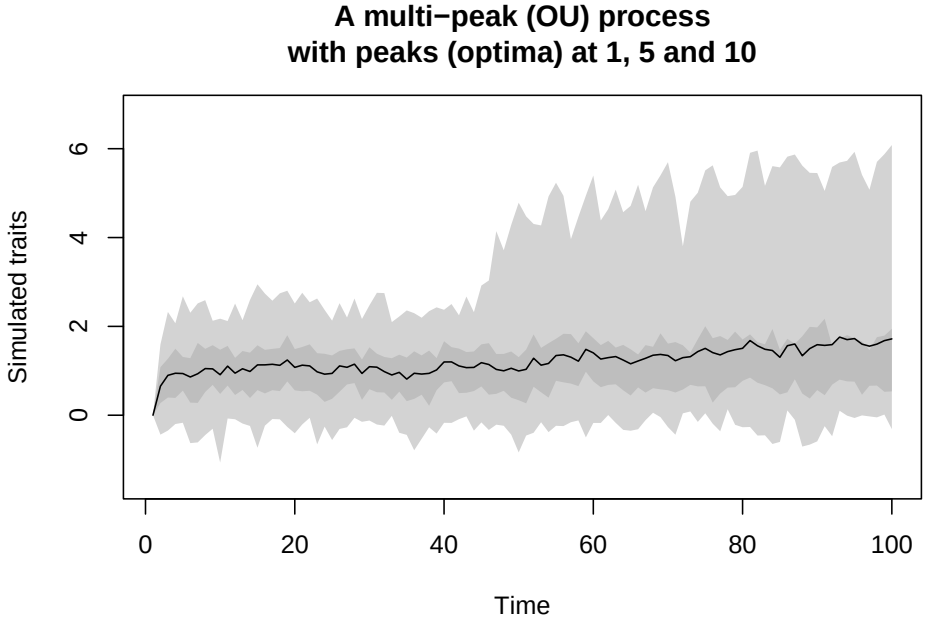


- `no.process`: this process has… no process. In other words, this is a non time dependent process (the simulated value does not depends on the ancestors' value nor the branch length). It's basically a place holder for a random sampling function like `rnorm` (default), `runif`, `rlnorm`, etc…

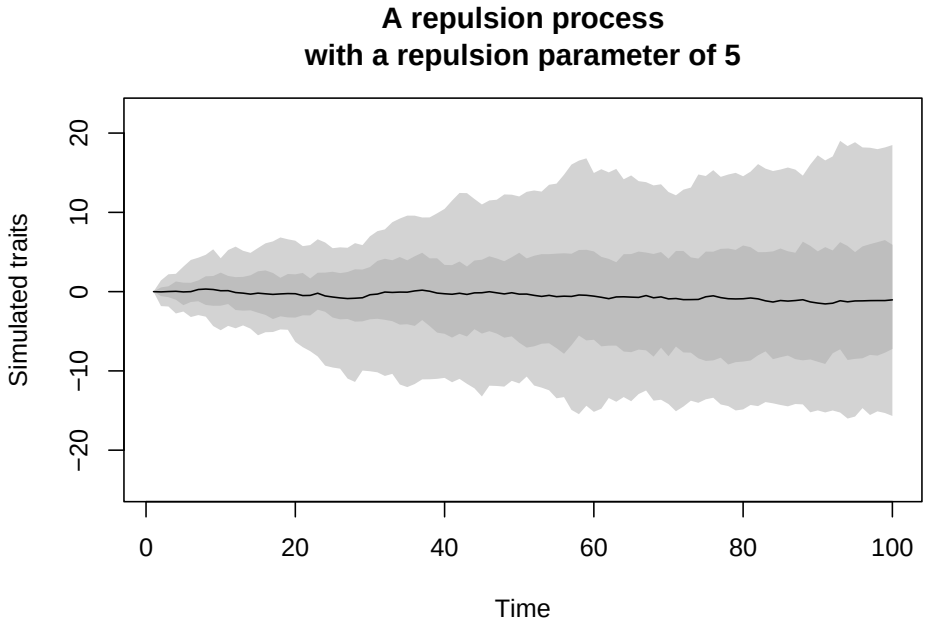


- `multi.peak.process`: this process is a modified version of the `OU.process` that can take multiple local optima. The default OU

process has one optimum towards which the values are drawn with the alpha parameter (the elastic band) which is usually 0. However, with this `multi.peak.process` we can set multiple values towards which values can be attracted with the same alpha parameter.



- `repulsion.process`: this is a modified version of the `BM.process` where instead of accumulating normally through time, new trait values are more likely to be different than their ancestor following a `repulsion` parameter.



3.9 Testing the traits with `test`

This bit is more for development. We highly suggest leaving `test = TRUE` so that `make.traits` returns an error if a process or its additional arguments (`process.args`) are not formatted correctly. `make.traits` will send an error if the trait cannot be directly passed to `dads`. However, in some specific cases (again, probably mainly for development and debugging) it could be useful to skip the tests using `test = FALSE`.

3.10 Templates for making your very own process

As detailed above, any process of your own design will work as long as it is a function that takes at least the arguments `x0` and `edge.length`. You can be imaginative and creative when designing your own process but here are two detailed example functions for a unidimensional Brownian Motion and Ornstein-Uhlenbeck process that you can use for a start (or not). Remember it is good practice for `dads` processes to set all the arguments with default values (just in case).

> Note that the functions below are not equal to the already implemented `BM.process` and `OU.process` but an easier to edit version that you can use as a template:

3.10.1 A simple Brownian Motion process template

```
## A simple Brownian motion process
my.BM.process <- function(x0 = 0, edge.length = 1, sd = 1, ...) {
    ## Drawing a random number from a normal distribution
    ## with x0 as the and a given standard deviation
    ## and depending on branch (edge) length
    result <- rnorm(n = 1, mean = x0, sd = sqrt(sd^2 * edge.length))

    ## Return the number
    return(result)
}
```

3.10.2 A simple Ornstein-Uhlenbeck process template

```
## A simple Ornstein-Uhlenbeck motion process
my.OU.process <- function(x0 = 0, edge.length = 1, var = 1, alpha = 1, ...) {
    ## Calculate the mean based on alpha
    mean <- x0 * exp(-alpha)
    ## Calculate the standard deviation based on alpha and the variance
    sd <- sqrt(var/(2 * alpha) * (1 - exp(-2 * alpha)))
    ## Draw a random number from a normal distribution
    ## using this mean and standard deviation
    ## and depending on branch (edge) length
    result <- rnorm(n = 1, mean = mean, sd = sqrt(sd^2 * edge.length))

    ## Return the number
    return(result)
}
```

Chapter 4

Modifying the birth-death process

"modifiers" have a similar structure than "traits" where you can design an object with increasing complexity, starting with the simplest modifiers that doesn't modify anything (using the default arguments):

```
## Making a default modifier (no modification)
my_default_modifiers <- make.modifiers()
my_default_modifiers
```

```
##  ---- dads modifiers object ----
## No modifiers applied to the branch length, selection and speciation processes (default).
```

Similarly to "traits" objects, "modifiers" are also printed by default using print.dads. You can see details about what's actually in the object using print.dads(my_default_modifiers, all = TRUE). However, contrary to "traits", you cannot plot "modifiers".

4.1 The default modifier (how the process is working)

The modifiers modify the core of the birth death process as implemented in dads.

By default, the birth-death process in dads uses this modifier:

```
## What is actually in the default modifier?
print(make.modifiers(), all = TRUE)
```

```
## $waiting
```

```
## $waiting$fun
## function (bd.params, lineage = NULL, trait.values = NULL, modify.fun = NULL)
## {
##     return(rexp(1, sum(lineage$n * (bd.params$speciation + bd.params$extinction))))
## }
## <bytecode: 0x55e27b08d1b8>
## <environment: namespace:dads>
##
## $waiting$internal
## NULL
##
##
## $selecting
## $selecting$fun
## function (bd.params, lineage = NULL, trait.values = NULL, modify.fun = NULL)
## {
##     return(sample(lineage$n, 1))
## }
## <bytecode: 0x55e27b11eb90>
## <environment: namespace:dads>
##
## $selecting$internal
## NULL
##
##
## $speciating
## $speciating$fun
## function (bd.params, lineage = NULL, trait.values = NULL, modify.fun = NULL)
## {
##     return(runif(1) < (bd.params$speciation/(bd.params$speciation +
##         bd.params$extinction)))
## }
## <bytecode: 0x55e27b127560>
## <environment: namespace:dads>
##
## $speciating$internal
## NULL
##
##
## $call
## $call$waiting
## $call$waiting$fun
## [1] "default"
##
##
## $call$selecting
```

```
## $call$selecting$fun
## [1] "default"
##
##
## $call$speciating
## $call$speciating$fun
## [1] "default"
```

This contains a lot of information, much of it is actually not useful for using the modular aspects of `dads` at a high user level. I.e. unless you want to code very specific things, you won't need most of the information. The essential are these three functions in the elements named `"waiting$fun"`, `"selecting$fun"` and `"speciating$fun"`:

1. `"waiting$fun"` is the branch length function (defined in details below) which returns a randomly drawn number from an exponential distribution with the rate of the number of taxa multiplied by the speciation and extinction rate: $n \times (\lambda + \mu)$ (here is uses the extra function `sum` for internal modularity reasons). **This function is responsible for the growth of the length/age of the tree**

```
rexp(1, sum(lineage$n * (bd.params$speciation + bd.params$extinction)))
```

With `lineage$n` being the number of lineages and `bd.params$speciation` and `bd.params$extinction` the speciation and extinction parameters (these specific terms are defined in details below).

2. `"selecting$fun"` is the selection function (defined in details below) which, after the waiting time defined above returns a randomly selected lineage (a tip) among the existing ones. **This function is responsible for the branch selection**.

```
sample(lineage$n, 1)
```

3. `"speciating$fun"` is the speciation function (defined in details below) which randomly draws a number (`runif(1)`) and depending on the speciation and extinction parameters makes the selected lineage (from 2.) speciate (`TRUE`) or go extinct (`FALSE`). **This function is responsible for the speciation or extinction of species**.

```
runif(1) < (bd.params$speciation/(bd.params$speciation + bd.params$extinction))
```

We will have a look at all these aspects below in more details which should make things clearer (but you can always refer to the default here for information).

4.2 The branch length function (`branch.length`)

The first argument in `"modifiers"` is the branch length function (`branch.length`) this is the function that will be executed in `dads` to generate branch length.

Note that in the `dads` algorithm, branch length is not generated *directly* but actually results of the waiting time. In other words, the `branch.length` function just affects waiting time for all taxa present at any time in the simulation. These taxa can then either go extinct (stopping the "growth" of it's branch length) or survive (continuing the "growth").

By default, branch length (or waiting/growth) is a randomly drawn number from an exponential distribution with the rate of the number of taxa multiplied by the speciation and extinction rate: $n \times (\lambda + \mu)$ (where n is the number of taxa currently present in the simulation, λ and μ are respectively the speciation and extinction rates). This default function is simply called `branch.length` in `dads` and can be used as a modifier as follows:

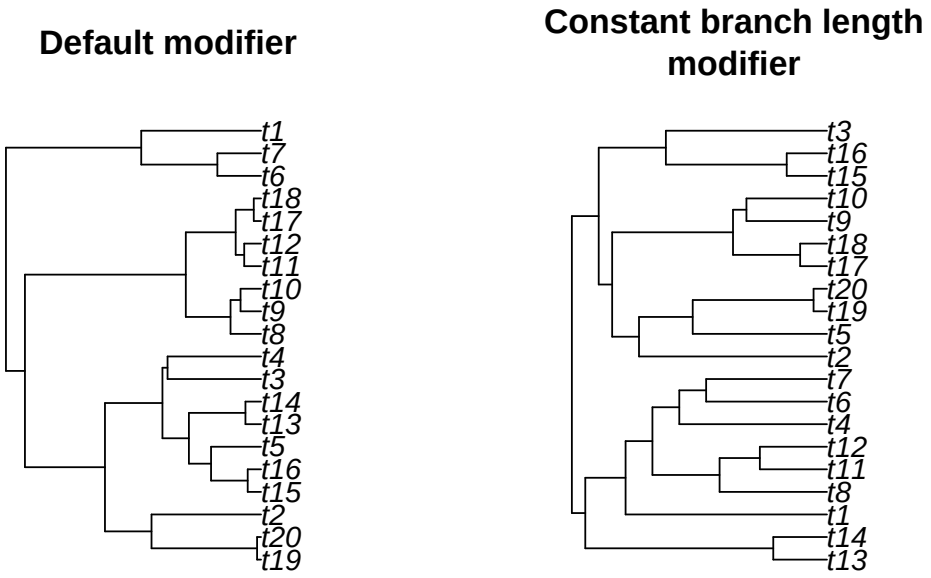
```
## Specifying the default modifier
default_modifiers <- make.modifiers(branch.length = branch.length)

## Setting some parameters for generating trees
bd_params <- list(extinction = 0)
stop_rule <- list(max.living = 20)

## Generating a tree with the default branch length parameter
set.seed(0)
default_tree <- dads(bd.params = bd_params,
                     stop.rule = stop_rule,
                     modifiers = default_modifiers)
```

Of course, the point of the modularity here is that you can provide your own function for generating branch length. For example, we might be interested in what our tree would look like if we'd use a simple constant branch length generation (instead of randomly drawing it from an exponential distribution). We can do so by declaring our own `branch.length` function and adding it to a `"modifiers"` object.

```
## A constant branch length generator
## (note that the output must be numeric, not integer)
constant.brlen <- function() {
    return(as.numeric(1))
}

## Creating the modifiers object
constant_modifier <- make.modifiers(branch.length = constant.brlen)

## Generating a new tree with this modifier
set.seed(0)
modified_tree <- dads(bd.params = bd_params,
                      stop.rule = stop_rule,
                      modifiers = constant_modifier)
```

And we can visualise the difference between both resulting trees:

```
par(mfrow = c(1,2))
plot(default_tree,  main = "Default modifier")
plot(modified_tree, main = "Constant branch length\nmodifier")
```



It is of course possible to use more complex branch length modifiers that intakes different conditions and specific modification rather than simply always output a value of one.

4.2.1 The allowed arguments

You can create a function for `branch.length`, `selection` and `speciation` that involve any of the following arguments:

- `bd.params`: a named list containing `"numeric"` values that contains the birth death parameters (at least `"speciation"` and `"extinction"`);
- `lineage`: a named list containing the lineage data (see below).
- `trait.values`: a `"matrix"` containing `"numeric"` values with the trait names as column names and the lineages ID as row numbers (you can use it with the function `parent.traits` to access the trait of the previous node for example).
- `modify.fun`: a `"list"` of named `"function"` (usually passed through `condition` and `modify`).

The `lineage` list contains the following elements (missing elements are allowed):

- `lineage$parents`: an `"integer"` vector: the list of parent lineages;
- `lineage$livings`: an `"integer"` vector: the list of lineages still not extinct;

- `lineage$drawn`: a single `"integer"`: the ID of the selected lineage;
- `lineage$current`: a single `"integer"`: the selected lineage (is equal to `lineage$livings[lineage$drawn]`);
- `lineage$n`: a single `"integer"`: the current number of non extinct lineage (is equal to `length(lineage$livings)`);
- `lineage$split`: a `"logical"` vector: the list of splits for each lineage (`TRUE`), the number of total tips is equal to `sum(!lineage$split)`.

In general, unless you know what you're doing, you can ignore most arguments for specific modifiers since they are handled automatically within the `dads` function. Therefore any argument can be left undeclared or missing and is always handled internally. For example, if you did not declare `lineage$n` as a function argument but are using `lineage$n` in the function, `lineage$n` will be detected and treated as a current argument automatically as set accordingly within the birth death process (e.g. `lineage$n` will be set to the current number of taxa every iteration of the process).

For example, we can create a function that increases branch length proportional to the number of species "alive" at each time of the simulation in a discrete way. I.e. for discrete numbers of taxa, the branch length increases by jumps (ten fold) every 5 taxa:

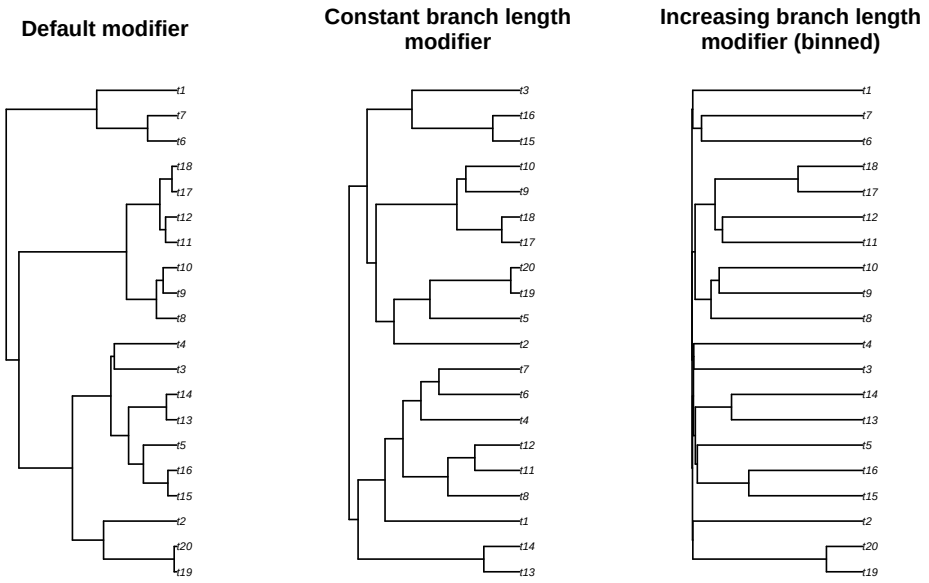
```
## A more complex binned.branch.length function
increasing.brlen <- function(bd.params, lineage) {

    ## Setting the cumulated birth and death
    birth_death <- bd.params$speciation + bd.params$extinction

    ## Returning branch lengths depending on different number of taxa
    if(lineage$n <= 5) {
        return(1    * rexp(1, sum(5 * birth_death)))
    }
    if(lineage$n <= 10) {
        return(10   * rexp(1, sum(10 * birth_death)))
    }
    if(lineage$n <= 15) {
        return(100  * rexp(1, sum(15 * birth_death)))
    }
    if(lineage$n <= 20) {
        return(1000 * rexp(1, sum(20 * birth_death)))
    } else {
        return(1000 * rexp(1, sum(lineage$n * birth_death)))
    }
}
```

We can then create it as a `"modifiers"` object and run a new simulation:

```
## Creating a modifiers
increasing_modifier <- make.modifiers(branch.length = increasing.brlen)

## Generating a new tree with this modifier
set.seed(0)
increasing_tree <- dads(bd.params = bd_params,
                        stop.rule = stop_rule,
                        modifiers = increasing_modifier)
```

And we can visualise the difference between the resulting trees:

```
par(mfrow = c(1,3))
plot(default_tree,    main = "Default modifier")
plot(modified_tree,   main = "Constant branch length\nmodifier")
plot(increasing_tree, main = "Increasing branch length\nmodifier (binned)")
```



4.3 The selection function (`selection`)

The `selection` function is used in the birth death process to know which lineage to select when running a speciation (or extinction!) event. By default, this function randomly selects one taxon that is currently not extinct (using: `sample(1:lineage$n, 1)`). Similarly as `branch.length` it is possible to modify this part of the birth death process. For example, we could simply select always the last created lineage (to create a "ladder" or most asymmetric tree):

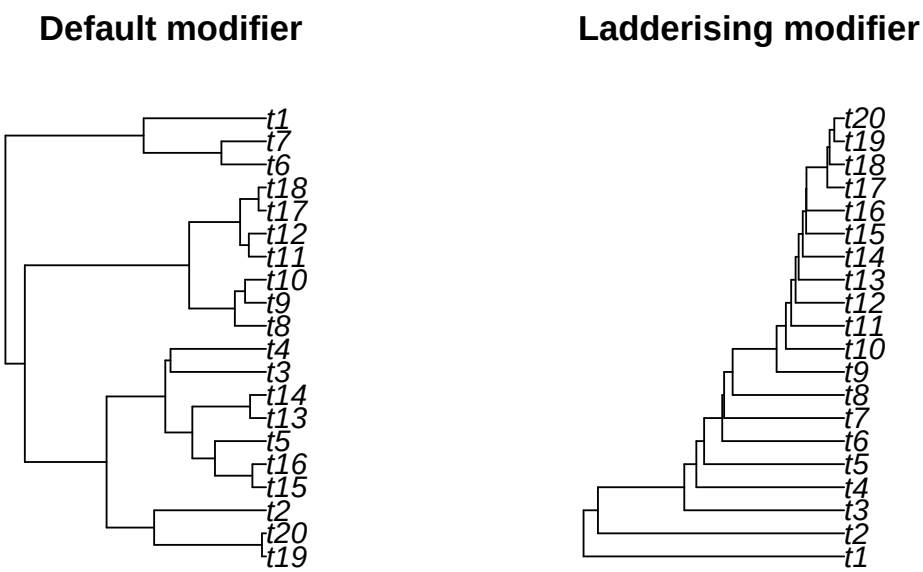
```
## Our function to always select the last taxon
## (making sure it returns an integer)
```

```
select.last <- function(lineage) {
    return(as.integer(lineage$n))
}
```

> Note that here the function can only intake the allowed arguments as described above (here `lineage$n`: the number of current living taxa).

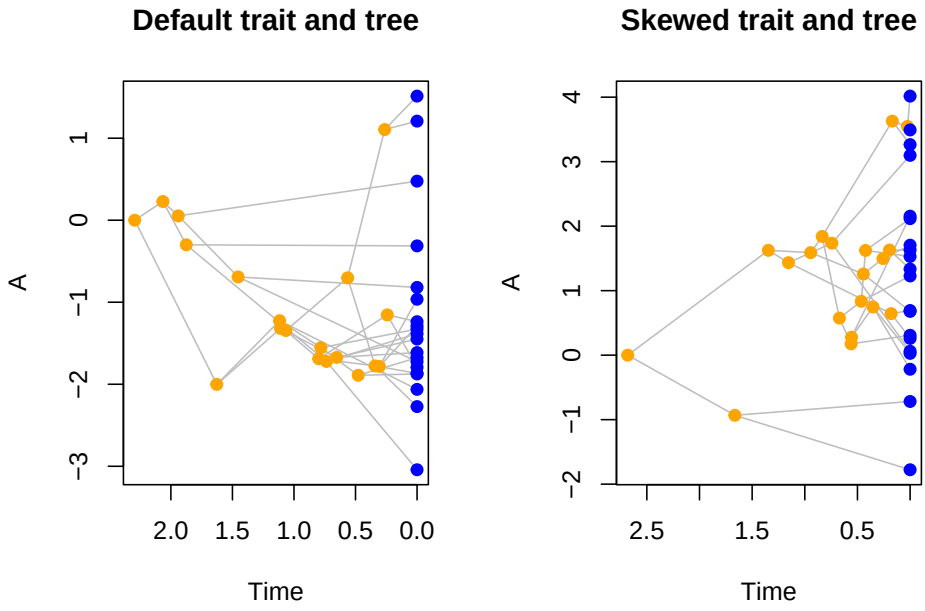
We can then create a `"modifiers"` object the same way as before using this time the `selection` argument:

```
## A modifier for selection
ladderised_modifier <- make.modifiers(selection = select.last)
## Generating a new tree with this modifier
set.seed(0)
ladderised_tree <- dads(bd.params = bd_params,
                        stop.rule = stop_rule,
                        modifiers = ladderised_modifier)
## Displaying the results
par(mfrow = c(1,2))
plot(default_tree,    main = "Default modifier")
plot(ladderised_tree, main = "Ladderising modifier")
```



Again, it is of course possible to make the modifier more complex and in combination with other elements of the tree. For example, we can create a `"dads"` object that also generates a BM trait and add to it this object a `selection` modifier that only selects tips with positive trait values (only species with positive trait values will speciate).

```r
## Our function that only select taxa with positive trait values
select.positive <- function(trait.values, lineage) {

    ## Selecting the taxa names with positive values for the first trait
    positives <- as.integer(rownames(trait.values)[which(trait.values[, 1] >= 0)])

    ## Combine the descendants of the current lineages (lineage$parents)
    ## with the species that have speciated (seq_along(lineages$split))
    ## to have a table of pairs of parents/splits
    parents_split_table <- cbind(lineage$parents, seq_along(lineage$split))
    ## Select the current taxa that descend from a node with a positive value
    positive_living <- parents_split_table[which(lineage$parents %in% positives), 2]

    ## Select one tip randomly in the ones with descendants with positive values
    return(sample(which(lineage$livings %in% positive_living), 1))
}

## Creating the modifier
positive_skew <- make.modifiers(selection = select.positive)

## Creating a (default) trait object
BM_trait <- make.traits()

## Simulate a tree and trait with no modifier
set.seed(1)
default_dads <- dads(bd.params = bd_params,
                     stop.rule = stop_rule,
                     traits    = BM_trait)

## Simulate a tree and trait with the modifier
set.seed(1)
skewed_trait_dads <- dads(bd.params = bd_params,
                          stop.rule = stop_rule,
                          traits    = BM_trait,
                          modifiers = positive_skew)

## Plotting the differences in trees and traits
par(mfrow = c(1, 2))
plot(default_dads, main = "Default trait and tree")
plot(skewed_trait_dads, main = "Skewed trait and tree")
```



4.4 The speciation function (`speciation`)

The third function that can be used to modify the birth death process is the `speciation` function. This one is used during the birth death process to decide whether a lineages speciates (creating a node and two new lineages) or goes extinct (creating a tip).

> Note that the `speciation` function only affects tips or nodes before the simulation reaches the `stop.rule`. The then surviving lineages are all automatically transformed into tips.

By default, the `speciation` function is trigger a speciation even if a number randomly drawn from a uniform distribution is lower than the ratio between the speciation and the speciation and extinction parameter. If the randomly drawn number is higher, the lineage goes extinct.

```
## The speciation in pseudo-code:
runif(1) < speciation/ (speciation + extinction)
```

Creating a `"modifiers"` with a `speciation` function works the same way as for `branch.length` and `selection` but the function that will be used needs to output a logical value (see table below). Once the function is created simply input your function for speciation in the modifier and run the `dads` function with that modifier:

```
## Speciating or going extinct randomly
## (regardless of the extinction parameter)
```

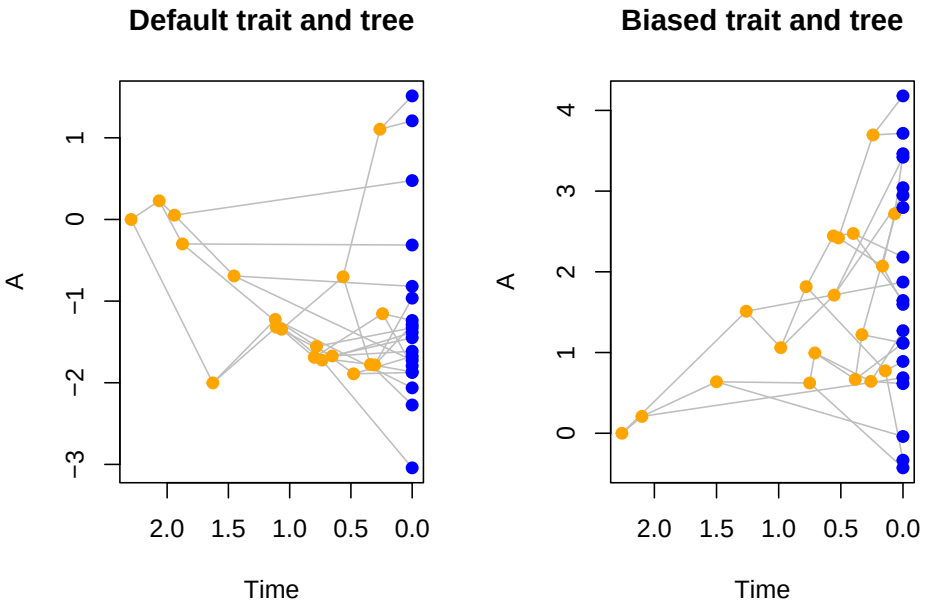
```
random.extinct  <- function() {
    return(sample(c(TRUE, FALSE), 1))
}

## Creating the modifiers object
random_extinction <- make.modifiers(speciation = random.extinct)

## Generating a new tree with this modifier
set.seed(2)
modified_tree <- dads(bd.params = bd_params,
                      stop.rule = stop_rule,
                      modifiers = random_extinction)

par(mfrow = c(1,2))
plot(default_tree,  main = "Default modifier")
plot(modified_tree, main = "Random extinction\nmodifier")
```

Note how loads of lineages end up going extinct even if the extinction parameter is set to 0!

And again, we can make some more advanced modifiers: for example, one where a tip always goes extinct if their ancestor has a negative trait value (here we will also introduce the utility function `parent.trait` that automatically selects the trait values of the parent of the current lineage.

```
## A modifier for removing tips with negative values
bias.trait <- function(trait.values, lineage) {
    if(parent.traits(trait.values, lineage) < 0) {
        ## Go extinct!
        return(FALSE)
    } else {
        ## Speciate
        return(TRUE)
    }
}

## Creating the modifier
biased_trait <- make.modifiers(speciation = bias.trait)

## Simulate a tree and trait with the modifier
set.seed(1)
biased_trait_dads <- dads(bd.params = bd_params,
                          stop.rule = stop_rule,
                          traits    = BM_trait,
                          modifiers = biased_trait)
```

```
## Plotting the differences in trees and traits
par(mfrow = c(1, 2))
plot(default_dads, main = "Default trait and tree")
plot(biased_trait_dads, main = "Biased trait and tree")
```



4.5 Summary of the inputs and outputs for the `branch.length`, `selection` and `speciation` modifiers

modifier name	accepted input (arguments)	required output (class)
`branch.length`	`bd.params`, `lineage`, `trait.values`	`"numeric"`
`selection`	`bd.params`, `lineage`, `trait.values`	`"integer"`
`speciation`	`bd.params`, `lineage`, `trait.values`	`"logical"`

4.6 The condition and modify functions (`condition` and `modify`)

In the examples above, we have seen how to specify modifications to the birth death process (via `branch.length`, `selection` and `speciation`), however, one might note that these modifications are not dynamic. In other words, throughout the process, the modifications remain constant (even if they are conditional).

It is however possible to code the `"modifiers"` so that they can be affected by `"events"` objects (see next chapter on `events`).

To do so, you can formally declare conditions (`condition`) and modifications (`modify`) as internal functions that can then be modified my an `"events"` object. `condition` and `modify` are hard coded in the `branch.length` function that they concern, i.e. they are variables (functions) within the function.

For example in the `speciation` part of a modifier, the default is to trigger an event from a uniform distribution and then check if that value is smaller than (

$$speciation/(speciation + extinction)$$

):

```
## The default speciation algorithm
speciation <- function(bd.params) {
    ## Randomly trigger an event
    trigger_event <- runif(1)

    ## Speciate?
    return(trigger_event < (bd.params$speciation/(bd.params$speciation + bd.params$extinction)))
}
```

It is possible with some specific condition to modify this trigger by providing a `condition` and `modify` function to the speciation function. That is, if the `condition` is met, apply the `modify` function to the algorithm. For example here we can edit the default speciation function to have a modification (`modify = double.the.trigger`: doubling the trigger value) happening half the time (`condition = half.the.time`):

```
## A conditional function that triggers half the time
half.the.time <- function() return(sample(c(TRUE, FALSE), 1))
## A modification that doubles the value to trigger the event
double.the.trigger <- function(x) return(x*2)
## A conditional modifier
make.modifiers(speciation = speciation,
               condition  = half.the.time,
               modify     = double.the.trigger)
```

```
##  ---- dads modifiers object ----
## Default branch length process.
## Default selection process.
## Speciation process is set to speciation with a condition (half.the.time) and a modifier (doubl
```

Effectively, this will internally modify the speciation function as follows

```
## The default speciation algorithm
speciation <- function(bd.params) {
    ## Randomly trigger an event
```

```
    trigger_event <- runif(1)

    ## Modify the triggering
    if(half.the.time()) { ## This running the half.the.time function drawing TRUE or F
        trigger_event <- double.the.trigger(trigger_event) ## This will double the ran
    }

    ## Speciate?
    return(trigger_event < (bd.params$speciation/(bd.params$speciation + bd.params$ext
}
```

These `condition` and `modify` functions can be applied to all the `modifiers` elements (`selection`, `branch.length` and `speciation`). However, they are typically used in `events` that will modify… the `modifiers` (see the `events` section)!

```
## This build of rgl does not include OpenGL functions.  Use
##  rglwidget() to display results, e.g. via options(rgl.printRglwidget = TRUE).
```

Chapter 5

Adding events to simulations

One other major feature of the `dads` package is to allow simulations to run with specified events to occur during the simulation. These are typically events that can drastically change the course of the simulation. For example, you might want simulate some mass extinction at some specific point in time. However, these can also be more subtle, like the internal change of parameter values when reaching a specific trait value. These are all handled by `dads` with the `events` object that you can create using `make.events` and has the same overall logic as `make.traits` and `make.modifiers`.

`events` require three main arguments: * the `target` which designates what the extinction should affect (e.g. the taxa, the speciation rate, etc…); * the `condition` which designates when to trigger the event; * the `modification` which designates what to exactly modify when the event is triggered.

There are several more arguments that can be passed to `make.events` but they are discussed later on. First let's focus on these three main arguments:

5.1 Target

The target of the event is what the event is going to modify in the birth-death algorithm. You can only have one target per event (along with one condition and one modification) but you can create events that contain multiple events (i.e. multiple triplets of target/condition/modification). The targets that are currently available are: * `"taxa"` to modify anything linked to the `lineage` list. E.g. making half of the living taxa go extinct. * `"bd.params"` to modify anything linked to the `bd.params` object. For example you might want to change the distribution of one of the parameter after some `conditions`. This is typically

done by updating the object using the argument `update` from `make.bd.params`. * `"traits"` to modify anything linked to the `traits` object. For example you might want to change the trait process after some `conditions`. This is typically done by updating the object using the argument `update` from `make.traits`. * `"modifiers"` to modify anything linked to the `modifiers` object. For example you might want to change the speciation rule after some `conditions`. This is typically done by updating the object using the argument `update` from `make.modifiers`. * `"founding"` this target is a bit more special and allows you run a nested `dads` object in the simulation. It is cover in a specific section below.

We will see some example of these targets by illustrating the `conditions` and `modifications` below.

5.2 Conditions

`condition` is a function that returns a logical value. When a specific condition is met, it should return `TRUE` and trigger the event, else it should return `FALSE`.

Currently there are three conditions functions implemented in `dads` but you can easily come up with your own version of them.

All `condition` functions in `dads` take at least to arguments: `x` for the variable of interest (e.g. time, number of taxa, trait value, etc…) and `condition`, the relational operator to evaluate. A relational operator is the proper (fancy) term designating all the comparisons you're regularly using in `R` like `==` (is equal?) `<` (is smaller?) `>=` (is bigger or equal?). You can get the full list in the base manual (using `?Comparison` or any of the relational operator in a function form).

> Note on the function form of a operator in `R`: if you weren't aware, functions in `R` are *usually* described in the format `function(arguments)`, however many other functions you are commonly using are written as operators, for example `(1 + 1)`. These are still functions though! And you can always use them in the classic function format by quoting them in back ticks: the following does the same as `(1+1)`: `` `+`(1, 1) ``. You can use that for parsing your relational operators as functions in `condition`. If you want you condition to be equal, use the syntax `condition = `==``. More details on how functions *really* work in the Advanced `R` book!

- `time.condition` will trigger the `event` once a certain time is reached in the simulations and is the simplest/more basic condition with no arguments other than the time required. For example `time.condition(4, condition = `==`)` will trigger the `event` once the simulations reached 4 units of time. Easy.
- `taxa.condition` will trigger the `event` once a certain number of taxa is reached. This can be considered including or excluding fossil species. For

example `taxa.condition(42, condition = `>=`, living = TRUE)` will trigger the `event` if there is at least 42 living species.
- `trait.condition` will trigger the `event` once a certain trait value is reached. This function allows to say which trait(s) to target (by default, the first one using `trait = 1`), what value of the trait to target (by default `what = max`) and whether to use and absolute trait value or not (`absolute = TRUE`). For example `trait.condition(1/3, condition = `>`, trait = 1, what = sd)` will trigger the condition after the standard deviation of the first trait reaches 1/3.

5.3 Modifications

After defining the `event`'s `target` and `condition`, you'll need to also specify what it should modify. These are basically functions that should modify a specific aspect of the target Depending on the target you can modify the following:

- if the target is `"taxa"` you can modify the internal `lineage` list by removing living species using `random.extinction` or `trait.extinction`.
- if the target is `"taxa"` you can modify the lineage tracker by removing living species using `random.extinction` or `trait.extinction`.
- if the target is `"bd.params"` you can modify the `bd.params` object using `update.bd.params`.
- if the target is `"traits"` you can modify the `traits` object using `update.traits`.
- if the target is `"modifiers"` you can modify the `modifiers` object using `update.modifiers`.
- (for the `"founding"` target see below)

The most straightforward example if for `modifications` on `bd.params`, `traits` or `modifiers` objects since they use the same syntax as for their generic `make.X` function. For example, for `make.traits`, you can update a `trait` using the `update` argument as follows:

```
## A BM trait in two dimensions
(BM_2D <- make.traits(n = 2, process = BM.process))
```

```
##  ---- dads traits object ----
## 2 traits for 1 process (A:2) with one starting value (0).
```

```
## Updating the 2D BM into a 2D OU
(OU_2D <- make.traits(update = BM_2D, process = OU.process))
```

```
##  ---- dads traits object ----
## 2 traits for 1 process (A:2) with one starting value (0).
```

So basically the functions `update.X` apply the update to the object `X` when the event happens. The following `event` updates the `bd.params` object by setting the extinction parameter to 1/3 when reaching a 10 species:

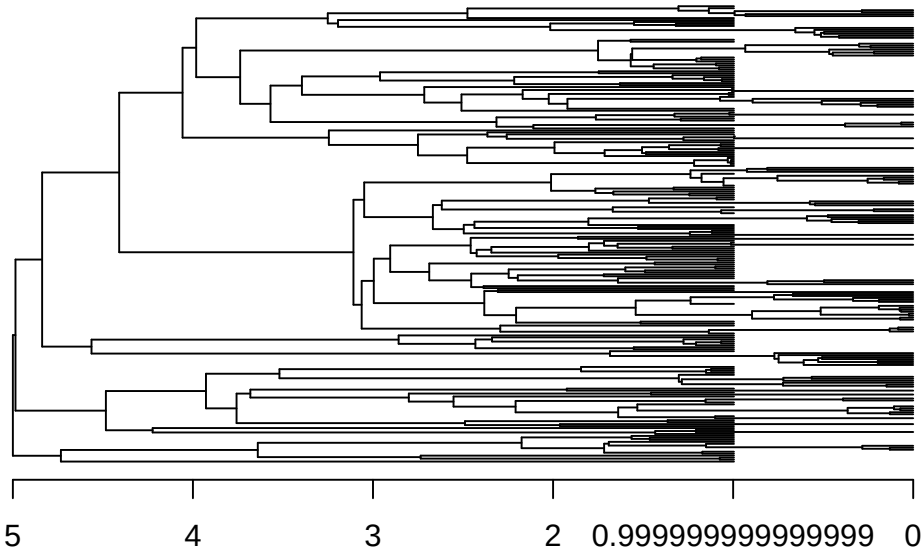
```
make.events(target       = "bd.params",
            condition    = taxa.condition(10, condition = `>=`),
            modification = update.bd.params(extinction = 1/3))
```

Note that by default events are triggered only once in the whole simulation, so although the example above states that the condition is reaching at least 10 taxa, it will not trigger every times it reaches more than 10 taxa. You can change the number of times the events can be triggered using the argument `replications` (by default it's set to `replications = 0` for triggering it only once).

5.4 Examples

Here are some example illustrating how to generate events. For a simple example, we can create a extinction event that will remove 80% of species after reaching time 4:

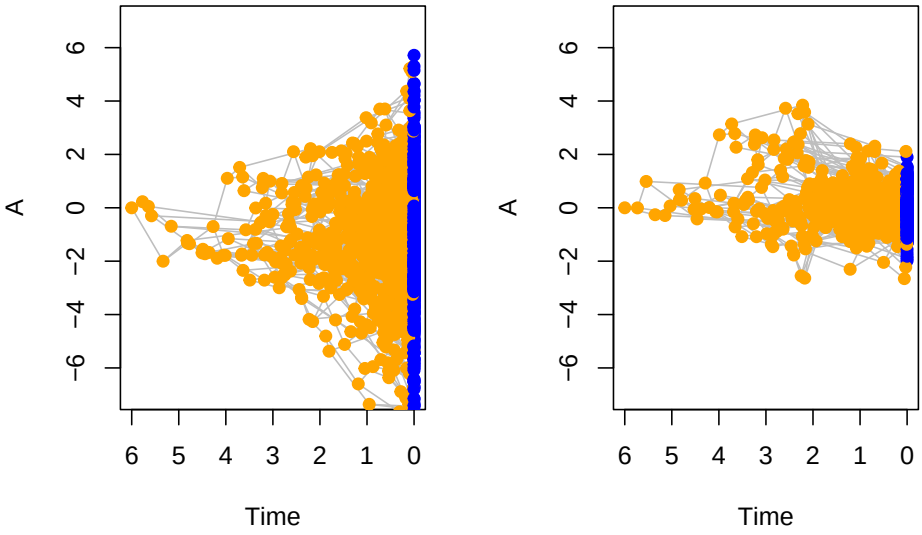
```
## 80% mass extinction at time 4
mass_extinction <- make.events(
                      target       = "taxa",
                      condition    = time.condition(4),
                      modification = random.extinction(0.8))

## Simulation parameters
stop.rule <- list(max.time = 5)
bd.params <- list(extinction = 0, speciation = 1)

## Running the simulations
set.seed(123)
results <- dads(bd.params = bd.params,
                stop.rule = stop.rule,
                events    = mass_extinction)
## Plotting the results
plot(results, show.tip.label = FALSE)
axisPhylo()
```



Or for a slightly more complex example, we can can change the trait process from a BM to an OU when the trait values reaches an upper 95% quantile value above 2:

```
## The 95% upper quantile value of a distribution
upper.95 <- function(x) {
    return(quantile(x, prob = 0.95))
}
## Create an event to change the trait process
change_process <- make.events(
                  target       = "traits",
                  ## condition is triggered if(upper.95(x) > 3)
                  condition    = trait.condition(3, condition = `>`, what = upper.95),
                  modification = update.traits(process = OU.process))

## Set the simulation parameters
bd.params <- list(extinction = 0, speciation = 1)
stop.rule <- list(max.time = 6)
traits    <- make.traits()

## Run the simulations
set.seed(1)
no_change <- dads(bd.params = bd.params,
                  stop.rule = stop.rule,
                  traits    = traits)
set.seed(1)
process_change <- dads(bd.params = bd.params,
                       stop.rule = stop.rule,
```

```
                        traits    = traits,
                        events    = change_process)
## Plot the results
par(mfrow = c(1,2))
plot(no_change, ylim = c(-7, 7))
plot(process_change, ylim = c(-7, 7))
```



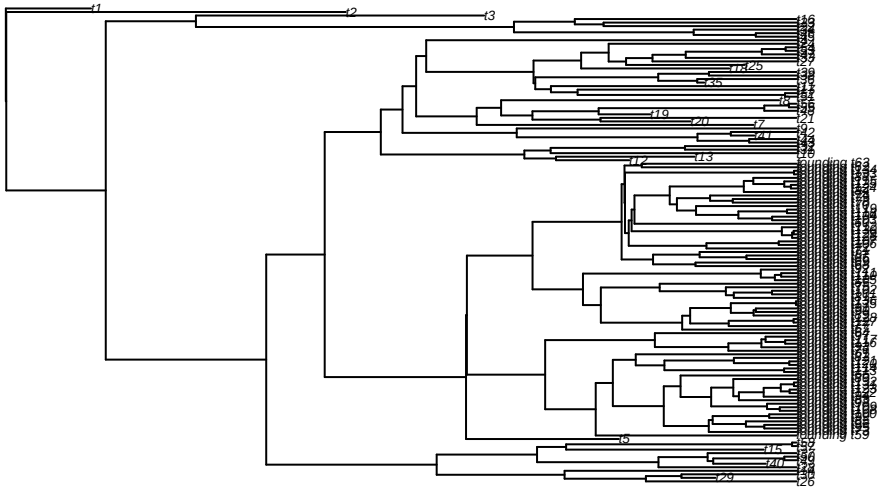
5.5 Founding events

Founding events are a specific `target` for `events` that allows you to… simulate a birth-death process within the current one! It basically allows you to simulate a specific `dads` process i.e. using the `dads` function with it's own `traits`, `modifiers`, `bd.params` and `events` (and yes, that's including `events` that have their own `founding` event)… This is basically the ultimate nested boss of modularity!

The founding event will basically run an internal `dads` process (i.e. simulating a tree and, optionally, some data) resulting in a founding sub-tree. It will then branch this sub-tree to the rest of the simulation that continued normally in the mean time. You can specify the founding event using the inbuilt `founding.event` modification event. This function intakes the exact same arguments as `dads` to simulate the sub-tree with its own parameters. Additionally, we will use the `additional.args` argument from `make.events` to specify a prefix for the founding tree tips (to make them easier to distinguish).

```
## Set up parameters
stop.rule <- list(max.time = 4)
bd.params <- make.bd.params(speciation = 1, extinction = 0.3)
```

```
## Events that generate a new process (founding tree - with no extinction)
founding_event <- make.events(
                  target       = "founding",
                  condition    = taxa.condition(10),
                  modification = founding.event(
                                    bd.params = make.bd.params(speciation = 2,
                                                               extinction = 0)),
                  additional.args = list(prefix = "founding_"))

## Simulations
set.seed(11)
founding_tree <- dads(bd.params = bd.params,
                      stop.rule = stop.rule,
                      events    = founding_event)
plot(founding_tree, cex = 0.4)
```



Note that the nestedness here is potentially endless, for example, you can parse an `event` argument to the `founding.event` that will generate another founding event, etc…

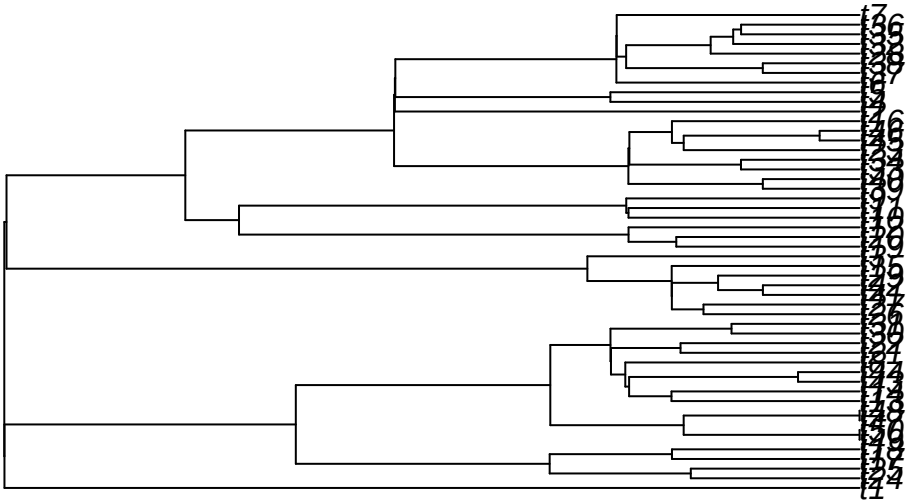
Chapter 6

Other functionalities

6.1 `make.bd.params`

In most examples above, birth-death parameters where set as fixed values, i.e. a speciation rate of λ and an extinction rate of μ throughout the simulations (with maybe some events modifying these rates). However, it is of course possible to set these rates as specific or changing distributions! You can do this using the `make.bd.params` function a provide either a `vector` of values:

```
## An example where the speciation is randomly sampled among three values
my_bd_params <- make.bd.params(speciation = c(1/3,42))

## Building a tree using this set of parameters
set.seed(123)
plot(dads(stop.rule = list(max.taxa = 50), bd.params = my_bd_params))
```



```
## Note the regions in the tree with short branches
## (that's the speciation being 1/3 while the others are speciation = 42)
```

Or directly a function from which to sample:

```
## Another example where speciation is drawn from the interval (0, 1)
make.bd.params(speciation = runif)
```

```
##  ---- dads birth-death parameters object ----
## speciation: runif.
## extinction: 0.
```

In this example, the `"bd.params"` object passed to the `dads` function will allow the birth-death process to sample the speciation parameter each time it is called (e.g. during the speciation/extinction step, the branch length step, etc.). If using a function, you can fine tune the arguments to be passed to that function using the `speciation.args` or the `extinction.args` arguments (as a named list matching the function's arguments):

```
## Speciation is drawn from the interval (0.5, 1.5)
make.bd.params(speciation = runif,
               speciation.args = list(min = 0.5, max = 1.5))
```

```
##  ---- dads birth-death parameters object ----
## speciation: runif (with optional arguments).
## extinction: 0.
```

When using distributions for both the speciation and extinction parameters, you can run into the undesired problem of having an extinction rate that is higher than your speciation rate (and thus your tree dying out - unless this is the desired behaviour). You can avoid this problem by using the `joint` distribution argument this will ensure that the sampling of the extinction rate is always

lower or equal to the speciation rate.

```
## Joint speciation and extinction sampled from uniform distribution
## with speciation always >= to extinction
make.bd.params(speciation = runif, extinction = runif, joint = TRUE)
```

```
##  ---- dads birth-death parameters object ----
## joint sampling for:
## speciation: runif.
## extinction: runif.
```

Finally to avoid negative sampling values, you can use the argument `absolute` that will make all the sampled values positive. This argument is set to `TRUE` by default so you shouldn't have to worry about it most of the time unless you specifically need negative sampled values for your parameters.

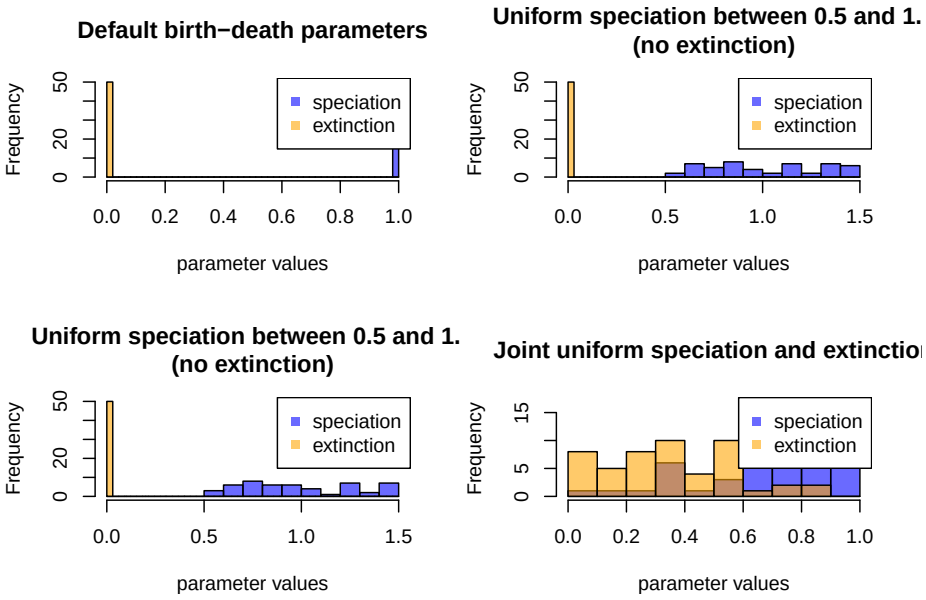
```
## Making the speciation sampling always positive
make.bd.params(speciation = rnorm, absolute = TRUE)
```

```
##  ---- dads birth-death parameters object ----
## speciation: rnorm.
## extinction: 0.
## (using absolute values)
```

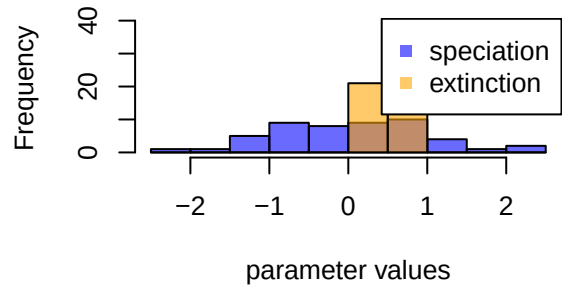
The two other possible arguments for this function are `test` and `update` that work the same as for `make.traits` or `make.modifiers`

If you are a visual person and your `bd.params` objects are getting a bit too complicated to remember, you can always quickly plot them (the function will sample from the `bd.params` object and show the results):

```
par(mfrow = c(2,2))
plot(make.bd.params(), main = "Default birth-death parameters")
plot(make.bd.params(speciation = runif,
               speciation.args = list(min = 0.5, max = 1.5)),
     main = "Uniform speciation between 0.5 and 1.5\n(no extinction)")
plot(make.bd.params(speciation = runif,
                    speciation.args = list(min = 0.5, max = 1.5)),
     main = "Uniform speciation between 0.5 and 1.5\n(no extinction)")
plot(make.bd.params(speciation = runif, extinction = runif, joint = TRUE),
     main = "Joint uniform speciation and extinction")
```




```
plot(make.bd.params(speciation = rnorm, extinction = runif,
                    joint = FALSE, abs = FALSE),
     main = "Disjoint normal speciation and uniform extinction")
```



6.2 `drop.things`

You can use the function `drop.things` to drop specific elements of the tree and data at the same time by providing the argument `what` to be `"fossils"` for tips that went extinct, `"livings"` tips that where alive at the end of the simulation or `"singles"` for internal nodes. Alternatively you can use the function aliases `drop.fossils`, `drop.livings` or `drop.singles` for the exact same results:

```
## A random tree with fossils and traits and internal nodes every 0.5 times
set.seed(3)
my_data <- dads(stop.rule = list(max.taxa = 20),
```

```
                bd.params = list(speciation = 1, extinction = 1/3),
                traits    = make.traits(), save.steps = 0.5)

## A tree with 20 tips and 54 nodes
my_data$tree
```

```
##
## Phylogenetic tree with 20 tips and 54 internal nodes.
##
## Tip labels:
##   t1, t2, t3, t4, t5, t6, ...
## Node labels:
##   n1, n2, n3, n4, n5, n6, ...
##
## Rooted; includes branch lengths.
```

```
## And a dataset with 74 rows
dim(my_data$data)
```

```
## [1] 74  1
```

```
## Removing the fossil species
drop.things(my_data, what = "fossils")$tree
```

```
##
## Phylogenetic tree with 8 tips and 31 internal nodes.
##
## Tip labels:
##   t13, t14, t15, t16, t17, t18, ...
## Node labels:
##   n1, n2, n7, n10, n11, n13, ...
##
## Rooted; includes branch lengths.
```

```
dim(drop.fossils(my_data)$data)
```

```
## [1] 39  1
```

```
## Removing the living species
drop.things(my_data, what = "livings")$tree
```

```
##
## Phylogenetic tree with 12 tips and 37 internal nodes.
##
## Tip labels:
##   t1, t2, t3, t4, t5, t6, ...
## Node labels:
##   n1, n2, n3, n4, n5, n6, ...
```

```
##
## Rooted; includes branch lengths.
```

```
dim(drop.livings(my_data)$data)
```

```
## [1] 49  1
```

```
## Removing the internal nodes
drop.things(my_data, what = "singles")$tree
```

```
##
## Phylogenetic tree with 20 tips and 19 internal nodes.
##
## Tip labels:
##   t1, t2, t3, t4, t5, t6, ...
## Node labels:
##   n1, n7, n33, n39, n54, n40, ...
##
## Rooted; includes branch lengths.
```

```
dim(drop.singles(my_data)$data)
```

```
## [1] 39  1
```

```
## Removing the internal nodes AND the fossils
drop.singles(drop.fossils(my_data))
```

```
##  ---- dads object ----
## Simulated diversity data (x$tree):
##
## Phylogenetic tree with 8 tips and 7 internal nodes.
##
## Tip labels:
##   t13, t14, t15, t16, t17, t18, ...
## Node labels:
##   n7, n33, n54, n53, n34, n45, ...
##
## Rooted; includes branch lengths.
##
## Simulated disparity data (x$data):
## 1 trait for 1 process (A) with one starting value (0).
```

6.3 "dads" internal utilities

The package also provides utilities for internal functions, namely for designing `modifiers` or `events` more easily. These functions don't do anything useful on their own but are optimised to be used internally in `dads`. For all these functions, you can look at the internal manual for an example (i.e. using

?<function_name>)

So far the package has the following internals:

Function	What it does	Where can it be used
parent.traits	selects the trait values of the current lineage's parents (i.e. direct ancestor)	in make.modifiers
taxa.condition	provides a trigger for an event dependent on the number of taxa	in make.events
time.condition	provides a trigger for an event dependent on time	in make.events
trait.condition	provides a trigger for an event dependent on trait values	in make.events

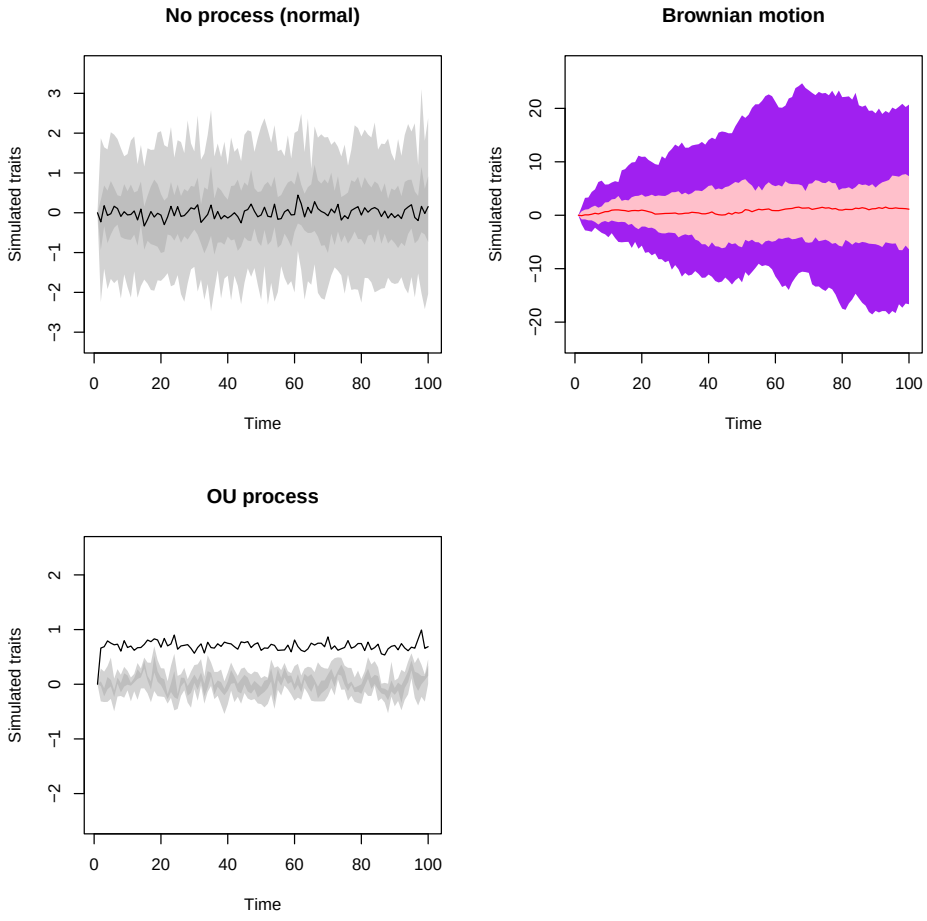
"dads" objects can be directly plotted in dads using the S3 plot.dads function (or just plot(x) if x is of class "dads").

Chapter 7

Plotting traits

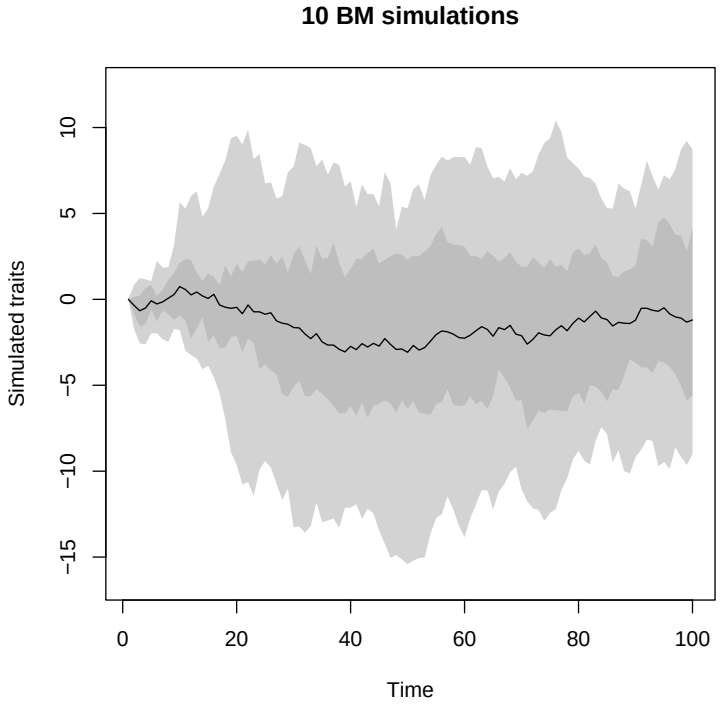
"dads" "traits" objects are covered in the traits section. You can use plot.dads to plot them by choosing which specific trait to plot using the trait argument (default is 1):

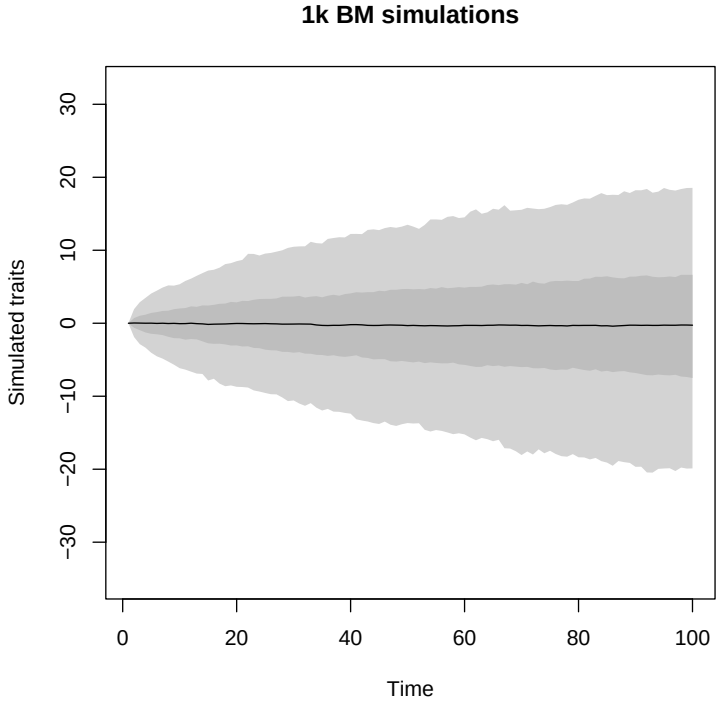
```
## Making a list of three traits
list_of_traits <- make.traits(process = c(no.process, BM.process, OU.process), trait.names = c("N
## Plotting each trait separately
par(mfrow = c(2, 2))
plot(list_of_traits, trait = 1)
## Using different colours options
plot(list_of_traits, trait = 2, col = c("red", "purple", "pink"))
## Not using the default plot name
plot(list_of_traits, trait = 3, main = "OU process",
     cent.tend = sd, quantiles = c(10, 30))
```



You can also control the number of replicates in the simulation by using the `simulations` option (the default is 50). Bigger numbers leads to more time but smoother looking plots while smaller ones are more stochastic:

```
par(mfrow = c(2,1))
plot(list_of_traits, trait = 2, simulations = 10, main = "10 BM simulations")
plot(list_of_traits, trait = 2, simulations = 1000, main = "1k BM simulations")
```


10 BM simulations
Simulated traits
10
5
0
−5
−10
−15
0
20
40
60
80
100
Time

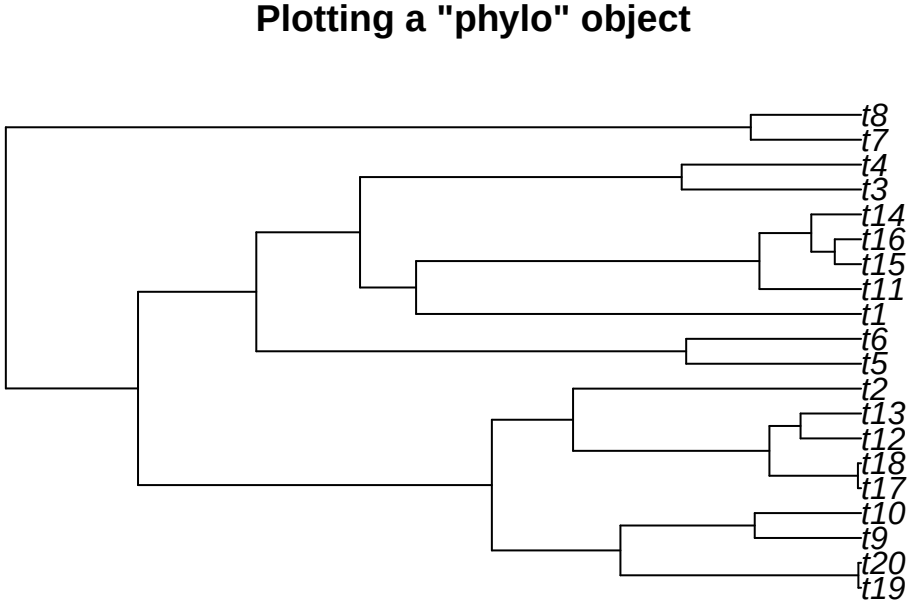

1k BM simulations
Simulated traits
30
20
10
0
−10
−20
−30
0
20
40
60
80
100
Time

Chapter 8

Plotting dads results

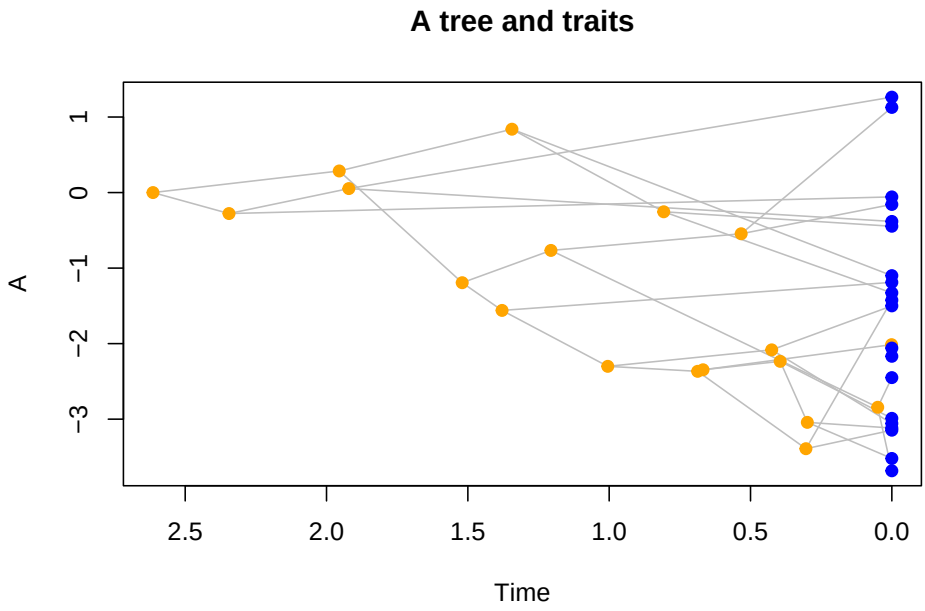
If `dads` is used to plot only a tree (and outputs a `"phylo"` object), you can use the function `plot.phylo` from the `ape` package to plot your tree. You'll find all the tree plotting option in the `?plot.phylo` manual page.

```
## A simple pure birth tree
my_tree <- dads(stop.rule = list(max.taxa = 20))
plot(my_tree, main = "Plotting a \"phylo\" object")
```



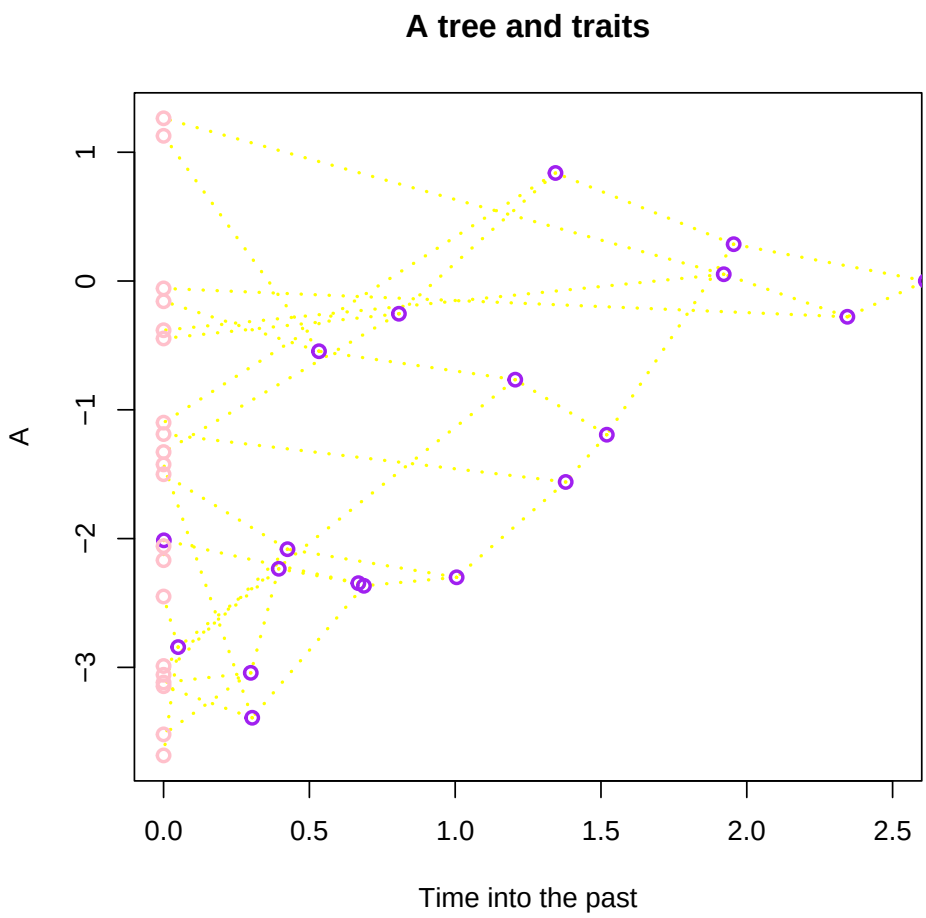
However, if you also simulated a trait along with the tree you can use the `plot.dads` function to plot both the tree and the trait:

```
## A simple pure birth tree with a BM process
my_tree <- dads(stop.rule = list(max.taxa = 20), traits = make.traits())
## Playing with the default options
plot(my_tree, main = "A tree and traits")
```



By default, elements are coloured as follows: nodes and tips are points and coloured in blue if they are tips (light blue if they are fossils) and nodes are in orange. Branches linking them are grey lines. You can of course change this colour palette to something of you preference by calling the normal arguments that can be passed to `points` or `lines`. For example `pch` for the point type or `lty` for the line type. Furthermore, time is plotted conventionally from left to right (left is towards the past, right is towards the present) but you can change that by specifying `xlim`.

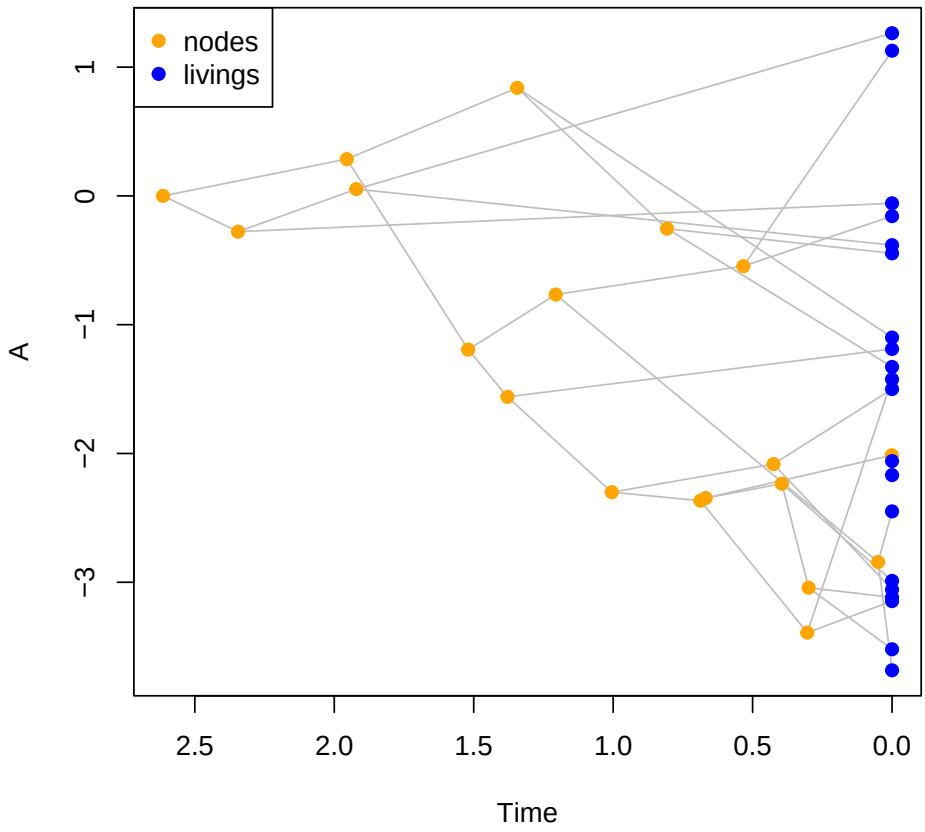
```
## Playing with some more options
plot(my_tree, main = "A tree and traits",
     ## Changing nodes colours and type
     col = c(tips = "pink", nodes = "purple"), pch = 21,
     ## Changing edges colour and type
     lwd = 2, lty = 3, edges = "yellow",
     ## Changing the x axis orientation and label
     xlim = c(0, 2.5), xlab = "Time into the past")
```



Note that you can modify colours using the `col` argument by providing a clear indication of what you want to colour (e.g. `col = c(tips = "blue", fossils = "orange")` will apply the colours to the unambiguously named elements) or by providing a function that will scale the colours with time (e.g. `col = grDevices::heat.colors` will use the specified function to change the elements colours with time).

And you can add a default legend by using `legend = TRUE` (if you don't want to add the default legend you can add it after your plot using `legend(...)`).

```
## Adding the default legend
plot(my_tree, legend = TRUE)
```



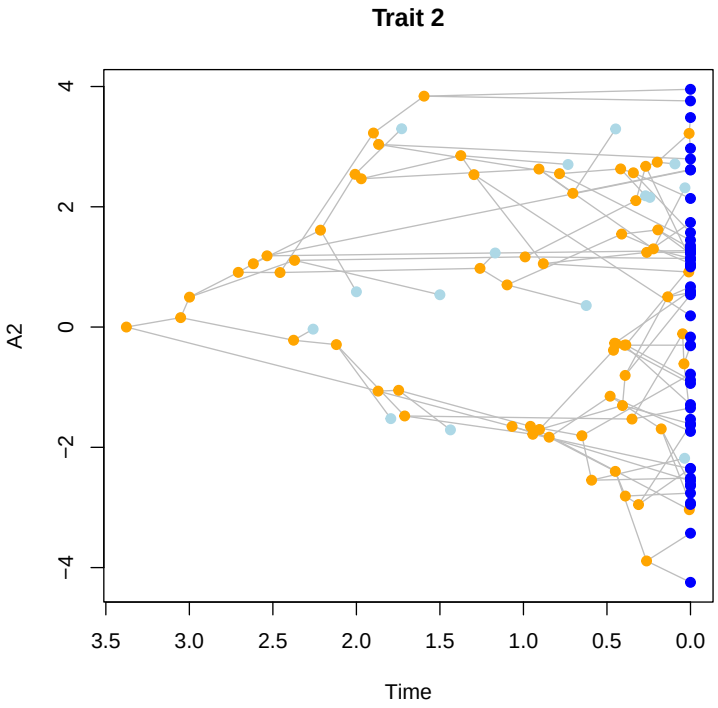
8.1 3D version!

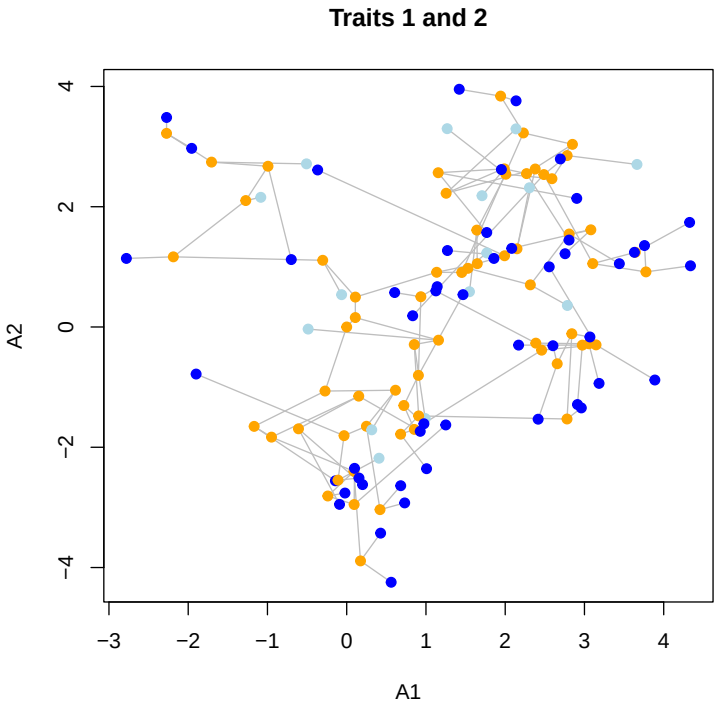
Of course, if you're simulating multiple traits, you can always plot different ones.

```
## Specifying a 3D trait process
my_3D_trait <- make.traits(n = 3, )
## Simulating a birth death tree with that 3D trait
my_data <- dads(bd.params = list(extinction = 0.2),
                stop.rule = list(max.living = 50),
                traits    = my_3D_trait)
```

You can toggle which trait to plot using the `trait` option. Either by providing a single value to plot that specific trait against time or by providing two traits.

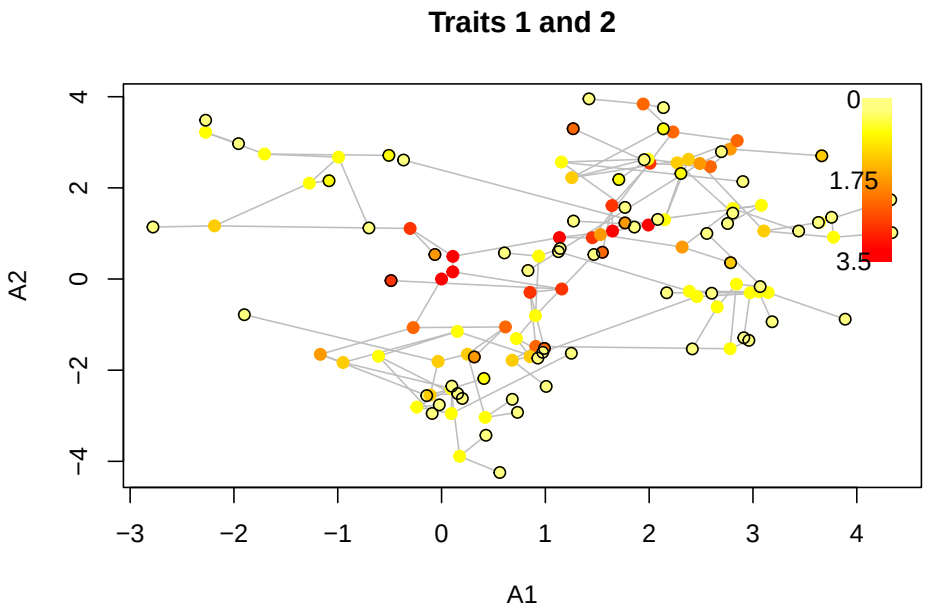
```
par(mfrow = c(2,1))
## Plotting the second trait
plot(my_data, trait = 2, main = "Trait 2")
## Plotting the correlation between trait 1 and 2
plot(my_data, trait = c(1,2), main = "Traits 1 and 2")
```


Trait 2
A2
4
2
0
−2
−4
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
Time


Traits 1 and 2
A2
4
2
0
−2
−4
−3
−2
−1
0
1
2
3
4
A1

As mentioned above, the `col` argument can take a function for scaling the elements colours with time. This can be useful for adding time a third dimension to these 2D plots:

```
## Plotting the correlation between trait 1 and 2
## with time as a 3rd dimensions
plot(my_data, trait = c(1,2), main = "Traits 1 and 2",
     col = grDevices::heat.colors, legend = TRUE,
     ## Highlighting the tips in black for visibility
     tips.nodes = "black")
```



But there's more! You can also plot these results using actual 3D plots that you can spin and all that! This is done through the `rgl` package and activated in `dads` using the option `use.3D = TRUE`.

If two traits are provided, the 3rd dimensions is going to be time by default:

```
## Plotting the tree and 2 traits in 3D: woah!
plot(my_data, trait = c(1, 2), use.3D = TRUE)
rglwidget()
```

```
## 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
```

If three traits are provide, you can set up a fourth coloured dimension as time (similarly to the 2D plots with the colour gradients). This uses the `rgl` functions, namely `lines3d`, `points3d` (if `type = "p"`; default) or `sphere3d` (if `type = "s"`). The additional arguments `...` are directly handled and attributed to the corresponding function.

```
## Plotting the tree and 3 traits in 3D
plot(my_data, trait = c(1, 2, 3), use.3D = TRUE,
     col = grDevices::heat.colors, type = "s", radius = 0.1)
rglwidget()
```

```
## 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
```

Chapter 9

Specific examples

In this section I will illustrate a series of examples of specific more complex scenarios. They can be all run independently and used as a basis for your own specific scenarios.

Here is a table summarising which functionalities are used in which example:

Functionality showcase	Example
`bd.params`	reducing speciation rate, generating a subtree with no extinction
`make.events`	mass extinction, negative trait extinction, background extinction, reducing speciation rate, changing trait process, changing trait correlation, changing modifiers, change branch length, generating a subtree with no extinction, generating a subtree with a different process
`make.traits`	negative trait extinction, changing trait process, changing trait correlation, generating a subtree with a different process
`make.modifiers`	changing modifiers, change branch length

And more specifically for the events:

Functionality showcase	Example
`time.condition`	mass extinction, negative trait extinction, reducing speciation rate, changing trait process, changing modifiers
`random.extinction`	mass extinction
`trait.extinction`	negative trait extinction
`taxa.condition`	background extinction, change branch length
`update.bd.params`	background extinction, reducing speciation rate
`update.traits`	changing trait process, changing trait correlation
`trait.condition`	changing trait correlation, generating a subtree with a different process
`update.modifiers`	changing modifiers, change branch length
`parent.traits`	changing modifiers
`founding.event`	generating a subtree with no extinction, generating a subtree with a different process

9.1 Random mass extinction after some time

For this scenario, we want to generate a pure birth tree (no traits and no extinction) where 80% of the species go extinct two thirds into the scenario.

For that we first need to first set our simulation parameters: we will be running the simulation for 6 time units and with a speciation rate of 1 (and extinction of 0 - default).

```
## Setting the parameters
stop_time_6 <- list(max.time = 6)
speciation_1 <- make.bd.params(speciation = 1)
```

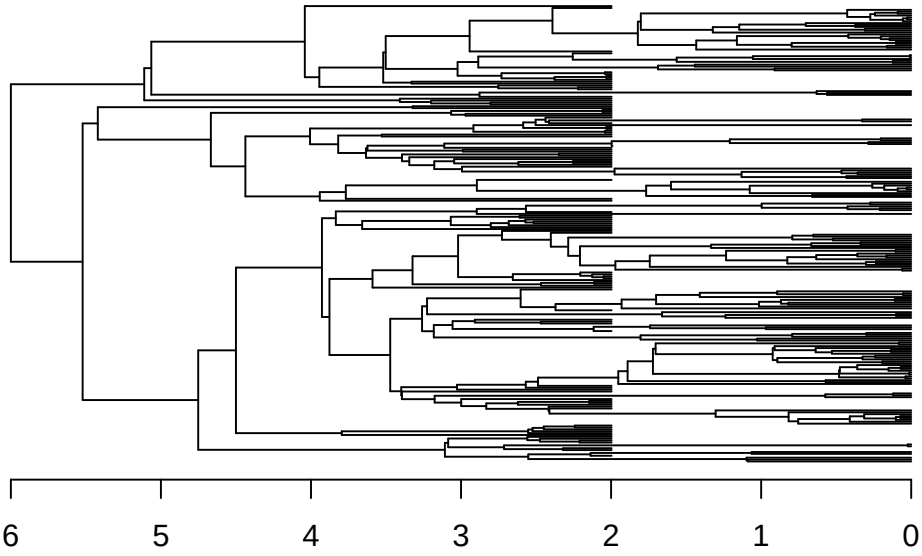
We will then create a event that triggers when reaching half of the simulation (using `time.condition(2.5)`). This event will target `"taxa"`, i.e. the number of species, and randomly remove 80% of them (using `random.extinction(0.8)`):

```
## 80% mass extinction at time 4
mass.extinction <- make.events(
                    condition = time.condition(4),
```

```
                    target = "taxa",
                    modification = random.extinction(0.8))
```

Once these parameters are defined, we can run the simulations and plot the results:

```
## Running the simulations
set.seed(1)
results <- dads(stop.rule = stop_time_6,
                bd.params = speciation_1,
                events    = mass.extinction)
## Plotting the results
plot(results, show.tip.label = FALSE)
axisPhylo()
```



9.2 Species with negative trait values go extinct after some time

For this scenario, we want to generate a pure birth tree with a one dimensional Brownian Motion trait for 5 time unit. We then want the species with a negative trait value go extinct after 4 time unit.

First we need to set up the simulation parameters: * The stopping rule (5 time units) * The birth-death parameters (speciation rate of 1)

```
## Simulation parameters
stop_time_5 <- list(max.time = 5)
speciation_1 <- make.bd.params(speciation = 1)
```

Then set up our trait which is a one dimensional Brownian Motion

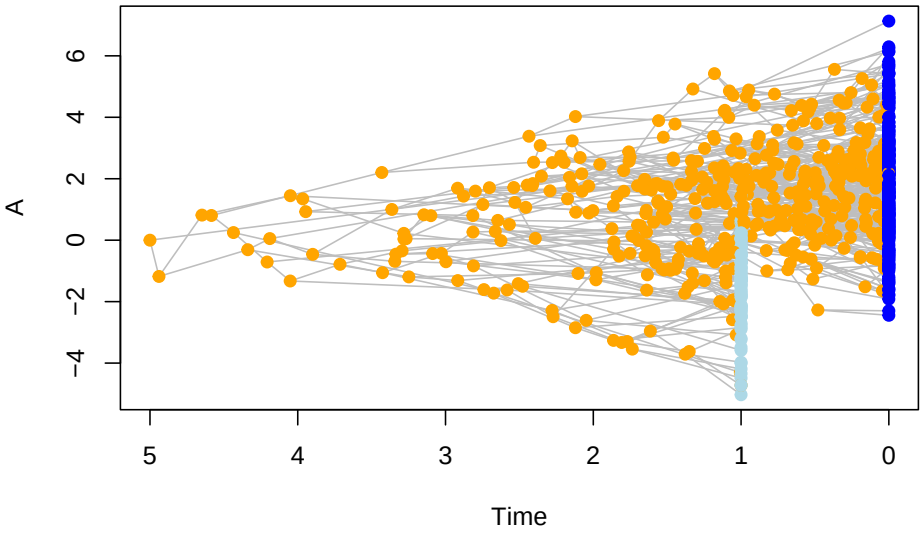
```
## Trait
simple_bm_trait <- make.traits(n = 1, process = BM.process)
```

And our extinction event which triggers after reaching time 4 (`time.condition(4)`), targets the `"taxa"` and modifies the extinction for species with traits lower than 0

```
## Extinction of any tips with trait < 1 at time 4
trait.extinction <- make.events(
                        target = "taxa",
                        condition = time.condition(4),
                        modification = trait.extinction(x = 0,
                                                        condition = `<`))
```

Once these parameters are defined, we can run the simulations and plot the results:

```
## Running the simulations
set.seed(7)
results <- dads(stop.rule = stop_time_5,
                bd.params = speciation_1,
                traits    = simple_bm_trait,
                events    = trait.extinction)
## Plotting the results
plot(results)
```



9.3 Adding a background extinction after reaching a number of living taxa

For this scenario, we want to generate a pure birth tree until reaching 50 living taxa but with an extinction rate appearing after reaching 30 taxa.

First we need to set up the simulation parameters: * The stopping rule (50 taxa max) * The birth-death parameters (speciation rate of 1)

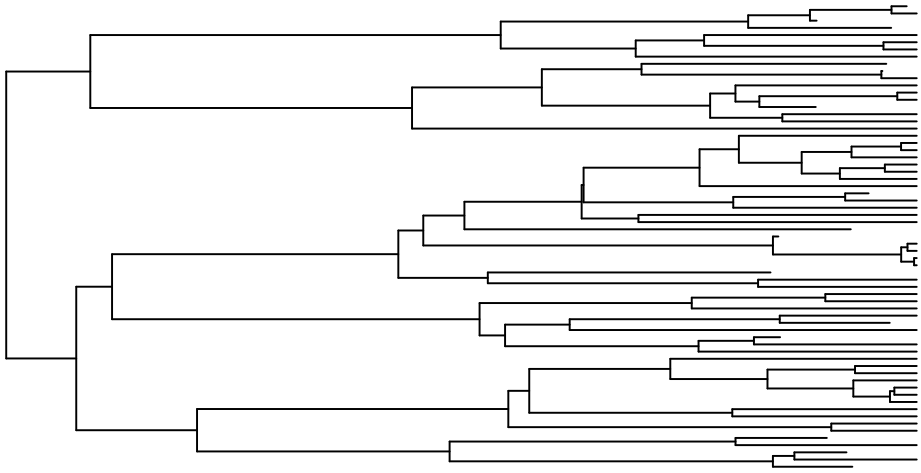
```
## Simulation parameters
stop_taxa_50 <- list(max.living = 50)
speciation_1 <- make.bd.params(speciation = 1)
```

And our change in extinction rate event which triggers after reaching 30 taxa (`taxa.condition(30)`), targets the `"bd.params"` (birth-death parameters) sets the extinction rate to 0.5 (`update.bd.params(extinction = 0.5)`):

```
## Adding an extinction parameter after 30 taxa
background.extinction <- make.events(
                      condition    = taxa.condition(30),
                      target       = "bd.params",
                      modification = update.bd.params(extinction = 0.5))
```

Once these parameters are defined, we can run the simulations and plot the results:

```
## Running the simulations
set.seed(2)
results <- dads(stop.rule = stop_taxa_50,
                bd.params = speciation_1,
                events    = background.extinction)
## Plotting the results
plot(results, show.tip.label = FALSE)
```



9.4 Reducing speciation rate after some time

For this scenario, we want to generate a brith tree for 6 times units with random speciation rates (i.e. drawn from a uniform (0.5;1) distribution) which reduces after time 4 through the simulations to a fixed value of 1/3.

First we need to set up the simulation parameters: * The stopping rule (6 time units) * The birth-death parameters (speciation randomly drawn between 0.5 and 1)

```
## Simulation parameters
stop_time_6 <- list(max.time = 6)
random_speciation <- make.bd.params(speciation = runif, speciation.args = list(min = 0
```

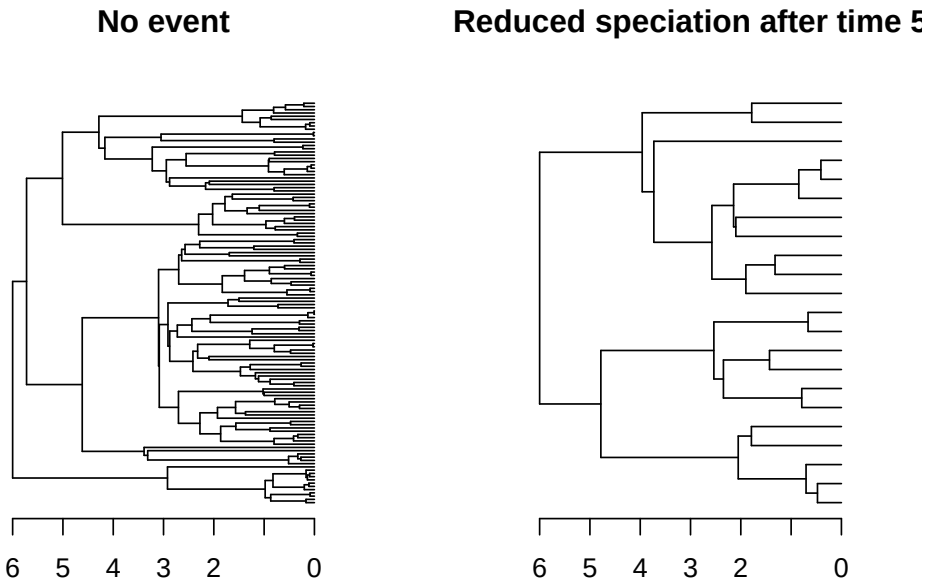
And our extinction event which triggers after reaching time 4 (`time.condition(4)`), targets the `"bd.params"` and modifies the speciation rate to 1/3:

```
## Reducing speciation after reaching time 4
reduced.speciation <- make.events(
                        condition    = time.condition(4),
                        target       = "bd.params",
                        modification = update.bd.params(speciation = 1/3))
```

Once these parameters are defined, we can run the simulations and plot the results. We can contrast the results with the scenario without an event (but same random seed):

```
## No event
set.seed(42)
no_event <- dads(stop.rule = stop_time_6,
                 bd.params = random_speciation)
```

```
## Reduced speciation event
set.seed(42)
reduced_speciation_event <- dads(stop.rule = stop_time_6,
                                 bd.params = random_speciation,
                                 events    = reduced.speciation)

## Plot both trees
par(mfrow = c(1, 2))
plot(no_event, main = "No event", show.tip.label = FALSE)
axisPhylo()
plot(reduced_speciation_event, main = "Reduced speciation after time 5", show.tip.label = FALSE)
axisPhylo()
```



9.5 Changing the trait process after some time

For this scenario, we want to generate a pure birth tree with a one dimensional Brownian Motion trait for 5 time unit which then changes to an OU process.

First we need to set up the simulation parameters: * The stopping rule (6 time units) * The birth-death parameters (speciation rate of 1)

```
## Simulation parameters
stop_time_6 <- list(max.time = 6)
speciation_1 <- make.bd.params(speciation = 1)
```

Then set up our trait which is a one dimensional Brownian Motion

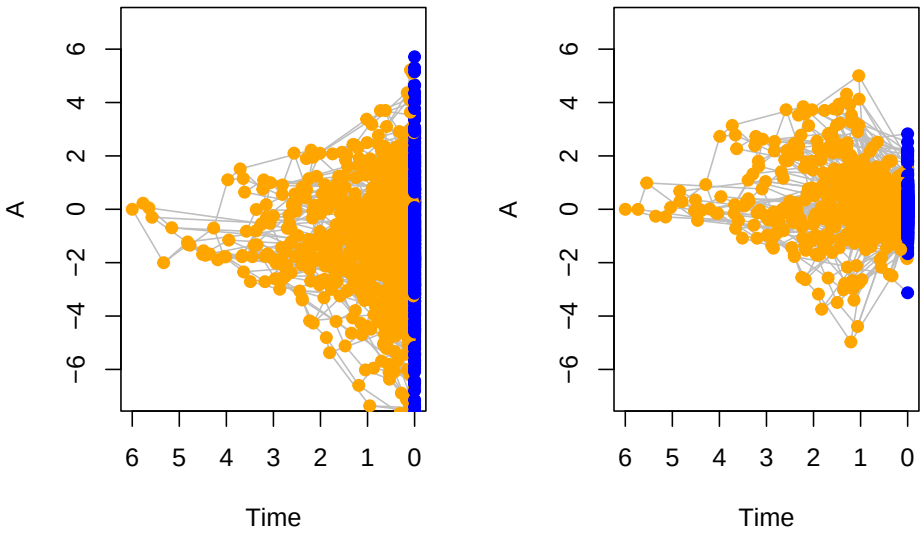
```
## Trait
simple_bm_trait <- make.traits(n = 1, process = BM.process)
```

And our event which triggers after reaching time 5 (`time.condition(5)`), targets the `"traits"` and modifies the process to `OU.process`.

```
## Create an event to change the trait process
change.process.to.OU <- make.events(
                  condition    = time.condition(5),
                  target       = "traits",
                  modification = update.traits(process = OU.process))
```

Once these parameters are defined, we can run the simulations and plot the results. We can contrast the results with the scenario without an event (but same random seed):

```
## Run the simulations without change
set.seed(1)
no_change <- dads(stop.rule = stop_time_6,
                  bd.params = speciation_1,
                  traits    = simple_bm_trait)
## Run the simulations with change
set.seed(1)
process_change <- dads(stop.rule = stop_time_6,
                       bd.params = speciation_1,
                       traits    = simple_bm_trait,
                       events    = change.process.to.OU)
## Plot the results
par(mfrow = c(1,2))
plot(no_change, ylim = c(-7, 7))
plot(process_change, ylim = c(-7, 7))
```



9.6 Changing trait correlation after reaching a trait value

For this scenario, we want to generate a pure birth tree with a 2 dimensional Brownian Motion trait with a strict correlation between the two dimensions (1:1) that loosen up when a taxa reaches an absolute value of 2.

First we need to set up the simulation parameters: * The stopping rule (100 taxa) * The birth-death parameters (speciation rate of 1)

```
## Set the parameters
stop_taxa_100 <- list(max.taxa = 100)
speciation_1 <- make.bd.params(speciation = 1)
```

Then set up our trait which is a 2 dimensional Brownian Motion with a correlation matrix Sigma ()

```
## A 2D variance covariance matrix
vcv_matrix <- matrix(1, 2, 2)

## A correlated 2D Brownian Motion
correlated_2D_BM <- make.traits(n = 2, process = BM.process,
                      process.args = list(Sigma = vcv_matrix))
```

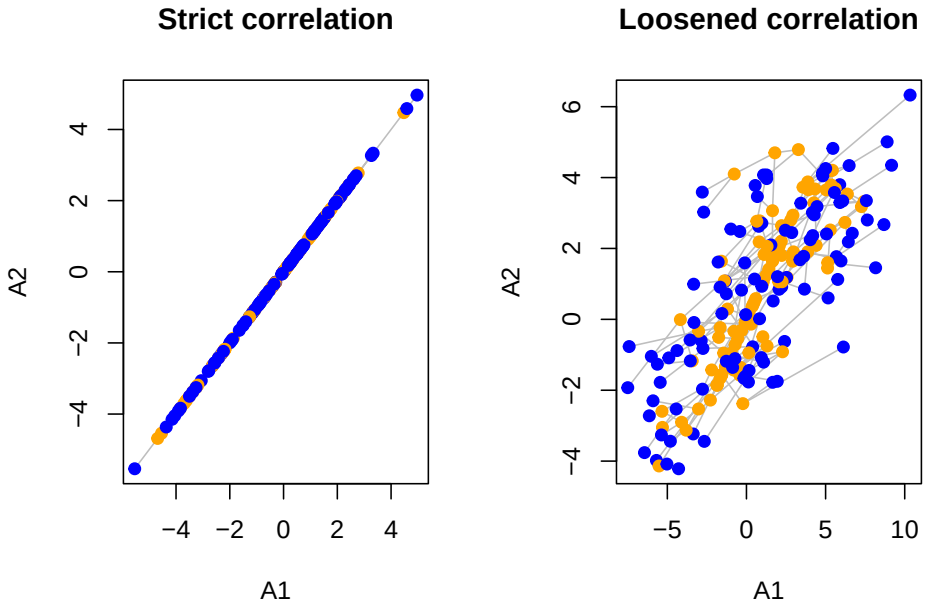
And our event which triggers after a taxa gets the trait value 3 (`trait.condition(3, absolute = TRUE)`), targets the `"traits"` and modifies the traits correlation

```
## New correlation
new_vcv <- matrix(c(10,3,3,2),2,2)
```

```
## Event changing a trait correlation
correlation.change <- make.events(
    condition    = trait.condition(3, absolute = TRUE),
    target       = "traits",
    modification = update.traits(process.args = list(Sigma = new_vcv)))
```

Once these parameters are defined, we can run the simulations and plot the results. We can contrast the results with the scenario without an event (but same random seed):

```
## Run the simulations
set.seed(2)
no_event <- dads(stop.rule = stop_taxa_100,
                 bd.params = speciation_1,
                 traits    = correlated_2D_BM)
set.seed(2)
change_correlation <- dads(stop.rule = stop_taxa_100,
                           bd.params = speciation_1,
                           traits    = correlated_2D_BM,
                           events    = correlation.change)

## Visual testing
par(mfrow = c(1,2))
plot(no_event, trait = c(1,2), main = "Strict correlation")
plot(change_correlation, trait = c(1,2), main = "Loosened correlation")
```



And we can visualise this change through time:

```
## 3D plot
plot(change_correlation, trait = c(1:2), use.3D = TRUE)
rglwidget()
```

```
## 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
```

9.7 Event for changing a modifier: speciation event increase for species with negative values

For this scenario, we want to generate a pure birth tree with a one dimensional Brownian Motion trait for 4 time units. After three time units, we want the speciation rule to increase for species which ancestors have a negative trait value.

First we need to set up the simulation parameters: * The stopping rule (5 time units) * The birth-death parameters (speciation rate of 1)

```
## Set the parameters
stop_time_4 <- list(max.time = 4)
speciation_1 <- make.bd.params(speciation = 1)
```

Then set up our trait which is a one dimensional Brownian Motion

```
## Trait
simple_bm_trait <- make.traits(n = 1, process = BM.process)
```

And a modifier that is the default birth-death algorithm rules

```
## Birth death rules (default)
default_modifiers <- make.modifiers()
```

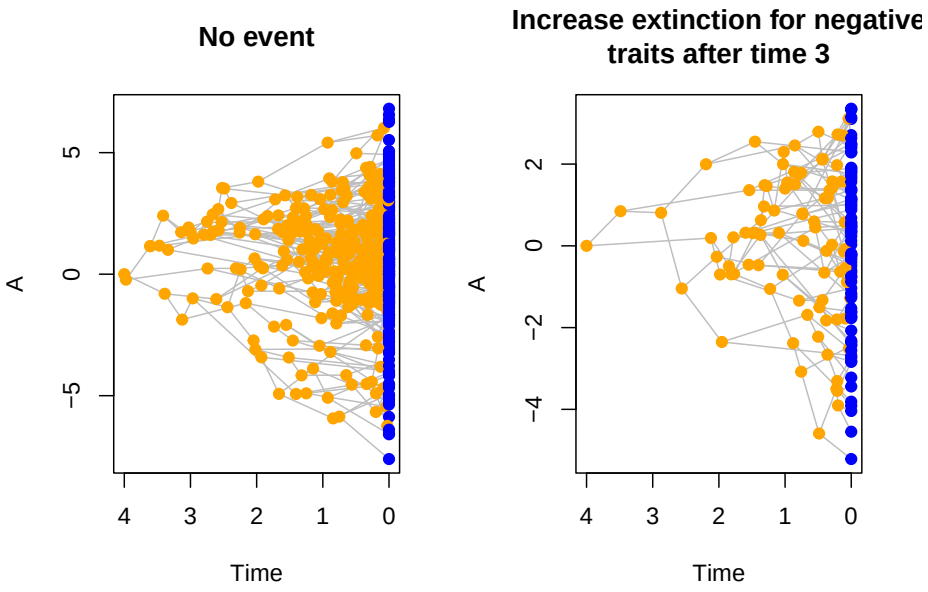
And our extinction event which triggers after reaching time 3 (`time.condition(3)`), targets the `"modifiers"` and modifies birth-death rule as follows: * When a species is descendant from a parent with a negative trait value (`negative.trait.condition`); * Then increase your chances of going extinct by +1 (`increase.extinction.1`)

```
## New condition and new modifier (increasing speciation if trait is negative)
negative.trait.condition <- function(trait.values, lineage) {
    return(parent.traits(trait.values, lineage) < 0)
}
increase.extinction.1 <- function(x, trait.values, lineage) {
  return(x + 1)
}

## Update the modifier
change.speciation <- make.events(
    condition    = time.condition(3),
    target       = "modifiers",
    modification = update.modifiers(speciation = speciation,
                                    condition  = negative.trait.condition,
                                    modify     = increase.extinction.1))
```

Once these parameters are defined, we can run the simulations and plot the results. We can contrast the results with the scenario without an event (but same random seed):

```
set.seed(4)
no_event <- dads(stop.rule = stop_time_4,
                 bd.params = speciation_1,
                 traits    = simple_bm_trait,
                 modifiers = default_modifiers)
set.seed(4)
change_spec <- dads(stop.rule = stop_time_4,
                    bd.params = speciation_1,
```

```
                    traits    = simple_bm_trait,
                    modifiers = default_modifiers,
                    events    = change.speciation)

## Visualise the results
par(mfrow = c(1,2))
plot(no_event, main = "No event")
plot(change_spec, main = "Increase extinction for negative\ntraits after time 3")
```



9.8 Changing branch length when reaching n taxa

For this scenario, we want to generate a pure birth tree until reaching 100 taxa. After reaching 30 taxa we want branch length growth to increase 50 folds.

First we need to set up the simulation parameters: * The stopping rule (100 taxa) * The birth-death parameters (speciation rate of 1)

```
## Set the parameters
stop_taxa_100<- list(max.taxa = 100)
speciation_1 <- make.bd.params(speciation = 1)
```

Then a modifier that is the default birth-death algorithm rules

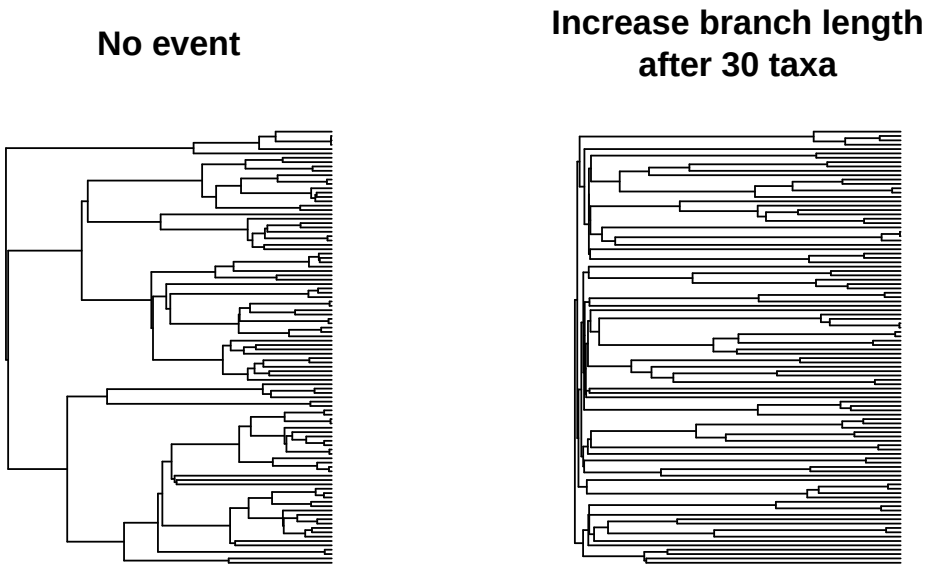
```
## Birth death rules (default)
default_modifiers <- make.modifiers()
```

And event which triggers after reaching 30 taxa (`taxa.condition(30)`), targets the `"modifiers"` and modifies the branch length generation rule to a 50 fold increase (`increase.50.folds`)

```
## Multiplying branch length 50 folds
increase.50.folds <- function(x, trait.values, lineage) {
  return(x * 50)
}
## Event for increasing branch length after reaching 30 taxa
increase_brlen <- make.events(
                  condition    = taxa.condition(30),
                  target       = "modifiers",
                  modification = update.modifiers(
                                    branch.length = branch.length,
                                    modify        = increase.50.folds))
```

Once these parameters are defined, we can run the simulations and plot the results. We can contrast the results with the scenario without an event (but same random seed):

```
## Run the simulations
set.seed(5)
no_event <- dads(stop.rule = stop_taxa_100,
                 bd.params = speciation_1,
                 modifiers = default_modifiers)
set.seed(5)
increased_brlen <- dads(stop.rule = stop_taxa_100,
                        bd.params = speciation_1,
                        modifiers = default_modifiers,
                        events    = increase_brlen)

## Visualise the results
par(mfrow = c(1,2))
plot(no_event, main = "No event", show.tip.label = FALSE)
plot(increased_brlen, main = "Increase branch length\nafter 30 taxa",
    show.tip.label = FALSE)
```



9.9 Founding event: a generating a subtree with no fossils

For this scenario, we want to generate a birth-death tree for 4 time units. After reaching 10 taxa, one random taxa will give birth to a sub-tree that is a pure birth tree (no extinction).

First we need to set up the simulation parameters: * The stopping rule (5 time units) * The birth-death parameters (speciation rate of 1 and extinction of 0.2)

```
## Set up parameters
stop_time_4 <- list(max.time = 4)
spec_1_ext_02 <- make.bd.params(speciation = 1, extinction = 0.2)
```

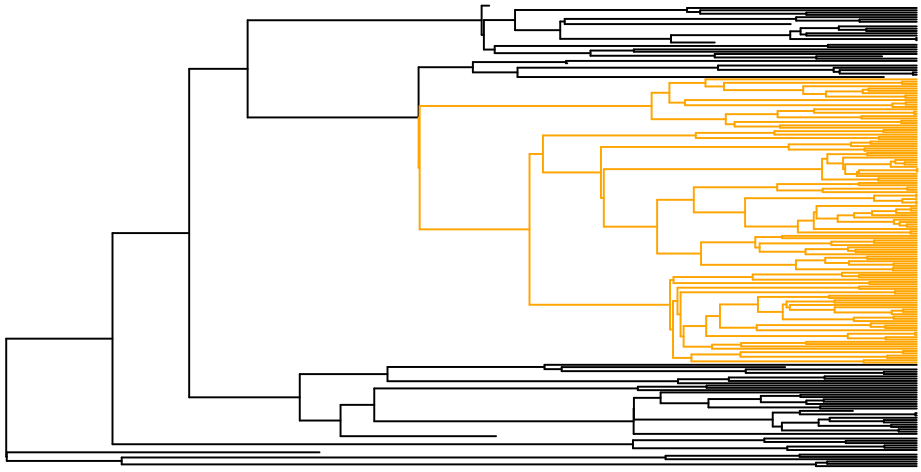
And our event which triggers after reaching 10 taxa (`taxa.condition(10)`), and generates a subtree ("founding") that is a pure birth tree (no extinction and speciation rate of 2).

```
## Setting the pure-birth parameters
speciation_2 <- make.bd.params(speciation = 2)

## Events that generate a new process (founding effects)
founding_event <- make.events(
                  condition    = taxa.condition(10),
                  target       = "founding",
                  modification = founding.event(
                                    bd.params = speciation_2),
                  additional.args = list(prefix = "founding_"))
```

Note we are prodviding an additional argument `prefix` here so that we can track which species are part of the sub tree for colouring them down the line.

Once these parameters are defined, we can run the simulations and plot the results:

```
## Simulations
set.seed(11)
founding_tree <- dads(stop.rule = stop_time_4,
                      bd.params = spec_1_ext_02,
                      events    = founding_event)

## Selecting the edges colours
tip_values <- rep("black", Ntip(founding_tree))
tip_values[grep("founding_", founding_tree$tip.label)] <- "orange"
edge_colors <- match.tip.edge(tip_values, founding_tree, replace.na = "black")

## Plotting the results
plot(founding_tree, show.tip.label = FALSE, edge.color = edge_colors)
```



9.10 Founding event: a generating a subtree a different process

For this scenario, we want to generate a birth-death tree with a one dimensional Brownian Motion trait for 6 time unit. After a taxa reaches the value 3 or higher, it gives birth to a sub-tree that generates an OU trait with an optimum at the value 3.

First we need to set up the simulation parameters: * The stopping rule (6 time units) * The birth-death parameters (speciation rate of 1 and extinction of 1/3)

```
## The tree parameters
stop_time_6 <- list(max.time = 6)
speciation_1_extinction_03 <- make.bd.params(speciation = 1,
                                             extinction = 0.3)
```

Then set up our trait which is a one dimensional Brownian Motion

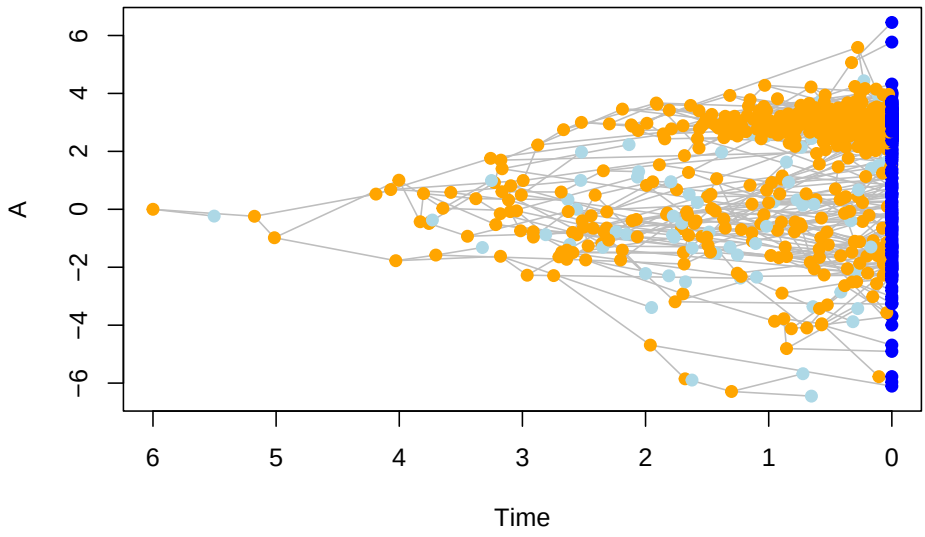
```
## Trait
simple_bm_trait <- make.traits(n = 1, process = BM.process)
```

When a taxa reaches the value 3 `trait.condition`, it generates a pure birth tree (`speciation = 2`) with an OU trait with the optimum value 3.

```
## The OU trait with an optimum value of 3
OU_3 <- make.traits(process = OU.process,
                    start = 3, process.args = list(optimum = 3))
## The pure birth parameters
speciation_2 <- make.bd.params(speciation = 2)

## The founding event
new_OU_trait <- make.events(
                    condition    = trait.condition(3, condition = `>=`),
                    target       = "founding",
                    modification = founding.event(
                                    bd.params = speciation_2,
                                    traits    = OU_3))
```

Once these parameters are defined, we can run the simulations and plot the results:

```
## Simulating the tree
set.seed(1)
founding_tree <- dads(stop.rule = stop_time_6,
                      bd.params = speciation_1_extinction_03,
                      traits    = simple_bm_trait,
                      events    = new_OU_trait)
plot(founding_tree)
```


A
6
4
2
0
−2
−4
−6
6
5
4
3
2
1
0
Time