# dads manual

Thomas Guillerme (guillert@tcd.ie), Alex Slavenko (email) and others

2020-12-04

# 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 Installing and running the package

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

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

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

## 1.3 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!

## 1.4 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:

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

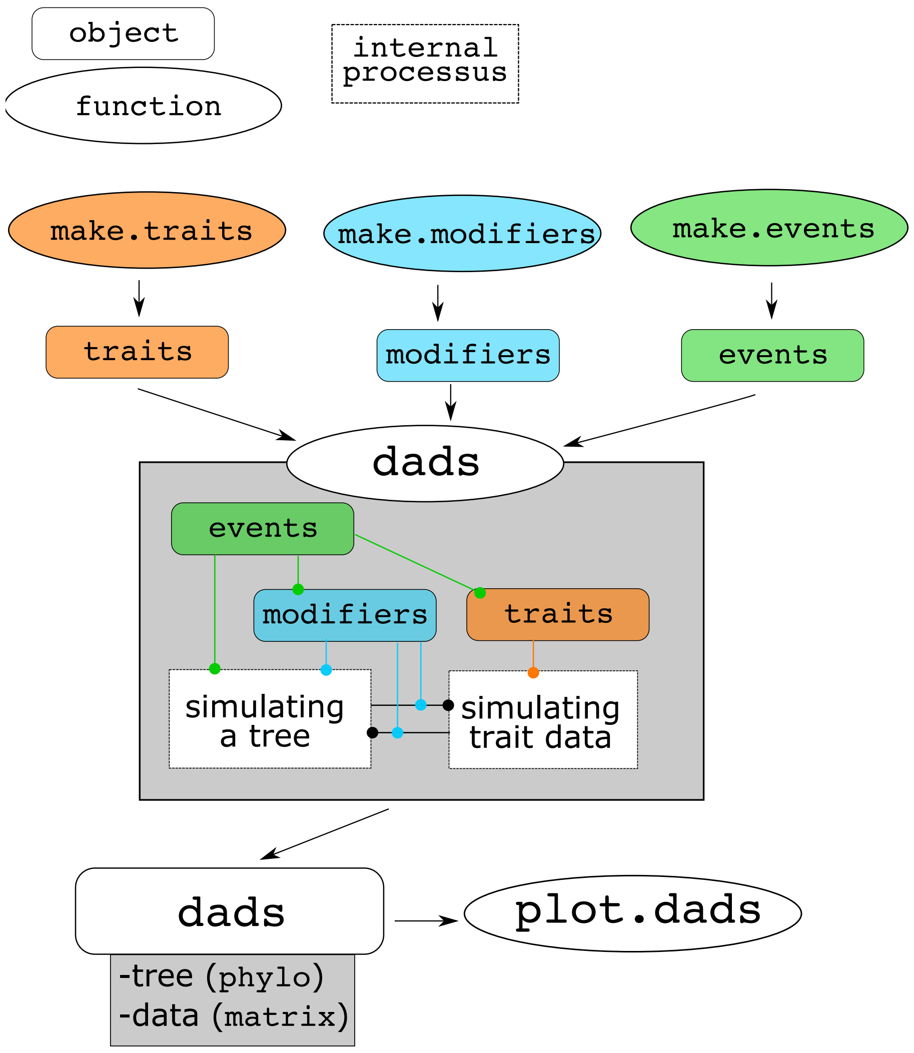


Figure 1.1: Schematised summary of the dads package architecture

# 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:

```r
## 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:

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

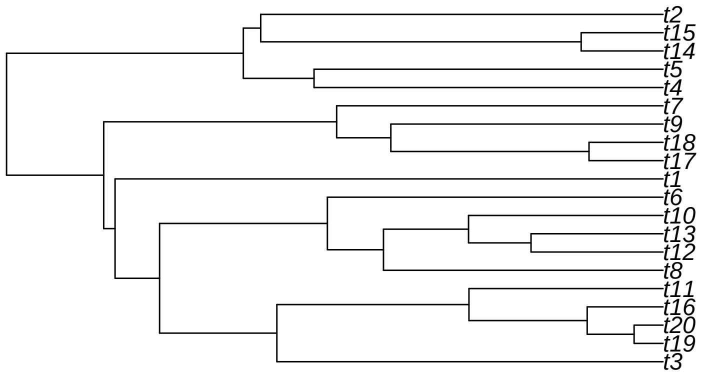
> Note that here we could have specified more than one stopping rule, for example, we might want to run a simulation and stop it if it either reaches 10 taxa or the age 2 using `stop.rule = list(max.time = 2, max.taxa = 10)`. The simulation will then stop when either of these conditions are met.

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

```r
## 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.
```

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



### 2.1.1 Changing the birth-death parameters

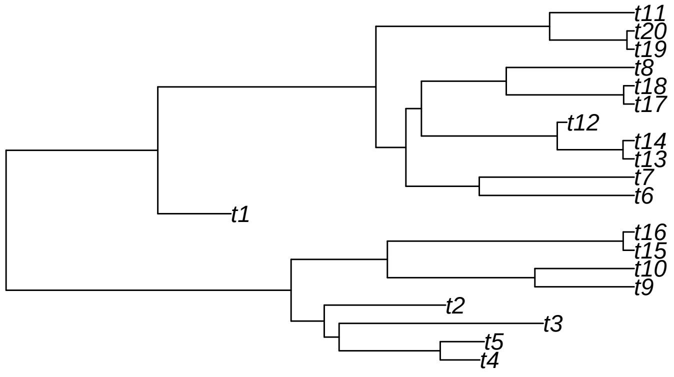
People familiar with the birth-death models might have noticed that we did not specify two important things here: the speciation parameter (sometimes called "lambda" or "birth") and the extinction parameter (sometimes called "mu", "death" or "background extinction"). By default `dads` runs a pure birth 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:

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

> Note that here it is not necessary to specify `extinction = 1` since this is the default option, you can always just change the parameter of interest (e.g. changing `extinciton = 0` to `extinction = 1/3`). However, we think it's good practice to attribute both parameters specifically to avoid any confusion.

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

```r
## 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:

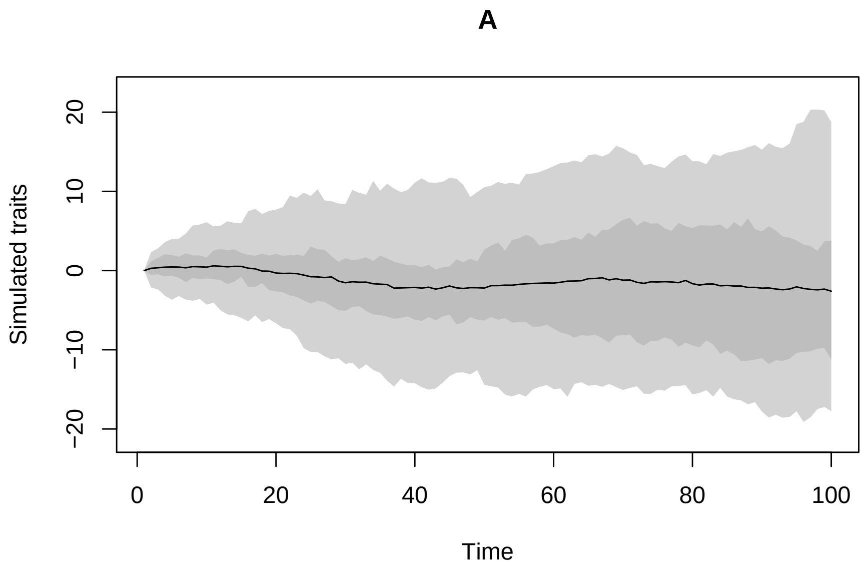
```
## Creating the traits object
my_trait <- make.traits()
```

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:

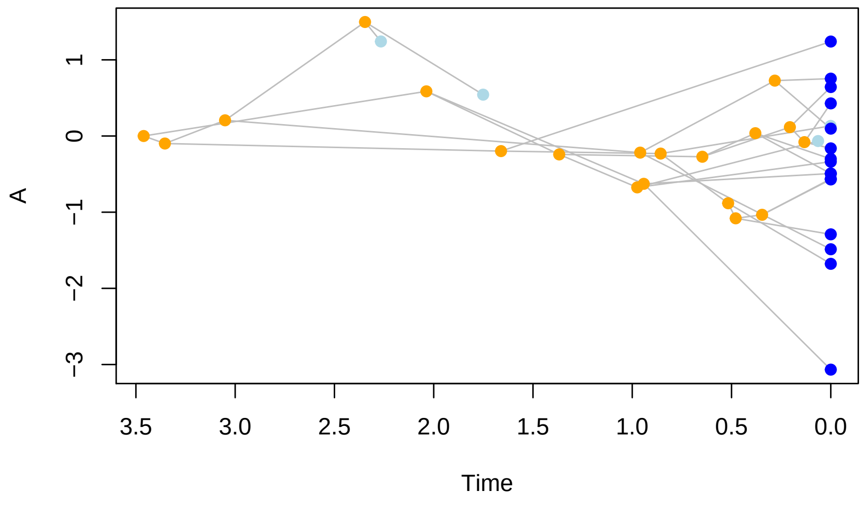
```
## Simulate disparity and diversity
my_data <- dads(bd.params = my_params,
                stop.rule = my_stop_rule,
                traits    = my_trait)
```

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:

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

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

```r
## 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):

```r
## 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 (default).
```

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:

```r
## 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:
## 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 add 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. First we are going to create the modification function. This function must intake the argument `x` and, in our case, return a logical value: `TRUE` is for speciate and `FALSE` is for go extinct. We want this `modify` function to always return `FALSE` (go extinct) since it will be triggered by `condition` (we will see that in a minute).

```
## Going extinct
staying.alive <- function(x) return(TRUE)
```

Now we want this function to only trigger when an ancestor has a negative trait value. 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`s understanding of what you which to asses:

```
## 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 (if the data is multidimensional). 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(
                        ## The speciation function (default)
                        speciation = speciation,
                        ## What to modify
                        modify     = staying.alive,
                        ## When to modify it
                        condition  = negative.ancestor)
## 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 (stayi
```

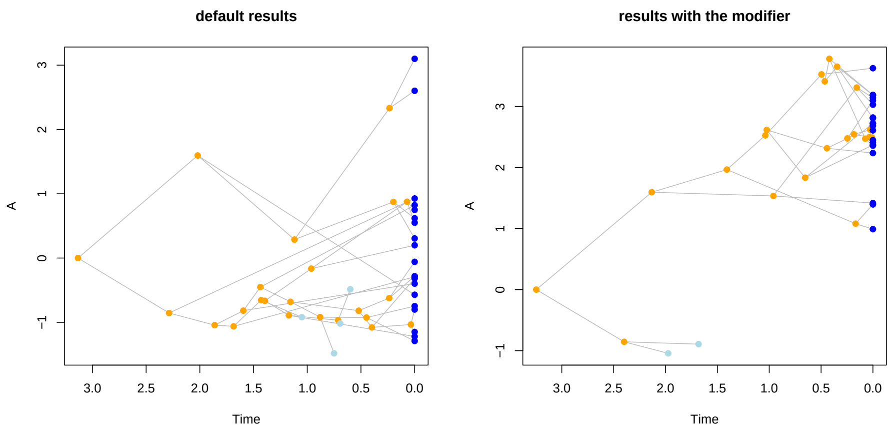
Note that the `make.modifiers` function tests whether the input is compatible with `dads` by default so unless you have an error message, your `modifiers` will work! We can now simulate our tree and traits with our modifier: species will go extinct if their ancestor have a negative trait value:

```r
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:
## Default branch length process.
## Default selection process.
## Speciation process is set to speciation with a condition (negative.ancestor) and a modifier (stayi
##
## 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! Also, although species

```r
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

# Making trait processes with `make.traits()`

## 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).

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 (though not dependent on either the previous state nor the branch length):

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

> Note that the argument `edge.length` is set to `1` by default. In general we highly recommend to set all arguments but `x0` to a default value (this really helps the speeding up the `dads` function).

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) {
    return(rnorm(1, mean = x0))
}
```

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:

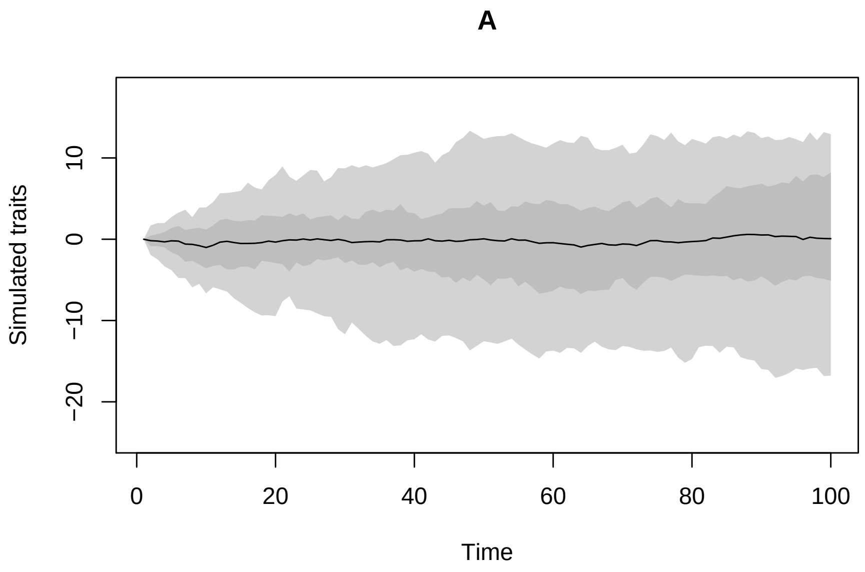
```
## The class of the object
class(my_trait_object)
```

```
## [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:

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

```
## $A
## $A$process
## function (x0, edge.length = 1, Sigma = diag(length(x0)), ...)
## {
##     return(t(MASS::mvrnorm(n = 1, mu = x0, Sigma = Sigma * edge.length,
##           ...)))
## }
## <bytecode: 0x7f7eeee57f58>
## <environment: namespace:dads>
##
## $A$start
## [1] 0
##
## $A$trait_id
## [1] 1
```

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.

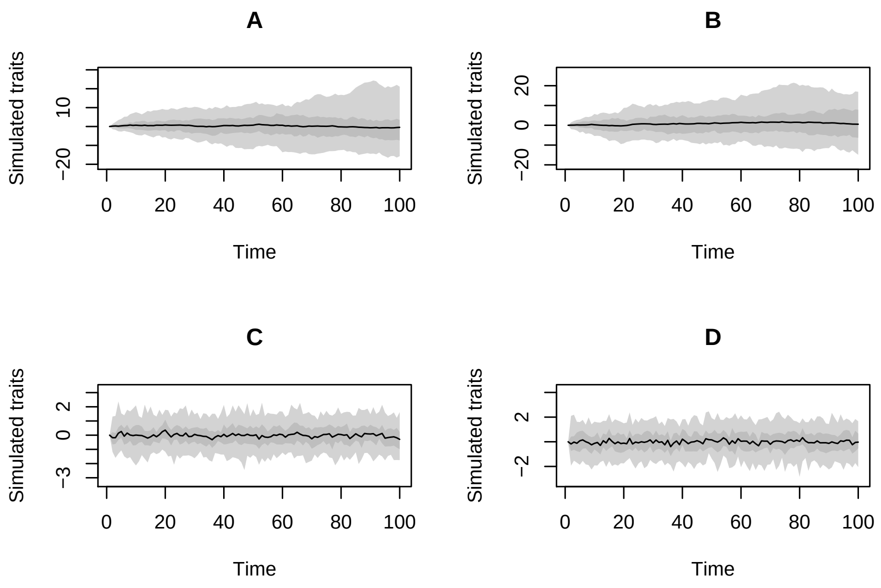
Note that it is possible to make `"traits"` objects with multiple processes (that can be the same):

```r
## 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`:

```r
## 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).
```

## 3.3 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 (0)
## 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:

```r
## 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).
```

```r
## 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 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 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`.

## 3.7 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 but `x0` with default values (just in case). Also, note that the functions below are not equal to the already implemented `BM.process` and `OU.process` but are rather generalised/simplified version that you can use as a template

### 3.7.1 A simple Brownian Motion process template

```
## A simple Brownian motion process
my.BM.process <- function(x0, 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 = sd * edge.length)

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

### 3.7.2 A simple Ornstein-Uhlenbeck process template

```
## A simple Ornstein-Uhlenbeck motion process
my.OU.process <- function(x0, 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 = sd * branch.length)

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

# Chapter 4

# Making modifiers with `make.modifiers()`

`"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 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 or 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, $\lambda$ and $\mu$ 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:

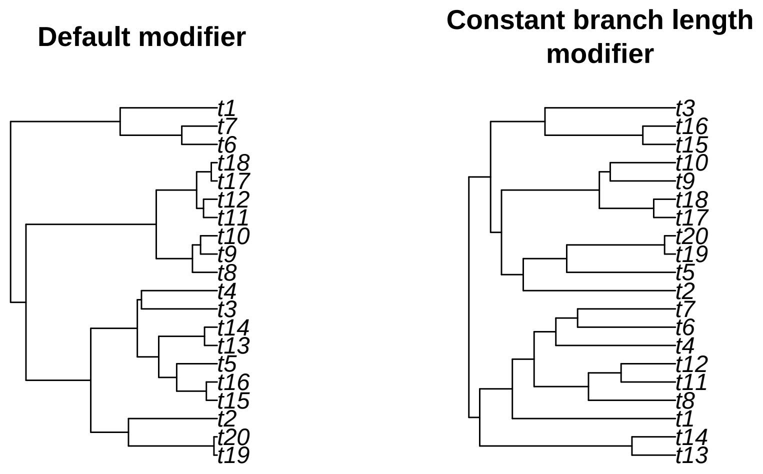
```r
## Specifying the default modifier
default_modifiers <- make.modifiers(branch.length = branch.length)

## Setting some parameters for generating trees
bd_params  <- list(extinction = 0)
stope_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 = stope_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.

```r
## 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 = stope_rule,
                      modifiers = constant_modifier)
```

And we can visualise the difference between both resulting trees:

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



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

### 4.1.1 The allowed arguments

It is of course possible to design some more advanced function to interact with the birth death process. 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 `n.taxa` as a function argument but are using `n.taxa` in the function, `n.taxa` will be detected and treated as a current argument automatically as set accordingly within the birth death process (e.g. `n.taxa` 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):

```
## 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)))
    }
}
```

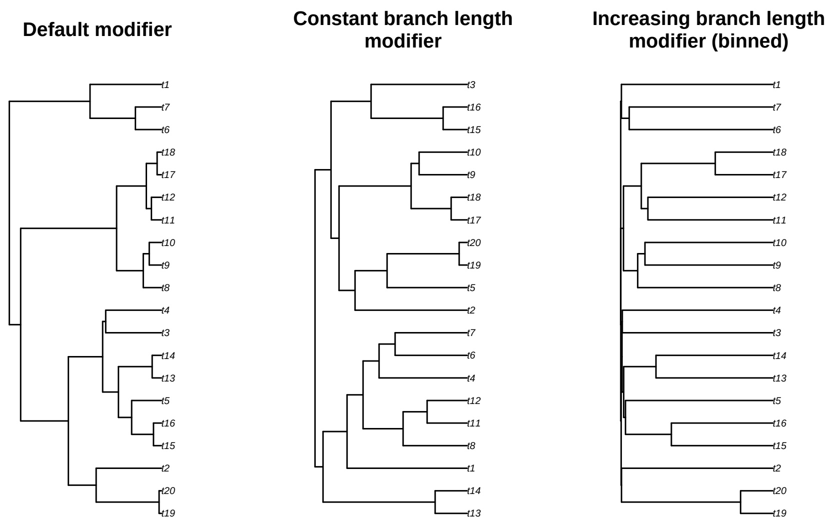
We can then create it as a `"modifiers"` object and run a new simulation:

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

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

And we can visualise the difference between the resulting trees:

```r
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)")
```



## 4.2 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:n.taxa, 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):

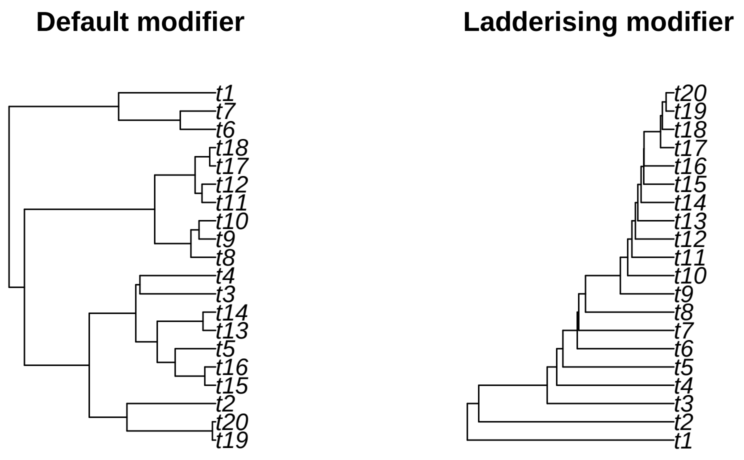
```r
## Our function to always select the last taxon
## (making sure it returns an integer)
```

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

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

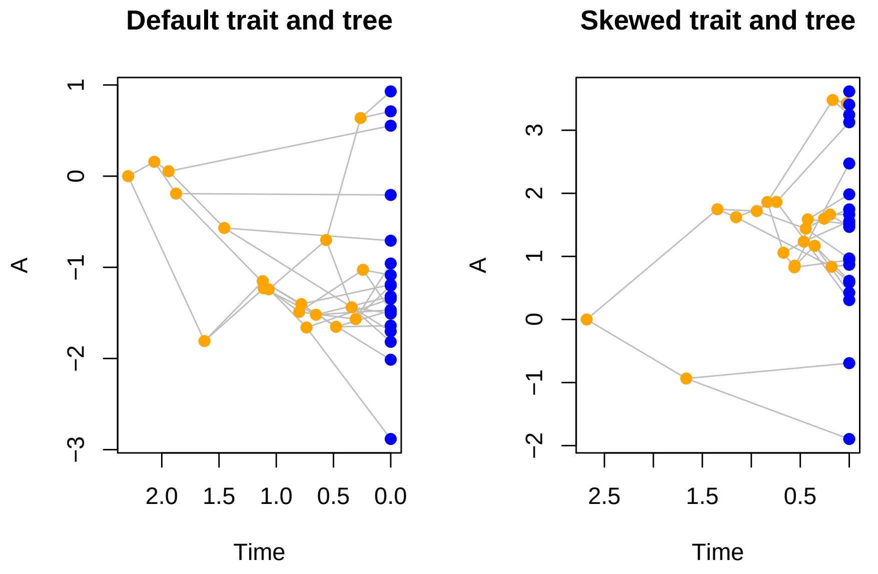
We can then create a `"modifiers"` object the same way as before using this time the `selection` argument:

```
## 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 = stope_rule,
                        modifiers = ladderised_modifier)
## Displaying the results
par(mfrow = c(1,2))
plot(default_tree,    main = "Default modifier")
plot(ladderised_tree, main = "Laddering 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 creates a trait a add to it a `selection` modifier that only selects for tips with positive trait values.

```r
## Our function that only select taxa with positive trait values
select.positive <- function(trait.values, lineage) {
    ## Selecting the taxa with positive traits only
    positives <- as.integer(rownames(trait.values)[which(trait.values[, 1] >= 0)])

    ## Select the current taxa that descend from a node with a positive value
    positive_living <- cbind(lineage$parents, seq_along(lineage$split)
                             )[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 = stope_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 = stope_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")
```



## 4.3 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)
```

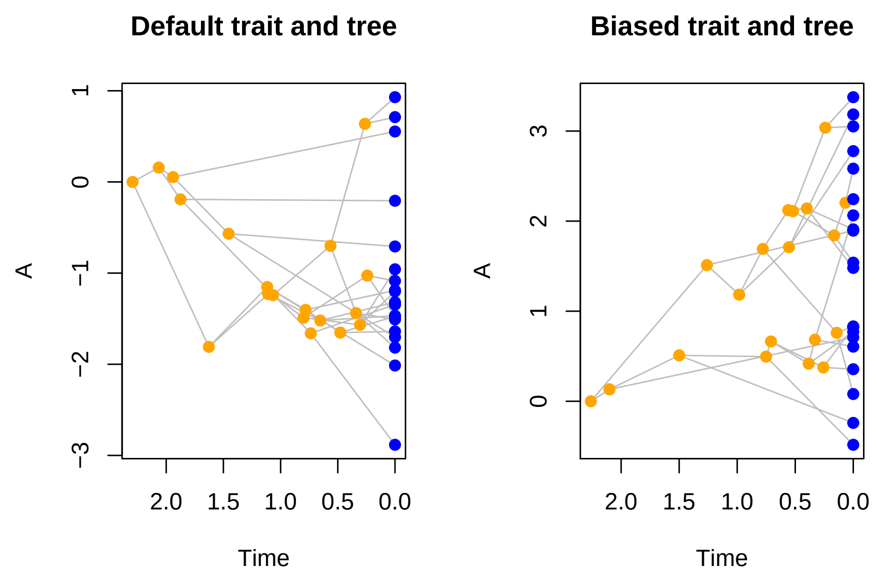
Creating a `"modifiers"` with a `speciation` function works the same way as for `branch.length` and `selection` but the function that will be used needs to output a logical value (see table below). Once the function is created simply input your function for speciation in the modifier and run the `dads` 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(0)
modified_tree <- dads(bd.params = bd_params,
                      stop.rule = stope_rule,
                      modifiers = random_extinction)

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

Note how every lineage end up going extinct! 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 = stope_rule,
                          traits    = BM_trait,
                          modifiers = biased_trait)
```

```r
## 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")
```



## 4.4 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"` |

## 4.5 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).

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.

## 4.6 Combining and editing modifiers (`add`)

## 4.7 Testing modifiers (`test`)

## 4.8 Demo runnable

```r
bd.params <- list(speciation = 1, extinction = 1/3)
traits <- make.traits()
stop.rule <- list(max.taxa = 20, max.living = Inf, max.time = Inf)
modifiers <- NULL
events <- NULL
null.error <- NULL

## modifiers
condition <- function(trait.values, parent.lineage) return(parent.traits(trait.values, parent.lin
modify <- function(x) return(x * 20)

## Setting up the different modifiers
modify_speciation <- make.modifiers(speciation    = speciation.trait,
                                    condition     = condition,
                                    modify        = modify)
modify_brlen <- make.modifiers(branch.length = branch.length.trait,
                               condition     = condition,
                               modify        = modify)
modify_speciation_brlen <- make.modifiers(branch.length = branch.length.trait,
                                          speciation    = speciation.trait,
                                          condition     = condition,
                                          modify        = modify)

## Test normal (no modifiers)
set.seed(1)
test <- dads(bd.params, stop.rule, traits, null.error = 20)
par(mfrow = c(4,2))
```

```r
plot(test$tree)
plot.dads(test, main = "random tree + trait")

## Test with modifiers
set.seed(1)
trait_table <- NULL
test <- dads(bd.params, stop.rule, traits,
                                modifiers = modify_speciation,
                                null.error = 20)
plot(test$tree)
plot.dads(test, main = "Skewed speciation")

set.seed(1)
trait_table <- NULL
test <- dads(bd.params, stop.rule, traits,
                                modifiers = modify_brlen,
                                null.error = 20)
plot(test$tree)
plot.dads(test, main = "Skewed branch length")

set.seed(1)
trait_table <- NULL
test <- dads(bd.params, stop.rule, traits,
                                modifiers = modify_speciation_brlen,
                                null.error = 20)
plot(test$tree)
plot.dads(test, main = "Skewed branch length and speciation")
```