BeePODYNA : package for population dynamics

December 19, 2019

Julia Guerra - M2 BEE

Contact me

Abstract

This document provides guidance for the effective use of the 'BeePODYNA' package for modelling biological population dynamics. BeePODYNA allows the simulation of one or more interacting populations, the monitoring of their evolution and the graphic representation of the process.

Contents

1 Setting up

BeePODYNA has been created by a group of students from the BEE Master's in Grenoble. It is a novel project that brings together the first steps in the world of code by its creators. Therefore, any suggestion or bug-report is more than welcome. Here, I introduce the main features of the package and I perform some basic simulations that allow to understand the logical order of the application for which it has been built.

The installation of the BeePODYNA package cannot be done from a CRAN repository yet. However, it is available online:

```
  install_github("BEE-Univ-Grenoble/BeePODYNA")
library(BeePODYNA) # don't forget to load it tough
```

1.1 Base elements

1.1.1 Populations, Communities

BeePODYNA has been produced by a group of students. The main objective of BeePODYNA is the monitoring of biological systems. As such, the first object to be managed will be the populations and communities (population groups). Each population object is identified with a *label*, and is composed of several parameters: (a) the population' size for time 0 ; (b) its growth rate; (c) (not mandatory) the environment biotic capacity.

```
mackerel <- population(label = "Mackerel", initial_size = 200, growth_rate = 1, capacity = 1000)
is_population(mackerel)

#  [1] TRUE
```

Populations can also be created from lists containing those same features and even several population' sizes, as follows:

```
tunafish <- list(label = "Tuna",
                 size = c(20,15,30,35,40,39,42,45),
                 time = c(1:8),
                 # obtaining GR from birth and death rates : "growth_rate" function
                 growth_rate = growth_rate(birth_rate = 1.7, death_rate = 1.1),
                 capacity = 24)

is_population(tunafish)

#  [1] FALSE

tunafish <- as_population(tunafish)
is_population(tunafish)

#  [1] TRUE
```

Several functions included in this package allow access to populations' base parameters. Anyway, the faster way to get a general view is the `summary()` function.

```
grate_population(tunafish)

## [1] 0.6

label(tunafish)      ;     size_population(tunafish)     ;     capacity_population(tunafish)

## [1] "Tuna"
## [1] 20 15 30 35 40 39 42 45
## [1] 24
```

```
summary(tunafish)

## [[1]] - Population label:   Tuna
##
## [[2]] - Population size:
##   Generations Size
## #           n   45
## #         n-1   42
## #         n-2   39
## #         n-3   40
## #         n-4   35
## #         n-5   30
## #          n0   20
##
##
## [[3]] - Generations:
## This population has subsisted for  8  generations.
##
## [[4]] - Rates of population change between generations:
##        Generations  Rates.of.change Sizes
```

```
## #             [ n ]                  -    45
## #    n over [ n-1 ] 1.07142857142857    42
## # n-1 over [ n-2 ] 1.07692307692308    39
## # n-2 over [ n-3 ]             0.975    40
## # n-3 over [ n-4 ] 1.14285714285714    35
## # n-4 over [ n-5 ] 1.16666666666667   ...
## # (n1 over [ n0 ])              0.75    20
##
##
## [[5]] - Biotic capacity:
## Biotic capacity is =  24 individuals
## Population has reached  188 %  of it at generation 'n'.
##
## - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

A community is a group of two or more populations occurring simultaneously in space and time. Respective parameters of populations within a community and their interactive responses will decide the future of the community. Therefore, it is more interesting to work with communities, for which we use the function `community()`. Similarly, communities' parameters can be consulted by means of the `summary()` function or in a separate way.

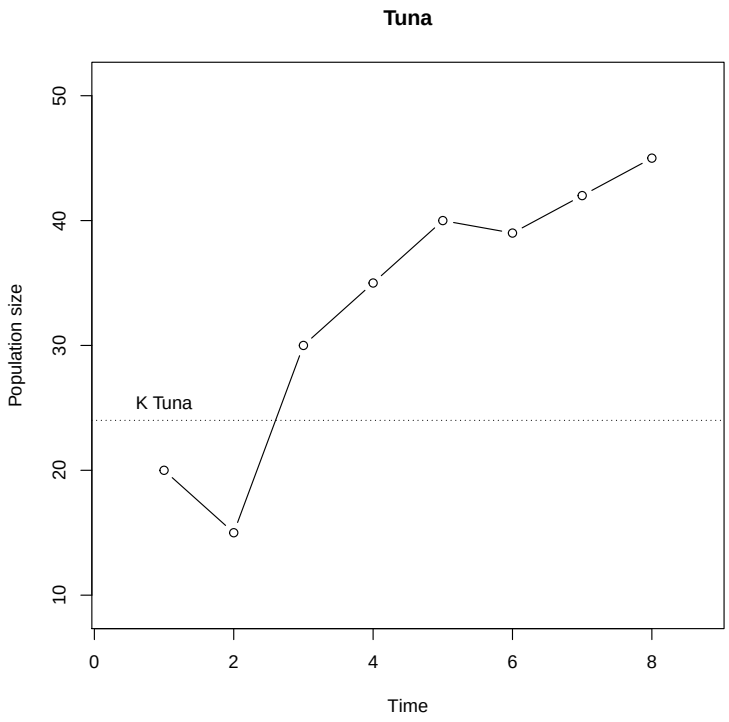
```
highseas <- community(label = 'Sea', tunafish , tunafish)
is_community(highseas)

## [1] TRUE

length(highseas) # interesting Method for getting number of populations inside

## [1] 2

label(highseas)

## [1] "Sea"
```

Even if the communities' object structure is somehow particular, BeePODYNA counts on a 'short-cut' method for accessing the populations (or sub-objects) inside the communities. For instance, you can take a look at any of the `tunafish` population present within `highseas` by calling `highseas$tunafish`.

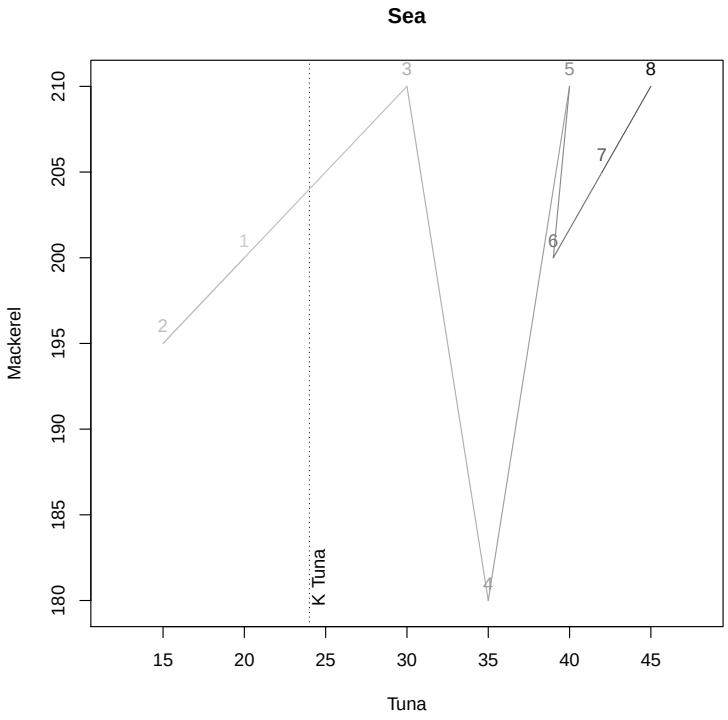
1.2 Graphical representation

BeePODYNA includes several interesting features to customize populations' plots. Moreover, a specific way of plotting communities has been included on it.

```
plot(tunafish)
```




```
mackerel <- as_population(
  list(label = "Mackerel",
                 size = c(200,195,210,180,210,200,205,210),
                 time = c(1:8),
                 growth_rate = 1.3,
                 capacity = 1000))

highseas <- community(label = 'Sea', tunafish , mackerel)
com_plot(highseas)
```



1.3 Species' interactions

Since the ultimate goal of the BeePODYNA package is the simulation of interacting species, most functions of the package expect the presence of a matrix describing the parameters of interaction between species. Even if the species within the simulated community do not present effective interaction between them, it is necessary to include such a matrix (in which might be composed by null coefficients). You can create the interaction matrix by hand:

```
fishes_IM <- c(0.4,0.2)
fishes_IM <- as_interactions(fishes_IM)
# will consider first coefficient for first species, second for second species, etc
fishes_IM
```

```
##       Pop_1 Pop_2
## Pop_1   0.0   0.2
## Pop_2   0.4   0.0
## attr(,"class")
## [1] "interactions"
```

Or by means of the `interactions()` function:

```
fish_IM2 <- interactions(nb_pop  = 2, x.interactions = c(0.4,-0.2),
                         labels = names(highseas$populations))
fish_IM2
```

```
##          Tuna Mackerel
## Tuna      0.0     -0.2
## Mackerel  0.4      0.0
## attr(,"class")
## [1] "interactions"
```

```
is_interactions(fish_IM2)
```

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

2 BeePODYNA objects

This is a standarized kind of object that regroups communities and interactions' data in order to carry on the wished simulations. They are the 'dining room' of the BeePODYNA package, since everything will happen inside them. BeePODYNA objects will be composed of a comunity and its interaction matrix, but you will also need to specify which kind of mathematical functions might act in their populations' dynamics.

```
highseas_l <- beepodyna(label = "Highseas_logistic",
                        community = highseas,
                        interactions = fish_IM2,
                        functions = c(logistic_growth,logistic_growth),
                        # Note: each function applied to each population in the same order
                        verbose = TRUE)

label(highseas_l) # assessing attributes from a beepodyna object

## [1] "Highseas_logistic"

length(highseas_l)

## [1] 2
```

3 Running simulations

Let's see now how to use the BeePODYNA package to perform simulations of greater or lesser complexity. It is possible to run one-step simulations, as well as multiple iterations' dynamics that will run a loop on the previous one-step process.

```
highseas_l1 <- simulate_onestep(highseas_l)
highseas_l8 <- simulate_n_pop_dynamic(highseas_l, n = 8) # 8 iterations
```

3.1 Custom models' simulations

BeePODYNA package basically includes exponential growth and logistic growth functions. However, if these available functions for modelling populations' dynamics are not complex enought for your modelling framework, it is possible to run simulations based on a self-added mathematical model. This section will show you how to create and implement additional functions for BeePODYNA simulations.

But first, let's take a look at one of our celebrities: the logistic growth function.

```
exponential_growth

## function(pop, community, interactions) {
##   target_pop <- community$populations[[pop]]
##   last_time  <- target_pop$time[length(target_pop$time)]
##   last_size  <- target_pop$size[length(target_pop$size)]
##
##   new_size <- last_size * (1 + target_pop$growth_rate)
##   new_time <- last_time + 1
##
##   target_pop$size <- append(target_pop$size, new_size)
##   target_pop$time <- append(target_pop$time, new_time)
##
##   community$populations[[pop]] <- target_pop
##
##   community
```

```
## }
## <environment: namespace:BeePODYNA>
```

The arguments that this function needs are, then, the biological system (either a population or a community, but both must be included) and the interactions' matrix. If the user wants to include a function, it is mandatory to follow these basis.

Then, let's see how to include a new function. I guess you already know the Lotka-Volterra population dynamics' model for preys and predators. It is basically made of two populations, for which the interaction coefficients are the opposite : positive for predators, negative for preys. Because of these differences in population dunamics, writing two different functions seem a good approach(see equation 1). However, note that this fact will make impossible to apply this function to a model in with more than two species within.

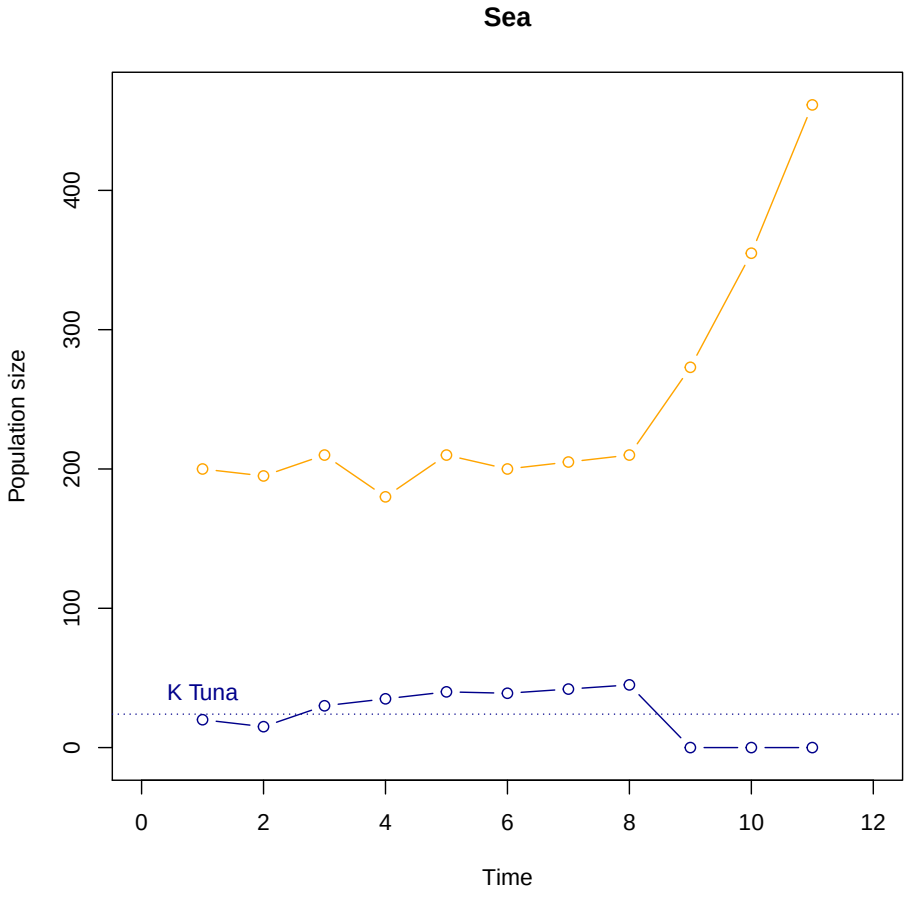
$$\begin{cases} prey_{t+1} = r_{prey} \cdot prey_t - a \cdot prey_t \cdot pred_t \\ pred_{t+1} = r_{pred} \cdot pred_t - b \cdot prey_t \cdot pred_t \end{cases} \tag{1}$$

*with : **r** the growth rate, **a** the interaction of pred on prey (predation) and **b** the interaction of prey on pred (nutrition)*

```
lotka_prey <- function(pop, community, interactions){
  target_pop <- community$populations[[pop]] # pop is the number of the population in the community
  last_time  <- target_pop$time[length(target_pop$time)]
  last_size  <- target_pop$size[length(target_pop$size)]

  pred_pop <- community$populations[[-pop]]
  pred <- label(pred_pop)
  last_size_pred  <- pred_pop$size[length(pred_pop$size)]

  int <- interactions[pop,pred]

  new_size <- last_size * target_pop$growth_rate + int * last_size * last_size_pred
  new_time <- last_time + 1

  if(new_size<=0) new_size = 0

  target_pop$size <- append(target_pop$size, new_size)
  target_pop$time <- append(target_pop$time, new_time)

  community$populations[[pop]] <- target_pop

  community
}

lotka_pred <- function(pop, community, interactions){
  target_pop <- community$populations[[pop]] # pop is the number of the population in the community
  last_time  <- target_pop$time[length(target_pop$time)]
  last_size  <- target_pop$size[length(target_pop$size)]

  prey_pop <- community$populations[[-pop]]
  prey <- label(prey_pop)
  last_size_prey  <- prey_pop$size[length(prey_pop$size)]

  int <- interactions[pop,prey]

  new_size <- last_size * target_pop$growth_rate + int * last_size * last_size_prey
  new_time <- last_time + 1
```

```
  if(new_size<=0) new_size = 0

  target_pop$size <- append(target_pop$size, new_size)
  target_pop$time <- append(target_pop$time, new_time)

  community$populations[[pop]] <- target_pop

  community
}
```

```
fishes_LV <- beepodyna(label = "fishes_LV", community = highseas, interactions = fish_IM2,
  functions = c(lotka_pred,lotka_prey), verbose = TRUE)

plot(simulate_n_pop_dynamic(fishes_LV, 3)$community, col = c("darkblue","orange"))
```



Dependances

Very few basic dependences are needed for this package to perform perfectly. These packages are included in the basic R installation, but function troubleshooting can be caused by misuse of these packages.