BeePODYNA : A R tool to simulate populations dynamics.

December 20, 2019

Maxime Jaunatre, Master 2 BEE Grenoble

Mail

This document provide an introduction to the BeePODYNA package, a collective work produced by the M2 BEE class of Grenoble for a modelisation course. This package is made to provide a new tool for simulation and observation of population dynamics. To do so, this package regroup multiple functions to simulate a population, its interactions with other populations in a community and the dynamic result of interaction during a time period. Each element used in these simulations will be explain with the goal to provide a framework for each user to create it's own system.

Contents

First point, to install the package, you may want to load it from Github, as it's not yet avalaible on CRAN. This installation done with the devtools package.

```r
if("BeePODYNA" %in% installed.packages()) {
  library(BeePODYNA)
}else{
  library(devtools)
  install_github("BEE-Univ-Grenoble/BeePODYNA")
  library(BeePODYNA)
}
```

Populations dynamics are difficult to apprehend, as they depend on many factors when their development take place in a complexe ecosystem. It is therefore usefull to modelize them with informatic and mathematic tools, starting off with simple mechanisms. The modelization approach illustrated here is using simulations to produce informatic communities, an usefull feature to compare with empiric observations. The main goal of the vignette is then to explain the different aspect of simulating a population dynamic and finally produce a simulation of two interacting populations.

1 Objects

1.1 Populations

At the very base of the simulation is the population. It can either be created from scratch (`population()`) or imported from a list into a population object (`as_population()`). In every case, it can be useful to test if your object is a population before using it extensively. The population is defined as a very simple evolution of size during a time period, the maximum capacity of the environnement concerning this population and a growth rate. More complex populations with age levels (maturity for exemple) can be produced based on this model, but it require a carefull analysis of the growth rate and other interactions which be described later on.

```
worms <- population(label = "worms", initial_size = 13, growth_rate = 10, capacity = 500)
cats <- list(label = "cats", size = c(1,1.2,1.3,1.4,1.5), time = c(0:4),
             growth_rate = 1.5, capacity = 24)
is_population(cats)
cats <- as_population(cats)
is_population(cats)

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

If the growth rate of a population is missing in your dataset, you can compute it with a basic function (`growth_rate()`) provided in this package.

```
growth_rate_cat = growth_rate(birth_rate = 2, death_rate = 0.5) ; growth_rate_cat

#  [1] 1.5
```

Different attributes of a population are needed for further computations and are accessible with different functions. All of these attributes are also accessible with the `$` operator for more readible coding practices.

```
size_population(worms) ; grate_population(worms) ; label(worms) ; capacity_population(worms)

#  [1] 13
#  [1] 10
#  [1] "worms"
#  [1] 500

## worms$size ; worms$growth_rate ; worms$label ; worms$capacity
```

Finally, you can assess the different values of a populations object by opening it, but it is easier to get values from a summary, or look at a graphic representations. This last option is also very close to the basic plot function of the `graphic` package of R, enabling all the options for prettier plots.

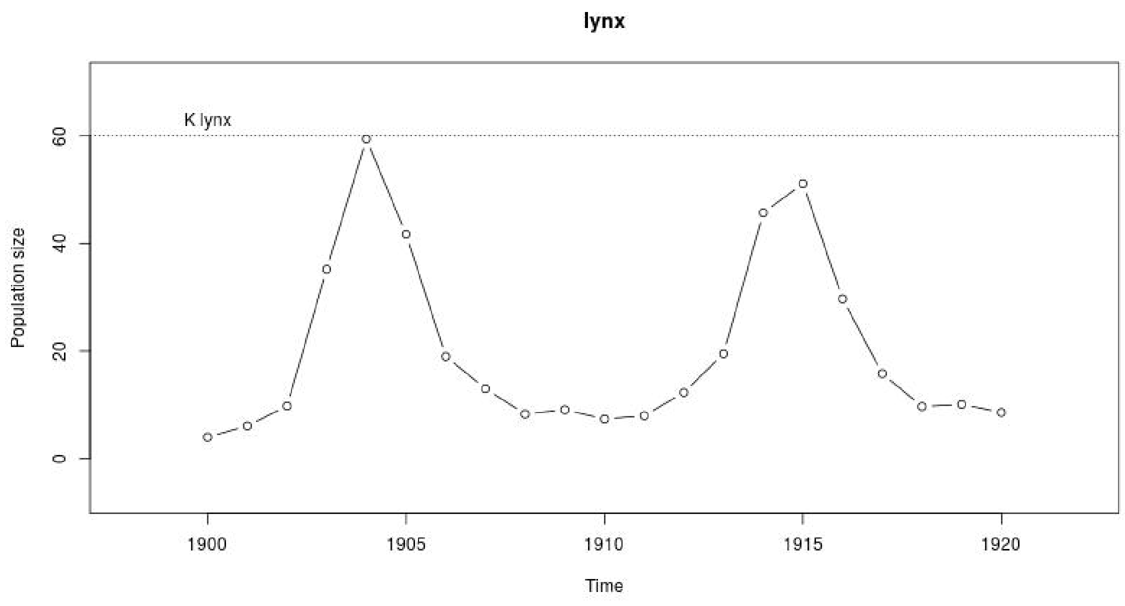

Figure 1: **Plot of population "lynx"**, extracted from the "hudson" example dataset

```
data(hudson)
lynx <- hudson$lynx
plot(lynx)
summary(lynx)
```

```
#  [[1]] - Population label:   lynx
#
#  [[2]] - Population size:
#    Generations Size
#  #           n  8.6
#  #         n-1 10.1
#  #         n-2  9.7
#  #         n-3 15.8
#  #         n-4 29.7
#  #         n-5 51.1
#  #          n0  4.0
#
#
#  [[3]] - Generations:
#  This population has subsisted for  21  generations.
#
#  [[4]] - Rates of population change between generations:
#         Generations   Rates.of.change Sizes
#  #            [ n ]                 -   8.6
#  #   n over [ n-1 ] 0.851485148514851  10.1
#  # n-1 over [ n-2 ]  1.04123711340206   9.7
#  # n-2 over [ n-3 ] 0.613924050632911  15.8
#  # n-3 over [ n-4 ] 0.531986531986532  29.7
#  # n-4 over [ n-5 ] 0.581213307240705   ...
#  # (n1 over [ n0 ])             1.525     4
#
#
#  [[5]] - Biotic capacity:
#  Biotic capacity is =  60 individuals
#  Population has reached  14 %  of it at generation 'n'.
#
#  - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

1.2 Community

For studying populations interactions, they are grouped in a community object. This is done with the `community()` function. There is also the possibility to check the class of a community object. As precised before for population, there is no theorical limit to the number of populations, and if they are from different species or not. Any user can then produce a great variety of model, from a multi-age species metapopulation to a complexe community with a rich trophic network.

```
worms <- population(label = "worms", initial_size = 13, growth_rate = 10, capacity = 500)
rats <- population(label = "rats", initial_size = 2, growth_rate = 5, capacity = 100)
cats <- population(label = "cats", initial_size = 1, growth_rate = 1, capacity = 20)
wasteland <- community(label = 'wasteland',worms,rats , cats)
is_community(wasteland)

#  [1] TRUE
```

Like a population, different attributes of a community are accessible with multiple functions. There is a modification of the `length()` function, providing the number of population composing it. The `$` operator is still working, but the community is build with a different aspect than population and it is then trickier to access some information as the user need to dig a long way inside the object.

```
label(wasteland) ; length(wasteland)

#  [1] "wasteland"
#  [1] 3

## wasteland$label ; length(wasteland$populations)
## wasteland$populations$rats$capacity # capacity of the rats population
```

A summary function is also provided to expose different values of a community object. A function to graphically represent a community is close to the precedent one for population. The addition of multiple populations that are in a community put in perspective the need to differenciate them with graphical aspects as shown in the figure 2.

```
data(hudson)
plot(hudson, col = c("tan3","slateblue3"), cex = c(1,1.5), pch = c(1,2))
graphics.off()
summary(hudson)

#  [[1]] - Community label:   Hudson
#
#  [[2]] - Population size:
#    Generations hare lynx
#  #        [ n ] 30.0  4.0
#  #      [ n-1 ] 47.2  6.1
#  #      [ n-2 ] 70.2  9.8
#  #      [ n-3 ] 77.4 35.2
#  #      [ n-4 ] 36.3 59.4
#  #      [ n-5 ] 20.6 41.7
#  #       [ n0 ] 24.7  8.6
#
#
#  [[3]] - Generations:
#  This community has subsisted for  21  generations.
#
#  [[4]] - Rates of population change between generations:
#    Generations   Rates of change for...       hare      lynx
#  #        1920                      [ n ]        NA        NA
#  #        1919             n over [ n-1 ] 0.6355932 0.6557377
#  #        1918 [ n-1 ] over [ [ n-2 ] ] 0.6723647 0.6224490
```

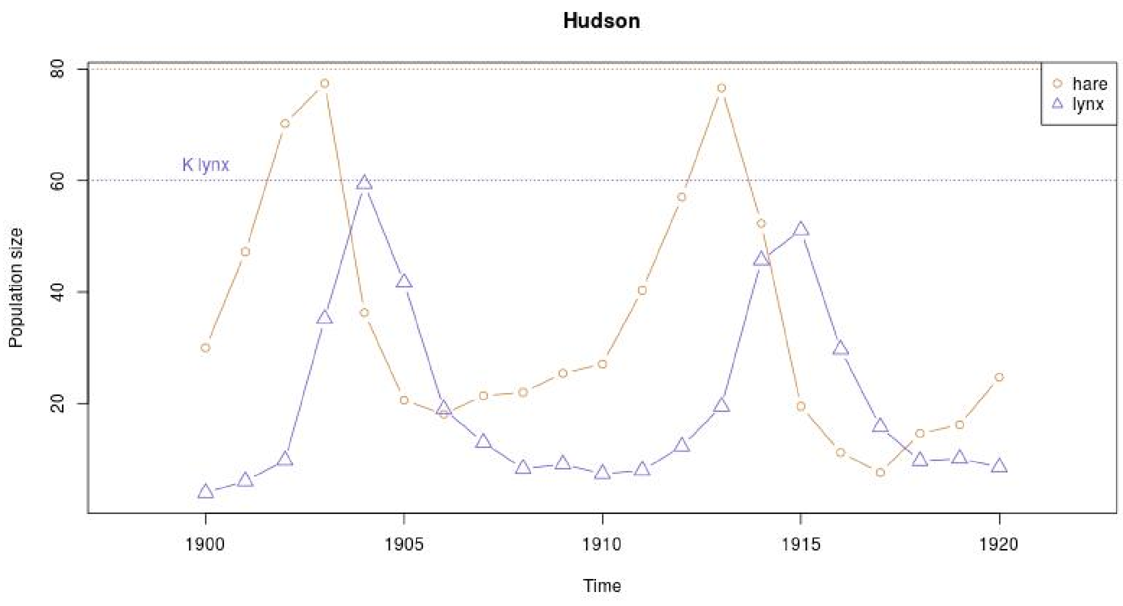

Figure 2: **Plot of the "hudson" community**, composed of 2 populations; extracted from the "hudson" example dataset

```
#  #        1917 [ n-2 ] over [ [ n-3 ] ] 0.9069767 0.2784091
#  #        1916 [ n-3 ] over [ [ n-4 ] ] 2.1322314 0.5925926
#  #        1915 [ n-4 ] over [ [ n-5 ] ] 1.7621359 1.4244604
#  #        1900         (n1 over [ n0 ]) 0.6558704 1.1744186
#
#
#  [[5]] - Biotic capacity:
#    Reached biotic capacity at [ n ]        hare               lynx
#  #                                   30.875 % 14.3333333333333 %
#
#  - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

Although there is a simple way to represent a community, this package provide another tool to visualize the interactions between two populations, with a line following the time period. It is however important to note that it is less open to modification of color (Figure 4).

```
com_plot(hudson)
```

Note: for an easier utilisation of a community in different analysis, populations can be quickly assessed within the condition that their labels are known to the user. These two commands return the same thing, the lynx population of the hudson community example dataset.

```
hudson$populations$lynx
hudson$lynx
```

1.3 Interaction matrix

Even if the simulation is done with one population or multiple populations which don't interact, an interaction matrix must be provided. It can be created by hand or imported from a vector. However, as it is important to have a correct interaction matrix, it may be easier to build it by hand. Like all other objects, it's better to check if your object is correctly formated before using it in simulations.

To remind every user, the interaction matrix is thought to use negative value for negative interaction,

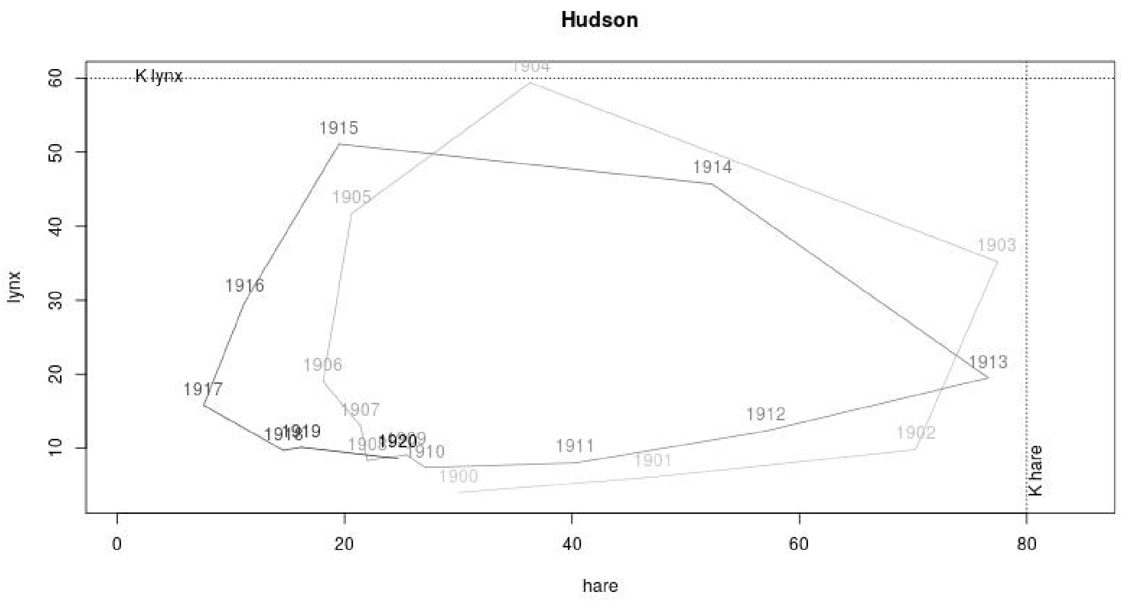

Figure 3: **Plot with the com_ plot() function of the "hudson" community**, composed of 2 populations; extracted from the "hudson" example dataset

but this also depend on the functions you provide, as they take these values in account. This information is similar with the way interactions works, from the column value to the row value or the other way around. As this system offer great opportunities to the user, it is important to keep in mind that many issues can be created with different ways of building the functions and the interactions matrix.

```
hudson_int <- interactions(nb_pop  = 2, x.interactions = c(0.0006,-0.09),
                           labels = names(hudson$populations), verbose = TRUE)
is_interactions(hudson_int)

#  [1] TRUE

wasteland_int <- c(0.2,0,-2,0.5,0,-0.5)
is_interactions(wasteland_int)

#  [1] FALSE

wasteland_int <- as_interactions(wasteland_int)
is_interactions(wasteland_int)

#  [1] TRUE
```

1.4 BeePODYNA

Last object to regroup all informations, the beepodyna object is composed of a community, a matrix of interactions between the populations of the community proposed and the functions to use in further simulations.

```
hudson_logist <- beepodyna(label = "hudson_logist", community = hudson, interactions = hudson_int,
                           functions = c(logistic_growth,logistic_growth), verbose = TRUE)
```

It also have multiple attributes, which can be assessed by different functions.

```
label(hudson_logist) ; length(hudson_logist)

#  [1] "hudson_logist"
#  [1] 2
```

2 Simulations

The final point of this package is to simulate the effect of time over a community, based on the reiteration of interactions between populations.

2.1 Running a simulations

Different types of simulation can be produced. A one step iteration is possible, or multiple iterations (which is a loop of the one step process).

```
hudson_logist <- simulate_onestep(hudson_logist)
hudson_logist <- simulate_n_pop_dynamic(hudson_logist, n = 20)
```

2.2 Different models

As seen before, a great variety of models can simulated within this frameworks. But this package only provide basic functions for simulating a population dynamic. There is a exponential growth function (`exponential_growth()`, see equation 1) and a logistic growth function (`logistic_growth()`, see equation 2).

$$size_{t+1} = size_t \cdot (1 + r) \tag{1}$$

$$size_{t+1} = size_t + size_t \cdot r \cdot (1 - size_t / K) \tag{2}$$

with : ***r*** *the growth rate and* ***K*** *the capacity.*

Any user can import it's own function, if such functions respect some rules that will be explained here. As you can see below, every function used in simulations used 3 variables : 'pop', 'community', 'interactions'. This is obligatory even if these variables are not used in the code after (as seen with `logistic_growth()`, which don't rely on the interactions matrix). Other obligatory usage is the community object it must return.

```
logistic_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$size)]
#
#    new_size <- last_size + last_size * target_pop$growth_rate * (1 - last_size / target_pop$capacity)
#    new_time <- last_time + 1
#
#    target_pop$size <- append(target_pop$size, new_size)
#    target_pop$time <- append(target_pop$time, new_time)
#
#    community$populations[[pop]] <- target_pop
#
#    community
#  }
#  <bytecode: 0x55e2c6c51050>
#  <environment: namespace:BeePODYNA>
```

A function modify a population at a time in the community given, and the simulation run each population at a time before going further if there is multiple iterations. This must be taken into account if the rythm of interaction or growth is not the same among populations : the time step is the same in all populations.

Below is an example of a function based on a prey-predator interactions. In this example the two populations depend on different function to compute their $size_{t+1}$ (see equation 3). Therefore, you must program 2 different functions because the dynamics are different. *Note : this system of equations works with only two populations. Therefore users must be precautious when coding such functions and creating objects than will be used with.*

$$\begin{cases} prey_{t+1} = prey_t + prey_t \cdot r \cdot (1 - prey_t/K) + int_{pred/prey} \cdot pred_t \\ pred_{t+1} = pred_t + pred_t \cdot r \cdot (1 - pred_t/K) + int_{prey/pred} \cdot prey_t \end{cases} \tag{3}$$

with : ***r*** *the growth rate and* ***K*** *the capacity.*

```
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 + last_size * target_pop$growth_rate * (1 - last_size / target_pop$capacity) + int * last
  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 + last_size * target_pop$growth_rate * (1 - last_size / target_pop$capacity) + int * last
  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
}

# creating the beepodyna and simulating the community
hudson_lotka <- beepodyna(label = "hudson_lotka", community = hudson, interactions = hudson_int,
  functions = c(lotka_prey,lotka_pred), verbose = TRUE)

plot(simulate_n_pop_dynamic(hudson_lotka,10)$community, col = c("tan3","slateblue3"),
     pch = c(1,2), xlim = c(1910,1930), ylim = c(0,100))
legend("bottomright", c("hare","lynx"),col = c("tan3","slateblue3"), pch = c(1,2))
```

3 Dependances

This package depend on very few packages, which are naturally loaded when installing R. Please checks for the correct installation and update of the following package before reporting troubleshootings.

```
c("R.utils","grDevices","graphics") %in% installed.packages()

#  [1] TRUE TRUE TRUE
```

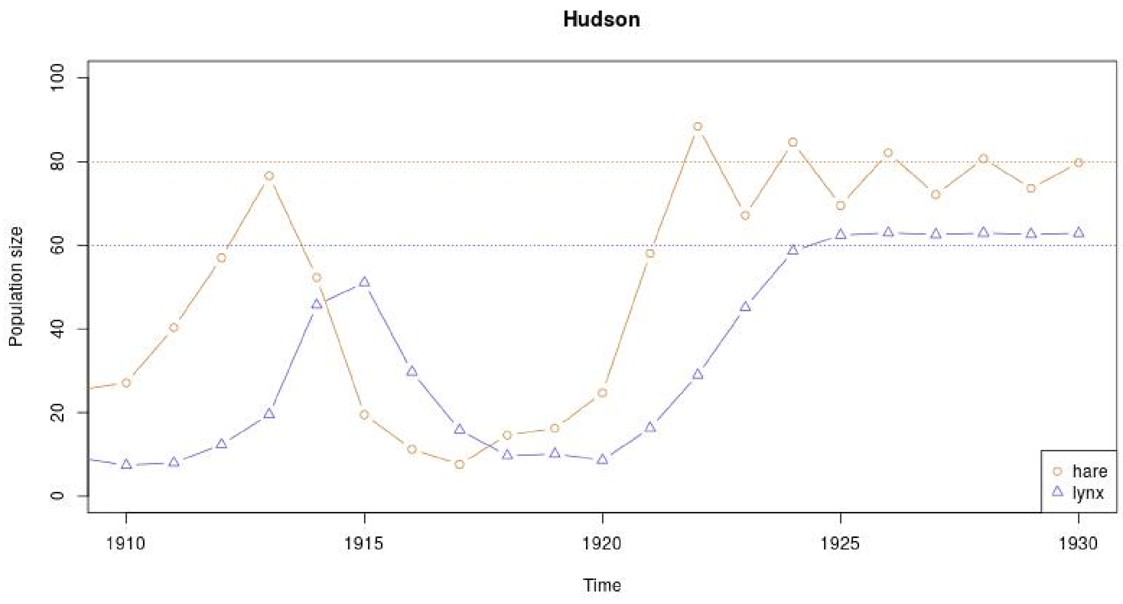

Figure 4: **Simulation of the 'hudson' community with an interaction taking capacity into account**; extracted from the "hudson" example dataset