RoLE Model User Guide

Renata Diaz, for now

4/21/2023

Table of contents

Preface

This is a user guide for working with the RoLE model, which includes the roleR and roleShiny R packages.

Part I

Background

What is the RoLE Model? Who is involved? What are these packages?

1 Process modeling in ecology and evolution

What is process modeling? Why do we use it for eco-evo

2 Models rolled into RoLE

The RoLE model implements versions of a number of established models.

2.1 Ecological neutral theory

2.2 Double-neutral ecological and population genetic drift

2.3 Competitive coexistence

2.4 Lotka-Volterra

2.5 Island biogeography

3 RoLE Mission

3.1 Scientific vision

3.2 Operating principles

3.3 Code of Conduct

Part II

How a RoLE model works

No code, high level narrative (or visuals!) of what happens in a RoLE model.

4 Components of a RoLE model

Next we will examine what the components of the RoLE model are, what each means biologically, and how to access them within roleR's classes and structure.

4.1 Local community

A local community is an assemblage of interacting species that occur in the same location and ecosystem. For example given a series of islands where each has just one type of habitat and no barriers, each could be considered a different local community.

The 'localComm' class in roleR stores the state of what species are present in local community, their abundances and genetic sequences, and what traits individuals have. Then we can simulate how this community changes over time.

4.1.1 Individual-indexed vectors

The 'localComm' slots 'indSpecies', 'indTrait', and 'indSeqs' are vectors containing species IDs, traits, and sequences for each individual.

We can access an individual's information by indexing these vectors. for example given a 'localComm' called my_local, my_local@indSpecies[2] is the ID of the species of individual 2, and my_local@indSeqs[5] is the genetic sequence of individual 5.

4.1.2 Species-indexed vectors

The 'localComm' slots 'spAbund', 'spTrait', and 'spGenDiv' are vectors contain the abundance (number of individuals), average trait value, and genetic diversity of each species. (how is genetic diversity calculated?)

Just like with the individual-level vectors, we can access species information by indexing. For example given a 'localComm' called my_local, my_local@spAbund[2] is the abundance of species 2, and my_local@spTrait[5] is the mean trait value of all living individuals of species 5.

4.2 Metacommunity

A metacommunity is a group of local communities linked by dispersal (immigration) of organisms between them.

The 'metaComm' class in roleR contains the abundances and traits of each species in the metacommunity. This can be understood as describing the species that occur outside our focal 'localComm' local community but have the capacity to disperse (immigrate) to it. Species with higher abundance in the metacommunity are more likely to disperse to our local community.

4.2.1 Species-indexed vectors

Because 'metaComm' contains only species and not individuals it only has vectors that are species-indexed.

For example given a 'metaComm' called my_meta, my_meta@spAbund[2] is the abundance of species 2, and my_meta@spTrait[5] is the mean trait value of all living individuals of species 5.

4.3 Phylogeny

The roleR phylogeny class 'rolePhylo' contains the phylogenetic relationships between all the species in the local community AND the metacommunity.

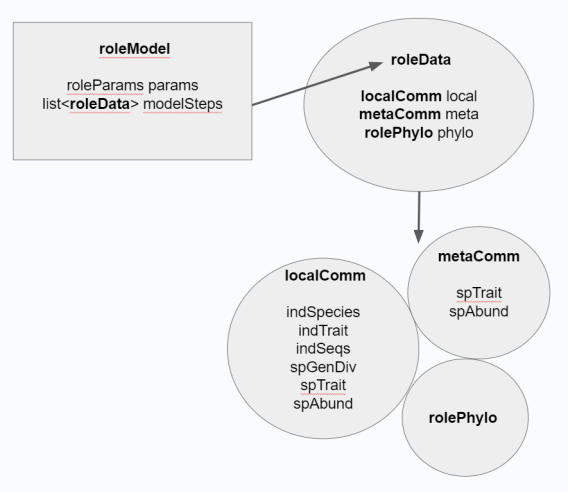
It contains various slots that you can access, but you will almost always be better off coercing it to the 'phylo' class of the R package 'ape'. Given a 'rolePhylo' called 'phy' and with the 'ape' package loaded, you can coerce it using phy <- as(phy,'phylo').

4.4 Wrapping it all together

We have seen how the local community 'localComm', the metacommunity 'metaComm', and the phylogeny 'rolePhylo' work together to represent the state of our model system. For bookkeeping, we hold them together in a 'roleData' object, which has slots 'local', 'meta' and 'phylo'.

Finally, because we want to understand not just the end state of a simulated process, but also how it has changed over time, we store each progressing 'roleData' state of the model in a list within the 'roleModel' class. The diagram below describes this structure.

```
knitr::include_graphics("components.png")
```


roleModel
roleParams params
list<roleData> modelSteps
roleData
localComm local
metaComm meta
rolePhylo phylo
metaComm
spTrait
spAbund
localComm
indSpecies
indTrait
indSeqs
spGenDiv
spTrait
spAbund
rolePhylo

5 Model inputs: Setting the stage

Having just explored the components of the model, we will now review the input parameters that "set the stage" by defining both how these components are initialized and how they change over the course of the simulation.

We will also review the default configurations of parameters available in roleR and their meanings and assumptions.

5.1 Initialization param

init_type is the model of colonization used to initialize the simulation, either "oceanic_island" or "bridge_island"

The bridge island model has the initial individuals in the local community arriving through a land bridge, while the oceanic has no bridge and is populated by a single dispersal event. Thus, for oceanic island models, all individuals are of a SINGLE species sampled proportional to meta community species abundances, while in bridge island models, individuals are sampled of MANY species proportional to their abundances.

5.2 Community shape params

individuals_local is the number of individuals in the local community

individuals_meta is the number of individuals in the metacommunity

species_meta is the number of species in the metacommunity

5.3 Birth vs. dispersal param

dispersal_prob is probability of dispersal (immigration) occurring from the metacommunity to the local community at each time step

Every time step, either birth or immigration ALWAYS happens, so the probability of birth is just 1 minus the dispersal_prob.

5.4 Speciation and extinction params

speciation_local is the probability of speciation occurring in the local community at each time step.

The new local species can be either from a birth in the local community or an immigration from the metacommunity - the aforementioned dispersal_prob param determines the chance of each of these scenarios.

speciation_meta is the rate of speciation in the metacommunity.

This is used during the simulation of the start phylogeny for metacommunity species. In addition, the sum of speciation_meta and extinction_meta is equal to the average lifetime of phylo branches, and the larger this value the less new individual traits will deviate from the parent's individual trait (if parent is in the local comm) or the metacommunity mean (if parent in the metacommunity).

extinction_meta is the rate of extinction in the metacommunity. Like speciation_meta, it is used in the initial phylogeny simulation and in relation to trait deviation.

5.5 Traits, filtering, and competition params

trait_sigma is the rate of Brownian trait evolution in the metacommunity. It determines how much the trait of a new individual deviates from its parent, i.e. how fast traits change.

env_sigma is the selectivity of the environmental filter, i.e. how strongly the environment selects which trait values are a good match for it. The larger the value, the less chance an individual will survive if it's far from the trait optimum (which is 0).

comp_sigma is the size of the competition kernel: how strongly distance from the traits of other species selects which trait values are likely to survive. The larger the value, the greater the chance an individual will survive if it is close in trait space to others

5.6 Genetics params

mutation_rate

num_basepairs

5.7 Meta params

niter is the number of iteration time steps to run the simulation for.

niterTimestep is the frequency (in numbers of iterations) at which the model state is snapshot and saved in a model's model steps object.

5.8 Pre-set configurations

untbParams

6 Initialization

6.1 The initial state of the model

Let's review how the model is initialized under the hood.

First, a phylogeny for the species in the metacommunity is simulated based on the rates of speciation and extinction in the metacommunity (speciation_meta and extinction_meta parameters).

Second, abundances for the species in the metacommunity are simulated using a log series model.

Third, traits for the species in the metacommunity are simulated across the phylogeny using a Brownian motion model.

Finally, the colonization from our metacommunity to the new local community is simulated (depending on the init_type parameter, which chooses the model).

In the oceanic island case, a single dispersal event of just one species occurs, for example through a vegetation raft. This means that all individuals in the local will start as that one species.

In the bridge island case, many species arrive at once through a land bridge, making the initial individuals a variety of different species.

7 Timestep-to-timestep

What happens when the model actually runs? To really understand the model, we have to understand its steps and the logic and implications of each one.

Each model run is composed of several timesteps, and during each step events like death, birth, immigration, and speciation happen to individuals or species in the model.

These events happen randomly based on the model parameters, causing the state of the local and meta communities to change as the model is run.

7.1 Deaths

Every timestep, no matter what, an individual in the local community dies.

If we are using a neutral model (parameter neut_delta = 1), a completely random individual is always chosen for death. This is the definition of neutral evolution!

But if the model is not neutral, an individual's chance of death (or ability to survive in its environment) is based on its ability to compete with other individuals and survive in its environment.

The closer the trait of the individual is to the environmental optimum of 1, a perfect match to its environment, the more likely it is to survive.

But at the same time, the more individuals that have similar traits to it, the more the resources each needs will overlap, and this competition will make it harder to survive. (Add more on theory here).

An outcome of RoLE's individual-based model framework is that intraspecific competition is modeled, and even emphasized. Since individuals of the same species typically have similar traits, they will compete more. (could add info on what comp and env sigma do, but this info might belong in parameters section).

7.2 Birth and dispersal

Every timestep, no matter what, EITHER birth or dispersal causes a new individual to appear in the local community.

Birth is when an individual in the local community reproduces to make a single offspring.

Dispersal is when an individual is born in the metacommunity and immigrates from the metacommunity to the local community.

The chance of each is based on the dispersal_prob param, which is simply the chance of dispersal (making 1-dispersal_prob the chance of birth).

7.3 Speciation

Speciation in RoLE works by making the individual that was just birthed or immigrated a new species.

Speciation may or may not happen each timestep, with a chance equal to speciation_local.

Realistically speciation_local is an extremely small value like 0.001 to capture the rarity of speciation events over time.

When speciation happens, a new tip is added to the phylogeny for the new species.

7.4 Trait change

When birth, dispersal, or speciation happens, the new individual gets a new trait deviating from the traits of its parent.

This deviation depends on the trait_sigma parameter - the larger the trait_sigma the more child traits will deviate from parent traits. (this is also based on speciation meta and extinction meta, which could be worth explaining)

8 On model completion: backwards-time population genetics

9 Extracting model results

We have seen that the 'runRole()' function runs an initialized model to completion. What kinds of results can we get from a run model?

A useful way to store model results is in a dataframe where each row is a saved model state and each column contains different info about that state.

We use the 'getSumStats()' function to do this for a model. The first argument is the model itself, while the second argument is a named list of functions. Each function expresses one way that you want to summarize the model state, and the name of that function in the list expressed the name to give to the column in your new dataframe.

```
my_stats <- getSumStats(my_model,list(rich = richness, hill_abund=hillAbund))
```

The above code supplies the roleR built-in functions 'richness' and 'hillAbund', with data extracted using these functions to be placed in columns called 'rich' and 'hill_abund' respectively.

But what are these functions, and how can you write your own specialized ones?

9.1 Raw community state

Several functions simply get raw information about the community state.

'rawAbundance'

'rawSpAbundance'

'rawSppID'

'rawTraits'

'rawGenDiv'

'rawSeqs'

'rawBranchLengths'

'rawApePhylo'

9.2 Summary statistics

The remaining built-in functions are more complex, summarizing information from model components rather than just returning the raw components.

'richness'

'hillAbund'

'hillGenetic'

'hillTrait'

'hillPhylo'

9.3 Writing custom getSumStats functions

Part III

Running RoLE models

Ok, how do I actually play with RoLE?

10 Installation

10.1 Compiled binaries

We recommend installing our pre-compiled binaries from r-universe.

```
install.packages('roleR', repos = c('https://role-model.r-universe.dev', 'https://cloud.r-
```

10.2 From source

You can also install from source from our GitHub repository, but it is not recommended.

```
library(remotes)
remotes::install_github("role-model/roleR",dependencies=TRUE)
```

11 Your first RoLE model

Let's go through the steps of running simple neutral model.

11.1 Model specification

First we make a roleParams object to contain the parameters for our model. The roleParams constructor has defaults for all 19 params, here we just change the number of iterations and how often to save the model state.

```
my_params <- roleParams(niter=10000,niterTimestep = 100)
```

Then we make a roleModel object using these params.

```
my_model <- roleModel(my_params)
```

11.2 Running

To run, we call runRole on the model.

```
my_model <- runRole(my_model)
```

11.3 Results and interpretation

Finally let's visualize a simple result - the species richness of the local community over time.

```
my_model_stats <- getSumStats(my_model)

library(ggplot2)
ggplot(my_model_stats, aes(iteration, richness)) +
    geom_line()
```

12 RoLE Experiments

12.1 RoLE Experiments

13 Data sharing and reproducibility

13.1 Saving model results

It is good practice to save your models and experiments so that other researchers can understand and repeat your results. Before saving you will want to attach context metadata to your object using the 'setContext' function.

```
my_model <- setContext(my_model,author="Jacob Idec",datestring="5/4/2023",description="A m
```

Then you can save using the 'writeRole' function

```
writeRole(my_model,filename="my_model")
```

13.2 Stochasticity and non-repeatability

May want to allow setting a seed in roleModels and experiments - note that this doesn't work currently because the random number generator is on the Cpp side

14 RoLE models at scale

14.1 Parallel and cluster computing

When running a roleExperiment, you can run each model on a different processor core in Windows, Mac, and Linux by using the 'cores' argument of runRole.

```
my_expr <- runRole(my_expr, cores=5)
```

Part IV

Use cases

Overview of main (forseen) use cases for RoLE.

15 RoLE Models for In-Silico Exploration

Generate and test hypotheses for how different types of conditions/interventions affect model outcomes.

16 Linking pattern to process in empirical data

16.1 Many-to-one mapping

16.2 Likelihood-free inference

16.3 Worked example

17 RoLE in the Classroom

Tips, tricks, and sample curricula from using RoLE as a teaching tool.

Part V

Troubleshooting

What to do when it doesn't run.

18 Contact

We respond to GitHub issues!

Part VI

Acknowledgements

We gratefully acknowledge funding support from NSF awards DBI-2208901 to Renata Diaz and DBI-2104147 to the RoLE model team