EcologicalNetworks.jl - analysing ecological networks

Timothée Poisot [1,2] Zacharie Belisle [1]

1: Université de Montréal, Département de Sciences Biologiques; **2**: Québec Centre for Biodiversity Sciences

Abstract: Networks are a convenient way to represent many interactions among ecological entities. The analysis of ecological networks is challenging for two reasons. First, there is a plethora of measures that can be applied (and some of them measure the same property); second, the implementation of these measures is sometimes difficult. We present `EcologicalNetwork.jl`, a package for the *julia* programming language. Using a layered system of types to represent several types of ecological networks, this packages offers a solid library of basic functions which can be chained together to perform the most common analyses of ecological networks.

Keywords:
ecological networks
Julia
graph theory

Correspondence to Timothée Poisot – `timothee.poisot@umontreal.ca`

Background

The analysis of ecological networks is an increasingly common task in community ecology and related fields (Delmas et al. 2018). Ecological networks provide a compact and tractable representation of interactions between multiple species, populations, or individuals. The methodology to analyse them, grounded in graph theory, scales from small number of species to potentially gigantic graphs of thousands of partners. The structural properties derived from the analysis of these graphs can be mapped onto the ecological properties of the community they depict. Because there is a large number of questions one may seek to address using the formalism of networks (Poisot et al. 2016b), there has been an explosion in the diversity of measures offered. As such, it can be difficult to decide on which measure to use, let alone which software implementation to rely on.

At the same time, the recent years have seen an increase in the type of applications of network theory in ecology. This includes probabilistic graphs (Poisot et al. 2016a), investigation of species functional roles in the network (Baker et al. 2014), comparison of networks across space and time (Poisot et al. 2012) and on gradients (Pellissier et al. 2017), to name a few. As the breadth and complexity of analyses applied to ecological networks increases, there is a necessity to homogenize their implementation. To the ecologist wanting to analyse ecological networks, there are a variety of choices; these include `enaR` (Borrett & Lau 2014) for food webs, `bipartite` (Dormann et al. 2008) and `BiMat` (Flores et al. 2016) for bipartite networks, and more general graph-theory libraries such as `networkx` (Hagberg et al. 2008) and `igraph` (Csardi & Nepusz 2006), which are comprehensive but may lack ecology-specific approaches and measures. Additional packages are even more specific, such as `bmotif` for bipartite motifs enumeration (Simmons et al. 2018), or `pymfinder` (Mora et al. 2018). Most of these packages are focused on either food webs or bipartite networks, and therefore do not provide a unified ecosystem for users to develop their analyses in; more general libraries come the closer, but

they require a lot of groundwork before they can be effectively used to conduct ecological analyses. There is a gap in the current software offering.

In this manuscript, we describe `EcologicalNetworks`, a package for the *Julia* programming language (Bezanson et al. 2017). *Julia* is rapidly emerging as a new standard for technical computing, as it offers the ease of writing of traditional interpreted languages (like *R* or *python*) with up to *C*-like performance. More importantly, code performance can be achieved by writing *only* pure-*Julia* code, *i.e.* without having to write the most time-consuming parts in other languages like *C* or *C++*. This results in more cohesive, and more maintainable code, to which users can more easily contribute.

The goal of this package is to provide a general environment to perform analyses of ecological networks. It offers a hierarchy of types to represent ecological networks, and includes common measures to analyse them. This package has been designed to be easily extended, and offers small, single-use function, that can be chained together to build complex analyses. The advantage of this design is that, rather than having to learn the interfaces (and options) of many different packages, the analyses can be seemlessly integrated in a single environment – this solves the problem identified by Delmas et al. (2018), namely that software for ecological network research is extremely fragmented. In addition, many measures and analyses of network structure are likely to re-use the same basic components. Consolidating the methodology within a single package makes it easier to build a densely connected codebase. Whenever possible, we have also overloaded methods for the code *Julia* language, so that the code feels idiomatic. We showcase the usage of `EcologicalNetworks` through a number of simple applications: null-hypothesis significance testing, network comparison, modularity optimisation, random extinctions, and the prediction of missing interactions.

Methods and features

Installation instructions for *Julia* itself are found at https://julialang.org/downloads/ – this manuscript specifically describes version 1.0.0 of `EcologicalNetworks.jl` (currently unreleased), which works on the 0.7 and 1.0 releases of *Julia*. The code is released under the MIT license. Functions in the package are documented through the standard *Julia* mechanism (`?connectance`, for example), and a documentation is available online at http://poisotlab.io/EcologicalNetworks.jl/latest/. `EcologicalNetworks.jl` can currently be downloaded anonymously its *GitHub* repository, by first entering the package mode of the *julia* REPL (`]`), and typing:

```
add https://github.com/PoisotLab/EcologicalNetworks.jl#develop
```

In this section, we will list the core functions offered by the package, discuss the type system, and highlight the most important aspects of the user interface. This manuscript has been written so that all examples can be reproduced from scratch.

Overview of package capacities

The `EcologicalNetworks` package offers functions to perform the majority of common ecological networks analyses – we follow the recommendations laid out in Delmas et al. (2018). The key functions include species richness (`richness`); connectance (`connectance`) and linkage density (`linkage_density`); degree (`degree`) and specificity (`specificity`); null models (`null1`, `null2`, `null3in`, `null3out`); constrained network permutations (`shuffle`); random networks (`rand`); nestedness (η and `nodf`); shortest path (`number_of_paths`, `shortest_path`); centrality measures (`centrality_katz`, `centrality_closeness`, `centrality_degree`); motif counting

(`find_motif`); modularity (`Q`), realized modularity (`Qr`), and functions to optimize them (`lp` and `salp` for label propagation with or without simulated annealing, `brim`); β-diversity measures (`βs`, `βos`, `βwn`); trophic level analysis (`fractional_trophic_level`, `trophic_level`); complementarity analysis (`AJS`; `EAJS`; `overlap`). These functions use the rich type system to apply the correct method depending on the type of network, and rely on a simple user-interface to let users chain them together (as explained in the next section). This package is *not* a series of wrappers around functions, that would provide ready-made analyses. Instead, it provides functions which can be chained, to let users develop their own analyses.

Type system

Networks are divided according to two properties: their partiteness (unipartite and bipartite), and the type of information they contain.

Partiteness	Int. strength	Type	Interactions
Unipartite	Binary	`UnipartiteNetwork`	`AbstractBool`
	Quantitative	`UnipartiteQuantitativeNetwork`	`Number`
	Probabilistic	`UnipartiteProbabilisticNetwork`	`AbstractFloat`
Bipartite	Binary	`BipartiteNetwork`	`AbstractBool`
	Quantitative	`BipartiteQuantitativeNetwork`	`Number`
	Probabilistic	`BipartiteProbabilisticNetwork`	`AbstractFloat`

All of these types share a `Matrix` field `A` containing the adjacency matrix, and either one `Vector` field `S` (unipartite case) or two `Vector` fields `T` and `B` (bipartite case) containins the species (in the bipartite case, the species are divided between the top layer `T` and bottom layer `B`). The species can be represented as `String` or `Symbol`, with support for more types anticipated. In addition, there are a number of type unions (fig. 1). The purpose of these types is to help users write functions that target the correct combination of networks.

Fortunately, end-users will almost never need to understand how data are represented within a type – the package is built around a number of high-level interfaces (see the next section) to manipulate and access information about species and interactions. The type system is worth understanding in depth when writing additional functions for which performance is important. But in the context of other analyses, the functions described in the next section should be used.

Interface

There are a number of high-level functions to interact with networks. An array of the species can be returned with `species(N)`, and this can further be split between rows and columns with, respectively, `species(N,1)` and `species(N,2)`. Another high-level function is `interactions`, which returns a list of tuples, one for each interaction in the network.

```
N = web_of_life("A_HP_001")
first(interactions(N))
```

```
(from = "Ctenophthalmus proximus", to = "Microtus arvalis", strength = 2)
```

We also implement an iteration protocol (`for interaction in network ...`), which returns the same objects as the `interactions` function.

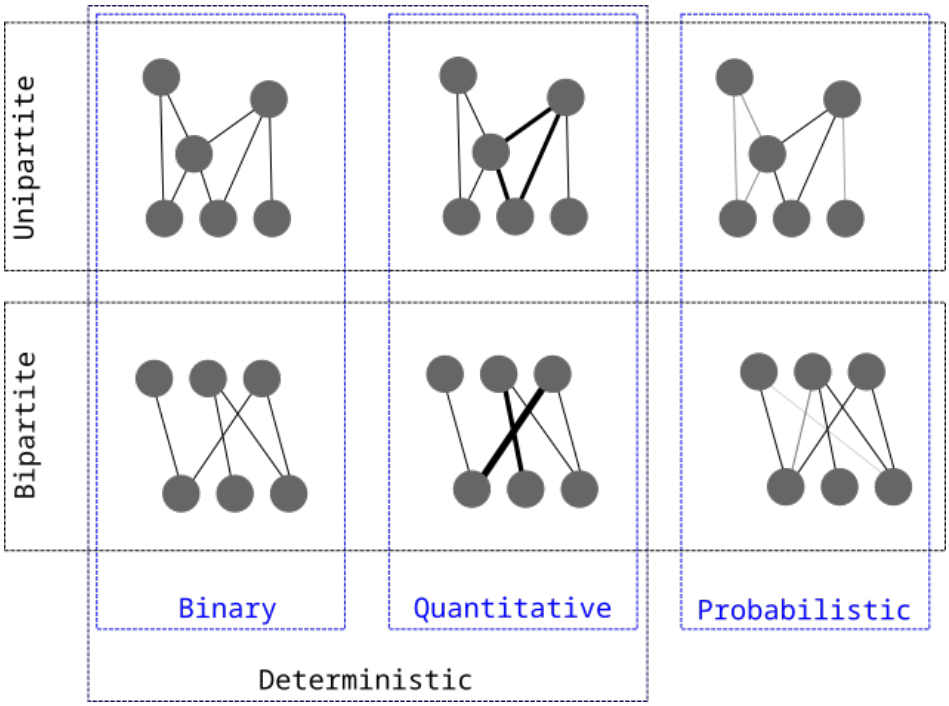

Figure 1 Union types defined by `EcologicalNetworks` – all networks belong to the `AbstractEcologicalNetwork` supertype. The ability to target specific combinations of types allows to write the correct methods for multiple classes of networks at once, while being able to specialize them on specific types.

The network itself can be accessed as an array, either using the *position* of the species (which is not advised to do as a user, since species are identified by names/symbol), or their names. This can be used to get the value of an interaction:

```
N["Ctenophthalmus proximus", "Microtus majori"]
```

```
27
```

There is a shortcut to test the *existence* of the interaction:

```
has_interaction(N, "Ctenophthalmus proximus", "Microtus majori")
```

```
true
```

Indexing can also be used to look at a subset of the network, in which case a new network is returned:

```
Ctenophthalmus = filter(x -> startswith(x, "Ctenophthalmus"), species(N; dims=1))
Apodemus = filter(x -> startswith(x, "Apodemus"), species(N; dims=2))
N[Ctenophthalmus, Apodemus]
```

```
5×2 bipartite quantitative ecological network (Int64, String) (L: 8)
```

When using slices, the package is not necessarily preserving the *order* of species. The package also uses ranges (the `simplify` function removes species without interactions):

```
simplify(N[Ctenophthalmus,:])
```

```
5×8 bipartite quantitative ecological network (Int64, String) (L: 23)
```

The `simplify` function will return another network, but there is a `simplify!` variant which will edit the network *in place*. Finally, we can get the set of predecessors or successors to a species – for example, the parasites of *"Apodemus sylvaticus"* are:

```
N[:,Apodemus[1]]
```

```
Set(["Ctenophthalmus inornatus", "Hystrichopsylla satunini", "Ceratophyllus
 sciurorum", "Ctenophthalmus shovi", "Megabothris turbidus", "Myoxopsylla j
ordani", "Amphipsylla georgica", "Rhadinopsylla integella", "Ctenophthalmus
 proximus", "Nosopsyllus fasciatus", "Leptopsylla segnis", "Palaeopsylla ca
ucasica", "Leptopsylla taschenbergi", "Amphipsylla rossica", "Ctenophthalmu
s hypanis", "Hystrichopsylla talpae"])
```

Whenever possible, we have overloaded base methods from the language, so that the right syntax is immediately intuitive to *Julia* users. For example, removing interactions whose intensity is below a certain threshold is done through the `isless` operation, *e.g.* we can select the sub-network made of interactions stronger than 20:

```
S = simplify(N ≥ 20)
```

Use-cases

In this section, we will use data from Hadfield et al. (2014) to illustrate a variety of network analyses – null hypothesis significance testing for nestedness, pairwise network β-diversity, modularity analysis, simulation of extinctions, and finally the application of a machine learning technique to infer possible missing interactions.

`EcologicalNetworks` comes with a variety of datasets, notably the <web-of-life.es> database. We will get the data from Hadfield et al. (2014) from this source:

```
ids = getfield.(filter(x -> occursin("Hadfield", x.Reference), web_of_life()), :ID);
networks = convert.(BinaryNetwork, web_of_life.(ids));
```

Null-hypothesis significance testing

One common analysis in the network literature is to compare the observed value of a network measure to the expected distribution under some definition of "random chance". This is usually done by (i) generating a matrix of probabilities of interactions based on connectance (Fortuna & Bascompte 2006), degree distribution (Bascompte et al. 2003; Weitz et al. 2013), (ii) performing random draws of this matrix under various constraints on its degeneracy (Fortuna et al. 2010), and (iii) comparing the empirical value to its random distribution, usually through a one-sided t-test. We will illustrate this approach by comparing the observed value of nestedness (measured using the η measure of Bastolla et al. (2009)) to the random expectations under four null models. We will get the first network from the Hadfield et al. (2014) dataset to illustrate this approach:

```
N = networks[1]
```

`EcologicalNetworks` comes with functions to generate probabilistic matrices under the four most common null models – for example

```
P1 = null2(N)
```

All probabilistic networks can be used to generate random samples, by calling the `rand` function, possibly with a number of samples:

```
R1 = rand(P1, 9)
```

This allows to rapidly create random draws from a probabilistic null model, as illustrated in fig. 2.

```
Error: type ValueIterator has no field x
```

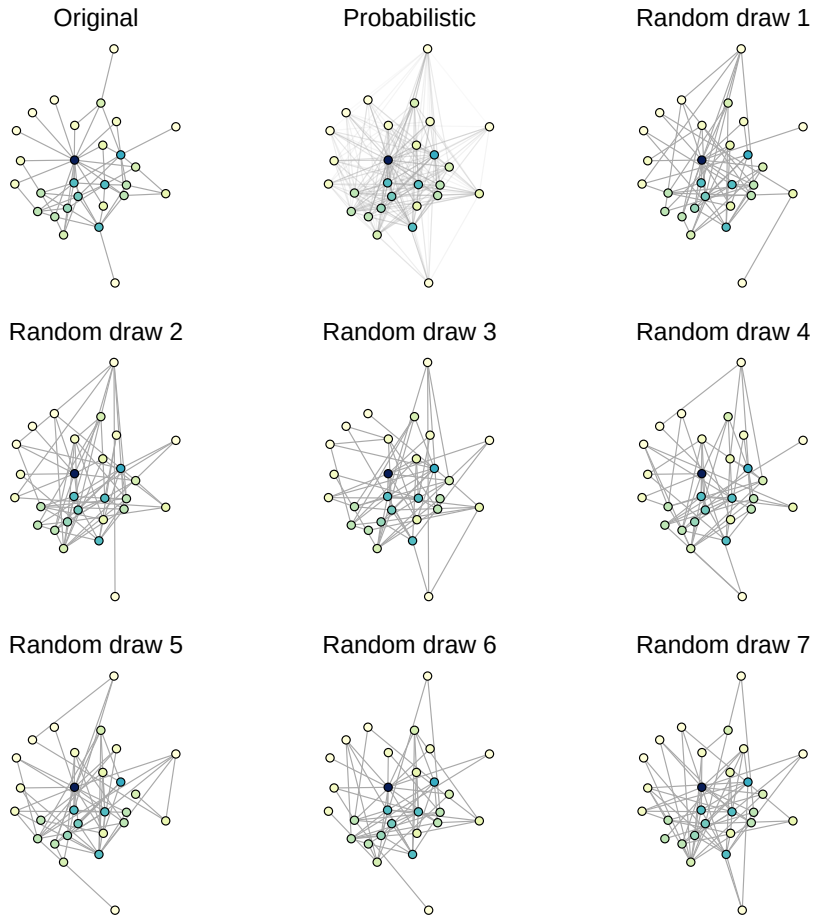

Figure 2 Illustration of the network (upper-left corner), probabilistic network generated by the null model, and of 8 random draws. The color of each node represents its degree in the original network, and the position of species is conserved across panels.

To simplify the code, we may want to wrap this into a function (note that the functions for null models accept networks of any partiteness, but they have to be binary). This function will take a network, a type of null model, and a number of replicates, and return the random draws. We will use four null models (as per Delmas et al. 2018), `null1` (all interactions have equal probability), `null2` (interactions probability depends on the degree of both species), and `null3in` and `null3out` (interactions probability depends on the in-degree or out-degree of the species). These networks are likely to have some degenerate matrices (as per Fortuna et al. (2010)), that is to say, some species end up disconnected from the rest of the network. One way to remove them is to apply a `filter`, using the `isdegenerate` function.

```
function nullmodel(n::T, f::Function, i::Integer) where {T<:BinaryNetwork}
  @assert f in [null1, null2, null3in, null3out]
  sample_networks = rand(f(n), i)
  filter!(!isdegenerate, sample_networks)
  length(sample_networks) == 0 && throw(ErrorException("No valid randomized networks; increase i ($(i))"))
  return sample_networks
end
```

```
sample_size = 5_000

S1 = nullmodel(N, null1, sample_size)
S2 = nullmodel(N, null2, sample_size)
S3i = nullmodel(N, null3in, sample_size)
S3o = nullmodel(N, null3out, sample_size)
```

This function will return the randomized networks that have the same richness as the empirical one. We can now measure the nestedness of the networks in each sample:

```
nS1 = η.(S1)
nS2 = η.(S2)
nS3i = η.(S3i)
nS3o = η.(S3o)
```

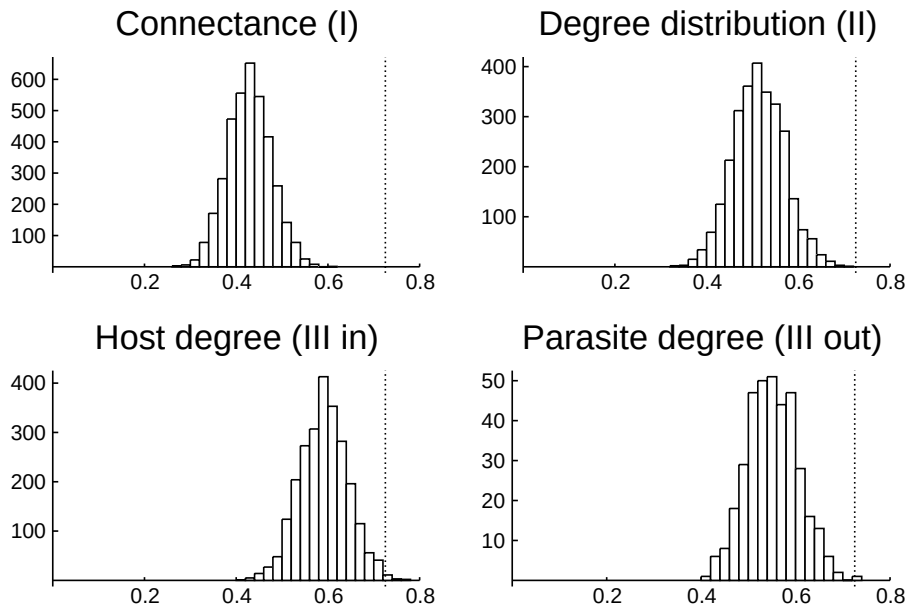

Figure 3 Distribution of nestedness values for the empirical network (solid black line) and for random draws based on four null models. This analysis is frequently used to determine whether the nestedness of an observed network is significant.

Network beta-diversity

In this section, we will use the approach of Poisot et al. (2012) to measure the dissimilarity between bipartite host-parasite networks. We use the networks from Hadfield et al. (2014), which span the entirety of Eurasia. Because these networks are originally quantitative, we will remove the information on interaction strength using `convert`. Note that we convert to an union type (`BinaryNetwork`) – the `convert` function will select the appropriate network type to return based on the partiteness. The core operations on sets (`union`, `diff`, and `intersect`) are implemented for the `BinaryNetwork` type. As such, generating the "metaweb" (*i.e.* the list of all species and all interactions in the complete dataset) is:

```
metaweb = reduce(union, networks)
```

From this metaweb, we can measure β'_{OS} (Poisot et al. 2012), *i.e.* the dissimilarity of every network to the expectation in the metaweb. Measuring the distance between two networks is done in two steps. We follow the approach of Koleff et al. (2003), in which dissimilarity is first partitioned into three components (common elements, and elements unique to both

samples), then the value is measured based on the cardinality of these components. As in Poisot et al. (2012), the function to generate the partitions are βos (dissimilarity of interactions between shared species), βs (dissimilarity of species composition), and βwn (whole network dissimilarity). The output of these functions is passed to one of the functions to measure the actual β-diversity. We have implemented the 24 functions from Koleff et al. (2003), and they are named KGLdd, where dd is the two-digits code of the function in Table 1 of Koleff et al. (2003).

```
βcomponents = βos.(metaweb, networks);
βosprime = KGL02.(βcomponents);
```

The average dissimilarity between the local interactions and interactions in the metaweb is 0.27. We have also presented the distribution in fig. 4. Finally, we measure the pairwise distance between all networks:

```
S, OS, WN = Float64[], Float64[], Float64[]
for i in 1:(length(networks)-1)
  for j in (i+1):length(networks)
    push!(S, KGL02(βs(networks[i], networks[j])))
    push!(OS, KGL02(βos(networks[i], networks[j])))
    push!(WN, KGL02(βwn(networks[i], networks[j])))
  end
end
```

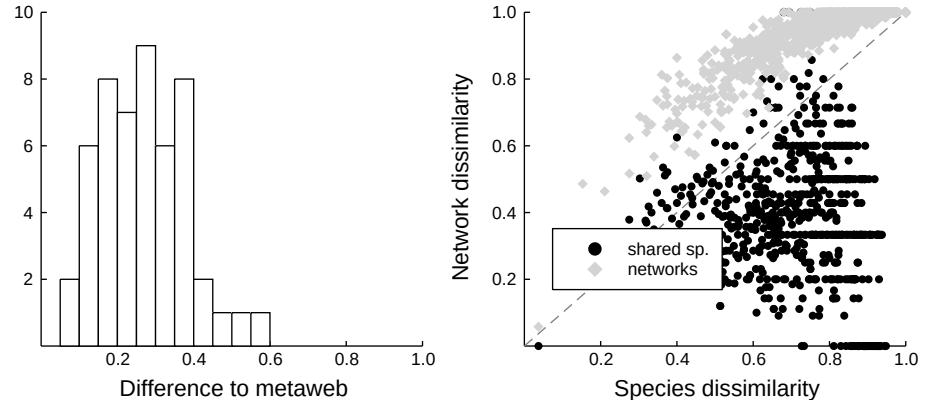

Figure 4 Left panel: values of β'_{OS} for the 51 networks in Hadfield et al. (2014). Right panel: species dissimilarity is not a good predictor of interaction dissimilarity between shared species.

Modularity

In this example, we will show how the modular structure of an ecological network can be optimized. The Finding the optimal modular structure can be a time-consuming process, as it relies on heuristic which are not guaranteed to converge to the global maximum. There is no elegant alternative to trying multiple approaches, repeating the process multiple time, and having some luck.

We will use again the first network from the Hadfield et al. (2014) dataset in this example, which has a small number of species. For the first approach, we will generate random partitions of the species across 3 to 12 modules, and evaluate 20 replicate attempts for each of these combinations. The output we are interested in is the number of modules, and the overall modularity (Barber 2007).

```
n = repeat(3:12, outer=20)
m = Array{Dict}(length(n))
```

```
for i in eachindex(n)
  # Each run returns the network and its modules
  # We discard the network, and assign the modules to our object
  _, m[i] = n_random_modules(n[i])(N) |> x -> brim(x...)
end
```

Now that we have the modular partition for every attempt, we can count the modules in it, and measure its modularity:

```
q = Q.(N, m)
c = (m .|> values |> collect) .|> unique .|> length
```

The relationship between the two is represented in fig. 5. Out of the 200 attempts, we want to get the most modular one, *i.e.* the one with highest modularity. In some simple problems, there may be several partitions with the highest value, so we can either take the first, or one at random:

```
optimal = rand(find(q .== maximum(q)))
best_m = m[optimal]
```

This partitions has 5 motifs. `EcologicalNetworks` has other routines for modularity, such as LP (Liu & Murata 2009), and a modified version of LP relying on simulated annealing.

```
Error: type ValueIterator has no field x
```

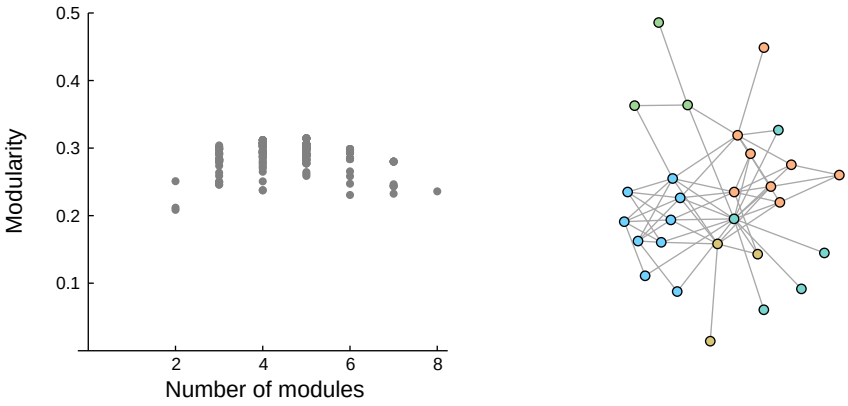

Figure 5 Left, relationship between the number of modules in the optimized partition and its modularity. Right, representation of the network where every node is colored according to the module it belongs to in the optimal partition.

Extinctions

In this illustration, we will simulate extinctions of hosts, to show how the package can be extended by using the core functions described in the "Interface" section. Simply put, the goal of this example is to write a function to randomly remove one host species, remove all parasite species that end up not connected to a host, and measuring the effect of these extinctions on the remaining network. Rather than measuring the network structure in the function, we will return an array of networks to be manipulated later:

```
function extinctions(N::T) where {T <: AbstractBipartiteNetwork}

  # We start by making a copy of the network to extinguish
  Y = [copy(N)]

  # While there is at least one species remaining...
```

```
  while richness(last(Y)) > 1
    # We remove one species randomly
    remain = sample(species(last(Y); dims=2), richness(last(Y); dims=2)-1, replace=false)

    # Remaining species
    R = last(Y)[:,remain]
    simplify!(R)

    # Then add the simplified network (without the extinct species) to our collection
    push!(Y, copy(R))
  end
  return Y
end
```

```
extinctions (generic function with 1 method)
```

One classical analysis is to remove host species, and count the richness of parasite species, to measure their robustness to host extinctions (Memmott et al. 2004) – this is usually done with multiple scenarios for order of extinction, but we will focus on the random order here. Even though `EcologicalNetworks` has a built-in function for richness, we can write a small wrapper around it:

```
function parasite_richness(N::T) where {T<:BinaryNetwork}
  return richness(N; dims=1)
end
```

```
parasite_richness (generic function with 1 method)
```

Writing multiple functions that take a single argument allows to chain them in a very expressive way: for example, measuring the richness on all timesteps in a simulation is `N |> extinctions .|> parasite_richness`, or alternatively, `parasite_richness.(extinctions(N))`. In fig. 6, we illustrate the output of this analysis on 100 simulations (average and standard deviation) for one of the networks.

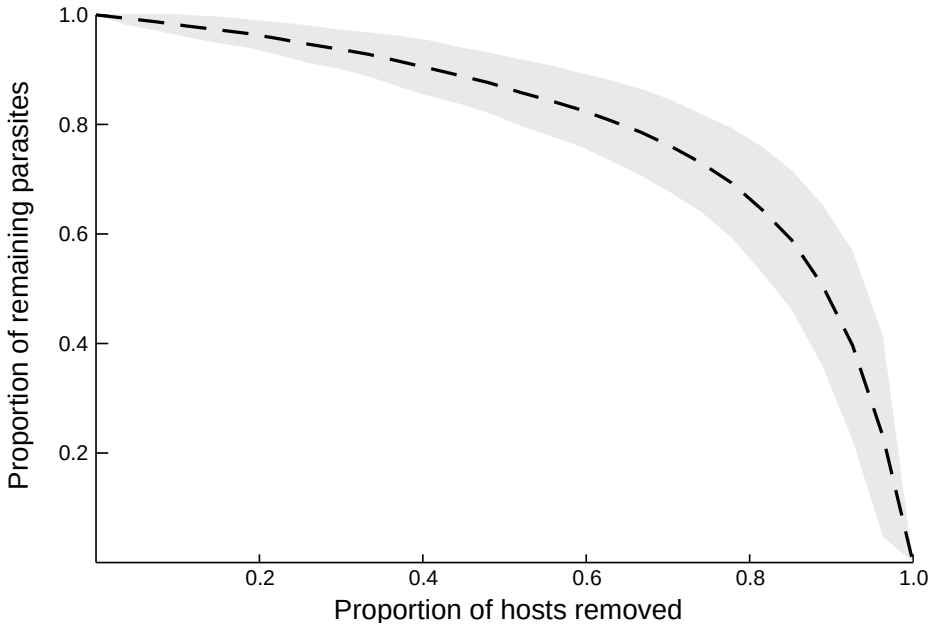

Figure 6 Output of 100 random extinction simulations, where the change in parasite richness was measured every timestep. This example shows how the basic functions of the package can be leveraged to build custom analyses rapidly.

Interaction imputation

In the final example, we will apply the linear filtering method of Stock et al. (2017) to suggest which negative interactions may have been missed in a network. Starting from a binary network, this approach generates a quantitative network, in which the weight of each interaction is the likelihood that it exists – for interactions absent from the original network, this suggests that they may have been missed during sampling. This makes this approach interesting to guide empirical efforts during the notoriously difficult task of sampling ecological networks (Jordano 2016b, 2016a).

In the approach of Stock et al. (2017), the filtered interaction matrix (*i.e.* the network of weights) is given by

$$F_{ij} = \alpha_1 Y_{ij} + \alpha_2 \sum_k \frac{Y_{kj}}{n} + \alpha_3 \sum_l \frac{Y_{il}}{m} + \alpha_4 \frac{\sum Y}{n \times m}\,, \tag{1}$$

where α is a vector of weights summing to 1, and (n, m) is the size of the network. Note that the sums along rows and columns are actually the in and out degree of species. This is implemented in `EcologicalNetworks` as the `linearfilter` function. As in Stock et al. (2017), we set all values in α to $1/4$. We can now use this function to get the top interaction that, although absent from the sampled network, is a strong candidate to exist based on the linear filtering output:

```
N = networks[50]
F = linearfilter(N)
```

We would like to separate the weights in 3: observed interactions, interactions that are not observed in this network but are observed in the metaweb, and interactions that are never observed. `EcologicalNetworks` has the `has_interaction` function to test this, but because `BinaryNetwork` are using Boolean values, we can look at the network directly:

```
scores_present = sort(
  filter(int -> N[int.from, int.to], interactions(F)),
  by = int -> int.probability,
  rev = true);

scores_metaweb = sort(
  filter(int -> (!N[int.from,int.to])&(metaweb[int.from, int.to]), interactions(F)),
  by = int -> int.probability,
  rev = true);

scores_absent = sort(
  filter(int -> !metaweb[int.from,int.to], interactions(F)),
  by = int -> int.probability,
  rev = true);
```

The results of this analysis are presented in fig. 7: the weights F_{ij} of interactions that are present locally ($Y_{ij} = $ true) are *always* larger that the weight of interactions that are absent; furthermore, the weight of interactions that are absent locally are equal to the weight of interactions that are also absent globally, strongly suggesting that this network has been correctly sampled.

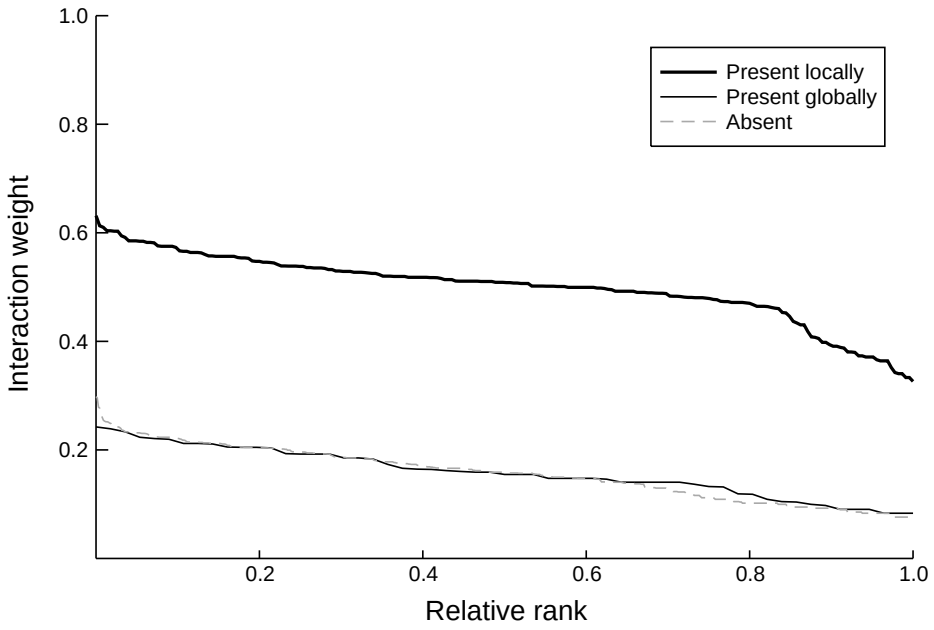

Figure 7 Relative weights (higher weights indicates a larger chance that the interaction has been missed when sampling) in one of the host-parasite networks according to the linear filter model of Stock et al. (2017).

Conclusion

We have illustrated the core approach of `EcologicalNetworks`, a *Julia* package to analyse ecological networks of species interactions. It is built to be extendable, and to facilitate the development of flexible network analysis pipelines. `EcologicalNetworks` has been designed to be robust, easy to write code with, maintainable, and fast (in that order). We think that by providing a rich system of types, coupled with specialized methods, it will allow ecologists to rapidly implement network analyses. Bug reports and features requests can be submitted at https://github.com/PoisotLab/EcologicalNetworks.jl/issues.

References

Baker et al. (2014). Species' roles in food webs show fidelity across a highly variable oak forest. *Ecography.* 38:130–9.

Barber. (2007). Modularity and community detection in bipartite networks. *Phys Rev E.* 76:066102.

Bascompte et al. (2003). The nested assembly of plant-animal mutualistic networks. *PNAS.* 100:9383–7.

Bastolla et al. (2009). The architecture of mutualistic networks minimizes competition and increases biodiversity. *Nature.* 458:1018–20.

Bezanson et al. (2017). Julia: A Fresh Approach to Numerical Computing. *SIAM Rev.* 59:65–98.

Borrett & Lau. (2014). enaR: An r package for Ecosystem Network Analysis. *Methods Ecol Evol.* 5:1206–13.

Csardi & Nepusz. (2006). The igraph Software Package for Complex Network Research. *InterJournal.* Complex Systems:1695.

Delmas et al. (2018). Analysing ecological networks of species interactions. *Biol Rev.*:112540.

Dormann et al. (2008). Introducing the bipartite Package: Analysing Ecological Networks. *R News.* 8:8–11.

Flores et al. (2016). BiMAT: a MATLAB package to facilitate the analysis and visualization of bipartite networks. *Methods Ecol Evol.* 7:127–32.

Fortuna & Bascompte. (2006). Habitat loss and the structure of plantanimal mutualistic networks. *Ecol Lett.* 9:281–6.

Fortuna et al. (2010). Nestedness versus modularity in ecological networks: two sides of the same coin? *J Anim Ecol.* 78:811–7.

Hadfield et al. (2014). A Tale of Two Phylogenies: Comparative Analyses of Ecological Interactions. *Am Nat.* 183:174–87.

Hagberg et al. (2008). Exploring Network Structure, Dynamics, and Function using NetworkX. In: Varoquaux et al., eds. *Proceedings of the 7th Python in Science Conference.* Pasadena, CA USA; pp. 11–5.

Jordano. (2016a). Chasing Ecological Interactions. *PLOS Biol.* 14:e1002559.

Jordano. (2016b). Sampling networks of ecological interactions. *Funct Ecol.*

Koleff et al. (2003). Measuring beta diversity for presence–absence data. *J Anim Ecol.* 72:367–82.

Liu & Murata. (2009). Community Detection in Large-Scale Bipartite Networks. *2009 IEEE/WIC/ACM International Joint Conference on Web Intelligence and Intelligent Agent Technology.* Institute of Electrical & Electronics Engineers (IEEE);

Memmott et al. (2004). Tolerance of pollination networks to species extinctions. *Proc Biol Sci.* 271:2605–11.

Mora et al. (2018). pymfinder: a tool for the motif analysis of binary and quantitative complex networks. *bioRxiv.*:364703.

Pellissier et al. (2017). Comparing species interaction networks along environmental gradients. *Biol Rev Camb Philos Soc.*

Poisot et al. (2012). The dissimilarity of species interaction networks. *Ecol Lett.* 15:1353–61.

Poisot et al. (2016a). The structure of probabilistic networks. Vamosi, ed. *Methods Ecol Evol.* 7:303–12.

Poisot et al. (2016b). Describe, understand and predict: why do we need networks in ecology? *Funct Ecol.* 30:1878–82.

Simmons et al. (2018). bmotif: a package for counting motifs in bipartite networks.

Stock et al. (2017). Linear filtering reveals false negatives in species interaction data. *Sci Rep.* 7:45908.

Weitz et al. (2013). Phage–bacteria infection networks. *Trends Microbiol.* 21:82–91.