ECOGRAPHY

Software notes

EcologicalNetworks.jl: analysing ecological networks of species interactions

Timothée Poisot, Zachary Bélisle, Laura Hoebeke, Michiel Stock and Piotr Szefer

T. Poisot (https://orcid.org/0000-0002-0735-5184) ✉ (timothee.poisot@umontreal.ca) and Z. Bélisle, Dépt de Sciences Biologiques, Univ. de Montréal, Montréal, Canada. TP also at: Québec Centre for Biodiversity Sciences, McGill Univ., Montréal, Canada. – L. Hoebeke and M. Stock, KERMIT, Dept of Data Analysis and Mathematical Modelling, Ghent Univ., Belgium. – P. Szefer, Faculty of Science, Univ. of South Bohemia, České Budějovice, Czech Republic.

Ecography
42: 1–12, 2019
doi: 10.1111/ecog.04310

Subject Editor: Michael Borregaard
Editor-in-Chief: Miguel Araújo
Accepted 8 August 2019

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 'EcologicalNetworks.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, graph theory, Julia

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 and 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 and 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.

In this manuscript, we describe EcologicalNetworks, a package for the 'Julia' programming language (Bezanson et al. 2017) released under the MIT license and openly developed at <https://github.com/PoisotLab/EcologicalNetworks.jl>. 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. In addition, Julia can be called from other languages: the JuliaCall package for R, or pyjulia for python, allow users of these languages to seamlessly integrate Julia code in their analyses. This incurs a significant performance cost which in our opinion justifies learning enough of Julia to proficiently perform the analyses, but is nevertheless a useful transition tool.

The goal of this package is to provide a general environment to perform analyses of ecological networks, following the best practices in the field as outlined by Delmas et al. (2018). 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 (with examples given in the online documentation), 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 seamlessly 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. If an operation on a network (e.g. what is the total interaction strength) can be meaningfully represented by an existing Julia function (e.g. sum), it will be made this way; for example, the function to re-organize interactions across a network is called shuffle (after the Random.shuffle function), and the function to randomly draw from a probabilistic network is rand. 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 ver. 0.2 of EcologicalNetworks.jl, which works on Julia releases 0.7 and 1.0. The code is released under the MIT license. EcologicalNetworks.jl can be installed by first entering the package mode of the Julia REPL (by typing]), and typing:

add EcologicalNetworks. The package can then be used with using EcologicalNetworks

Functions in the package are documented through the standard Julia mechanism (?connectance, for example), and a documentation describing the functionalities in the package is available online at <http://poisotlab.github.io/EcologicalNetworks.jl/stable/>.

We have also developed a companion package, 'EcologicalNetworksPlots.jl', which is intended to provide basic network visualisation functions (graphs and heatmaps) through the 'Plots.jl package'. EcologicalNetworksPlots.jl can be installed in the same way as any other Julia package. After updates to either packages, they can be kept up to date following the usual process from a Julia project: enter the package mode by typing] at the REPL, then type up. Both packages follow semantic versioning.

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: the code.jl distributed as part of the supplementary material can be used to generate all figures in the manuscript, and the Project.toml and Manifest.toml have the exact version of every package in the dependency chain used (Supplementary material Appendix 1).

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 (but are not limited to) species richness (richness); connectance (connectance) and linkage density (linkage_density); degree (degree) and specificity (specificity); null models (null1, null2, null3, null4); constrained network permutations (shuffle); random networks (rand); nestedness (η and nodf); shortest path (number_of_paths, shortest_path); information

theoretic analysis (`entropy`, `conditional_entropy`, `mutual_information`, `information_decomposition`); centrality measures (`centrality_katz`, `centrality_closeness`, `centrality_degree`); resilience (`symmetry`, `heterogeneity`, `resilience`); motif counting (`find_motif`); modularity (`Q`), realized modularity (`Qr`) and functions to optimize them (`lp` and `salp` for label propagation without or with simulated annealing, `brim`); β-diversity measures (`βs`, `βos`, `βwn`); trophic level analysis (`fractional_trophic_level`, `trophic_level`); complementarity analysis (`AJS`; `EAJS`; `overlap`); structural random models (`cascademodel`, `mpnmodel`, `nichemodel`, `nestedhierarchymodel`). 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. As such, we believe it is important to think of this package as a library for the analysis of ecological networks: the basic components can be chained to build entire analyses in a very flexible and expressive way.

Type system

Data on ecological networks can vary if the network is unipartite or bipartite, and as a function of the nature of the information represented by interactions (presence/absence, strength or probability). This yields six combinations of partiteness × nature of interaction. Because some measures are expressed differently depending on the specific type of network, we have designed a complete type system, summarized in the Table 1.

Though this may seem excessive, this allows the package to take advantages of Julia's dispatch, i.e. picking the correct method of a function based on the type of its arguments. As an example, the function for the number of links (`links`) uses specialized measures depending on the type of the network (i.e. quantitative, binary or probabilistic). As such, using types also protects the user: if a measure should not be applied to a particular type of network, it will not have an associated method, and will fail explicitly rather than returning a spurious result.

Unions and abstract types

In addition to the core types, there are a number of type unions (Fig. 1). For example, some algorithms will work in the same way on all non-probabilistic networks (`DeterministicNetwork`), or on all ecological networks (`AbstractEcologicalNetwork`). The purpose of these types is to help users write functions that target the correct combination of networks, by writing increasingly specific methods for types requiring them.

Internal representation of information

All 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) containing the species (in the bipartite case, the species are divided between the top layer `T` and bottom layer `B`). Users who wish to inspect the actual matrix can use `N.A` (where `N` is a network). We recommend that users do not access these objects directly; the interactions can be accessed in a richer format using `interactions`, and species can be accessed with `species`. 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 and will be sufficient.

Note that by default, species can only be represented by either a `String` or `Symbol`. This can very easily be changed by adding methods to the non-exported `is_valid_species` function. The default declaration is:

```
is_valid_species(::Type{T}) where
{T <: Any}=false
is_valid_species(::Type{T}) where
{T <: Union{Symbol,String}}=true
```

Should a user want to allow different types, like for example `MyCustomSpecies`, a simple declaration of a new method is sufficient:

```
import EcologicalNetworks:
is_valid_species
```

Table 1. Network type system.

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`

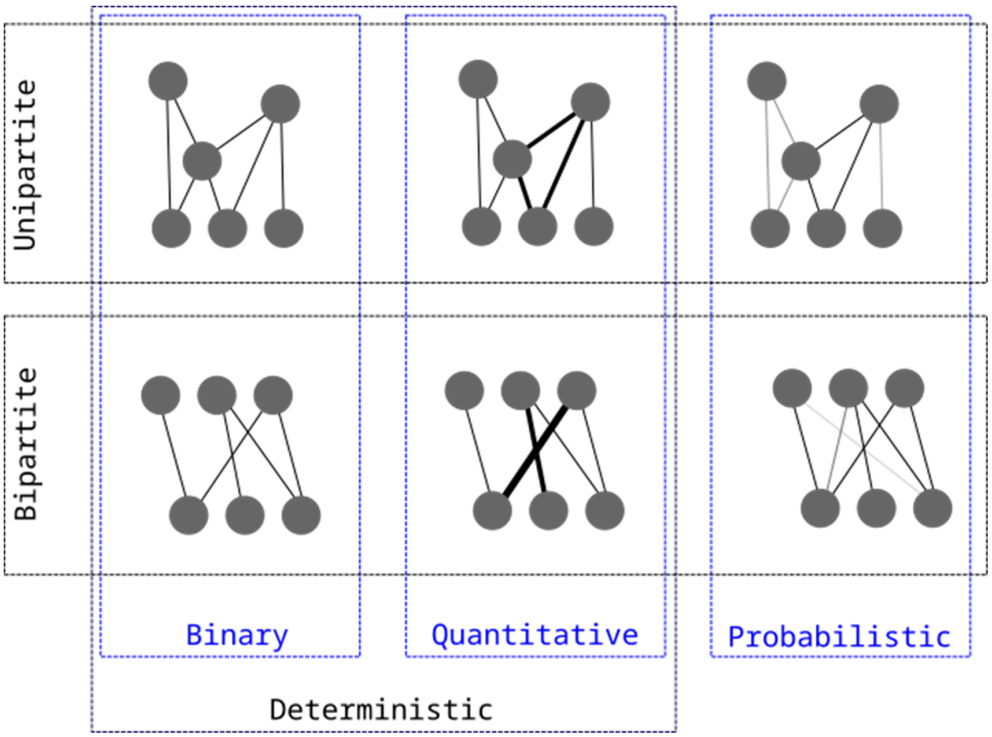

Figure 1. Union types defined by `EcologicalNetworks` – all networks belong to the `AbstractEcologicalNetwork` supertype. The ability to target specific combinations of types allows to write the correct methods for multiple classes of networks at once, while being able to specialize them on specific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

```
is_valid_species(::Type{T}) where
{T <: MyCustomSpecies}=true
```

Finally, the `eltype` function can be used to determine the types of the species and interactions. For example, we can define a simple network (in which case the species names will be generated automatically), and look at the type of its information:

```
N=UnipartiteNetwork(rand(Bool, (3, 3)))
eltype(N)
(Bool, String)
```

Interface

There are a several high-level functions to interact with networks. An array of the species can be returned with `species(N)` where `N` is the networks, and this can further be split between rows and columns with, respectively, `species(N; dims=1)` and `species(N; dims=2)` – note that this interface borrows the `dims` keyword argument found in almost all Julia functions working across multidimensional arrays, thereby making the package consistent with its language. Another high-level function is `interactions`, which returns a list of tuples, one for each interaction in the network.

In the following section, we will use an example host–parasite network to illustrate the core elements of the package interface:

```
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.

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. To illustrate this function, we will first select all parasites (`dims=1`) whose name starts with *Ctenophtalmus*:

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

```
5-element Array{String,1}:
 "Ctenophthalmus proximus"
 "Ctenophthalmus hypanis"
 "Ctenophthalmus inornatus"
 "Ctenophthalmus shovi"
 "Ctenophthalmus euxinicus"
```

We can do the same for hosts whose name starts with *Apodemus*:

```
Apodemus=filter(
     x -> startswith(x, "Apodemus"),
     species(N; dims=2)
)
2-element Array{String,1}:
 "Apodemus sylvaticus"
 "Apodemus mystacinus"
```

Finally, the network of interactions between *Ctenophtalmus* sp. and *Apodemus* sp. is given by:

```
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), where `:` means 'all nodes in this dimension' (note that the following is essentially how one would access an array in Julia):

```
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. Note that the `copy` method is also defined for networks, so that `simplify(N)` is equivalent to `simplify!(copy(N))` (and is indeed defined this way).

Finally, we can get the set of predecessors (species that establish interactions with) or successors (species to which interactions are established) 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 jordani", "Amphipsylla
georgica", "Rhadinopsylla integella",
"Ctenophthalmus proximus", "Nosopsyllus
fasciatus", "Leptopsylla segnis",
"Palaeopsylla caucasica", "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)
9×6 bipartite ecological network (Bool,
String) (L: 15)
```

Use-cases

In this section, we will use data on ectoparasites of rodents from Eurasia, reported by 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.jl` 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:

```
all_hp_data=filter(x ->
occursin("Hadfield", x.Reference),
web_of_life());
ids=getfield.(all_hp_data, :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' (e.g. as in Fig. 2). As of now `EcologicalNetworks.jl` focuses on generating binary (presence/absence of interactions) matrices, but expanding the functions for quantitative null models is feasible. This is usually done by 1) generating a matrix of probabilities of interactions based on connectance (Fortuna and Bascompte 2006), degree distribution (Bascompte et al. 2003, Weitz et al. 2013), 2) performing random draws of this matrix under various constraints on its degeneracy (Fortuna et al. 2010) and 3) 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]
18×10 bipartite ecological network (Bool,
String) (L: 61)
```

`EcologicalNetworks` comes with functions to generate probabilistic matrices under the four most common null models: `null1` for constraints on connectance, `null2` for constraints on degree distribution, `null3` with arguments

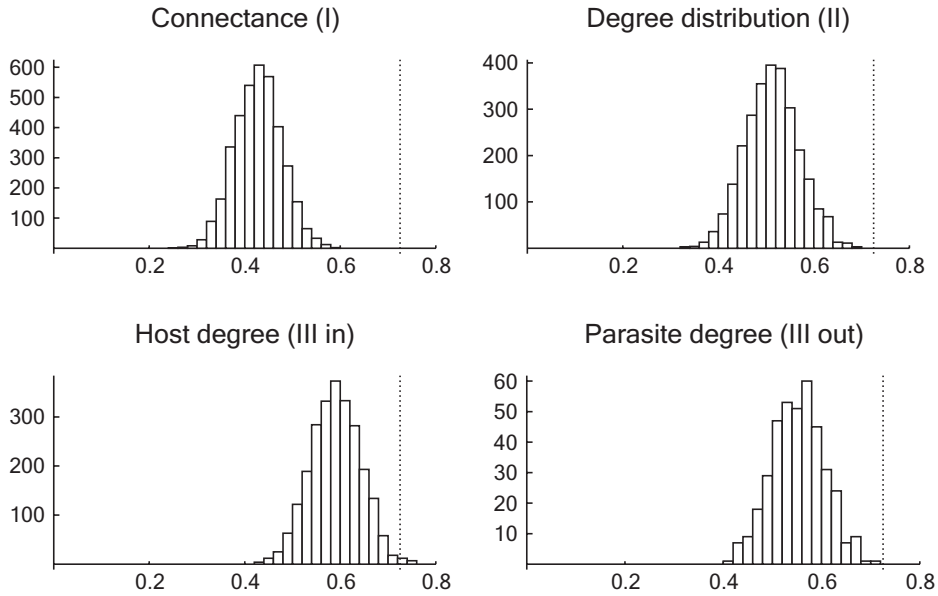

Figure 2. Distribution of nestedness values for the empirical network (solid black line) and for random draws based on four null models. This analysis is frequently used to determine whether the nestedness of an observed network is significant.

for degree distributions on either side and `null4` which considers the degree distributions on both sides as independent. Although the names are not necessarily the most descriptive, they have been used this way in the ecological networks literature, and the way they work is documented (see also Delmas et al. 2018 for an overview of their assumptions).

One can for example generate the probabilistic null model of Type II for a specific empirical network with:

```
P1=null2(N)
18×10 bipartite probabilistic ecologi-
cal network (Float64, String) (L: 60.9
9999999999999)
```

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)
9-element Array{EcologicalNetworks.
BipartiteNetwork{Bool,String},1}:
 18×10 bipartite ecological network
 (Bool, String) (L: 64)
 18×10 bipartite ecological network
 (Bool, String) (L: 56)
 18×10 bipartite ecological network
 (Bool, String) (L: 59)
 18×10 bipartite ecological network
 (Bool, String) (L: 64)
 18×10 bipartite ecological network
 (Bool, String) (L: 75)
 18×10 bipartite ecological network
 (Bool, String) (L: 64)
 18×10 bipartite ecological network
 (Bool, String) (L: 58)
 18×10 bipartite ecological network
 (Bool, String) (L: 64)
 18×10 bipartite ecological network
 (Bool, String) (L: 56)
```

This allows to rapidly create random draws from a probabilistic null model, as illustrated in Fig. 3.

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 `null3` (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}
  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
nullmodel (generic function with
1 method)
```

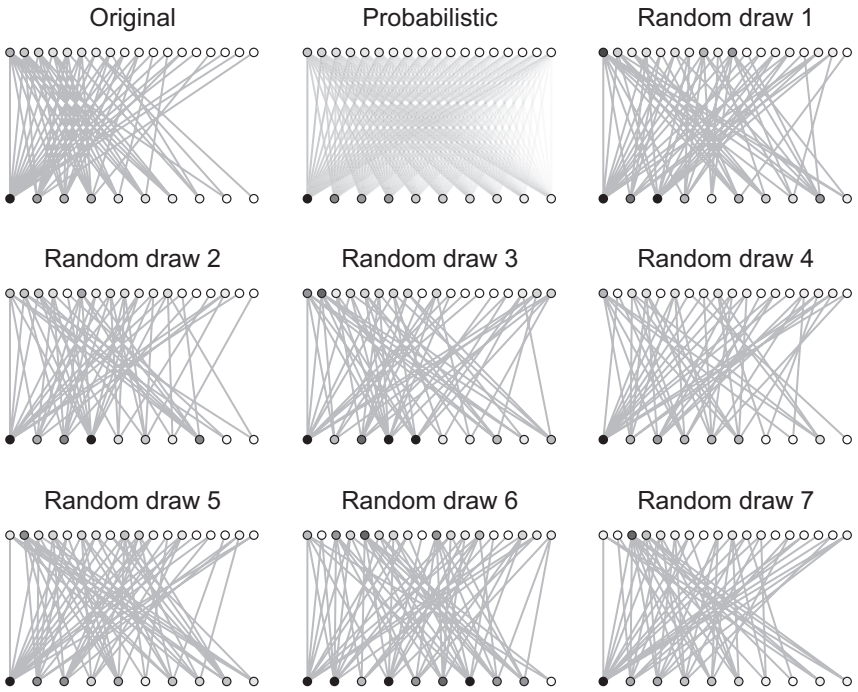

Figure 3. Illustration of the network (upper-left corner), probabilistic network generated by the null model, and of eight random draws. The color of each node represents its degree, and the position of species is conserved across panels.

We can now call this function with different null models. Note how the `null3` function takes keywords arguments, and so rather that passing it directly, one can pass an anonymous function:

```
sample_size=5_000

S1=nullmodel(N, null1, sample_size);
S2=nullmodel(N, null2, sample_size);
S3i=nullmodel(N, n -> null3(n; dims=2),
sample_size);
S3o=nullmodel(N, n -> null3(n; dims=1),
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 (as always in Julia, unicode characters can be type by using their escape sequence, which are given in the documentation; the `.` between a function name and its arguments is a shortcut to vectorize the code):

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

Network beta-diversity

In this section, we will use the approach of Poisot et al. (2012) to measure the dissimilarity between bipartite host and 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)
206×121 bipartite ecological network
(Bool, String) (L: 2131)
```

From this metaweb, we can measure βos′ (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, n) for n in
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 (because we use a symmetric measure, we only need $n \times (n-1)$ distances):

```
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
```

Modularity

In this example, we will show how the modular structure of an ecological network can be optimized. 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 times, 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 3–12 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}(undef, 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=map(x -> Q(N,x), 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(findall(q.== maximum(q)));
best_m=m[optimal];
```

This partitions has five modules. `EcologicalNetworks` has other routines for modularity, such as LP (Liu and Murata 2009), and a modified version of LP relying on simulated annealing. We can finally look at the functional roles of the species.

```
roles=functional_cartography(N, best_m)
Dict{String,Tuple{Float64,Float64}}
with 28 entries:
  "Ctenophthalmus inornatus" =>
  (-0.0764719, 0.56)
  "Chionomys nivalis"        =>
  (1.30002, 0.0)
  "Hystrichopsylla satunini" =>
  (-0.83666, 0.444444)
```

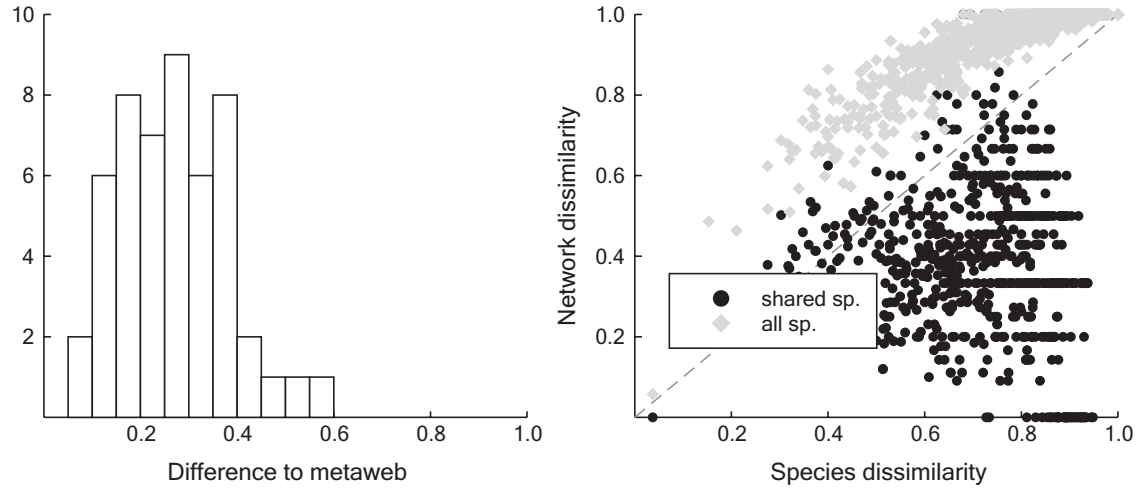

Figure 4. Left panel: values of βos′ for the 51 networks in Hadfield et al. (2014). Right panel: species dissimilarity is not a good predictor of interaction dissimilarity between shared species.

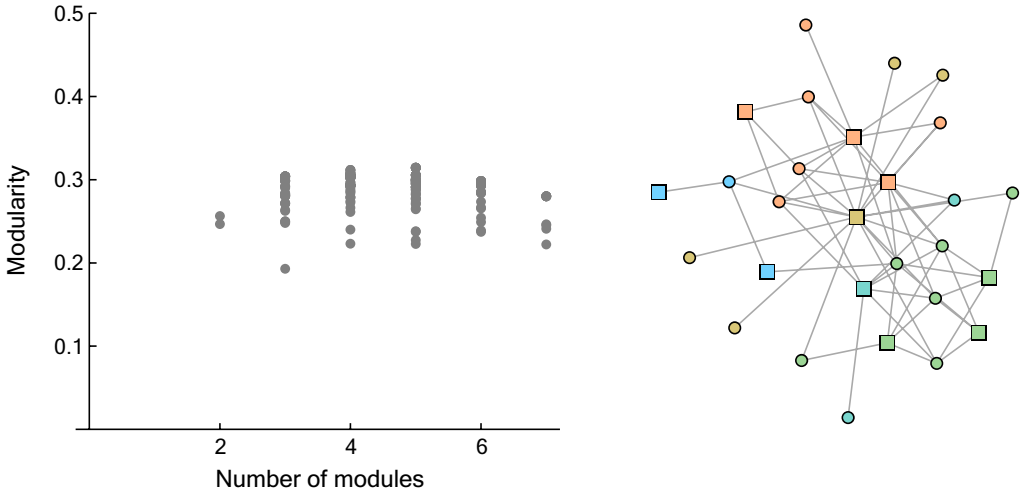

Figure 5. Left, relationship between the number of modules in the optimized partition and its modularity. Right, representation of the network where every node is colored according to the module it belongs to in the optimal partition.

```
"Ceratophyllus sciurorum"  =>
(-0.447214, 0.0)
"Apodemus sylvaticus"      =>
(1.78885, 0.726563)
"Ctenophthalmus shovi"     =>
(0.0, 0.56)
"Amalaraeus penicilliger"  =>
(-0.57735, 0.0)
"Megabothris turbidus"     =>
(-0.0764719, 0.75)
"Myoxopsylla jordani"      =>
(1.1547, 0.625)
"Amphipsylla georgica"     =>
(-1.45297, 0.5)
"Sorex araneus"            =>
(0.0, 0.0)
"Rhadinopsylla integella"  =>
(-1.45297, 0.5)
"Apodemus mystacinus"      =>
(1.67332, 0.617284)
"Microtus majori"          =>
(0.83666, 0.59375)
"Ctenophthalmus proximus"  =>
(0.0, 0.56)
"Myoxus glis"              =>
(-0.57735, 0.0)
"Nosopsyllus fasciatus"    =>
(-0.447214, 0.5)
"Leptopsylla segnis"       =>
(0.0, 0.375)
"Palaeopsylla caucasica"   =>
(-0.447214, 0.0)
⋮                          => ⋮
```

This function returns a tuple (an unmodifiable set of values) of coordinates for every species, indicating its within-module contribution, and its participation coefficient. These results can be plotted to separate species in module hubs, network hubs, peripherals and connectors (Fig. 6). Note that in the context of ecological networks, this classification (following Olesen et al. 2007) is commonly used. It derives from previous work by Guimerà and Nunes Amaral (2005) on metabolic networks, which subdivides the place in seven (rather than four) regions. For the sake of completeness, we have added the seven regions of the Guimerà and Nunes Amaral (2005) to the plot as well.

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

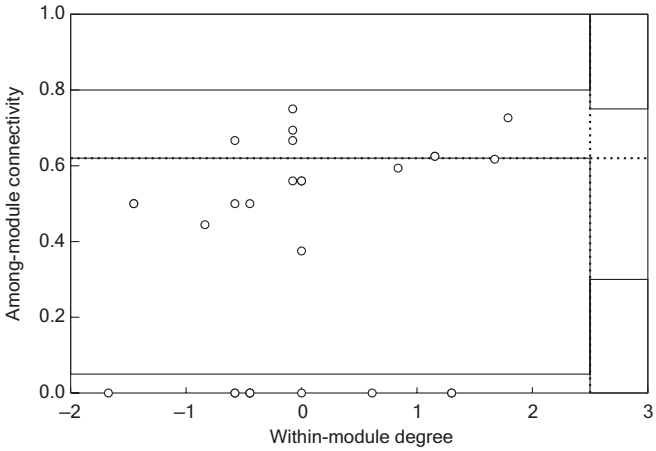

Figure 6. Functional roles of the species in the most modular partition found. All species score very low along both axes, making them 'peripherals' – this is a strong indication that the modular structure is not meaningful.

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. 7, we illustrate the output of this analysis on 100 simulations (average and standard deviation) for one of the networks.

Additionally, the sum of these three components is always equal to the logarithm of the product of the species richness of the two trophic levels:

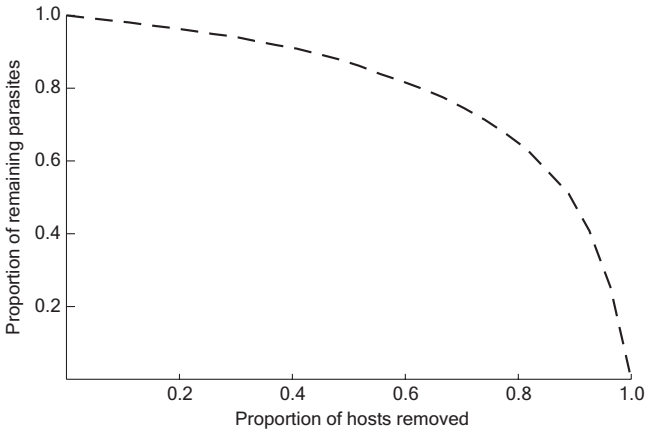

Figure 7. Output of 100 random extinction simulations, where the change in parasite richness was measured every timestep. This example shows how the basic functions of the package can be leveraged to build custom analyses rapidly.

Information theoretic indices

Notions from information theory can also be used to study the distribution of species interaction networks. The total potential entropy of an interaction network can be decomposed into three distinct components:

- D: the difference in entropy compared to a uniform distribution;
- I: the mutual information between the interaction levels;
- V: the variation of information.

The value of these components gives us insight into the structure of a network. A large deviation from the uniform distribution indicates that one or more interactions dominate the network, restricting the freedom of choice in the network. The mutual information quantifies the level of organization of the network, i.e. the limitations on possible interactions and can be seen as a measure for the efficiency of the network (Ulanowicz 2001). Finally, the variation of information quantifies the uncertainty that remains when the whole structure of the interaction network is known. A large value corresponds to a large variety of possible interaction partners. This index can be seen as a measure of the network's stability. Strong restriction of the interactions and thus freedom of choice of the species, decreases the stability of the network.

Separate functions are available to compute the different indices, however the function `information_decomposition` performs the entire decomposition at once. These calculations can be done for the joint distribution, as well as for the marginal distributions of the two interaction levels, by changing an optional argument of the function.

We can apply this to the first network of the Hadfield et al. (2014) networks. This function will return a dictionary containing the D-, I- and V-component.

```
inf_decomp_joint=information_
decomposition(N)
Dict{Symbol,Float64} with 3 entries:
  :I => 2.38221
  :D => 0.872887
  :V => 6.62907
```

The different components of the decomposition can be visualised in a barplot or a De Finetti diagram. Firstly, we note that the D-component is relatively low, both for the hosts, parasites and the network in general, indicating that the distributions do not deviate strongly from the uniform distribution. The small I-component indicates that there are not a lot of restrictions on the specificity of the interactions. Therefore, the network has a low level of efficiency. The V-component however is large. The species have a large freedom of choice, resulting in a stable network.

The sum of these three components is always equal to the logarithm of the product of the species richness of the two interaction levels:

```
log2(richness(N, dims=1) * richness(N,
dims=2))
9.884170519108435
```

This product quantifies the maximum number of possible interactions in a network, where each species of the top level interacts with every species of the bottom level, without restrictions.

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 2016a, b).

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

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

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)
```

```
35×27 bipartite probabilistic
ecological network (Float64, String)
(L: 225.99999999999997)
```

We would like to separate the weights in three: 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. 8: 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.

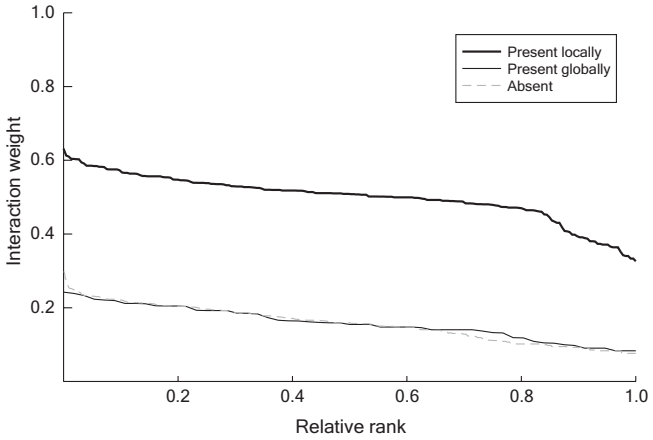

Figure 8. Relative weights (higher weights indicates a larger chance that the interaction has been missed when sampling) in one of the host–parasite networks according to the linear filter model of Stock et al. (2017).

Conclusion

We have illustrated the core approach of `EcologicalNetworks`, a Julia package to analyse ecological networks of species interactions. It is built to be extensible, and to facilitate the development of flexible network analysis pipelines. `EcologicalNetworks` has been designed to be robust, easy to write code with, maintainable (so that bugs can be fixed rapidly, and new features as well as performance improvement added easily), 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>.

The development of this package is done openly on GitHub. We are accepting new functions, bug fixes and alternative implementations as pull requests, which will undergo code review. Note that in order to ensure the reliability of the package, we rely on two approaches. First, we strictly control the methods that are implemented: only measures with clear ecological relevance, and no known glaring issues, will be part of the package. Because `EcologicalNetworks.jl` works as a library, it is easy to expand it to write custom methods. Second, the code undergoes continuous integration and is covered by a robust suite of unit tests; in addition, we perform integration testing for all new releases, by running typical network analyses.

To cite EcologicalNetworks or acknowledge its use, cite this software note.

Acknowledgements – We thank Stephen J Beckett, Ignasi Bartomeus, David Garcia-Calleja and Michael Krabbe Borregaard for comments on an earlier version of this manuscript.
Funding – MS is supported by the Research Foundation – Flanders (FWO17/PDO/067).
Author contributions – TP and ZB designed the case studies and wrote the code. PS developed the random network functions. MS and LH developed the information theory and resilience functions. All authors contributed to the manuscript. ZB performed usability test of the package.

References

Baker, N. J. et al. 2014. Species' roles in food webs show fidelity across a highly variable oak forest. – Ecography 38: 130–139.
Barber, M. J. 2007. Modularity and community detection in bipartite networks. – Phys. Rev. E 76: 066102.
Bascompte, J. et al. 2003. The nested assembly of plant–animal mutualistic networks. – Proc. Natl Acad. Sci. USA 100: 9383–9387.
Bastolla, U. et al. 2009. The architecture of mutualistic networks minimizes competition and increases biodiversity. – Nature 458: 1018–1020.
Bezanson, J. et al. 2017. Julia: a fresh approach to numerical computing. – SIAM Rev. 59: 65–98.
Borrett, S. R. and Lau, M. K. 2014. enaR: an R package for ecosystem network analysis. – Methods Ecol. Evol. 5: 1206–1213.
Csardi, G. and Nepusz, T. 2006. The igraph software package for complex network research. – InterJournal Complex Syst. 1695.
Delmas, E. et al. 2018. Analysing ecological networks of species interactions. – Biol. Rev. 112540.
Dormann, C. F. et al. 2008. Introducing the bipartite package: analysing ecological networks. – R News 8: 8–11.
Flores, C. O. et al. 2016. BiMAT: a MATLAB package to facilitate the analysis and visualization of bipartite networks. – Methods Ecol. Evol. 7: 127–132.
Fortuna, M. A. and Bascompte, J. 2006. Habitat loss and the structure of plant-animal mutualistic networks. – Ecol Lett. 9: 281–286.
Fortuna, M. A. et al. 2010. Nestedness versus modularity in ecological networks: two sides of the same coin? – J. Anim. Ecol. 78: 811–817.
Guimerà, R. and Nunes Amaral, L. A. 2005. Functional cartography of complex metabolic networks. – Nature 433: 895–900.
Hadfield, J. D. et al. 2014. A tale of two phylogenies: comparative analyses of ecological interactions. – Am. Nat. 183: 174–187.
Hagberg, A. A. et al. 2008. Exploring network structure, dynamics and function using NetworkX. – In: Varoquaux, G. et al. (eds), Proceedings of the 7th python in science conference. Pasadena, CA USA, pp. 11–15.
Jordano, P. 2016a. Chasing ecological interactions. – PLoS Biol. 14: e1002559.
Jordano, P. 2016b. Sampling networks of ecological interactions. – Funct. Ecol. 30: 1883–1893.
Koleff, P. et al. 2003. Measuring beta diversity for presence–absence data. – J. Anim. Ecol. 72: 367–382.
Liu, X. and Murata, T. 2009. Community detection in large-scale bipartite networks. 2009 IEEE/WIC/ACM International Joint Conference on web intelligence and intelligent agent technology. – Inst. of Electrical and Electronics Engineers (IEEE).
Memmott, J. et al. 2004. Tolerance of pollination networks to species extinctions. – Proc. R. Soc. B 271: 2605–2611.
Mora, B. B. et al. 2018. pymfinder: a tool for the motif analysis of binary and quantitative complex networks. – bioRxiv 364703.
Olesen, J. M. et al. 2007. The modularity of pollination networks. – Proc. Natl Acad. Sci. USA 104: 19891–19896.
Pellissier, L. et al. 2017. Comparing species interaction networks along environmental gradients. – Biol. Rev. 93: 785–800.
Poisot, T. et al. 2012. The dissimilarity of species interaction networks. – Ecol. Lett. 15: 1353–1361.
Poisot, T. et al. 2016a. The structure of probabilistic networks. – Methods Ecol. Evol. 7: 303–312.
Poisot, T. et al. 2016b. Describe, understand and predict: why do we need networks in ecology? – Funct. Ecol. 30: 1878–1882.
Simmons, B. I. et al. 2018. bmotif: a package for counting motifs in bipartite networks.
Stock, M. et al. 2017. Linear filtering reveals false negatives in species interaction data. – Sci. Rep. 7: 45908.
Ulanowicz, R. E. 2001. Information theory in ecology. – Comput. Chem. 25: 393–399.
Weitz, J. S. et al. 2013. Phage–bacteria infection networks. – Trends Microbiol. 21: 82–91.

Supplementary material (available online as Appendix ecog-04310 at <www.ecography.org/appendix/ecog-04310>). Appendix 1.