Intro to BioEnergeticFoodWebs

by Chris Griffiths, Eva Delmas and Andrew Beckerman, Oct. 2021.

This document follows on from 'Getting started', 'Basic Julia commands' and 'Differential Equations in Julia' and assumes that you're still working in your active project.

This document introduces the `BioEnergeticFoodWebs.jl` and `EcologicalNetworks.jl` packages. It demonstrates how to run the BioEnergetic Food Web (BEFW) model, how to vary variables of interest (e.g., productivity) and construct experiments designed to investigate the effect of different variables on population and community dynamics.

For those that are unfamilar with the BEFW and it's application in Julia, we advise checking out the **MEE paper** before we start. Remember, the BEFW model is also based on a system of differential equations and is solved using the same engine as the `DifferentialEquations.jl` package.

Load packages

```
md"## Load packages"
```

You'll need the following packages for this tutorial:

```
md"
You'll need the following packages for this tutorial:
"
```

```
using BioEnergeticFoodWebs   , EcologicalNetworks   , JLD2   , Statistics   , Plots
   , CSV   , DataFrames   , Random
```

The `JLD2.jl` package will be useful later as it allows you to directly export and load a BEFW output object. Let's also set a random seed for reproducibility:

```
MersenneTwister(21)
```

```
Random.seed!(21)
```

Preamble

One of main advantages of running food web models in Julia is that simulations are fast and can be readily stored in your active project. With this in mind, make a new folder in your project called `out_objects` (right click > New Folder). Alternatively, you can create an `out_objects` folder directly using `mkdir()`:

```
# We've already create a folder called out_objects in our project but an example of
mkdir() would be:
mkdir("example_folder")
# if you haven't created an out_objects folder yet, simply replace "example_folder"
with "out_objects".
```

Running the BEFW

There are four major steps when running the BioEnergetic Food Web model in Julia:

1. Generate an initial network
2. Fix parameters
3. Simulate
4. Explore output and plot

Initial network

Before running the BEFW model, we have to construct an initial random network (an adjacency matrix) using **the niche model**. The network is characterised by the number of species in the network and its **connectance** value. Here, we generate a network of 20 species with a connectance value of 0.15:

```
md"
## Running the BEFW

There are four major steps when running the BioEnergetic Food Web model in Julia:
1. Generate an initial network
2. Fix parameters
3. Simulate
4. Explore output and plot

### Initial network
Before running the BEFW model, we have to construct an initial random network (an
adjacency matrix) using [the niche model](https://www.nature.com/articles/35004572?
cacheBust=1510239451067). The network is characterised by the number of species in
the network and its [connectance](https://en.wikipedia.org/wiki/Ecological_network)
value. Here, we generate a network of 20 species with a connectance value of 0.15:
"
```

```
A_bool = 20×20 (String) unipartite ecological network (L: 58 - Bool)
```

```
# generate network
A_bool = EcologicalNetworks.nichemodel(20,0.15)
```

```
Ad = 20×20 Matrix{Bool}:
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 0  0  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 1  1  1  1  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  1  1  1  1  0  0  0  0  0  0  0  0  0  0  0  0
 ⋮                 ⋮                 ⋮                 ⋮
 0  0  0  0  0  0  0  0  0  0  1  1  1  1  1  1  1  0  0
 0  0  0  0  0  0  1  1  1  1  1  1  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  1  1  1  1  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  1  1  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0  1  1  1  1  1  1  1  1
 0  0  1  1  1  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0
```

```
# convert the UnipartiteNetwork object into a matrix of 1s and 0s
Ad = adjacency(A_bool)
```

```
A = 20×20 Matrix{Int64}:
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 0  0  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 1  1  1  1  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  1  1  1  1  0  0  0  0  0  0  0  0  0  0  0  0
 ⋮                 ⋮                 ⋮                 ⋮
 0  0  0  0  0  0  0  0  0  0  1  1  1  1  1  1  1  0  0
 0  0  0  0  0  0  1  1  1  1  1  1  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  1  1  1  1  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  1  1  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0  1  1  1  1  1  1  1  1
 0  0  1  1  1  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0
```

```
A = Int.(Ad)
```

In the above code chunk, we are saving the output from running the `nichemodel` as `A_bool` and then using the `A` part of `A_bool` to construct our initial random network. Within `A`, 1s indicate an interaction among species and 0s no interaction. In the packages used here, the networks are directed from `i` (rows) to `j` (columns), describing the direction of the interaction (i eats j), not of the flow of biomass.

You can check the connectance of A using:

```
co = 0.145
```

```
# calculate connectance
co = sum(A)/(size(A,1)^2)
```

Here, connectance is calculated as the number of realised links (sum of 1s in `A`) divided by the number of species in `A` squared. This end part (species^2) describes the maximum number of possible links in the network `A`.

Parameters

Prior to running the BEFW model, you have to create a vector of model parameters using the `model_parameters` function. Numerous parameter values can be specified within the `model_parameters` function, however, most of them have default values that are built into the `BioEnergeticFoodWebs.jl` package. For simplicity, we use the default values here:

```
p =
  Dict{Symbol, Any}(
     :α ⇒ 1.0
     :e_carnivore ⇒ 0.85
     :Γh ⇒  [0.0, 0.0, 0.0, 0.5, 0.0, 0.5, 0.5, 0.5, 0.5,  more ,0.5]
     :m_producer ⇒ 1.0
     :h ⇒ 1.0
     :c ⇒  [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,  more ,0.0]
     :vertebrates ⇒  BitVector: [false, false, false, false, false, false, false, false,
     :tmpA ⇒  []
     :rewire_method ⇒ :none
     :trophic_rank ⇒  [1.0, 1.0, 1.0, 2.0, 1.0, 2.2, 2.75, 2.5, 2.5,  more ,2.5]
     :w ⇒ 20×20 Matrix{Float64}:
            0.0  0.0  0.0   0.0   0.0   0.0   0.0       …  0.0    0.0    0.0    0.0    0.
            0.0  0.0  0.0   0.0   0.0   0.0   0.0          0.0    0.0    0.0    0.0    0.
            0.0  0.0  0.0   0.0   0.0   0.0   0.0          0.0    0.0    0.0    0.0    0.
            0.0  0.0  1.0   0.0   0.0   0.0   0.0          0.0    0.0    0.0    0.0    0.
            0.0  0.0  0.0   0.0   0.0   0.0   0.0          0.0    0.0    0.0    0.0    0.
            0.2  0.2  0.2   0.2   0.2   0.0   0.0       …  0.0    0.0    0.0    0.0    0.
            0.0  0.0  0.0   0.0   0.25  0.25  0.25         0.0    0.0    0.0    0.0    0.
            ⋮                        ⋮                ⋱  ⋮
            0.0  0.0  0.0   0.0   0.0   0.0   0.0          0.125  0.125  0.125  0.0    0.
            0.0  0.0  0.0   0.0   0.0   0.0   0.166667  …  0.0    0.0    0.0    0.0    0.
            0.0  0.0  0.0   0.0   0.0   0.0   0.25         0.0    0.0    0.0    0.0    0.
            0.0  0.0  0.0   0.0   0.0   0.0   0.0          0.0    0.0    0.0    0.0    0.
            0.0  0.0  0.0   0.0   0.0   0.0   0.0          0.125  0.125  0.125  0.125  0.
            0.0  0.0  0.25  0.25  0.25  0.25  0.0          0.0    0.0    0.0    0.0    0.
     :TSR_type ⇒ :no_response
     :e_herbivore ⇒ 0.45
     :productivity ⇒ :species
     :efficiency ⇒ 20×20 Matrix{Float64}:
                    0.0   0.0   0.0   0.0   0.0   0.0   0.0   …  0.0   0.0   0.0   0.0
                    0.0   0.0   0.0   0.0   0.0   0.0   0.0      0.0   0.0   0.0   0.0
                    0.0   0.0   0.0   0.0   0.0   0.0   0.0      0.0   0.0   0.0   0.0
                    0.0   0.0   0.45  0.0   0.0   0.0   0.0      0.0   0.0   0.0   0.0
                    0.0   0.0   0.0   0.0   0.0   0.0   0.0      0.0   0.0   0.0   0.0
                    0.45  0.45  0.45  0.85  0.45  0.0   0.0   …  0.0   0.0   0.0   0.0
                    0.0   0.0   0.0   0.0   0.45  0.85  0.85     0.0   0.0   0.0   0.0
                    ⋮                             ⋮           ⋱     ⋮
                    0.0   0.0   0.0   0.0   0.0   0.0   0.0      0.85  0.85  0.85  0.85
                    0.0   0.0   0.0   0.0   0.0   0.0   0.85  …  0.0   0.0   0.0   0.0
                    0.0   0.0   0.0   0.0   0.0   0.0   0.85     0.0   0.0   0.0   0.0
                    0.0   0.0   0.0   0.0   0.0   0.0   0.0      0.0   0.0   0.0   0.0
                    0.0   0.0   0.0   0.0   0.0   0.0   0.0      0.85  0.85  0.85  0.85
                    0.0   0.0   0.45  0.85  0.45  0.85  0.0      0.0   0.0   0.0   0.0
     :K ⇒  [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,  more ,1.0]
     :S ⇒ 20
     :extinctionstime ⇒  []
     :is_producer ⇒  BitVector: [true, true, true, false, true, false, false, false, fals
     :dp ⇒ #11
     :x ⇒  [0.138, 0.138, 0.138, 0.3141, 0.138, 0.3141, 0.3141, 0.3141, 0.3141,  more ,
     :Z ⇒ 1.0
     :extinctions ⇒  []
     :dry_mass_293 ⇒  [0.153846, 0.153846, 0.153846, 0.153846, 0.153846, 0.153846, 0.153
     :np ⇒ 5
     :A ⇒ 20×20 Matrix{Int64}:
            0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
            0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
            0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
            0  0  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
            0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
```

```
 1  1  1  1  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  1  1  1  1  0  0  0  0  0  0  0  0  0  0  0
 ⋮                 ⋮                 ⋮                 ⋮
 0  0  0  0  0  0  0  0  0  0  1  1  1  1  1  1  1  0  0
 0  0  0  0  0  0  1  1  1  1  1  1  0  0  0  0  0  0  0
 0  0  0  0  0  0  1  1  1  1  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  1  1  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0  1  1  1  1  1  1  1
 0  0  1  1  1  1  0  0  0  0  0  0  0  0  0  0  0  0  0

:Γ ⇒  [0.0, 0.0, 0.0, 0.5, 0.0, 0.5, 0.5, 0.5, 0.5,    more ,0.5]

:ar ⇒  [0.0, 0.0, 0.0, 16.0, 0.0, 16.0, 16.0, 16.0, 16.0,    more ,16.0]

:bodymass ⇒  [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,    more ,1.0]

:is_herbivore ⇒  BitVector: [false, false, false, true, false, true, true, true, tru

   more
)
```

```
# create model parameters
p = model_parameters(A)
# in the most simple case, the model_parameters function simply requires A
```

For more information and a full list of the parameters and their defaults values type `?model_parameters` in the REPL.

If you want to view, check or extract any of the parameter values in `p` use the `p[:name]` notation. For example, you can view a vector of each species' trophic rank using:

```
[1.0, 1.0, 1.0, 2.0, 1.0, 2.2, 2.75, 2.5, 2.5, 1.0, 3.0, 3.0, 4.0, 2.5, 3.5, 3.0, 2.75, 2.
```

```
# view trophic ranks:
p[:trophic_rank]
```

Simulate

To run the BEFW model, we first assign biomasses at random to each species and then simulate the biomass dynamics forward using the `simulate` function:

```
bm =
 [0.331066, 0.300448, 0.0291859, 0.295575, 0.0361821, 0.963121, 0.150602, 0.344048, 0.562
```

```
# assign biomasses
bm = rand(size(A,1))
```

```
out =
  Dict{Symbol, Any}(
      :p ⇒   Dict(:α ⇒ 1.0, :e_carnivore ⇒ 0.85, :Γh ⇒ [0.0, 0.0, 0.0, 0.5, 0.0,    more
```

```
      :B ⇒ 8023×20 adjoint(::Matrix{Float64}) with eltype Float64:
             0.331066  0.300448  0.0291859    0.295575   …  0.0369967   0.0530368  0.82237
             0.275087  0.251328  0.00817244   0.173187      0.0408813   0.0646593  0.94844
             0.247361  0.227229  0.0029042    0.0991832     0.0386371   0.0798476  1.0428
             0.240935  0.22239   0.00121857   0.056877      0.033843    0.098953   1.09539
             0.250236  0.232034  0.000573703  0.0330167     0.0287532   0.122374   1.1061
             0.271583  0.252987  0.000294967  0.0195387  …  0.0239862   0.150397   1.08238
             0.302566  0.283188  0.000163569  0.0118501     0.0197161   0.183055   1.03421
             ⋮                                          ⋱
             0.559035  0.559035  0.0          0.0           0.00834797  0.0        0.35599
             0.559035  0.559035  0.0          0.0           0.00835872  0.0        0.35599
             0.559035  0.559035  0.0          0.0           0.00836805  0.0        0.35599
             0.559035  0.559035  0.0          0.0        …  0.00837562  0.0        0.35599
             0.559035  0.559035  0.0          0.0           0.00838156  0.0        0.356
             0.559035  0.559035  0.0          0.0           0.00838598  0.0        0.356
      :t ⇒   [0.0, 0.25, 0.5, 0.75, 1.0, 1.25, 1.5, 1.75, 2.0,    more ,2000.0]
)
```

```
# simulate
out = simulate(p, bm, start=0, stop=2000)
# this might take a few seconds
```

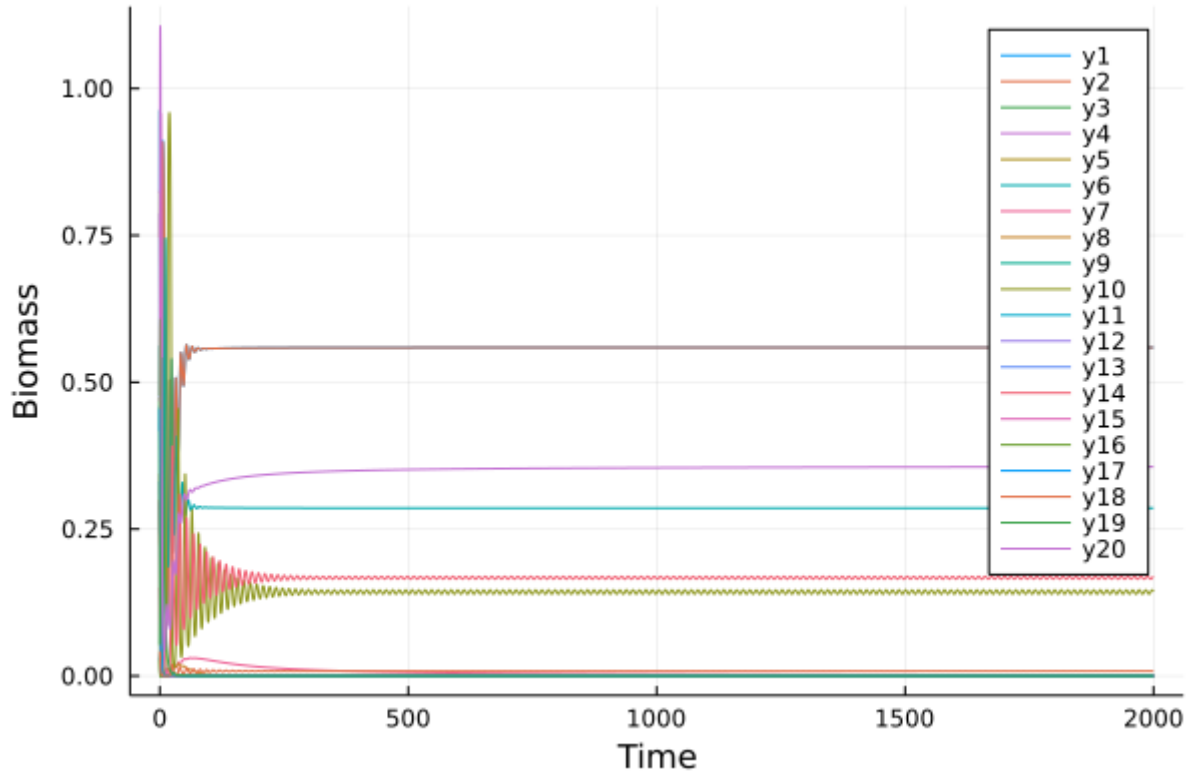
The `simulate` function requires the model parameters `p` and the species biomasses `bm`. In addition, you can specify the timespan of the simulation (using the `start` and `stop` arguments), fix a species extinction threshold (using `extinction_threshold`) and select a solver (using `use`). For more information type `?simulate` in the REPL.

Output and plot

Once the simulation finishes, the output is stored as a dictionary called `out`. Within `out` there are three entries:

1. `out[:p]` - lists the parameters
2. `out[:B]` - biomass of each species through time
3. `out[:t]` - timesteps (these typically increase in 0.25 intervals)

The biomass dynamics of each species can then be plotted. Similar to the `DifferentialEquations.jl` package, the `BioEnergeticFoodWebs.jl` package also has it's own built in plotting recipe:




```
plot(out[:t],out[:B], legend = true, ylabel = "Biomass", xlabel = "Time")
```

You'll notice that the biomass dynamics are noisey during the first few hundred time steps, these are the system's transient dynamics. The dynamics then settle into a steady state where the system can be assumed to be at equilbirum. You'll also notice that some species go extinct and some persist, the initial number of species in the food web (20 in this case) can found using `out[:p][:S]` and the identity of those that went extinct using `out[:p][:extinctions]`.

The `BioEnergeticFoodWebs.jl` package also has a range of built in functions that conveniently calculate some of the key metrics of the food web, these include the total biomass, the diversity, the species persistence and the temporal stability:

```
biomass = 2.079634374182744
```

```
# total biomass
biomass = total_biomass(out, last=1000)
```

```
diversity = 0.8152947375528959
```

```
# diversity
diversity = foodweb_evenness(out, last=1000)
```

```
persistence = 0.4
```

```
# persistence
persistence = species_persistence(out, last=1000)
```

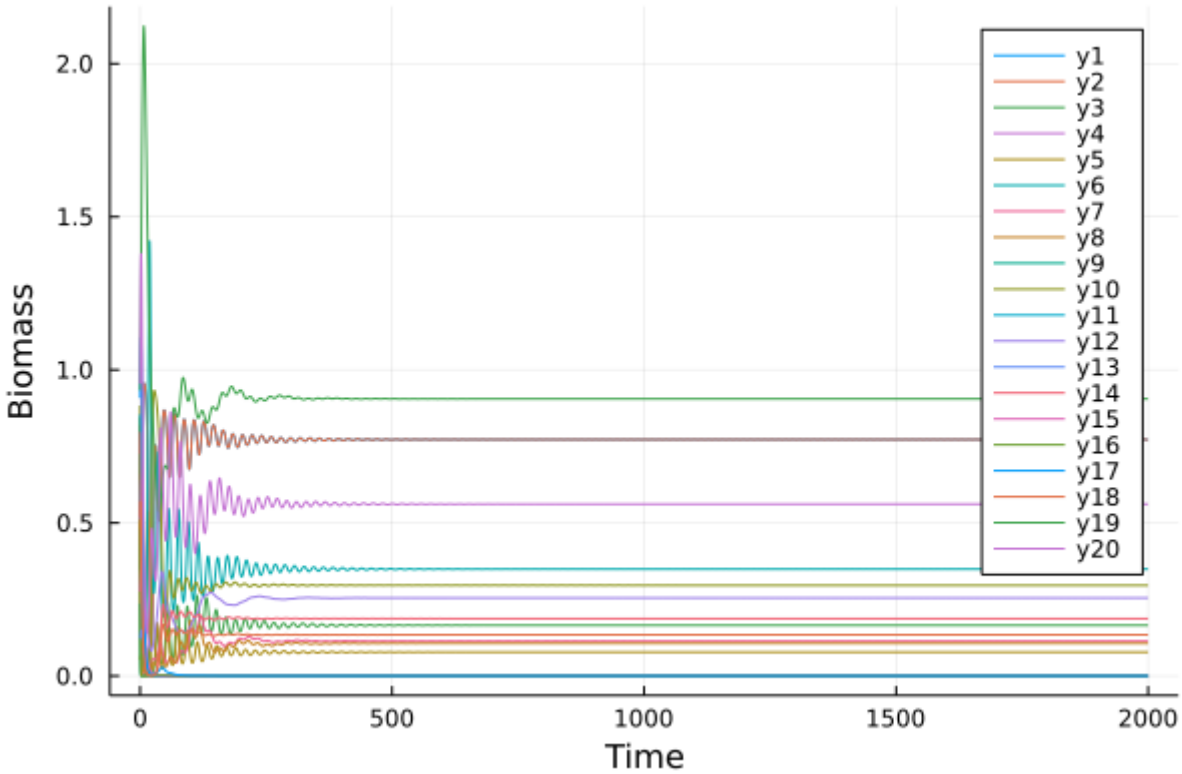
```
stability = -0.009341145049945184
```

```
# stability
stability = population_stability(out, last=1000)
```

Each of these functions will output a single value. This value is the average over the `last 1000` time steps. For more information, use `?` to access the help files on each function in the REPL (e.g., `?species_persistence`).

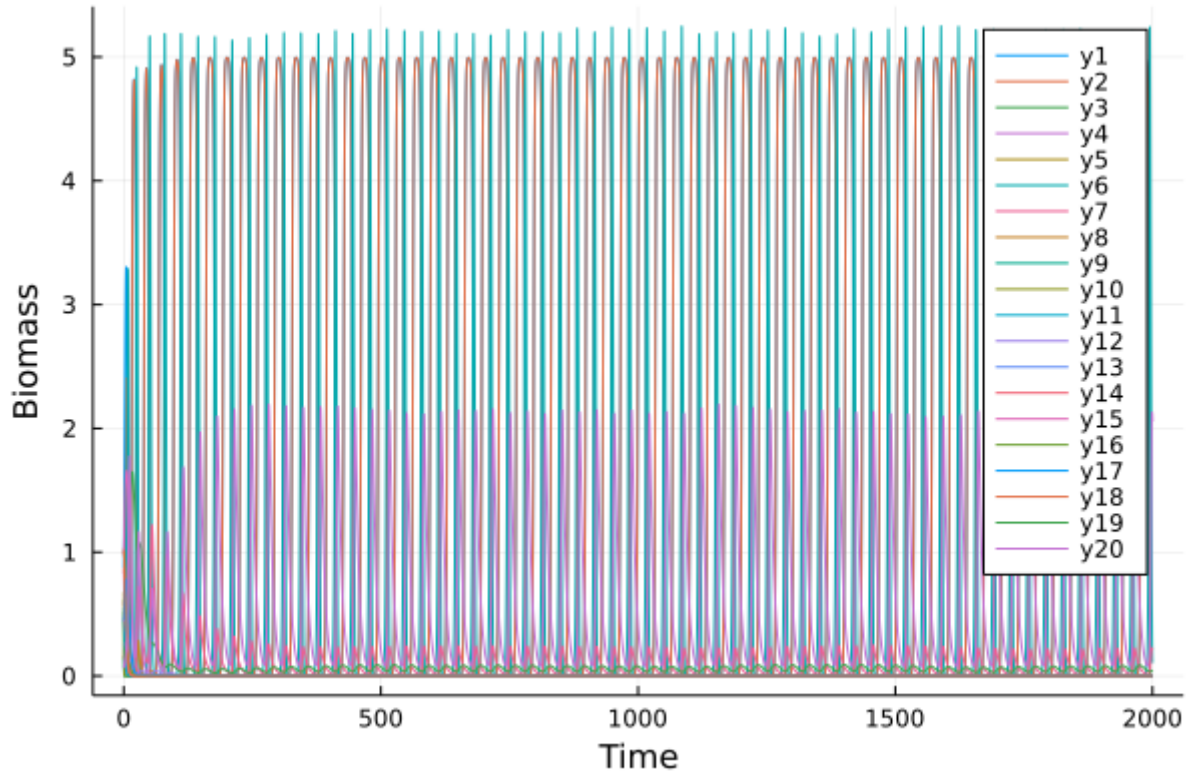
Variables

Once you've got the BEFW model running, the next step is to vary a variable of interest and rerun. For example, we might be interested in what affect a small change in Z (consumer-resource body mass ratio) has on the estimated food web and its biomass dynamics. The default value for Z is 1.0, but what happens if we increase it to 10.0:




```
begin
    # set Z (has to be a floating number not an integer)
    Z = 10.0
    # create model parameters
    p_z = model_parameters(A, Z = Z)
    # assign biomasses
    bm_z = rand(size(A,1))
    # simulate
    out_z = simulate(p_z, bm_z, start=0, stop=2000)
    # plot
    plot(out_z[:t], out_z[:B], legend = true, ylabel = "Biomass", xlabel = "Time")
end
```

Similarly, what happens if we also increase the carrying capacity (K) of the resource from 1.0 (default) to 5.0:




```
begin
    # set K (has to be a floating number not an integer)
    K = 5.0
    # create model parameters
    p_K = model_parameters(A, Z = Z, K = [K]) # K needs to be provided in []'s
    # assign biomasses
    bm_K = rand(size(A,1))
    # simulate
    out_K = simulate(p_K, bm_K, start=0, stop=2000)
    # plot
    Plots.plot(out_K[:t], out_K[:B], legend = true, ylabel = "Biomass", xlabel =
"Time")
end
```

As you've probably guessed, the main message here is that many variables can be changed in the BEFW model and it's super easy to do so. Some changes will have large effects and some not so much. In the next step, we take this one step further.

Quick note - the `begin` and `end` used in the above code chunks and those that follow aren't needed when running Julia in VS Code. They are simply an artifact of the markdown software we have used and can be ignored. They just provide a way of running chunks of code without seperating them onto seperate lines.

Experiments

The next step is to construct a computional experiment designed to investigate the effect of different variables on population and community dynamics. To do this we construct a gradient of variables as vectors and then simulate the BEFW model multiple times using a loop. To illustrate this, we're going to reproduce example 1 from **Delmas et al. 2016**. The aim of this example is to investigate the effect of increasing K on food web diversity. In addition, we're also going to allow α (interspecific competition relative to intraspecific competition) to vary and repeat the experiment 5 times with 5 different initial networks.

First, we define the experiment by creating vectors of our variables and fixing the number of repetitions:

```
α =   [0.92, 1.0, 1.08]
```

```
α = [0.92, 1.0, 1.08]
# 0.92 - the interspecific competition is smaller than the intraspecific competition promoting coexistence
# 1.0 - neutrally stable
# 1.08 - the intraspecific competition is smaller the interspecific competition favouring competitive exclusion
```

```
k =
  [0.1, 0.16681, 0.278256, 0.464159, 0.774264, 1.29155, 2.15443, 3.59381, 5.99484, 10.0]
```

```
# vector of K
k = exp10.(range(-1,1,length=10))
# log scale from 0.1 to 10
```

```
reps = 5
```

```
# number of reps
reps = 5
```

We then create a dataframe to store the outputs:

```
df =
```

α	K	network	diversity	stability	biomass

```
# dataframe
df = DataFrame(α = [], K = [], network = [], diversity = [], stability = [], biomass = [])
```

and construct a `while` loop to generate the 5 unique initial networks, each of which contains 20 species with a connectance value of 0.15:

```
begin
    # list to store networks
    global networks = []
    # monitoring variable
    global l = length(networks)
    # while loop
    while l < reps
        # generate network
        A_bool = EcologicalNetworks.nichemodel(20,0.15)
        # convert the UnipartiteNetwork object into a matrix of 1s and 0s
        Ad = adjacency(A_bool)
        A = Int.(Ad)
        # calculate connectance
        co = sum(A)/(size(A,1)^2)
        # ensure that connectance = 0.15
        if co == 0.15
            push!(networks, A)
            # save network is co = 0.15
        end
        global l = length(networks)
    end
end
```

We can then run the simulations by looping, using nested `for` loops, over the unique values of α and K, as well as the 5 unique initial networks. After each simulation we will save each output object to our active project as a `JLD2` file and store any output metrics of interest in our dataframe:

```
# loop over networks
for h in 1:reps
    A = networks[h]
    # here, you might want to save a copy of the initial network using writedlm(A)

    # loop over α
    for i in 1:length(α)
        # loop over K
        for j in 1:length(k)

        # create model parameters

        p = model_parameters(A, α = α[i], K = [k[j]], productivity=:competitive) # again, k values need []'s
        # assign biomasses
        bm = rand(size(A,1))
        # simulate
        out = simulate(p, bm, start=0, stop=2000)

        # dummy naming variables
        α_num = α[i]
        K_num = k[j]
        # save `out` as a JLD2 object using the @save macro:
        @save "out_objects/model_output, network = $h, alpha = $α_num, K = $K_num.jld2" out

        # calculate output metrics
        diversity = foodweb_evenness(out, last = 1000)
        stability = population_stability(out, last = 1000)
        biomass = total_biomass(out, last = 1000)

        # push to df
        push!(df, [α[i], k[j], h, diversity, stability, biomass])

        # print some stuff - see how the simulation is progressing
        println(("α = $α_num", "K = $K_num", "network = $h"))
        end
    end
end
# the code will be much faster if you remove the @save command
```

We can then explore the outputs and plot our results. Here, instead of using the built in plotting recipe, we will construct a plot that matches figure 1 in **Delmas et al. 2016**. Specifically, we will plot food web diversity (y-axis) as a function of K (x-axis) and α (colour):

	α	K	network	diversity	stability	biomass
1	0.92	0.1	1.0	1.0	-0.00049472	0.105633
2	0.92	0.16681	1.0	1.0	-0.000473063	0.176206
3	0.92	0.278256	1.0	0.794265	-0.000383165	0.293979
4	0.92	0.464159	1.0	0.746302	-0.00039277	0.488871
5	0.92	0.774264	1.0	0.686863	-0.0015427	0.812639
6	0.92	1.29155	1.0	0.964897	-0.000777836	1.0517
7	0.92	2.15443	1.0	0.993487	-0.00174054	1.27699
8	0.92	3.59381	1.0	0.99946	-0.0031786	1.44472
9	0.92	5.99484	1.0	0.999898	-0.00533331	1.56164
10	0.92	10.0	1.0	0.999141	-0.00935681	1.63925
	more					
150	1.08	10.0	5.0	0.594069	-0.827963	4.14109

```
# explore output
df
```

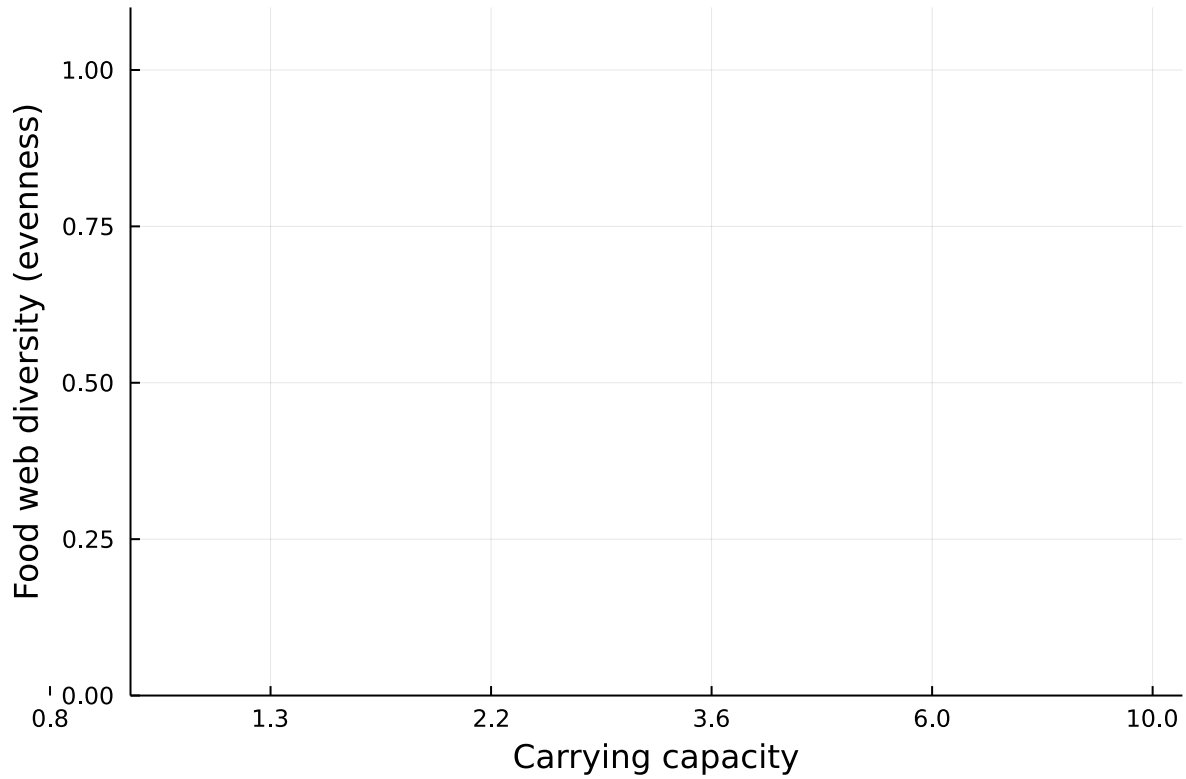
	α	K	network	diversity	stability	biomass
1	0.92	0.1	1.0	1.0	-0.00049472	0.105633
2	0.92	0.16681	1.0	1.0	-0.000473063	0.176206
3	0.92	0.278256	1.0	0.794265	-0.000383165	0.293979
4	0.92	0.464159	1.0	0.746302	-0.00039277	0.488871
5	0.92	0.774264	1.0	0.686863	-0.0015427	0.812639
6	0.92	1.29155	1.0	0.964897	-0.000777836	1.0517

```
first(df,6)
```

	α	K	network	diversity	stability	biomass
1	1.08	0.774264	5.0	NaN	-0.000790586	0.774243
2	1.08	1.29155	5.0	NaN	-0.000839386	1.29152
3	1.08	2.15443	5.0	0.509891	-1.03284	2.11173
4	1.08	3.59381	5.0	0.834122	-0.00180268	3.01781
5	1.08	5.99484	5.0	0.845222	-0.00350984	4.62282
6	1.08	10.0	5.0	0.594069	-0.827963	4.14109

```
last(df,6)
```

pl =




```
# plot
# initialise an empty plot
pl = Plots.plot([NaN], [NaN],
                label = "",
                ylims = (0,1.1),
                leg = :bottomright,
                foreground_colour_legend = nothing,
                xticks = (log10.(k), string.(round.(k, digits = 1))),
                xlabel = "Carrying capacity",
                ylabel = "Food web diversity (evenness)")
```

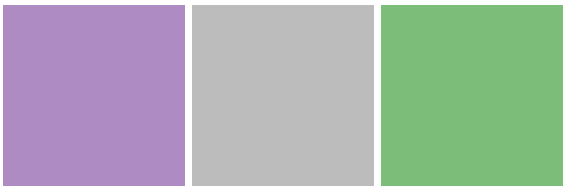
shp = [:square, :diamond, :utriangle]

```
# set marker shapes
shp = [:square, :diamond, :utriangle]
```

```
ls =  [:solid, :dash, :dot]
```

```
# set line types
ls = [:solid, :dash, :dot]
```

```
clr =
```

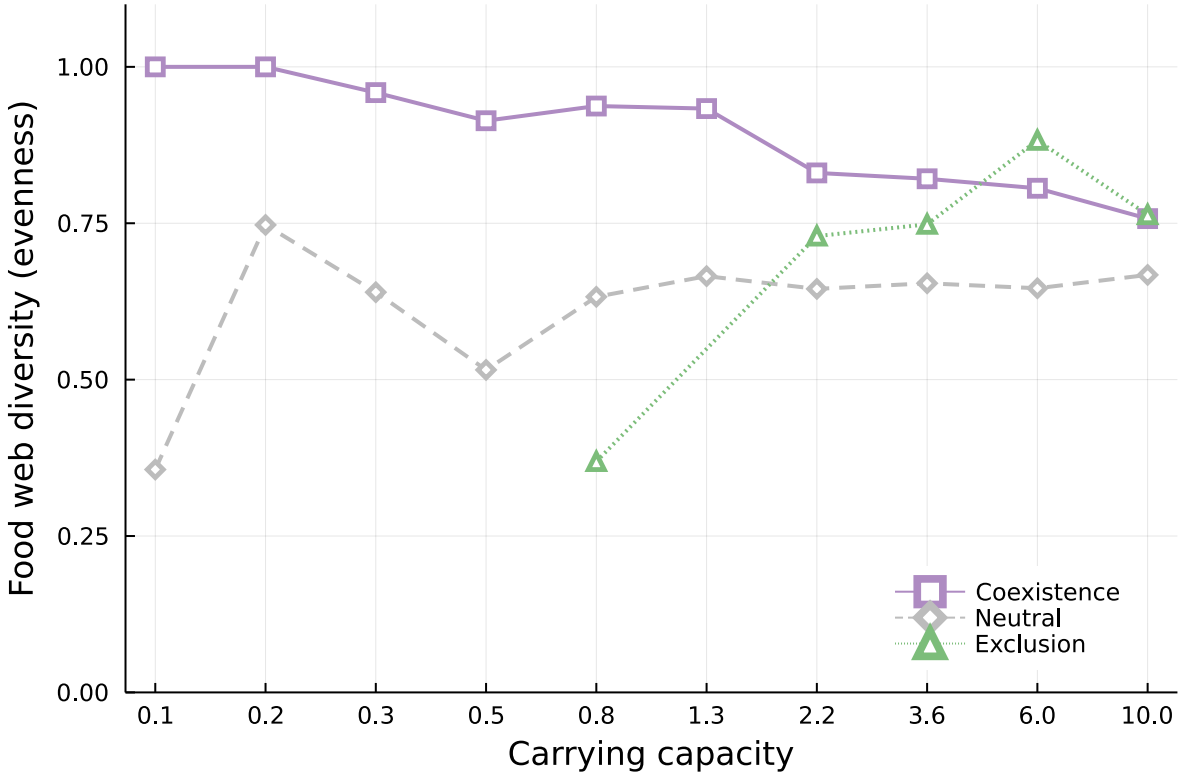


```
# set colours
clr = [RGB(174/255, 139/255, 194/255), RGB(188/255, 188/255, 188/255), RGB(124/255,
189/255, 122/255)]
# when we define colours in Julia they are printed
```

```
lbl =  ["Coexistence", "Neutral", "Exclusion"]
```

```
# set legend labels
lbl = ["Coexistence", "Neutral", "Exclusion"]
```

```
# make the plot
for (i, α) in enumerate(α)
    # subset
    tmp = df[df.α .== α, :]
    # remove NaN values
    tmp = tmp[.!(isnan.(tmp.diversity)), :]
    # calculate mean across reps
    gdf = groupby(tmp, :K)
    meandf = combine(gdf, :diversity => mean)
    # command to avoid printing legends multiple times
    l = i == 1 ? lbl[i] : ""
    # add to pl
    plot!(pl, log10.(meandf.K), meandf.diversity_mean,
              msc = clr[i],
              mc = :white,
              msw = 3,
              markershape = shp[i],
              linestyle = ls[i],
              lc = clr[i],
              lw = 2,
              label = lbl[i],
              seriestype = [:line :scatter])
end
```




```
# display plot
plot(pl)
```

Finally, we can save our dataframe as a .csv file:

```
"My_data.csv"
```

```
# save
CSV.write("My_data.csv", df)
```