Julia in VS Code #4
Introduction to BioEnergeticFoodWebs

Chris Griffiths & Eva Delmas

January 7, 2021

This document follows on from "Julia in VS Code #1, #2 and #3" 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.

1 Packages

First, import your package manager, activate and instantiate (only neccessary if you've closed VS Code between tutorials):

```
import Pkg
Pkg.activate(".")
Pkg.instantiate()
```

Then install the `BioEnergeticFoodWebs.jl`, `EcologicalNetworks.jl` and `JLD2.jl` packages and let Julia know you want to use them:

```
# install
Pkg.add("BioEnergeticFoodWebs")
Pkg.add("EcologicalNetworks")
Pkg.add("JLD2")
Pkg.add("Statistics")
# use
using BioEnergeticFoodWebs
using EcologicalNetworks
using JLD2
using Statistics
```

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:

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

2 Preamble

One of main advantegeous of running food web models in Julia is that simulations are fast and can be readily stored in your active project. With this in might, 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("out_objects/")`.

3 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

3.1 Initial network

Before running the BEFW model, we have to construct an initial random network 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:

```
# generate network
A_bool = EcologicalNetworks.nichemodel(20,0.15)
# convert the UnipartiteNetwork object into a matrix of 1s and 0s
A = Int.(A_bool.A)
# 1s indicate an interaction among species and 0s no interaction. In the packages used
here, the networks are directed from i to j (i eats j), describing the direction of the
interaction, not of the flow of biomass.
```

You can check the connectance of A using:

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

3.2 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:

```
# 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.

3.3 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:

```
# assign biomasses
bm = rand(size(A,1))
# select biomasses at random between ]0:1[

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

The `simulate` 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.

3.4 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
Plots.plot(out[:t], out[:B], legend = true, ylabel = "Biomass", xlabel = "Time")
# this may take a minute to render
```

You'll notice that the biomass dynamics are noisey during the first few hundred time steps, these are 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 number of species in the food web 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 convientially calculate some the key metrics of the food web, these include the total biomass, the diversity, the species persistence and the temporal stability:

```
# total biomass
biomass = total_biomass(out, last=1000)
# diversity
diversity = foodweb_evenness(out, last=1000)
# persistence
persistence = species_persistence(out, last=1000)
# 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.

4 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 reduce it to 10.0:

```
# set Z
Z = 10.0
# create model parameters
p = model_parameters(A, Z = Z)
# assign biomasses
bm = rand(size(A,1))
# simulate
out = simulate(p, bm, start=0, stop=2000)
# plot
Plots.plot(out[:t], out[:B], legend = true, ylabel = "Biomass", xlabel = "Time")
```

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

```
# set K
K = 5.0
# create model parameters
p = model_parameters(A, Z = Z, K = K)
# assign biomasses
bm = rand(size(A,1))
# simulate
out = simulate(p, bm, start=0, stop=2000)
# plot
Plots.plot(out[:t], out[:B], legend = true, ylabel = "Biomass", xlabel = "Time")
```

As you've probably guessed, the main message here is many variables can be changed in the BEFW model and it's super easy to do so. In the next step, we take this one step further.

5 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 illistrate 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 relatively 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:

```
# vector of α
```

```
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

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

# number of reps
reps = 5
```

We then create a dataframe to store the outputs:

```
# 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:

```
# 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
    A = Int.(A_bool.A)
    # 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
```

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])
        # 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 (line colour):

```
# explore output
describe(df)
first(df,6)
last(df,6)

# 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)")

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

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

# 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
```

```
# 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
    meandf = by(tmp, :K, :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:

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