Functional Response

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

This document builds on the previous tutorials, in particular, 'Intro to BioEnergeticFoodWebs' and introduces the functional response, the theory (and maths) behind it, and how it is used in the BEFW. The aim of this tutorial, and those that follow, is to delve deeper into the inner workings of the BEFW model and demonstrate its utility. Specifically, they will highlight how certain processes can be manipulated and adapted to investigate biological questions of interest. Moreover, via coded examples, the tutorials will show how the BEFW model can be used to test the response of ecological systems to one, or many, anthropogenic stressors.

We again recommend reading the **MEE paper** before starting this tutorial and assume that you are still working in your active project.

Load packages

You'll need the following packages for this tutorial:

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

```
MersenneTwister(37)
```

```
Random.seed!(37)
```

The theory

The functional response is a classic concept in food web ecology and considers the relationship between predation rate and prey density (**Real 1977**). Specifically, it describes the number of prey that a predator consumes per unit of time as a function of the abundance or density of its prey (**Solomon 1949**; **Holling 1959**, **1965**, **1966**). The functional response epitomises the density-dependent nature of trophic interactions and therefore acts as a key building block to several food web models including the BEFW.

A predator's functional response can typically take one of three types. In all three, consumption rate typically increases with prey density, albeit the exact shape of this increase and at what point a maximum consumption rate is achieved will vary. Functional response types:

A. Type I: Consumption rate increases linearly until a threshold prey density is reached, beyond which consumption rate remains constant. In type I functional responses, it is assumed that the time needed to process prey is negligible, or that consumption doesn't interfere with prey searching.

B. Type II: Consumption rate shows a curvilinear increase, such that the proportion of prey consumed declines as prey density increases. In type II functional responses, it is assumed that the consumer is limited by its capacity to process prey.

C. Type III: Consumption rate exhibits a sigmoidal relationship with prey density, whereby the proportion of prey consumed peaks at intermediate prey densities and saturates at high prey densities. The superlinear part of a type III functional response is often justified by learning time, prey switching, or a combination, during which the efficiency of the predator increases.

Type II functional responses are most frequently observed in the wild. For example, a review of individual functional responses by **Jeschke et al. (2004)** found that the frequency of type II functional responses (77%) far exceeded that of type III (13%) and type I (3%). Type II functional responses are also commonly used in food web models, however, type IIIs are also employed as they are known to boost species persistence by increasing ecological stability (because predation is low when prey densities are also low; **Williams & Martinez 2004**).

FR in the BEFW (and the math)

The BEFW model describes the flow of biomass across trophic levels, whereby species (characterised by a given body mass and metabolic rate) gain and lose biomass based on consumer-resource interactions (i.e. consumption). Gains and losses depend on a species' current biomass, a functional response and an interaction-specific assimilation efficiency (see below). In the orginial bio-energetic model of **Yodzis and Innes (1992)**, the functional response is modelled as a function of **consumer-specific maximum consumption rates and half-saturation densities**. However, this isn't the only approach, and instead it is often convenient to use a classical approach which uses **interaction-specific attack rates and handling times**. Both of these approaches are available in the BEFW model, however, they do require slightly different arguments within the `model_parameters` function:

1. Bio-energetic: `functional_response = :bioenergetic`
2. Classical: `functional_response = :classical`

The bio-energetic approach is the **default**, however, as you'll see in the next few tutorials, we have shifted to using the classical approach, especially when incorporating the effects of temperature.

The `model_parameters` function will take care of the rest, namely calculating various rates based on allometric scaling. However, you are able to manually change these rates using the `p[:rate] = new_value` notation.

The two approaches share multiple parameters:

- `e_carnivore` is a carnivore's assimilation efficiency (default value = `0.85`).
- `e_herbivore` is a herbivore's assimilation efficiency (default value = `0.45`).
- `c` quantifies the strength of intraspecific predator interference (**Beddington 1975** and **DeAngelis et al. 1975**). Predator interference is the degree to which increases in a predator's biomass negatively affect its feeding rates. Changing `c` requires the specification of either one value that is common to all consumers or a vector of consumer-specific values (default value = `0.0` i.e. no predator interference).
- `h` is the Hill exponent. It controls the shape of the functional response and allows you to shift between the three types (default value = `1` i.e. type II).
- `y_invertebrate` and `y_vertebrate` are the maximum consumption rates for the invertebrates and ectotherm vertebrates, respectively.
- `Γ` is the half saturation density, also referred to as B_0 (defaut value = `0.5`)

When using the `:bioenergetic` functional response (FR), the following equations are used:

$$gains_i = \sum_{j \in resources} B_i x_i y_i FR_{ij}$$

$$losses_i = \sum_{j \in consumers} \frac{B_j x_j y_j FR_{ji}}{e_{ji}}$$

where B is current biomass, x and y are metabolic rate and maximum consumption rate, respectively, e is assimilation efficency of consumer i when consuming resource j and FR is

$$FR_{ij} = \frac{\omega_{ij} B_j^h}{B_0^h + c_i B_i B_0^h + \sum_{k=resources} \omega_{ik} B_k^h}$$

where ω_{ij} describes a consumer i's preference for resource j. By default ω_{ij} is calculated as $1/n$ where n is the number of resources that are available to consumer i (i.e. we assume homogenous consumption effort across all possible resources).

In comparison, when the `:classical` functional response is used the consumer-specific maximum consumption rate y_i and half saturation density B_0 are transformed into interaction-specific attack rates ar_{ij} and handling times ht_{ij} using the following substitutions:

$$ht_{ij} = 1/y_i$$

$$ar_{ij} = 1/(B_0 ht_{ij})$$

and the following equations are used:

$$gains_i = \sum_{j \in resources} e_{ij} B_i FR_{ij}$$

$$losses_i = \sum_{j \in consumers} B_j FR_{ji}$$

where FR is

$$FR_{ij} = \frac{ar_{ij} B_j^h}{1 + c_i B_i + \sum_{k=resources} ht_{ik} ar_{ik} B_k^h}$$

Quick version check

Before progressing, it is worth checking that you're using the most up-to-date version of the `BioEnergeticFoodWebs` package. Quickly run the following code in your REPL:

```julia
import Pkg
```

```julia
Pkg.status()
```

You should see the following:

```
[9b49b652] BioEnergeticFoodWebs v1.2.0
https://github.com/PoisotLab/BioEnergeticFoodWebs.jl.git#dev-2.0.0
```

which details the current developmental branch of the BEFW model. This version contains all the additional code and functionality needed to use the `:classical` functional response and incorporate the effects of temperature and enrichment. If you don't see the above, quickly remove the older version (probably v.1.2.0, i.e. without the #dev-2.0.0) using `Pkg.rm('BioEnergeticFoodWebs')` and reinstall using `Pkg.add('BioEnergeticFoodWebs#dev-2.0.0')`. Remember, you can also enter the package manager directly using `]` key.

Using `:bioenergetic`

Define a simple network:

```
md"
Define a simple network:
"
```

```
A = 4×4 Matrix{Int64}:
     0  1  0  0
     0  0  1  1
     0  0  0  0
     0  0  0  0
```

```
A = [0 1 0 0 ; 0 0 1 1 ; 0 0 0 0 ; 0 0 0 0]
```

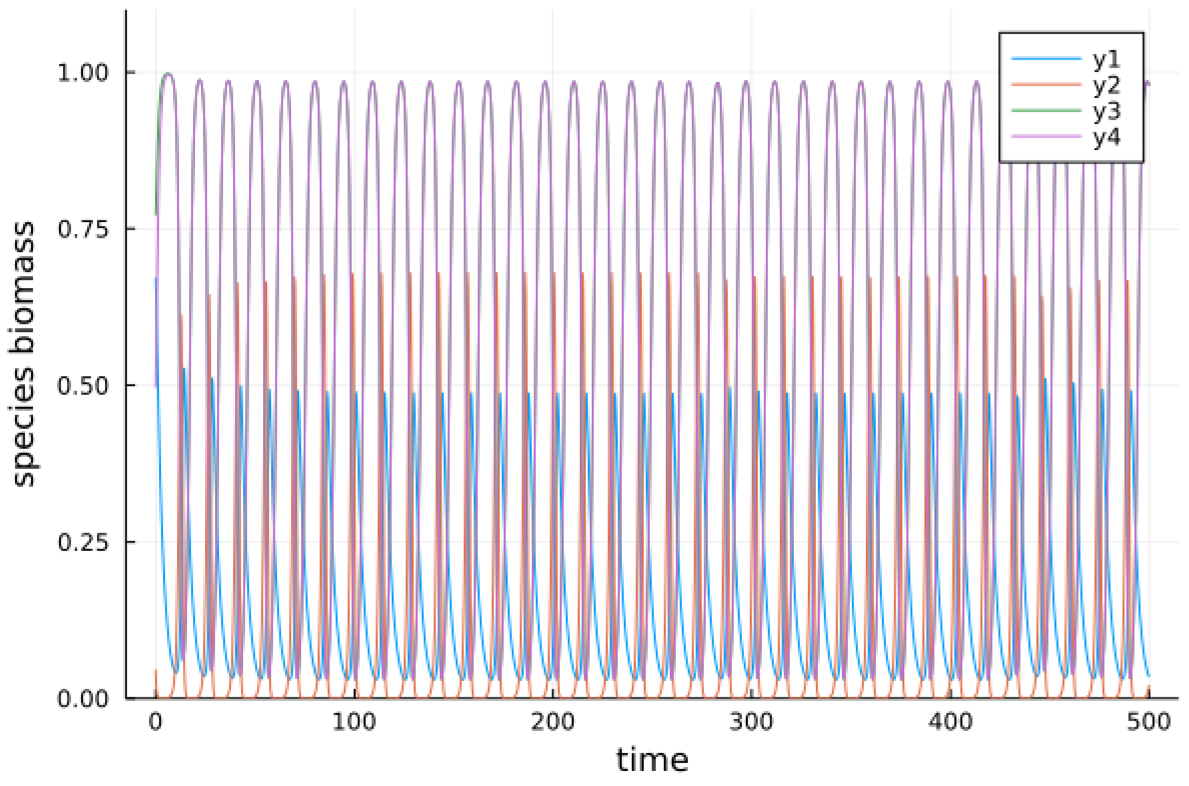
fix parameters using `model_parameters`:

```
p_bio =
  Dict(:α ⇒ 1.0, :e_carnivore ⇒ 0.85, :Γh ⇒ [0.5, 0.5, 0.0, 0.0], :m_producer ⇒ 1.0, :
```

```
p_bio = model_parameters(A, functional_response = :bioenergetic, h = 1.0)
```

here we're using a type II functional response (`h = 1.0`).

We then define some initial biomasses `b0`, simulate and plot:



```
begin
    b0 = rand(size(A,1))
    sim_bio = simulate(p_bio, b0, stop = 500)
    plot(sim_bio[:t], sim_bio[:B], xlabel = "time", ylabel = "species biomass",
ylims = (0,1.1))
end
```

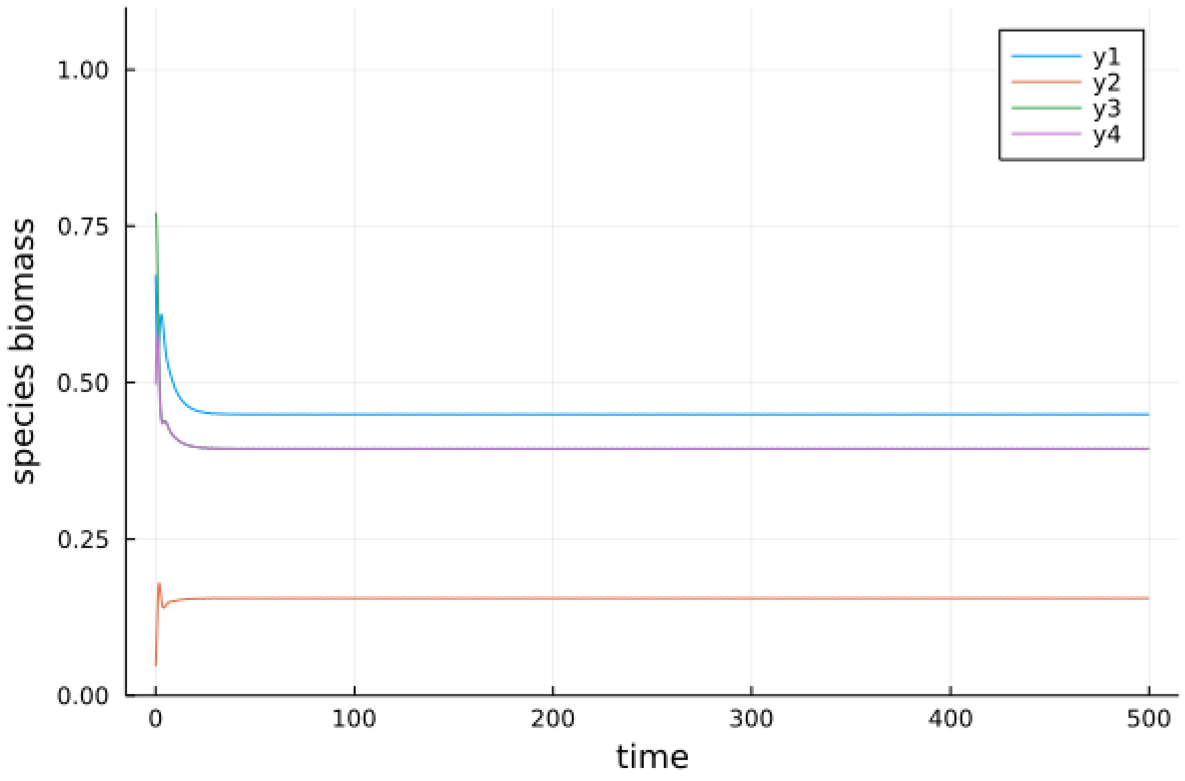
Using `:classical`

Fix parameters:

```
p_classical =
  Dict(:α ⇒ 1.0, :e_carnivore ⇒ 0.85, :Γh ⇒ [0.25, 0.25, 0.0, 0.0], :m_producer ⇒ 1.0,
```

```
p_classical = model_parameters(A, functional_response = :classical, h = 2.0)
```

here we're using a type III functional response (`h = 2.0`), we then simulate and plot:



```
begin
    sim_classical = simulate(p_classical, b0, stop = 500)
    plot(sim_classical[:t], sim_classical[:B], xlabel = "time", ylabel = "species
biomass", ylims = (0,1.1)) # Dynamics are much more stable!
end
```

Worked example

To demonstrate how the BEFW model can be used to investigate the effect of different functional responses on population and community dynamics, we're going to provide a coded example. First we define some functional response types:

```
fr_type =
  [(h = 1.0, c = 0.0, name = "Type II"), (h = 2.0, c = 0.0, name = "Type III"), (h = 1.0, 
```

```
fr_type = [(h = 1.0, c = 0.0, name = "Type II")
         , (h = 2.0, c = 0.0, name = "Type III")
         , (h = 1.0, c = 1.0, name = "Type II with predator interference")]
# Remember, the h and c parameters dictate the shape and the type of the functional
response being used (e.g. type II: h = 1 and c = 0).
```

fix the number of repetitions:

```
reps = 10
```

```
reps = 10
```

and create an empty array object to store the outputs:

```
df_outputs =   []
```

```
df_outputs = []
```

Moreover, to make things interesting we're going to add a range of consumer-resource body mass ratios (Z) to our experiment:

```
mass_ratio =   [1.0, 10.0, 100.0]
```

```
mass_ratio = [1.0, 10.0, 100.0]
```

We then generate some initial networks using the niche model, each of which contains 20 species with a connectance value of 0.2:

```
begin
    # list to store networks
    global networks = []
    # monitoring variable
    global l = length(networks)
    # while loop
    while l < reps
        # generate network
        A_bool = EcologicalNetworks.nichemodel(20,0.2)
        # 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 == 0.2
            push!(networks, A)
            # save network is co = 0.15
        end
        global l = length(networks)
    end
end
```

```
[20×20 Matrix{Int64}:                                                       , 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
  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
  1   1   1   1   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   1   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   1   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0    0   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   0   0    0   1   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    1   1   1   1   1   1   1   1
  0   0   0   1   1   1   1   1   1   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   1   1   1   1   1   1   1   0   0   0   0    0   0   0   0   0   1   1   1
  0   0   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   0
  1   1   1   1   1   1   1   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   0   0   0   0   1   1   1   1   1   1   1   1   1   1   1    0   0   0   0   1   1   1   1
```

```
networks
```

We then use nested `for` loops to loop over our experimental design, `simulate` dynamics and store metrics (`biomass`, `species_persistence` and `population_stability`) of interest:

```
for f in fr_type
    for z in mass_ratio
        for (i, a) in enumerate(networks)
            # fix model parameters
            p = model_parameters(a, h = f.h, c = [f.c], Z = z, functional_response = :bioenergetic)
            # provide some initial biomasses
            bio = rand(size(a,1))
            # simulate
            s = simulate(p, bio, stop = 1000)
            # calculate outputs
            out = (fr = f.name, Z = z, id = i, cv = population_stability(s, last = 250), persistence = species_persistence(s, last = 250), biomass = total_biomass(s, last = 250))
            # push! to store
            push!(df_outputs, out)
            # print some stuff - see how the simulation is progressing
            fr = f.name
            println(("fr = $fr", "Z = $z", "network = $i"))
        end
    end
end
```

Above, we've used the `:bioenergetic` approach but this could easily be changed to `:classical` and re-run.

We then coerce `df_outputs` to be a dataframe:

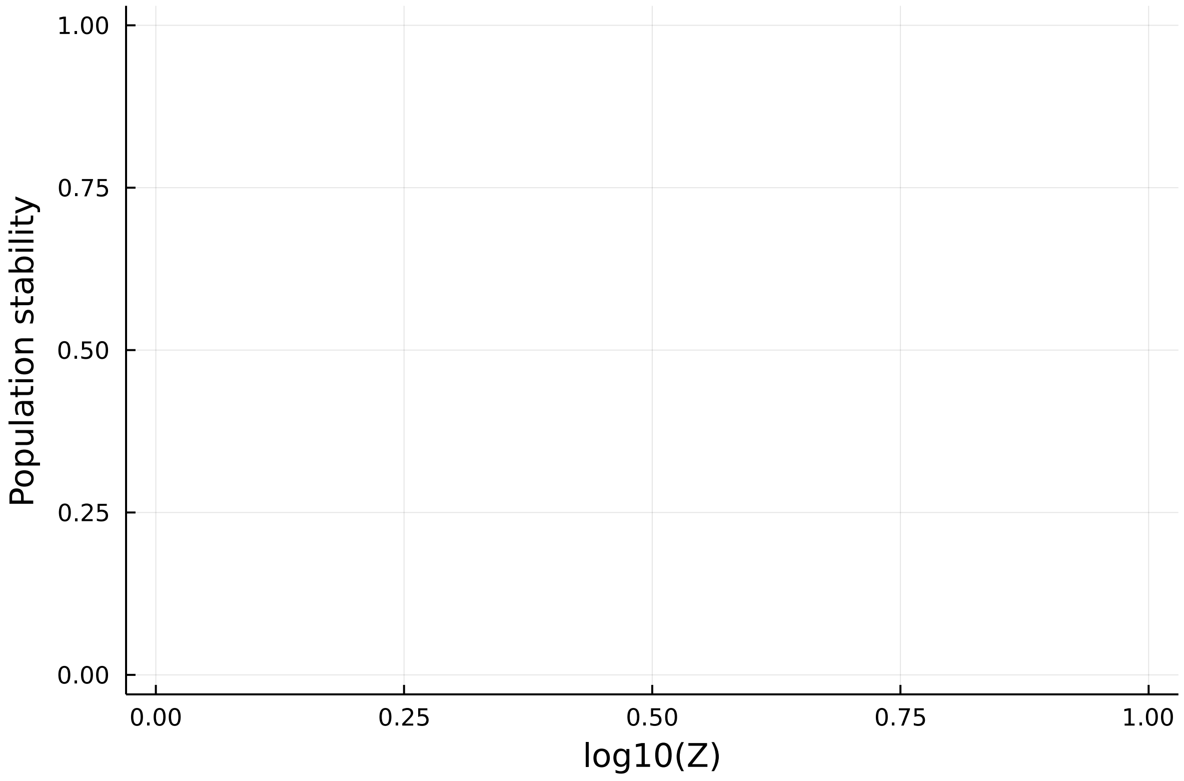
`df =`

	fr	Z	id	cv	persistence	bi
1	"Type II"	1.0	1	-0.050942	0.3	1.
2	"Type II"	1.0	2	-0.0294244	0.3	1.
3	"Type II"	1.0	3	-0.400052	0.65	2.
4	"Type II"	1.0	4	-0.948115	0.3	1.
5	"Type II"	1.0	5	-0.00193621	0.35	2.
6	"Type II"	1.0	6	-0.00416997	0.25	0.
7	"Type II"	1.0	7	-0.402353	0.35	1.
8	"Type II"	1.0	8	-0.00880921	0.3	0.
9	"Type II"	1.0	9	-0.521461	0.45	1.
10	"Type II"	1.0	10	-0.00211502	0.65	3.
	more					
90	"Type II with predator interference"	100.0	10	-5.26343e-5	0.8	13.

```
df = DataFrame(df_outputs)
```

and plot mean population stability by `mass_ratio` and `fr_type`:

```
plt =
```



```
plt = plot([NaN], [NaN], xlabel = "log10(Z)",
                ylabel = "Population stability", label = "") #prepare empty plot
```

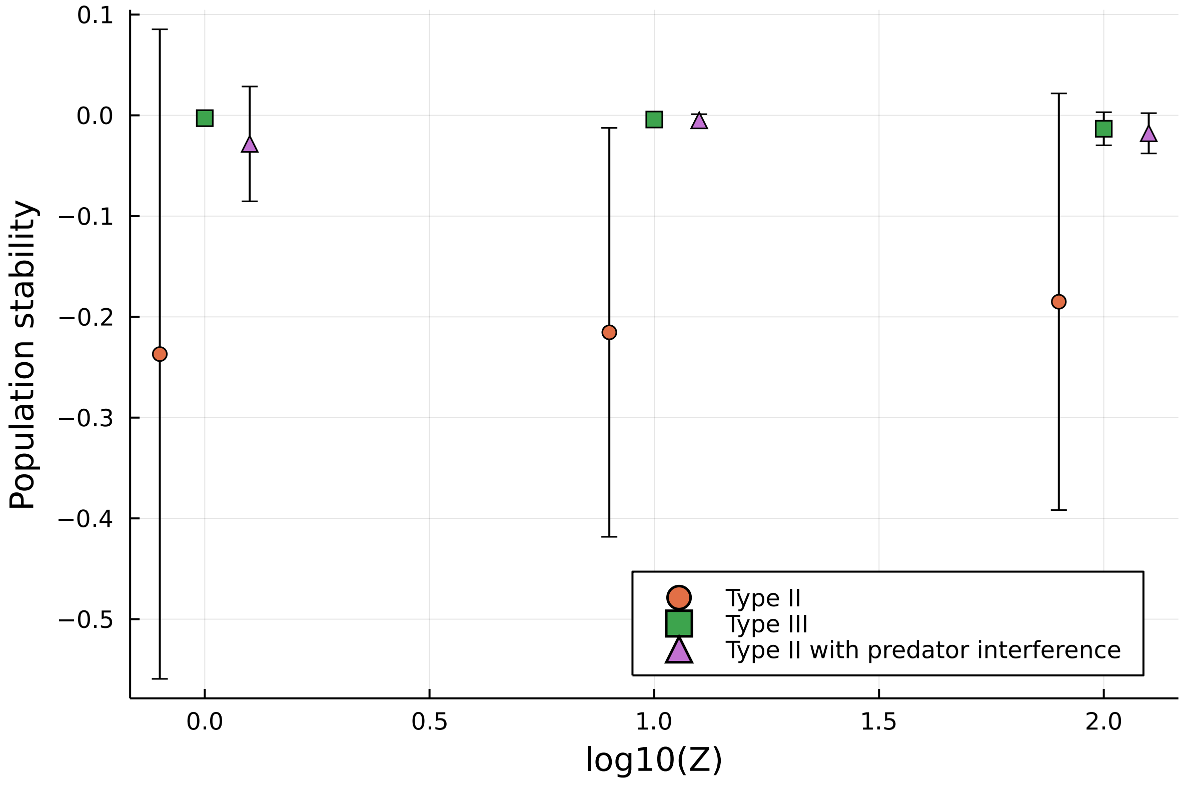
```
mtypes =  [:circle, :rect, :utriangle]
```

```
mtypes = [:circle, :rect, :utriangle]
```

```
jitterZ =  [-0.1, 0.0, 0.1]
```

```
jitterZ = [-0.1, 0, 0.1]
```

```
for (i,f) in enumerate(fr_type)
    mean_cv = []
    std_cv = []
    for z in mass_ratio
        tmp = df[(df.fr .== f.name) .& (df.Z .== z),:cv]
        tmp = tmp[.!isnan.(tmp)]
        push!(mean_cv, mean(tmp))
        push!(std_cv, std(tmp))
    end
    scatter!(log10.(mass_ratio) .+ jitterZ[i], mean_cv, markershape = mtypes[i],
label = f.name, yerror = std_cv, legend=:bottomright)
end
```



Population stability
0.1
0.0
−0.1
−0.2
−0.3
−0.4
−0.5
0.0
0.5
1.0
1.5
2.0
log10(Z)
● Type II
■ Type III
▲ Type II with predator interference

```
plt
```