Import package manager `Pkg` and activate:

```
import Pkg
```

```
Pkg.activate("../")
```

Temperature Effects

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

Global temperature is expected to increase by 1-4°C in the next century, a change that could have drastic impacts on ecological systems and the services they provide (**IPCC 2021**). Temperature enters ecological systems at the level of the individual, driving biological rates of metabolism, growth and consumption (**Simmons et al. 2021**). Despite this, the effects of increasing temperature are often felt at a range of ecological scales through reductions in individual fitness, changes in population-level traits (e.g. body size distributions), species-level extinctions, and the loss of ecological complexity (e.g. **Dell et al. 2011**; **Pawar et al. 2016**; **Fussmann et al. 2016**; **Tabi et al. 2019**).

In previous versions of the BEFW model, biological rates are modelled as a function of mass, and are therefore temperature independent, limiting the utility of the model to investigate temperature effects (**Delmas et al. 2016**). In this tutorial, we will explain how this limitation is overcome, and how, via Boltzmann Arrhenius terms and the Metabolic Theory of Ecology (**Gillooly et al. 2001**; **Brown et al. 2004**; **Savage et al. 2004**), biological rates in the BEFW model can be altered to account for temperature effects. As in previous tutorials, we will first explain the theory (including maths and plots), then provide a worked example that is explicity designed to demonstrate how the BEFW model can be used to investigate the effects of temperature on population and community dynamics.

Load packages

You'll need the following packages for this tutorial:

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

```
MersenneTwister(37)
```

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

Quick version check

Again, we recommend quickly checking that you are using the current developmental branch of the BEFW model. To do this, execute `Pkg.status()` in the REPL, you should see

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

If you don't see the above, use `Pkg.rm('BioEnergeticFoodWebs')` and `Pkg.add('BioEnergeticFoodWebs#dev-2.0.0')` to remove and reinstall the correct version of the package. A future you will thank you :)!

The theory

The BEFW model, and many other ecological models, lean heavily on the assumption that biological rates scale allometrically with body mass. By assuming this, the BEFW model simpifies the estimation of biological rates and allows complex dynamics to be simulated with relative ease. Alongside mass, we also expect biological rates to vary as a function of temperature (**Brown et al. 2004**). The inclusion of this dependency can be achieved in several ways (see **BEFW documentation**), however, the most common is via the Boltzmann Arrhenius equation:

$$q_i(T) = q_0 * M_i^{\beta} * exp(E - \frac{T_0 - T}{kT_0T})$$

This equation can be spilt into two parts: (1) an allometric relation between the mass M of species i and a given rate q and (2) an added term (the Boltzmann term) that describes how this relation is influenced by changes in temperature T. Here, q_0 is the intercept of the allometric relationship and β is the exponent. Both parameters are often based on empirical observations (e.g. **Ehnes et al. 2011**) with β describing the effect of M on q independent of T. The parameters for the Boltzmann term include k the Boltzmann's constant, E the mean activation energy of q and T_0 which is the reference temperature (20°C = 293K). All parameter values are listed in Table 1:

Table 1. Boltzmann Arrhenius parameters (see also **Binzer et al. 2012** & **Binzer et al. 2016**).

.	r_i	K_i	x_i	ar_{ij}	ht_{ij}
q_0	-15.68		-16.54	-13.1	9.66
β_i	-0.25	0.28	-0.31	0.25	-0.45
β_j				-0.8	0.47
E	-0.84	0.71	-0.69	-0.38	0.26

Here, r is growth rate, x is metabolic rate, and ar and ht are attack rates and handling times, respectively. K signifies the carrying capacity of the system, which can also be described as a function of M and T. See the next tutorial *Enrichment in the BEFW* for more details.

When using the Boltzmann-Arrhenius equations to estimate biological rates, we usually use the classical version of the functional response (`:classical` , see tutorial *Functional response*). This is because it is often easier to find empiricaly derived values for ar_{ij} and ht_{ij} than it is to find values for y (consumer-specific maximum consumption rates) and $B0$ (half saturation densities). It is important to remember that both attack rates and handling times describe an interaction between a consumer (j) and a resource (i), so the Boltzmann-Arrhenius equation for these two rates becomes:

$$q_ji(T) = q_0 * M_i^{\beta_i} * M_j^{\beta_j} * exp(E - \frac{T_0 - T}{kT_0T})$$

where M_i and M_j are the masses of the resource species and the consumer species, respectively.

Temperature effects in the BEFW

In BEFW model, four biological rates (x, r, ar and ht) can be modelled using Boltzmann Arrhenius equations. To illistrate the joint effects of mass and temperature on these rates, we first generate a range of temperatures in Kelvin (1°C = 273K):

```
T =
  [273.15, 274.15, 275.15, 276.15, 277.15, 278.15, 279.15, 280.15, 281.15, 282.15, 283.15,
```

```
T = [0:1:40;] .+ 273.15 # units = Kelvin
```

a range of masses:

```
M =   [10, 10000, 20000]
```

```
M = [10, 10000, 20000] # units = grams
```

and implement/plot on a rate by rate basis.

(1) Metabolic rate (x)

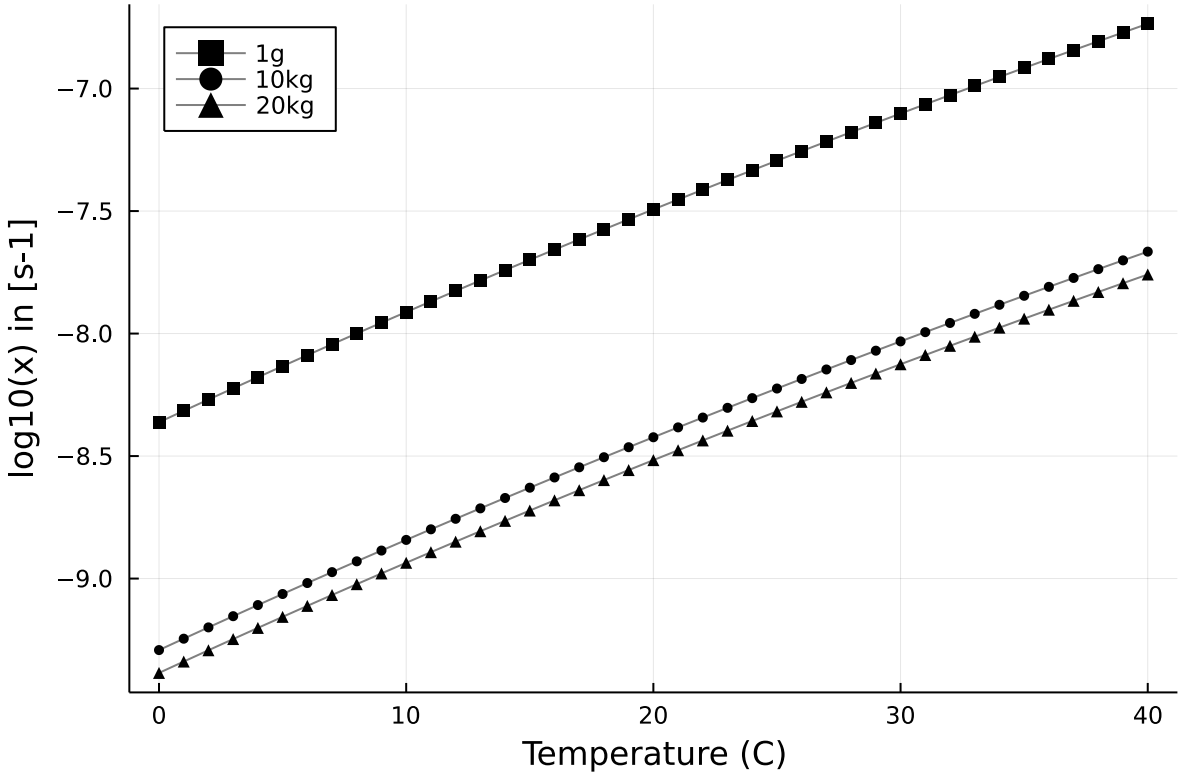
```
ScaleMetabolism (generic function with 1 method)
```

```
function ScaleMetabolism(M, T)
    x0 = exp(-16.54) # intercept
    sx = -0.31 # allometric exponent (mass-dependence)
    Ex = -0.69 # activation energy
    T0 = 293.15 # 20 Celsius in Kelvins (reference temperature)
    k = 8.617e-5 # Boltzman constant
    # return metabolic rate vector in 1/s
    return x0 .* (M .^ sx) .* exp(Ex .* ((T0 .- T) ./ (k .* T .* T0)))
end
```

```
x = 41×3 Matrix{Float64}:
 NaN  NaN  NaN
 NaN  NaN  NaN
 NaN  NaN  NaN
 NaN  NaN  NaN
 NaN  NaN  NaN
 NaN  NaN  NaN
 NaN  NaN  NaN
   ⋮
 NaN  NaN  NaN
 NaN  NaN  NaN
 NaN  NaN  NaN
 NaN  NaN  NaN
 NaN  NaN  NaN
 NaN  NaN  NaN
```

```
x = fill(NaN, length(T), length(M)) # preallocate an empty array
```

```
for (i,t) in enumerate(T)
    for (j,m) in enumerate(M)
        x[i,j] = ScaleMetabolism(m,t)
    end
end
```




```
plot(T .- 273.15, log10.(x), markershape = [:rect :circle :utriangle], ms = 3, mc =
:black, labels = ["1g" "10kg" "20kg"], lc = :grey, legend = :topleft, xlabel =
"Temperature (C)", ylabel = "log10(x) in [s-1]") # plot
```

(2) Growth rate (r)

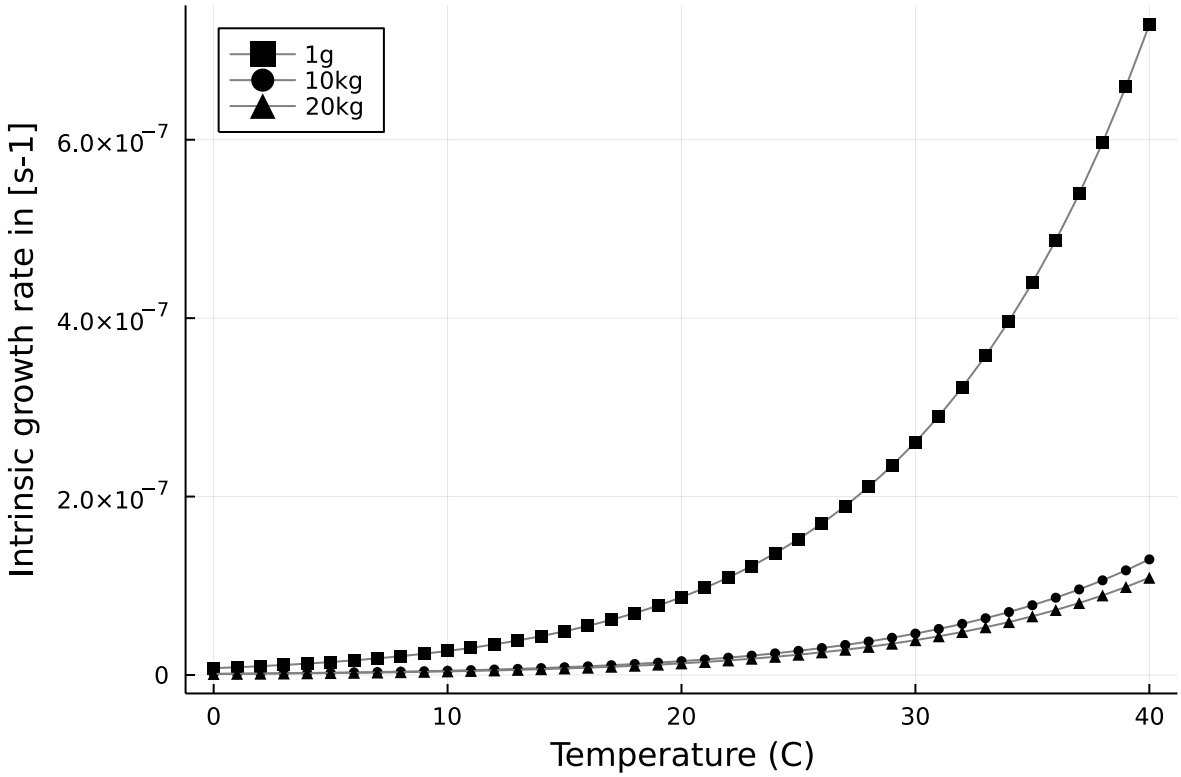
```
ScaleGrowth (generic function with 1 method)
```

```
function ScaleGrowth(M, T)
    r0 = exp(-15.68) # intercept
    βr = -0.25 # allometric exponent (mass-dependence)
    Er = -0.84 # activation energy
    T0 = 293.15 # 20 Celsius in Kelvins (reference temperature)
    k = 8.617e-5 # Boltzman constant
    # return growth rate vector in 1/s
    return r0 .* (M .^ βr) .* exp(Er .* ((T0 .- T) ./ (k .* T .* T0)))
end
```

```
r = 41×3 Matrix{Float64}:
 NaN  NaN  NaN
 NaN  NaN  NaN
 NaN  NaN  NaN
 NaN  NaN  NaN
 NaN  NaN  NaN
 NaN  NaN  NaN
 NaN  NaN  NaN
   ⋮
 NaN  NaN  NaN
 NaN  NaN  NaN
 NaN  NaN  NaN
 NaN  NaN  NaN
 NaN  NaN  NaN
 NaN  NaN  NaN
```

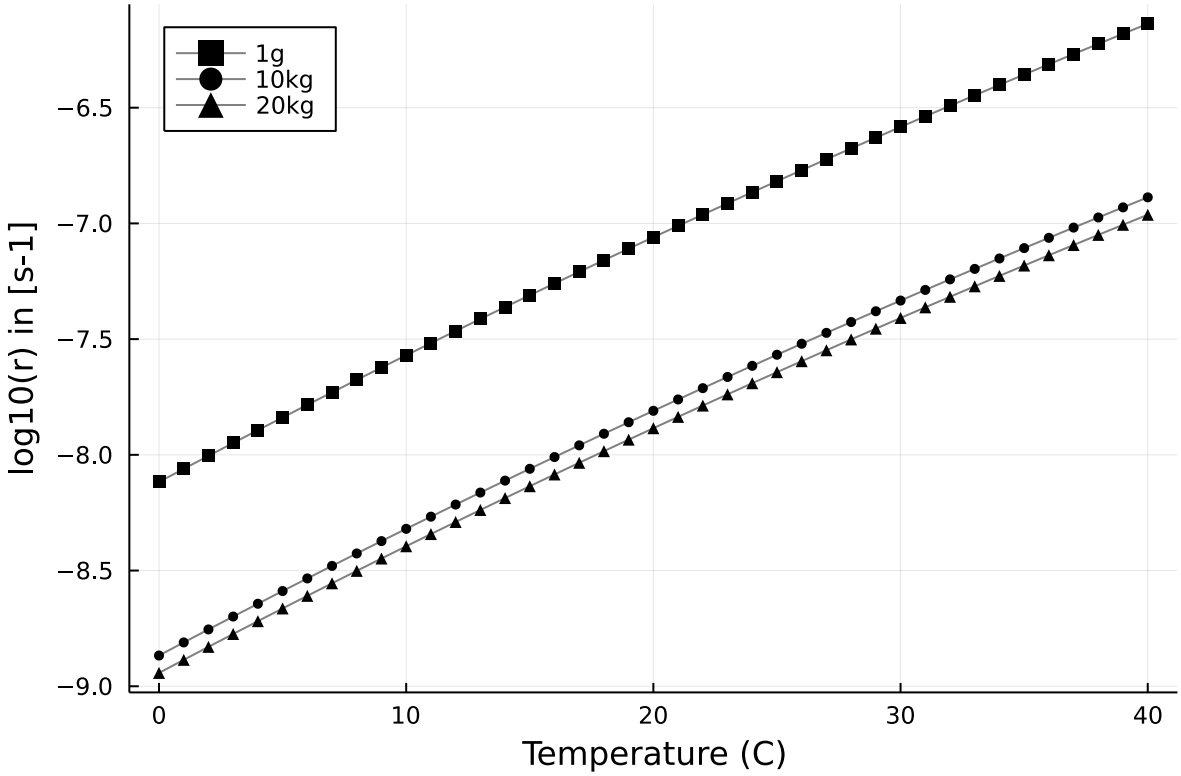
```
r = fill(NaN, length(T), length(M)) # preallocate an empty array
```

```
for (i,t) in enumerate(T)
    for (j,m) in enumerate(M)
        r[i,j] = ScaleGrowth(m,t)
    end
end
```




```
plot(T .- 273.15, r, markershape = [:rect :circle :utriangle], ms = 3, mc = :black, labels = ["1g" "10kg" "20kg"], lc = :grey, legend = :topleft, xlabel = "Temperature (C)", ylabel = "Intrinsic growth rate in [s-1]")
```

It's often easier to plot r on the log scale:




```
plot(T .- 273.15, log10.(r), markershape = [:rect :circle :utriangle], ms = 3, mc =
:black, labels = ["1g" "10kg" "20kg"], lc = :grey, legend = :topleft, xlabel =
"Temperature (C)", ylabel = "log10(r) in [s-1]")
```

(3) Attack rates (ar)

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

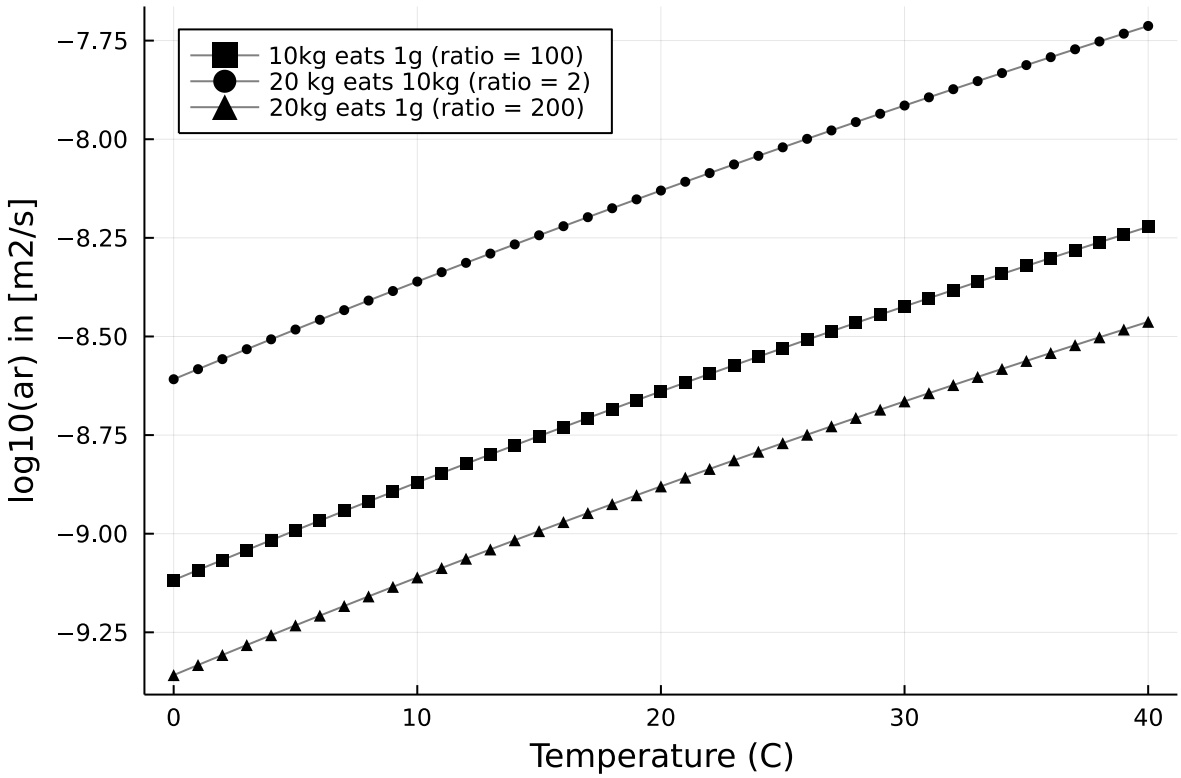
```
function ScaleAttack(m, T)
    a0 = exp(-13.1) # intercept
    βres = 0.25 # resource allometric exponent (mass-dependence for the resource)
    βcons = -0.8 # consumer allometric exponent (mass-dependence for the consumer)
    Ea = -0.38 # activation energy
    T0 = 293.15 # 20 Celsius in Kelvins (reference temperature)
    k = 8.617E-5 # Boltzman constant
    boltz = exp(Ea * ((T0-T)/(k*T*T0))) # calculate the Boltzman term (temperature
dependence term)

    aij = zeros(length(m), length(m))
    for i in eachindex(m) # i = rows => consumers
      for j in eachindex(m) # j = cols => resources
        mcons = m[i] ^ βcons # mass scaling for consumers
        mres = m[j] ^ βres # mass scaling for resources
        aij[i,j] = a0 * mres * mcons * boltz
      end
    end
    # return attack rate matrix in m2/s
    return aij
end
```

Remember, attack rates (ar) are defined for a consumer-resource interaction, so they depend on the masses of both species:

```
ar = 41×3 Matrix{Float64}:
      NaN  NaN  NaN
      NaN  NaN  NaN
      NaN  NaN  NaN
      NaN  NaN  NaN
      NaN  NaN  NaN
      NaN  NaN  NaN
      NaN  NaN  NaN
        ⋮
      NaN  NaN  NaN
      NaN  NaN  NaN
      NaN  NaN  NaN
      NaN  NaN  NaN
      NaN  NaN  NaN
      NaN  NaN  NaN
```

```
ar = fill(NaN, length(T), length(M)) # preallocate an empty array
```

```
for (i,t) in enumerate(T)
    ar_t = ScaleAttack(M, t) # returns a matrix with an attack rate value for all possible interactions
    # fix interactions (who eats who):
    ar[i,1] = ar_t[2,1] # sp. 2 eats sp. 1
    ar[i,2] = ar_t[3,2] # sp. 3 eats sp. 2
    ar[i,3] = ar_t[3,1] # sp. 3 eats sp. 1
end
```




```
plot(T .- 273.15, log10.(ar), markershape = [:rect :circle :utriangle], ms = 3, mc = :black, labels = ["10kg eats 1g (ratio = 100)" "20 kg eats 10kg (ratio = 2)" "20kg eats 1g (ratio = 200)"], lc = :grey, legend = :topleft, xlabel = "Temperature (C)", ylabel = "log10(ar) in [m2/s]")
```

(5) Handling times (ht)

Below we describe the implementation of the *power* method for estimating handling time. When working with the ADBM model (for food web generation or rewiring) you may want to use the *ratio* method instead. See **Petchey et al. 2008** for more details on the two methods.

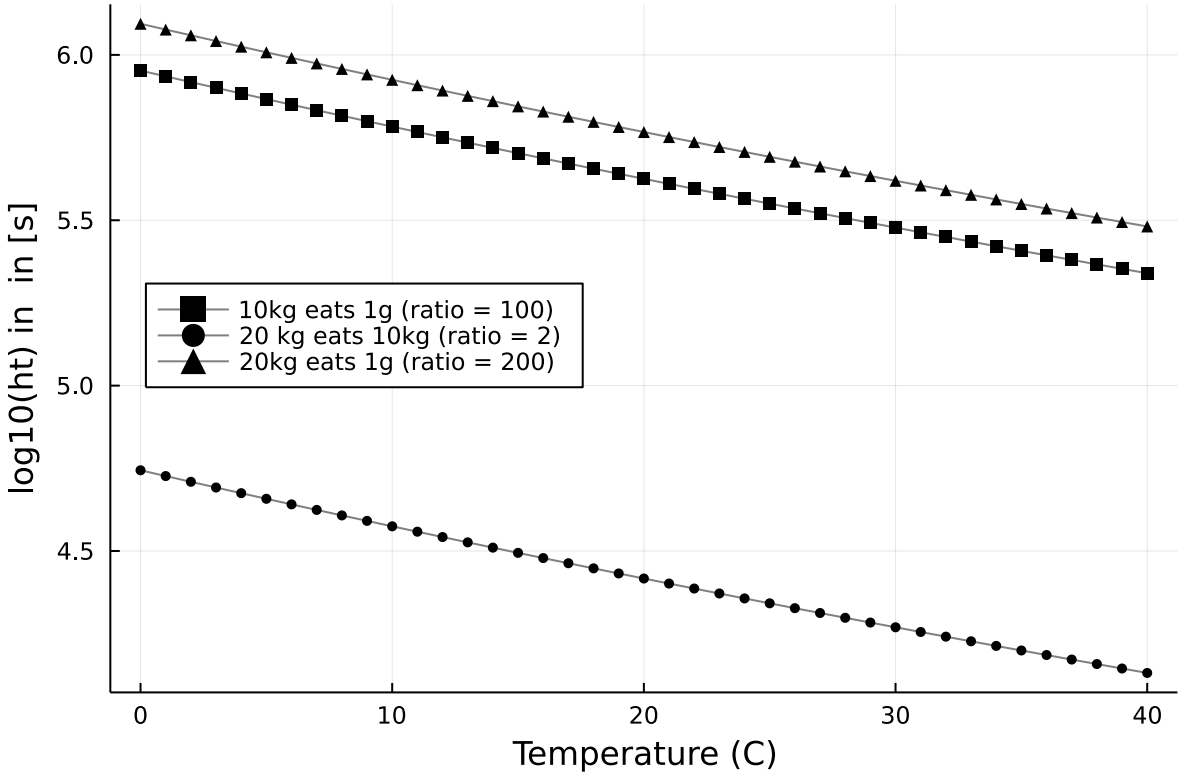
```
ScaleHandling (generic function with 1 method)
```

```
function ScaleHandling(m, T)
    h0 = exp(9.66) # intercept
    βres = -0.45 # resource allometric exponent (mass-dependence for the resource)
    βcons = 0.47 # consumer allometric exponent (mass-dependence for the consumer)
    Eh = 0.26 # activation energy
    T0 = 293.15 # 20 Celsius in Kelvins (reference temperature)
    k = 8.617E-5 # Boltzman constant
    boltz = exp(Eh * ((T0-T)/(k*T*T0))) # calculate the Boltzman term (temperature dependence term)

    hij = zeros(length(m), length(m))

    for i in eachindex(m) # i = rows => consumers
        for j in eachindex(m) # j = cols => resources
            mcons = m[i] ^ βcons # mass scaling for consumers
            mres = m[j] ^ βres # mass scaling for resources
            hij[i,j] = h0 * mres * mcons * boltz
        end
    end

    #return handling time matrix in [s]
    return hij
end
```

```
ht = 41×3 Matrix{Float64}:
 NaN  NaN  NaN
 NaN  NaN  NaN
 NaN  NaN  NaN
 NaN  NaN  NaN
 NaN  NaN  NaN
 NaN  NaN  NaN
 NaN  NaN  NaN
   ⋮
 NaN  NaN  NaN
 NaN  NaN  NaN
 NaN  NaN  NaN
 NaN  NaN  NaN
 NaN  NaN  NaN
 NaN  NaN  NaN
```

```
ht = fill(NaN, length(T), length(M)) # preallocate an empty array
```

```
for (i,t) in enumerate(T)
    ht_t = ScaleHandling(M, t) # returns a matrix with a handling time value for all possible interactions
    # fix interactions (who eats who):
    ht[i,1] = ht_t[2,1] # sp. 2 eats sp. 1
    ht[i,2] = ht_t[3,2] # sp. 3 eats sp. 2
    ht[i,3] = ht_t[3,1] # sp. 3 eats sp. 1
end
```




```
plot(T .- 273.15, log10.(ht), markershape = [:rect :circle :utriangle], ms = 3, mc = :black, labels = ["10kg eats 1g (ratio = 100)" "20 kg eats 10kg (ratio = 2)" "20kg eats 1g (ratio = 200)"], lc = :grey, legend = :left, xlabel = "Temperature (C)", ylabel = "log10(ht) in  in [s]")
```

ScaleRates!()

Each of the above functions (e.g. `ScaleHandling()`) provide the code neccessary to scale biological rates by mass and temperature. Each of these functions produce a vector or matrix which can be supplied directly to a `model_parameters` object using the following code:

```
20×20 Matrix{Float64}:
 15677.8        15677.8         5562.69  …   2693.3    3313.48  15677.8
 15677.8        15677.8         5562.69      2693.3    3313.48  15677.8
 46268.4        46268.4        16416.7       7948.49   9778.78  46268.4
 15677.8        15677.8         5562.69      2693.3    3313.48  15677.8
 15677.8        15677.8         5562.69      2693.3    3313.48  15677.8
 15677.8        15677.8         5562.69  …   2693.3    3313.48  15677.8
     1.01649e5      1.01649e5  36066.4      17462.4   21483.4       1.01649e5
     ⋮                                   ⋱
     1.95863e5      1.95863e5  69494.6      33647.4   41395.3       1.95863e5
 79484.9        79484.9        28202.3   …  13654.8   16799.0  79484.9
 15677.8        15677.8         5562.69      2693.3    3313.48  15677.8
 98692.6        98692.6        35017.5       16954.5   20858.6  98692.6
 79484.9        79484.9        28202.3       13654.8   16799.0  79484.9
 15677.8        15677.8         5562.69      2693.3    3313.48  15677.8
```

```
begin
    A = BioEnergeticFoodWebs.nichemodel(20,0.15) # propose a network
    Z = 10.0 # assign mass based on Z
    T20 = 293.15 # fix temperature at 20C
    p = model_parameters(A, Z = Z, T = T20)
    M20 = p[:bodymass] # assign body mass vector to M
    p[:x] = ScaleMetabolism(M20, T20)
    p[:r] = ScaleGrowth(M20, T20)
    p[:ar] = ScaleAttack(M20, T20)
    p[:ht] = ScaleHandling(M20,T20)
end
```

`p` will now contain updated values for the four biological rates each of which have been scaled to a temperature of 20°C (e.g. the value of $T20$).

To make this process easier, we've parcelled up the above functions into a larger function called `ScaleRates!()`. `ScaleRates!` takes the `p` object as an input and requires a value for the carrying capacity $k0$ to be specified (details can be found in the next tutorial *Enrichment in the BEFW*):

```
[10.0, 10.0, 19.0546, 10.0, 10.0, 10.0, 30.4534, 30.4534, 26.3027, 22.3872, 23.6229, 26.3
```

```
begin
    include("common_utils.jl") # loads the content of the common_utils.jl script
    p_new = model_parameters(A, Z = Z, T = T20, functional_response = :classical)
    ScaleRates!(p_new, 10.0) # k0 = 10.0
end
```

`ScaleRates!` acts by directly transforming the biological rates within `p` based on `p[:bodymass]` and `p[:T]`. You'll notice the addition of a `!` at the end of the function name, this is a Julia thing and means that the function transforms the input object instead of creating a new object. This function and several others have been compiled into a utility script called `common_utils.jl` which is available via the **github repo**. The idea is that the lab will all work from, update, and share the `common_utils.jl` script and therefore work with the same toolbox.

Timesteps/Units

In previous versions of the BEFW, the timestep of the model is normalised to the growth rate of smallest producer. This means that the `start` and `stop` arguments in the `simulate()` function are somewhat arbitary and the critical aspect is to ensure that model runs for sufficent timesteps (e.g. `stop = 2000`) to achieve steady-state dynamics (i.e. an equilibrium).

Above, we have demonstrated that biological rates can be scaled by M and T based on empirical derived parameters. The units of these rates are detailed in Table 2:

Table 2. Units of temperature scaled biological rates.

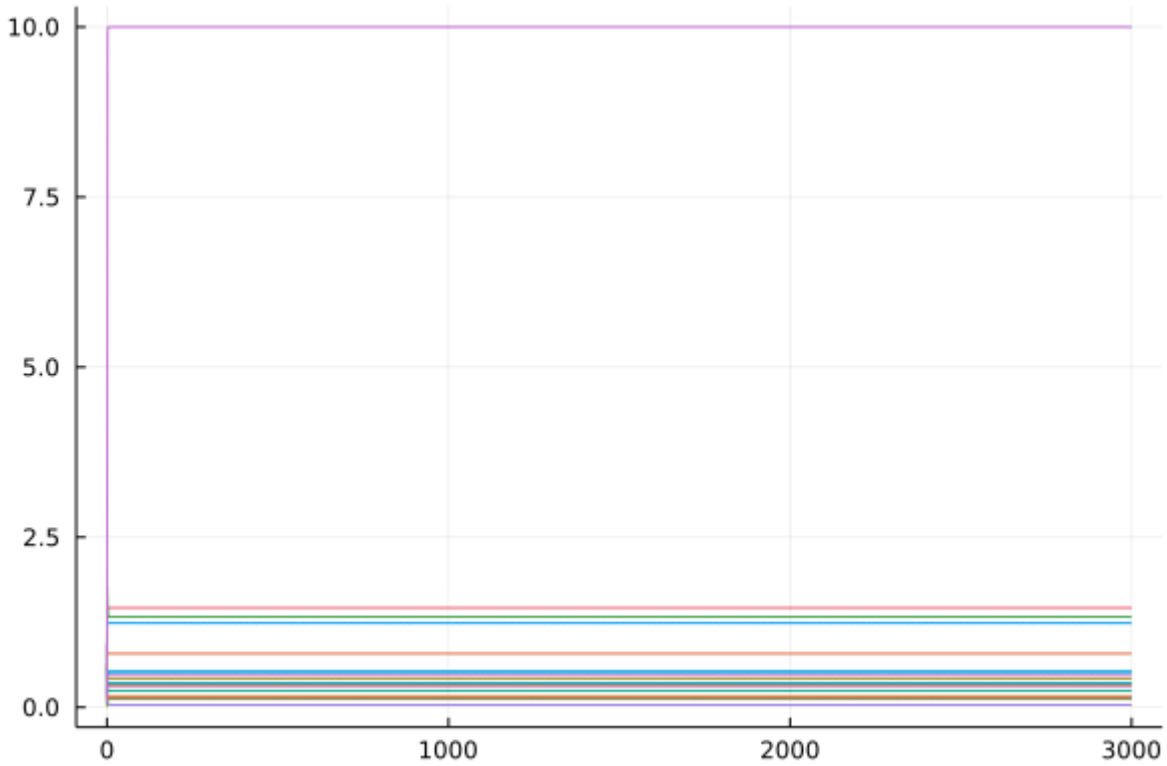
.	$unit$
x	1/s
r	1/s
ar	m2/s
ht	s

By including these rates in the BEFW model directly and not normalising, the newest version of the BEFW model simulates biomass (g/m2) dynamics in real time, whereby one timestep = one second (s). The dynamics of the BEFW model are slow, and as a result we recommend simulating for approximately three thousand years:

```
Dict(:p ⇒ Dict(:α ⇒ 1.0, :e_carnivore ⇒ 0.85, :Γh ⇒ [0.0, 0.0, 0.5, 0.0, 0.0,    more
```

```
begin
    p_new[:h] = 2.0 # Type III functional response to stabilise dynamics
    out = simulate(p_new, rand(20), stop = Int(60*60*24*365.25*3000), interval_tkeep
=     Int(60*60*24*365.25)) # simulate for 3000 years and save every year
end
```

The `interval_tkeep` argument sets the saving frequency of the dynamics. In theory, you could save every second, however, it will probably cause memory issues and will slow down the simulation. Consequently, we recommend saving at a time step that is reasonable and scales appropriately with the total length of the simulation. For instance, above we have simulated for 3000 years and saved every year.




```
plot(out[:t] ./ Int(60*60*24*365.25), out[:B], legend = false)
```

Worked example

To demonstrate how the scaling of biological rates with mass and temperature can be utilised in the BEFW model, we are going explore the joint effects of Z and T on population and community dynamics. We will first propose a range temperatures and Z values, initiate a network, and simulate. For simplicity, we're only going to use one network (`A` defined above) but this code could be easily extended to included multiple networks (see previous tutorials). Moreover, instead of outputting multiple metrics, we have chosen to output only total biomass and species persistence:

```
T_range =
  [273.15, 274.15, 275.15, 276.15, 277.15, 278.15, 279.15, 280.15, 281.15, 282.15, 283.15,
```

```
T_range = [0:1:40;] .+ 273.15 # temperature ranging from 0 to 40C
```

```
Z_range =   [10.0, 100.0, 1000.0, 10000.0, 100000.0]
```

```
Z_range = 10.0 .^ [1:1:5;] # consumer-resource mass ratio ranging from 10 to 100000
```

```
year = 31471200
```

```
year = Int(60*60*24*364.25) # 1 year in seconds
```

df =

T	Z	biomass	persistence

```
df = DataFrame(T = [], Z = [], biomass = [], persistence = []) # initialize outputs data frame
```

```
for z in Z_range
    for t in T_range
        println("T = $t and log(z) = $(log10(z))")
        p_tmp = model_parameters(A, Z = z, T = t, h = 2.0, functional_response = :classical)
        ScaleRates!(p_tmp, 10.0)
        s = simulate(p_tmp, rand(20), stop = year*3000, interval_tkeep = year)
        tmp = (T = t, Z = z, biomass = total_biomass(s, last = 500), persistence = species_persistence(s, last = 500))
        push!(df, tmp)
    end
end
```

In order to visualize our results as heatmaps (or a contour plots), we first reshape `df` to two 2D arrays:

```
TZ_array_biomass =
5×41 Matrix{Float64}:
 NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN  …  NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN
 NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN     NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN
 NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN     NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN
 NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN     NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN
 NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN     NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN
```

```
TZ_array_biomass = fill(NaN, length(Z_range), length(T_range)) #preallocate a 2d array for biomass values
```

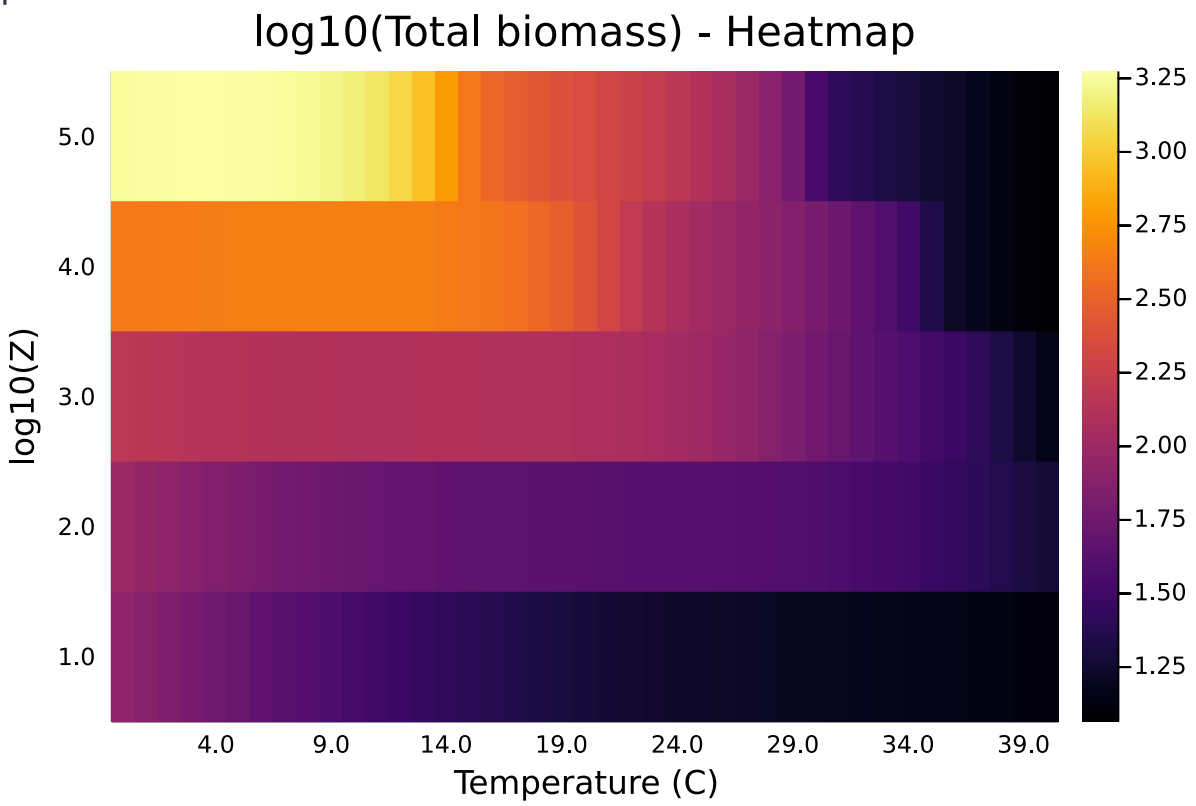
```
TZ_array_persistence =
5×41 Matrix{Float64}:
 NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN  …  NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN
 NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN     NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN
 NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN     NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN
 NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN     NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN
 NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN     NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN
```

```
TZ_array_persistence = fill(NaN, length(Z_range), length(T_range)) #preallocate a 2d array for persistence values
```

```
for (i,z) in enumerate(Z_range)
    for (j,t) in enumerate(T_range)
        tmp = df[(df.Z .== z) .& (df.T .== t),:]
        TZ_array_biomass[i,j] = tmp.biomass[1]
        TZ_array_persistence[i,j] = tmp.persistence[1]
    end
end
```

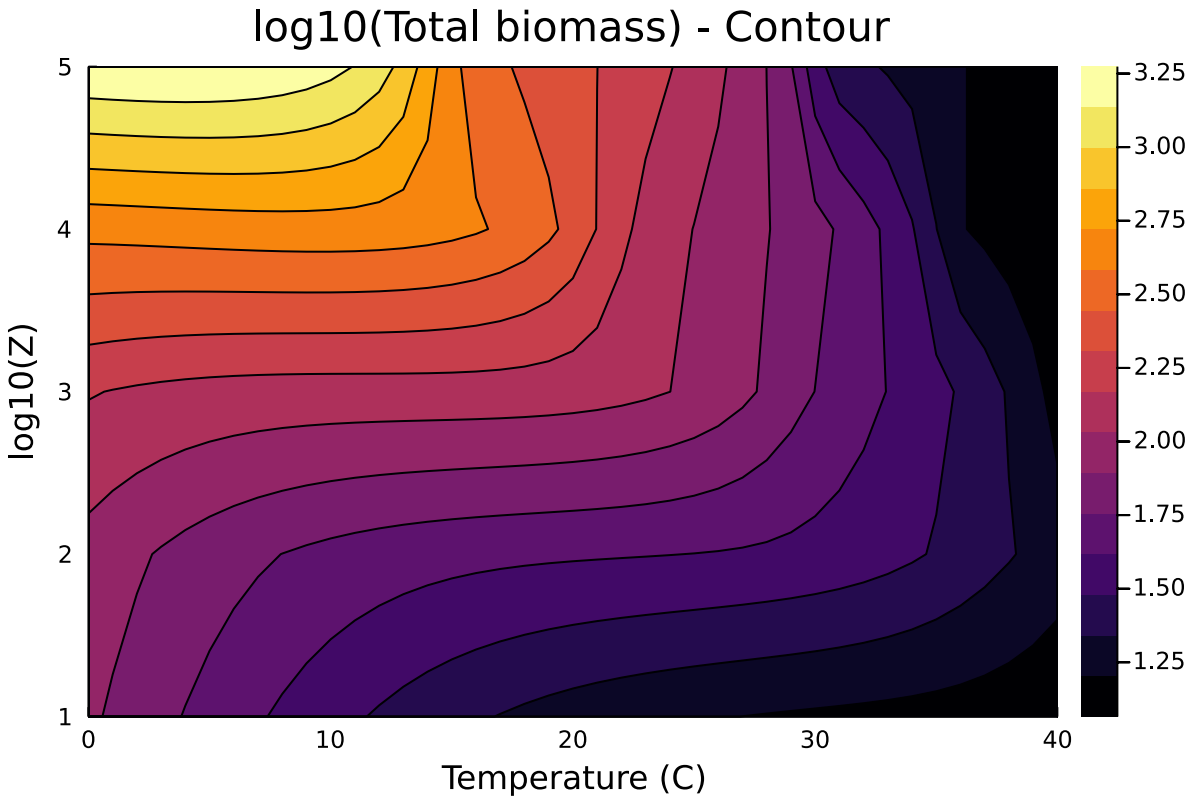
and plot:

p1 =



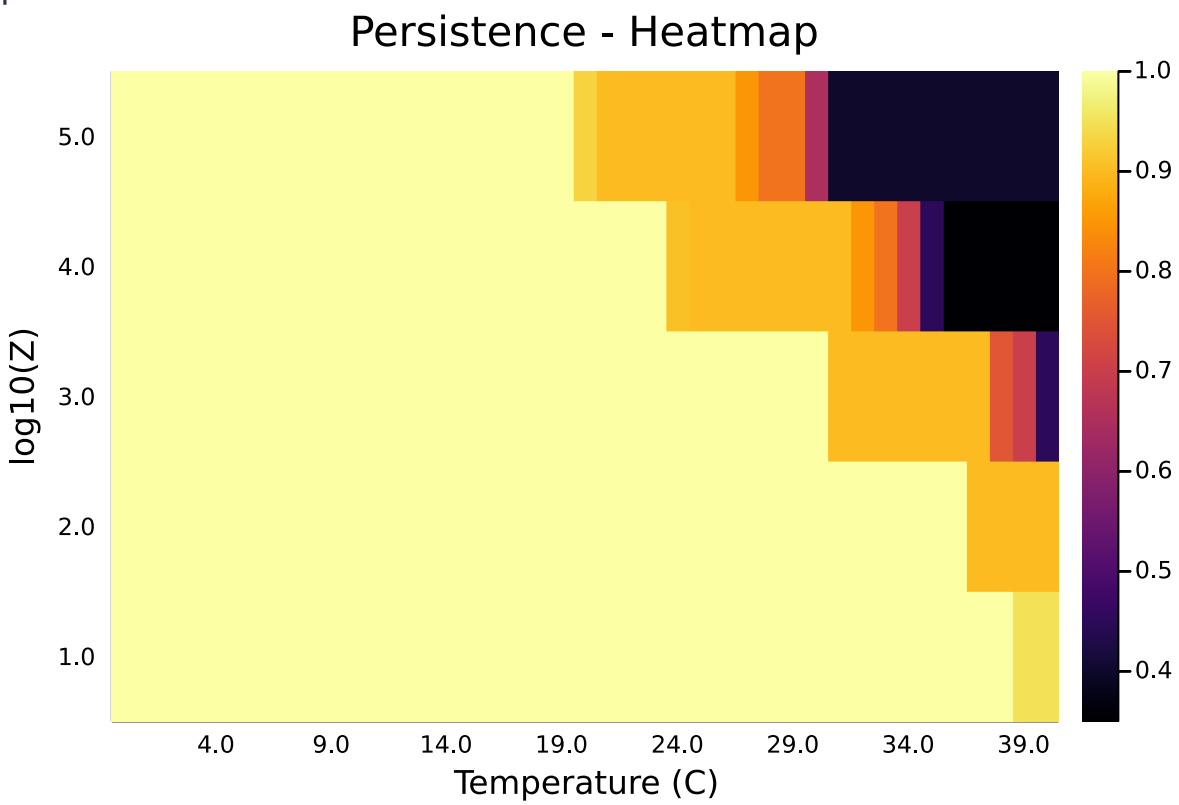

```
p1 = heatmap(string.(T_range .- 273.15), string.(log10.(Z_range)), log10.(TZ_array_biomass), title  = "log10(Total biomass) - Heatmap", xlabel = "Temperature (C)", ylabel = "log10(Z)")
```

p1b =




```
p1b = contour(T_range .- 273.15, log10.(Z_range), log10.(TZ_array_biomass), fill = true, title  = "log10(Total biomass) - Contour", xlabel = "Temperature (C)", ylabel = "log10(Z)")
```

```
p2 =
```




```
p2 = heatmap(string.(T_range .- 273.15), string.(log10.(Z_range)),
TZ_array_persistence, title  = "Persistence - Heatmap", xlabel = "Temperature (C)",
ylabel = "log10(Z)")
```

Above we see that total biomass reduces as temperature increases, and that this relationship appears more pronouced at smaller Z values. We also see that species persistence drops off at very high temperatures, especially when Z is high. This suggests that increases in temperature can cause species extinctions and results in communities with lower total biomass. It also shows that a relationship between Z and T is evident but that the nature of this relationship might differ based on the metric under investigation.

Questions to think about:

(1) Why does increasing temperature cause reductions in total biomass? Remember that body size is fixed in the BEFW model.

(2) When Z and T are high, why do species go extinct more often?

(3) How could this experiment be extended to gain further inference?