Homework 3 Solutions

> Due Date
>
> Thursday, 10/03/23, 9:00pm

Overview

Load Environment

The following code loads the environment and makes sure all needed packages are installed. This should be at the start of most Julia scripts.

```
import Pkg
Pkg.activate(@__DIR__)
Pkg.instantiate()
```

```
using Random
using CSV
using DataFrames
using Plots
using LaTeXStrings
using Distributions
```

Problems (Total: 50/60 Points)

Problem 1 (30 points)

The first step is to write function(s) to implement the dissolved oxygen simulation. With two releases, we can turn this into a two-box model, with the first box from the initial waste release ($x = 0$ km) to the second release ($x = 15$ km), and the second from the second release to the

end of the domain ($x = 50$ km). As a result, our lives will be easiest if we write a function to simulate each box with appropriate initial conditions, which we can then call for each river segment. An example of how this might look is below. Note that we need to compute B and N as well to get the appropriate initial conditions at the transition between boxes (and this might also help with debugging).

```
# mix_concentration: function to compute initial conditions by mixing inflow
 ↪  and new waste stream concentrations
# inputs:
#   - arguments ending in "_in" are inflow, those ending in "_st" are from
 ↪  the stream
#   - V is the volume (L/d), Q is the relevant concentration (mg/L); these
 ↪  should be Floats
# outputs:
#   - mixed concentration (a Float) in mg/L
function mix_concentration(V_in, Q_in, V_st, Q_st)
    Q_mix = ((V_in * Q_in) + (V_st * Q_st)) / (V_in + V_st)
    return Q_mix
end

# dissolved_oxygen: function to simulate dissolved oxygen concentrations for
 ↪  a given segment
# inputs:
#   - x: vector or range of downstream distances to simulate over
#   - C , B , N : initial conditions for DO, CBOD, and NBOD, respectively
 ↪  (mg/L)
#   - U: velocity of river (km/d)
#   - C  = saturation oxygen concentration (mg/L)
#   - ka, kc, kn: reaeration, CBOD decay, and NBOD decay rates, respectively
 ↪  (d^{-1})
function dissolved_oxygen(x, C , B , N , U, C , ka, kc, kn)

    # initialize vectors for C, B, and N
    # the zeros function lets us define a vector of the appropriate length
     ↪  with values set to zero
    C = zeros(length(x))
    B = zeros(length(x))
    N = zeros(length(x))

    # compute   values for the simulation
      = exp.(-ka * x / U)                                                    (1)
      = (kc / (ka - kc)) * (exp.(-kc * x / U) .-  )
```

```
      = (kn / (ka - kn)) * (exp.(-kn * x / U) .-  )

    # loop over values in x to calculate B, N, and C
    for (i, d) in pairs(x)                                                    ②
        B[i] = B  * exp(-kc * x[i] / U)
        N[i] = N  * exp(-kn * x[i] / U)
        C[i] = C  * (1 -  [i]) + (C  *  [i]) - (B  *  [i]) - (N  *  [i])
    end

    return (C, B, N)                                                          ③
end
```

① These will be vectors due to the broadcasting of `exp` and `-` over the `x` vector. We could also have computed the values in the loop below for each value of `x`.
② `pairs(x)` lets us directly iterate over indices (`i`) and values (`d`) in the vector `x`, rather than only iterating over indices and needing to look up the values `d=x[i]`.
③ While we don't need `B` and `N` for this solution, returning this tuple can be useful for debugging.

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

Next, let's simulate the concentrations. Hopefully this is intuitive, but one critical thing is that we need to compute the initial segment from $x = 0$ to 15 km, not just to 14, as $x = 15$ is the inflow for the initial condition of the segment after the second waste stream.

> **Note**
>
> I initially wrote this out as a script to debug, but then reformulated it as a function with an optional parameters for treatment of the two waste streams to solve Problems 2 and 3, which means I didn't have to copy and paste everything, possibly introducing new bugs.

```
# do_simulate: function to simulate dissolved oxygen concentrations over the
 ↪  entire river
# inputs:
#   - inflow: tuple with inflow properties: (Volume, DO, CBOD, NBOD)
#   - waste1: tuple with waste stream 1 properties: (Volume, DO, CBOD, NBOD)
#   - waste2: tuple with waste stream 2 properties: (Volume, DO, CBOD, NBOD)
#   - U: velocity of river (km/d)
#   - C  = saturation oxygen concentration (mg/L)
#   - ka, kc, kn: reaeration, CBOD decay, and NBOD decay rates, respectively
 ↪  (d^{-1})
```

```
function do_simulate(inflow, waste1, waste2, U, C , ka, kc, kn)
    # set up ranges for each box/segment
    x  = 0:1:15                                                    ①
    x  = (15:1:50) .- 15                                           ②

    V_inflow, C_inflow, B_inflow, N_inflow = inflow                ③
    V_ws1, C_ws1, B_ws1, N_ws1 = waste1
    V_ws2, C_ws2, B_ws2, N_ws2 = waste2

    # initialize storage for final C, B, and N
    # need to store d=0 so the total length should be d+1
    C = zeros(51)
    B = zeros(51)
    N = zeros(51)

    # compute initial conditions for first segment
    C  = mix_concentration(V_inflow, C_inflow, V_ws1, C_ws1)
    B  = mix_concentration(V_inflow, B_inflow, V_ws1, B_ws1)
    N  = mix_concentration(V_inflow, N_inflow, V_ws1, N_ws1)

    # conduct first segment simulation
    (C , B , N ) = dissolved_oxygen(x , C , B , N , U, C , ka, kc, kn)
    C[1:15] = C [1:end-1]                                          ④
    B[1:15] = B [1:end-1]
    N[1:15] = N [1:end-1]

    # compute initial conditions for second segment
    C  = mix_concentration(V_inflow + V_ws1, C [end], V_ws2, C_ws2)
    B  = mix_concentration(V_inflow + V_ws1, B [end], V_ws2, B_ws2)
    N  = mix_concentration(V_inflow + V_ws1, N [end], V_ws2, N_ws2)

    # conduct second segment simulation
    (C , B , N ) = dissolved_oxygen(x , C , B , N , U, C , ka, kc, kn)
    C[16:end] = C
    B[16:end] = B
    N[16:end] = N

    return (C, B, N)
end

# set variables for river dynamics
U = 6
```

```
C  = 10
ka = 0.55
kc = 0.35
kn = 0.25

# set initial parameters
C_inflow = 7.5 # DO concentration
B_inflow = 5.0 # CBOD
N_inflow = 5.0 # NBOD
V_inflow = 100 * 1_000 # volume converted to L
inflow = (V_inflow, C_inflow, B_inflow, N_inflow)

# set waste stream parameters
C_ws1 = 5.0
B_ws1 = 50.0
N_ws1 = 35.0
V_ws1 = 10 * 1_000
waste1 = (V_ws1, C_ws1, B_ws1, N_ws1)

C_ws2 = 4.0
B_ws2 = 45.0
N_ws2 = 35.0
V_ws2 = 15 * 1_000
waste2 = (V_ws2, C_ws2, B_ws2, N_ws2)

C, B, N = do_simulate(inflow, waste1, waste2, U, C , ka, kc, kn)
```

① The colon syntax sets up the range using the syntax `initial_value:stepsize:end_value`. In general a stepsize of 1 is implicit, but I've made it explicit here for illustration.
② This starts at 0 because we care about the distance from the initial condition, not the "absolute" distance.
③ Tuples (including multiple outputs from functions) can be unpacked into multiple variables this way to make the subsequent code more readable (versus just relying on indexing).
④ We don't need to store the last value because that occurs at the point of mixing with the next waste stream. We will use it to compute the mixed concentration at that point.

```
([7.2727272727272725, 6.718366940001292, 6.252736478441488, 5.865982487427475, 5.54922462545
```

Now we can plot the dissolved oxygen concentration, shown in Figure 1.

```
# create plot axis with labels, etc
p = plot(; xlabel="Distance Downriver (km)", ylabel="Dissolved Oxygen
 ↪  Concentration (mg/L)", legend=:top)                                    ①
plot!(p, 0:50, C, label="Simulated DO")
hline!([4], label="Regulatory Standard")
```

① These lines of code separate the creation of the plot axis (with labels, legend positions, etc) using `p = plot(...)` from the plotting of the data with `plot!(p, ...)`. You can do this all in a single `plot()` call, but this may sometimes make things more readable when there are a lot of style arguments for the axes.

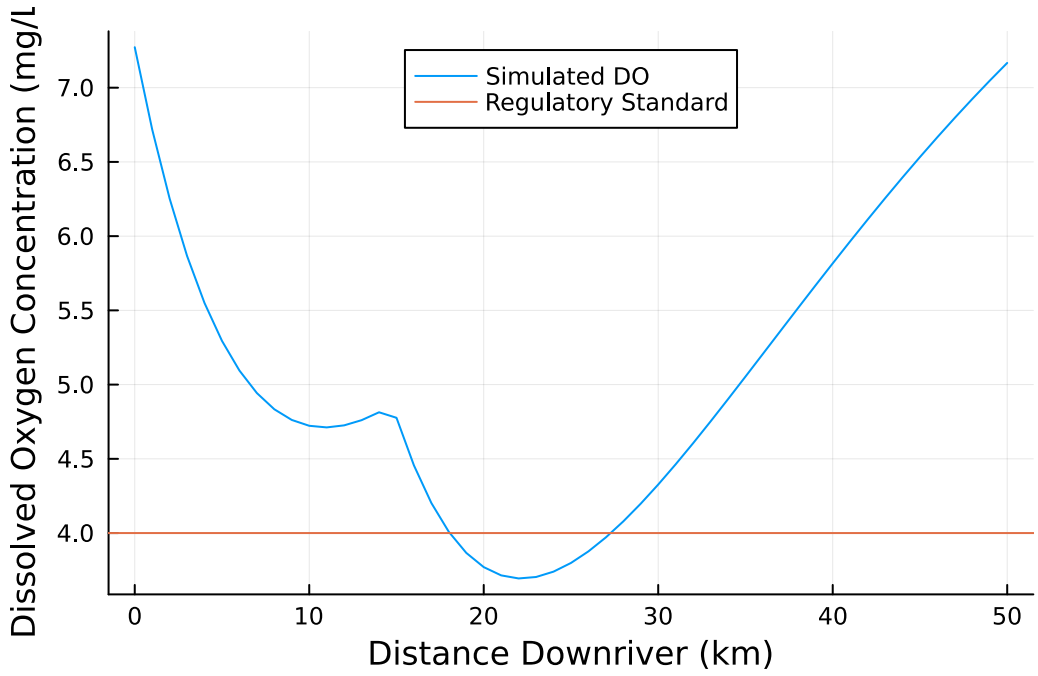

Figure 1: Simulated dissolved oxygen concentration for Problem 1 without treatment.

We can see that the DO concentration falls below the regulatory standard of 4 mg/L before 20 km downstream. To find the minimum value, we can use the `minimum()` function.

```
@show minimum(C);
```

```
minimum(C) = 3.694312052808094
```

So the minimum value is 3.7 mg/L.

To determine the minimum treatment of the waste streams needed to maintain compliance, we will write a function which will apply treatment levels `eff1` and `eff2` and evaluate our function above, returning the minimum DO concentration.

```
# waste_treat: function to simulate DO with treated discharges; treatments
 ↪  apply to CBOD and NBOD released
# inputs:
#   - eff1: efficiency of treatment for waste stream 1 (as a decimal)
#   - eff2: efficiency of treatment for waste stream 2 (as a decimal)
#   - inflow: tuple with inflow properties: (Volume, DO, CBOD, NBOD)
#   - waste1: tuple with waste stream 1 properties: (Volume, DO, CBOD, NBOD)
#   - waste2: tuple with waste stream 2 properties: (Volume, DO, CBOD, NBOD)
#   - U: velocity of river (km/d)
#   - C  = saturation oxygen concentration (mg/L)
#   - ka, kc, kn: reaeration, CBOD decay, and NBOD decay rates, respectively
 ↪  (d^{-1})
function waste_treat(eff1, eff2, inflow, waste1, waste2, U, C , ka, kc, kn)
    waste1_treated = (waste1[1], waste1[2], (1 - eff1) * waste1[3], (1 -
 ↪  eff1) * waste1[4])
    waste2_treated = (waste2[1], waste2[2], (1 - eff2) * waste2[3], (1 -
 ↪  eff2) * waste2[4])
    C, B, N = do_simulate(inflow, waste1_treated, waste2_treated, U, C , ka,
 ↪  kc, kn)
    return minimum(C)
end
```

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

Now we can evaluate this function over a range of treatment efficiencies. There are a number of ways to do this, but we'll use a trick which Julia makes easy: broadcasting using an anonymous function.

```
# evaluate treatment efficiencies between 0 and 1
effs = 0:0.01:1
treat1 = (e1 -> waste_treat(e1, 0, inflow, waste1, waste2, U, C , ka, kc,
 ↪  kn)).(effs)
treat2 = (e2 -> waste_treat(0, e2, inflow, waste1, waste2, U, C , ka, kc,
 ↪  kn)).(effs)
```

```
101-element Vector{Float64}:
 3.694312052808094
```

```
3.715106569604958
3.7359010864018223
3.7566956031986853
3.7774901199955497
3.7982846367924123
3.819079153589276
3.8398736703861394
3.8606681871830038
3.8814627039798673

4.711700098308851
4.711700098308851
4.711700098308851
4.711700098308851
4.711700098308851
4.711700098308851
4.711700098308851
4.711700098308851
4.711700098308851
```

To find the minimum treatment level which ensures compliance, we can now find the position of the first value where the minimum value is at least 4 mg/L, and look up the associated efficiency. Julia provides the `findfirst()` function which lets you find the index of the first value satisfying a Boolean condition.

```
# find indices and treatment values
idx1 = findfirst(treat1 .>= 4.0)
idx2 = findfirst(treat2 .>= 4.0)
@show effs[idx1];
@show effs[idx2];
```

```
effs[idx1] = 0.19
effs[idx2] = 0.15
```

So the minimum treatment level for waste stream 1 is 19% and the minimum treatment level for waste stream 2 is 15%.

There is no "right" answer to the question of which treatment option you would pick, so long as your solution is thoughtful and justified.

- For example, one could argue that as waste stream 1 on its own does not result in a lack of compliance (which we can see from Figure 1, as the initial "sag" has started to recover prior to waste stream 2), waste stream 2 ought to be treated.

- On the other hand, waste stream 1 has a much more negative impact on the inflow dissolved oxygen levels, and waste stream 2 might not cause a lack of compliance without that effect, which might suggest that waste stream 1 should be treated.
- Another consideration might be the relative cost of treating each waste stream, which we have no information on, or whether these waste streams are from municipal, residual, or industrial sources (in other words, non-profit vs. profit).

Problem 2 (20 points)

Loading the data:

```
# Dataset from https://zenodo.org/record/3973015
# The CSV is read into a DataFrame object, and we specify that it is comma
 ↪  delimited
forcings_all = CSV.read("data/ERF_ssp585_1750-2500.csv", DataFrame,
 ↪  delim=",")

# Separate out the individual components
# Get total aerosol forcings
forcing_aerosol_rad = forcings_all[!,"aerosol-radiation_interactions"]
forcing_aerosol_cloud = forcings_all[!,"aerosol-cloud_interactions"]
forcing_aerosol = forcing_aerosol_rad + forcing_aerosol_cloud
# Calculate non-aerosol forcings from the total.
forcing_total = forcings_all[!,"total"]
forcing_non_aerosol = forcing_total - forcing_aerosol
t = Int64.(forcings_all[!,"year"])
idx_2100 = findfirst(t .== 2100) # find the index which corresponds to the
 ↪  year 2100
```

```
351
```

The energy balance model is given by

$$cd\frac{dT}{dt} = F - \lambda T.$$

Discretizing with Forward Euler, we get

$$\begin{aligned} cd\frac{T(t+\Delta t) - T(t)}{\Delta t} &= F(t) - \lambda T(t) \\ T(t+\Delta t) &= T(t) + \frac{\Delta t}{cd}\left(F(t) - \lambda T(t)\right). \end{aligned}$$

In Julia code (and with the assumed parameter values), this becomes the following function (replacing $F(t) = F_{\text{nonaerosol}} + \alpha F_{\text{aerosol}}$):

```
# ebm: function to simulate the energy balance model
# inputs:
#   - t: vector of time values.
#   - F_nonaerosol: vector of non-aerosol radiative forcing values
#   - F_aerosol: vector of aerosol radiative forcing values
#   - c, d,  ,  , Δt: model parameters, see below.
function ebm(t, F_nonaerosol, F_aerosol, c, d,  ,  , Δt)
    T = zeros(length(t))                                                    ①
    F = F_nonaerosol +   * F_aerosol
    for s in 1:length(t)-1                                                  ②
        T[s+1] = T[s] + Δt * (F[s] -   * T[s]) / (c * d)
    end
    return T
end

c = 4.184e6 # specific heat of water, J/K/m^2
d = 86 # ocean mixing depth, m
  = 2.1 # climate feedback factor
  = 0.8 # aerosol scaling factor
Δt = 31_558_152 # annual time step, s

temps = ebm(t, forcing_non_aerosol, forcing_aerosol, c, d,  ,  , Δt)
```

① The `zeros` function initializes a vector with zero values of the given length. This is a good way to initialize storage when you aren't worried about distinguishing non-set values from "true" zeroes.

② While normally `eachindex(t)` is preferred instead of `1:length(t)`, in this case we need to iterate only until the second-to-last index of `t` as we set the next value `T[s+1]` in the loop.

```
751-element Vector{Float64}:
 0.0
 0.022731483349329427
 0.040278289494741326
 0.05240826578450054
 0.05895069636157351
 0.06063584224928685
 0.058278631700672184
 0.036311110702349894
```

```
 0.030604998748376 28
 0.037780681173548 78

 6.034906444978151 5
 6.033928949865114
 6.032952263810847
 6.031978595303466
 6.031006621457839 5
 6.030034177683138
 6.029061472805756
 6.028090703835366
 6.027123866872981
```

The simulated temperatures are plotted in Figure 2:

```
p = plot(; xlabel="Year", ylabel="Global Mean Temperature Anomaly (°C)",
 ↪  legend=false)                                                          ①
plot!(p, t, temps)                                                         ②
xlims!(p, (1750, 2100))                                                    ③
```

① This syntax creates an empty plot object with the right axis labels, legend settings, etc; this is not necessary but can make the code more readable.

② `plot!` is different than `plot` because it adds elements to an existing plot object (in this case, `p`) rather than creating a new one. Since `p` is the last plot object created, we could have just written `plot!(t, temps)`, but adding `p` explicitly minimizes the chances of accidentally plotting onto the wrong axis.

③ Similarly, `xlims!` changes the x limits for an existing plot object (analogously, there is also `ylims!`).

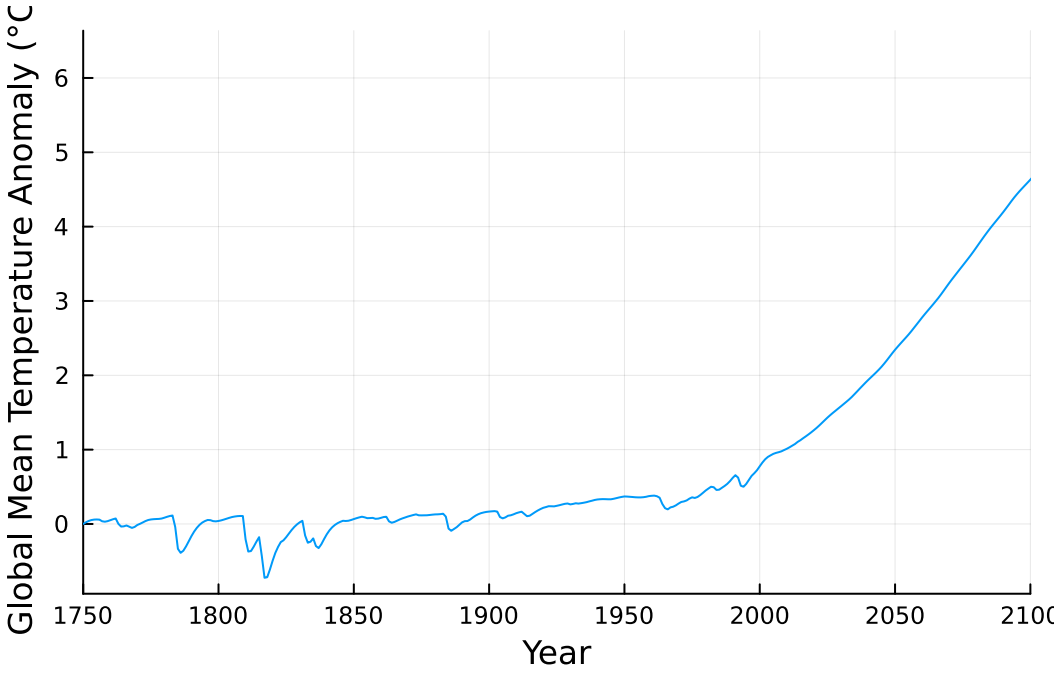

Figure 2: Global mean temperature anomalies (in °C, relative to 1750) from 1750 to 2100 simulated using the energy balance model and RCP 8.5 forcings and parameters given in the problem statement.

For the Monte Carlo analysis, we'll start with 10,000 samples of λ and see how the simulation looks.

```
n_samples = 10_000                                                    ①
 _dist = LogNormal(log(2.1), log(2) / 4)
 _samples = rand( _dist, n_samples)
h = histogram( _samples; xlabel="Climate Feedback Factor (°C/W/m²)",
 ↪  ylabel="Count", legend=false)
vline!(h, [2.1], color=:red, linestyle=:dash) # the original value for
 ↪  context
```

① Creating a variable for the number of samples which can then be used later makes it easier to change the number of samples down the road without possibly creating bugs (if *e.g.* we forget to change the value everywhere).

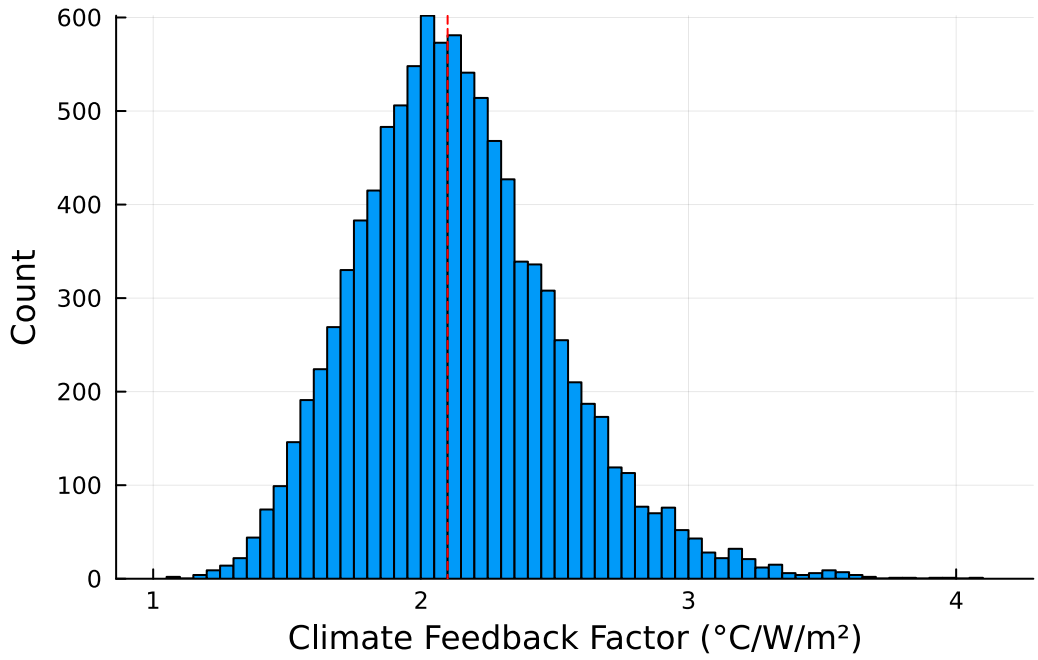

Figure 3: Samples of the climate feedback factor λ. The red line is the deterministic value we used in the original simulation above.

Now, simulate:

```
temp_samples = [ebm(t, forcing_non_aerosol, forcing_aerosol, c, d,  ,  ,
 ↪  Δt)[idx_2100] for   in  _samples]                                    ①
```

① We directly get the values of from `_samples` instead of looping over the indices (which is slightly slower and makes the line longer).

```
10000-element Vector{Float64}:
 4.250082752131921
 3.9271173030904514
 4.319573586572354
 4.214336791186512
 3.965556983548686
 4.016195966014413
 5.1292927884473105
 3.89544258735652
 4.618135058155357
 4.649638054098068
```

```
4.167520892980213
6.582517364306635
4.251714824229311
4.413022632352279
4.367754079778502
5.867576008321597
5.175423565112294
4.456277811290564
5.176899204379472
```

The expected value after this many simulations is the mean, or 4.68, and the 95% confidence interval is (4.67, 4.7) (using the formula for the standard error). The confidence interval around the expected value is pretty tight, but let's look at how the estimates of the mean and the confidence interval evolve over the course of the simulations to see what an "efficient" sample size might have been:

```
# compute running estimate of the expected value and standard deviation
# first, pre-allocate memory for better efficiency
T_est = zeros(n_samples)                                                    ①
T_se = zeros(n_samples)
for i = 1:n_samples
    T_est[i] = mean(temp_samples[1:i])                                      ②
    if i > 1                                                                ③
        T_se[i] = std(temp_samples[1:i]) / sqrt(i)
    end
end

p = plot(; xlabel="Number of Samples", ylabel="Global Temperature Anomaly
 ↪  (°C)", legend=false)
plot!(p, 1:n_samples, T_est, ribbon=1.96 * T_se)
ylims!(p, (4.5, 5.0))
```

① Here we can see the benefits of setting the variable `n_samples` above; if we had hard-coded the value and needed to change it, we would also have needed to change it in all three of these lines.

② There are more efficient ways to do this updating, for example `T_est[i] = (T_est[i-1] * (i-1) + T_est[i]) / i`, which avoids re-computing the average for the first `i-1` simulations, but in this case it's fast enough.

③ We can't compute the standard deviation for a single value, so we just start with `i == 2`.

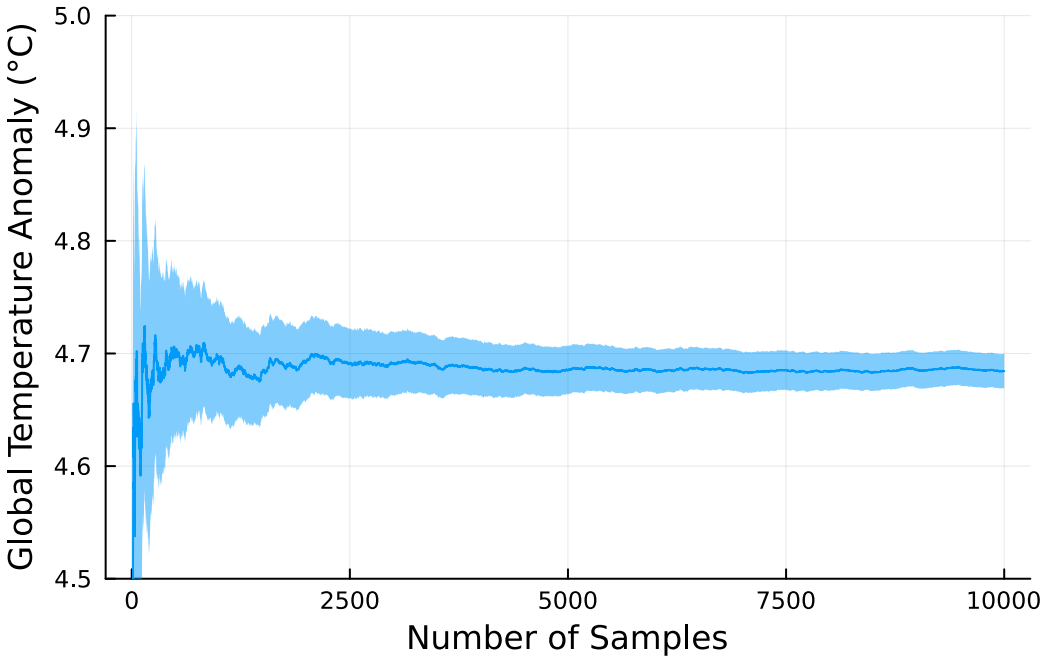

Figure 4: Monte Carlo mean and 95% confidence interval of the simulated temperature in 2100 as the number of samples increases.

If we wanted tighter confidence intervals, we could run the simulation longer (though recall the $1/\sqrt{n}$ error law, which means to get an order of magnitude reduction, we'd need approximately 100,000 samples), but it looks from Figure 4 as though the estimate had stabilized around 8,000-9,000 samples, though ultimately it depends on the desired level of precision in both the estimate and the confidence interval. In this case, regardless, we're looking at an estimated temperature anomaly of around $4.7°C$ once we exceed 5,000 samples, and it isn't clear what more precision necessarily would get us.

Problem 3 (10 points)

This problem is only required for students in BEE 5750.

A factory is planning a third wastewater discharge into the river downstream of the second plant. This discharge would consist of 5 m^3/day of wastewater with a dissolved oxygen content of 4.5 mg/L and CBOD and NBOD levels of 50 and 45 mg/L, respectively.

In this problem:

- Assume that the treatment plan for waste stream 2 that you identified in Problem 1 is still in place for the existing discharges. If the third discharge will not be treated, under

the original inflow conditions (7.5 mg/L DO), how far downstream from the second discharge does this third discharge need to be placed to keep the river concentration from dropping below 4 mg/L?

Solution

Based on how we solved Problem 1, we need to modify our solution to include a third discharge with an unknown distance from the second discharge.

> **Tip**
>
> We could have solved Problem 1 in a more modular way, which included multiple discharges (using a vector of tuples instead of explicitly specifying `waste1` and `waste2`, which would have made this solution simpler).

```
# do_simulate: function to simulate dissolved oxygen concentrations over the
 ↪  entire river
# inputs:
#   - d: distance between waste stream 2 and waste stream 3
#   - inflow: tuple with inflow properties: (Volume, DO, CBOD, NBOD)
#   - waste1: tuple with waste stream 1 properties: (Volume, DO, CBOD, NBOD)
#   - waste2: tuple with waste stream 2 properties: (Volume, DO, CBOD, NBOD)
#   - waste3: tuple with waste stream 3 properties: (Volume, DO, CBOD, NBOD)
#   - C , B , N : initial conditions for DO, CBOD, and NBOD, respectively
 ↪  (mg/L)
#   - U: velocity of river (km/d)
#   - C  = saturation oxygen concentration (mg/L)
#   - ka, kc, kn: reaeration, CBOD decay, and NBOD decay rates, respectively
 ↪  (d^{-1})
function do_simulate2(d, inflow, waste1, waste2, waste3, U, C , ka, kc, kn)
    # set up ranges for each box/segment
    x  = 0:1:15
    x  = (15:1:15+d) .- 15
    x  = (15+d:1:50) .- (15 + d)


    V_inflow, C_inflow, B_inflow, N_inflow = inflow
    V_ws1, C_ws1, B_ws1, N_ws1 = waste1
    V_ws2, C_ws2, B_ws2, N_ws2 = waste2
    V_ws3, C_ws3, B_ws3, N_ws3 = waste2

    # initialize storage for final C, B, and N
```

```
    # need to store d=1 so the total length should be d+1
    C = zeros(51)
    B = zeros(51)
    N = zeros(51)

    # compute initial conditions for first segment
    C  = mix_concentration(V_inflow, C_inflow, V_ws1, C_ws1)
    B  = mix_concentration(V_inflow, B_inflow, V_ws1, B_ws1)
    N  = mix_concentration(V_inflow, N_inflow, V_ws1, N_ws1)

    # conduct first segment simulation
    (C , B , N ) = dissolved_oxygen(x , C , B , N , U, C , ka, kc, kn)
    C[1:15] = C [1:end-1]
    B[1:15] = B [1:end-1]
    N[1:15] = N [1:end-1]

    # compute initial conditions for second segment
    C  = mix_concentration(V_inflow + V_ws1, C [end], V_ws2, C_ws2)
    B  = mix_concentration(V_inflow + V_ws1, B [end], V_ws2, B_ws2)
    N  = mix_concentration(V_inflow + V_ws1, N [end], V_ws2, N_ws2)

    # conduct second segment simulation
    (C , B , N ) = dissolved_oxygen(x , C , B , N , U, C , ka, kc, kn)
    C[16:16+d] = C
    B[16:16+d] = B
    N[16:16+d] = N

    # compute initial conditions for third segment
    C  = mix_concentration(V_inflow + V_ws1 + V_ws2, C [end], V_ws3, C_ws3)
    B  = mix_concentration(V_inflow + V_ws1 + V_ws2, B [end], V_ws3, B_ws3)
    N  = mix_concentration(V_inflow + V_ws1 + V_ws2, N [end], V_ws3, N_ws3)

    # conduct second segment simulation
    (C , B , N ) = dissolved_oxygen(x , C , B , N , U, C , ka, kc, kn)
    C[16+d:end] = C
    B[16+d:end] = B
    N[16+d:end] = N


    return (C, B, N)
end
```

```
# set up treatment level for waste stream 2 based on solution for Problem 1
B_ws2 = (1 - 0.15) * 45.0
N_ws2 = (1 - 0.15) * 35.0
waste2 = (V_ws2, C_ws2, B_ws2, N_ws2)

# set parameters for waste stream 3
C_ws3 = 4.5
B_ws3 = 50.0
N_ws3 = 45.0
V_ws3 = 5.0 * 1_000 # convert from m^3/day to L/day
waste3 = (V_ws3, C_ws3, B_ws3, N_ws3)
```

```
(5000.0, 4.5, 50.0, 45.0)
```

Now we can evaluate `do_simulate2` over the different values of `d` to find the minimum value of `C`.

```
# three_discharge_minC: function to find the minimum DO concentration based
 ↪  on the three discharge model
# inputs:
#   - d: distance between waste stream 2 and waste stream 3
#   - inflow: tuple with inflow properties: (Volume, DO, CBOD, NBOD)
#   - waste1: tuple with waste stream 1 properties: (Volume, DO, CBOD, NBOD)
#   - waste2: tuple with waste stream 2 properties: (Volume, DO, CBOD, NBOD)
#   - waste3: tuple with waste stream 3 properties: (Volume, DO, CBOD, NBOD)
#   - C , B , N : initial conditions for DO, CBOD, and NBOD, respectively
 ↪  (mg/L)
#   - U: velocity of river (km/d)
#   - C  = saturation oxygen concentration (mg/L)
#   - ka, kc, kn: reaeration, CBOD decay, and NBOD decay rates, respectively
 ↪  (d^{-1})
function three_discharge_minC(d, inflow, waste1, waste2, waste3, U, C , ka,
 ↪  kc, kn)
    C, B, N = do_simulate2(d, inflow, waste1, waste2, waste3, U, C , ka, kc,
 ↪  kn)
    return minimum(C)
end

# evaluate over all distances from 1 to 35 m downstream using broadcasting
 ↪  and anonymous functions
d = 1:1:35
```

```
minC_d = (dist -> three_discharge_minC(dist, inflow, waste1, waste2, waste3,
 ↪  U, C , ka, kc, kn)).(d)
idx = findfirst(minC_d .>= 4.0)
min_dist = d[idx]
@show min_dist;
```

```
min_dist = 14
```

So, without treatment, the third discharge should be placed 14 m downstream from the second discharge.

References

List any external references consulted, including classmates.