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: 30 Points)

Problem 1 (30 points)

The first step is to write function(s) to implement the Streeter-Phelps 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_analytic(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_analytic (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 subsequent problems, 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_analytic(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_analytic(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_analytic(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_analytic(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, color=:blue, label="Analytic Solution")
hline!(p, [4], color=:black, linestyle=:dot, 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.

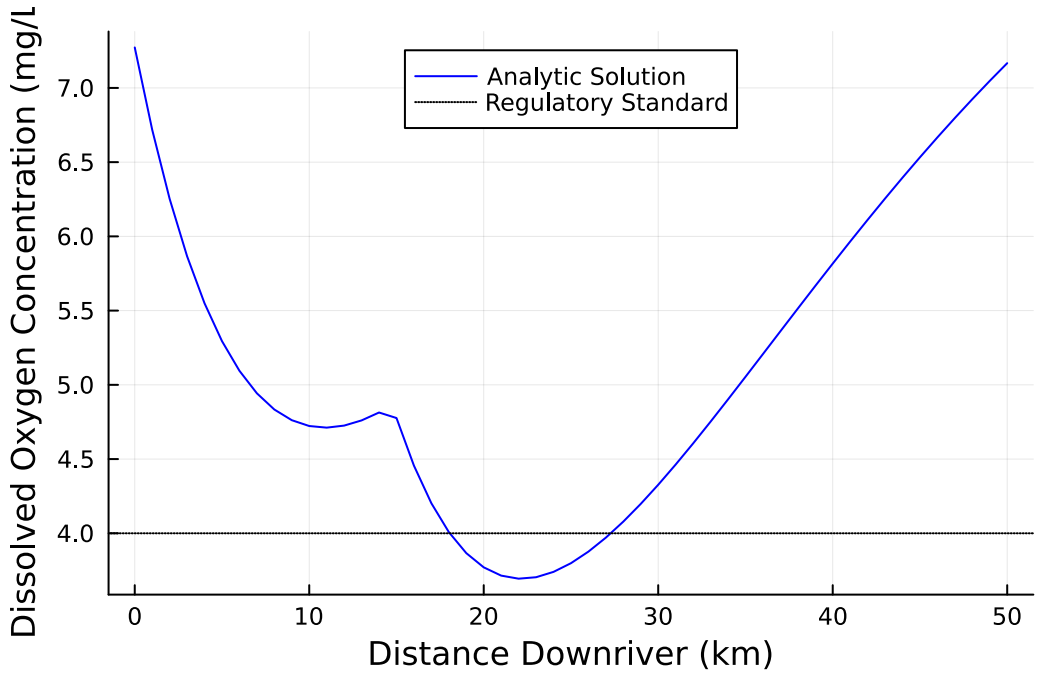

Figure 1: Simulated dissolved oxygen concentration for Problem 1 without treatment.

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.

Problem 1.2

Instead of the analytic Streeter-Phelps equation, now we want to use a numerically-integrated solution to the DO equation ("neglecting" P, R, and S_B by combining with the other constant terms):

$$\begin{aligned} C(x+\Delta x) &= C(x) + \frac{\Delta x}{U}(k_a(C_s - C(x)) + P - R - S_B \\ &\quad - k_c B(x) - k_n N(x)) \\ B(x+\Delta x) &= B(x)\exp\left(\frac{-k_c \Delta x}{U}\right) \\ N(x+\Delta x) &= N(x)\exp\left(\frac{-k_n \Delta x}{U}\right) \end{aligned}$$

One implementation of this solution:

```
# do_numerical: function to numerically simulate dissolved oxygen
 ↪  concentrations over a single box
# inputs:
#   - L: length of the river segment corresponding to the box
#   - Δx: spatial resolution (length step) in km
#   - C0: initial DO concentration of the box (after any mixing has occurred)
 ↪  in mg/L
#.  - B0: initial CBOD concentration of the box (after any mixing has
 ↪  occurred) in mg/L
#.  - N0: initial NBOD concentration of the box (after any mixing has
 ↪  occurred) in mg/L
#   - ka, kc, kn: reaeration, CBOD decay, and NBOD decay rates, respectively
 ↪  (d^{-1})
#   - C  = saturation oxygen concentration (mg/L)
#   - U: velocity of river (km/d)
function do_numerical(L, Δx, C0, B0, N0, ka, kc, kn, Cs, U)
    n = Int(L / Δx) # number of length steps
    C = zeros(n + 1)
    B = zeros(n + 1)
    N = zeros(n + 1)
    C[1] = C0
    B[1] = B0
    N[1] = N0
    for i = 1:n
        B[i+1] = B[i] * exp(-kc * Δx / U)
        N[i+1] = N[i] * exp(-kn * Δx / U)
```

```
        C[i+1] = C[i] + (Δx / U) * (ka * (Cs - C[i]) - kc * B[i] - kn * N[i])
    end
    return (C, B, N)
end

# do_simulate_numerical: function to numerically simulate dissolved oxygen
 ↪  concentrations over the entire river
# inputs:
#   - L: tuple with lengths of each box (in km from the first waste
 ↪  discharge)
#.  - Δx: spatial resolution of simulation (in km)
#   - 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_numerical_simulate(L, Δx, inflow, waste1, waste2, U, C , ka, kc,
 ↪  kn)
    # set up ranges for each box/segment
    L1, L2 = L
    x  = 0:Δx:L1                                                          ①
    x  = 0:Δx:L2                                                          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 n+1
    n = Int(sum(L) / Δx)
    C = zeros(n + 1)
    B = zeros(n + 1)
    N = zeros(n + 1)

    # 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 ) = do_numerical(L1, 0.5, C , B , N , ka, kc, kn, C , U)
    C[1:length(C )-1] = C [1:end-1]                                          ④
    B[1:length(B )-1] = B [1:end-1]
    N[1:length(N )-1] = 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 ) = do_numerical(L2, 0.5, C , B , N , ka, kc, kn, C , U)
    C[length(C ):end] = C
    B[length(B ):end] = B
    N[length(N ):end] = N

    return (C, B, N)
end

C1, B1, N1 = do_numerical_simulate((15, 35), 0.5, inflow, waste1, waste2, U,
 ↪  C , ka, kc, kn)
```

```
([7.2727272727272725, 6.971590909090909, 6.6951976781381255, 6.442125991305724, 6.2110271250
```

Now let's add this simulation to the plot from Problem 1.1:

```
plot!(p, 0:0.5:50, C1, color=:darkorange, label="Numerical Solution")
```

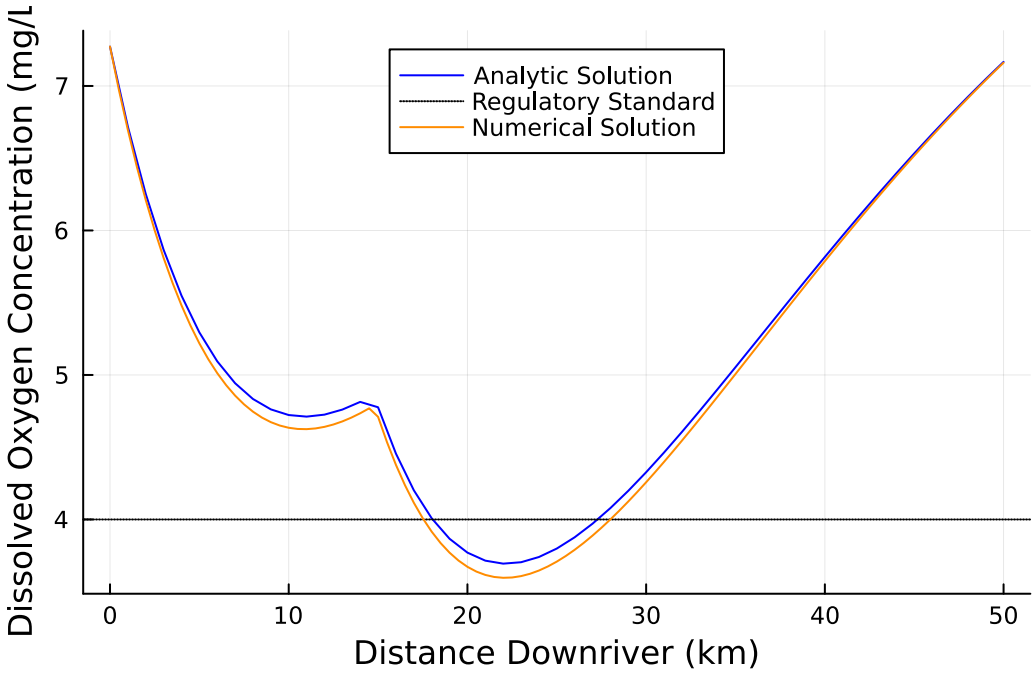

Figure 2: Numerical and analytic DO solutions for Problem 1.

The minimum value is 3.6 mg/L. What is the cause of the difference? At each length step, the values of `C`, `B`, and `N` are taken from the previous step, 0.5 km away. Prior to the DO rebound, this means that the corresponding CBOD and NBOD values, and therefore the level of deoxygenation, are higher than in the analytic solution. At the same time, the level of dissolved oxygen is greater than in the analytic solution, so there is reduced reaeration. The combined effect is that the DO reduces more quickly than in the analytic solution at each step, and the minimum is lower. Conversely, once the minimum is reached and reaeration begins to dominate, because the DO level is lower, there is a more aggressive reaeration, and the numerical solution can catch up as the CBOD and NBOD become more negligible downstream.

Problem 1.3

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_analytic(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 4.2 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 .>= 4.2)
idx2 = findfirst(treat2 .>= 4.2)
@show effs[idx1];
@show effs[idx2];
```

```
effs[idx1] = 0.32
effs[idx2] = 0.26
```

So the minimum treatment level for waste stream 1 is 32% and the minimum treatment level for waste stream 2 is 26%.

Problem 1.4

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 1.5

To conduct this Monte Carlo experiment, we need to draw samples from the given LogNormal$(2.0, 0.15)$ distribution. We'll start by drawing 5,000 samples and see if this is sufficient; your answer may be different if this was more or less than you drew.

```
nsamp = 5_000
samps = rand(LogNormal(2.0, 0.15), nsamp)
histogram(samps, xlabel="River Inflow DO Concentrations (mg/L)",
 ↪  ylabel="Count")
```

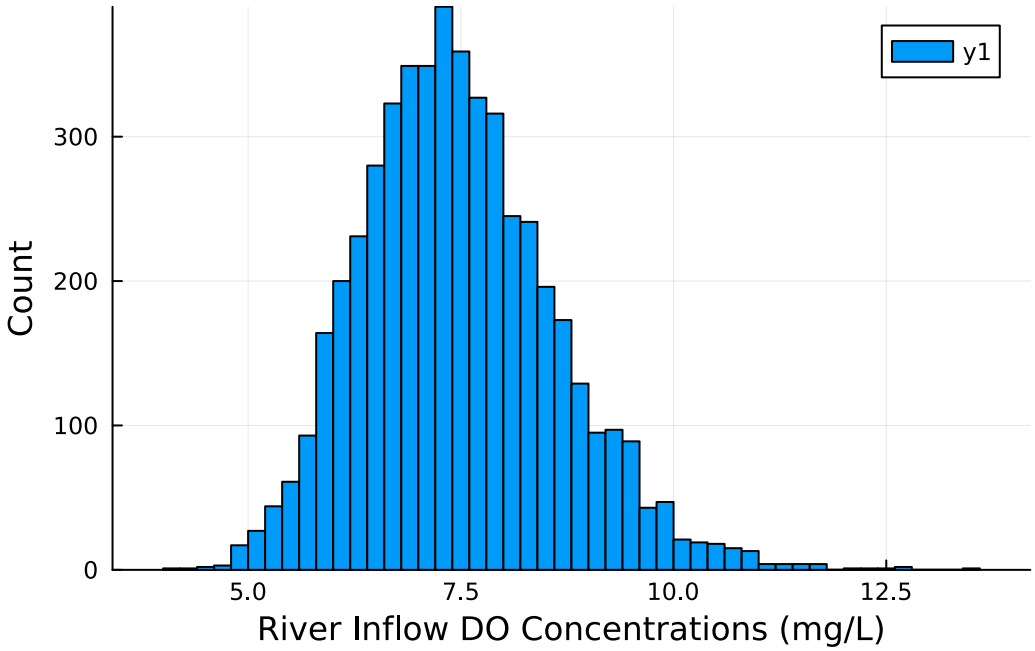

Figure 3: Samples drawn from the LogNormal(2.0, 0.15) distribution.

Now we need to evaluate our model for each of these, using the same `eff1` value that we

obtained in Problem 1.3, and we will save the statistic that we are interested in, which is whether the DO falls below the regulatory limit.

```
compliance = zeros(nsamp)
for i = 1:nsamp
    inflow = (V_inflow, samps[i], B_inflow, N_inflow) # this changes the
 ↪  inflow tuple to have the sampled DO level
    do_out = waste_treat(effs[idx1], 0, inflow, waste1, waste2, U, C , ka, kc,
 ↪  kn)
    compliance[i] = minimum(do_out) > 4.0
end
```

Now we can compute the Monte Carlo estimate as a function of the sample size and the confidence interval.

```
mc_est = zeros(nsamp)
mc_se = zeros(nsamp)
for i = 1:nsamp
    if i == 1
        mc_est[i] = compliance[i]
        mc_se[i] = 0
    else
        mc_est[i] = (mc_est[i-1] * (i-1) + compliance[i]) / i
        mc_se[i] = std(compliance[1:i]) / sqrt(i)
    end
end

plot(mc_est, ribbon=1.96 * mc_se, ylabel="Compliance Probability",
 ↪  xlabel="Sample", label="Monte Carlo Estimate")
```

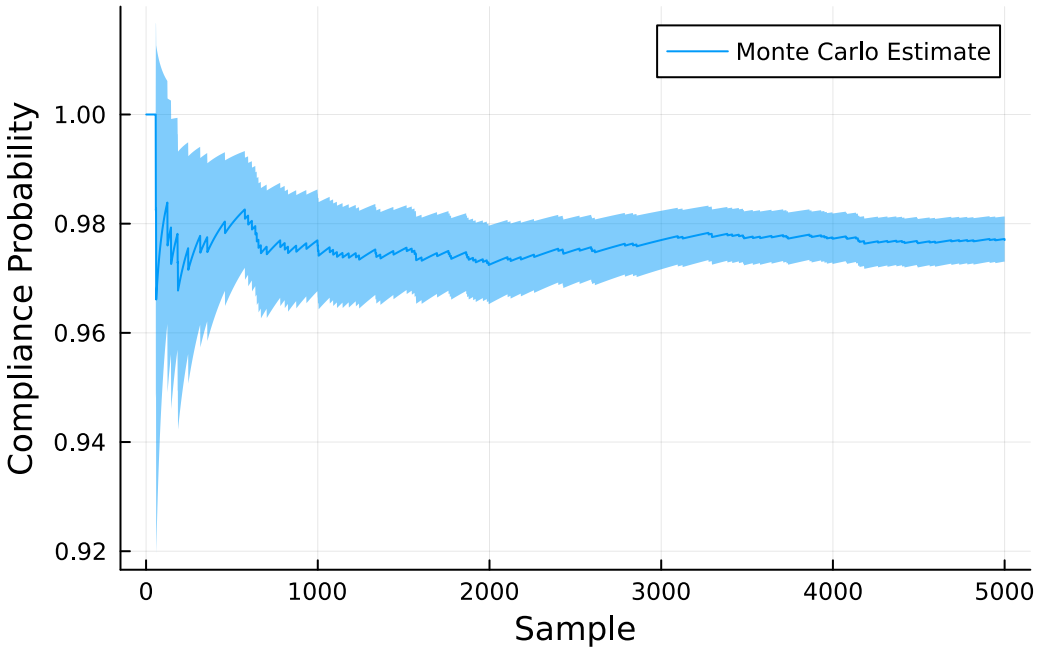

Figure 4: Results from the Monte Carlo simulation in Problem 1.

The Monte Carlo estimate of the compliance probability is about 98%, with a 95% confidence interval of (97%, 98%) (obtained by getting a half-width of the confidence interval of $1.96 \times \text{SE}$), where SE is the Monte Carlo standard error.

We can see that this estimate has stabilized after about 2,000 samples, so our use of 5,000 can be justified so long as the level of uncertainty is appropriate for a given decision. For example, if any failure under these conditions is intolerable, we would not even need this many simulations, as before 1,000 simulations it becomes clear that this failure will occur. If there is some other standard, then we would have to compare that to the confidence interval to see if our decision would be ihfluenced by further refinement of the estimate.

References

List any external references consulted, including classmates.