BEE 4750 Lab 4: Simulation-Optimization

Name: Anthony Nicolaides

ID: ajn68

> **Due Date**
>
> Friday, 11/17/23, 9:00pm

Setup

The following code should go at the top of most Julia scripts; it will load the local package environment and install any needed packages. You will see this often and shouldn't need to touch it.

In [ ]:
```
import Pkg
Pkg.activate(".")
Pkg.instantiate()
```

```
 Activating project at `~/Documents/Julia/BEE4750/labs/lab04-anthonynic28`
```

In [ ]:
```
using Random # for random seeds
using Distributions # statistical distribution interface
using Roots # find zeros of functions
using Metaheuristics # search algorithms
using Plots # plotting
```

Overview

In this lab, you will experiment with simulation-optimization with the shallow lake problem. The goal of this experimentation is to get an understanding of how to work with simulation-optimization methods and the impact of some choices involved in using these methods.

Free free to delete some of the illustrative cells and code blocks in your notebook as you go through and solve the lab problems...this might help reduce some potential confusion while grading about what your answer is.

Introduction

Due to ongoing economic activity, a town emits phosphorous into a shallow lake (with a concentration of a_t), which also receives non-point source runoff (concentration y_t) from the surrounding area. The concentration of the lake at time $t + 1$ is given by

$$X_{t+1} = X_t + a_t + y_t + \frac{X_t^q}{1 + X_t^q} - bX_t,$$

where:

Parameter	Value
a_t	point-source phosphorous concentration from the town
y_t	non-point-source phosphorous concentration
q	rate at which phosphorous is recycled from sediment
b	rate at which phosphorous leaves the lake

and $X_0 = 0$, $y_t \sim LogNormal(\log(0.03), 0.25)$, $q = 2.5$, and $b = 0.4$.

The goal of the optimization is to maximize the town's average phosphorous concentration releases (as a proxy of economic activity): $\max \sum_{t=1}^T a_t/T$ over a 100-year period. We have decided (initially) that an acceptable solution is one which will result in the lake eutrophying no more than 10% of the time.

The non-point source samples can be sampled using the following code block:

In [ ]:
```
Random.seed!(1)

T = 100 # length of simualtion
n_samples = 1_000 # replace with number of samples if you experiment

# add q and b
q = 2.5
b = 0.4

P_distribution = LogNormal(log(0.03), 0.25)
y = rand(P_distribution, (T, n_samples)) # sample a T x n_samples matrix
```

```
100×1000 Matrix{Float64}:
 0.0294753  0.0459864  0.023513   …  0.0259183  0.0260934  0.0284652
 0.034263   0.0222782  0.0459188     0.0288482  0.0480454  0.0531018
 0.0245199  0.0296271  0.0445619     0.0246404  0.0250734  0.0304308
 0.055448   0.0312     0.0228208     0.0298609  0.0428105  0.0256198
 0.0401417  0.024978   0.0458244     0.0228935  0.0286062  0.0238694
 0.0320754  0.021759   0.0471452  …  0.0472771  0.0187508  0.0306753
 0.0464641  0.0416385  0.0246833     0.0382252  0.0288505  0.0226561
 0.0244027  0.0432707  0.0341214     0.0238988  0.0427204  0.0316143
 0.0231156  0.0279197  0.0217747     0.0231772  0.0335662  0.0324465
 0.0276303  0.0305858  0.0440326     0.0289394  0.0312328  0.0173388
 ⋮                                ⋱
 0.025665   0.0341366  0.0274747     0.0283546  0.0458031  0.0277959
 0.0405629  0.0421121  0.0252557     0.0450377  0.0284411  0.0206434
 0.0228445  0.0223746  0.0210942     0.0442834  0.0337672  0.0287835
 0.0252604  0.0462868  0.0358435     0.0146043  0.023085   0.0240258
 0.0237704  0.0378816  0.0291353  …  0.0442565  0.0326019  0.021258
 0.0335883  0.0218181  0.0331417     0.0510932  0.0366858  0.0296993
 0.0312679  0.0187135  0.0534007     0.0451143  0.026161   0.0183542
 0.0382682  0.0310429  0.0296539     0.0471901  0.0283451  0.0219015
 0.0256315  0.0232021  0.0239607     0.0350929  0.0293862  0.0348934
```

We write the lake model as a function:

```
In [ ]:
# lake function model
# inputs:
#   a: vector of point-source releases (to be optimized)
#   y: randomly-sampled non-point sources
#   q: lake phosphorous recycling rate
#   b: phosphorous outflow rate
#
# returns:
#   series of lake phosphorous concentrations
function lake(a, y, q, b, T)
    X = zeros(T + 1, size(y, 2))
    # calculate states
    for t = 1:T
        X[t+1, :] = X[t, :] .+ a[t] .+ y[t, :] .+ (X[t, :] .^ q ./ (1 .+ X[t
    end
    return X
end
```

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

However, this isn't sufficient on its own! `Metaheuristics.jl` (and most simulation-optimization packages) require the use of a *wrapper* function, which accepts as inputs both parameters to be optimized (in this case, point-source releases `a`) and parameters which will be fixed (the others; see below for how to incorporate these into the syntax) and returns the required information for the optimization procedure.

`Metaheuristics.jl` wants its optimizing wrapper function to return (in order):

- the objective(s) (in this case, the mean of `a`, $\sum_t a_t/T$),

- a vector of the degrees to which the solution fails to achieve any inequality constraints (positive values indicate a larger failure, values below zero are considered acceptable)
- a vector of the degrees to which the solution fails to achieve any equality constraints (only values of zero indicate success), which in this case is not relevant, so we just return `[0.0]`.

In [ ]:
```
# function producing optimization outputs
# inputs:
#   a: vector of point-source releases (to be optimized)
#   y: randomly-sampled non-point sources
#   q: lake phosphorous recycling rate
#   b: phosphorous outflow rate
#
# returns:
#   - objective: mean value of point-source releases
#   - inequality constraint failure vector
#   - equality constraint failure vector (in this case, always [0.0])
function lake_opt(a, y, q, b, T, Xcrit)
    X = lake(a, y, q, b, T)
    # calculate exceedance of critical value
    Pexceed = sum(X[T+1, :] .> Xcrit) / size(X, 2)
    failconst = [Pexceed - 0.1] # replace 0.1 if you experiment with the fai
    return mean(a), failconst, [0.0]
end
```

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

To optimize using DE (differential evolution), use the following syntax:

```
results = optimize(f, bounds,
DE(options=Options(f_calls_limit=max_evals)))
```
where `bounds` is a `Matrix` of lower bounds (first row) and upper bounds (last row), and `max_evals` is an integer for the maximum number of evaluations.

- For example, to set bounds for all decision variables between 0 and 0.5, you can use

```
bounds = [zeros(T) 0.5ones(T)]'
```

- Increasing `max_evals` can help you find a better solution, but at a larger computational expense.
- You can use an anonymous function to fix values for non-optimized parameters, *e.g.*

```
y = ...
q = ...
b = ...
T = ...
Xcrit = ...
results = optimize(a -> lake_opt(a, y, q, b, t, Xcrit), bounds,
DE(options=Options(f_calls_limit=max_evals)))
```
Then to get the approximated minimum value:

```
fx = minimum(result)
```
and the approximated minimizing value:

```
x = minimizer(result)
```
The last piece is to get the critical value (to identify failures), which we can do using `Roots.jl`, which finds zeros of functions:

In [ ]:
```
# define a function whose zeros are the critical values
P_flux(x) = (x^q / (1 + x^q)) - b * x
# use Roots.find_zero() to find the non-eutrophication and non-zero critical
Xcrit = find_zero(P_flux, (0.1, 1.5))
```

```
0.6678778690448219
```

Problems

Problem 1 (2 points)

Using the default setup above, find the approximate optimum value. What is the value of the objective function, and how many failures (you can evaluate the `lake` function using your solution to find how many end-values are above the critical value).

In [ ]:
```
Random.seed!(1) # set seed for consistent output

# default values
bounds = [zeros(T) 0.1ones(T)]'
max_evals = 5000
results = optimize(a -> lake_opt(a, y, q, b, T, Xcrit), bounds,
    DE(options=Options(f_calls_limit=max_evals)))
```

```
Optimization Result
===================
 Iteration:       5
 Minimum:         0.0369952
 Minimizer:      [0.0660058, 0.04691, 0.00351889, ..., 0.0603576]
 Function calls:  5000
 Feasibles:      3 / 1000 in final population
 Total time:     17.7418 s
 Stop reason:    Maximum objective function calls exceeded.
```

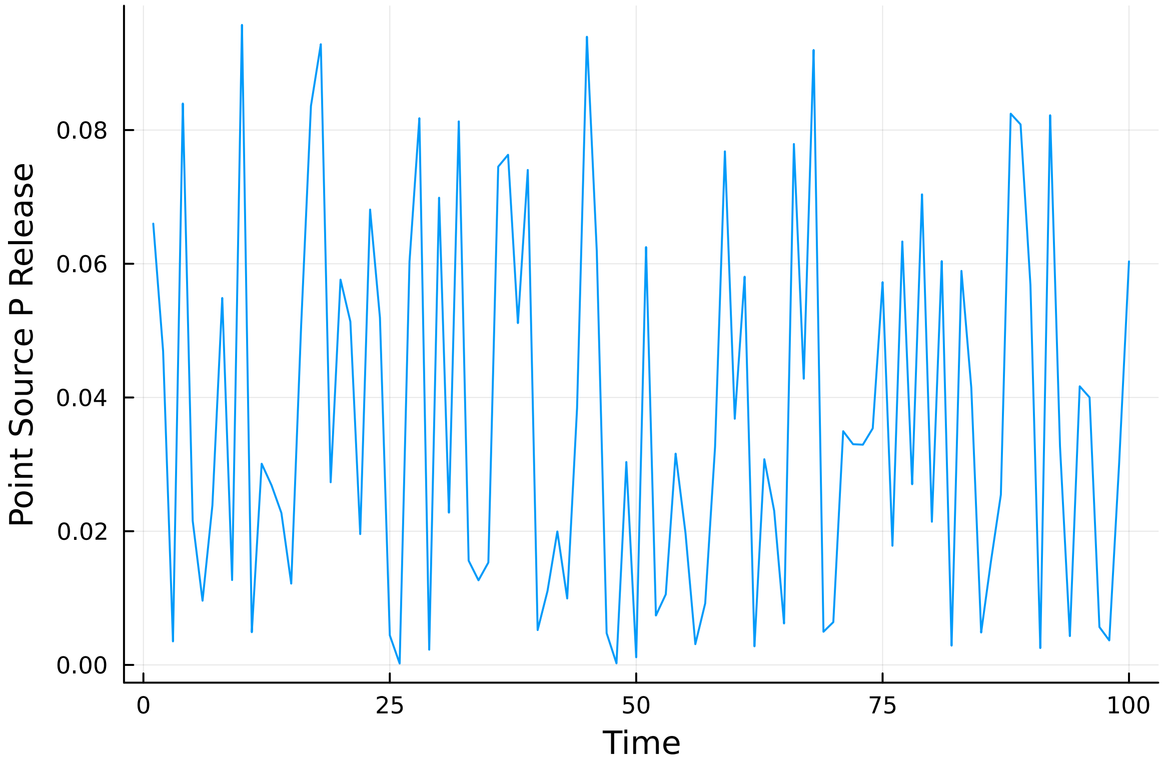
In [ ]:
```
fx = minimum(results)
x = minimizer(results)
println("approximate optimum value is ", round(fx, digits = 6))
```

```
approximate optimum value is 0.036995
```

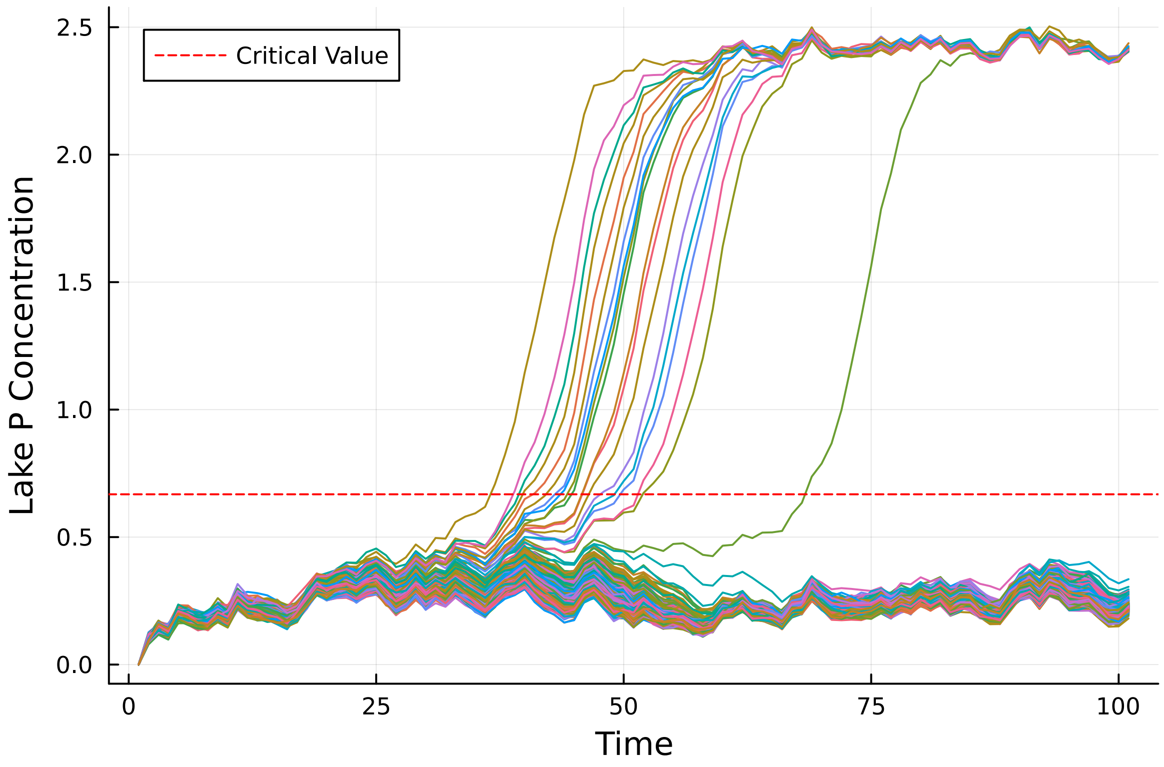
In [ ]:
```
lake_concentration = lake(x, y, q, b, T)
num_failures = count(i -> i > Xcrit, lake_concentration[end, :])
println(num_failures, " end-values are above criitical value (failure)")
println(round(100 * (num_failures / n_samples), digits=3),
    "% of the samples are failures")
```

```
19 end-values are above criitical value (failure)
1.9% of the samples are failures
```

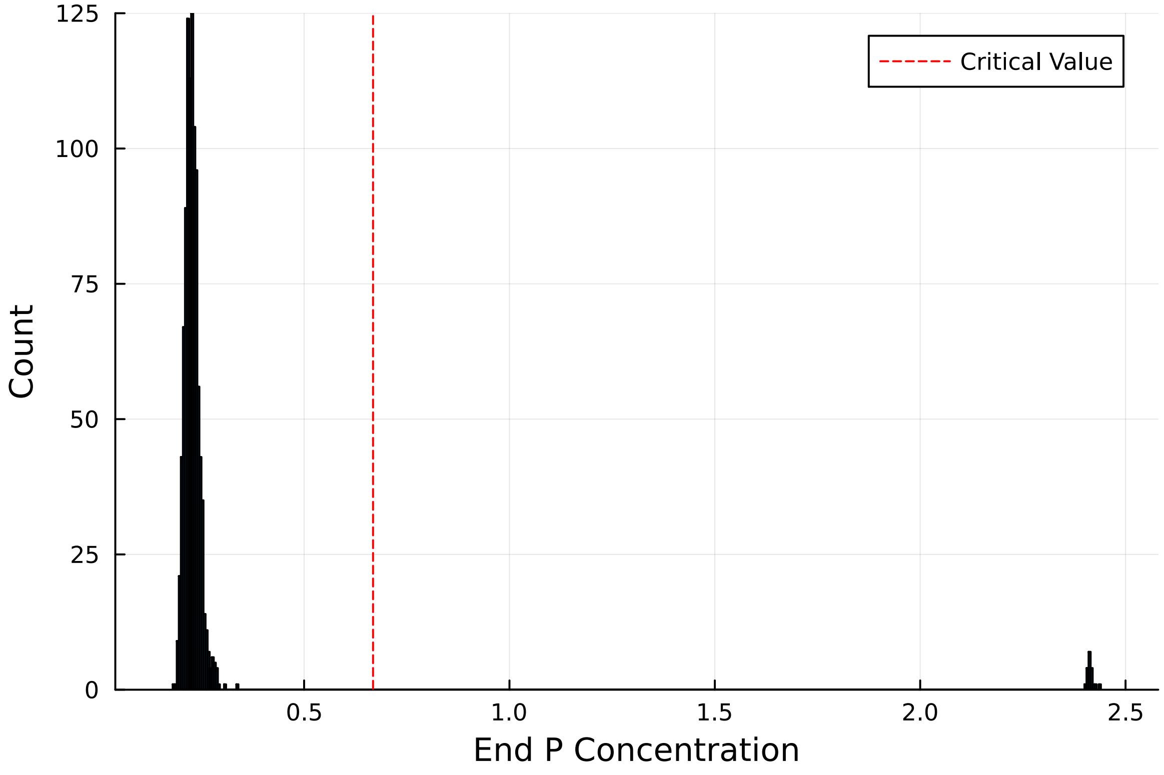
In [ ]:
```
plot(x, xlabel="Time", ylabel="Point Source P Release", legend=false)
```



In [ ]:
```
plot(lake_concentration,
    xlabel="Time", ylabel="Lake P Concentration", label="")
hline!((1:T, Xcrit),
    line=:red, linestyle=:dash, label="Critical Value", legend=true)
```



```
In [ ]: histogram(lake_concentration[end, :],
            xlabel="End P Concentration", ylabel="Count", label="")
        vline!((1:T, Xcrit),
            line=:red, linestyle=:dash, label="Critical Value", legend=true)
```



Problem 2 (5 points)

Feel free to experiment with some of the degrees of freedom in finding the optimum value. For example:

- What failure probability are you using to characterize acceptable solutions?
- How many Monte Carlo samples are you using?
- What bounds are you searching over for the releases?
- How many function evaluations are you using for the search?
- What is the impact of different `Metaheuristics.jl` algorithms?

Note that you might want to modify some of these together: for example, lower acceptable failure probabilities often require more function evaluations to find acceptable values, and more Monte Carlo samples increase computational expense, so fewer function evaluations may be completed in the same time.

Provide a description of what you've modified and why. What was the new solution that you found? Does it satisfy the constraints?

What I modified & Why

I first changed the lower bound from 0 to 0.01. I wanted to experiment and see how would the model react if there was always a non-zero point source of P release.

Running the simulation with this as the only modification gives a 99.9% of the samples as failures due to the restriction of the tighter bounds.

To compensate for the tighter bounds, I increased the amount of Monte Carlo samples from 1000 to 2000.

I decided to also increase the max function evaluations from 5000 to 10000 in order to further compensate for the tighter bounds.

With these modifications, the simulation now took 60 seconds instead of 20 seconds, and decreases the amount of samples are failures.

Results

I found that the model found 1/1000 feasibles as oppose to the default values 3/1000 feasibles.

Also, slightly more of the simulated lake concentrations reach the critical value (eutrophication). Default values found about 1.9% of the samples were failures, but my modified model gives 2.0% of the samples being failures. I was able to get roughly the same percetnage of failures with the samples by doubling both the Monte Carlo sample size and max function evaluations. If the sample size and max evaluations stayed the same with the tighter bounds, then the percentage of failures drastically increases

(99.9%), so the increasing the samples and evaluations is justified to find more solutions for the tighter bounds.

The optimal value is slightly higher compared to the default values (0.040453 vs 0.036995) which isn't surprising since the bounds are tighter and the lower bound was increased. Increasing the lower bound cut off a lot of the optimal values that took advantage of the lower bound being zero.

In [ ]:
```
# Only increase lower bound

Random.seed!(1) # set seed for consistent output
T = 100
n_samples = 1000 # increase sample size due to tighter bounds
y = rand(P_distribution, (T, n_samples)) # sample a T x n_samples matrix

bounds = [0.01ones(T) 0.1ones(T)]' # increase lower bound
max_evals = 5000 # increase evaluations due to tighter bounds
results = optimize(a -> lake_opt(a, y, q, b, T, Xcrit), bounds,
    DE(options=Options(f_calls_limit=max_evals)))

x = minimizer(results)

lake_concentration = lake(x, y, q, b, T)
num_failures = count(i -> i > Xcrit, lake_concentration[end, :])
println(num_failures, " end-values are above criitical value (failure)")
println(round(100 * (num_failures / n_samples), digits=3),
    "% of the samples are failures")
```

```
999 end-values are above criitical value (failure)
99.9% of the samples are failures
```

In [ ]:
```
# Increase lower bound, samples, and evaluations

Random.seed!(1) # set seed for consistent output
T = 100
n_samples = 2000 # increase sample size due to tighter bounds
y = rand(P_distribution, (T, n_samples)) # sample a T x n_samples matrix

bounds = [0.01ones(T) 0.1ones(T)]' # increase lower bound
max_evals = 10000 # increase evaluations due to tighter bounds
results = optimize(a -> lake_opt(a, y, q, b, T, Xcrit), bounds,
    DE(options=Options(f_calls_limit=max_evals)))
```

```
Optimization Result
===================
  Iteration:       10
  Minimum:         0.0404532
  Minimizer:       [0.0513889, 0.0694465, 0.0215873, ..., 0.0595595]
  Function calls:  10000
  Feasibles:       1 / 1000 in final population
  Total time:      70.2708 s
  Stop reason:     Maximum objective function calls exceeded.
```

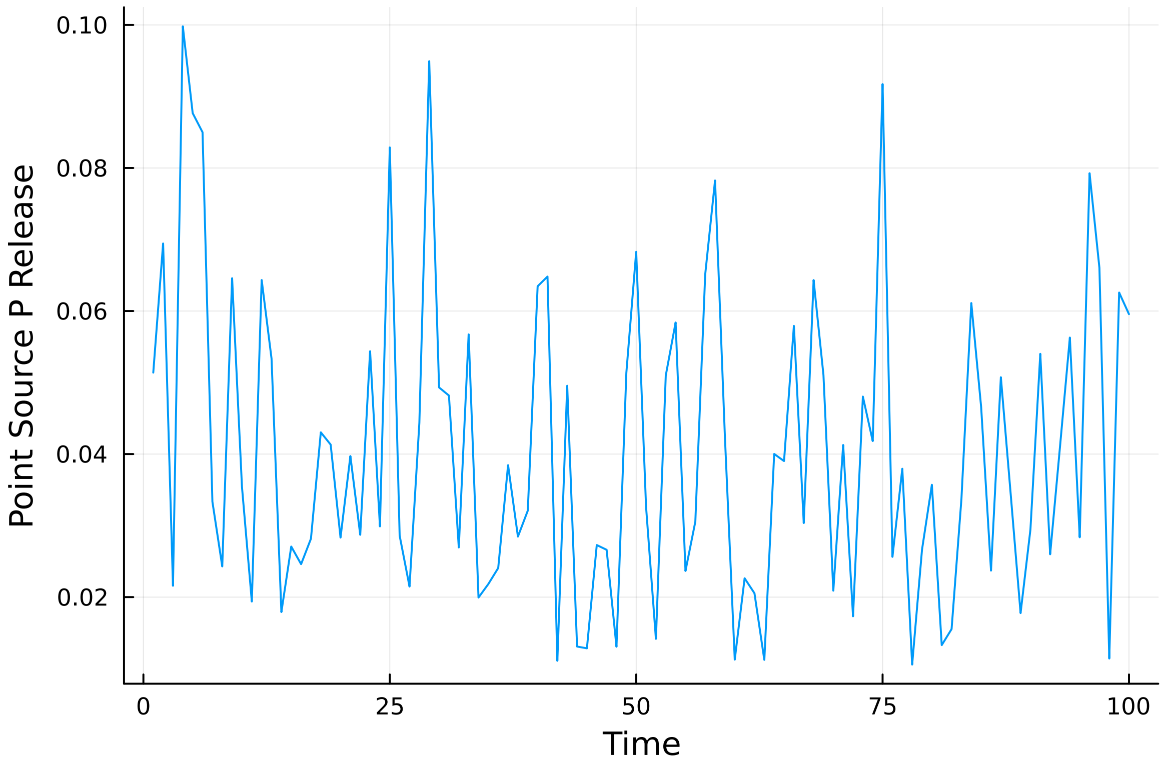
In [ ]:
```julia
# Increase lower bound, samples, and evaluations

fx = minimum(results)
x = minimizer(results)
println("approximate optimum value is ", round(fx, digits = 6))
```

```
approximate optimum value is 0.040453
```

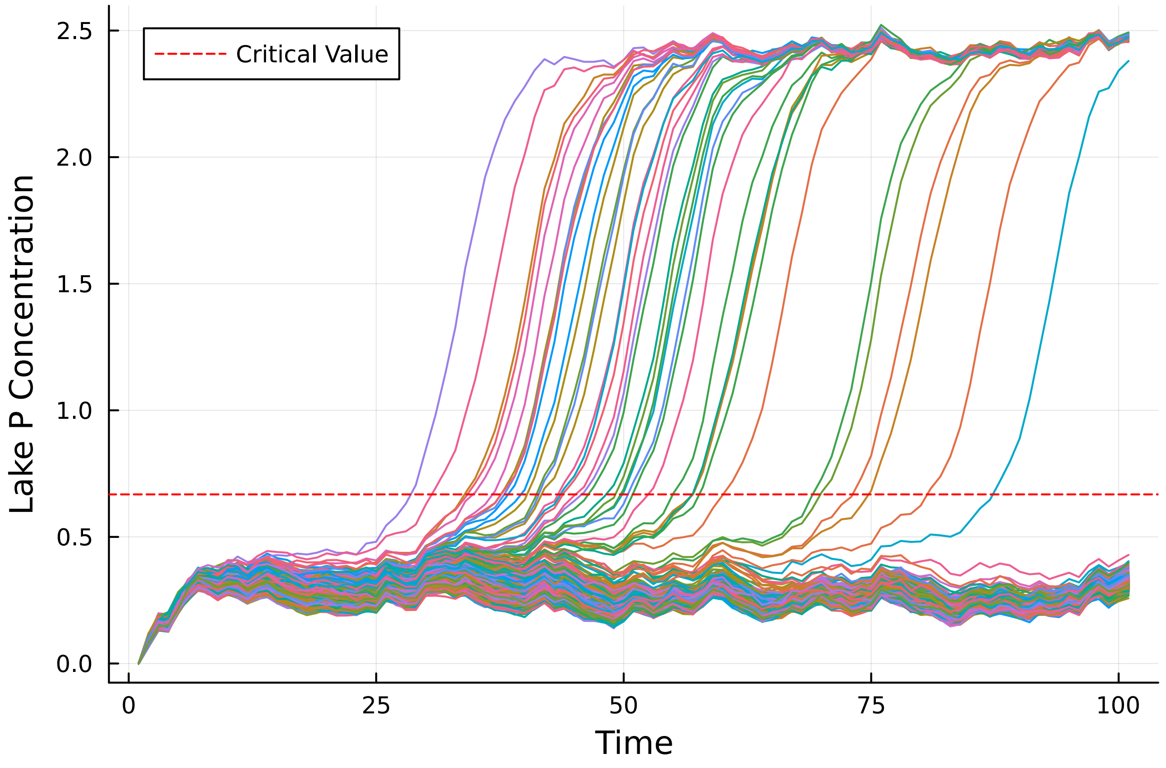
In [ ]:
```julia
# Increase lower bound, samples, and evaluations

lake_concentration = lake(x, y, q, b, T)
num_failures = count(i -> i > Xcrit, lake_concentration[end, :])
println(num_failures, " end-values are above criitical value (failure)")
println(round(100 * (num_failures / n_samples), digits=3),
    "% of the samples are failures")
```

```
40 end-values are above criitical value (failure)
2.0% of the samples are failures
```

In [ ]:
```julia
plot(x, xlabel="Time", ylabel="Point Source P Release", legend=false)
```

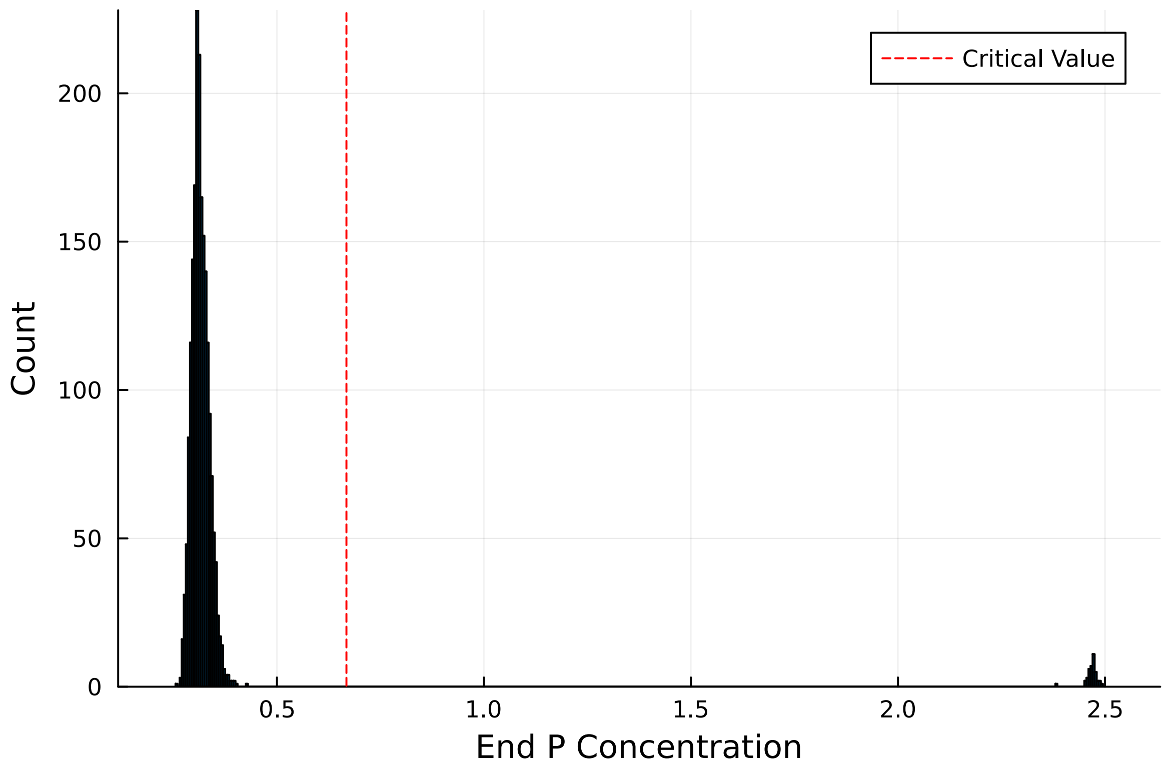


```
In [ ]: plot(lake_concentration,
            xlabel="Time", ylabel="Lake P Concentration", label="")
        hline!((1:T, Xcrit),
            line=:red, linestyle=:dash, label="Critical Value", legend=true)
```



```
In [ ]: histogram(lake_concentration[end, :],
            xlabel="End P Concentration", ylabel="Count", label="")
```

```
vline!((1:T, Xcrit),
    line=:red, linestyle=:dash, label="Critical Value", legend=true)
```



Problem 3 (3 points)

What did you learn about the use of these methods? Compare with your experience with linear programming from earlier in the semester.

I learned a lot of useful techniques using the methods demonstrated in this lab. I learned that every mdoel has trade offs between computation expense and finding the optimal value. I learned about Monte Carlo error, utilizing a large number of samples to help ensure the difference between function values and noise. I also, more generally, demonstrated the importance of sample size and max evaluations in simulation optimization.

Specifically with Problem 2, I demonstrated that sample size and evaluations plays a big role in finding solutions that satisfy constraints. If I allow the model to search through more samples and evaluate the function more it can actually find a larger percentage of the samples being successful (2.0%). In fact, by doing this I get a similar percetnage of failures as running the simluation with the default values (1.9%). If i kept the tighter bounds and the same other default values, the percentage of samples being failures would be about 99.9%, this highlights how heuristics can influence simulation optimization.

This related back to linear programming in terms of contraints. By decreasing the size of the bounds, I make the potential solutions smaller, given the default search parameters.

However by allowing more samples and evaluations, I gave the simulation more time to find the optimal solutions despite the more restricted contraint. With simulation optimization, we must choose when to stop the simulation using max function evaluations and by choosing a sample size that is not too computationally expensive. These heuristics, make it difficult to know how many samples and evaluations to use to give an actual optimal value. In contrast, linear programming stops when it has reached the optimal solution (or the closest it can get to it) and does not rely on heuristics. In linear programming, you do not have to worry about Monte Carlo error or heuristics which gives you more confidence that the LP optimum is reliable. Whereas, the simulation optimization there is a lot of uncertinaty because it relies heavily on heuristics.

References

Put any consulted sources here, including classmates you worked with/who helped you.

BEE 4750 11/8 Lecture "Simulation-Optimization" Slides

BEE 4750 9/29 Lecture "Linear Programming" Slides