BEE 4750 Lab 4: Simulation-Optimization

Name: Jonathan Marcuse

ID: jrm564

> **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 `~/Desktop/BEE4750-2/lab04-jrmarcuse`
```

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 $X_0 = 0$, $y_t \sim \text{LogNormal}(\log(0.03), 0.25)$, $q = 2.5$, 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

        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 given variables
q = 2.5;
b = 0.4
```

```
0.4
```

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)
        bounds = [zeros(T) 0.5*ones(T)];
        max_evals = 1000;
        results = optimize(a ->
        lake_opt(a,y,q,b,T,Xcrit),bounds,
        DE(options=Options(f_calls_limit=max_evals)))
```

```
Optimization Result
===================
  Iteration:       1
  Minimum:         0.210588
  Minimizer:       [0.121521, 0.0485756, 0.399366, …, 0.0721527]
  Function calls:  1000
  Feasibles:       0 / 1000 in final population
  Total time:      12.9770 s
  Stop reason:     Maximum objective function calls exceeded.
```

```
In [ ]: #Find the optimum value
        fx = minimum(results);
        x = minimizer(results);
        #Find number of failures
        display(lake_opt(x,y,q,b,T,Xcrit))
```

```
(0.21058759608000296, [0.9], [0.0])
```

The approximate optimum value is about 0.211 for the objective function, and there are 1000 failures, meaning none of the final values are feasible.

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?

In [ ]:
```
Random.seed!(1)
bounds = [zeros(T) 0.5*ones(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:         0.193087
 Minimizer:       [0.207163, 0.0325769, 0.427649, ..., 0.249454]
 Function calls:  5000
 Feasibles:       0 / 1000 in final population
 Total time:      62.2585 s
 Stop reason:     Maximum objective function calls exceeded.
```

I modified the total number of evaluations to be 5000 instead of 1000 and saw that it reduced the objective value, and appears to make the data more accurate.

In [ ]:
```
Random.seed!(1)
bounds = [0.1*ones(T) 0.5*ones(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:         0.25447
 Minimizer:       [0.26573, 0.126062, 0.44212, ..., 0.299563]
 Function calls:  5000
 Feasibles:       0 / 1000 in final population
 Total time:      65.9621 s
 Stop reason:     Maximum objective function calls exceeded.
```

When increasing the lower bound from 0 to 0.1, I found that the objective value increased which makes sense as now many values between 0 and 0.1 are no longer used in the optimization.

In [ ]:
```
Random.seed!(1)
bounds = [zeros(T) 0.5*ones(T)];
max_evals = 1000;
results = optimize(a ->
lake_opt(a,y,q,b,T,Xcrit),bounds,
DE(options=Options(f_calls_limit=max_evals)))
```

```
Optimization Result
===================
 Iteration:       1
 Minimum:         0.210588
 Minimizer:       [0.121521, 0.0485756, 0.399366, ..., 0.0721527]
 Function calls:  1000
 Feasibles:       0 / 1000 in final population
 Total time:      13.3182 s
 Stop reason:     Maximum objective function calls exceeded.
```

The above is the same as in question 1 for comparison to the simulation below where both bounds and range change.

In [ ]:
```
Random.seed!(1)
bounds = [zeros(T) 0.1*ones(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:      61.1570 s
 Stop reason:     Maximum objective function calls exceeded.
```

I manipulated both the bounds I was searching over and the maximum number of evaluations. I changed these two because the bounds searching over is likely to have a major effect on the optimal value, and then i increased max_evals because with more iterations the data will likely be more accurate.

With bounds held constant and max evaluations increased, the optimal value decreased slightly. If the max evaluations were held constant but the bounds were lowered or increased, the optimal value would decrease or increase respectively. In my final change of parameters above, I lowered the bounds to 0-0.1 and increased max evaluations to 5000 which was the only trial where the optimizer produced 3 feasible simulations. The optimal value was also very small as the bounds were adjusted to their lowest point out of my trials.

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 that the max evals and bounds have a great influence over the optimal value. When the bounds are changed, the area that is being searched over is altered, which in turn influences the possibility of feasibility. This was especially seen by my last example trial where I performed the greatest change in bounds and it was the only time the optimizer produced any feasible simulations. This is the same as our past work in linear programming when we changed the bounds of variables and then reevaluate for the solution, which can provide completely different solutions. In the past labs and homeworks we may change the upper or lower bounds on constraints which would in turn change the range of the decision variables and produce similar effects. When changing the maximum number of evaluations I saw that the optimal value would change slightly and potentially produce a more accurate result, however it took much longer to run. It is much more computationaly heavy, but has the ability to produce more reliable data.

References

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