HW 1 Solutions

2025-09-14

Due Date

Thursday, 9/11/25, 9:00pm

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.

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

Standard Julia practice is to load all needed packages at the top of a file. If you need to load any additional packages in any assignments beyond those which are loaded by default, feel free to add a `using` statement, though you may need to install the package.

```julia
using Random
using Plots
using GraphRecipes
using LaTeXStrings
using Distributions
```

```julia
# this sets a random seed, which ensures reproducibility of random number
 ↪  generation. You should always set a seed when working with random
 ↪  numbers.
Random.seed!(1)
```

```
TaskLocalRNG()
```

Problems (Total: 30 Points)

Problem 1 (12 points)

Problem 1.1

```julia
function minimum(array)
    # initialize the minimum value counter
    min_value = 0
    # update minimum values
    for i in 1:length(array)
        if array[i] < min_value
            min_value = array[i]
        end
    end
    # return found minimum
    return min_value
end

array_values = [89, 90, 95, 100, 100, 78, 99, 98, 100, 95]
@show minimum(array_values);
```

```
minimum(array_values) = 0
```

Solution:

The code initializes the minimum value (`min_value`) at `0`, which is below any of the values in `array_values`. Instead, if we initialize `min_value = array[1]`, any smaller values will be identified by the loop.

A correct solution might look like:

```julia
function minimum(array)
    # initialize the minimum value counter
    min_value = array[1]
    # update minimum values
    for i in 2:length(array)
        if array[i] < min_value
            min_value = array[i]
        end
    end
    # return found minimum
    return min_value
end

array_values = [89, 90, 95, 100, 100, 78, 99, 98, 100, 95]
@show minimum(array_values);
```

```
minimum(array_values) = 78
```

Problem 1.2

Your team is trying to compute the average grade for your class, but the following code produces an `UndefVarError`.

```julia
# enter student grade vector
student_grades = [89, 90, 95, 100, 100, 78, 99, 98, 100, 95]
# compute class average
function class_average(grades)
  average_grade = mean(student_grades)
  return average_grade
end

@show average_grade;
```

```
UndefVarError:
UndefVarError: `average_grade` not defined in `Main.Notebook`
Suggestion: check for spelling errors or missing imports.
Stacktrace:
 [1] macro expansion
   @ show.jl:1232 [inlined]
 [2] top-level scope
   @ ~/Teaching/BEE4750/website/solutions/hw01/hw01.qmd:142
```

Solution:

The `UndefVarError` is due to a problem with **scope**. The function `class_average()` is defined with an input `grades`, while `student_grades` is a global variable defined outside the function, and therefore is not available to the function. Additionally, `average_grade` is a variable defined within the `class_average()` function, and therefore is not accessible outside of its operation. If we replace `student_grades` inside the `class_average()` function with `grades`, and replace `average_grade` with the output of the function, the code works as intended:

```julia
# enter student grade vector
student_grades = [89, 90, 95, 100, 100, 78, 99, 98, 100, 95]
# compute class average
function class_average(grades)
  average_grade = mean(grades)
  return average_grade
end

@show class_average(student_grades);
```

```
class_average(student_grades) = 94.4
```

Problem 1.3

Your team wants to know the expected payout of an old Italian dice game called *passadieci* (which was analyzed by Galileo as one of the first examples of a rigorous study of probability). The goal of passadieci is to get at least an 11 from rolling three fair, six-sided dice. Your strategy is to compute the average wins from 1,000 trials, but the code you've written below produces a `MethodError`.

```julia
# function to simulate a passadieci roll: 3 6-sided dice
function passadieci()
    # this rand() call samples 3 values from the vector [1, 6]
    roll = rand(1:6, 3)
    return roll
end
# set number of trials and initialize outcome vector
n_trials = 1_000
outcomes = zero(n_trials)
# simulate number of passadieci rolls and count wins
for i = 1:n_trials
    outcomes[i] = (sum(passadieci()) > 11)
end
win_prob = sum(outcomes) / n_trials # compute average number of wins
@show win_prob;
```

```
MethodError:
MethodError: no method matching setindex!(::Int64, ::Bool, ::Int64)
The function `setindex!` exists, but no method is defined for this
 ↪  combination of argument types.
Stacktrace:
 [1] top-level scope
   @ ~/Teaching/BEE4750/website/solutions/hw01/hw01.qmd:178
```

Solution:

The line producing the `MethodError` is `outcomes[i] = (sum(passadieci()) > 11)`. The problem is actually with a prior line, `outcomes = zero(n_trials)`, which creates `outcomes`.

`zero()` is the Julia function to get a zero element (the additive identity) based on the type of the input. For example:

```julia
zero(3)
```

```
0
```

```julia
zero(3.0)
```

```
0.0
```

```julia
zero(false)
```

```
false
```

Since `zero()` returns a scalar element (as seen in the examples), trying to index its output with `outcomes[i]` results in the error, as Julia does not know how to index into a scalar (in this case, an `Int`).

However, the goal of the relevant line in the script is to initialize a vector consisting of zeros with length `n_trials`. What we actually want is the function `zeros()`. Fixing this gets rid of the error (I've also fixed another error with the code, namely the threshold for a "win", but this isn't essential for this problem):

```julia
# function to simulate a passadieci roll: 3 6-sided dice
function passadieci()
    # this rand() call samples 3 values from the vector [1, 6]
    roll = rand(1:6, 3)
    return roll
end
# set number of trials and initialize outcome vector
n_trials = 1_000
outcomes = zeros(n_trials)
```

```
# simulate number of passadieci rolls and count wins
for i = 1:n_trials
    outcomes[i] = (sum(passadieci()) >= 11)
end
win_prob = sum(outcomes) / n_trials # compute average number of wins
@show win_prob;
```

```
win_prob = 0.502
```

Problem 1.4

You're interested in writing some code to remove the mean of a vector from all of its components. You've written the following code and tried to test it on a random vector, but your code returns a `MethodError`.

```
# function to remove mean from a vector
function remove_mean(vect)
    # fucntion to compute the mean
    function compute_mean(vect)
        element_sum = 0 # initialize sum
        # compute mean and return
        for v in vect
            element_sum += v
        end
        return element_sum / length(vect)
    end

    m = compute_mean(vect) # compute mean
    # return demeaned vector
    return vect - m
end

random_vect = rand(1_000)
@show remove_mean(random_vect)
```

```
MethodError: MethodError(-, ([0.18289047842684425, 0.45597770410169736,
↪  0.9594724029201155, 0.017948696356010374, 0.5520891228901076,
↪  0.6353820717192517, 0.8136704848401125, 0.5645968320230955,
↪  0.10068341809548298, 0.09426267900500329  …  0.6001629008883226,
↪  0.17244039728995997, 0.7377333228188104, 0.5621409455051939,
↪  0.6144918333339874, 0.3355388856551146, 0.4096374458764567,
↪  0.6427983222170646, 0.5104261043946906, 0.6606913442154358],
↪  0.5042740941744438), 0x0000000000006906)
MethodError: no method matching -(::Vector{Float64}, ::Float64)
For element-wise subtraction, use broadcasting with dot syntax: array .-
↪  scalar
The function `-` exists, but no method is defined for this combination of
↪  argument types.
Closest candidates are:
  -(::Missing, ::Number)
   @ Base missing.jl:123
  -(::ChainRulesCore.NoTangent, ::Any)
   @ ChainRulesCore
↪  ~/.julia/packages/ChainRulesCore/Vsbj9/src/tangent_arithmetic.jl:61
  -(::Any, ::ChainRulesCore.NotImplemented)
   @ ChainRulesCore
↪  ~/.julia/packages/ChainRulesCore/Vsbj9/src/tangent_arithmetic.jl:50
  ...
Stacktrace:
 [1] remove_mean(vect::Vector{Float64})
   @ Main.Notebook ~/Teaching/BEE4750/website/solutions/hw01/hw01.qmd:243
 [2] macro expansion
   @ show.jl:1232 [inlined]
 [3] top-level scope
   @ ~/Teaching/BEE4750/website/solutions/hw01/hw01.qmd:247
```

Solution:

The line that causes the error is `return vect - m`. The problem is that Julia does not make assumptions about how to handle ambiguous operations like adding or subtracting a scalar (`m`) from a vector (`vect`): is this intended and should be done element-wise, or is it a sign that the wrong type was passed (*e.g.* the other input was intended to be a vector)? To make the intent of the coder explicit, Julia asks that you **broadcast** an operation that is intended to be applied to a scalar (such as addition or subtraction of another scalar) to be applied element-wise to a vector, instead of assuming that this is the intended behavior.

To broadcast a function `f()` over a vector `v`, you use a period, as in `f.(v)`. In this case, we want to change the problematic line to `return vect .- m`:

```julia
# function to remove mean from a vector
function remove_mean(vect)
    # fucntion to compute the mean
    function compute_mean(vect)
        element_sum = 0 # initialize sum
        # compute mean and return
        for v in vect
            element_sum += v
        end
        return element_sum / length(vect)
    end

    m = compute_mean(vect) # compute mean
    # return demeaned vector
    return vect .- m
end

random_vect = rand(1_000)
@show remove_mean(random_vect)
```

```
remove_mean(random_vect) = [-0.402858193077954, 0.17163216613218168, -0.18169592947668356, 0
```

```
1000-element Vector{Float64}:
 -0.402858193077954
  0.17163216613218168
 -0.18169592947668356
  0.060514380686886415
 -0.2591266252711959
 -0.01737820236308285
  0.37319528099391985
  0.23362003830781186
 -0.3090151132440434
 -0.06824664473090836

 -0.12715199682996414
  0.3553833867930265
  0.2658817795877152
  0.3071997855512709
  0.35638130936574974
  0.15303872493357518
 -0.10958007508724765
```

```
-0.11638803141199039
 0.4389100002859272
```

Problem 2 (18 points)

Cheap Plastic Products, Inc. is operating a plant that produces 100m^3/day of wastewater that is discharged into Pristine Brook. The wastewater contains 1kg/m^3 of YUK, a toxic substance. The US Environmental Protection Agency has imposed an effluent standard on the plant prohibiting discharge of more than 20kg/day of YUK into Pristine Brook.

Cheap Plastic Products has analyzed two methods for reducing its discharges of YUK. Method 1 is land disposal, which costs $X_1^2/20$ dollars per day, where X_1 is the amount of wastewater disposed of on the land (m^3/day). With this method, 20% of the YUK applied to the land will eventually drain into the stream (*i.e.*, 80% of the YUK is removed by the soil).

Method 2 is a chemical treatment procedure which costs \$1.50 per m^3 of wastewater treated. The chemical treatment has an efficiency of $e = 1 - 0.005X_2$, where X_2 is the quantity of wastewater (m^3/day) treated. For example, if $X_2 = 50$m^3/day, then $e = 1 - 0.005(50) = 0.75$, so that 75% of the YUK is removed.

Cheap Plastic Products is wondering how to allocate their wastewater between these three disposal and treatment methods (land disposal, chemical treatment, and direct disposal) to meet the effluent standard while keeping costs manageable.

The flow of wastewater through this treatment system is shown in Figure 1. Modify the edge labels (by editing the `edge_labels` dictionary in the code producing Figure 1) to show how the wastewater allocations result in the final YUK discharge into Pristine Brook. For the `edge_label` dictionary, the tuple (i, j) corresponds to the arrow going from node i to node j. The syntax for any entry is `(i, j) => "label text"`, and the label text can include mathematical notation if the string is prefaced with an `L`, as in `L"x_1"` will produce x_1.

```julia
A = [0 1 1 1;
    0 0 0 1;
    0 0 0 1;
    0 0 0 0]

names = ["Plant", "Land Treatment", "Chem Treatment", "Pristine Brook"]
# modify this dictionary to add labels
edge_labels = Dict((1, 2) => "", (1,3) => "", (1, 4) => "",(2, 4) => "",(3,
↪  4) => "")
shapes=[:hexagon, :rect, :rect, :hexagon]
xpos = [0, -1.5, -0.25, 1]
ypos = [1, 0, 0, -1]
```

```
p = graphplot(A, names=names,edgelabel=edge_labels, markersize=0.15,
 ↪  markershapes=shapes, markercolor=:white, x=xpos, y=ypos)
display(p)
```

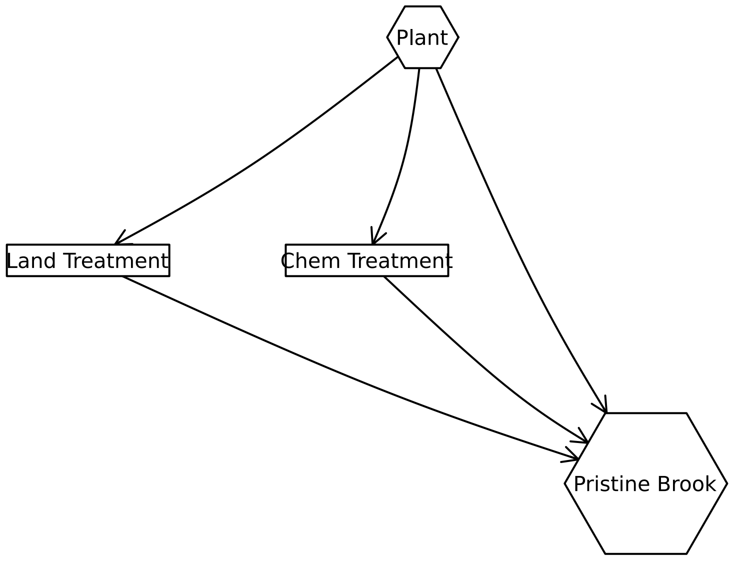

Figure 1: System diagram of the wastewater treatment options in Problem 4.

Problem 2.1

Formulate a mathematical model for the treatment cost and the amount of YUK that will be discharged into Pristine Brook based on the wastewater allocations. This is best done with some equations and supporting text explaining the derivation. Make sure you include, as additional equations in the model, any needed constraints on relevant values. You can find some basics on writing mathematical equations using the LaTeX typesetting syntax here, and a cheatsheet with LaTeX commands can be found on the course website's Resources page.

Solution:

These equations will be derived in terms of X_1 (the land disposal amount, in kg/day) and X_2 (the chemically treated amount, in kg/day), where $X_1 + X_2 \leq 100$ kg/day. Note that we don't need to explicitly represent the amount of directly disposed YUK, as this is $100 - X_1 - X_2$ and so is not a free variable.

The amount of YUK which will be discharged is

$$
\begin{aligned}
D(X_1, X_2) &= 100 - X_1 - X_2 + 0.2X_1 + 0.005X_2^2 \\
&= 100 - 0.8X_1 + (0.005X_2 - 1)X_2 \\
&= 100 - 0.8X_1 + 0.005X_2^2 - X_2
\end{aligned}
$$

The cost is

$$C(X_1, X_2) = X_1^2/20 + 1.5X_2.$$

Problem 2.2

Implement your systems model as a Julia function which computes the resulting YUK discharge and cost for a particular treatment plan.

Solution:

A Julia function for this model could look like:

```julia
# we will assume that X , X  are vectors so we can vectorize
# the function; hence the use of broadcasting. This makes unpacking
# the different outputs easier as each will be returned as a vector.
# Note that even though this is vectorized, passing scalar inputs
# will still work fine.
function yuk_discharge(X , X )
    # Make sure X  + X  <= 100! Throw an error if not.
    if any(X  .+ X  .> 100)                                          ①
        error("X  + X  must be less than 200")
    end
    yuk = 100 .- 0.8X  .+ (0.005X  .- 1) .* X                       ②
    cost = X .^2/20 .+ 1.5X
    return (yuk, cost)
end
```

① Checking for these kinds of errors is useful when there are hard limits on what arguments can be passed in. This syntax lets you throw an error which says something is going wrong in the code. In general, Julia style is to try to do a computation and throw an error if something goes wrong.

② We use broadcasting here to work on vectors of arguments for efficiency. This is in no way required.

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

Problem 2.3

Use your function to experiment with 1,000 different combinations of wastewater discharge and treatment. You can do this with either a grid search or by sampling from a Dirichlet distribution (a Dirichlet$(1, n)$ distribution will generate uniformly-weighted n-dimensional vectors whose components add up to 1; see the `Distributions.jl` documentation for how to sample from probability distributions in Julia). Plot the results of these experiments. Do any satisfy the YUK effluent standard (plot this as well as a dashed red line). What was the cost of solutions satisfying the standard? What can you say about the tradeoff between treatment cost and YUK discharge? You don't have to find an "optimal" solution to this problem, but what do you think would be needed to find a better solution?

Solution:

In this case, we will sample from a Dirichlet distribution, but a more systematic search would also be acceptable.

```julia
yuk_distribution = Dirichlet(3, 1)                                          ①
# Need to scale samples from 0 to 200, not 0 to 1
yuk_samples = 100 * rand(yuk_distribution, 1000)
D, C = yuk_discharge(yuk_samples[1,:], yuk_samples[2, :])

# Plot the discharge vs. cost and add a line for the regulatory limit
p = scatter(D, C, markersize=2, label="Treatment Samples")                  ②
vline!(p, [20], color=:red, label="Regulatory Limit")                       ③
# Label axes
xaxis!(p, "YUK Discharge (kg/day)")                                         ④
# For the y-axis label, we need to "escape" the $ by adding a slash
# otherwise it interprets that as starting math mode
yaxis!(p, "Treatment Cost (\$/day)")                                        ⑤
```

① The Dirichlet distribution is over combinations of values which add up to 1, which is what we want for shares of the total YUK discharge. The 3D Dirichlet distribution with parameters equal to 1 is basically uniform over these combinations. See: https://juliastats.org/Distributions.jl/stable/multivariate/#Distributions.Dirichlet.

② This is a basic scatter plot with a label for the plot elements. If we wanted to turn the legend off in any plot, use `legend=false` as an argument.

③ This is how to add a vertical line to a plot with a label. The syntax for a horizontal line is `hline(...)`. The `!` at the end of `vline!()` is important: this is standard Julia syntax to distinguish commands which *mutate* (or change) their input (in this case, the first argument `p`, the plot object), as this is not always desirable behavior.

④ This is how to change axis labels. Notice that this also mutates the input plot.

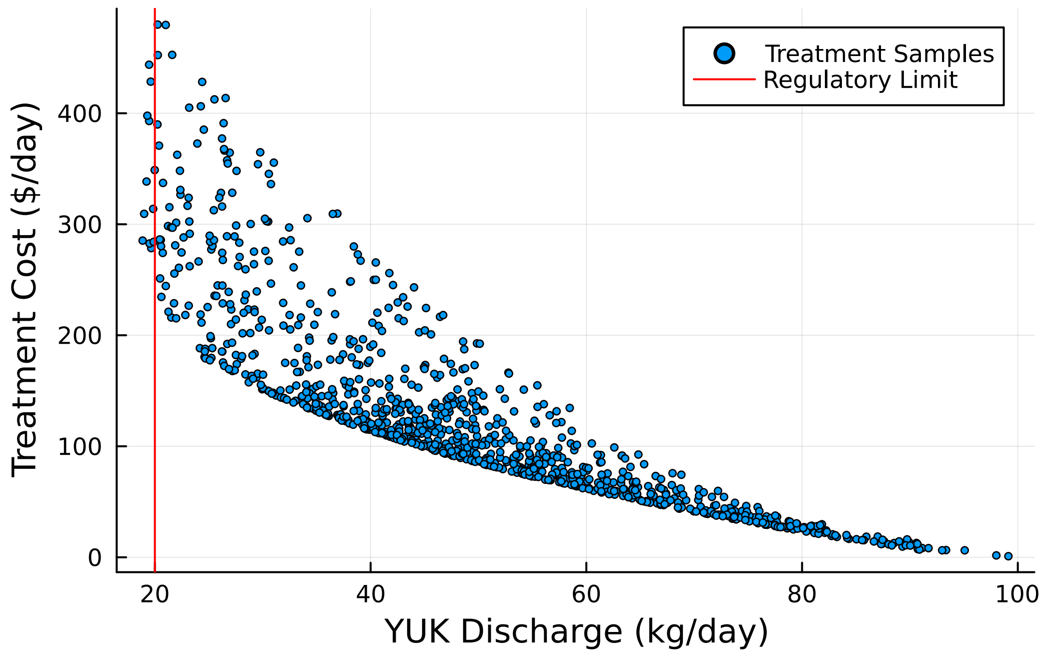

Figure 2: Sampled solutions for the wastewater allocation problem in Problem 2, showing cost vs. YUK concentration. The red line marks the regulatory discharge limit of 20kg/day.

We can see that there are a few treatment strategies which comply with the limit, but they are fairly expensive. This is an example of a *tradeoff* between two objectives[1], where one has to make a choice between what objectives to prioritize. But one thing to note is that just choosing an expensive strategy does not guarantee compliance.

Problem 2.4

Find the strategies which minimize cost and YUK discharge (these will be different strategies) analytically and find the values of the objective metrics. Plot these values in the plot that you created for Problem 2.3. How do their values compare to the spread of values that you found in that problem? Would you select either of them (explain why or why not)?

Solution:

To minimize these functions, we want to look at either the critical points (where the partial derivatives are zero) or at the constraints. For cost, $dC/dX_2 = 1.5$ is never zero, so there are no critical points. However, the lowest-cost point occurs at the boundary $X_2 = 0$ and $X_1 = 0$,

[1]More on this later in the semester!

where there is no cost (unsurprisingly, doing nothing often costs the least). But this does not comply with the constraint, as the YUK discharge is 100 kg/day.

The YUK discharge function D has the following partial derivatives:

$$\begin{aligned}
\frac{\partial D}{\partial X_1} &= -0.8 \\
\frac{\partial D}{\partial X_2} &= 0.01X_2 - 1,
\end{aligned}$$

so we need to check the values along the boundaries $X_1 = 100 - X_2$, $X_1 = 0$, and $X_2 = 0$.

$$D(100 - X_2, X_2) = 20X_2 - 0.005X_2^2,$$

which has a partial derivative

$$\frac{\partial D}{\partial X_2}(100 - X_2, X_2) = -0.2 + 0.01X_2.$$

This has a minimum at $X_2 = 20$ and therefore $X_1 = 80$, with a corresponding YUK discharge of 18 kg/day. If $X_2 = 0$, the minimum value (at $X_1 = 100$) is a discharge of 20 kg/day, while the minimum value along $X_1 = 0$ is 50 kg/day ($X_2 = 100$). Thus, the minimum occurs at $X_1 = 80$ and $X_2 = 20$, with a corresponding cost of \$350/day.

Adding these points to the plot:

```julia
# least cost solution
least_cost = yuk_discharge(0, 0)
scatter!(p, least_cost, color=:orange, label="Least Cost Solution")
# least discharge solution
least_discharge = yuk_discharge(80, 20)
scatter!(p, least_discharge, color=:purple, label="Least Discharge Solution")
```

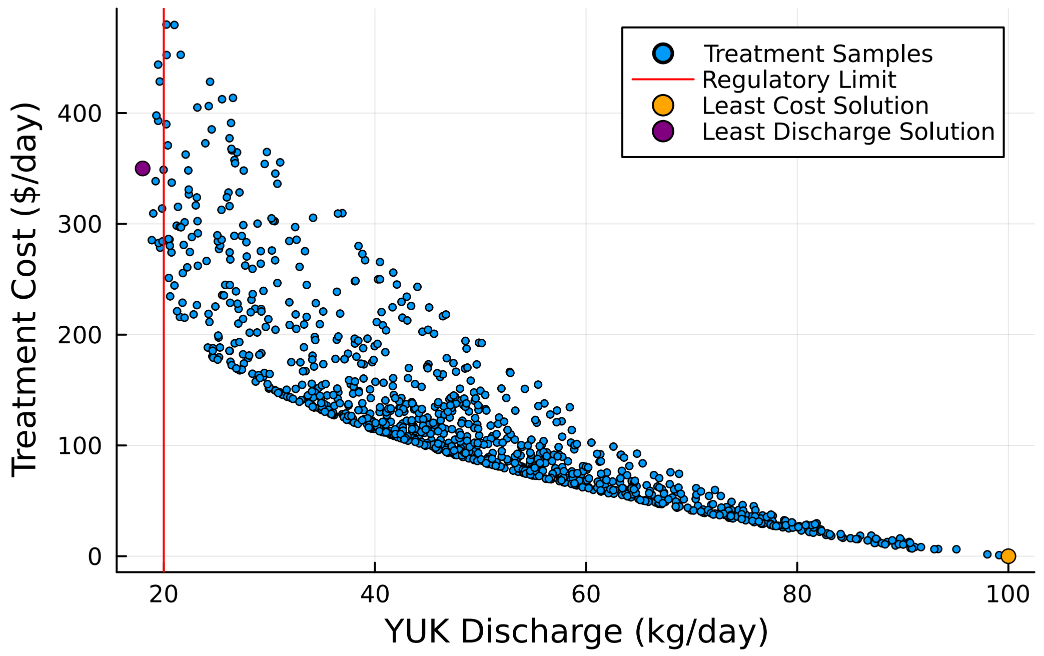

Figure 3: Added the solutions from Problem 2.4 (purple for least-discharge, orange for least-cost circles) to the sampled solutions from Problem 2.3.

The least-cost solution, unsurprisingly, would be pretty bad given how much the YUK discharge violates the regulatory constraint. However, the least-discharge solution looks pretty good: there are more expensive treatment plans which comply with the regulatory constraint, but not many that are less expensive.

References

List any external references consulted, including classmates.