Homework 1 Solutions

2024-09-07

Overview

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

```
using Random
using Plots
using GraphRecipes
using LaTeXStrings
```

```
# 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: 50/60 Points)

Problem 1 (15 points)

The following subproblems all involve code snippets that require debugging. For each of them:

- identify and describe the logic and/or syntax error;
- write a fixed version of the function;
- use your fixed function to solve the problem.

Problem 1.1

The problem is with the initialization `min_value = 0`, which means no other values can be below it. Instead, we can initialize `min_value` to be `array[1]` and start looping at index `i=2`:

```
function minimum(array)
    min_value = array[1]                                                    ①
    for i in 2:length(array)
        if array[i] < min_value
            min_value = array[i]
        end
    end
    return min_value
end

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

① Initializing `min_value` at `array[1]` ensures that we start with a candidate value; then we can loop beginning with index 2.

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

Problem 1.2

There are two issues here.

1. The first error is trying to access `average_grades`, which is only defined inside the `class_average()` function. This is an issue of *scope*: the variable `average_grades` doesn't exist *globally*.
2. The second error is that `mean()` is not part of the `Base` Julia library, but rather the `Statistics` package (part of the usual Julia installation, but needs to be explicitly imported). We could import it with `using Statistics` and use `mean()`, but in this case let's just take the sum and divide by the length.

```
student_grades = [89, 90, 95, 100, 100, 78, 99, 98, 100, 95]
function class_average(grades)
  average_grade = sum(grades) / length(grades)
  return average_grade
end
```

```
avg_grade = class_average(student_grades)                                    ①
@show avg_grade;
```

① Now `avg_grade` exists after being assigned the output of `class_average()`. Note that we didn't reuse the name `average_grade` as that could result in strange outcomes if notebook cells were run out of order.

```
avg_grade = 94.4
```

Problem 1.3

The `setindex` error comes from the use of `zero()` instead of `zeros()`:

- `zero(n)` creates a zero variable of the same type of the argument `n` (*e.g.* `zero(1)` is `0` and `zero(1.5)` is `0.0`).
- `zeros(n)` creates an array of zeroes of dimension `n`, where `n` can be an integer or a tuple (for a matrix or higher-dimensional array).

As a result, the original call `outcomes = zero(n_trials)` sets `outcomes=0`, but then when we try to set `outcomes[1]` in the loop, this is undefined as a scalar does not have an index, resulting in the error.

```
function passadieci()
    # this rand() call samples 3 values from the vector [1, 6]
    roll = rand(1:6, 3)
    return roll
end
n_trials = 1_000
outcomes = zeros(n_trials)                                                   ①
for i = 1:n_trials
    outcomes[i] = (sum(passadieci()) > 11)                                   ②
end
win_prob = sum(outcomes) / n_trials # compute average number of wins
@show win_prob;
```

① Changed `zero` to `zeroes`; note that it's generally preferable to initialize an array of the desired size instead of creating an empty vector and using `append`, as that approach can get quite slow as the number of `append` calls increases.
② Now that `outcomes` is a vector, we can access its indexed values.

```
win_prob = 0.374
```

We could also use comprehensions and broadcasting (applying a function across each element of an array) instead of initializing `outcomes` as a zero vector and looping to fill it:

```
rolls = [passadieci() for i in 1:n_trials]                    ①
outcomes = sum.(rolls) .> 11                                  ②

win_prob = sum(outcomes) / n_trials # compute average number of wins
@show win_prob;
```

① This is an example of a *comprehension*, which is an inline loop that produces an array; the advantage is that this is sometimes more readable than an explicit loop when the in-loop commands are short.

② This is an example of *broadcasting* (actually twice), indicated by the use of the decimal `.` in function calls. `sum.(v)` applies the function `sum` to every element of `v`, so that adds each of the individual roll vectors to get the sum of those three dice, and `.>` does an element-wise comparison of each of those sums to `11`. You would get an error if you tried `sum.(rolls) > 11` because Julia does not want to make assumptions about your intent in comparing a vector with a scalar.

```
win_prob = 0.364
```

Problem 2 (5 points)

Let's outline the steps in `mystery_function`:

1. Initialize an empty vector.
2. If a value `v` is not already in `y`, add `v` to `y`.
3. Return after looking at all values.

This means that `mystery_function` selects and returns the unique values in `values`, which is confirmed by the test case.

There are many ways to add comments, but we could comment as follows:

```
# mystery_function:
#    Inputs:
#       - values: vector of numeric values
#    Outputs:
#       - vector of unique values from the input
function mystery_function(values)
    y = [] # initialize as an empty vector because we don't know how many
 ↪  values we will end up with
```

```
    for v in values
        if !(v in y) # if a value is not already in y
            append!(y, v) # append to y
        end
    end
    return y
end

list_of_values = [1, 2, 3, 4, 3, 4, 2, 1]
@show mystery_function(list_of_values);
```

```
mystery_function(list_of_values) = Any[1, 2, 3, 4]
```

The built-in Julia function which does the same thing is `unique()` (found using a Google search for "unique Julia vector function").

```
@show unique(list_of_values);
```

```
unique(list_of_values) = [1, 2, 3, 4]
```

Problem 3 (10 points)

You're interested in writing some code to remove the mean of a vector.

- Write a function `compute_mean(v)` which sums all of the elements of a vector `v` using a `for` loop and computes the mean.
- Make a random vector `random_vect` of length 10 using Julia's `rand()` function. Use your `compute_mean()` function to calculate its mean and subtract it from `random_vect` **without a loop** (using a Julia technique called *broadcasting*; feel free to consult the Julia documentation and search as necessary). Check that the new vector has mean zero.

Our `compute_mean` function should:

1. Initialize a running sum at 0;
2. Loop over all elements of `v`;
3. Add each element in turn to the running sum;
4. Divide the running sum by the number of elements and return.

```
function compute_mean(v)
    v_sum = 0
    for val in v
        v_sum += val
    end
    return v_sum / length(v)
end

random_vect = rand(10)
rand_mean = compute_mean(random_vect)
@show rand_mean;
```

```
rand_mean = 0.3573789691628376
```

To subtract off the mean from `random_vect`, we can broadcast the subtraction operator by putting a decimal in front: `.-`.[1]

```
random_vect_demean = random_vect .- rand_mean
@show compute_mean(random_vect_demean);
```

```
compute_mean(random_vect_demean) = 2.2204460492503132e-17
```

We have produced a mean-zero random vector! But this isn't *exactly* zero due to numerical precision. This doesn't really matter, as the non-zero entries are insignificant digits, which we can see if we round (which we should do anyway):

```
@show round(compute_mean(random_vect_demean); digits=1);                    (1)
```

(1) The `round(y; digits=n)` function rounds `y` to `n` digits after the decimal place, defaulting to `n=0` (which rounds to the nearest integer, though printing a decimal due to the *type* of the variable).

```
round(compute_mean(random_vect_demean); digits = 1) = 0.0
```

[1] As a reminder, broadcasting involves applying a function element-wise. If we just tried to subtract `random_vect - rand_mean`, Julia would throw an error because it doesn't know if it should try element-wise subtraction or if we made a mistake in trying to subtract a scalar from a vector, and Julia's design is to err on the side of throwing an error unless we specifically say that we want an element-wise operation through broadcasting.

Problem 4 (20 points)

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.

```
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) => L"X_1", (1,3) => L"X_2", (1, 4) => L"100 - X_1 -
 ↪  X_2", (2, 4) => L"0.2X_1",(3, 4) => L"0.005X_2^2")
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)
```

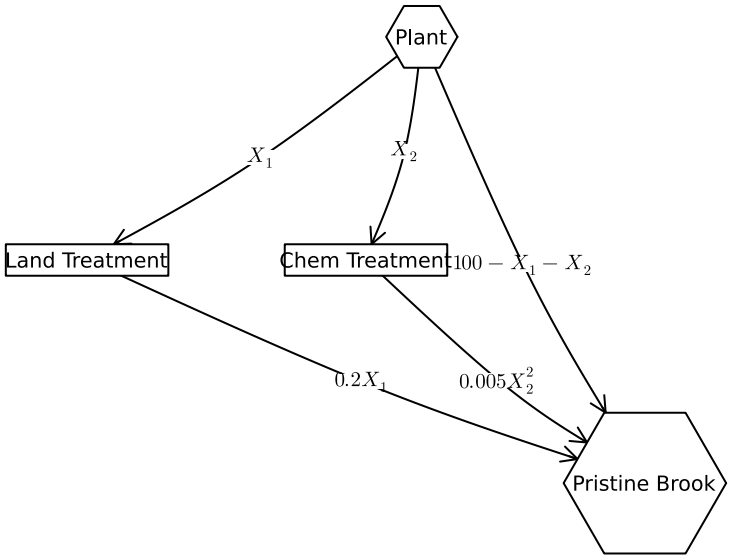

Figure 1: System diagram of the wastewater treatment options in Problem 4.

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

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

A Julia function for this model could look like:

```
# 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_1 + X_2 must be less than 200")
    end
    yuk = 100 .- 0.8X_1 .+ (0.005X_1 .- 1) .* X_2                    ②
    cost = X_2 .^2/20 .+ 1.5X_2
    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)
```

Now, let's experiment with different outcomes.[2] Some other options include just randomly sampling values (but be careful of not sampling impossible combinations of X_1 and X_2), manually searching, or setting up a grid of combinations.

```
# Install and load Distributions.jl
Pkg.add("Distributions")                                              ①
using Distributions

yuk_distribution = Dirichlet(3, 1)                                    ②
# 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")            ③
vline!(p, [20], color=:red, label="Regulatory Limit")                 ④
# Label axes
xaxis!(p, "YUK Discharge (kg/day)")                                   ⑤
# 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)")
```

[2] We left this intentionally open for you to conceptualize how to generate combinations and to look into different ways of implementing these in Julia. For a more systematic approach, we can sample combinations from a Dirichlet distribution, which samples combinations which add up to 1. This will require installing and loading the `Distributions.jl` package (we will spend more time working with `Distributions.jl` later).

① This is how we add new packages that are not in the existing environment and then load them.

② 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.

③ 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.

④ 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.

⑤ This is how to change axis labels. Notice that this also mutates the input plot.

```
  Resolving package versions...
No Changes to `~/Teaching/BEE4750/website/solutions/hw01/Project.toml`
No Changes to `~/Teaching/BEE4750/website/solutions/hw01/Manifest.toml`
```

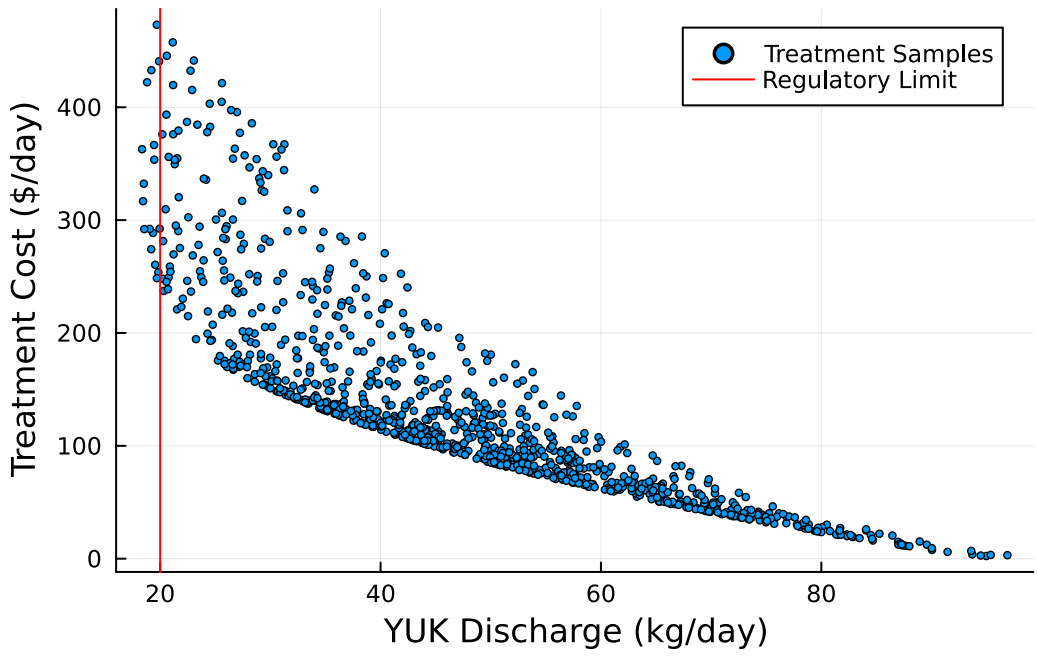

Figure 2: Sampled solutions for the wastewater allocation problem in Problem 4, showing cost vs. YUK concentration. The red line marks the regulatory discharge limit of 20kg/day.

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[3], 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 5 (10 points)

Problem 5.1

Here is one solution: Julia includes a function `iseven()` which returns 1 if the argument is even and 0 if it is odd. So we can use a comprehension to evaluate `!iseven(x)` (the `!` is Boolean negation) over the range `0:149`, which will return a vector of 1s and 0s, and then add up the vector to get the count of odd numbers. Another approach could be to use the `isodd()` function directly.

```
odd_count = sum([!iseven(x) for x in 0:149])
```

```
75
```

Problem 5.2

```
function polynomial(x, a)
    p = 0 # initialize value of polynomial
    for i in eachindex(a)                    ①
        p += a[i] * x^(i-1)                  ②
    end
    return p
end

a = [1, 0, -1, 2]
x = 2
@show polynomial(x, a);                      ③
```

① `for i in eachindex(a)` is the same thing as `for i = 1:length(a)` (iterating over the indices of `a`) but is preferred in Julia to provide more flexibility with how the array `a` is indexed, such as for multidimensional arrays. There is also `for b in a` which iterates over the values in `a` rather than the indices, and `for (i, b) in pairs(a)` which iterates over both the indices and values without requiring the line `b = a[i]`.

② Julia indexing starts at 1, so `i=1` corresponds to the constant term, or a power of 0.

[3] More on this later in the semester!

③ The `@show` macro formats the output of the command nicely and prints it; the semi-colon at the end suppresses the normal output of the notebook cell, which is printed by default without the semi-colon.

```
polynomial(x, a) = 13
```