Homework 0

BEE 4750/5750

Due: Sep 02, 2022 by 9:00pm ET

Overview

Learning Objectives

After completing Homework 0, you should be able to:

- clone GitHub repositories;
- use Julia to write basic code;
- use `Weave.jl` to compile reports which incorporate Julia code;
- submit assignments using Gradescope.

Homework 0 will only be graded based on completion, but make sure you don't run into any problems with this workflow, which will be repeated for future homework assignments.

Problems

Problem 1: Writing Some Basic Code

This problem focuses on writing basic Julia code and including and referencing that code in a `Weave.jl` report.

Problem 1.1: *Writing A Basic Function*

Write a function to compute the square of a number. Include your code in a fenced code block like that below:

```
function square_number(x)
    # your code here
    return output
end
```

Problem 1.2: ***Referencing Output***

Write a sentence where you include a real-time calculation of 5^2 using your function. You don't need to do this by evaluating your function in real time, but you can use syntax like the following:

```
"We can see that $x^2 =$ `j square_number(x)`."
```

Problem 1.3: ***Making Plots***

Evaluate `square_number()` over the interval $[-10, 10]$, then plot the results. This tutorial shows some plotting basics and provides links to other resources. *Make sure that you label your axes and any other relevant plot elements; this will be part of future rubrics.*

To include a plot in your report dynamically, you can use a code block that outputs a figure; see the `Weave.jl` documentation for a description of the chunk options that you can use with figures. The use of the `label` chunk option lets you refer to the plot in your report using LaTeX, using `\ref{fig:square-plot}` (or whatever label). Note that this won't work with HTML. You can also just directly include figures using standard Markdown if you've generated them separately.

Problem 2: Square root by Newton's method

This problem involves implementing an algorithm: in this case, Newton's method for computing square roots. It was shamelessly copied from MIT's Introduction to Computational Thinking course.

The algorithm is as follows:

Given $x > 0$, the desired output is $\sqrt{x}$.

1. Take a guess `a`
2. Divide `x` by `a`
3. Update `a` to be the average of `x/a` and `a`,
4. Repeat until `x/a` is close enough to `x`.
5. Return `a` as the square root.

Problem 2.1: ***Justify Step 3***

Why must $\sqrt{x}$ lie between x/a and a, as in Step 3 of the above algorithm? Take advantage of any math typesetting you might need using LaTeX.

Problem 2.2: *Implement the Algorithm*

Implement the above algorithm in Julia. Notice that Step 4 requires some interpretation for "close enough"; this is usually done by including an additional parameter specifying an error tolerance.

Test your code by outputting $\sqrt{2}$.

Problem 3: Working with Vectors and Matrices

We won't be doing much (if any) linear algebra in this class, but vectors and matrices are useful data structures, so let's see how to use them.

Problem 3.1: *Generating Random Values*

Make a random vector of length 20 using the `rand()` function. *Note*: We'll see later in the course how to use `Distributions.jl` to sample from particular distributions; using `rand()` in this way just samples from a uniform distribution over the unit interval $[0, 1]$.

Problem 3.2: *Calculating a Mean*

Implement your own `mean()` function to calculate the mean of a vector using a `for` loop and the random vector from above. Then write another function `demean()` which subtracts the mean from every element of the vector using your `mean()` function.

> **NOTE: Mutation**
>
> Julia has particular conventions around functions which *mutate* the input data. What do we mean by mutation? A function can operate on data by modifying the original vector, or by creating a new vector. For example, in this problem, you might write the following function:

```
function demean(vect)
    ... # some other code might be needed here
    for i in 1:length(vect)
       vect[i] -= mean(vect)
    end
end
```

> Given how Julia handles memory and passing arguments (the details of which are not essential), this would mutate the original array and change the underlying data. In general, this is undesirable behavior unless high levels of code optimization are required (which they won't generally be for this course, and rule #1 of coding is not to prematurely optimize code), as the original data then cannot be reused.

Instead, try to work with copies of data, particularly when they aren't really large. This is always done when you use the `=` operator; *e.g.* `x = 2*x`.

If you write a function which does mutate the original data, the convention in Julia is to append an exclamation mark (`!`) to its name to indicate that it is doing so. So `sort(x)` returns a sorted copy of `x`, while `sort!(x)` sorts `x` in place. In other words, our function above should have been called `demean!(vect)`.

Problem 3.3: ***Accessing Array Elements with Indices***

Create a vector of 10 elements, where the center 6 elements are equal to 1 and the others are equal to 0. *Remember that indexing in Julia starts with 1, not 0.*

Problem 3.4: ***Working with Matrices***

Using the `rand()` function, create a random 5x5 matrix. Then subtract the mean of each column from that column.

Problem 4: Writing a Simulation Model

In this problem, you will put some of the previous pieces together, and also see how to draw samples from probability distributions in Julia.

We will use the following model of shallow-lake eutrophification (Carpenter et al., 1999; Quinn et al., 2017), which we will discuss in more depth later in the course. Let X_t denote the phosphorous amount (dimensionless) in a lake at time t. Over the course of $t = 1, \ldots, T$ years, the phosphorous dynamics in the lake are

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

where a_t is the anthropogenic (controlled, point source) phosphorous input, y_t is the natural (uncontrolled, non-point source) phosphorous input, q is a parameter controlling the phosphorous recycling rate, and b is a parameter controlling the rate at which phosphorous is lost from the lake.

For this problem, we will assume that $a_t = a$ is a constant level of discharge and y_t are random variables distributed according to a log-normal distribution with log-mean μ and standard deviation σ. The parameter values are:

Parameter	Value	Units
a	0.4	dimensionless
q	2	dimensionless
b	0.42	dimensionless
μ	log(0.03)	dimensionless
σ	0.005	dimensionless
T	100	years

Problem 4.1: ***Simulating Random Variables***

Probability distributions in Julia are handled using the `Distributions.jl` package. If this package is added to a Julia project (*e.g. included in a relevant `Project.toml`), we can load it by including the following code. This code assumes that the `Project.toml` is locted in the current working directory (which it will be for your assignment repositories), but the path can be changed to any location as noted.

```
# load the Pkg package manager
import Pkg
# activate the project environment. The "." references the current working
directory. This can be changed to any particular path.
Pkg.activate(".")
# instantiate the project environment. This installs any needed packages.
Pkg.instantiate()

# load Distributions.jl
using Distributions
```

The above code will also work for any other Julia package. Typical Julia style is to load `Pkg` and activate the project environment at the start of a document, while needed packages can be loaded at the beginning or as needed.

`Distributions.jl` provides a consistent interface to work with probability distributions, which is described in its documentation. For example, to work with a normal distribution with mean 3.5 and standard deviation 1, we can create the distribution:

```
my_normal_dist = Normal(3.5, 1)
```

Then we can draw random values from this distribution. To draw one value, use

```
rand(my_normal_dist)
```

while to draw a vector of multiple values, use

```
rand(my_normal_dist, 100)
```

Notice that once the distribution is specified, `rand` doesn't care what type of distribution it is! This is a nice feature of `Distributions.jl`, and also allows it to cleanly play with other libraries like `StatsBase.jl` and `Plots.jl`. `Distributions.jl` can construct many different probability distributions, which are specified in the documentation.

Now, draw 100 samples from a log-normal distribution with log-mean μ and standard deviation σ. These will represent the natural phosphorous inflows y_t for $t = 1, \ldots, T$.

Problem 4.2: ***Writing the Simulation Function***

Next, let's write a function which simulates the lake phosphorous dynamics. Write a simulation function which takes as arguments the discharge level a, the vector of natural runoffs y_t, the phosphorous-cycling parameters b and q, the simulation length T, and an initial lake phosphorous level X_0, and which returns the time series of lake phophorous levels X_t.

Problem 4.3: ***Plot Phosphorous Levels***

Use your function to simulate the lake phosphorous levels from $t = 1$ to 100 years, starting from an initial concentration of X_0. Make a plot of the lake phosphorous levels. Does anything look interesting about this plot?

References

Carpenter, S. R., Ludwig, D., & Brock, W. A. (1999). Management of Eutrophication for Lakes Subject to Potentially Irreversible Change. Ecological Applications, 9(3), 751-771. https://doi.org/10.2307/2641327

Quinn, J. D., Reed, P. M., & Keller, K. (2017). Direct policy search for robust multi-objective management of deeply uncertain socio-ecological tipping points. Environmental Modelling & Software, 92, 125-141. https://doi.org/10.1016/j.envsoft.2017.02.017