Problem Set 1

Quantiative Economics, Fall 2024

October 17, 2024

Due date: 31.10.2024

General Notes

This is your first programming problem set in `Julia`, designed to assess your understanding of fundamental programming concepts. It serves as a checkpoint to help you determine whether you have a strong grasp of the basics or if you need further practice. As the course progresses, the topics will become increasingly complex, so it's essential to build a solid foundation in core areas such as conditional statements (`if` statements), loops, functions, and vectorized operations. This problem set is intended to provide you with information about your current level of skill.

At this stage, it is still relatively easy to catch up on any gaps in your knowledge, but remember that programming is a skill that can only be developed through practice.

In future problem sets, you may have the freedom to choose which problems to solve. However, for this first set, you are required to solve all the given problems. Good luck and don't hesitate to start early, ask us for advice and use online resources!

General Hints:

1. Throughout this Problem Set, you are frequently asked to write functions. However, it is often easier to first test your code with some arbitrary parameter values before encapsulating the code within a function. Once your code works for a specific benchmark parameter, you can then put it into a function.

2. You will need three packages to complete this Problem Set. Be sure to add them using the procedure we discussed in class (refer to the second video that was sent to you via email for further details). Ensure that you are working in the correct folder, enter package mode, activate the local environment (`activate .`), add the required packages using `add DelimitedFiles, Plots, Statistics`, and finally import the packages in your code with: `using DelimitedFiles, Plots, Statistics`.

3. Remember to upload your group's code and write-up to GitHub. You need to share the link to your repository with us. You are

expected to use the same repository for all future Problem Sets.

Problem 1: Odd or Even

Write a function `odd_or_even(n)` that takes an integer n and prints:

- "Odd" if the number is odd.
- "Even" if the number is even.[1]

[1] Hint: In `Julia`, there is a function `iseven()` that returns `true` if its input is an even number. Another way to check if a number is even is by using the remainder operator %. For example, `6%2` will return 0, which means the number is even. You can use either method.

Example:

```
odd_or_even(7)   # Output: Odd
odd_or_even(12)  # Output: Even
```

Important: The `if` statements should be inside the body of the function. You can define the function as follows:

```
function xyz(n)
    # Your code here
end
```

Inside the function body, add the relevant conditional statements using `if`, `elseif`, and `else` clauses.

Problem 2: Boolean operators

Write a function `compare_three(a, b, c)` that takes three numbers and prints:

- "All numbers are positive" if all three numbers are greater than zero.[2]
- "At least one number is not positive" if at least one number is not greater than zero.[3]
- "All numbers are zero" if all three numbers are equal to zero.

[2] Hint: Recall the Boolean operators. For example, `x > 0 && x < 10` will return `true` if x is greater than 0 AND less than 10. On the other hand, `x < 0 || x > 10` will return `true` if x is either less than 0 OR greater than 10.

[3] Hint: You can use the negation operator `!`. For example, if `x = -2`, then `x > 0` returns `false`. But `!(x > 0)` returns `true` because it checks if x is NOT greater than zero.

Example:

```
compare_three(1, 2, 3)    # Output: All numbers are positive
compare_three(-1, 5, 7)   # Output: At least one number is not positive
compare_three(0, -4, 3)   # Output: At least one number is not positive
compare_three(0, 0, 0)    # Output: All numbers are zero
```

Problem 3: Factorial Calculation Using a Loop

Write a function `my_factorial(n)` that takes a positive integer n and calculates its factorial using a loop. The factorial of a number n, denoted as $n!$, is the product of all positive integers from 1 to n.

The formula for the factorial of n is:

$$n! = 1 \times 2 \times 3 \times \cdots \times (n-1) \times n$$

Hints:

1. Initialize a variable `result=1` (multiplying by 1 is neutral).
2. Use a `for` loop to iterate over each integer from 1 to n.
3. In each iteration, multiply `result` by the current number.
4. After the loop ends, return the value of `result`.

Example:

```
my_factorial(5)  # Output: 120 (because 5! = 1 × 2 × 3 × 4 × 5 = 120)
my_factorial(7)  # Output: 5040 (because 7! = 1 × 2 × 3 × 4 × 5 × 6 × 7 = 5040)
```

Problem 4: Count Positive Numbers Using a Loop

Write a function `count_positives(arr)` that takes an array of numbers `arr`, loops over the elements, and counts how many numbers in the array are positive. The function should print the total count of positive numbers.

Example:

```
count_positives([1, -3, 4, 7, -2, 0])  # Output: 3
count_positives([-5, -10, 0, 6])       # Output: 1
```

Hint: Initialize a counter (e.g., `counter = 0`) before you start the loop.

Problem 5: Plotting Powers of x Using a Loop

Write a function `plot_powers(n)` that takes a positive integer n and plots the powers of x from x^1 to x^n, in the range $[-10, 10]$. The function should loop over the values 1 to n and add each power plot to the same graph.

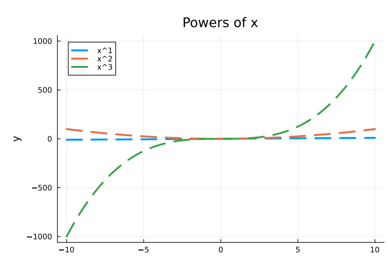

Figure 1: The output of `plot_powers(3)` should look something like that

Steps to follow:

1. **Ensure that the Plots package is imported.** Use the command `using Plots` to import the package.
2. **Define the function:** Define a function `plot_powers(n)`, where n represents the highest power to be plotted.
3. **Initialize an empty plot:** Create an empty plot that will serve as the base for adding multiple plots. [4]
4. **Set up a loop:** Use a `for` loop to iterate through the numbers from 1 to n.

[4] Hint: You can initialize an empty plot with `power_plot = plot()`. This empty plot will later be filled with lines representing the corresponding powers of x.

5. **Define the power function:** In each iteration, define a function that computes x^i. This will allow you to plot consecutive powers of x for each value of i in the loop.

6. **Plot each function:** Add each new power function to the plot by updating `power_plots`. [5] Plot the function x^i over the range $[-10, 10]$ with a step size of 0.2. Customize the plot by:

   - Labeling each line with according power x^i.
   - Setting the line width to 3.
   - Using dashed lines for each plot.
   - Labeling the x-axis as "x".
   - Labeling the y-axis as "y".
   - Adding a title to the plot, such as "Powers of x".

7. **Return the final plot:** After looping through all the powers, return the complete plot that includes all the individual power functions of x.

[5] Hint: Use the exclamation mark (`plot!()`) to modify an existing plot.

Hint: Remember to assign a name to the returned plot when calling your function. For example, `my_plot = plot_powers(3)` will store the resulting plot in the object `my_plot`.

Hint: If the plot does not automatically display, try using the command `display(my_plot)` to ensure that the plot appears in the plots panel.

Problem 5: Count Positive Numbers Using Broadcasting

Write a function `count_positives_broadcasting(arr)` that takes an array of numbers `arr` and counts how many numbers are positive using broadcasting.

Steps to Follow:

1. **Create a Boolean array:** Use broadcasting to compare each element of the array with 0. This will create a Boolean array where each element is `true` if the corresponding number is positive, and `false` otherwise. The syntax for broadcasting uses the dot operator, e.g., `arr .> 0`, which applies the comparison element-wise.[6].

2. **Count the positive numbers:** Use the `sum()` function to count the number of `true` values in the Boolean array. In Julia, `true` is treated as 1 and `false` as 0, so the sum of the Boolean array will give the total number of positive values. [7].

[6] Given an array `arr = [1, -3, 4, 7, -2, 0]`, broadcasting `arr .> 0` will produce the Boolean array `[true, false, true, true, false, false]`.

[7] Applying `sum()` to this Boolean array: `[true, false, true, true, false, false]` will count the number of `true` values. In this case, the sum is 3, indicating that there are 3 positive numbers in the original array.

Example:

```
count_positives_broadcasting([1, -3, 4, 7, -2, 0])  # Output: 3
count_positives_broadcasting([-5, -10, 0, 6])       # Output: 1
```

Problem 6: Standard Deviation Using Broadcasting and Vectorized Operations

Write a function `standard_deviation(x)` that takes an array of numbers **x** and calculates the standard deviation using broadcasting and vectorized operations, without explicit loops.

Formula:

For a sample, use the unbiased estimator for the standard deviation:

$$\hat{\sigma} = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \hat{\mu})^2}$$

where $\hat{\mu}$ is the estimated sample mean, and n is the number of elements in the sample.

Steps to Follow:

1. **Calculate the mean:** The mean is calculated by summing all elements of the array and dividing by the number of elements:

   $$\hat{\mu} = \frac{\text{sum}(\mathbf{x})}{\text{length}(\mathbf{x})}$$

   In `Julia`, the function `length()` returns the number of elements in the vector.

2. **Calculate the squared differences from the mean:** Use broadcasting to subtract the mean of each element of the array. Define the difference vector **d** as:

   $$\mathbf{d} = \mathbf{x} - \hat{\mu}$$

   This operation should be done element-wise using broadcasting. You will want Julia to perform such operation:

   $$\mathbf{d} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{n-1} \\ x_n \end{bmatrix} - \begin{bmatrix} \hat{\mu} \\ \hat{\mu} \\ \vdots \\ \hat{\mu} \\ \hat{\mu} \end{bmatrix}$$

   Now, square the differences to form the **squared_d** vector:

   $$\textbf{squared\_d} = \begin{bmatrix} (x_1 - \hat{\mu})^2 \\ (x_2 - \hat{\mu})^2 \\ \vdots \\ (x_{n-1} - \hat{\mu})^2 \\ (x_n - \hat{\mu})^2 \end{bmatrix}$$

Remember to use the dot operator "." for broadcasting the square operator in Julia.

3. **Calculate the variance:** Compute the variance by averaging the squared differences (subtracting 1 for degrees of freedom correction):

$$\text{variance} = \frac{\text{sum}(\textbf{squared\_d})}{n-1}$$

where n is the length of the array **x**.

4. **Calculate the standard deviation:** The standard deviation is the square root of the variance:

$$SD = (\text{variance})^{\frac{1}{2}}$$

5. Return the SD value

Example:

```
standard_deviation([1, 2, 3, 4, 5])  # Output: 1.5811388300841898
standard_deviation([5, 10, 15])      # Output: 5.0
standard_deviation(2:7)              # Output: 1.8708286933869707
```

Problem 7: Import the Data, Plot It, and Calculate Correlations

In this exercise, you will import a dataset, visualize the data, and calculate correlations between variables. The dataset is extracted from the Current Population Survey (CPS). The three columns in the file represent consecutively:

1. **Earnings:** The total annual earnings (first column).
2. **Education:** A numerical representation of the level of education obtained by an individual (second column).
3. **Hours Worked:** The number of hours worked per week by an individual (third column).

Your task is to analyze the bivariate relationships between the "earnings" variable (first column) and either the "education" (second column) or "hours worked" (third column) variables.

The dataset is available in the file `dataset.txt`, which contains three columns separated by commas. Follow the steps below to complete this task.

Note: Due to the time constraints, we will not introduce the wonderful `DataFrames.jl` package, which is used to handle data in a more structured way (similar to `R`'s dataframes). Feel free to explore this package on your own. For this exercise, however, you can import the data into a standard matrix.

Steps to follow:

1. **Import the required packages:** Make sure to load the necessary packages for data import, plotting, and statistical calculations. You will need `DelimitedFiles` for reading the dataset, `Plots` for visualization, and `Statistics` to calculate correlations [8].

2. **Load the dataset:** Import the data from the file `dataset.txt`. The file contains three columns, which can be loaded into a matrix. Each row represents an individual, and each column contains the data for earnings, education, and hours worked, respectively [9].

3. **Plot the data:** Create scatter plots to visualize the relationships between (1) "earnings" and "education" and (2)"earnings" and "hours worked". Plot "earnings" on the y-axis and the other variables on the x-axis. Requirements:

   - Label the axis accordingly
   - Title the plots
   - Use green markers for the relationship between "earnings" and "education" and red for the relationship between "earnings" and "hours worked".

4. **Calculate the correlation:** Calculate the correlation between "earnings" and both "education" and "hours worked".

5. **Analyze the results:** After calculating the correlations and visualizing the relationships analyze the relationship, discuss the findings.

[8] Hint: Use the following commands to import the packages:
`using DelimitedFiles, Plots, Statistics`

[9] Hint: You can use the `readdlm` function to load the data into a matrix or array. Example:
`data = readdlm("dataset.csv", ',', Float64)`

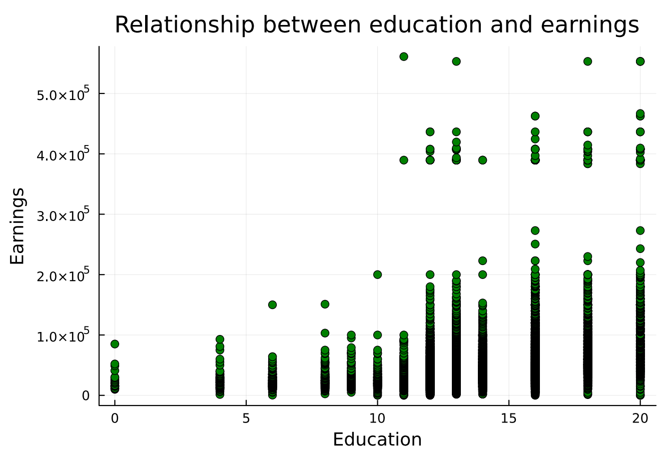

Figure 2: You should produce a scatter plot visualizing the relationship between earnings and earnings which looks like this