Coding Tests

Wenzhi Wang

October 26, 2024

A Github repository (link here) is created to store codes and summary report for the coding tasks.

1. Question 1: Simulations

To present the results in a more simplified and unified way, from questions 1.1. to questions 1.4., I create a Python class that is able to conduct all the procedures required in the task. The complete codes are presented in Appendix Figure A.1, and are stored in the "codes/Question1.py" script file.

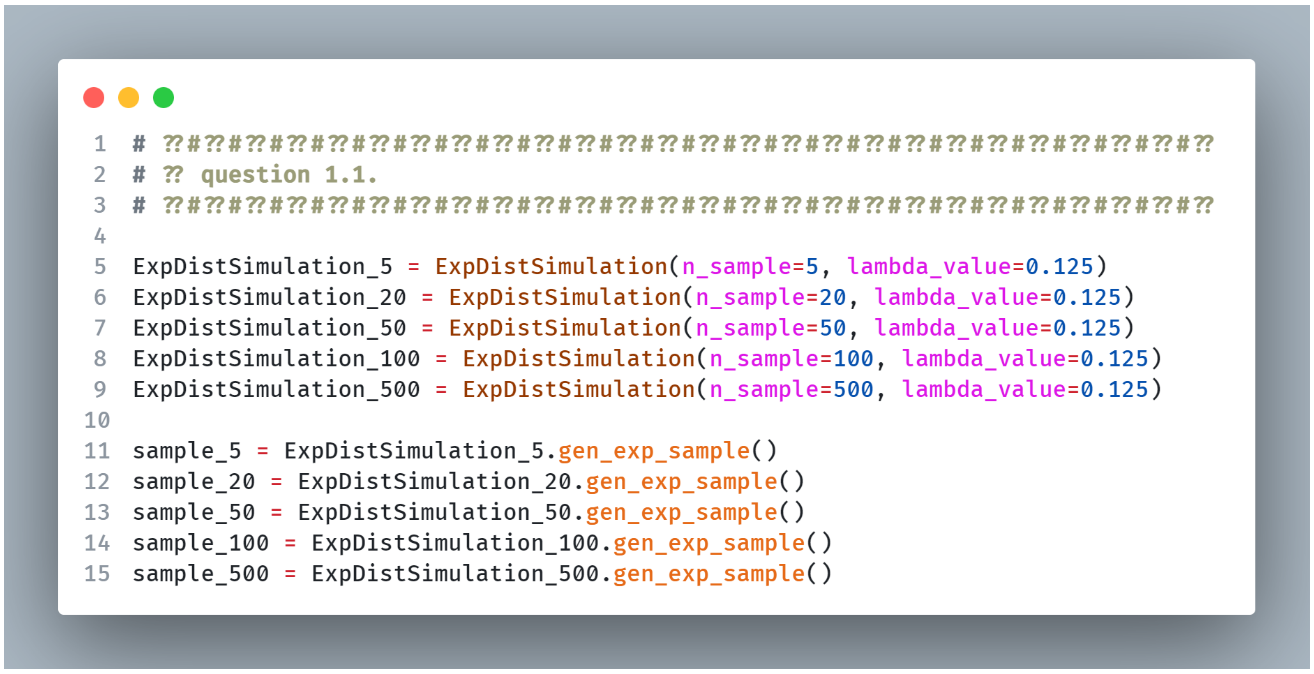
The `ExpDistSimulation` class has three key attributes: `lambda_value`, `n_sample`, and `n_simulation`, which represents the distribution parameter of the exponential distribution, the sample size in each simulation, and the number of simulations, respectively. It has the following key methods and perform procedures required in the coding tasks:

- `gen_exp_sample` generates a random draw from an exponential distribution (with default randomization seed 1234).
- `cal_sample_mean` calculates the (unadjusted) sample mean from any random draw.
- `adj_sample_mean` calculates the adjusted sample mean from any random draw.
- `simulation` conducts simulation, and for each simulated sample, it calculates the adjusted sample mean and stores it.
- `simulation_plot` plots the empirical distribution of the simulated adjusted sample mean.

1.1. Question 1.1.

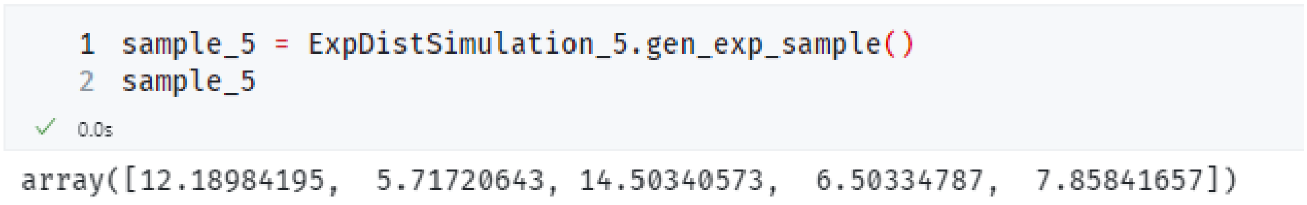
First, I initialize different object instances with different sample size parameters. Then, I generate a random sample for each sample size. The codes are presented in Figure 1, with one example output shown in Figure 2 (*I don't show all the random sample, as they actually don't matter so much for subsequent key results but can greatly reduce the readability of this report.*).

Figure 1: Codes for Question 1.1.



```python
# ??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??
# ?? question 1.1.
# ??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??

ExpDistSimulation_5 = ExpDistSimulation(n_sample=5, lambda_value=0.125)
ExpDistSimulation_20 = ExpDistSimulation(n_sample=20, lambda_value=0.125)
ExpDistSimulation_50 = ExpDistSimulation(n_sample=50, lambda_value=0.125)
ExpDistSimulation_100 = ExpDistSimulation(n_sample=100, lambda_value=0.125)
ExpDistSimulation_500 = ExpDistSimulation(n_sample=500, lambda_value=0.125)

sample_5 = ExpDistSimulation_5.gen_exp_sample()
sample_20 = ExpDistSimulation_20.gen_exp_sample()
sample_50 = ExpDistSimulation_50.gen_exp_sample()
sample_100 = ExpDistSimulation_100.gen_exp_sample()
sample_500 = ExpDistSimulation_500.gen_exp_sample()
```

Figure 2: Output for Question 1.1.



```python
sample_5 = ExpDistSimulation_5.gen_exp_sample()
sample_5
```

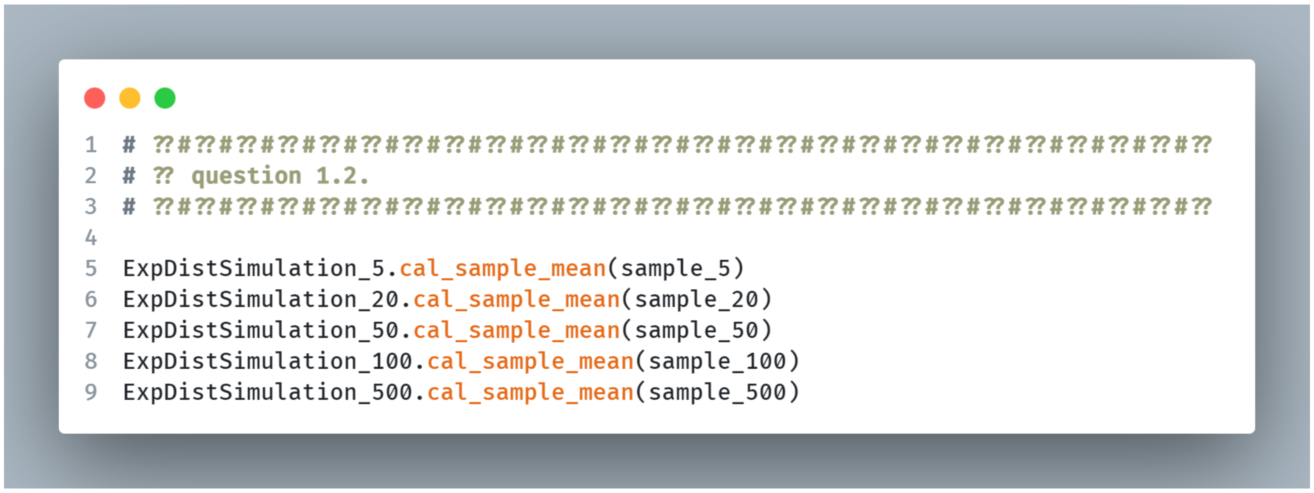
✓ 0.0s

```
array([12.18984195,  5.71720643, 14.50340573,  6.50334787,  7.85841657])
```

1.2. Question 1.2.

The codes for answering this question are in Figure 3. Again, for the $n = 5$ case, the raw sample mean is 9.354443709511397.

Figure 3: Codes for Question 1.2.

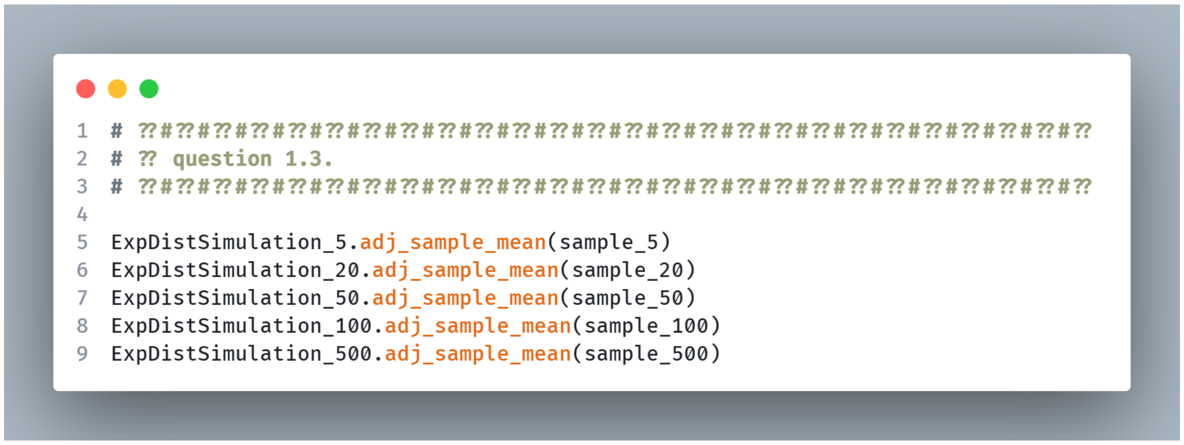


```
# ??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??
# ?? question 1.2.
# ??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??

ExpDistSimulation_5.cal_sample_mean(sample_5)
ExpDistSimulation_20.cal_sample_mean(sample_20)
ExpDistSimulation_50.cal_sample_mean(sample_50)
ExpDistSimulation_100.cal_sample_mean(sample_100)
ExpDistSimulation_500.cal_sample_mean(sample_500)
```

1.3. Question 1.3.

The codes for answering this question are in Figure 4. Again, for the $n = 5$ case, the adjusted sample mean is 3.0286282061644623.

Figure 4: Codes for Question 1.3.



```
# ??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??
# ?? question 1.3.
# ??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??

ExpDistSimulation_5.adj_sample_mean(sample_5)
ExpDistSimulation_20.adj_sample_mean(sample_20)
ExpDistSimulation_50.adj_sample_mean(sample_50)
ExpDistSimulation_100.adj_sample_mean(sample_100)
ExpDistSimulation_500.adj_sample_mean(sample_500)
```

1.4. Question 1.4.

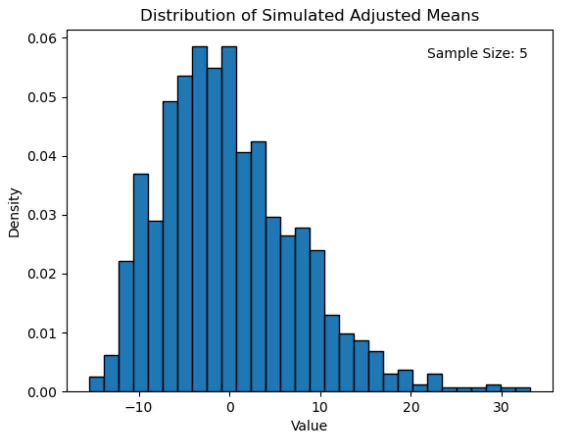
Next, I repeat the procedures in question 1.1.-1.3. for 1000 times, and plots the empirical distribution of the simulated adjusted means.

The results are in Figure 5.

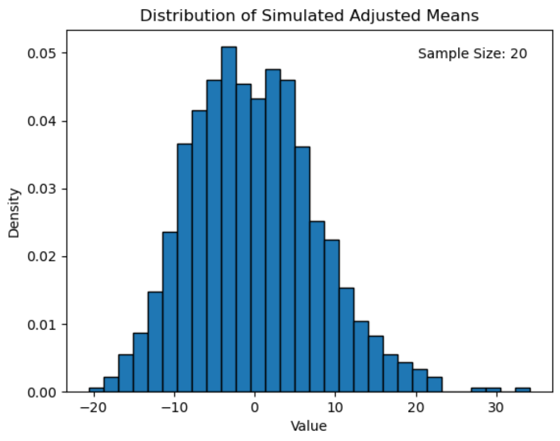
Comments: First, we should expect these adjusted means across different simulated samples should have a normal distribution if the sample size in each simulation sample goes to infinity, as the Central Limit Theorem suggests. It is clear that the empirical distributions become more similar to a normal distribution, as n, the sample size in each simulation increases. For example, the $n = 5$ case is obviously left-skewed and have a fat right tail, but these properties almost vanish for the $n = 500$ case. This is because the Central Limit Theorem states the asymptotic properties of our statistic of interest, and the asymptotic distribution holds only if we draw enough sample points from the distribution.

Figure 5: Distribution of the simulated adjusted means with different sample sizes

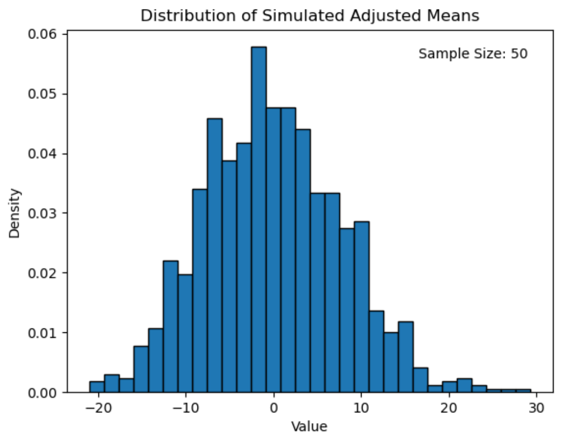
(a) $n = 5$



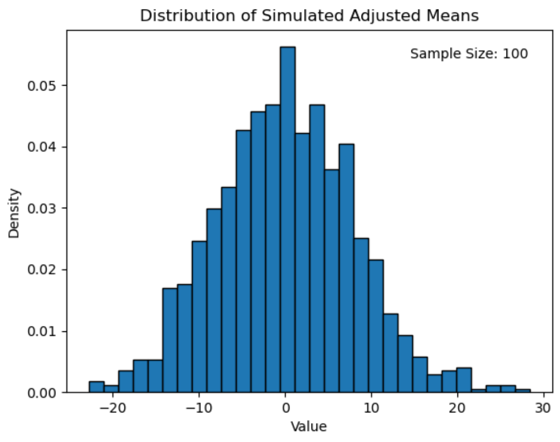
(b) $n = 20$



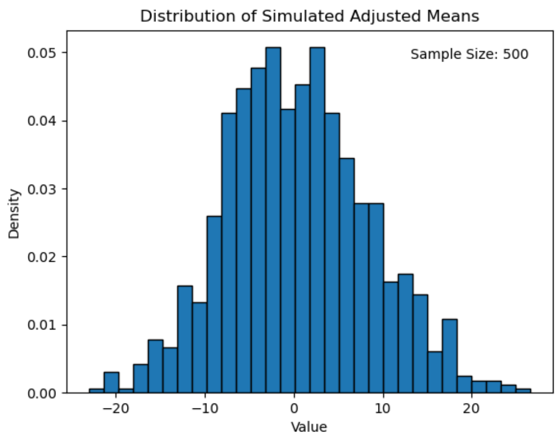
(c) $n = 50$



(d) $n = 100$



(e) $n = 500$



1.5. Question 1.5.

I guess this question is about MLE, thus the moment condition shouldn't be

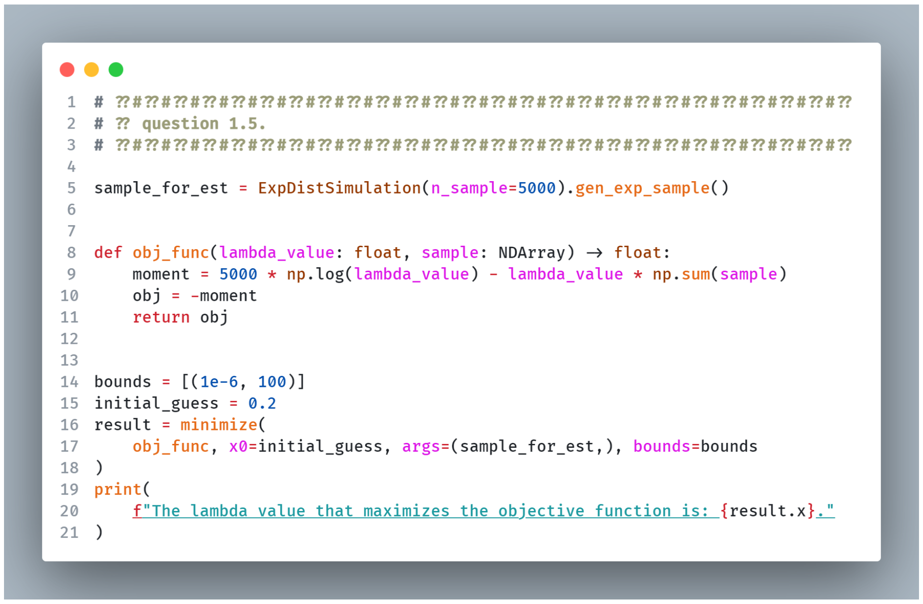
$$5000 \cdot \log(\lambda) - \lambda \sum_{i=1}^{5000} x_i = 0.$$

I use the following optimization problem for estimating the λ:

$$\max_{\lambda} \left[5000 \cdot \log(\lambda) - \lambda \sum_{i=1}^{5000} x_i \right].$$

In particular, I first create a sample with 5000 random draws, then define the objective function (which is the negative log-likelihood, since I use a minimization routine), as shown in Figure 6.

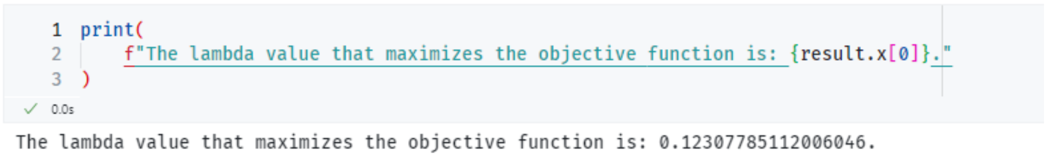
Figure 6: Codes for Question 1.5.



```python
# ??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??
# ?? question 1.5.
# ??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??#??

sample_for_est = ExpDistSimulation(n_sample=5000).gen_exp_sample()


def obj_func(lambda_value: float, sample: NDArray) → float:
    moment = 5000 * np.log(lambda_value) - lambda_value * np.sum(sample)
    obj = -moment
    return obj


bounds = [(1e-6, 100)]
initial_guess = 0.2
result = minimize(
    obj_func, x0=initial_guess, args=(sample_for_est,), bounds=bounds
)
print(
    f"The lambda value that maximizes the objective function is: {result.x}."
)
```

The results (togher with the codes screenshot) are in Figure 7. The estimated λ, $\widehat{\lambda}$ is 0.123.

Figure 7: Answer for Question 1.5.



```python
print(
    f"The lambda value that maximizes the objective function is: {result.x[0]}."
)
```

```
0.0s
The lambda value that maximizes the objective function is: 0.12307785112006046.
```

1.6. Question 1.6.

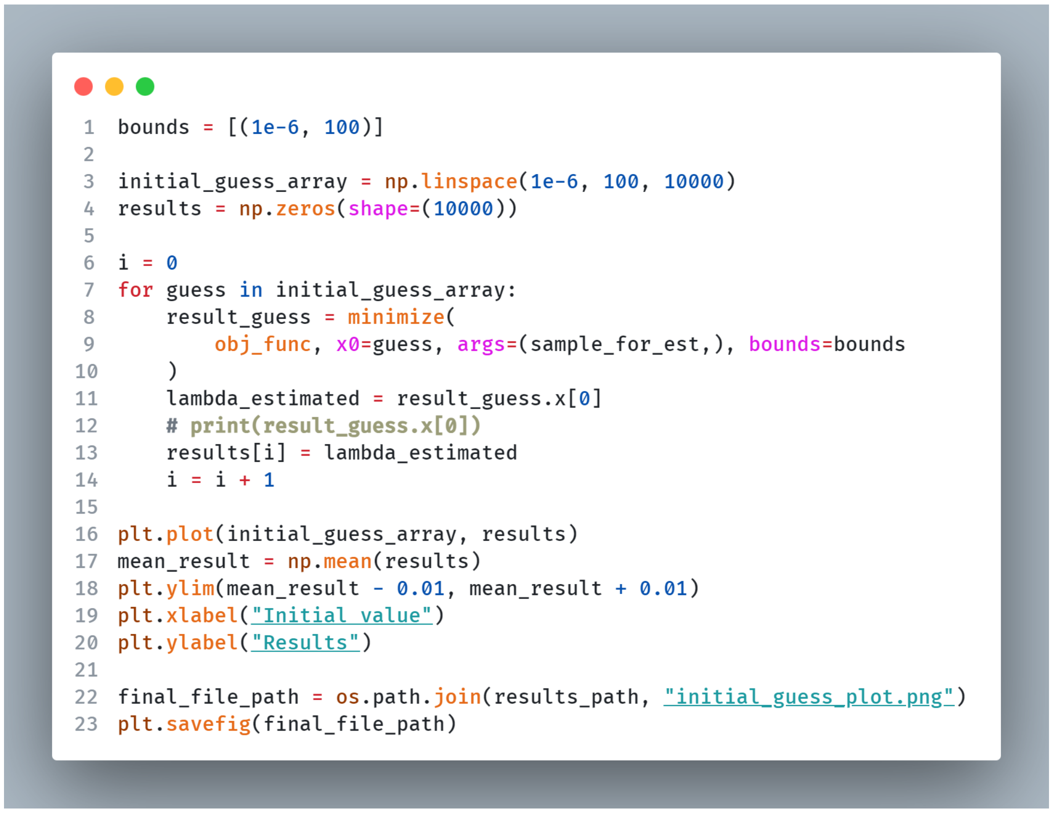
When restricting the search over intervals $[0.5, 1]$ and $[0.2, 1]$, the optimizer value is 0.5, and 0.2, respectively. These boundary values suggest that the maximization solution in these two cases fail to reach the true maximization (which can be calculated by the first-order condition). Therefore, we need to extend these narrow bound conditions when conducting the numerical maximization.

But when the restriction is $[0.1, 1]$, we can reach the true optimizer: 0.123, since we have extended our bound condition to a broader interval.

1.7. Question 1.7.

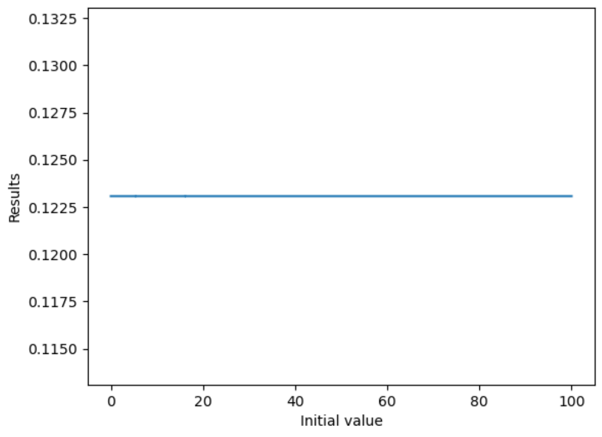
To answer this question, I create a equi-distanced grid over $[10^{-6}, 100]$ as different initial values to start the search, and then for each initial value, I conduct the optimization routine, and stores the results in an array, and plots the results against different starting values. The codes are presented in Figure 8, and the results are in Figure 9.

Figure 8: Codes for Question 1.7.



```python
bounds = [(1e-6, 100)]

initial_guess_array = np.linspace(1e-6, 100, 10000)
results = np.zeros(shape=(10000))

i = 0
for guess in initial_guess_array:
    result_guess = minimize(
        obj_func, x0=guess, args=(sample_for_est,), bounds=bounds
    )
    lambda_estimated = result_guess.x[0]
    # print(result_guess.x[0])
    results[i] = lambda_estimated
    i = i + 1

plt.plot(initial_guess_array, results)
mean_result = np.mean(results)
plt.ylim(mean_result - 0.01, mean_result + 0.01)
plt.xlabel("Initial value")
plt.ylabel("Results")

final_file_path = os.path.join(results_path, "initial_guess_plot.png")
plt.savefig(final_file_path)
```

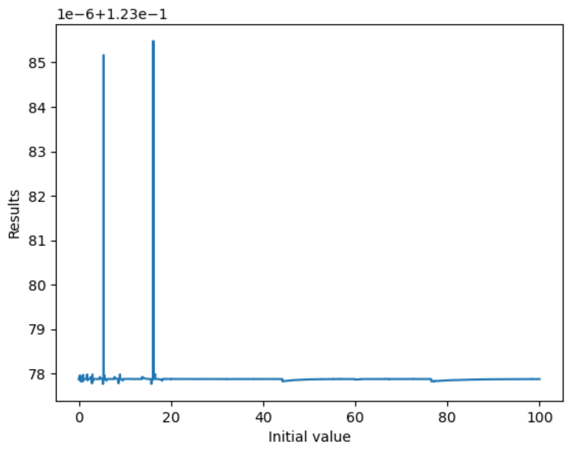
Figure 9: Answer for Question 1.7.



It is easy to see that the starting guess values doesn't matter for this MLE problem.

To see the tiny difference between different starting points, we can rescale the y-axis and have the following graph - Figure 10. Notice that the scale of the y-axis is 10^{-6}!

Figure 10: Answer for Question 1.7. - Rescaling the y-axis



2. Estimation

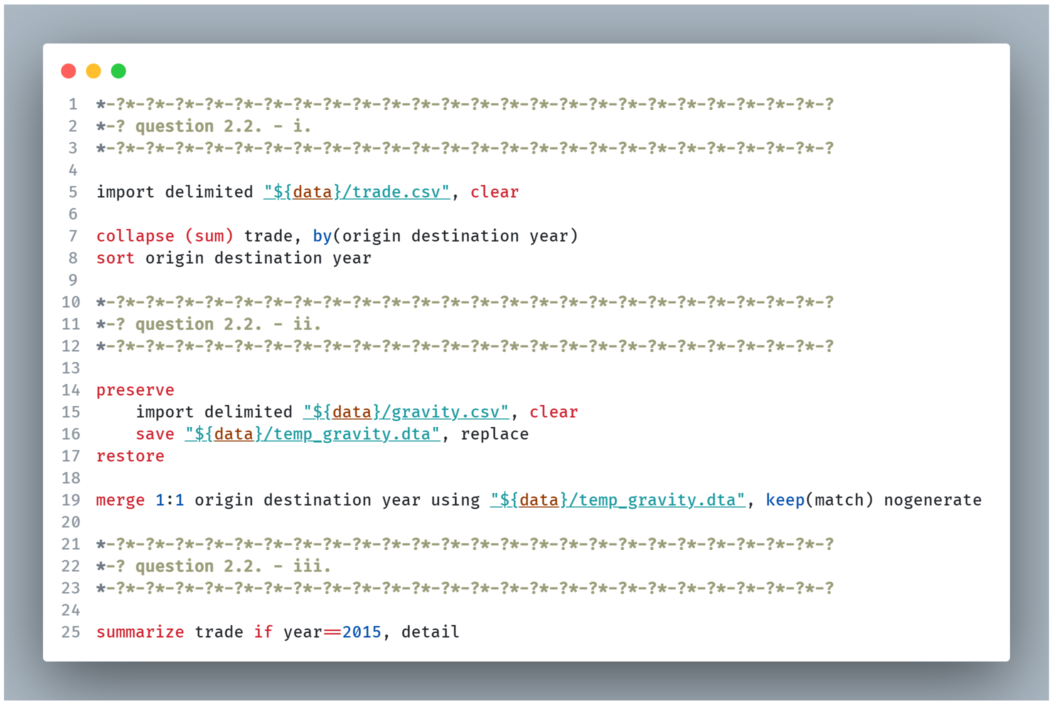
Appendix Figure A.2 stores some Stata commands for initial setup, such as codes for defining several path macros, and specifying the version. The corresponding do file is “codes/Question2.do”.

2.2. Dataset Construction

The Stata codes for this question are in Figure 11. The required moments are

- Mean: 311065.
- Standard deviation: 3193502.
- P10: 2.597.
- P50: 1161.257.
- P90: 2346.26.
- Number of observations: 28803.

Figure 11: Codes for Question 2.2.



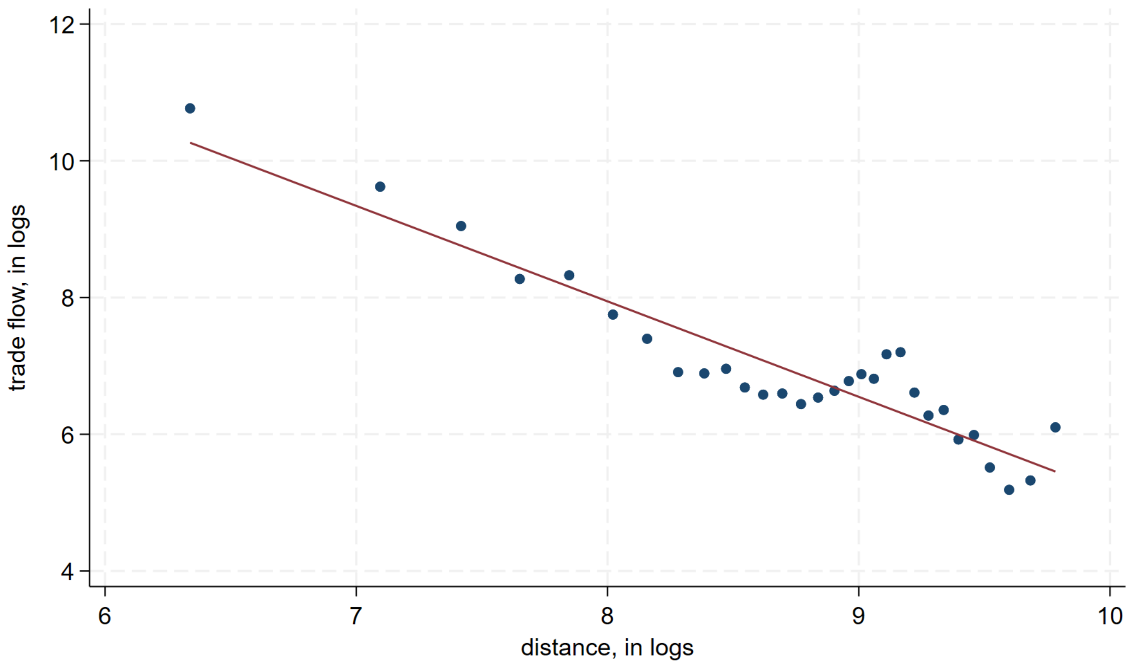
```
*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?
*-? question 2.2. - i.
*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?

import delimited "${data}/trade.csv", clear

collapse (sum) trade, by(origin destination year)
sort origin destination year

*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?
*-? question 2.2. - ii.
*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?

preserve
    import delimited "${data}/gravity.csv", clear
    save "${data}/temp_gravity.dta", replace
restore

merge 1:1 origin destination year using "${data}/temp_gravity.dta", keep(match) nogenerate

*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?
*-? question 2.2. - iii.
*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?*-?

summarize trade if year==2015, detail
```

2.3. Estimation

2.3.1. The Binscatter Plot

The binscatter plot is in Figure 12. The raw correlation between log(trade flows) and log(distance) is -0.2608. The linear fit (with a constant) of the regression coefficient on log(distance) is -1.396478.

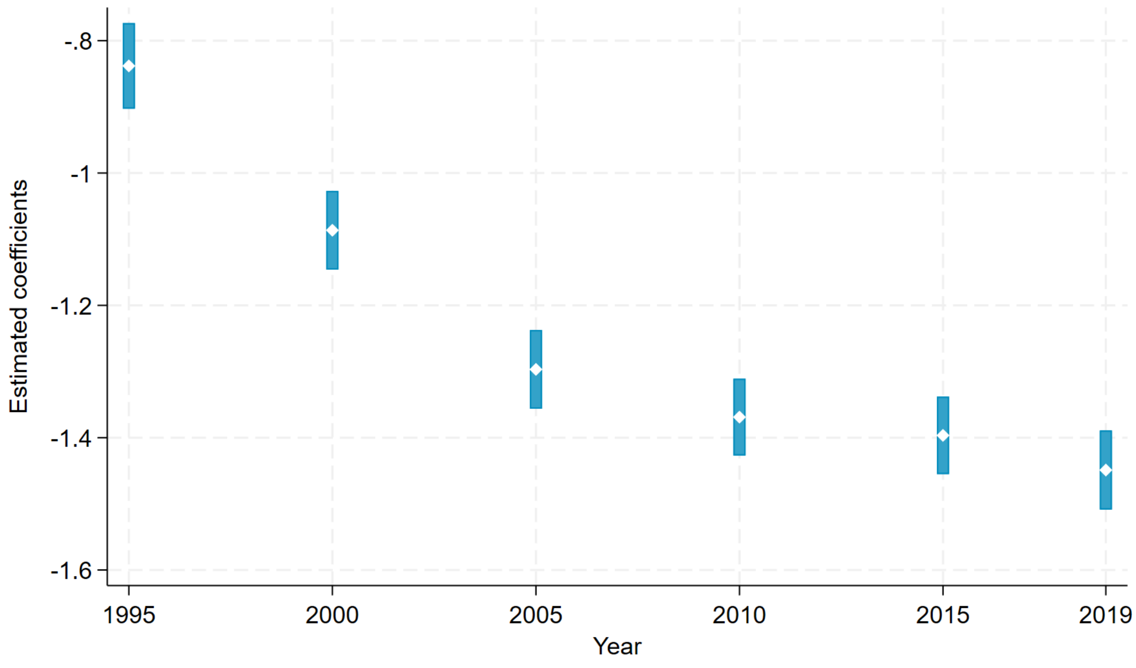
Figure 12: Binscatter plot of trade flows against distance



2.3.2. Distance Coefficients Across Years

The results are in Figure 13. In the graph, I report the estimated coefficients using sample from different years, and the 95% confidence intervals, which are calculated based on clustered standard errors (two-way clustering on both the origin and the destination countries).

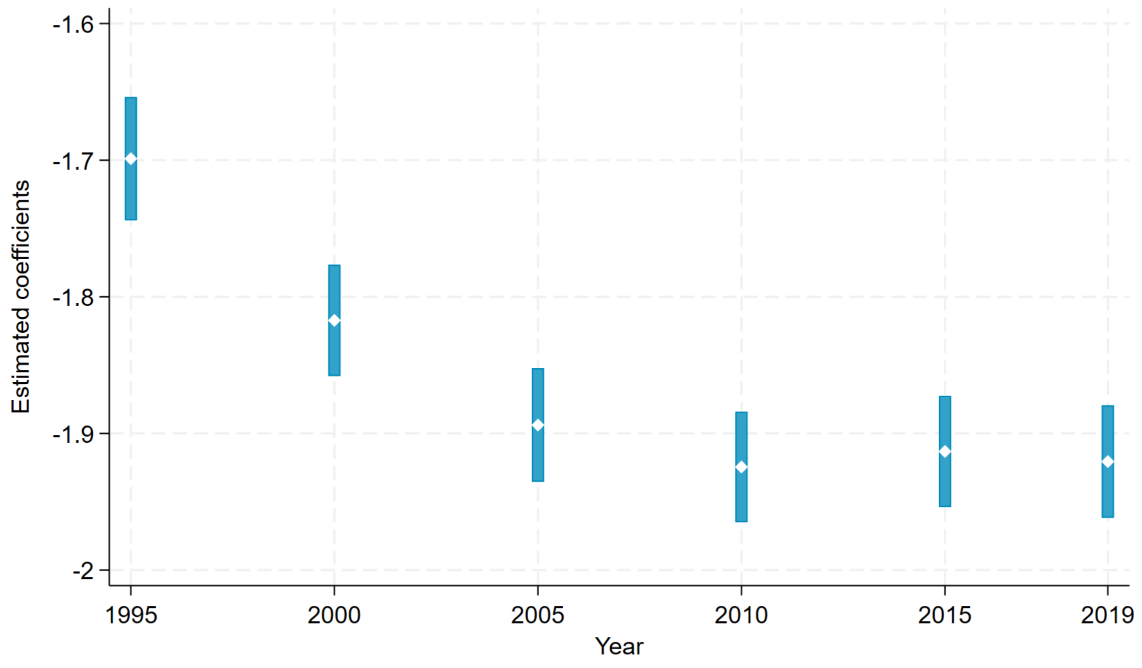
Figure 13: Coefficients on log(distance) for different years



2.3.3. Distance Coefficients Across Years With Importer and Exporter FEs

The results are in Figure 14, where the estimated coefficients using sample from different years, and the 95% confidence intervals, which are calculated based on robust standard errors are reported.

Figure 14: Coefficients on log(distance) for different years, with importer-year and exporter-year FEs



Discussion:

In Figure 14, where importer-year and exporter-year FEs are controlled, we are essentially reporting how much percentage of trade flows will decrease between two given countries in a given year, if their distance has increased by 1%. Controlling for importer and exported fixed effects allow us to partial out the effects of some country attributes, such as economy size (e.g., as measured by GDP) from the estimated coefficients. However, in Figure 13, where the FEs are not controlled, the variation used to identify the distance coefficient could also come from different importer-exporter pair with different distance.

2.3.4. Adding Other Controls

The required table is Table 1, where the robust standard errors are reported in parentheses.

Table 1

	Trade flows
	(1)
distance, in logs	-1.588***
	(0.02)
sharing a border	1.100***
	(0.10)
sharing a language	1.008***
	(0.05)
having colonial ties	0.558***
	(0.12)
having a trade agreement	0.555***
	(0.04)
R-squared	0.724
Obs	26938

Notes. Sample includes trade flows between different countries in 2015. Importer and exporter fixed effects are controlled. Robust standard error are reported.

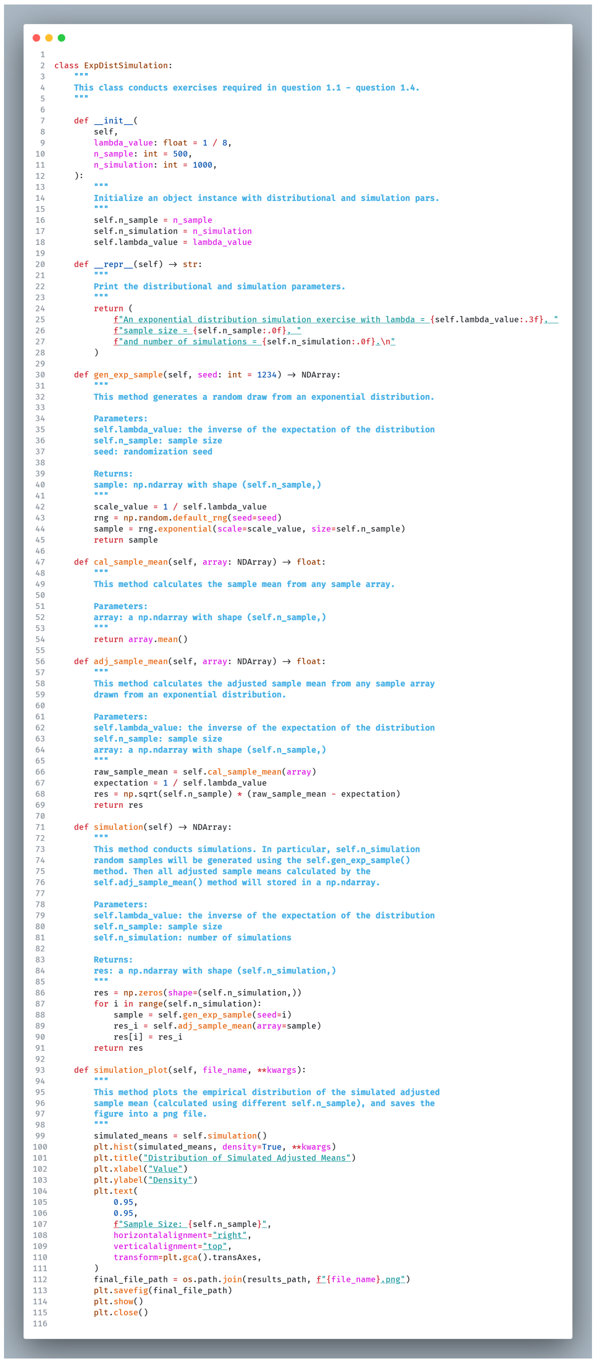
Discussion:

In the table, the coefficient on distance is essentially reporting how much percentage of trade flows will decrease between two given countries who don't share a border, or a language, or colonial ties, and don't have a trade agreement in a given year, if their distance has increased by 1%. Compared with Figure 14, the effect of distance on the trade flows between two countries decreases when we add more controls that represent other notions of "distance" between two countries. The decrease is most likely due to the fact that these controls (especially "sharing a border") are negatively correlated with the geographical distance measure.

All other controls measure the easiness for trades to happen between different countries. In particular, "sharing a border" measures geographical distance, "sharing a language" and "sharing colonial ties" measures some type of cultural linkages, and "having a trade agreement" measures some institutional costs for international trade. For instance, if two countries share the same language, then it is easier for importer firms to communicate with the exporter firms – they are more likely to match each, negotiate with each other and reach a business contract. Therefore, it is intuitive that these factors will contribute positively to trade flows between two countries.

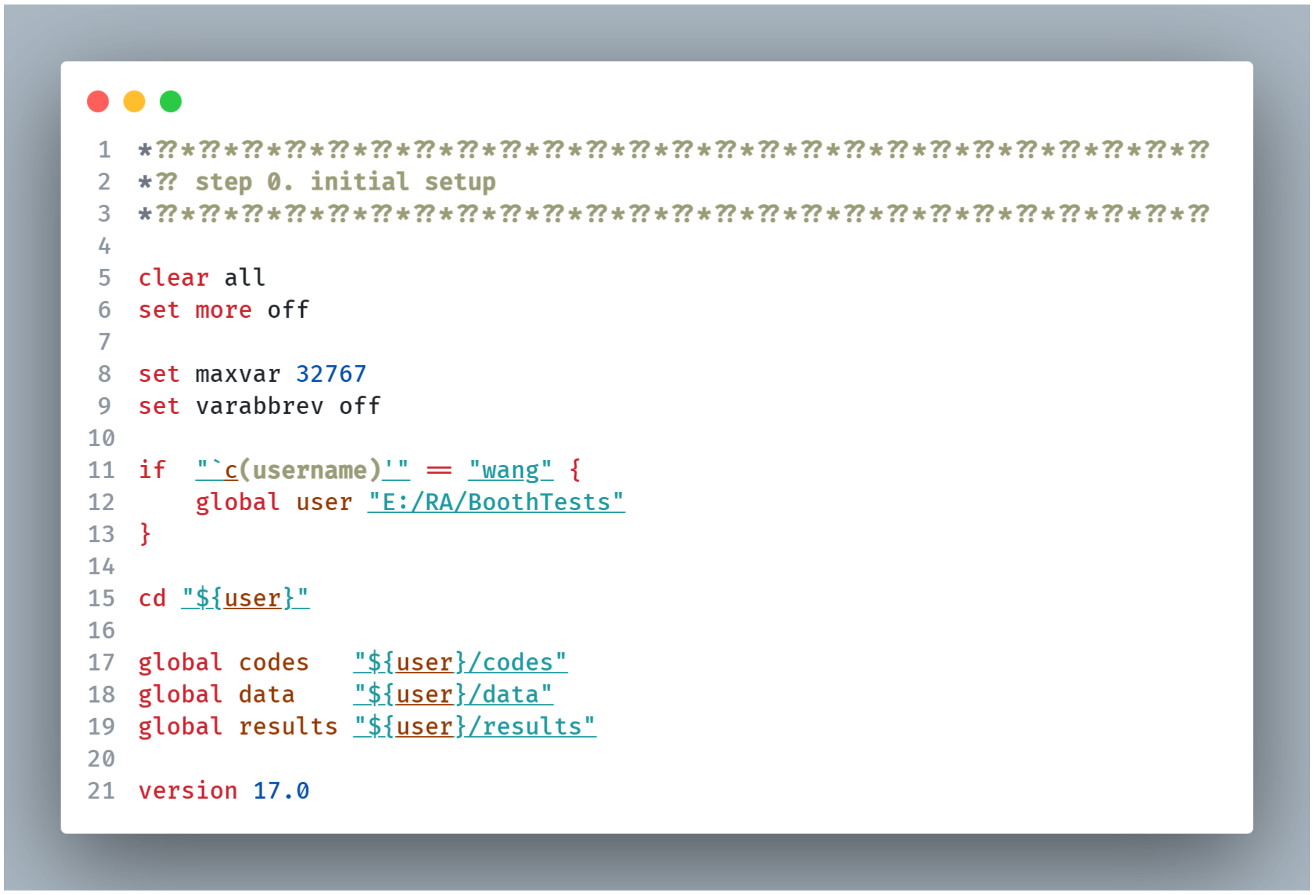
A. Appendix

Figure A.1



```python

class ExpDistSimulation:
    """
    This class conducts exercises required in question 1.1 - question 1.4.
    """

    def __init__(
        self,
        lambda_value: float = 1 / 8,
        n_sample: int = 500,
        n_simulation: int = 1000,
    ):
        """
        Initialize an object instance with distributional and simulation pars.
        """
        self.n_sample = n_sample
        self.n_simulation = n_simulation
        self.lambda_value = lambda_value

    def __repr__(self) → str:
        """
        Print the distributional and simulation parameters.
        """
        return (
            f"An exponential distribution simulation exercise with lambda = {self.lambda_value:.3f}, "
            f"sample size = {self.n_sample:.0f}, "
            f"and number of simulations = {self.n_simulation:.0f}.\n"
        )

    def gen_exp_sample(self, seed: int = 1234) → NDArray:
        """
        This method generates a random draw from an exponential distribution.

        Parameters:
        self.lambda_value: the inverse of the expectation of the distribution
        self.n_sample: sample size
        seed: randomization seed

        Returns:
        sample: np.ndarray with shape (self.n_sample,)
        """
        scale_value = 1 / self.lambda_value
        rng = np.random.default_rng(seed=seed)
        sample = rng.exponential(scale=scale_value, size=self.n_sample)
        return sample

    def cal_sample_mean(self, array: NDArray) → float:
        """
        This method calculates the sample mean from any sample array.

        Parameters:
        array: a np.ndarray with shape (self.n_sample,)
        """
        return array.mean()

    def adj_sample_mean(self, array: NDArray) → float:
        """
        This method calculates the adjusted sample mean from any sample array
        drawn from an exponential distribution.

        Parameters:
        self.lambda_value: the inverse of the expectation of the distribution
        self.n_sample: sample size
        array: a np.ndarray with shape (self.n_sample,)
        """
        raw_sample_mean = self.cal_sample_mean(array)
        expectation = 1 / self.lambda_value
        res = np.sqrt(self.n_sample) * (raw_sample_mean - expectation)
        return res

    def simulation(self) → NDArray:
        """
        This method conducts simulations. In particular, self.n_simulation
        random samples will be generated using the self.gen_exp_sample()
        method. Then all adjusted sample means calculated by the
        self.adj_sample_mean() method will stored in a np.ndarray.

        Parameters:
        self.lambda_value: the inverse of the expectation of the distribution
        self.n_sample: sample size
        self.n_simulation: number of simulations

        Returns:
        res: a np.ndarray with shape (self.n_simulation,)
        """
        res = np.zeros(shape=(self.n_simulation,))
        for i in range(self.n_simulation):
            sample = self.gen_exp_sample(seed=i)
            res_i = self.adj_sample_mean(array=sample)
            res[i] = res_i
        return res

    def simulation_plot(self, file_name, **kwargs):
        """
        This method plots the empirical distribution of the simulated adjusted
        sample mean (calculated using different self.n_sample), and saves the
        figure into a png file.
        """
        simulated_means = self.simulation()
        plt.hist(simulated_means, density=True, **kwargs)
        plt.title("Distribution of Simulated Adjusted Means")
        plt.xlabel("Value")
        plt.ylabel("Density")
        plt.text(
            0.95,
            0.95,
            f"Sample Size: {self.n_sample}",
            horizontalalignment="right",
            verticalalignment="top",
            transform=plt.gca().transAxes,
        )
        final_file_path = os.path.join(results_path, f"{file_name}.png")
        plt.savefig(final_file_path)
        plt.show()
        plt.close()

```

Figure A.2



```
*⁇*⁇*⁇*⁇*⁇*⁇*⁇*⁇*⁇*⁇*⁇*⁇*⁇*⁇*⁇*⁇*⁇*⁇*⁇*⁇*⁇*⁇*⁇*⁇*⁇
*⁇ step 0. initial setup
*⁇*⁇*⁇*⁇*⁇*⁇*⁇*⁇*⁇*⁇*⁇*⁇*⁇*⁇*⁇*⁇*⁇*⁇*⁇*⁇*⁇*⁇*⁇*⁇*⁇

clear all
set more off

set maxvar 32767
set varabbrev off

if  "`c(username)'" == "wang" {
    global user "E:/RA/BoothTests"
}

cd "${user}"

global codes   "${user}/codes"
global data    "${user}/data"
global results "${user}/results"

version 17.0
```