

CSE 215

Operation Research and Optimization Techniques

Lab 3

Team Members

---

Name	ID
بهاء خالد محمد علي إبراهيم	21010383
بولا هاني فايز جرجس	21010387
عبد الله محمد علي زين العابدين	21011644
علي محمود صبحي أحمد الجيار	21010842
عمر الضوي إبراهيم الضوي	21010864

# Simulation Report: Tandem Queue System

---

## 1. Introduction

This report presents the results of simulating a tandem queue system with two single servers. The goal of the simulation is to estimate the time average number of customers in the system under various conditions and to compare these results with steady-state predictions.

## 2. Methodology

### 2.1 Simulation Parameters

- Arrival Process: Poisson with rate  $\lambda$
- Service Times: Exponential with rates  $\mu_1$  for server 1 and  $\mu_2$  for server 2
- Initial Queue Length:  $q$  customers in the first queue
- Time Horizon:  $T$
- Number of Simulations:  $N$

### 2.2 Parameter Values

- $\lambda$ : {1, 5}
- $\mu_1$ : {2, 4}
- $\mu_2$ : {3, 4}
- $T$ : {10, 50, 100, 1000}
- $q$ : {0, 1000}

### 2.3 Simulation Procedure

1. Generate interarrival and service times according to their prescribed distributions.
2. Simulate the queue length process and compute the corresponding integral.
3. Perform the simulations  $N$  times for each combination of parameters.
4. Report the average number of customers in the system.
5. Compare the simulation results with steady-state predictions.

## 3. Results and comparison of variation parameters values

For Average Customers (Theoretical):

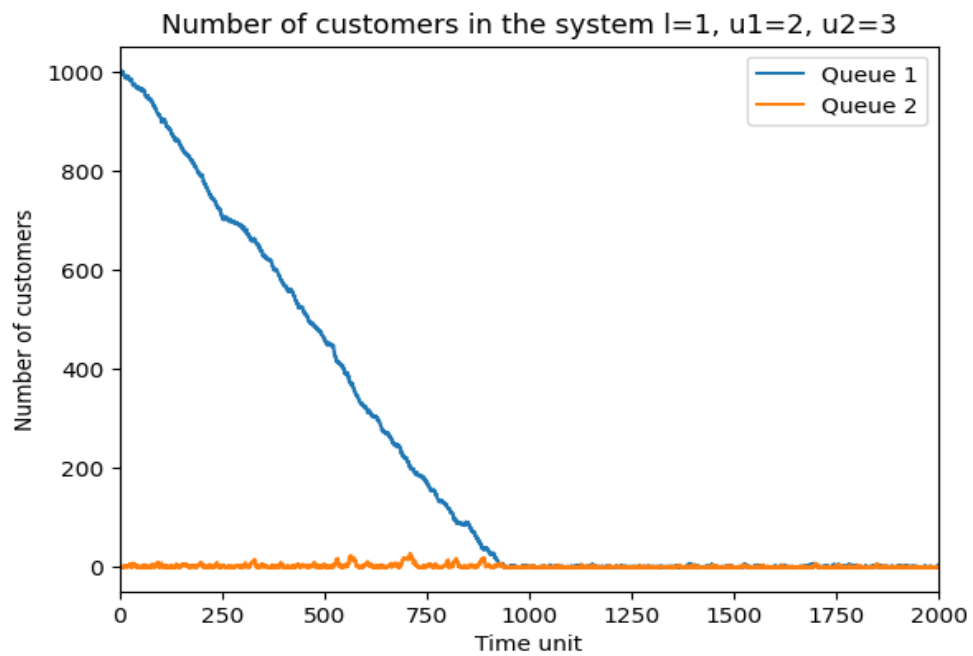
$$E[L_1 + L_2] = \frac{\rho_1}{1-\rho_1} + \frac{\rho_2}{1-\rho_2}$$

$$\text{Such } \rho_1 = \frac{\lambda}{\mu_1}, \rho_2 = \frac{\lambda}{\mu_2}$$

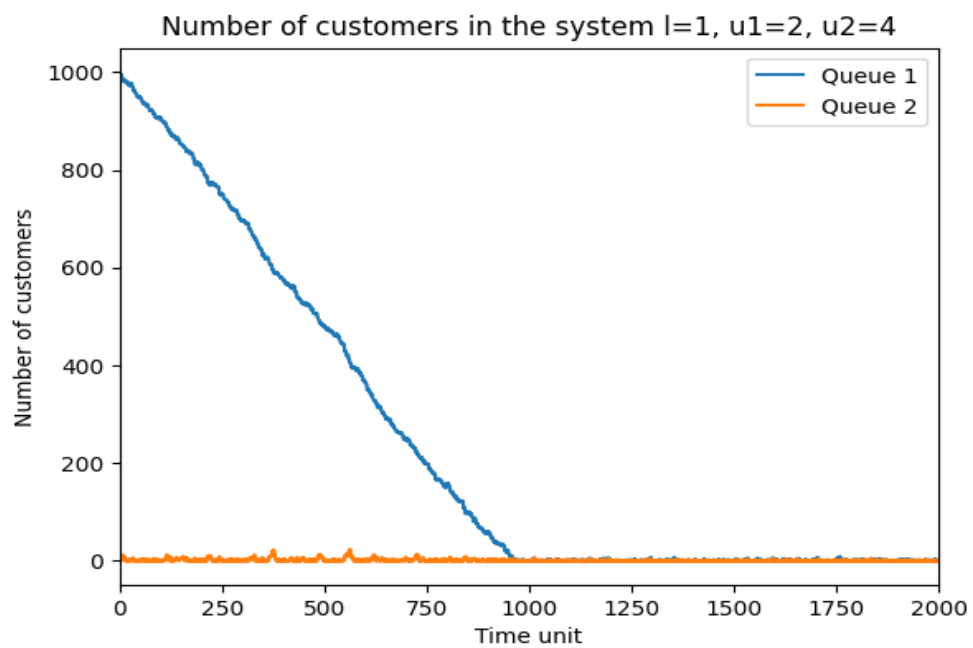
q	$\lambda$	$\mu_1$	$\mu_2$	T	Average Customers (Simulation)	Average Customers (Theoretical)
0	1	2	3	10	1.486935464	1.5
0	1	2	3	50	1.641040923	1.5
0	1	2	3	100	1.634509182	1.5
0	1	2	3	1000	1.668024378	1.5
0	1	2	4	10	1.294854253	1.33
0	1	2	4	50	1.447186272	1.33
0	1	2	4	100	1.529164905	1.33
0	1	2	4	1000	1.52118838	1.33
0	1	4	3	10	0.977539346	0.83
0	1	4	3	50	0.996007964	0.83
0	1	4	3	100	1.024060104	0.83
0	1	4	3	1000	1.022531944	0.83
0	1	4	4	10	0.857905588	0.67
0	1	4	4	50	0.87173762	0.67
0	1	4	4	100	0.887511444	0.67
0	1	4	4	1000	0.891812213	0.67
0	5	2	3	10	17.90671732	Unstable
0	5	2	3	50	78.82434118	Unstable
0	5	2	3	100	153.3930292	Unstable
0	5	2	3	1000	1504.382831	Unstable
0	5	2	4	10	16.99824381	Unstable
0	5	2	4	50	76.76154689	Unstable
0	5	2	4	100	151.5823561	Unstable
0	5	2	4	1000	1503.021321	Unstable
0	5	4	3	10	12.40162593	Unstable
0	5	4	3	50	55.07545208	Unstable
0	5	4	3	100	102.6618931	Unstable
0	5	4	3	1000	1005.107941	Unstable
0	5	4	4	10	11.16628622	Unstable
0	5	4	4	50	39.69233543	Unstable
0	5	4	4	100	72.20666508	Unstable
0	5	4	4	1000	563.551261	Unstable

Q	$\lambda$	$\mu_1$	$\mu_2$	T	Queue 1 Final Size	Queue 2 Final Size
1000	1	2	3	2000	4	1
1000	1	2	4	2000	1	0
1000	1	4	3	2000	1	0
1000	1	4	4	2000	1	0
1000	5	2	3	2000	6787	0
1000	5	2	4	2000	6988	1
1000	5	4	3	2000	3086	2101
1000	5	4	4	2000	3080	25

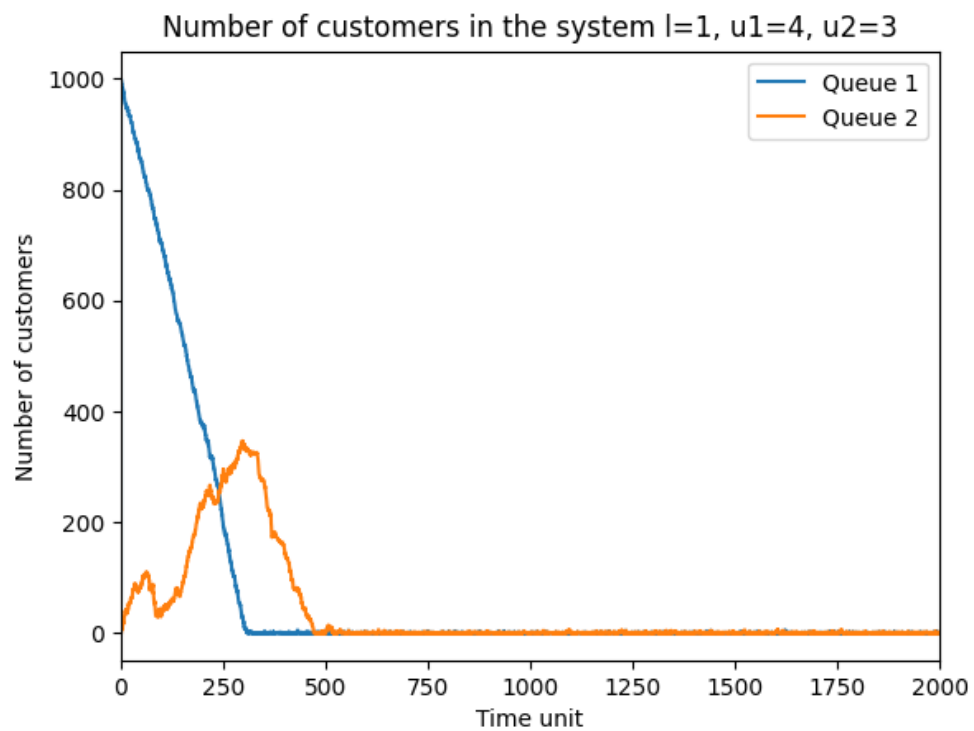
#### 4. Plots and comparisons



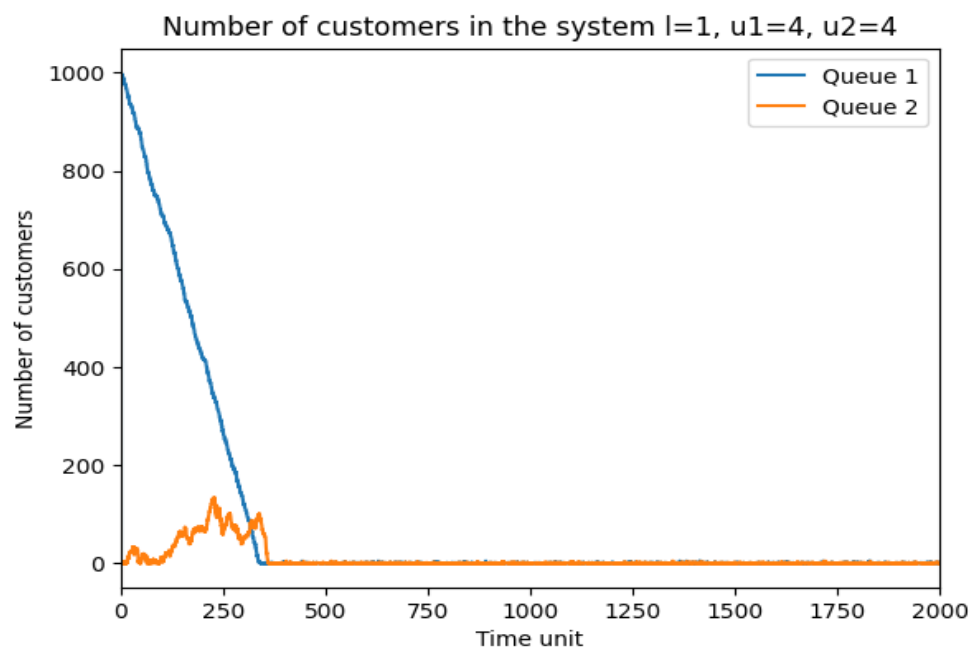
Stable System.



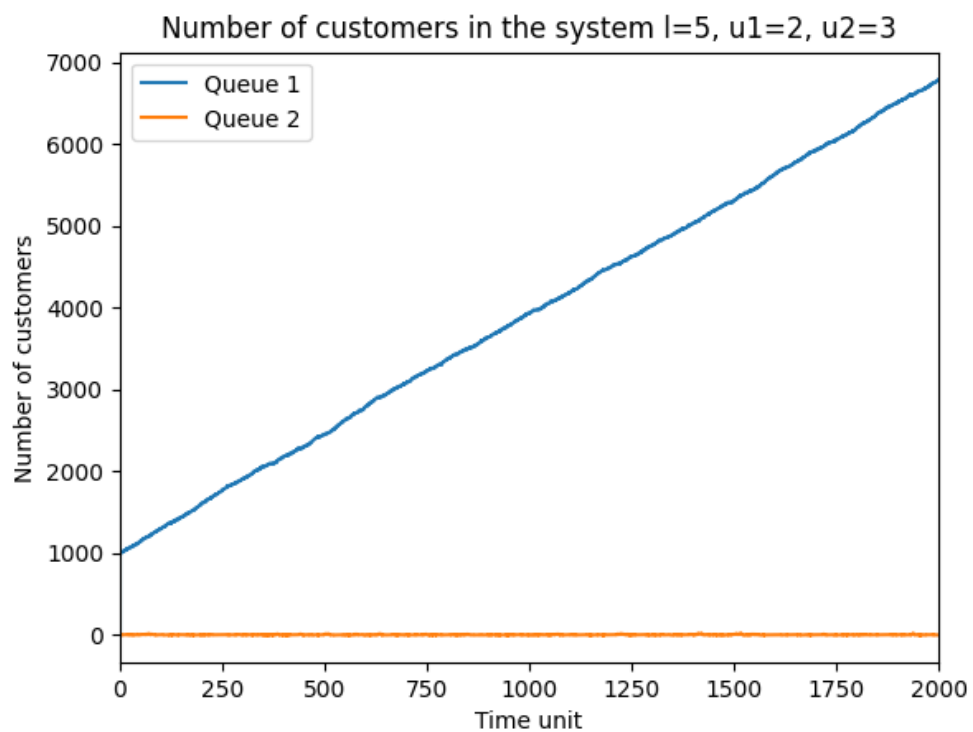
Stable System.



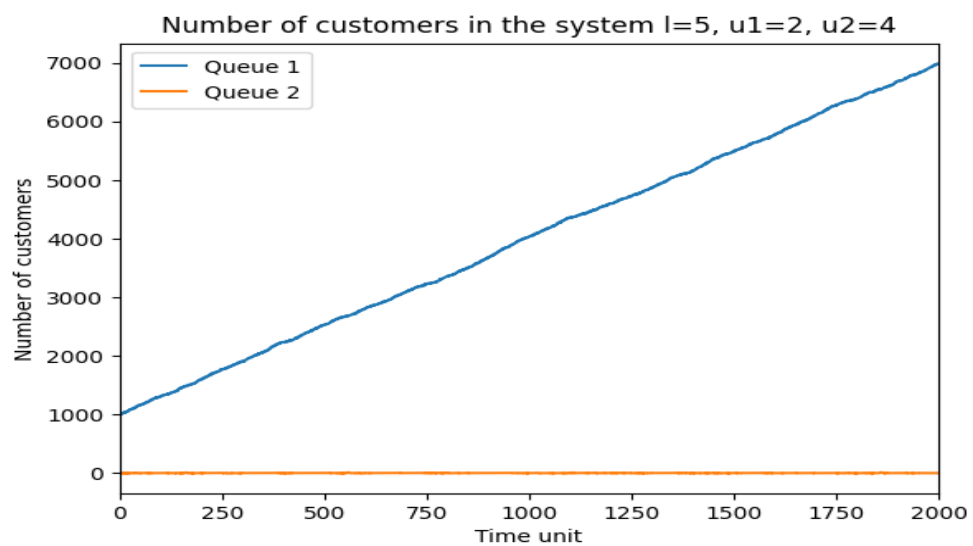
Stable System.



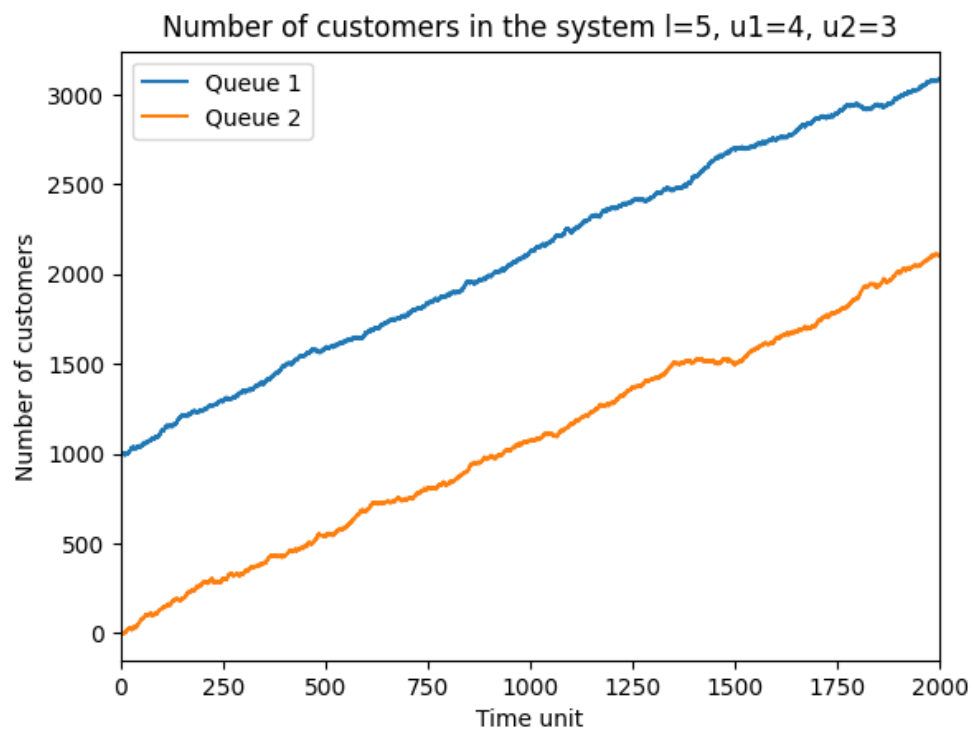
Stable System.



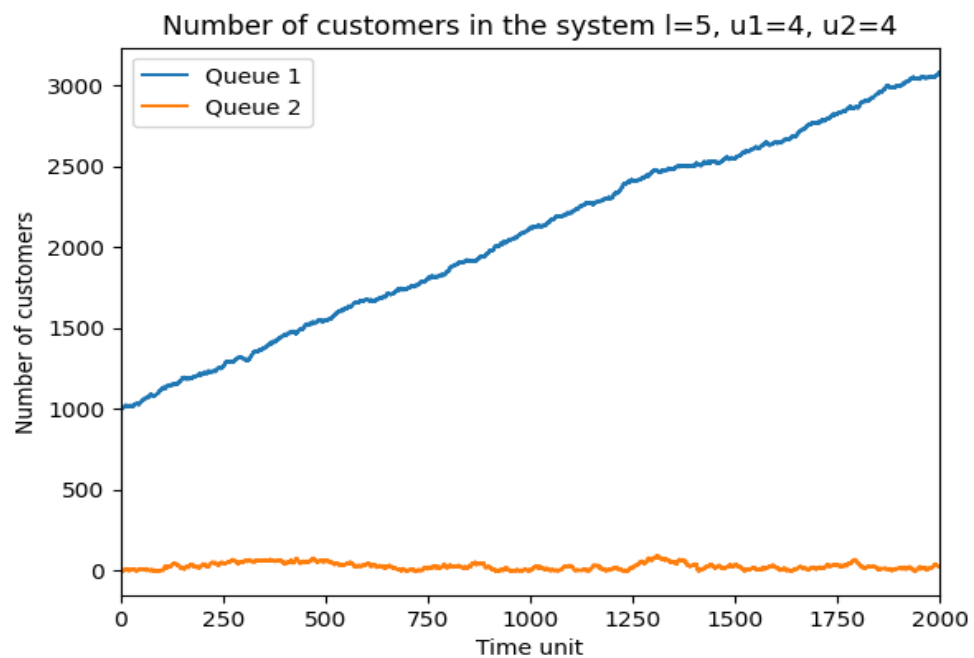
Unstable System.



Unstable System.



Unstable System.



Unstable System.



## 5. Code Explanation

### 5.1. Generating data

```
def __generate_arrival_time(self, rate: float, interval: int) → list[float]:
    time = []
    current_time = 0.0
    while current_time < interval:
        current_time += np.random.exponential(1.0 / rate)
        time.append(current_time)
    return time

2 usages

def __generate_service_time(self, rate: float, size: int) → list[float]:
    time = []
    for _ in range(size):
        time.append(np.random.exponential(1.0 / rate))
    return time
```

For service time the data must follow an Exponential Distribution with lambda equals  $(1 / \text{service rate})$  for each of the two queues.

Although the arrival time follows Poission Distribution we can generate the data as Poission data then sort them ascendingly in order  $O(n \log(n))$  but a better way was used as the intervals were divided to be an Exponential Distribution with increasing order, so the sorting is not needed any more and the order is  $O(n)$ .

## 5.2. Simulation Class

```
class Simulation:
    def __init__(self):
        self.__time = []
        self.__queue_1 = []
        self.__queue_2 = []

    2 usages
    def get_time_data(self):
        return self.__time

    2 usages
    def get_queue_1_data(self):
        return self.__queue_1

    2 usages
    def get_queue_2_data(self):
        return self.__queue_2
```

Just getters to retrieve the data after the end of the simulation process.

```

def simulate(self, arrival_rate: float, service_1_rate: float, service_2_rate: float, interval: int,
            initial_capacity: int):
    current_time = 0.0
    index = 0
    arrival_time = [0.0 for _ in range(initial_capacity)]
    arrival_time = arrival_time + self.__generate_arrival_time(arrival_rate, interval)
    service_time_1 = self.__generate_service_time(service_1_rate, len(arrival_time))
    service_time_2 = self.__generate_service_time(service_2_rate, len(arrival_time))
    queue_1 = []
    queue_2 = []
    queue_1_service_end = sys.maxsize
    queue_2_service_end = sys.maxsize
    while current_time <= interval:
        current_time = min(arrival_time[index], queue_1_service_end, queue_2_service_end)
        # new arrival
        if current_time == arrival_time[index]:
            queue_1.append(index)
            if len(queue_1) == 1:
                queue_1_service_end = current_time + service_time_1[index]
            index += 1

        # queue 1 service end
        elif current_time == queue_1_service_end:
            finished_index = queue_1.pop(0)
            queue_2.append(index)
            if len(queue_2) == 1:
                queue_2_service_end = current_time + service_time_2[finished_index]
            if len(queue_1) > 0:
                queue_1_service_end = current_time + service_time_1[queue_1[0]]
            else:
                queue_1_service_end = sys.maxsize

        # queue 2 service end
        else:
            queue_2.pop(0)
            if len(queue_2) > 0:
                queue_2_service_end = current_time + service_time_2[queue_2[0]]
            else:
                queue_2_service_end = sys.maxsize

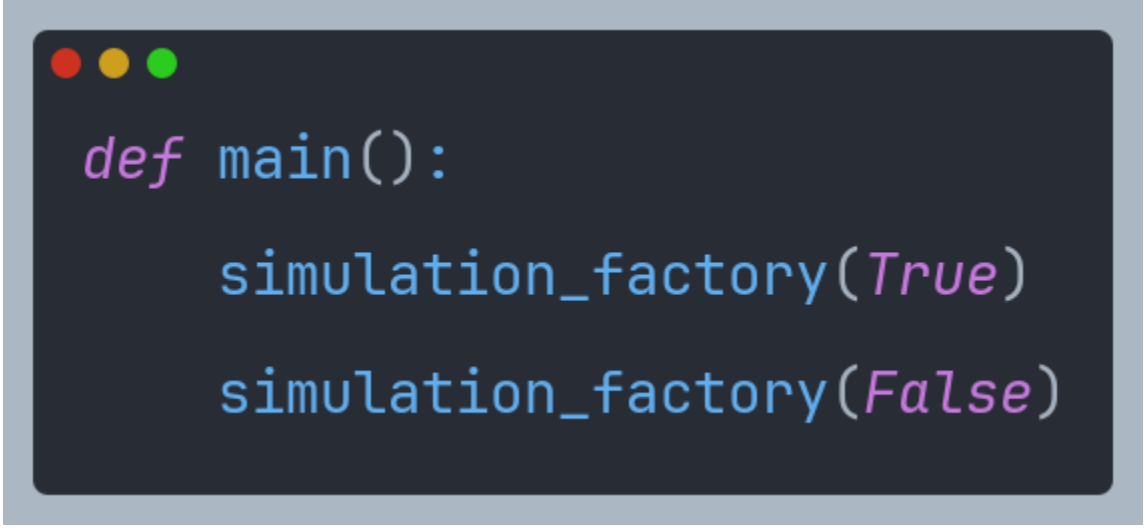
        # store current state
        self.__time.append(current_time)
        self.__queue_1.append(len(queue_1))
        self.__queue_2.append(len(queue_2))

```

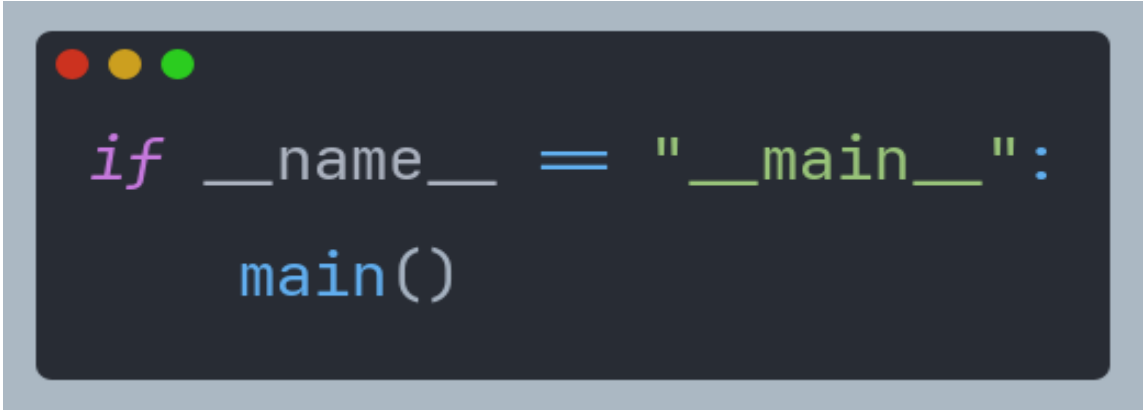
The main logic of the simulation as first the arrival time and service time for every server are generated and the condition of that the first queue may start with people waiting inside is considered, then two empty queues are generated to track the processes inside the system for every moment we care about studying it.

In this process we only care about the nearest event to happen between the three events (New process, A process has finished the first service going to the second one, A process has finished the second service and now leaving the system) so the algorithm tracks the nearest one of them then update and store the system state for each event.

### 5.3. Main Class

A code editor window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. It contains the following Python code:

```
def main():  
    simulation_factory(True)  
    simulation_factory(False)
```

A code editor window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. It contains the following Python code:

```
if __name__ == "__main__":  
    main()
```

Calling the main function to start running the program for first and second lab requirements.

```

def simulation_factory(part_A: bool):
    table = []
    arrival_rate = (1, 5)
    service_1_rate = (2, 4)
    service_2_rate = (3, 4)
    N = 200
    T = (10, 50, 100, 1000)
    part_B_time = 2000
    q0 = 1000

    for l in arrival_rate:
        for u1 in service_1_rate:
            for u2 in service_2_rate:
                if part_A:
                    for t in T:
                        total_time = 0.0
                        for i in range(N):
                            simulation = Simulation()
                            simulation.simulate(l, u1, u2, t, initial_capacity: 0)
                            time_data = np.array(simulation.get_time_data())
                            queue_1_data = np.array(simulation.get_queue_1_data())
                            queue_2_data = np.array(simulation.get_queue_2_data())
                            integral_1 = np.trapz(y=queue_1_data, x=time_data)
                            integral_2 = np.trapz(y=queue_2_data, x=time_data)
                            total_time += ((1 / t) * (integral_1 + integral_2))

                        print(f"For l={l}, u1={u1}, u2={u2}, T={t}, the time average number of customers in the system "
                              f"E[T]={total_time / N:.3f}")
                        table.append([0, l, u1, u2, t, total_time / N])

                    print("-----")
                    print("-----")
                else:
                    simulation = Simulation()
                    simulation.simulate(l, u1, u2, part_B_time, q0)
                    time_data = np.array(simulation.get_time_data())
                    queue_1_data = np.array(simulation.get_queue_1_data())
                    queue_2_data = np.array(simulation.get_queue_2_data())
                    plt.plot(*args: time_data, queue_1_data, label="Queue 1")
                    plt.plot(*args: time_data, queue_2_data, label="Queue 2")
                    plt.xlabel("Time unit")
                    plt.ylabel("Number of customers")
                    plt.title(f"Number of customers in the system l={l}, u1={u1}, u2={u2}")
                    plt.legend()
                    plt.xlim(*args: 0, part_B_time)
                    plt.show()
                    table.append([1000, l, u1, u2, part_B_time, queue_1_data[-1], queue_2_data[-1]])

    if part_A:
        df = pd.DataFrame(table, columns=["q", "l", "u1", "u2", "T", "E[T]"])
        df.to_csv(path_or_buf: "part_a.csv", index=False)
    else:
        df = pd.DataFrame(table, columns=["q", "l", "u1", "u2", "T", "Queue 1 final size", "Queue 2 final size"])
        df.to_csv(path_or_buf: "part_b.csv", index=False)

```

Here the code generates all the combinations of arrival rates, service rates for first and second servers and the interval of time to study.

For the first requirement we run the simulation 200 times to get a better average of the values and we used the trapezoidal rule to get the area under each curve of the two curves of each queue to calculate the average processes inside the system, although the rule gives an approximated values the results of the data were acceptable and close to the analytical values calculated.

For the second requirement we ran the program once and drew the state of the system at every event point.

At the end we save the data of the simulation in a .csv file to use them if needed.