Traffic Engineering

1st Lab Project Report

Simulation of Poisson Distributions and M/M/1 queues

Group 8
Francisco Bento - 93581
Pedro Tracana - 93610

May 5, 2024

1 Introduction

The aim of this report is to explore and evaluate the functioning of Poisson arrival processes and their relevance in queue processing applications. It contains the formulation of algorithms to simulate the designated scenarios, followed by insights into how each algorithm works and what analysis can be drawn from the results achieved. The analysis of findings is complemented by graphical illustrations. Finally, the report concludes by summarizing key takeaways and reflections on the accomplished work.

2 Code and Algorithm Structure

2.1 Arrival Process Generation

To simulate a Poisson process, the first step required is to build a function that generates a pseudo-random variable, δt, for a sequence of N events with λ as parameter. This function then returns a vector containing the respective timestamps of these events, which are determined based on δt and normalized by units of time. This function is denoted as *exponential_distribution(n, l)*.

The next step involves counting in two instances: the first instance allows us to count the frequency of events occurring in a unitary time interval, and the second instance allows us to count the number of times each value of ”number of events” appears in the first counting, creating a dictionary representing the relation {number of events : frequency}.

Since the goal of this exercise is to plot the experimental data against the theoretical Poisson distribution, it is necessary to calculate both percentages for the experimental data and generate the theoretical percentages for the Poisson distribution (represented by the function *poisson_pmf(k,lambda_)*, normalized by the maximum value of events in a unit of time found in the experimental data.

Finally, both histograms can be plotted on the same scale to check if the experimental data follows the theoretical distribution.

All related code is present in Annex A.

2.2 Superposition of Poisson Processes

This exercise follows a similar structure to the previous one but now involves generating four independent sequences with different λ values: (3, 7, 13, 15) and combining them into a single sequence.

To begin, it’s essential to normalize the event generation for each process. The Least Common Multiple for {3, 7, 13, 15} is 1365, representing the minimum value where all arrival rate timelines converge. Multiplying this value by each λ gives the events to generate for each process. For those seeking a larger dataset, multiplying the event array by a factor (defaulted to 1) allows to have more precise statistics.

First, generate the four independent sequences using both the event numbers and λ arrays as inputs for the function *exponential_distribution(n,l)*, similar to the previous exercise.

Next, count the frequency of events per unit time interval for all sequences and sort them. Then, sum the frequencies for each interval, treating the sequences as one.

Similarly to the previous exercise, after obtaining the total frequency of events per unit interval, count the occurrences of each event number, forming a dictionary of {event number: summed frequency}.

To plot the histogram of the sampled data against the theoretical Poisson distribution with superposition, it is necessary to both calculate percentages for the experimental data and generate the theoretical percentages for the superposition of Poisson. To achieve the theoretical percentages, function *poisson_pmf(k,lambda_)* receives as λ argument the sum of all 4 λ's for all 4 independent sequences.

All related code is present in Annex B.

2.3 M/M/1 Queue Simulation

The objective of this exercise is to simulate a M/M/1 queue using Poisson processes as a discrete event simulation. The algorithm assumes for the arrival and processing processes parameters λ and μ respectively.

The simulator is only required to support two types of events: packets arriving at the system and packets at the end of processing by the server. This leads to an event list that contains the next events to be simulated (with a tuple shape of (timestamp, event_type{'in','out'}) and a M/M/1 packet queue that will hold the set of packets waiting to be processed by the M/M/1 server.

The steps for simulating the M/M/1 queue are based on those provided in the laboratory guide.

To pursue with the analysis for this exercise, the code has been modified to calculate the experimental and theoretical values of the following metrics: average queue size, average time a packet is in the system, average time a packet is in the queue and the percentage of time the server is busy.

These metrics are achieved by halting packet generation when the number of packets processed by the simulation equals the value chosen as input during code execution.

All related code is present in Annex C.

3 Experimental Results

3.1 Arrival Process Generation

The accuracy of histograms in mirroring the Poisson distribution depends on several factors, including the sample size N, the number of bins (set as the maximum value of events within a unitary time interval), and the parameter λ dictating the process's arrival rate.

Typically, larger sample sizes tend to yield more precise histograms. This is because a greater abundance of data points allows for a closer approximation of experimental data to the theoretical distribution, effectively minimizing the impact of noisy data points.

Moreover, the number of bins plays a crucial role in histogram resolution. Too few bins oversimplify the distribution, resulting in inappropriate shapes for both experimental and theoretical distributions. However, an excess of bins may introduce noise stemming from the randomness of experimental runs. Thus, selecting an appropriate number of bins is important for accurately representing distributions.

Lastly, lower values of λ for the same number of events typically produce more accurate experimental representations compared to the theoretical distribution. This is attributed to higher λ values being more tightly concentrated around their mean and exhibiting lower variability, especially with relatively fewer events. Consequently, the narrower spread may not be visible in experimental distributions with insufficient data points.

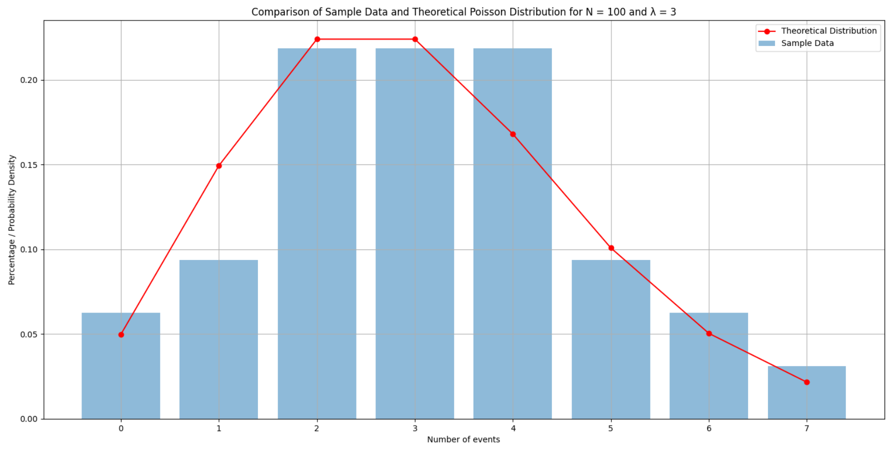


Figure 1: Histogram of Poisson process with $\lambda = 3$ and N = 100.

Starting with Figures 1 and 2, the last point made above becomes evident: for N = 100, the sampled data with $\lambda = 3$ follows fairly closely to the theoretical distribution, while for 2 the sampled data with $\lambda = 20$ fails to provide a clear picture of how the distribution presents itself.

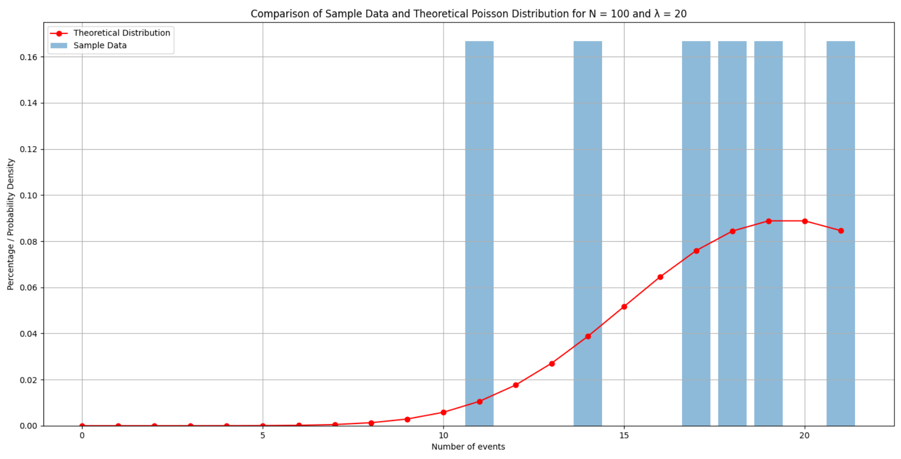


Figure 2: Histogram of Poisson process with $\lambda = 20$ and N = 100.

Transitioning to Figures 3 and 4, it is apparent that the histogram from 3 notably improves in accuracy as the number of events increases from 100 to 1000. Similarly, Figure 4 begins to exhibit sampled data that somewhat aligns with the theoretical distribution, proving the claim that larger sample sizes yield more precise and accurate histograms.

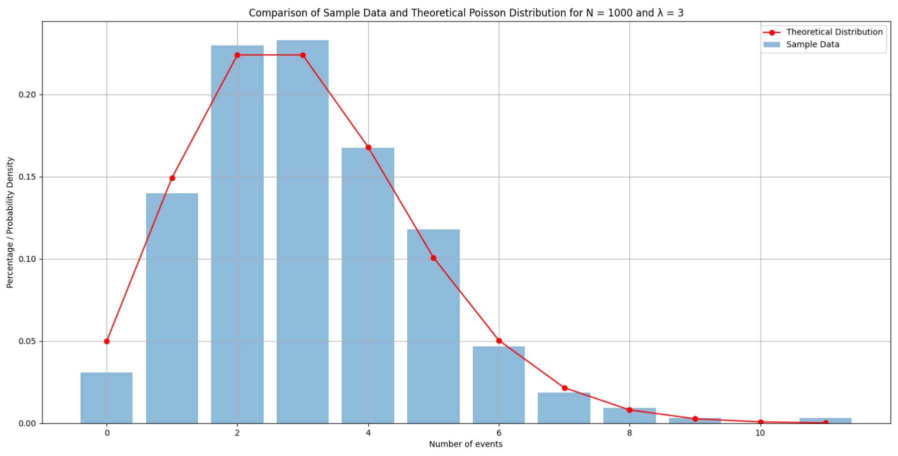


Figure 3: Histogram of Poisson process with $\lambda = 3$ and N = 1000.

In conclusion, Figure 5 showcases the histogram of sampled data for $\lambda = 20$ and N = 50000. With such a large sample size and an increased number of bins, this figure offers exceptional resolution. As a result, it closely aligns with the theoretical line of the Poisson distribution for these specific parameters.

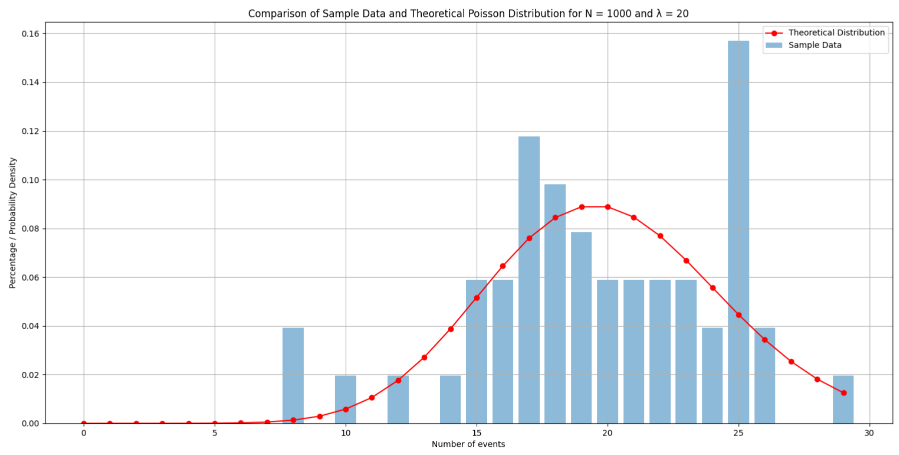


Figure 4: Histogram of Poisson process with λ = 20 and N = 1000.

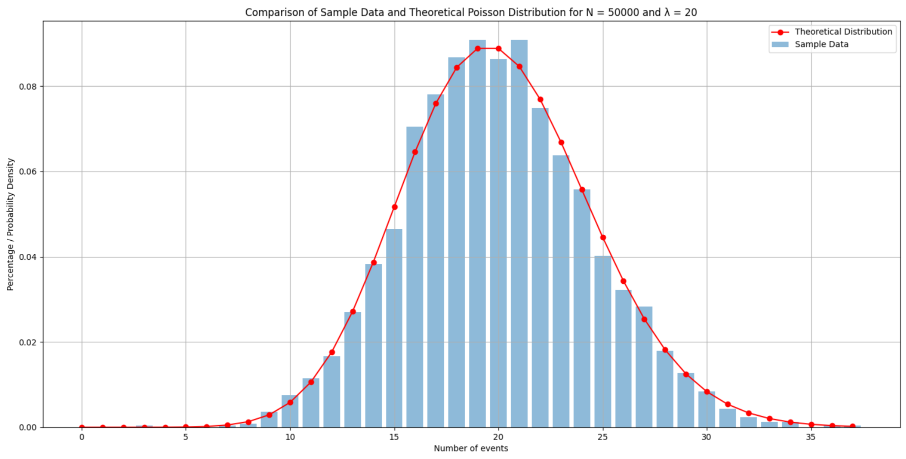


Figure 5: Histogram of Poisson process with λ = 20 and N = 50000.

3.2 Superposition of Poisson Processes

For the superposition of Poisson processes, it is expected that the combination of the 4 independent Poisson processes translates to a result that is also a Poisson arrival process, but with $\lambda = \lambda_1 + \lambda_2 + \lambda_3 + \lambda_4$.

To prove it, the theoretical Poisson distribution was generated based on the λ mentioned above. In this exercise, it is only possible to increase the number of events and discuss how the experimental superposition evolves with larger sample sizes in relation to the theoretical line.

Similarly to the previous exercise, increasing the number of events leads to more accurate and precise results, resulting in histograms that closely follow the expected Poisson distribution. Additionally, it is also expected that the higher event count enhances the resolution and detail of the experimental histogram and reduces the noise from random fluctuations as they become less pronounced.

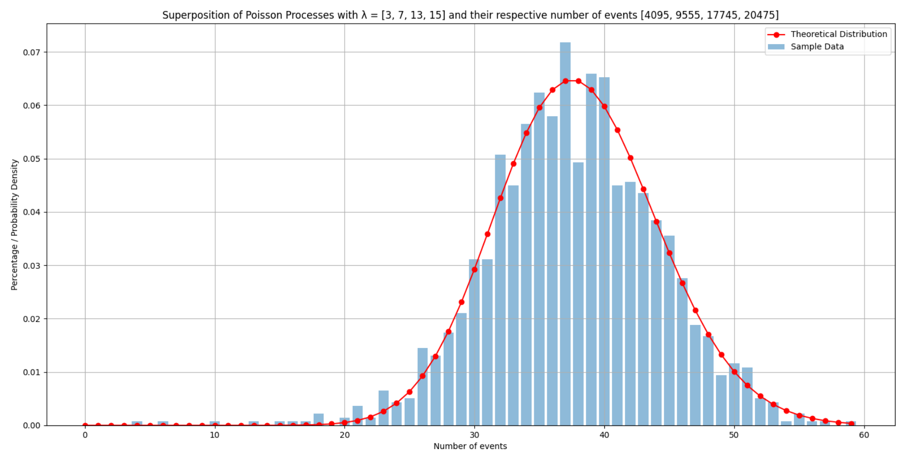


Figure 6: Histogram of the Superposition of Poisson processes with $\lambda = \{3, 7, 13, 15\}$ and the multiplier of the number of events set to 1.

Examining Figures 6, 7 and 8 sequentially reveals noticeable differences. In Figure 6, a distinct level of noise can be seen throughout the entire histogram, with some data points exhibiting considerable deviation from the expected values.

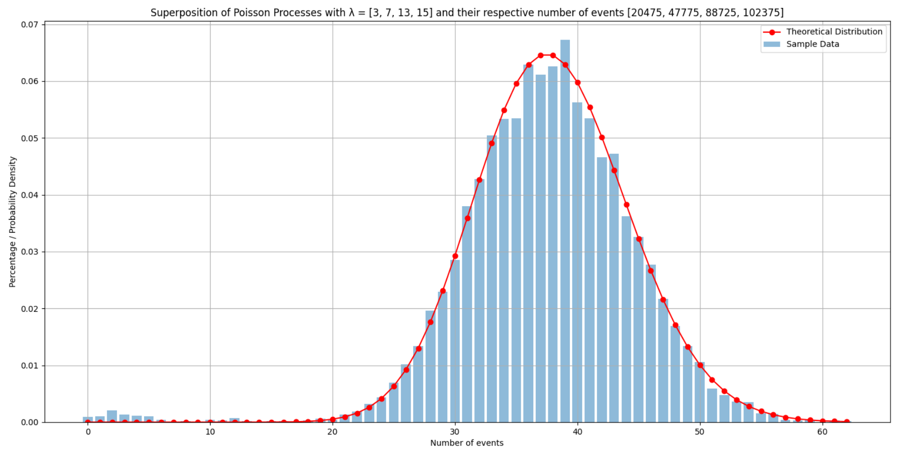


Figure 7: Histogram of the Superposition of Poisson processes with $\lambda = \{3, 7, 13, 15\}$ and the multiplier of the number of events set to 5.

However, upon multiplying the number of events by 5 (noting that the histogram in Figure 7 represents a different run from the previous figure), it becomes evident that these deviations are less pronounced, accompanied by a reduction in the overall histogram noise.

Finally, upon inspecting Figure 8, it becomes apparent that the noise data points have been effectively eliminated. The histogram now closely mirrors the theoretical Poisson distribution, a result achieved by setting the multiplier of the number of events by 50, making a case for the benefits described above in this section.

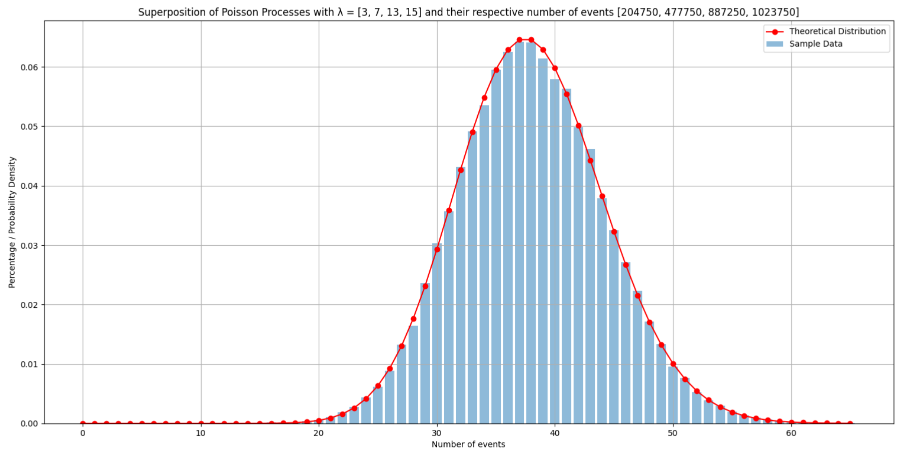


Figure 8: Histogram of the Superposition of Poisson processes with $\lambda = \{3, 7, 13, 15\}$ and the multiplier of the number of events set to 50.

3.3 M/M/1 Queue Simulation

In the M/M/1 queue exercise, the initial task involved simulating the queue under various scenarios where $\lambda <<< \mu$, $\lambda < \mu$, $\lambda = \mu$, $\lambda > \mu$, and $\lambda >>> \mu$. These combinations enable us to determine if the simulation is functioning correctly and if any calculations were misrepresented.

Table 1 displays the outcomes of our simulation conducted with various arrival and service rates for a total of N = 10000 events.

N=10000 events	**Average Time in System(W_s)**		**Average Time in Queue(W_q)**		**Average Queue Size(L_q)**		**Server Utilization % (ρ)**	
	Experimental	**Theoretical**	**Experimental**	**Theoretical**	**Experimental**	**Theoretical**	**Experimental**	**Theoretical**
$\lambda = 2, \mu = 10$	0.125858	0.125	0.0245918	0.024999	0.14975	0.04999	20.0344	20.0
$\lambda = 7, \mu = 10$	0.311633	0.333333	0.211673	0.233333	1.79476	1.633333	69.8706	70.0
$\lambda = 9, \mu = 10$	0.942254	1.0	0.840420	0.9	7.875662	8.1	90.410971	90.0
$\lambda = 10, \mu = 10$	9.243872	inf	9.143187	inf	92.157071	inf	99.850995	100.0
$\lambda = 10.2, \mu = 10$	9.267934	-5.0000	9.166979	-5.10000	90.468061	-52.0200	99.604711	102.0
$\lambda = 14, \mu = 10$	140.820276	-0.25	140.721026	-0.35	2029.453153	-4.8999	99.973381	140

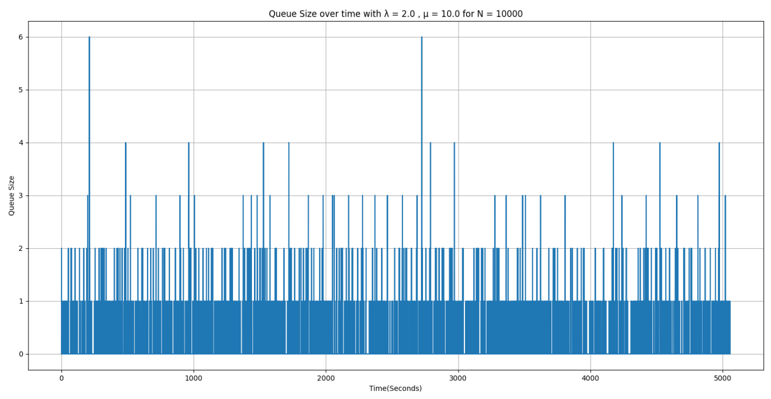
Table 1: M/M/1 queue statistics with different service and arrival rates for N = 10000 events.

By analyzing the graphs from Figure 9, we observe distinct behaviors depending on the relationship between arrival and service rates. In simulations where the arrival rate is lower than the service rate, depicted in graphs a) to c), the queue sizes remain relatively stable. Although they fluctuate around their mean value, they consistently stay manageable.

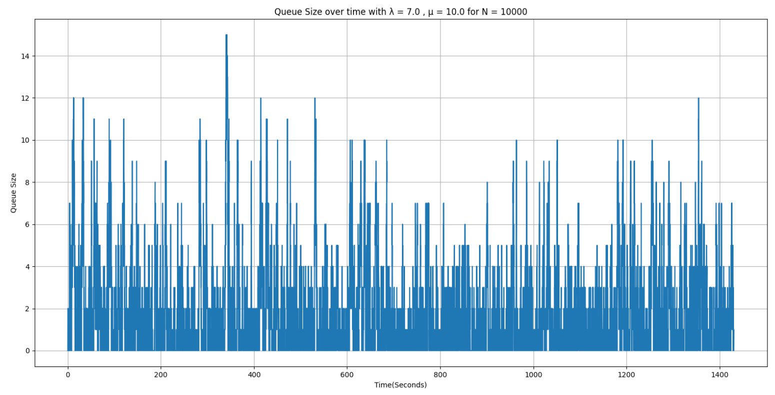
This stability arises from the server processing events faster than they arrive, leading to multiple instances of queue emptiness during the simulation. Notably, as we transition from graph a) to c), the frequency of queue size reaching zero diminishes progressively, aligning with expectations based on the given arrival and service rates.

Conversely, in graphs d) to f), the server struggles to cope with the faster arrival rates, resulting in a scenario where arrivals exceed the service capacity. Consequently, the queue size perpetually remains nonzero and tends towards infinity over time.

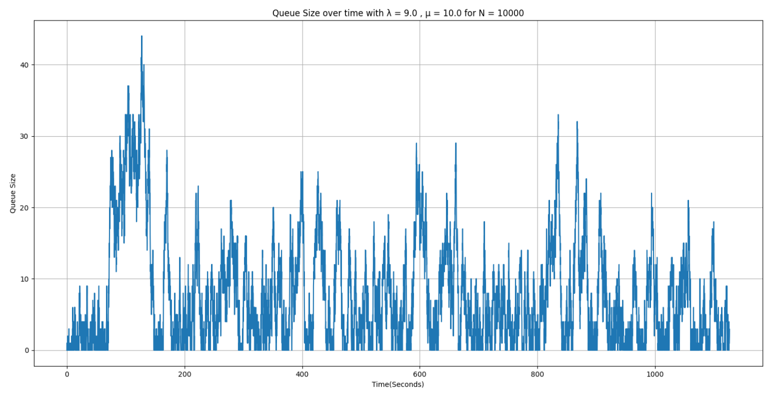
These observations align with the trends reflected in Table 1, where the average queue size consistently increases across the table, reflecting the impact of varying arrival and service rates on system behavior.



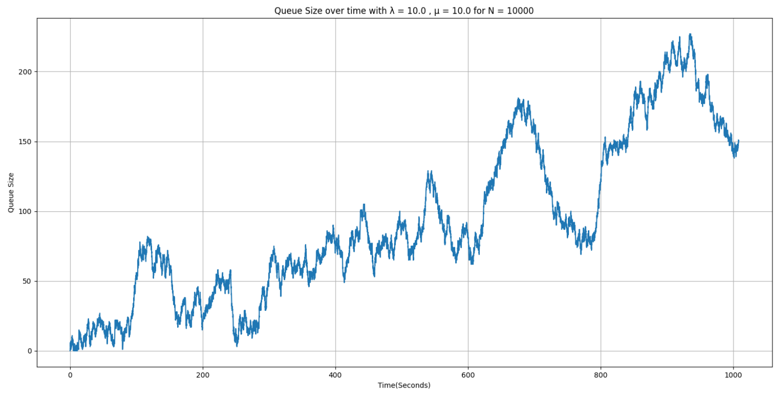
(a) Queue size over time for $\lambda = 2, \mu = 10$ and N $= 10000$



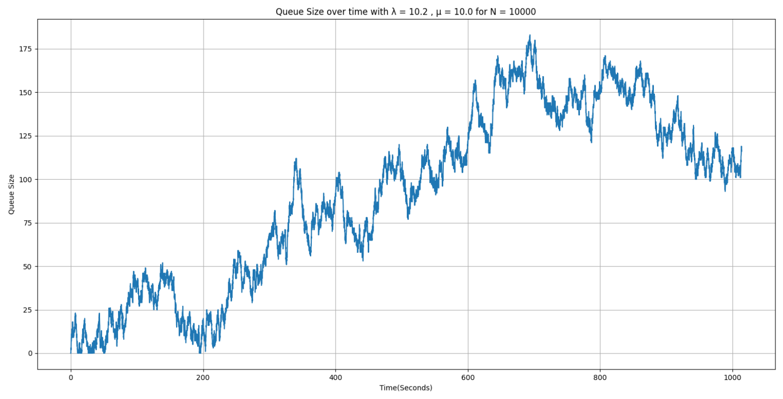
(b) Queue size over time for $\lambda = 7, \mu = 10$ and N $= 10000$.



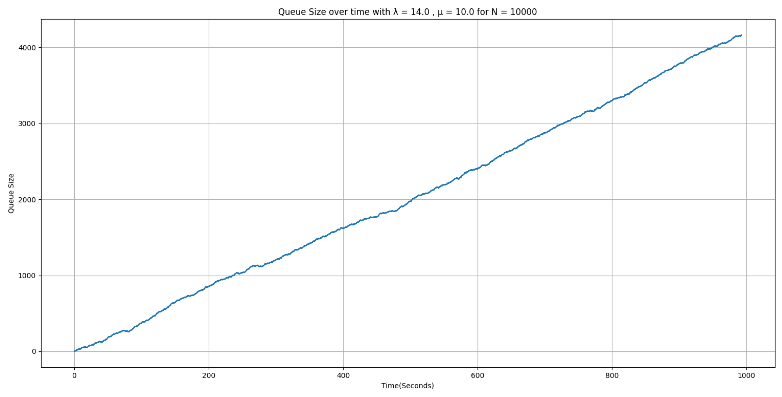
(c) Queue size over time for $\lambda = 9, \mu = 10$ and N $= 10000$



(d) Queue size over time for $\lambda = 10, \mu = 10$ and N $= 10000$.



(e) Queue size over time for $\lambda = 10.2, \mu = 10$ and N $= 10000$

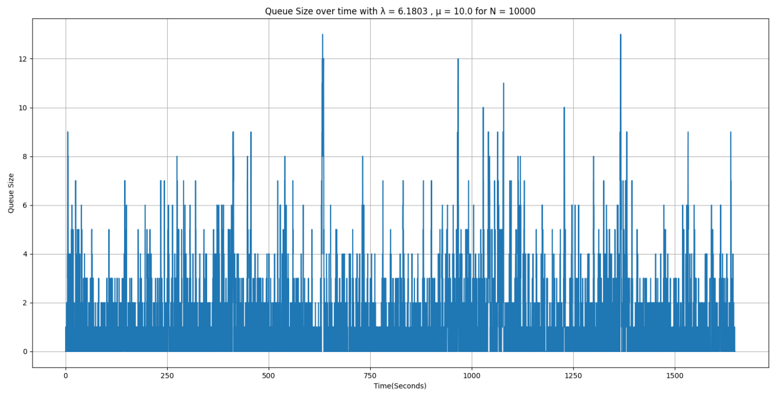


(f) Queue size over time for $\lambda = 14, \mu = 10$ and N $= 10000$.

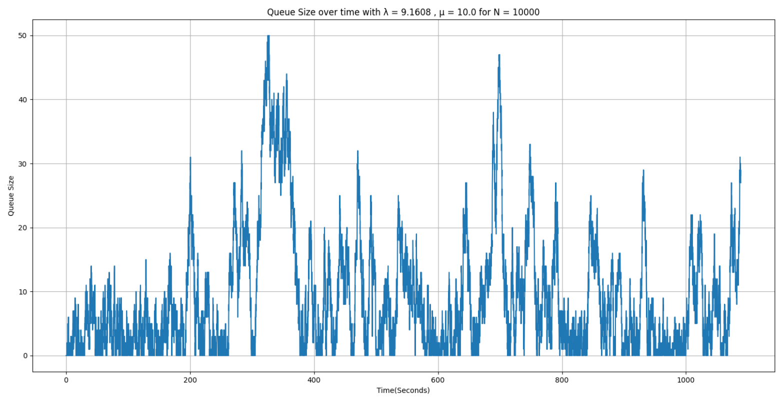
Figure 9: Queue sizes over time for all simulations runs present in Table 1.

N=10000 events	Average Time in System(W_s)		Average Time in Queue(W_q)		Average Queue Size(L_q)		Server Utilization % (ρ)	
	Experimental	Theoretical	Experimental	Theoretical	Experimental	Theoretical	Experimental	Theoretical
$\lambda = 6.1803, \mu = 10$	0.233685	0.26180	0.136527	0.161800	1.118	1	58.951941	61.803
$\lambda = 9.1608, \mu = 10$	1.074306	1.191611	0.975004	1.091611	9.344118	10	91.21772	91.608
$\lambda = 9.9020, \mu = 10$	2.410406	10.204081	2.310143	10.104081	22.766363	100	97.322554	99.02
$\lambda = 9.9900, \mu = 10$	10.49536	100.00	10.393518	99.900	103.07129	1000	99.9274	99.9

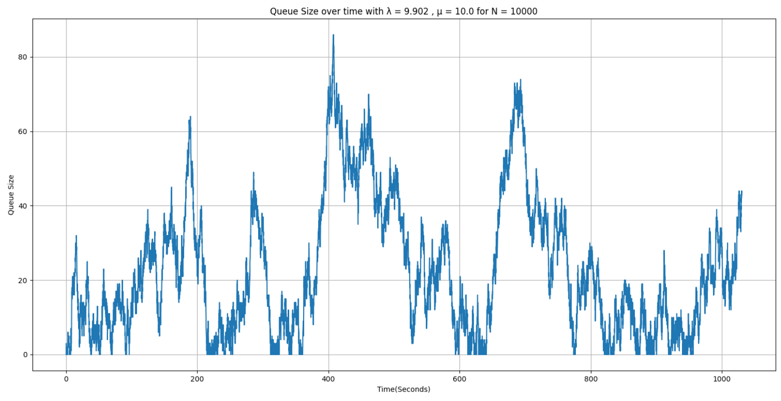
Table 2: M/M/1 queue statistics with arrival/service rates for target queue sizes L_q {1, 10, 100, 1000}.



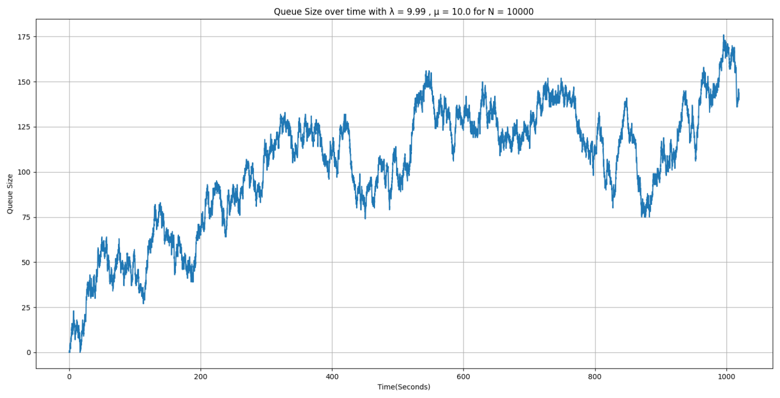
(a) Queue size over time for target $L_q = 1$ and N = 10000



(b) Queue size over time for target $L_q = 10$ and N = 10000.



(c) Queue size over time for target $L_q = 100$ and N = 10000.



(d) Queue size over time for target $L_q = 1000$ and N = 10000.

Figure 10: Queue sizes over time for target sizes in Table 2.

Now, let's delve into the task focused on targeting specific queue sizes. Table 2 showcases the results obtained from our simulation code, where we used arrival and service rate values leading to theoretical queue sizes of 1, 10, 100, and 1000.

These were successfully followed by the experimental values, which correspond to the mean values if they were to be done on the corresponding graphs in Figure 10. Notably, among these, graph d) stands out as the only one trending towards infinity over time. This observation reaffirms that simulations where the server utilization ρ approaches or exceeds 1 can result in queue sizes growing indefinitely.

Hence, we can confidently assert that our code has been effectively implemented and accurately mirrors the behavior of an M/M/1 queue.

$\lambda = 9.9020, \mu = 10$	**Average Time in System(W_s)**		**Average Time in Queue(W_q)**		**Average Queue Size(L_q)**		**Server Utilization % (ρ)**	
	Experimental	**Theoretical**	**Experimental**	**Theoretical**	**Experimental**	**Theoretical**	**Experimental**	**Theoretical**
N = 1000	2.651378	10.204081	2.5525	10.104081	26.915609	100	99.502695	99.02
N = 10000	3.603133	10.204081	3.502923	10.104081	34.559086	100	97.804781	99.02
N = 100000	5.493763	10.204081	5.39399	10.104081	53.742415	100	98.55412	99.02
N = 1000000	9.137767	10.204081	9.037693	10.104081	89.748817	100	99.034018	99.02

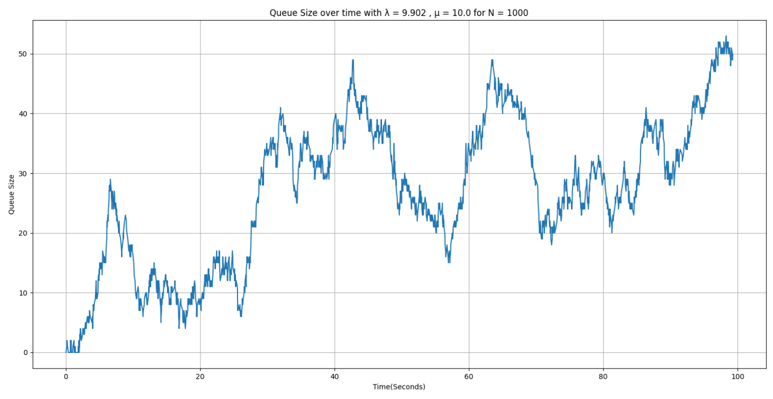
Table 3: M/M/1 queue statistics with $\lambda = 9.9020$ and $\mu = 10$ with N = $\{1000, 10000, 100000, 1000000\}$.

The last task of the M/M/1 simulation exercise involved identifying a scenario where the queue size reaches 100. To achieve this, we extracted the arrival and service rate values from Table 2 and utilized them to construct the scenario.

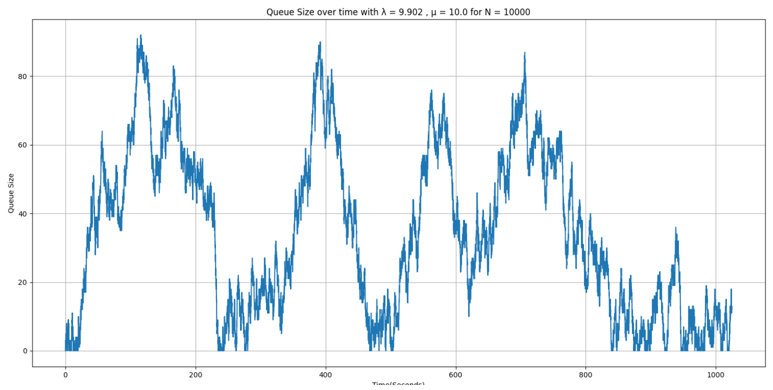
Specifically, we set the arrival rate λ to 9.9020 and the service rate μ to 10. Furthermore, we aimed to systematically increase the number of generated events to investigate the correlation between the number of events and the queue size converging to the theoretical value. For this purpose, we tested N = 1000, 10000, 100000, 1000000.

Upon examining the results presented in Table 3, we observed that the experimental average queue size progressively approached the theoretical value as the number of generated events increased. However, the histograms depicted in Figure 10 offer a different insight. For a smaller number of events, particularly in graphs a) and b), the system had limited time to stabilize, leading to scenarios where the server struggled to recover from being consistently busy due to the high variance associated with a small data sample size.

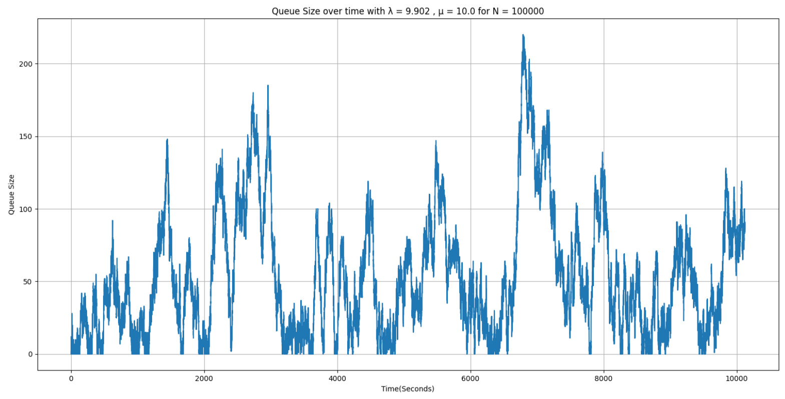
This phenomenon, known as transient behavior, manifests as the queue's inability to adjust to the arrival and service rates, resulting in fluctuations where the system fails to stabilize where supposedly it should be able dispatch all packets faster than they are arriving. As the number of events increases, the transient behavior becomes less pronounced, and the plot approaches a more stable behavior, eventually converging to the theoretical value with the appropriate number of generated events.



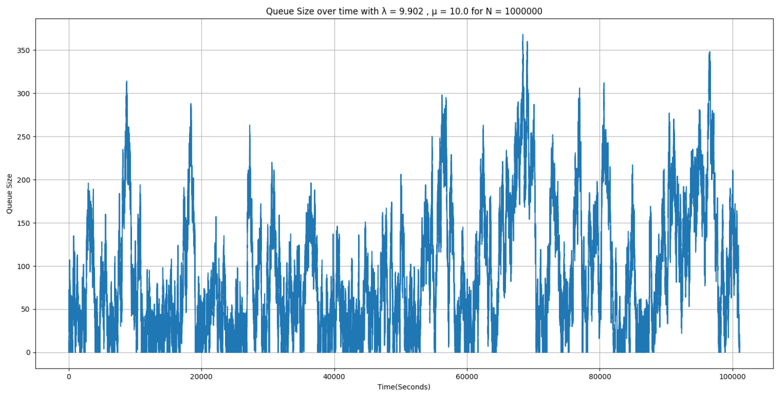
(a) Queue size over time for $L_q = 100$ and N = 1000.



(b) Queue size over time for $L_q = 100$ and N = 10000.



(c) Queue size over time for $L_q = 100$ and N = 100000.



(d) Queue size over time for $L_q = 100$ and N = 1000000.

Figure 11: Queue size over time for $L_q = 100$ and N = {1000, 10000, 100000, 1000000} in Table 3.

Appendices

A Poisson Arrival Process Generation

```
import math
import random
import time
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from collections import Counter

def exponential_distribution(n,l):
    # Generates the sequence of events according to the exponential
        distribution

    event_timestamp = np.zeros(n)
    time_intervals = np.zeros(n)
    time_ = 0

    for i in range(n):
        random.seed(time.time())
        u = random.random()
        deltat = -math.log(1-u)/l
        time_ += deltat
        time_intervals[i] = deltat
        event_timestamp[i] = int(math.floor(time_))
        i -= 1
    return event_timestamp

def plot_histogram(data,n,l):
    # print(data)
    # Count how many times each value appears
    value_counts = Counter(data.values())
    #print(value_counts)

    # Calculate total count
    total_count = sum(value_counts.values())
    #print(total_count)

    # Calculate the percentage for each number of events
    percentages = {count: (freq / total_count) for count, freq in
        value_counts.items()}
    #print(percentages)

    # Plotting sample data histogram
    plt.bar(percentages.keys(), percentages.values(), align='center',
        alpha=0.5, label='Sample Data')
    # Calculating and plotting Theoretical Poisson Distribution
    theoretical_probs = [poisson_pmf(k,l) for k in
        range(max(data.values())+1)]
    # print(theoretical_probs)
    plt.plot(range(max(data.values()) + 1), theoretical_probs,
        marker='o', linestyle='-', color='r', label='Theoretical
        Distribution')

    plt.xlabel('Number of events')
    plt.ylabel('Percentage / Probability Density')
    plt.title('Comparison of Sample Data and Theoretical Poisson
        Distribution for ' + 'N = ' + str(n) + ' and Lambda = ' +
        str(int(l)))
    plt.grid(True)
    plt.legend()
    plt.show()
    return 0

def poisson_pmf(k, lambda_):
    # Probability Mass Function (PMF) of the Poisson distribution
    return np.exp(-lambda_) * (lambda_**k) / np.math.factorial(k)

def main():
    n = int(input("Number of loops\n"))
    l = float(input("Lambda\n"))
    events = exponential_distribution(n,l)
    #print (events)

    # Count the frequency for each number of events occuring in a
        unitary time interval
    event_counting = Counter(events)

    # If any value is not present in the array, add it with a
        frequency of 0
    for i in range(int(max(events))+1):
       if i not in event_counting:
          event_counting[i] = 0

    # Sort the dictionary by keys
    sorted_event_counting = dict(sorted(event_counting.items()))

    # Plot both histograms for the Sample Data and Theoretical Poisson
        Distributions
    plot_histogram(sorted_event_counting,n,l)
    return 0

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

B Superposition of Poisson Processes

```
import math
import random
import time
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from collections import Counter

def exponential_distribution(n,l):
    # Generates the sequence of events according to the exponential
        distribution

    event_timestamp = np.zeros(n)
    time_intervals = np.zeros(n)
    time_ = 0

    for i in range(n):
        random.seed(time.time())
        u = random.random()
        deltat = -math.log(1-u)/l
        time_ += deltat
        time_intervals[i] = deltat
        event_timestamp[i] = int(math.floor(time_))
        i -= 1
    return event_timestamp

def plot_histogram(data,n,l):
    # Count how many times each value appears
    value_counts = Counter(data.values())
    # print(value_counts)

    # Calculate total count
    total_count = sum(value_counts.values())
    # print(total_count)

    # Calculate the percentage for each number of events
    percentages = {count: (freq / total_count) for count, freq in
        value_counts.items()}
    # print(percentages)

    # Plotting sample data histogram
    plt.bar(percentages.keys(), percentages.values(), align='center',
        alpha=0.5, label='Sample Data')

    # Calculating and plotting Theoretical Poisson Distribution
    theoretical_probs = [poisson_pmf(k,sum(l)) for k in
        range(max(data.values())+1)]
    # print(theoretical_probs)
    plt.plot(range(max(data.values()) + 1), theoretical_probs,
        marker='o', linestyle='-', color='r', label='Theoretical
        Distribution')

    plt.xlabel('Number of events')
    plt.ylabel('Percentage / Probability Density')
    plt.title('Superposition of Poisson Processes with Lambda = ' +
        str(l) + ' and their respective number of events ' + str(n))
    plt.grid(True)
    plt.legend()
    plt.show()
    return 0

def poisson_pmf(k, lambda_):
    # Probability Mass Function (PMF) of the Poisson distribution
    return np.exp(-lambda_) * (lambda_**k) / np.math.factorial(k)

def main():
    # Least Common Multiplier chosen is 1365.
    # Default Number of Events = Timeline * Lambda
    default_values = [4095,9555,17745,20475]
    n = [value * 50 for value in default_values]
    l = [3,7,13,15]

    events = [None] * 4
    for i in range(0,len(l)):
        events[i] = exponential_distribution(n[i],l[i])
    # print(events)

    # Count the frequency for each number of events occuring in a
        unitary time interval
    event_counting = [None] * 4
    for i in range(0,len(events)):
        event_counting[i] = Counter(events[i])
    # print(event_counting)


    # If any value is not present in any of the sequences, add it with
        a frequency of 0
    index = 0
    for sequence in events:
        for i in range(int(max(sequence))+1):
            if i not in (event_counting[index]):
                event_counting[index][i] = 0
        index += 1

    # Sort the dictionaries by keys
    sorted_event_counting = [None] * 4
    index = 0
    for sequence in event_counting:
        sorted_event_counting[index] = dict(sorted(sequence.items()))
        index += 1
    #print(sorted_event_counting[0])

    # Joining all sequences together
    sequence_sum = {}
    for sequence in sorted_event_counting:
        for i in range(int(max(sequence))+1):
            if i not in (sequence_sum):
                sequence_sum[i] = sequence[i]
            else:
                sequence_sum[i] += sequence[i]
    #print(sequence_sum)

    # Plot both histograms for the Sample Data and Theoretical Poisson
        Distribution with SuperPosition
    plot_histogram(sequence_sum,n,l)
    return 0

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

C M/M/1 Queue Simulation

```
import random
from statistics import mean
from matplotlib import pyplot as plt
import numpy as np
import math
import time
import queue as qu

def exponential_distribution(n,l):
    # Generates the sequence of events according to the exponential
        distribution

    event_timestamp = np.zeros(n)
    time_intervals = np.zeros(n)
    time_ = 0

    for i in range(n):
        random.seed(time.time())
        u = random.random()
        deltat = -math.log(1-u)/l
        time_ += deltat
        time_intervals[i] = deltat
        event_timestamp[i] = float(time_)
        i -= 1
    return event_timestamp

def stats( total_queue_time, total_processing_time,packets_served
    ,ending_timer,system_idle_time, arrival_rate,
    service_rate,queue_size_array):

    rho = arrival_rate / service_rate

    print ("------------------------------------")
    total_waiting_time = total_queue_time + total_processing_time
    average_packet_time_in_system = total_waiting_time/packets_served
         if packets_served > 0 else 0
    print("Average packet time in system: ",
         average_packet_time_in_system)
    try:
        Ws = 1 / (service_rate - arrival_rate)
    except ZeroDivisionError:
        Ws = float('inf')
    print("Theoretical Average Packet Time in the system: ", Ws)

    print ("------------------------------------")
    average_packet_time_in_queue = total_queue_time / packets_served
         if packets_served > 0 else 0
    print("Average packet waiting time: ",
         average_packet_time_in_queue)
    Wq = Ws - (1 / service_rate)
    print("Theoretical Average Packet Time in the queue",Wq)

    print ("------------------------------------")
    average_size = average_queue_size(queue_size_array)
    print("Average queue size: ", average_size)
    Lq = arrival_rate * Wq
    print("Theoretical Average Queue Size",Lq)

    print ("------------------------------------")
    total_busy_time = (ending_timer - system_idle_time)/ending_timer
         *100
    print("Percentage of time busy % : ", total_busy_time)
    print("Theoretical busy time % :",rho*100 )


    return

def average_queue_size(queue_size_array):
    sizes = [size[1] for size in queue_size_array]
    return mean(sizes)

def processing(arrival_rate, service_rate, event_cap):
```

```
event_list = []
queue = qu.Queue()
server_free = True
total_queue_time = float(0)
total_processing_time = float(0)
system_idle_time = float(0)
packet_arrived = 0
packets_served = 0
block_in_creation = False
# This list storages tuples that allow to understand the size of
    queue overtime per timestamp - tuple (timestamp,len(queue))
queue_size_array = []
timer = 0
event_list.append((0,"in"))
idle_since = float(0)

while event_list:
    # If number of events is the maximum value of events
    if packets_served >= event_cap:
        break

    # Step 1 - Read the next tuple from the event list
    current_event = event_list.pop(0)

    # Step 2 - If the event type is an arrival
    if current_event[1] == "in" and not block_in_creation:
        # Add new arrival to the queue and generate new arrival
            event to append into the list
        if packets_served <= event_cap:
            queue.put(current_event)
            next_event = (current_event[0] +
                exponential_distribution(1,arrival_rate)[0], "in")
            event_list.append(next_event)
            packet_arrived += 1
            event_list = sorted(event_list, key=lambda x: x[0])
        # Jump to Step 4
        else:
            block_in_creation = True
            continue

    # Step 3 - If the event type is a processing
    else:
        # Mark server as free and count the waiting time
        server_free = True
        idle_since = current_event[0]
        packets_served += 1

    # Step 4 - if the server is free and the queue is empty
    if (server_free and queue.empty()):
        timer = current_event[0]
```

```
            queue_size_array.append((current_event[0],queue.qsize()))
            continue

        if not server_free:
            timer = current_event[0]
            queue_size_array.append((current_event[0],queue.qsize()))
            continue

        # Step 5 - Process the packet inside the queue
        leaving_event = queue.get()
        server_free = False
        # If the previous event was a processing, calculate the idle
             time in between
        if idle_since is not None:
            system_idle_time += current_event[0] - idle_since
            #system_idle_time += leaving_event[0] - idle_since
            idle_since = None

        time_in_processing = current_event[0] +
             exponential_distribution(1,service_rate)[0]
        event_list.append((time_in_processing, "out"))
        event_list = sorted(event_list, key=lambda x: x[0])
        # Count the time the leaving packet was inside the queue
        total_queue_time += current_event[0]- leaving_event[0]
        # Count the processing time for the leaving packet
        total_processing_time += time_in_processing - current_event[0]
        timer = current_event[0]
        queue_size_array.append((current_event[0],queue.qsize()))

    ending_timer = timer
    stats(total_queue_time, total_processing_time, packets_served,
         ending_timer, system_idle_time, arrival_rate,
         service_rate,queue_size_array)

    #Plot the queue size over time
    x_axis = [value[0] for value in queue_size_array]
    y_axis = [value[1] for value in queue_size_array]
    plt.plot(x_axis,y_axis)
    plt.xlabel('Time(Seconds)')
    plt.ylabel('Queue Size')
    plt.title('Queue Size over time with Lambda = ' +
         str(arrival_rate) + ' , Miu = ' + str(service_rate) + ' for N
         = ' + str(event_cap))
    plt.grid(True)
    plt.show()
    return

if __name__== "__main__":
    #Lq = 1 -> rho = 0.61803 -> lambda = 6.1803 and miu = 10
    #Lq = 10 -> rho = 0.91608 -> lambda = 9.1608 and miu = 10
    #Lq = 100 -> rho = 0.99020 -> lambda = 9.9020 and miu = 10
    #Lq = 1000 -> rho = 0.99900 -> lambda = 9.9900 and miu = 10
    arrival_lambda = float(input("Arrival rate\n"))
    service_lambda = float(input("Service rate\n"))
    event_cap = int(input("event cap\n"))
    processing(arrival_lambda, service_lambda, event_cap)
```