Queueing theory phase2

Navid Najafi
mobin jelodar

August 2022

Simulation tasks:

Task1:Simulating of a simplified edge-server

finding rate of denials and Expectation number of jobs in que:

first we want to discuss theoretically:

Expectation of Jobs in a Finite-Capacity Queue with Feedback

Consider a queue system with the following parameters:

- Arrival rate: λ
- Service rate: μ
- Feedback probability: p
- Queue capacity: N

Effective Arrival Rate

Each job that completes service returns to the queue with probability p, leading to an effective arrival rate:

$$\lambda_{\text{eff}} = \lambda + p\mu$$

Expected Number of Jobs in the Queue

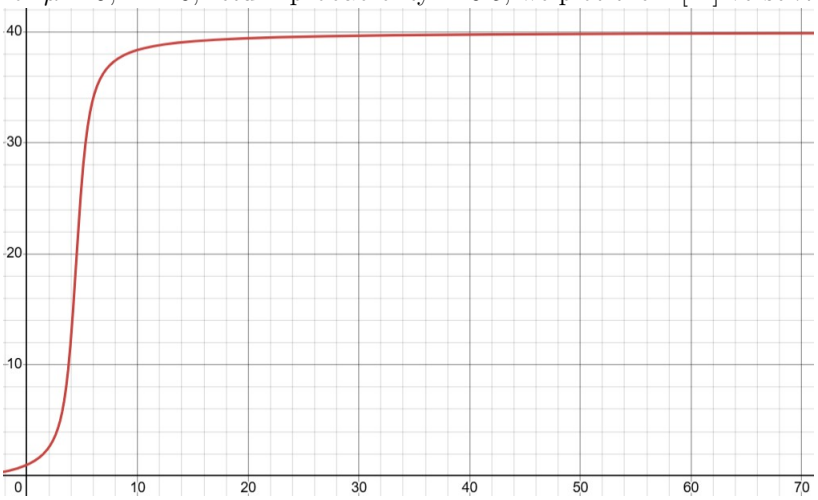
For an **M/M/1/N** queue with feedback, the expected number of jobs in the system $E[N]$ is approximately:

$$E[N] \approx \frac{\rho_{\text{eff}}(1-(N+1)\rho_{\text{eff}}^{N}+N\rho_{\text{eff}}^{N+1})}{(1-\rho_{\text{eff}})(1-\rho_{\text{eff}}^{N+1})}$$

we will plot $E[N]$ verse λ in the simulation
where the **effective utilization** ρ_{eff} is:

$$\rho_{\text{eff}} = \frac{\lambda_{\text{eff}}}{\mu} = \frac{\lambda + p\mu}{\mu}$$

for $\mu = 9$, N=40, return-probabibilty = 0.5, we plot the $E[N]$ verse λ:



it is very simillar to what we have drove in simulation.

Special Case: Infinite Queue Capacity ($N \to \infty$)

If the queue capacity is unlimited, the formula simplifies to the standard **M/M/1** expectation:

$$E[N] = \frac{\rho_{\text{eff}}}{1-\rho_{\text{eff}}}$$

Blocking Probability in an M/M/1/N Queue with Feedback

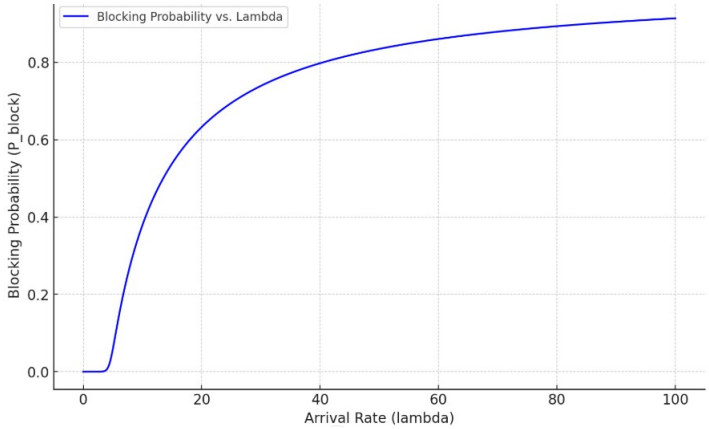
The blocking probability (denial rate) is given by:

$$P_{\text{block}} = \frac{\rho_{\text{eff}}^{N}}{\sum_{k=0}^{N}\rho_{\text{eff}}^{k}}$$

Using the geometric series sum formula, it simplifies to:

$$P_{\text{block}} = \frac{\rho_{\text{eff}}^{N}(1-\rho_{\text{eff}})}{1-\rho_{\text{eff}}^{N+1}}$$

for $\mu = 9$, N=40, return-probabibilty = 0.5, we plot the P_{block} verse λ:



we will plot the relation between $denial - rate$ and λ.
where the effective utilization is:

$$\rho_{\text{eff}} = \frac{\lambda + p\mu}{\mu}$$

- If $N \to \infty$, then $P_{\text{block}} \to 0$, meaning no job is denied.
- If p increases, more jobs return, leading to higher blocking.
- If μ increases, jobs leave faster, reducing blocking.
- If λ increases, more jobs arrive, increasing blocking. (we will see in the simulation further)

lets break down the code:
first we define our parameters:

```
# Parameters
lambda_rate = 4
mu_rate = 9
capacity = 40
return_probability = 0.5
simulation_time = 1000

time = 0
state = 0
times = [time]
states = [state]
counter_getrid = 0
```

now we write a loop section that in each iteration determines that weather next events in arrival or departure.

lets define T_i and S_i as random varaible $Exp(\lambda)$ and $Exp(\mu)$ in iteration number i.

we are going to see an arrival only if $T_i < S_i$ and vise versa.

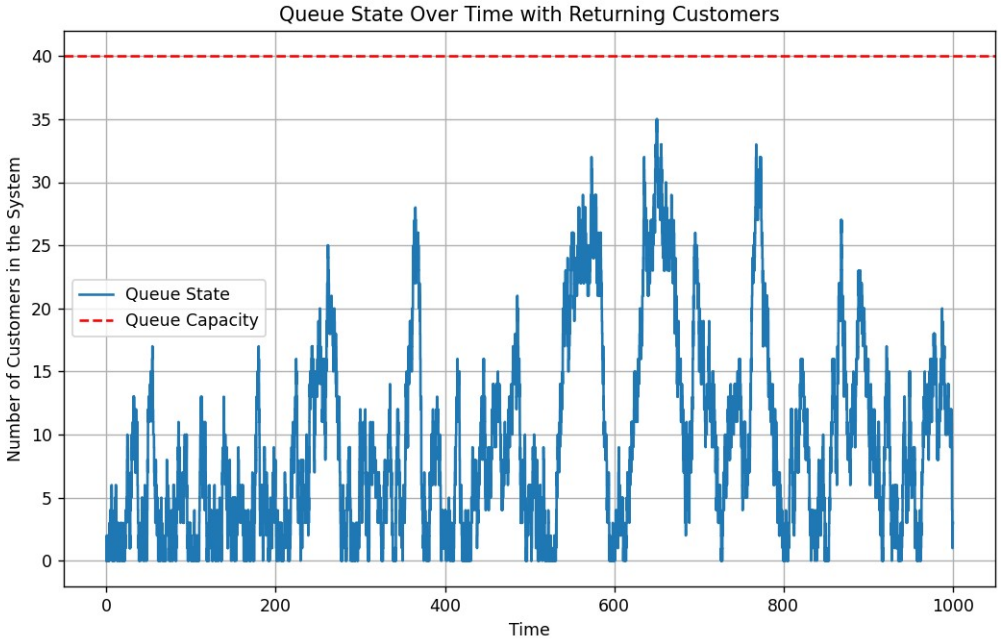
we have to handle that if the que if full we can not have any arrival and if the que is empty we can not have any departue. and for some reason we want

to capture the number of jobs that have been denied because of que being full. so the code will be like:

```
while time < simulation_time:
    # Calculate next arrival and service times
    arrival_time = np.random.exponential(1 / lambda_rate)
    service_time = np.random.exponential(1 / mu_rate) if state > 0 else float('inf')
    #next event
    if arrival_time < service_time:
        # Arrival
        time += arrival_time
        if state < capacity:
            state += 1
        else:
            counter_getrid += 1
    else:
        # departure
        time += service_time
        state -= 1
        # Check if the customer returns
        if random.random() < return_probability:
            if state < capacity:
                state += 1
    times.append(time)
    states.append(state)
```

now we print and plot the results:

for $\lambda = 4$, $\mu = 9$, $capacity = 40$, $P_{return} = 0.5$, simulation time $= 1000$, we have:



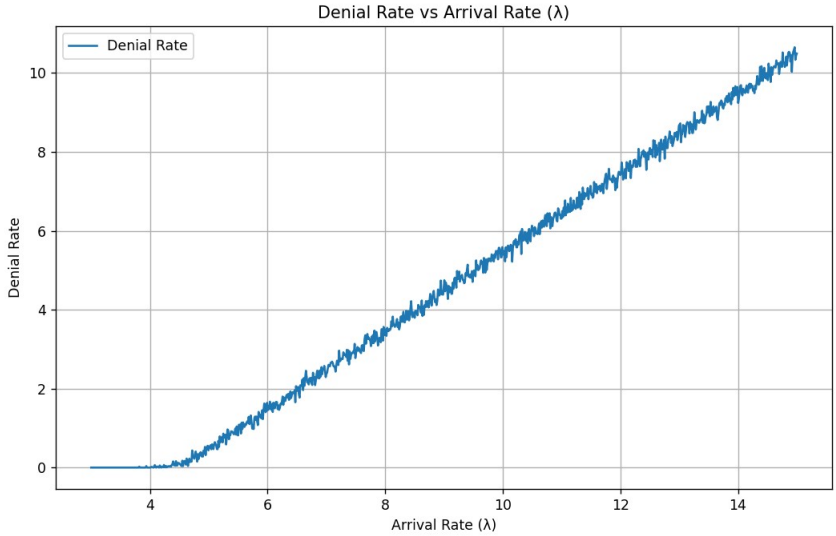
as you can see, the que never reached the limit, and statistically we have:

```
the rate of getting rid of jobs because the que is full equals to : 0
the Expectation of number of jobs in que : 9.808895123360013
```

now we want to find the relation between denial rate and rate of arrivals λ:

so we sweep λ and do the simulation and plot the keep denial rate and expectation number of jobs in que in a set for each λ, the result is like:

note that $\mu = 9$ is fixed.



we can prove by theory that this relation holds in steady state:

$$P[\text{a job being rejected}] = P_{block} = \frac{\lambda}{\mu} \times \frac{1}{N}$$

then we can see that relation between probability of rejection and λ is linear in approximation.

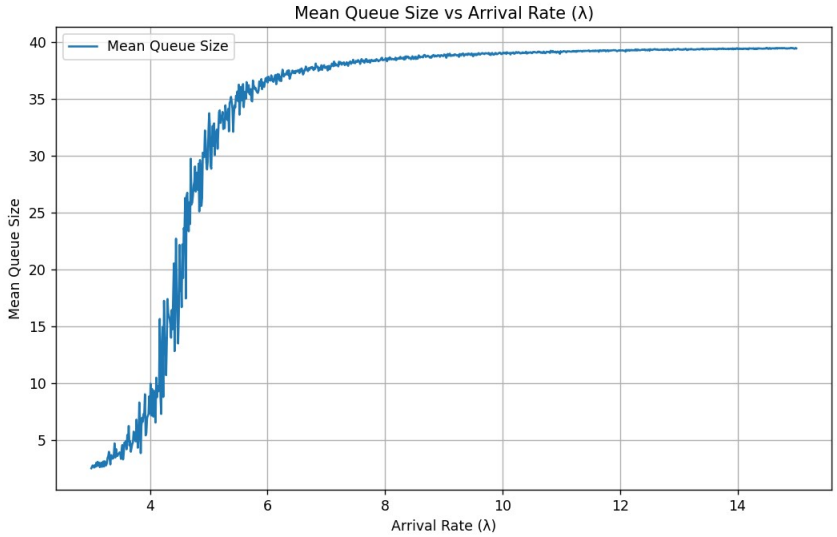
we saw the analytic (exact) relation before (in page 3)

if $\lambda < \frac{\mu}{2} = 4.5$ approximately, denial rate is 0, because the que has not reached saturation.

we can see the slop of graph is approximately 0.678, so the relation in like:

$$rate_{denials} = \begin{cases} 0 & \text{if } \lambda < 4.3 \\ 0.678(\lambda - 4.3) & \text{if } \lambda > 4.3 \end{cases} \tag{1}$$

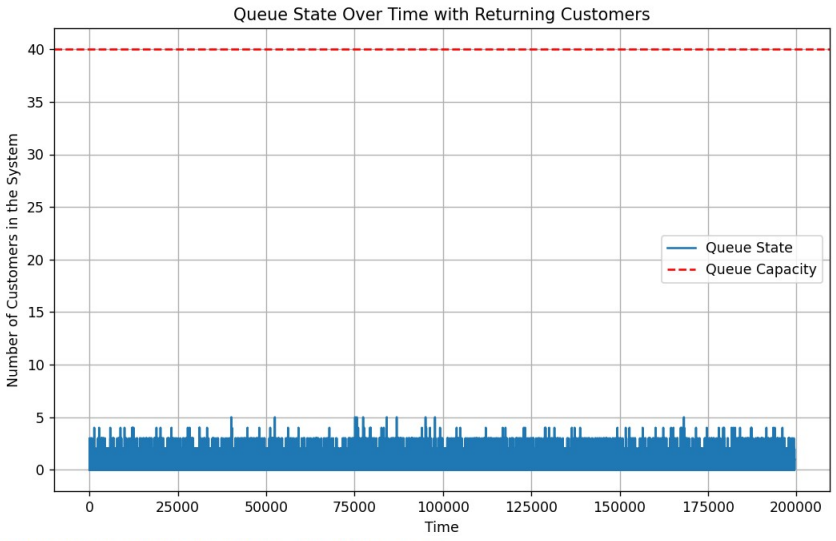
we expect that as rate of arrivals increase, expectation number of jobs existing in the que in 1000 unit of time, get closer and closer to limit:



now we change the code to work with a limit on total jobs (for example we have N bits totally to send) instead of simulation time.

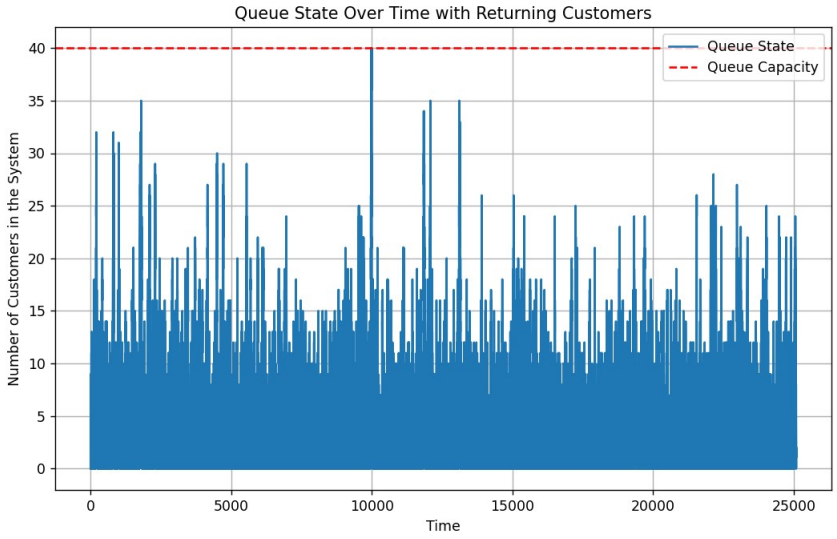
all we have done is to set a total job limitation that our while loop, iterates till the limit, also we have to keep the number of processed jobs. now we set $\mu = 10$.

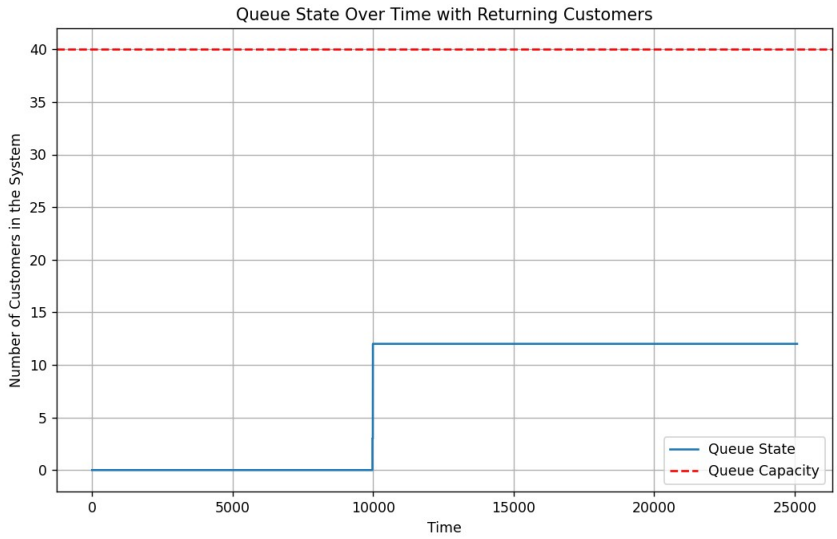
if $\lambda = 0.05\mu$:




```
The rate of denials because the queue is full equals to: 0.0
The expectation of the number of jobs in the queue: 0.7775484455156481
```

if $\lambda = 0.4\mu$:

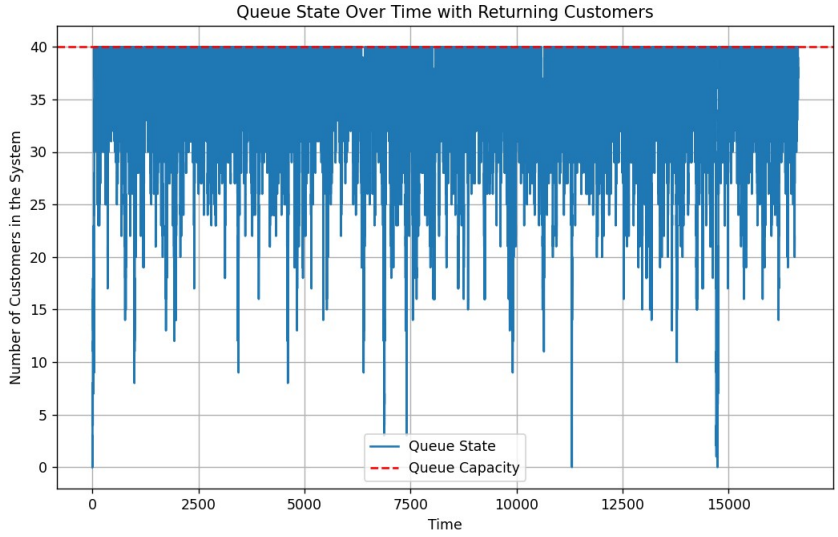


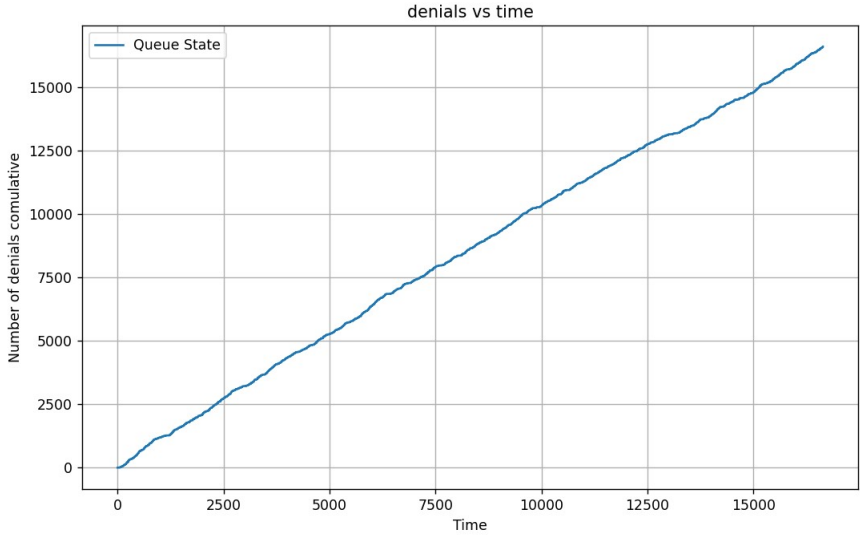



```
The rate of denials because the queue is full equals to: 0.00012
The expectation of the number of jobs in the queue: 4.736527843455906
```

so in this case, 0.012 percentage of jobs have been denied and this happend at time 10000. at that point of time the que got saturated and denied about 12 jobs, after that the que never got saturated.

if $\lambda = 0.6\mu$:

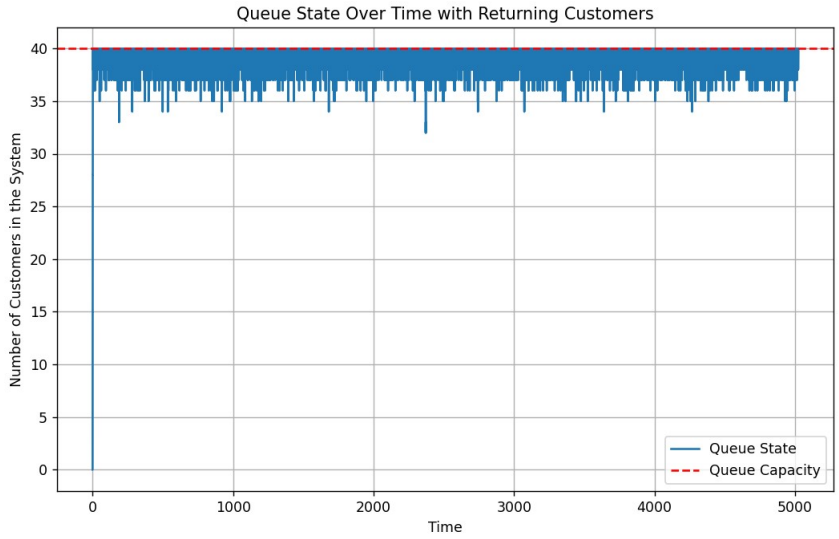


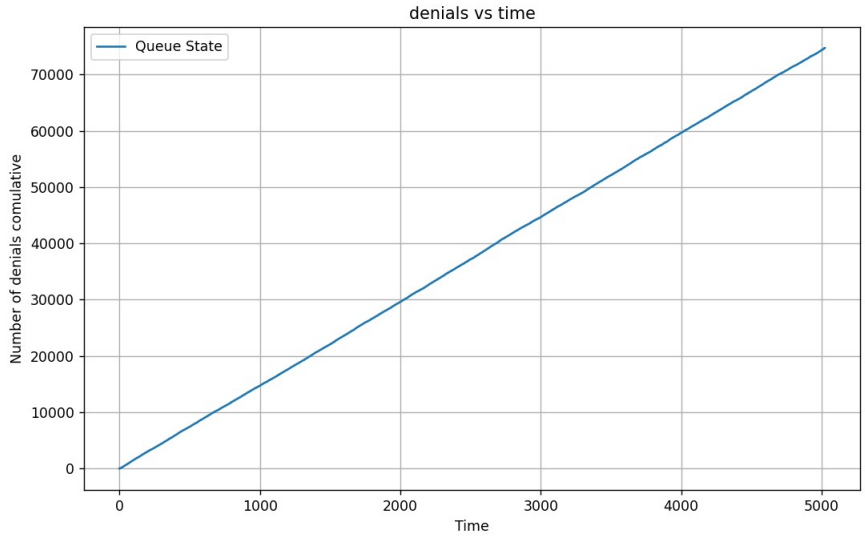



```
The rate of denials because the queue is full equals to: 0.16689
The expectation of the number of jobs in the queue: 35.0495212284778
```

we can see that about 16.69 percent of jobs have been denied. you see the track of denies verses time.

if $\lambda = 2\mu$:






```
The rate of denials because the queue is full equals to: 0.74698
The expectation of the number of jobs in the queue: 39.65575081469042
```

we can see that about 74.69 percent of jobs have been denied. you see the track of denies verses time.

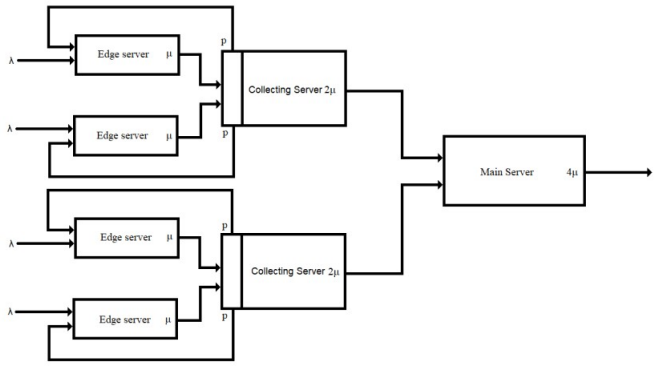
one observation is that rate of denies is constant verses time.

note that in this first section we did not get rid of primary data points and this can effect the values we obtained, because we always calculate and look for properties of system in steady state, then if we keep steady state data, it changes the total expectations, we will delete primary data points in next tasks.

Task2:Simulating of a simplified collecting-server and main-server

finding Expectation number of jobs in each que:

we want to simulate below system. 2 intermediate collecting servers and 1 main server, collecting data from 4 edge server.



in this task, we don't have much theory discussion, but we have to simulate mostly. any theory discussion that is need, is provided in the corresponding simulation section.

lets look at the code briefly:

in this part parameters have been defines.

```
# Parameters
mu_rate = 10
lambda_rate = 0.3 * mu_rate
capacity = 40
return_probability = 0.5
total_jobs = 100000

queues = [{"state": 0, "counter_denials": 0, "times": [0], "states": [0]} for _ in range(4)]
queue_a = {"state": 0, "times": [0], "states": [0]}
queue_b = {"state": 0, "times": [0], "states": [0]}
main_server = {"state": 0, "times": [0], "states": [0]}

time = 0
processed_jobs = 0
```

in a single loop, poison processes have been implement. first (like previous task) we have to determine the next event in the system.

```
while processed_jobs < total_jobs:

    arrival_times = [np.random.exponential(1 / lambda_rate) for _ in range(4)]
    service_times = [np.random.exponential(1 / mu_rate) if q["state"] > 0 else float('inf') for q in queues]

    service_time_a = np.random.exponential(1 / (2 * mu_rate)) if queue_a["state"] > 0 else float('inf')
    service_time_b = np.random.exponential(1 / (2 * mu_rate)) if queue_b["state"] > 0 else float('inf')

    service_time_main = np.random.exponential(1 / (4 * mu_rate)) if main_server["state"] > 0 else float('inf')

    # Determine next event and its time
    next_event_time = min(arrival_times + service_times + [service_time_a, service_time_b, service_time_main])
    time += next_event_time
```

now in a conditional structure, we handle the actions in each scenario.

if we have an arrival in the edge servers, if they are not full, the server accept the returning job. but if the server is full, it will be denied.

```
if next_event_time in arrival_times:
    queue_idx = arrival_times.index(next_event_time)
    processed_jobs += 1
    if queues[queue_idx]["state"] < capacity:
        queues[queue_idx]["state"] += 1
    else:
        queues[queue_idx]["counter_denials"] += 1
    queues[queue_idx]["times"].append(time)
    queues[queue_idx]["states"].append(queues[queue_idx]["state"])
```

now lets see what happens if a job got its servis in edge server:

```
elif next_event_time in service_times:
    queue_idx = service_times.index(next_event_time)
    queues[queue_idx]["state"] -= 1
    if random.random() < return_probability:
        if queues[queue_idx]["state"] < capacity:
            queues[queue_idx]["state"] += 1
    else:
        if queue_idx < 2:
            queue_a["state"] += 1
        else:
            queue_b["state"] += 1
    queues[queue_idx]["times"].append(time)
    queues[queue_idx]["states"].append(queues[queue_idx]["state"])
```

now it is time to see what is going on in collecting server:

```
elif next_event_time == service_time_a:
    queue_a["state"] -= 1
    main_server["state"] += 1
    queue_a["times"].append(time)
    queue_a["states"].append(queue_a["state"])

elif next_event_time == service_time_b:
    queue_b["state"] -= 1
    main_server["state"] += 1
    queue_b["times"].append(time)
    queue_b["states"].append(queue_b["state"])
```

and in the main server:

```
elif next_event_time == service_time_main:
    main_server["state"] -= 1
    main_server["times"].append(time)
    main_server["states"].append(main_server["state"])
```

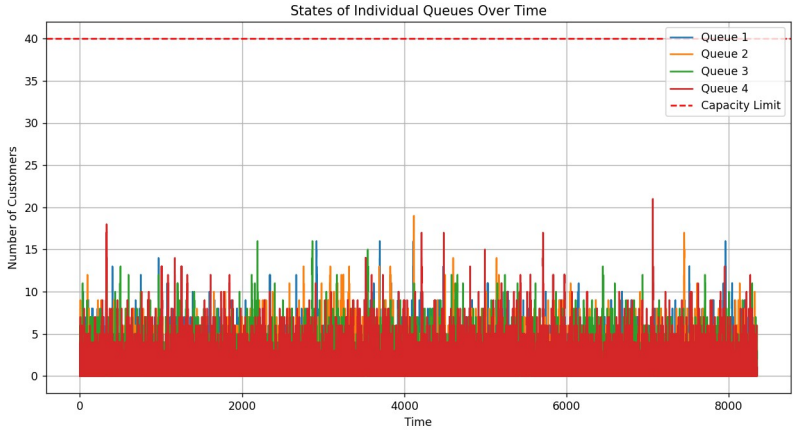
now we want to calculate parameters like Expectation of jobs in each server and rate of denials in edge servers. note that we do not include 5 first percentage of data:

```
start_index = int(0.05 * len(queues[0]["states"]))
for i, q in enumerate(queues):
    denials_rate = q["counter_denials"] / total_jobs
    mean_state = np.mean(q["states"][start_index:])
    print(f"Queue {i + 1}: Denial rate = {denials_rate:.4f}, Average state = {mean_state:.4f}")


start_index_a = int(0.05 * len(queue_a["states"]))
start_index_b = int(0.05 * len(queue_b["states"]))
start_index_main = int(0.05 * len(main_server["states"]))

mean_state_a = np.mean(queue_a["states"][start_index_a:])
mean_state_b = np.mean(queue_b["states"][start_index_b:])
mean_state_main = np.mean(main_server["states"][start_index_main:])

print(f"Queue A: Average state = {mean_state_a:.4f}")
print(f"Queue B: Average state = {mean_state_b:.4f}")
print(f"Main Server: Average state = {mean_state_main:.4f}")
```

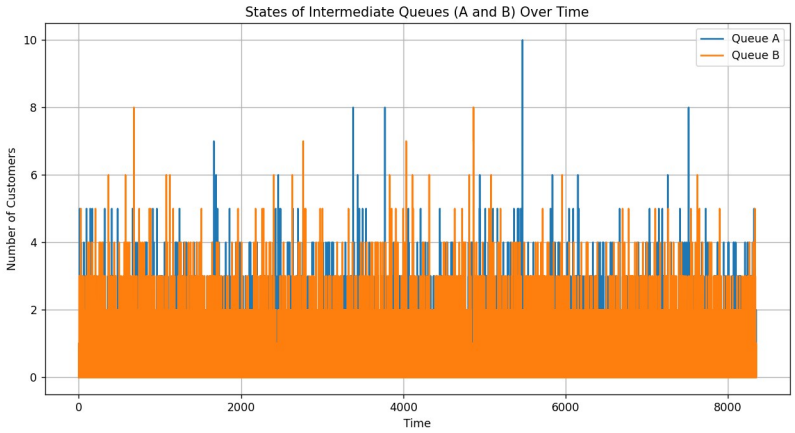
now we plot the results to see the effect of our system parameters λ, μ, P_{return} in the behavior of system:

for example for $\mu = 10$ and $\lambda = 0.3\mu$ and $P_{return} = 0.5$, lets see results:

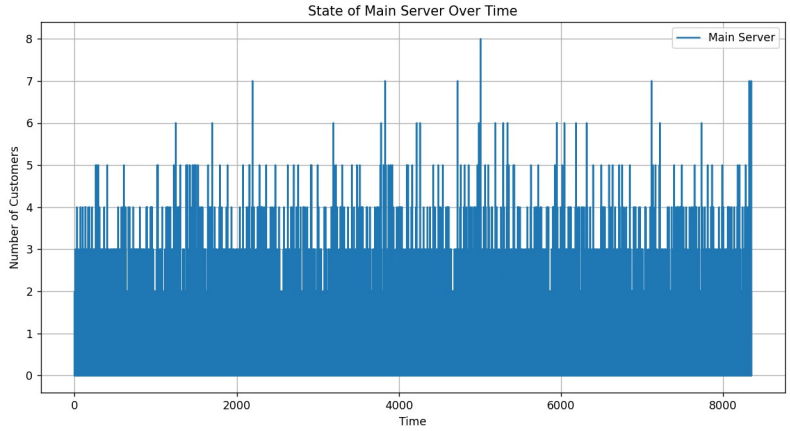
state of edge server:



state of collecting server:



state of main server:



here is statistical properties of system:

```
Queue 1: Denial rate = 0.0000, Average state = 2.1629
Queue 2: Denial rate = 0.0000, Average state = 2.1303
Queue 3: Denial rate = 0.0000, Average state = 2.1391
Queue 4: Denial rate = 0.0000, Average state = 2.2936
Queue A: Average state = 0.4221
Queue B: Average state = 0.4246
Main Server: Average state = 0.4234
```

lets talk about how we can calculate these statistics theoretically:

as edge servers are reached the capacity limitation, we can say that the throughput rate of them equals to their arrival rate $= \lambda$.

why? because we know that when $\lambda < \mu$, we have an steady state for the que, and then in steady steady state, we know that the jobs in que is almost constant the the rate of arrivals should be equal to departures.

collecting servers are simple que servers without any limitation and feedbacks, so the expectation jobs in them equals to :

note that the service rate of collecting server equals to 2μ and the input of each collecting server is cause of superposition of 2 edge server.

$$E[jobs = N] = \frac{2\lambda}{2\mu - 2\lambda}$$

so:

$$E[N] = \frac{2 * 3}{2 * 10 - 2 * 3} = \frac{3}{7} = 0.4285$$

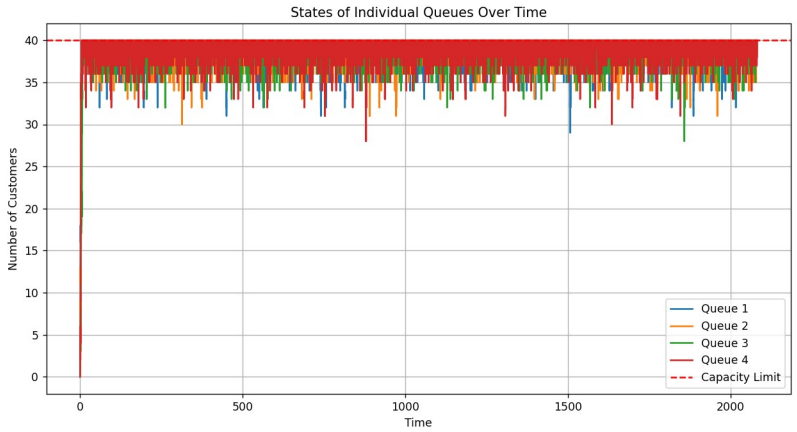
and for the main server (a simple que without capacity limitation and feedback):

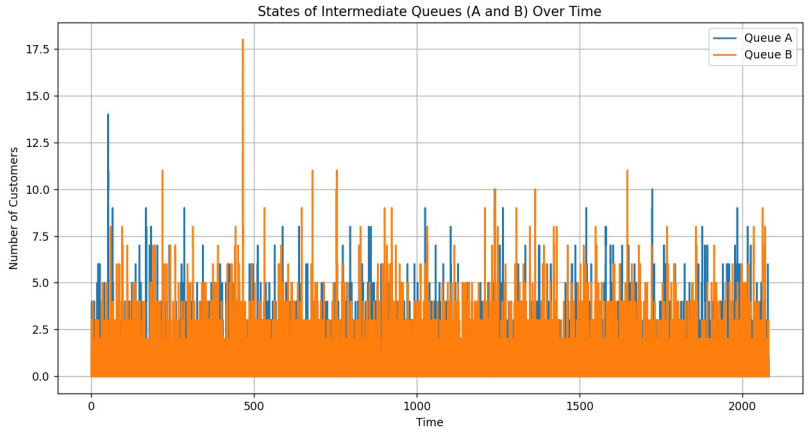
note that the service rate of main server equals to 4μ and the input of main server is cause of superposition of 4 edge server.

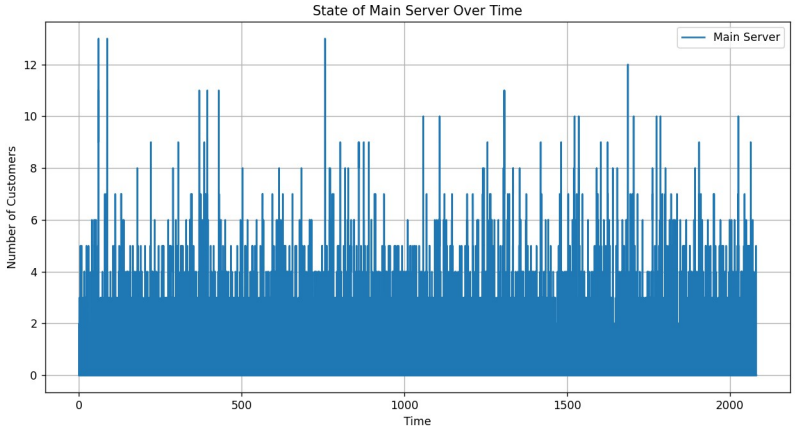
$$E[jobs = N] = \frac{4\lambda}{4\mu - 4\lambda} = 0.4285$$

it is so consistent with the simulation.

now we want to change $\lambda = 1.2\mu$:







and the statistics:

```
Queue 1: Denial rate = 0.1456, Average state = 39.2712
Queue 2: Denial rate = 0.1440, Average state = 39.2846
Queue 3: Denial rate = 0.1436, Average state = 39.2773
Queue 4: Denial rate = 0.1458, Average state = 39.2769
Queue A: Average state = 0.9916
Queue B: Average state = 1.0451
Main Server: Average state = 1.0414
```

as you can see about 14.56 percentage of jobs in the edge servers, have been denied.

here i want to make an intresting point:

the state of collecting servers and main server, never goes to infinity (as you change the λ and μ.

because the throughput of edge server is always less than its arrival rate and its service rate. as $\lambda \to \infty$ the throughput $\to \mu$ and vice versa. so as we set the service rate of collecting servers and main server, sum of the connecting edge server, the input rate of them in always less than output rate of them and this means there is always an steady state for these servers and they do not go to infinity.

in next task we will see the relation between throughput of servers and parameters of system.

note that in these two sections we simplified the returning process between collecting servers and edge servers and we simplified the memory of edge servers, it dosnt make a significant change in the results of furter tasks.

Task3:simulating our new system and finding the throughput rate in terms of λ, μ, P_{return} :

some insights about throughput rate:

Throughput Summary for a Queue with Capacity C:

- **Case 1:** $\lambda \gg \mu$
  - Description: Arrival rate is much larger than the service rate.
  - Behavior: The queue fills up quickly, and the system is almost always at capacity.
  - Throughput:
    $$\text{Throughput} \approx \mu$$
- **Case 2:** $\lambda > \mu$
  - Description: Arrival rate is slightly higher than the service rate.
  - Behavior: The system often reaches capacity, rejecting some arrivals.
  - Throughput:
    $$\text{Throughput} = \mu \cdot P_C$$
    where P_C is the probability of the system being full.
- **Case 3:** $\mu > \lambda$
  - Description: Service rate is higher than the arrival rate.
  - Behavior: The system is stable, with minimal blocking or rejection of arrivals.
  - Throughput:
    $$\text{Throughput} = \lambda \cdot (1 - P_C)$$
    For most practical systems, P_C is negligible, so:
    $$\text{Throughput} \approx \lambda$$
- **Case 4:** $\mu \gg \lambda$
  - Description: Service rate is much larger than the arrival rate.
  - Behavior: The server is often idle, and there is no queue buildup or rejection of arrivals.
  - Throughput:
    $$\text{Throughput} = \lambda$$

we can do some simplifications and summary the results as:
we assume:
if $\lambda > \mu : P_C \approx 1$
if $\mu > \lambda : P_C \approx 0$)
then:

$$throughput = T = Min(\lambda, \mu)$$

now we consider P_{return}:

$$\lambda_{eff} = \lambda + p \cdot Throughput$$

then:

$$T = Min(\lambda + pT, \mu)$$

$$T = \begin{cases} \frac{\lambda}{1-p} & \frac{\lambda}{1-p} < \mu \\ \mu & \frac{\lambda}{1-p} \geq \mu \end{cases}$$

it looks like we substitute λ with $\frac{\lambda}{1-p}$.
now we want to talk about P_C:

$$P_C = \frac{\rho^C}{\sum_{n=0}^{C} \rho^n}$$

Where:

$$\rho = \frac{\lambda}{\mu} \quad \text{(traffic intensity)}$$

$$\rho^C : \text{The probability of the system having } C \text{ jobs}$$

$$\sum_{n=0}^{C} \rho^n : \text{Normalization factor to ensure all probabilities sum to 1.}$$

Steps to Compute P_C: 1. Calculate $\rho = \frac{\lambda}{\mu}$. 2. Compute $\sum_{n=0}^{C} \rho^n$ using the formula for a geometric series:

$$\sum_{n=0}^{C} \rho^n = \frac{1 - \rho^{C+1}}{1 - \rho}, \quad \text{if } \rho \neq 1.$$

3. Substitute into P_C:

$$P_C = \frac{\rho^C (1 - \rho)}{1 - \rho^{C+1}}, \quad \text{if } \rho \neq 1.$$

now lets breakdown the code we wrote:

```
# Parameters
mu_rate = 10
lambda_rate = 9
capacity = 10
return_probability = 0.1
total_jobs = 100000

num_middle_servers = 3
num_edge_servers_per_middle = 5

edge_servers = [{"state": 0, "timeline": 0, "counter_denials": 0, "arrivals": 0, "times": [0], "states": [0]} for _ in range(num_edg
collecting_servers = [{"state": 0, "timeline": 0, "times": [0], "states": [0]} for _ in range(num_middle_servers)]
main_server = {"state": 0, "timeline": 0, "times": [0], "states": [0], "completed_jobs": 0}
```

now like before, in the while loop we generate some random variables and determine the next event:

```
arrival_times = [np.random.exponential(1 / lambda_rate) for _ in range(15)]

service_times = [np.random.exponential(1 / mu_rate) if q["state"] > 0 else float('inf') for q in queues]

service_times_collecting = [np.random.exponential(1 / (5*mu_rate)) if q["state"] > 0 else float('inf') for q in collecting_serv

service_time_main = np.random.exponential(1 / ( 15 * mu_rate)) if main_server["state"] > 0 else float('inf')

next_event_time = min(arrival_times + service_times + service_times_collecting + [service_time_main])
time += next_event_time
```

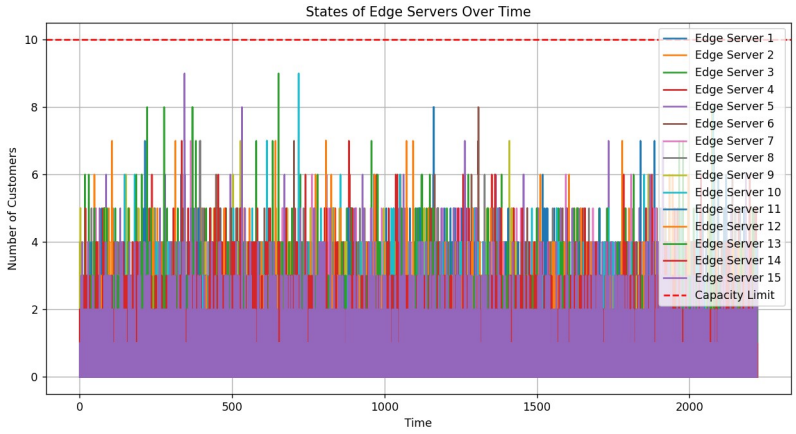
exactly like before we handle events in each situation:

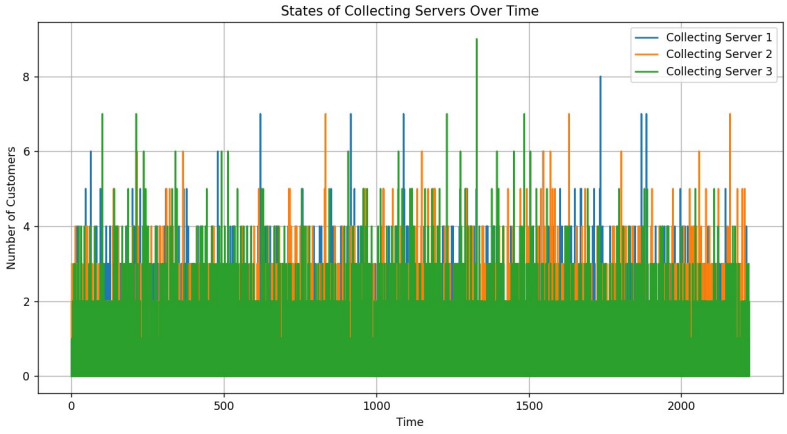
```
if next_event_time in arrival_times:

    queue_idx = arrival_times.index(next_event_time)
    processed_jobs += 1
    if queues[queue_idx]["state"] < capacity:
        queues[queue_idx]["state"] += 1
    else:
        queues[queue_idx]["counter_denials"] += 1
    queues[queue_idx]["times"].append(time)
    queues[queue_idx]["states"].append(queues[queue_idx]["state"])

elif next_event_time in service_times:

    queue_idx = service_times.index(next_event_time)
    queues[queue_idx]["state"] -= 1

    collecting_server_idx = queue_idx // 5
    collecting_servers[collecting_server_idx]["state"] += 1


    if random.random() < return_probability:
        if queues[queue_idx]["state"] < capacity:
            queues[queue_idx]["state"] += 1

    queues[queue_idx]["times"].append(time)
    queues[queue_idx]["states"].append(queues[queue_idx]["state"])
```

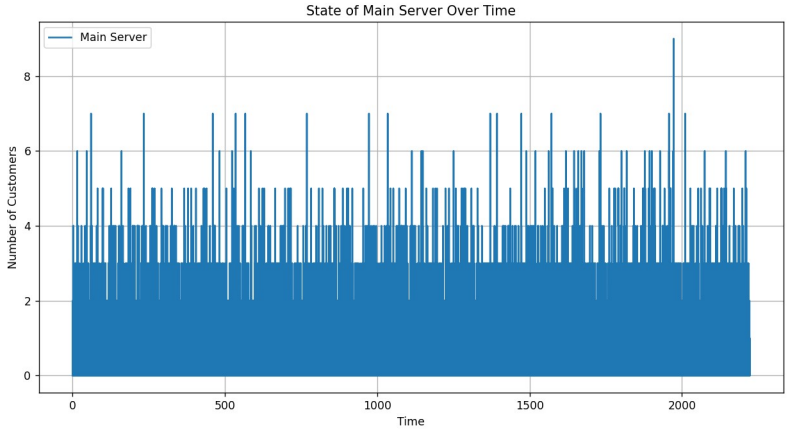
```
elif next_event_time in service_times_collecting:

    collecting_server_idx = service_times_collecting.index(next_event_time)
    collecting_servers[collecting_server_idx]["state"] -= 1
    main_server["state"] += 1
    processed_by_main_server += 1
    collecting_servers[collecting_server_idx]["times"].append(time)
    collecting_servers[collecting_server_idx]["states"].append(collecting_servers[collecting_server_idx]["state"])

elif next_event_time == service_time_main:
    main_server["state"] -= 1
    main_server["times"].append(time)
    main_server["states"].append(main_server["state"])
```

the similuation is finished now, and now we can calculate some properties and derive the results:

for $P = 0.1, k = 10, \lambda = 3, \mu = 10$:



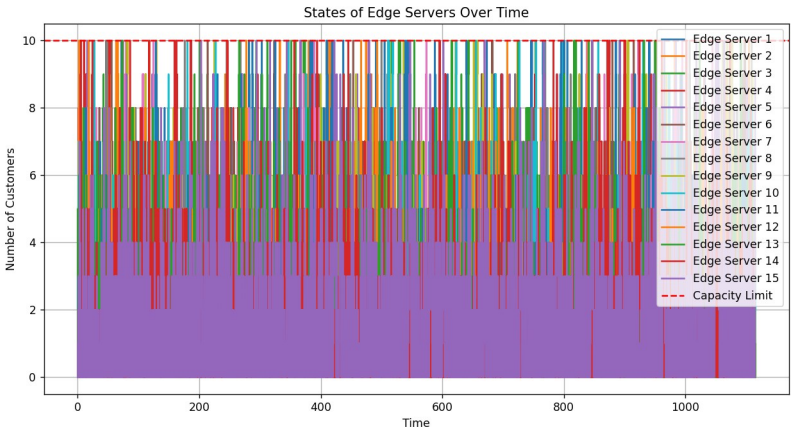


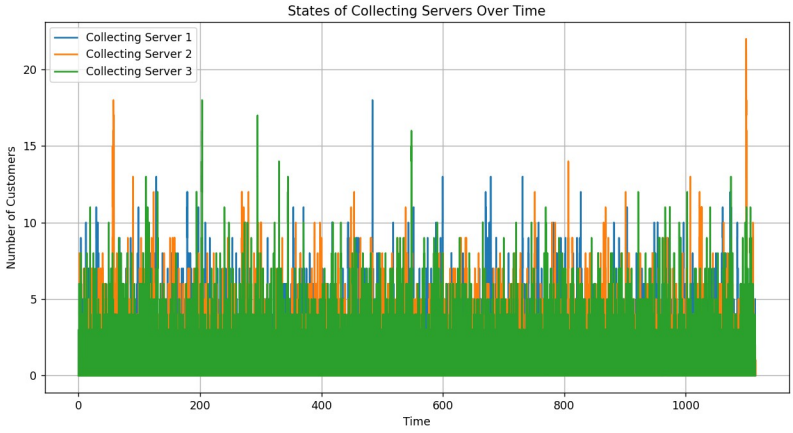

State of Main Server Over Time
Main Server
Number of Customers
Time
0
2
4
6
8
0
500
1000
1500
2000

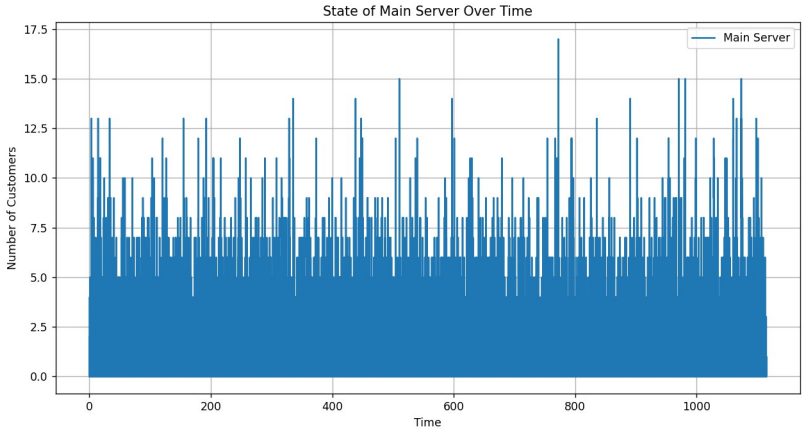
the statistics of system is:

```
Throughput of Main Server: 44.9913 jobs per unit time
Edge Server 1: Denial rate = 0.0000, Average state = 0.9527
Edge Server 2: Denial rate = 0.0000, Average state = 0.9137
Edge Server 3: Denial rate = 0.0000, Average state = 0.9529
Edge Server 4: Denial rate = 0.0000, Average state = 0.9393
Edge Server 5: Denial rate = 0.0000, Average state = 0.9105
Edge Server 6: Denial rate = 0.0000, Average state = 0.9157
Edge Server 7: Denial rate = 0.0000, Average state = 0.9153
Edge Server 8: Denial rate = 0.0000, Average state = 0.9250
Edge Server 9: Denial rate = 0.0000, Average state = 0.9359
Edge Server 10: Denial rate = 0.0000, Average state = 0.9340
Edge Server 11: Denial rate = 0.0000, Average state = 0.9115
Edge Server 12: Denial rate = 0.0000, Average state = 0.9415
Edge Server 13: Denial rate = 0.0000, Average state = 0.9468
Edge Server 14: Denial rate = 0.0000, Average state = 0.9363
Edge Server 15: Denial rate = 0.0000, Average state = 0.9317
Collecting Server 1: Average state = 0.4315
Collecting Server 2: Average state = 0.4451
Collecting Server 3: Average state = 0.4298
Main Server: Average state = 0.4324
```

and also for other cases (parameters for arrivals and ...:
for $P = 0.1, k = 10, \lambda = 6, \mu = 10$:



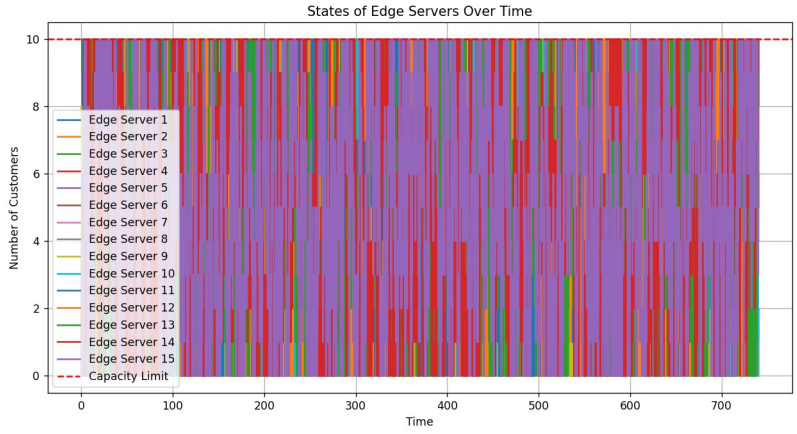


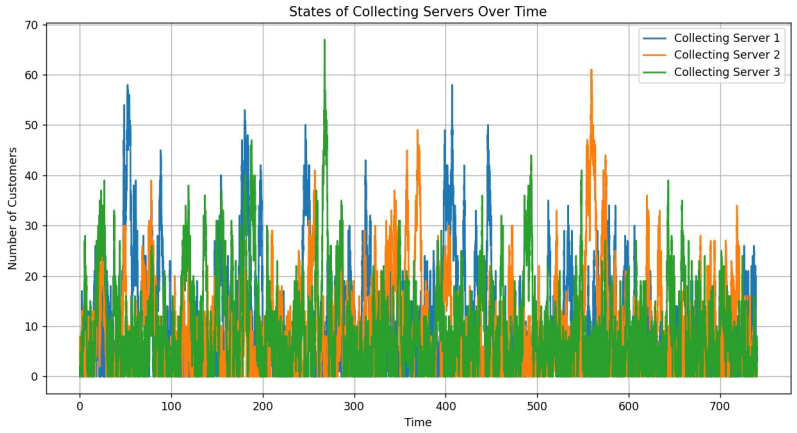


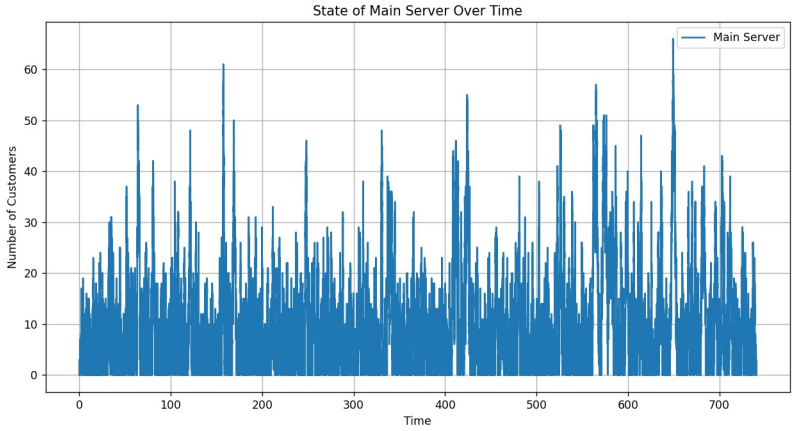
the statistics of system is:

```
Throughput of Main Server: 89.4653 jobs per unit time
Edge Server 1: Denial rate = 0.0002, Average state = 1.9026
Edge Server 2: Denial rate = 0.0002, Average state = 1.8807
Edge Server 3: Denial rate = 0.0003, Average state = 1.9245
Edge Server 4: Denial rate = 0.0002, Average state = 1.9818
Edge Server 5: Denial rate = 0.0001, Average state = 1.9540
Edge Server 6: Denial rate = 0.0002, Average state = 1.9017
Edge Server 7: Denial rate = 0.0001, Average state = 1.9007
Edge Server 8: Denial rate = 0.0001, Average state = 1.8362
Edge Server 9: Denial rate = 0.0001, Average state = 1.9473
Edge Server 10: Denial rate = 0.0002, Average state = 1.8897
Edge Server 11: Denial rate = 0.0003, Average state = 2.0030
Edge Server 12: Denial rate = 0.0002, Average state = 1.8757
Edge Server 13: Denial rate = 0.0002, Average state = 1.9987
Edge Server 14: Denial rate = 0.0001, Average state = 1.9287
Edge Server 15: Denial rate = 0.0002, Average state = 1.9942
Collecting Server 1: Average state = 1.4556
Collecting Server 2: Average state = 1.5276
Collecting Server 3: Average state = 1.5125
Main Server: Average state = 1.4578
```

for $P = 0.1, k = 10, \lambda = 6, \mu = 10$:







the statistics of system is:

```
Throughput of Main Server: 136.2914 jobs per unit time
Edge Server 1: Denial rate = 0.0070, Average state = 5.3931
Edge Server 2: Denial rate = 0.0055, Average state = 5.1585
Edge Server 3: Denial rate = 0.0063, Average state = 5.2844
Edge Server 4: Denial rate = 0.0054, Average state = 5.0195
Edge Server 5: Denial rate = 0.0061, Average state = 5.2198
Edge Server 6: Denial rate = 0.0057, Average state = 5.1221
Edge Server 7: Denial rate = 0.0062, Average state = 5.3921
Edge Server 8: Denial rate = 0.0062, Average state = 5.3471
Edge Server 9: Denial rate = 0.0066, Average state = 5.4061
Edge Server 10: Denial rate = 0.0068, Average state = 5.4103
Edge Server 11: Denial rate = 0.0062, Average state = 5.4414
Edge Server 12: Denial rate = 0.0063, Average state = 5.2531
Edge Server 13: Denial rate = 0.0052, Average state = 5.0382
Edge Server 14: Denial rate = 0.0053, Average state = 4.9812
Edge Server 15: Denial rate = 0.0065, Average state = 5.2453
Collecting Server 1: Average state = 10.3499
Collecting Server 2: Average state = 9.0548
Collecting Server 3: Average state = 9.2512
Main Server: Average state = 9.9250
```

we know that while we are increasing μ, we are increasing the return-jobs. and while μ is less thanλ, denial rate is high.

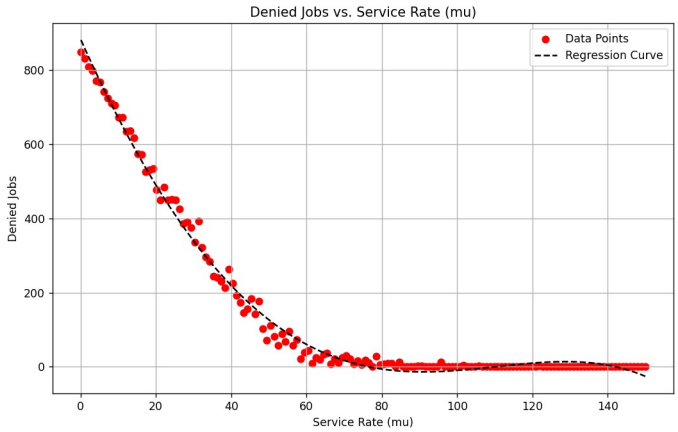
so if we define a cost function as below:

$$Cost = A \times denials + B \times returnjobs$$

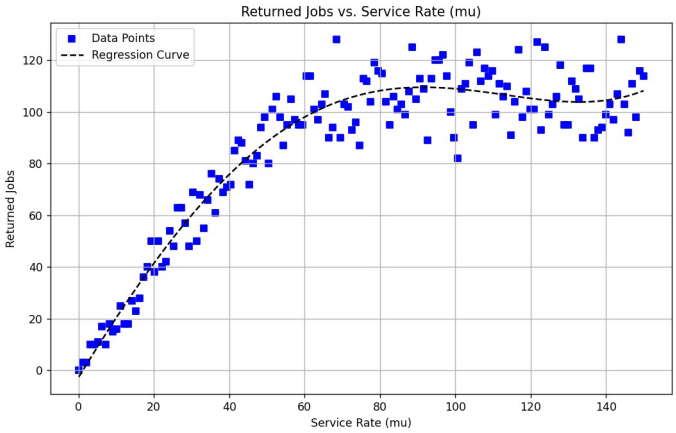
there is an optimum μ in the middle.
so we sweep μ:

```
# Parameters
lambda_rate = 50
capacity = 10
return_probability = 0.1
total_jobs = 1000
mu_values = np.linspace(1, 100, 100)

A = 1
B = 1

cost_values = []
denials_list = []
returns_list = []
```

lets plot denial-jobs verse service rate μ:

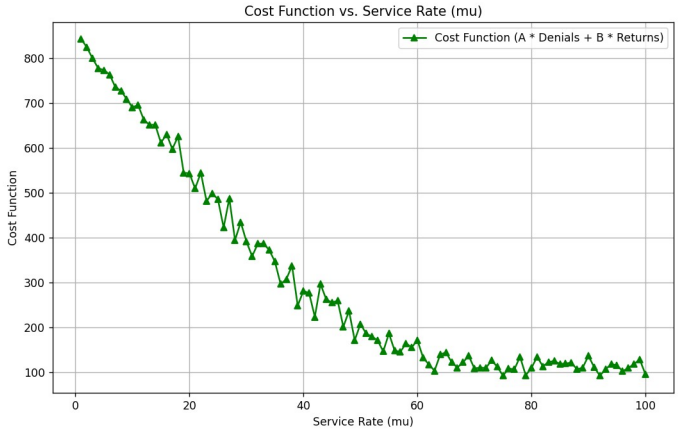


lets plot return-jobs verse service rate μ:



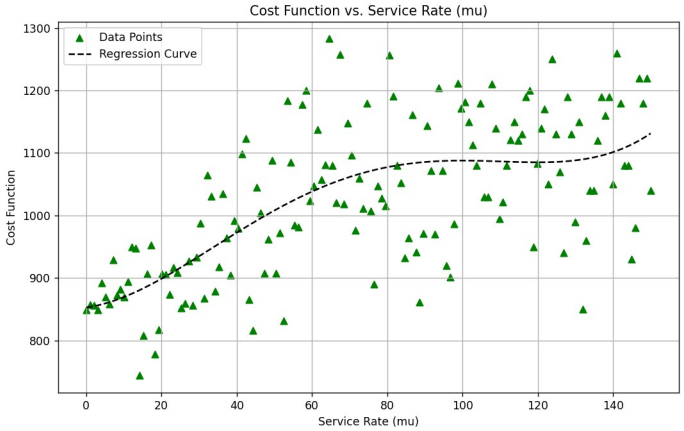
lets plot cost function verse service rate μ:

for A=B=1:

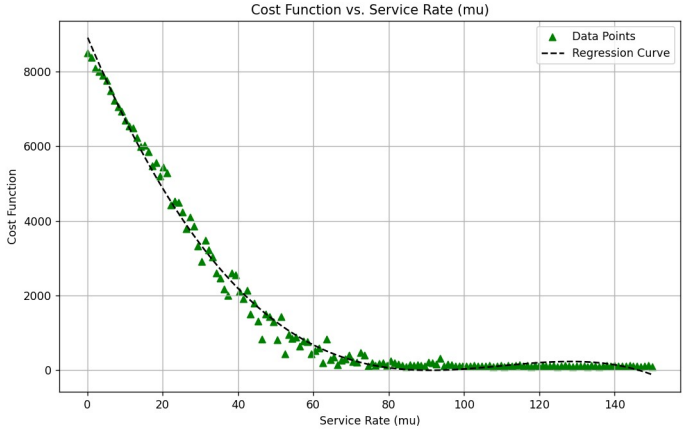


so we expect, if we increase B, then the cost function acts a ascending function and if we increase A, cost function acts as a descending function.

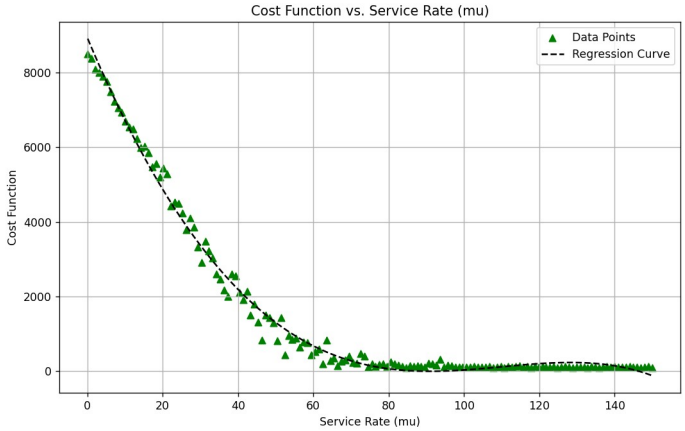
for A=1, B=10:

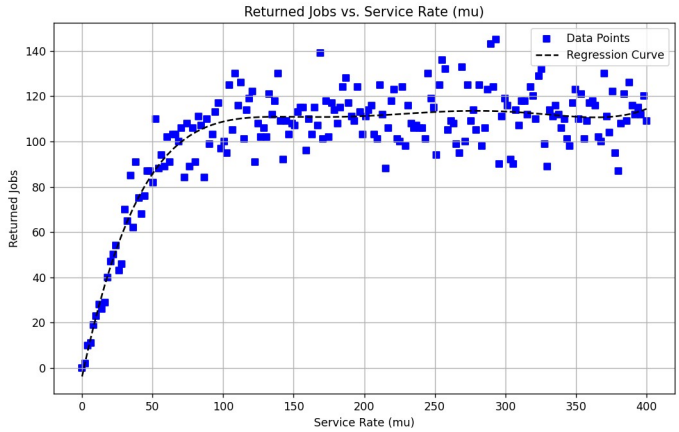


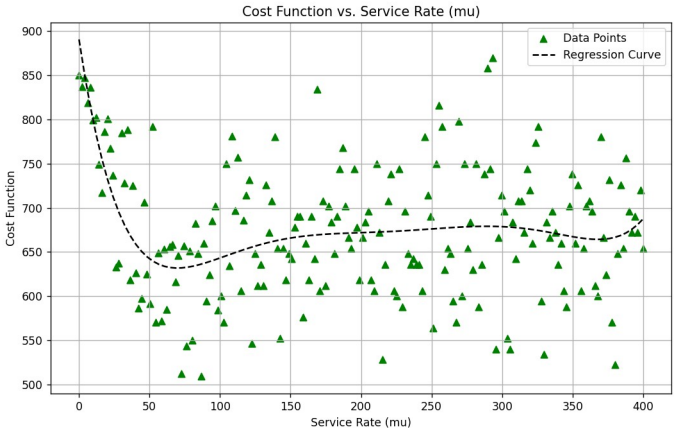
for A=10, B=1:



for A=1, B=6 and for mu=1 to 400:







now we can see that looks like there is an optimum in the middle for cost funtion. but as there is a high variance in return-job values we can not see it Vividly.

if we set p value in each iteration coresponding to μ like this:

$$P(\mu) = 2(\frac{1}{1+e^{-0.25\mu}} - 0.5)$$

lets take a theoriticall look at the problem.

$$C(\mu) = A \times P_{block} \times \text{number of jobs} + B \times N \times 2(\frac{1}{1+e^{-0.25\mu}} - 0.5)$$

we saw below equation before:

$$P[\text{a job being rejected}] = P_{block} = \frac{\lambda}{\mu} \times \frac{1}{N}$$

then:

$$C(\mu) = A \times N \times \frac{\lambda}{\mu} \times \frac{1}{N} + B \times N \times 2(\frac{1}{1+e^{-0.25\mu}} - 0.5)$$

we can easily see that P_{block} is decreasig with μ and P_{return} is increasing with μ by expectation.

now we can find derivative of C(μ with respect to μ.

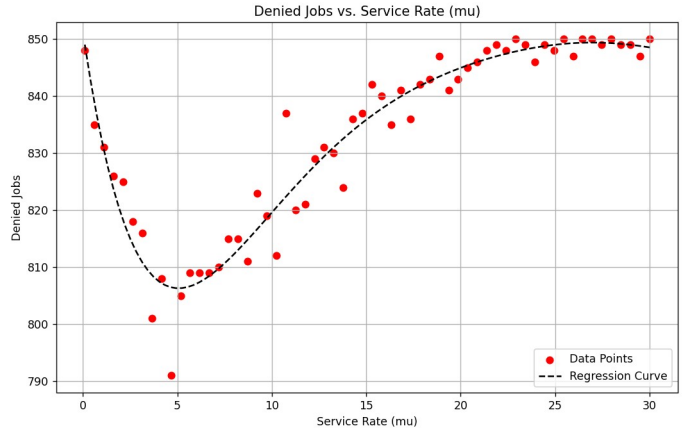
$$\frac{-A\lambda}{\mu^2} + 2BN(\frac{0.25e^{-0.25\mu}}{(1+e^{-0.25\mu})^2} = 0$$

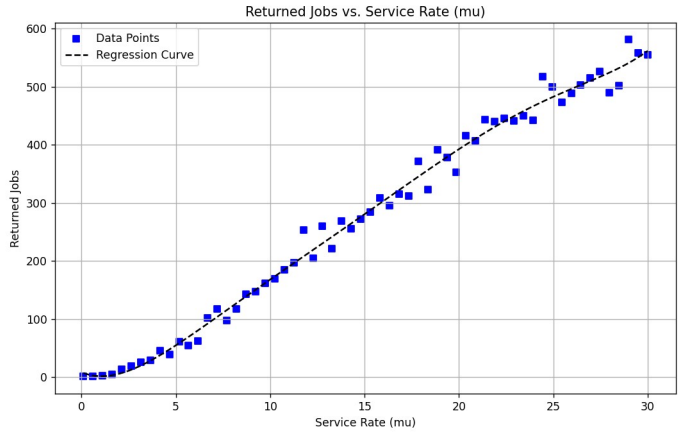
we can solve this equation for fixed A, B, λ, N with respect to μ.

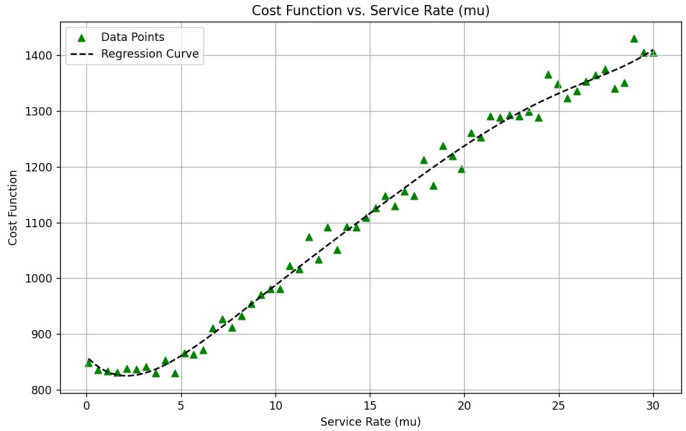
note that we can not obtain a closed-form answer but we will evaluate the answer in special cases.

we obtain results for different cases:

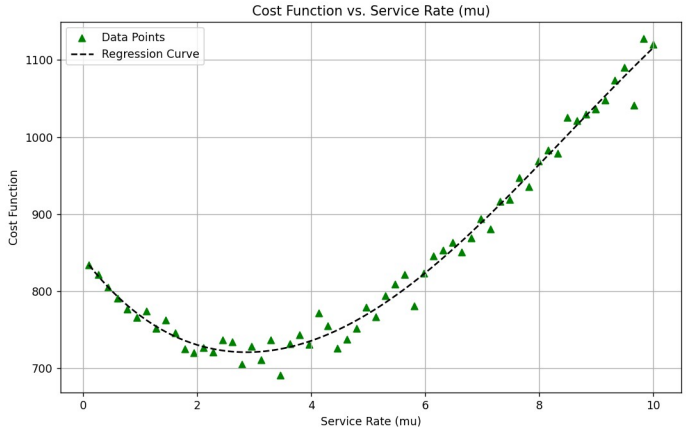
A=1, B=0.5:



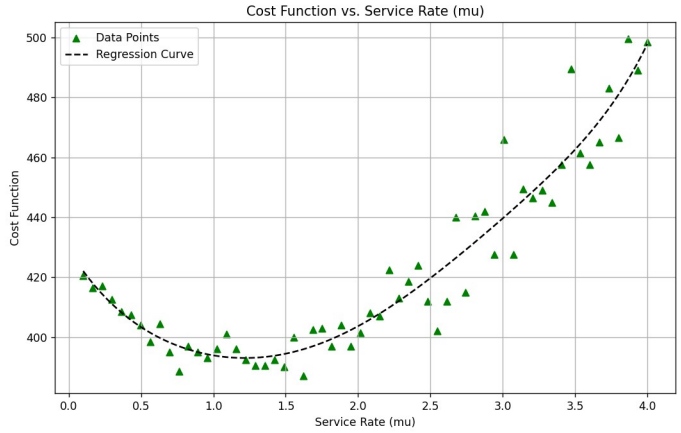




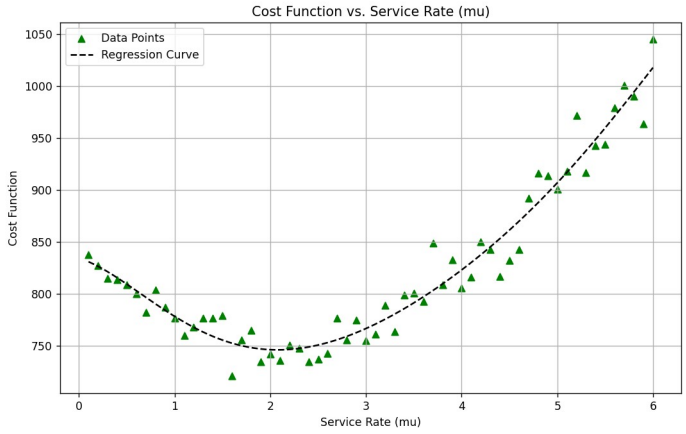
we have an optimum around $\mu = 3$ when $\lambda = 10$ is fixed.


Cost Function vs. Service Rate (mu)
Data Points
Regression Curve
Cost Function
Service Rate (mu)
1100
1000
900
800
700
0
2
4
6
8
10

A=0.5, B=1:


Cost Function vs. Service Rate (mu)
Data Points
Regression Curve
Cost Function
Service Rate (mu)
500
480
460
440
420
400
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0

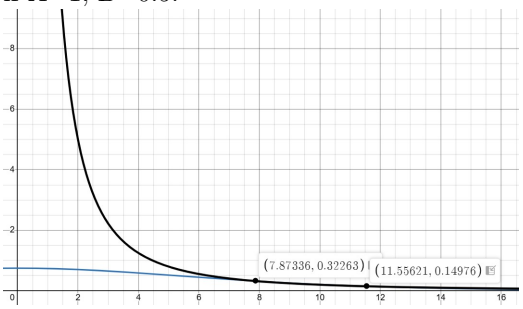
A=1, B=1:


Cost Function vs. Service Rate (mu)
Data Points
Regression Curve
Cost Function
Service Rate (mu)
1050
1000
950
900
850
800
750
0
1
2
3
4
5
6

now we evaluate the analytic equation that we obtained before:

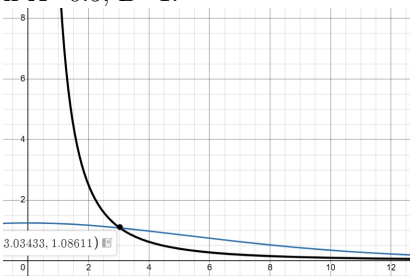
$$\frac{-A\lambda}{\mu^2} + 2BN(\frac{0.25e^{-0.25\mu}}{(1+e^{-0.25\mu})^2} = 0$$

if A=1, B=0.5:



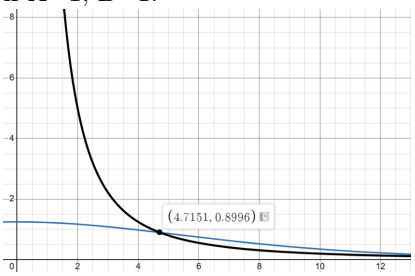
$$ArgMinC(\mu) = 7.873$$

if A=0.5, B=1:



$$ArgMinC((\mu) = 3.03$$

if A=1, B=1:



$$ArgMinC(\mu) = 4.71$$

if you compare to the simulation result, the result is very similar and the difference is because of variouse simplification that we had in the code and in the theory.

for example we assume simplified $P_{block}(\mu)$ as a linear relation.

we used a 6 degree polynomial regression from probabilistic data.(contains multiple rounding and errors and ....

and lots of simplification in our system logic in the code.

but the results are significantly close.