SIMULATION AND ANALYSIS OF THE G/G/1/K QUEUE

A Thesis

Presented to the Faculty of the Graduate School

of Cornell University

In Partial Fulfillment of the Requirements for the Degree of

Master of Science in Systems Engineering

by

Jingyao Tong

May 2025

ABSTRACT

This thesis analyzes single server queueing models with finite capacity, specifically focusing on analyzing the stationary probabilities for $M/G/1/K$, $G/M/1/K$, and $G/G/1/K$ queues. For $M/G/1/K$ and $G/M/1/K$, stationary probabilities are computed analytically through recursive formulations to validate the simulation results. Additionally, a simulated-based sensitivity analysis is also conducted on the $G/G/1/K$ queue to investigate how traffic intensity and buffer size affect critical performance measures, including average waiting time, queue length, and blocking probabilities. This analysis considers a wide range of interarrival and service distributions, including deterministic, exponential, gamma, geometric, hyperexponential, lognormal, truncated normal, uniform, and Weibull distributions.

BIOGRAPHICAL SKETCH

Jingyao Tong received a Bachelor's degree in Mathematics and a Bachelor's degree in Economics from Rutgers University. During graduate studies, he developed a strong interest in optimization, simulation, computer science, and queueing theory. These interests motivated his research in queueing systems. He is expected to graduate with a Master of Science in Systems Engineering from Cornell University in May 2025.

ACKNOWLEDGMENTS

I would like to express my deepest gratitude to my research advisor, Professor Jamol Pender, for his invaluable guidance and support throughout the research.

TABLE OF CONTENTS

NOTATIONS

λ	The arrival rate
μ	The service rate
P_K	The blocking probability for queue of size K
P_k	The probability of having k jobs in the queue
$P_{d,k}$	The probability of having k jobs in the queue at the departure instants
α_k	The probability of having k arrivals during a service time
β_k	The probability of having k service completion during an interarrival period
ρ	The traffic intensity

1 Introduction

Queues are prevalent in every scenario where the services are provided, from telecommunications to customer service. Queueing theory plays a crucial role in optimizing resource allocation and improving system efficiency. By analyzing the key performance metrics such as waiting time, queue length, and blocking probability, service providers can significantly improve the customer experience by minimizing their waiting times and profits by maximizing the number of successful service completions.

This thesis focuses on analyzing steady states probabilities of queues with finite capacities. For queues such as $M/G/1$, which assumes Poisson arrivals and general service times with unlimited buffer capacity, the model does not account for the space limitations when applying to practical systems. In contrast, $M/G/1/K$ queue with a finite buffer size is more suitable for real-world environments where the number of jobs can be wait in the system is restricted, as noted by Bose in [2]. In this case, a finite buffer size ensures that arriving customers are being blocked and leaved without service when system is full. This blocking effect reduces potential throughput and can severely worsen the system performance. Therefore, it is necessary to determine an appropriate buffer size for each combination of interarrival and service times distributions based on different scenarios.

While many studies utilized moment approximation methods to evaluate the queueing performance, these approaches could have many limitations when applying to real life situations. Smith [4] proposed a two-moment based method that

accurately approximates performance measures for $M/G/1/K$ queue. However, in practice, the estimations of mean and variance can be unreliable since large size samples are not always available. This challenge becomes even more complex when analyzing the $G/M/1/K$ and $G/G/1/K$ queues. Instead, this study employs a simulation-based approach which specifies the interarrival and service distributions. For $M/G/1/K$ and $G/M/1/K$ queues, stationary probabilities are also calculated through recursive formulations and used to verify the results from simulation, ensuring consistency in the performance evaluations.

2 Analysis of the M/G/1/K Queue

The $M/G/1/K$ queueing system represents a single server model with a finite buffer size K, where arrivals follow a Poisson process, and service times follow a general distribution.

In this section, a specific case of $M/G/1/K$ is examined in detail. The stationary probabilities are first derived using a recursive formulation, and then the results are used to validate against simulation-based results.

2.1 M/D/1/K

One specific type of service distributions considered in this study is the deterministic distribution. According to Bose's findings in [2], the stationary probabilities for

queue lengths $k = 0, 1, ..., K - 1$ are expressed as follows:

$$P_k = \frac{1}{P_{d,0} + \rho} P_{d,k} \tag{1}$$

and for the maximum queue length $k = K$,

$$P_K = 1 - \frac{1}{P_{d,0} + \rho} \tag{2}$$

and for $k = 0, 1, ..., K - 2$, terms $P_{d,k}$ are defined recursively:

$$P_{d,k} = P_{d,0}\alpha_k + \sum_{j=1}^{k+1} P_{d,j}\alpha_{k-j+1} \tag{3}$$

where α_k, the probability of having k jobs enter the queue during a period of service time, are given by:

$$\alpha_k = \int_{t=0}^{\infty} \frac{(\lambda t)^k}{k!} e^{-\lambda t} b(t) dt \tag{4}$$

For deterministic service distribution with mean service time Δ,

$$\alpha_k = \frac{(\lambda \Delta)^k}{k!} e^{-\lambda \Delta} \tag{5}$$

In order to find P_k, $P_{d,k}$ needs to be calculated first. For k = 0, 1,

$$\begin{aligned} P_{d,0} &= P_{d,0}\alpha_0 + P_{d,1}\alpha_0 \\ P_{d,1} &= 1 - P_{d,0} \end{aligned}$$

then by (5), we can get

$$\alpha_0 = e^{-\lambda\Delta}$$

Substituting α_0 into $P_{d,0}$ and get

$$\begin{aligned} P_{d,0} &= e^{-\lambda\Delta} \\ P_{d,1} &= 1 - e^{-\lambda\Delta} \end{aligned}$$

Thus, the stationary probabilities P_0, P_1, and P_2 can be calculated as:

$$\begin{aligned} P_0 &= \frac{e^{-\lambda\Delta}}{e^{-\lambda\Delta} + \lambda\Delta} \\ P_1 &= \frac{1 - e^{-\lambda\Delta}}{e^{-\lambda\Delta} + \lambda\Delta} \\ P_2 &= 1 - \frac{1}{e^{-\lambda\Delta} + \lambda\Delta} \end{aligned}$$

It becomes evident that even for small buffer capacity such as $K = 2$, manual calculation involves a lot of steps and is prone to make mistakes. Moreover, as K increases, it is very difficult to list and calculate $P_{d,k}$ and P_k. As a result, this study employs Python to systematically generate and solve the recursive equations, significantly enhancing computational accuracy and efficiency in determining stationary probabilities.

2.2 Calculation in Python

The calculation of stationary probabilities P_k at an arbitrary time instant depends on previously obtained probabilities $P_{d,k}$, the stationary probabilities at the departure instants. To compute the stationary probabilities P_K from $k = 0, ..., K$, three procedures are needed and will be described in the following subsections.

2.2.1 Calculating the Probabilities at Departure Instants

The initial step involves obtaining the probability of having k job arrivals during a single service time at the departure instants. Equation (3) computes $P_{d,k}$ for $k = 0, ..., K-2$, requiring values from α_0 to α_{K-2}.

Equation (4) that defines the computation of α_k is implemented in Python as the function *alpha(k, typeService)*, which takes an integer k and a string specifying the service distribution type. Iteratively calling this function with $k = 0, ..., K-2$ generates a list of α_k values.

After obtaining the α values, we can then substitute them into equation (3) to compute $P_{d,k}$ values. However, each stationary probability at the departure instant $P_{d,k}$ is expressed in terms of other $P_{d,j}$, creating a cyclic relationship among them. To solve this, the system is formulated as a set of simultaneous linear equations as

below:

$$
\begin{aligned}
(1-\alpha_0)P_{d,0} + (-\alpha_0)P_{d,1} + 0 \cdot P_{d,2} + 0 \cdot P_{d,3} + ... + 0 \cdot P_{d,K-1} &= 0 \\
(-\alpha_1)P_{d,0} + (1-\alpha_1)P_{d,1} + (-\alpha_0)P_{d,2} + 0 \cdot P_{d,3} + ... + 0 \cdot P_{d,K-1} &= 0 \\
(-\alpha_2)P_{d,0} + (-\alpha_2)P_{d,1} + (1-\alpha_1)P_{d,2} + (\alpha_0)P_{d,3} + ... + 0 \cdot P_{d,K-1} &= 0
\end{aligned}
$$

...

...

...

$$
P_{d,0} + P_{d,1} + P_{d,2} + P_{d,3} + ... + P_{d,K-1} = 1
$$

In order to efficiently construct and solve it in Python, this system of equations need to be converted into matrix form. As observed, except for the first and last equation where the coefficient of $P_{d,0}$ equals $1 - \alpha_0$ and 1 respectively, all of the other coefficients of $P_{d,0}$ are equal to $-\alpha_i$, where i corresponds to the i^{th} row.

After determining the coefficient for the first column ($P_{d,0}$), a general pattern was found on the remaining entries. The $(i, j)^{th}$ coefficient for row i and column j is a nonzero term if $j \leq i + 1$, and it equals to zero otherwise. For instance, the second row has nonzero coefficients for $P_{d,0}$, $P_{d,1}$, and $P_{d,2}$, but all entries beyond $P_{d,2}$ have coefficients of zero.

In addition, following equation (4), the coefficients are defined as α_{k-j+1} except

for the first and diagonal entries. The coefficients of the first entries have been discussed previously, and the entries for diagonal entries (i, i) equals $1 - \alpha_0$ when $i = 1$ and have the same coefficient of $1 - \alpha_1$ when $i \geq 2$.

Finally, the last equation ensures all the probabilities sum up to 1, so the coefficient for each $P_{d,k}$ is set to 1. Once the coefficients for all $P_{d,k}$ have been determined, the resulting system of equations can be expressed in matrix form as $A \cdot x = B$ and solve in Python, where:

$$A = \begin{bmatrix} 1-\alpha_0 & -\alpha_0 & 0 & 0 & \cdots & 0 \\ -\alpha_1 & 1-\alpha_1 & -\alpha_0 & 0 & \cdots & 0 \\ -\alpha_2 & -\alpha_2 & 1-\alpha_1 & -\alpha_0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \ddots & \vdots \\ & & & & \ddots & \vdots \\ 1 & 1 & 1 & \cdots & \cdots & 1 \end{bmatrix}, B = \begin{bmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ \vdots \\ 1 \end{bmatrix}$$

2.2.2 Calculating the Probabilities at Arbitrary Instants

After obtaining $P_{d,k}$ for $k = 0, ..., K - 1$, the stationary probabilities at arbitrary instants P_k can then be calculated by substituting these values into equation (1). A Python function $P(k)$ is created for this purpose, which accepts an integer k and returning the corresponding stationary probability P_k. Specifically, the function uses equation (1) when $0 \leq k \leq K - 1$ and uses equation (2) when $k = K$ for blocking probability.

2.3 Simulation in Python

2.3.1 Generating Arrival and Service Times

To simulate the queueing system, it is necessary to generate interarrival and service times for each job. For this $M/G/1/K$ model, a Python function is implemented to accept the service distribution type, arrival rate (λ), service rates (μ), and the number of jobs as input parameters. Then the function generates and returns a list of arrival times following an exponential distribution and a list of service times corresponding to the specified distribution type.

2.3.2 Simulation Process

With each job's arrival time and service time generated, we can proceed to simulate and record the probabilities of having $0 \rightarrow K$ jobs in the queue.

For each job, it will either be accepted if the queue is not full or blocked if the queue is full. When a job arrives, we first check how many jobs ahead of it are still in the queue. There will be two scenarios:

- The queue is not full:
  - The queue is empty ($queue\ size = 0$)
  - The queue is not empty ($0 < queue\ size < K$)
- The queue is full ($queue\ size = K$):

If the queue is empty in the first scenario, the job's arrival time is the service starting time. Otherwise, it will be the maximum value between the departure time of the previous job and the arrival time of the current job. Then we update the job departure time by adding the service time to the service starting time and set the blocked status to $False$. The waiting time will be the difference between the job's service starting time and its arrival time.

In the second scenario where the queue is full upon the current job's arrival, the job is blocked, and the blocked status will be set to $True$.

Once we run the simulation and have the queue length upon arrival for each job, we can count the occurrences of $0, ..., K$ and divide each of them by the total number of jobs to get the P_k.

2.4 Validation

In this section, both analytical calculations and simulation models are executed to compare the results for different types of service distribution. To ensure consistent performance evaluation, the buffer size K is fixed at 5, and the total simulating number of jobs is set to $100,000$. This setup enables a fair comparison of stationary probabilities across different service time distributions.

2.4.1 Deterministic

The service time is fixed at $t = \frac{1}{\mu}$, then the PDF $b(t)$ used in equation (4) is a delta function:

$$b(t) = \delta(t - \frac{1}{\mu})$$

where is zero everywhere for $t \neq \frac{1}{\mu}$.

k	0	1	2	3	4	5
Calculated P_k	0.3117	0.316	0.1969	0.1049	0.0537	0.0167
Simulated P_k	0.3115	0.3169	0.1966	0.1055	0.0533	0.0163

Table 1: Deterministic Service Distribution $\lambda = 0.7$, $\mu = 0.5$

2.4.2 Exponential

When service times follows a exponential distribution, the queue becomes a $M/M/1/K$ queue. The PDF $b(t)$ is:

$$b(t) = \mu e^{-\mu t}$$

k	0	1	2	3	4	5
Calculated P_k	0.3654	0.2436	0.1624	0.1083	0.0722	0.0481
Simulated P_k	0.3648	0.2455	0.1631	0.1079	0.0716	0.0471

Table 2: Deterministic Service Distribution $\lambda = 0.8$, $\mu = 1.2$

2.4.3 Gamma

The PDF used in calculation was:

$$b(t) = \frac{\mu^a t^{a-1} e^{-\mu t}}{\Gamma(a)}$$

for $t \geq 0$, where

$$\Gamma(a) = \int_0^\infty t^{a-1} e^{-t} dt$$

k	0	1	2	3	4	5
Calculated P_k	0.4112	0.2837	0.1588	0.0840	0.0437	0.0186
Simulated P_k	0.4181	0.2835	0.1562	0.0823	0.0422	0.0176

Table 3: Gamma Service Distribution $\lambda = 0.6$, $\mu = 1$, $shape = 2$

2.4.4 Geometric

For this discrete distribution, let λ be the interarrival rate and μ be the probability that a job successfully finishes service in a time interval, where $0 \leq \mu \leq 1$ in this case. Then the PMF is given by:

$$P(X = k) = (1 - \mu)^{k-1} \cdot \mu, \quad k = 1, 2, 3, ...$$

In this case, the probability of having k arrivals during a service time becomes:

$$\begin{aligned}\alpha_j &= \sum_{k=1}^{\infty} \frac{(\lambda k)^j}{j!} e^{-\lambda k} (1-\mu)^{k-1} \mu \\ &= \frac{\mu e^{-\lambda} \lambda^j}{j!} \sum_{k=1}^{\infty} k^j [(1-\mu) e^{\lambda}]^{k-1}\end{aligned}$$

Let $p = [(1-\mu)e^{\lambda}]$, then the summation becomes:

$$\sum_{k=1}^{\infty} k^j p^{k-1} = \frac{1}{p} Li_{-j}(p)$$

where $Li_{-j}(p)$ is the polylogarithm function of p with order $-j$, and this expression is also equivalent to $\frac{1}{1-p}$ of the j^{th} moment of $Geo(1-p)$. Thus, α_j becomes:

$$\alpha_j = \frac{\mu e^{-\lambda} \lambda^j}{p j!} Li_{-j}(p), \quad \text{where } p = (1-\mu)e^{\lambda}$$

k	0	1	2	3	4	5
Calculated P_k	0.0027	0.0100	0.0314	0.0973	0.3017	0.5568
Simulated P_k	0.0025	0.0095	0.0303	0.0958	0.3009	0.5610

Table 4: Geometric Service Distribution $\lambda = 0.9$, $\mu = 0.4$

2.4.5 Hyperexponential

For hyperexponential service distribution, a mixture of $d = 20$ exponential distributions is used, where each of them is assigned with a randomly generated probabilities p_i such that $\sum_{i=1}^{d} p_i = 1$ and uniformly sampled μ_i between 0.15 and 1.5 for

i^{th} from $i = 1, ..., 20$. Then the PDF is

$$\sum_{i=1}^{d} p_i \cdot \mu_i e^{-\mu_i t}$$

k	0	1	2	3	4	5
Calculated P_k	0.0374	0.0608	0.1003	0.1664	0.2768	0.3583
Simulated P_k	0.0406	0.0650	0.1002	0.1565	0.2485	0.3892

Table 5: Hyperexponential Service Distribution, $\lambda = 1.5$

2.4.6 Log-Normal

The PDF is given by:

$$b(t) = \frac{1}{t\sigma\sqrt{2\pi}} e^{-\frac{(\ln t - \mu)^2}{2\sigma^2}}$$

k	0	1	2	3	4	5
Calculated P_k	0.2360	0.2113	0.1723	0.1434	0.1217	0.1154
Simulated P_k	0.2333	0.2115	0.1735	0.1427	0.1210	0.1180

Table 6: Log-Normal Service Distribution $\lambda = 0.95$, $\mu = 1.1$, $\sigma = 1$

2.4.7 Truncated Normal

In this case, a truncated normal distribution is used instead of a standard normal distribution since the normal distribution is unbounded and allows negative times.

$$b(t) = \frac{\varphi(\frac{t-\mu}{\sigma})}{\sigma(\phi(\frac{b-\mu}{\sigma}) - \phi(\frac{a-\mu}{\sigma}))}$$

Therefore, to ensure all times are positively generated, the distribution is truncated at zero. Given the desired mean μ_{trunc} and standard deviation σ_{trunc}, it is necessary to find the corresponding μ and σ of the underlying normal distribution, which ensures that the desired statistics are yielded for the truncated normal distribution. According to [3], the mean and variance of a truncated normal distribution can be expressed in terms of the underlying normal distribution as follows:

$$\begin{aligned} \mu_{trunc} &= \mu + \sigma \cdot \frac{\varphi(a) - \varphi(b)}{\phi(b) - \phi(a)} \\ \sigma^2_{trunc} &= \sigma^2 + \sigma^2 \cdot \frac{a \cdot \varphi(a) - b \cdot \varphi(b)}{\phi(b) - \phi(a)} - \sigma^2 \cdot \frac{(\varphi(a) - \varphi(b))^2}{(\phi(b) - \phi(a))^2} \end{aligned}$$

where

$$\begin{aligned} a &= \frac{0 - \mu}{\sigma} \\ b &= \frac{\infty - \mu}{\sigma} \end{aligned}$$

and $\varphi(\cdot)$ and $\phi(\cdot)$ denote the standard normal PDF and the standard normal CDF respectively.

For $\lambda = 0.65$, desired $\mu = 0.8$, and $\sigma = 1.1$, solving this system for μ and σ got -4.1561 for μ and 2.8227 for σ. Substituting them into equation (4) to compute the α_k for $k = 0, 1, ..., 4$, the results are $[0.5365, 0.2587, 0.1181, 0.0514, 0.0214]$. This process is implemented in Python function *mu_sigma_solver*.

Then use equation (3) and equation (1) to solve for the stationary probabilities $P_0, P_1, ..., P_5$.

k	0	1	2	3	4	5
Calculated P_k	0.2488	0.2252	0.1863	0.1482	0.1160	0.0755
Simulated P_k	0.2514	0.2269	0.1870	0.1467	0.1131	0.0749

Table 7: Truncated Normal Service Distribution $\lambda = 0.65$, desired $\mu = 0.8$, and $\sigma = 1.1$

2.4.8 Uniform

$$b(t) = \frac{1}{b-a} \quad \text{for } t \in [a, b]$$

To ensure the service times follow a uniform distribution and have a mean of $\frac{a+b}{2} = \frac{1}{\mu}$, it is necessary to make $b = \frac{2}{\mu} - a$. Then the traffic intensity $\rho = \lambda \cdot \mathbb{E}[S] = \lambda \cdot \frac{a+b}{2}$.

k	0	1	2	3	4	5
Calculated P_k	0.0081	0.0276	0.0694	0.1652	0.3910	0.3387
Simulated P_k	0.0082	0.0276	0.0705	0.1655	0.3905	0.3376

Table 8: Uniform Service Distribution $\lambda = 1.2$, $a\_uniform = 1$, $b\_uniform = 1.5$

2.4.9 Weibull

The PDF is given by:

$$b(t) = cx^{c-1}e^{-x^c}$$

where c is the shape parameter.

k	0	1	2	3	4	5
Calculated P_k	0.3246	0.2690	0.1824	0.1162	0.0725	0.0352
Simulated P_k	0.3225	0.2704	0.1817	0.1173	0.0730	0.0351

Table 9: Weibull Service Distribution $\lambda = 0.7$, $\mu = 1$, $shape\_weibull = 1.5$

3 Analysis of the G/M/1/K Queue

Unlike the arrivals of $M/G/1/K$ queue, those of $G/M/1/K$ queue no longer possess the memoryless property, which introduces more complexity in analyzing the system. $G/M/1/K$ queueing system is a single server model with finite buffer size K, where the interarrival times follow a general distribution, and service times are exponentially distributed. This section presents both analytical approach and simulated model to explore the stationary probabilities under different interarrival time distributions.

3.1 Calculation in Python

According to [1], the stationary probability P_k for $k = 0, ..., K$ in $G/M/1/K$ queue is given by:

$$P_j = \sum_{i=0}^{K} P_i \cdot P_{ij} \tag{6}$$

where P_{ij} is the transition probability from state i to j. In order to calculate all the P_k values, the transition probabilities have to be first calculated. As defined in [1],

$$P_{ij} = \begin{cases} \beta_{i-j+1}, & j > 0 \\ \sum_{r=i+1}^{\infty} \beta_r, & j = 0 \end{cases} \tag{7}$$

where β_j represents the number of service completion during a period of time.

It is defined as:

$$\beta_j = \int_{t=0}^{\infty} \frac{(\mu t)^j}{j!} e^{-\mu t} a(t) dt \tag{8}$$

where $a(t)$ denotes the PDF function of the interarrival time distribution, and μ is the service rate. It is noticeable that this equation exhibits Poisson-like behavior, despite that the arrivals are general and not following Poisson process. This is due to the exponential service times distribution, which ensures that the number of completions in the time interval still follows a Poisson distribution.

A function *(beta(k, typeArrival)* that accepts k and arrival distribution type is created to calculate the β values:

By substituting β and P_{ij} values into equation (6), a similar structure of system of equations is acquired. For convenience, this is written in backwards order:

$$\begin{aligned}
0 \cdot P_0 + ... + (-\beta_0)P_{K-2} + (1-\beta_1)P_{K-1} + (-\beta_1)P_K &= 0 \\
0 \cdot P_0 + ... + (-\beta_0)P_{K-1} + (1-\beta_0)P_K &= 0 \\
&... \\
&... \\
&... \\
P_0 + P_1 + P_2 + ... + P_K &= 1
\end{aligned}$$

The resulting system of equations shows a more tractable pattern compared to the one from $M/G/1/K$ queue. The coefficients along the diagonal are equal to $1-\beta_1$ except in the first row, where the coefficient is $1-\beta_0$. Coefficients of the

first column are in form of $-\beta_i$ for i^{th} row. For all other entries, the coefficient of the $(i, j)^{th}$ entry is $-\beta_{i-j+1}$. Expressed in matrix form, the system can be written as $A \cdot x = B$ for Python implementation:

$$A = \begin{bmatrix} 1-\beta_0 & -\beta_1 & 0 & 0 & \cdots & 0 \\ -\beta_1 & 1-\beta_1 & -\beta_0 & 0 & \cdots & 0 \\ -\beta_2 & -\beta_2 & 1-\beta_1 & -\beta_0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \ddots & \vdots \\ & & & & \ddots & \vdots \\ 1 & 1 & 1 & \cdots & \cdots & 1 \end{bmatrix}, B = \begin{bmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ \vdots \\ 1 \end{bmatrix}$$

3.2 Simulation in Python

Instead of assuming the interarrival times to be exponential in $M/G/1/K$ model, the $G/M/1/K$ fixes service times to follow an exponential distribution, while allowing the arrival process to follow a general distribution.

Besides the modifications to general interarrival times distribution and exponential service times, the implementation of simulation in Python is similar to that of the $M/G/1/K$ model.

3.3 Validation

To validate the analytical calculation and simulation results, the buffer size K is fixed at 5, and the total simulating number of jobs is set to $100,000$. The PDFs used in this section are similar as those from $M/G/1/K$ except changing μ to λ, since the general interarrival times are generated instead of general service times.

3.3.1 Deterministic

k	0	1	2	3	4	5
Calculated P_k	0.6774	0.2197	0.0712	0.0230	0.0071	0.0017
Simulated P_k	0.6755	0.2228	0.0715	0.0222	0.0064	0.0015

Table 10: Deterministic Interarrival Distribution $\lambda = 0.6$, $\mu = 1$

3.3.2 Exponential

k	0	1	2	3	4	5
Calculated P_k	0.3041	0.2281	0.1711	0.1283	0.0962	0.0722
Simulated P_k	0.3003	0.2268	0.1694	0.1313	0.0995	0.0727

Table 11: Exponential Interarrival Distribution $\lambda = 0.6$, $\mu = 0.8$

3.3.3 Gamma

k	0	1	2	3	4	5
Calculated P_k	0.1872	0.1769	0.1673	0.1588	0.1525	0.1573
Simulated P_k	0.1890	0.1789	0.1678	0.1589	0.1508	0.1546

Table 12: Gamma Interarrival Distribution $\lambda = 0.75$, $\mu = 0.8$, $shape = 0.8$

3.3.4 Geometric

k	0	1	2	3	4	5
Calculated P_k	0.6692	0.2214	0.0732	0.0243	0.0083	0.0036
Simulated P_k	0.6411	0.2326	0.0840	0.0296	0.0095	0.0031

Table 13: Geometric Interarrival Distribution $\lambda = 0.5$, $\mu = 0.99$

3.3.5 Hyperexponential

In this scenario, instead of uniformly sampled μ values, the λ values are uniformly sampled from interval [0.15, 1.5].

k	0	1	2	3	4	5
Calculated P_k	0.1762	0.1710	0.1664	0.1629	0.1611	0.1623
Simulated P_k	0.1523	0.1565	0.1629	0.1704	0.1767	0.1812

Table 14: Hyperexponential Interarrival Distribution $\mu = 1$

3.3.6 Log-Normal

k	0	1	2	3	4	5
Calculated P_k	0.0282	0.0435	0.0703	0.1222	0.2375	0.4983
Simulated P_k	0.0283	0.0437	0.0701	0.1229	0.2353	0.4998

Table 15: Log-Normal Interarrival Distribution $\lambda = 1.5$, $\mu = 0.8$, $\sigma = 1$

3.3.7 Truncated Normal

k	0	1	2	3	4	5
Calculated P_k	0.1725	0.1704	0.1683	0.1659	0.1632	0.1597
Simulated P_k	0.1725	0.1716	0.1680	0.1653	0.1635	0.1591

Table 16: Normal Interarrival Distribution $\lambda = 1, \mu = 1, \sigma = 1$

3.3.8 Uniform

k	0	1	2	3	4	5
Calculated P_k	0.4021	0.2544	0.1609	0.1005	0.0583	0.0238
Simulated P_k	0.4025	0.2561	0.1600	0.1008	0.0578	0.0228

Table 17: Uniform Interarrival Distribution $\lambda = 0.1, \mu = 1$

3.3.9 Weibull

k	0	1	2	3	4	5
Calculated P_k	0.7859	0.1684	0.0361	0.0077	0.0016	0.0003
Simulated P_k	0.7850	0.1680	0.0372	0.0080	0.0015	0.0003

Table 18: Weibull Interarrival Distribution $\lambda = 0.5, \mu = 0.7, \sigma = 1$, *shape_weibull* = 1

4 Analysis of the G/G/1/K Queue

In this section, the G/G/1/K queue is explored, where both the interarrival and service times follow arbitrary distributions. Unlike the M/G/1/K and G/M/1/K discussed earlier, there are no closed-form equations to calculate stationary probabilities of G/G/1/K queue. Therefore, simulation based analysis is employed to in-

vestigate the relationships among key performance metrics. Since that the blocking probability (when $k = K$) is the major concern when analyzing stationary probabilities in most cases and the extensive computation that will cause due to large K values, instead of getting all stationary probabilities, this section primarily focuses on the blocking probability (when $k = K$). Specifically, this section conducts a sensitivity analysis to see how average queue length, average waiting time, and blocking probability vary with traffic intensity and how blocking probability can be reduced when increasing buffer size.

For a fixed $K = 5$, multiple combinations of interarrival and service distributions are considered, including exponential, Weibull, lognormal, normal distribution truncated at 0, etc.

4.1 Exponential Interarrival & Exponential Service Times

The top two plots demonstrate the impact of increasing traffic intensity (ρ from 0.1 to 1) on the average waiting time and average queue length. As ρ increases, both plots show a nonlinear upward trend, resulting a heavily loaded system.

The bottom-left plot shows the relationship between blocking probability and ρ. From the graph, the blocking probability remains close to 0 at low traffic intensities but increases exponentially as ρ approaches to 1, indicating the system's sensitivity to high traffic loads. On the other hand, the bottom-right plot evaluates how the buffer size K affects the blocking probability. The system benefits the most in reducing blocking probability as K increases from 1 to 5, and starts to have

diminishing returns beyond this point.

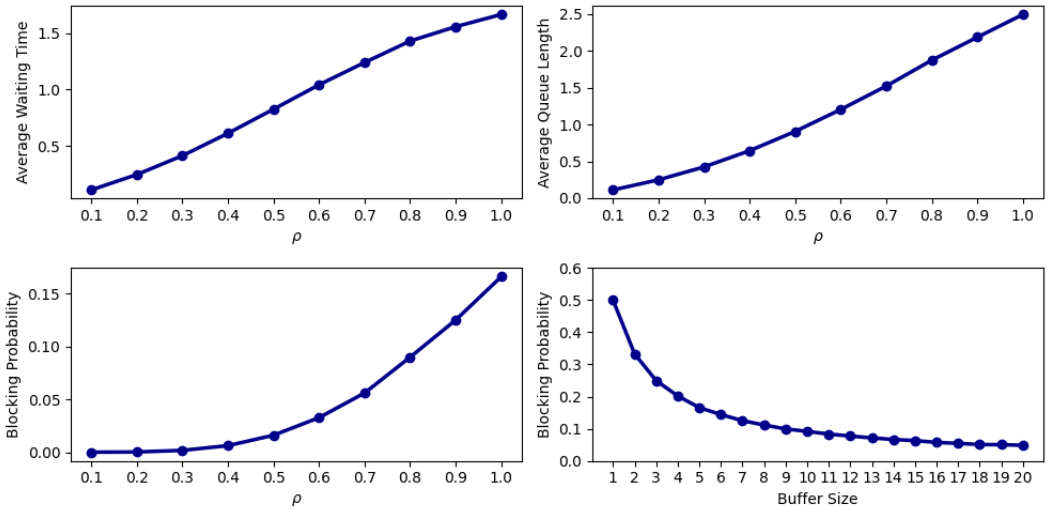

Figure 1: Exponential Interarrival & Exponential Service

4.2 Exponential Interarrival & Deterministic Service Times

In this case where the service times follow a deterministic distribution, the average waiting time and average queue length across varying traffic intensities are very similar with those of the $M/M/1/K$ queue. However, as illustrated in the lower left graph, the blocking probability is slightly lower than the exponential service case. This is likely because the absence of long-tail service times prevents the system from blocking more jobs. Additionally, The blocking probability is relatively high for small buffer sizes (e.g., K from 1 to 4) due to limited capacity for incoming jobs. Also, within this range, the blocking probability reduces sharply and benefits more for having additional unit of buffer size.

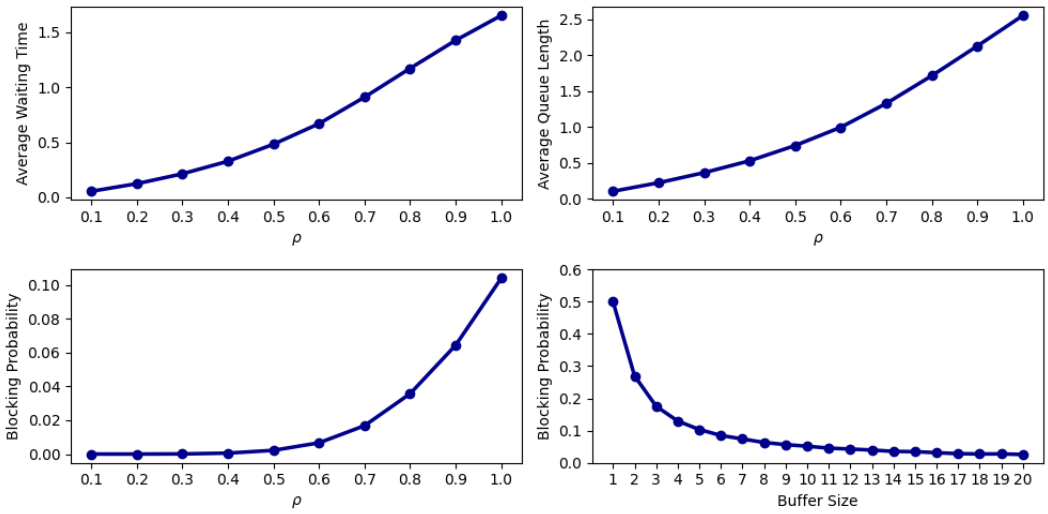

Figure 2: Exponential Interarrival & Deterministic Service

4.3 Lognormal Interarrival & Deterministic Service Times

In contrast of the previous two cases where the average waiting time and average queue length increased rapidly across the full range of traffic intensities, they remain close to 0 at low traffic intensities in this case and then begin to rise sharply when ρ exceeds 0.7. A similar trend is observed in blocking probability with varying ρ values. P_K stays in flat for $\rho \leq 0.7$ and then increases sharply. The system is highly efficient and stable under light traffic, but its performance declines rapidly as the system approaches to heavy utilization.

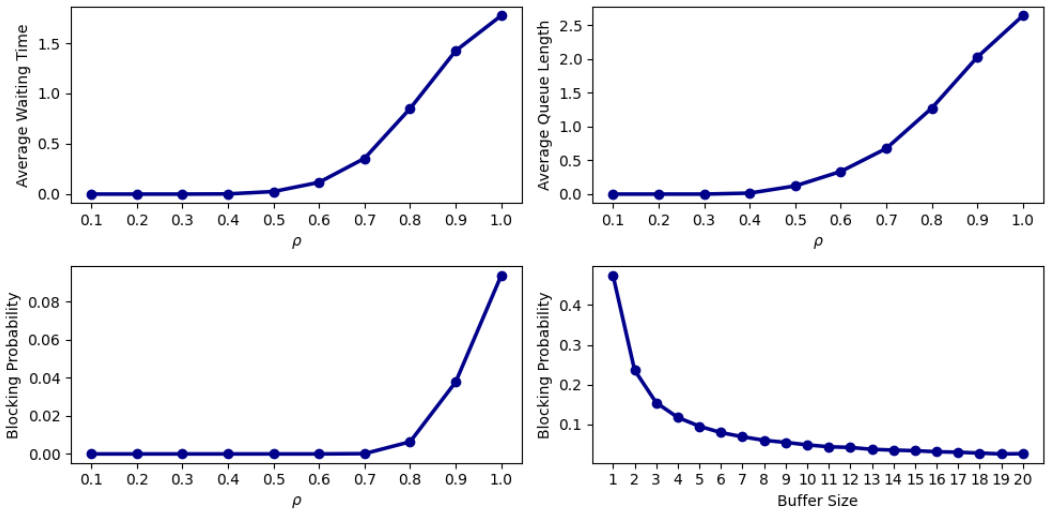

Figure 3: Lognormal Interarrival & Deterministic Service

4.4 Normal (Truncated) Interarrival & Uniform Service Times

For this case with truncated normal interarrival and uniform service times, the average waiting time remains low for small ρ values $(0.1, 0.4)$ but increases significantly as the traffic intensity approaches to 1, showing an s-shaped pattern. This behavior happens because the system clears jobs quickly at low traffic, and jobs accumulate fast and the system tends to remain full while at high traffic. A similar pattern is also shown for average queue length, although its growth is smoother and more gradual.

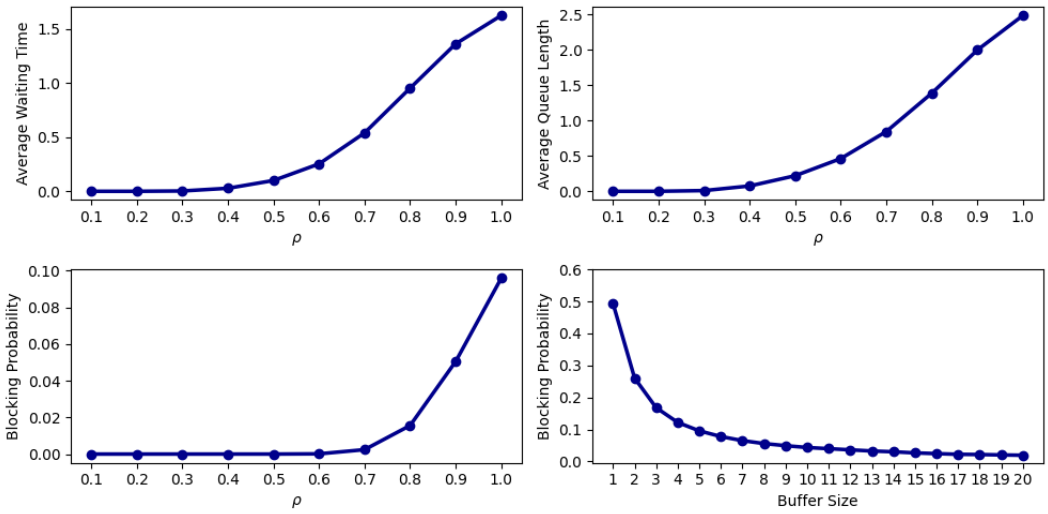

Figure 4: Normal (Truncated) Interarrival & Uniform Service

5 Conclusion

This thesis focused on analyzing the stationary probabilities for $M/G/1/K$ and $G/M/1/K$ queues, along with key performance metrics such as mean waiting time, mean queue length, and blocking probability for $G/G/1/K$ model. Analytical approach using recursive formulations was employed to validate the simulation-based method that was later applied to the $G/G/1/K$ model. Various types of distributions are considered to comprehensively access the $G/G/1/K$ model under different conditions. In addition, the varying traffic intensities and buffer sizes offer more practical implications for real-world scenarios, providing a flexible framework for evaluating realistic queueing models. While this study provides valuable insights for the single-server case, it can be extended to multi-server systems to better adapt to more complex scenarios. Future work could explore multi-server queueing sys-

tems with dynamic traffic patterns to further improve applicability and realism.

References

[1] U. Narayan Bhat. *An Introduction to Queueing Theory: Modeling and Analysis in Applications*, pages 108–121. Statistics for Industry and Technology. Springer Science+Business Media, New York, NY, 2 edition, 2015.

[2] Sanjay K. Bose. Analysis of a m/g/1/k queue without vacations. 2002.

[3] Jamol Pender. The truncated normal distribution: Applications to queues with impatient customers. *Operations Research Letters*, 43(1):40–45, 2015.

[4] J. MacGregor Smith. Optimal design and performance modelling of m/g/1/k queueing systems. *Mathematical and Computer Modelling*, 39:1049–1081, 2004.

6 Appendix

Function that calculates α_k values:

```
def alpha(k, type=type_service):
  if type == 'deterministic':
    return ((lam*(1/mu))**k) / math.factorial(k) *
        np.exp(-lam*(1/mu))
  elif type == 'geometric':
    return np.sum([((lam*n)**k / np.math.factorial(k)) *
        np.exp(-lam*n) * mu * (1 - mu)**(n - 1) for n in range(1,
        10000)])
  else:
    def pdf(t, type):
      if type == 'exponential' or type == 'poisson':
        return mu * np.exp(-mu*t)
      elif type == 'uniform':
        return uniform.pdf(t, loc = a_uniform, scale =
            b_uniform-a_uniform)
      elif type == 'gamma':
        # return gamma.pdf(t, a=shape_gamma,
            scale=(1/mu)/shape_gamma)
        return gamma.pdf(t, a=shape_gamma,
            scale=1/(mu*shape_gamma))
      elif type == 'normal':
        loc_param, scale_param, alpha, beta =
            truncatednormsolver.mu_sigma_solver(1/mu, sigma)
        # print(loc_param, scale_param, alpha, beta)
        return truncnorm.pdf(t, alpha, beta, loc=loc_param,
            scale=scale_param)
      elif type == 'lognormal':
        new_mu = np.log((1/mu)**2 / np.sqrt(sigma**2 +
            (1/mu)**2)) # mean for lognormal rv
        new_sigma = np.sqrt(np.log(sigma**2 / (1/mu)**2 + 1)) #
            sigma for lognormal rv
        return lognorm.pdf(t, s = new_sigma, scale =
            np.exp(new_mu))
      elif type == 'weibull':
        return weibull_min.pdf(t, c=shape_weibull, loc=1/mu,
            scale=sigma)
      elif type == 'hyperexponential':
        return sum([prob_hyperexponential[i] * random_mu[i] *
            np.exp(-random_mu[i]*t) for i in range(d)])
    def f(t):
```

```
        return ((lam*t)**k) / math.factorial(k) * np.exp(-lam*t) *
            pdf(t, type)

    return quad(f, 0, np.inf)[0]

alphas = [alpha(k, type_service) for k in range(K)] # calculate
    all the alphas
print(f'values for alphas: {[round(alphas[i], 4) for i in
    range(len(alphas))]}')
```

Functions that calculate $P_{d,k}$ and P_k values ($M/G/1/K$):

```
def Pdk(K):
    A = np.zeros((K, K)) # create matrix to store the
        coefficients of Pdks
    B = np.zeros(K)
    A[0, 0] = 1 - alphas[0] # entry (1, 1) is always this
    # this loop updaes the entries on diagonal and the first
        column
    for i in range(1, K-1):
        A[i, i] = 1 - alphas[1]
        if i > 0:
            A[i, 0] = -alphas[i]

    # now updates the coefficients for the rest
    for k in range(K - 1): # loop through each row
        for j in range(1, k+1+1): # loop through each column
            if k != j:
                A[k, j] = -alphas[k-j+1]

    # the sum of all Pdks equals to zero
    A[-1, :] = 1
    B[-1] = 1

    # print(A)
    Pdk = np.linalg.solve(A,B) # solve for Pdks
    return Pdk

Pdks = Pdk(K) # calculate all the Pdks
print(f'values for Pdks: {[Pdk(K)[i].round(4) for i in
    range(len(Pdk(K)))]}')
```

```
def P(k):
   if type_service == 'normal':
```

```
        loc_param, scale_param, alpha, beta =
            truncatednormsolver.mu_sigma_solver(1/mu, sigma)
        new_mean = truncnorm.moment(1, alpha, beta,
            loc=loc_param, scale=scale_param)
        rho = lam * new_mean
    else:
        rho = lam/mu
    if k <= K-1:
        return Pdk(K)[k] / (Pdk(K)[0] + rho)
    else:
        return 1 - 1 / (Pdk(K)[0] + rho)

Pks = [P(i) for i in range (K+1)]
print(type_service)
print(f'values for Pks: {[P(i).round(4) for i in range (K+1)]}')
```

Functions that calculate P_k values $(G/M/1/K)$:

```
def beta(k, type=type_arrival):
    if type == 'deterministic':
        return (((mu*(1/lam))**k / (math.factorial(k))) *
            np.exp(-mu*(1/lam)))
    elif type == 'geometric':
        return np.sum([((mu*n)**k / np.math.factorial(k)) *
            np.exp(-mu*n) * lam * (1 - lam)**(n - 1) for n in
            range(1, 1000)])
    else:
        def pdf(t, type):
            if type == 'exponential' or type == 'poisson':
                return lam * np.exp(-lam*t)
            elif type == 'uniform':
                return uniform.pdf(t, loc = a_uniform, scale =
                    b_uniform-a_uniform)
            elif type == 'gamma':
                return gamma.pdf(t, a=shape_gamma,
                    scale=(1/lam)/shape_gamma)
            elif type == 'normal':
                loc_param, scale_param, alpha, beta =
                    truncatednormsolver.mu_sigma_solver(lam,
                    sigma)
                return truncnorm.pdf(t, alpha, beta,
                    loc=loc_param, scale=scale_param)
            elif type == 'lognormal':
                new_lam = np.log((1/lam)**2 / np.sqrt(sigma**2 +
                    (1/lam)**2)) # mean for lognormal rv
                new_sigma = np.sqrt(np.log(sigma**2 / (1/lam)**2
```

```
                    + 1)) # sigma for lognormal rv
                return lognorm.pdf(t, s = new_sigma, scale =
                    np.exp(new_lam))
            elif type == 'weibull':
                return weibull_min.pdf(t, c=shape_weibull,
                    loc=1/lam, scale=sigma)
            elif type == 'hyperexponential':
                return sum([prob_hyperexponential[i] *
                    random_lam[i] * np.exp(-random_lam[i]*t) for
                    i in range(d)])
    def f(t):
      return (((mu*t)**k) / math.factorial(k)) * np.exp(-mu*t) *
          pdf(t, type)
    return quad(f, 0, np.inf)[0]
```

```
def Pak(K):
    A = np.zeros((K+1, K+1))
    B = np.zeros(K+1)

    for i in range(K):
        if i != 0:
            A[i, i] = 1 - betas[1]
            A[i, 0] = -betas[i]
        else:
            A[i, i] = 1 - betas[i]

    for i in range(K):
        for j in range(1, i+1+1):
            if i != j:
                A[i, j] = -betas[i-j+1]
    A[-1, :] = 1
    B[-1] = 1
    # print(A)
    Pak = np.linalg.solve(A, B)
    return Pak[::-1]

Pks = Pak(K)
print(Pks)
print(sum(Pks))
```

Generating arrival and service times:

```
# create a function that returns the arrival and service times
def arrivals_services(type_arrival=type_arrival, lam=lam, mu=mu,
    n=n):
```

```
    # arrival times
    if type_arrival == 'exponential' or type_arrival == 'poisson':
        arrival_times = np.random.exponential(scale=1/lam, size=n)
    elif type_arrival == 'uniform':
        arrival_times = uniform.rvs(loc=a_uniform,
            scale=b_uniform-a_uniform, size=n)
    elif type_arrival == 'gamma':
        arrival_times = np.random.gamma(shape = shape_gamma,
            scale = (1/lam)/shape_gamma, size=n) # mean = shape *
            scale
    elif type_arrival == 'normal':
        loc_param, scale_param, alpha, beta =
            truncatednormsolver.mu_sigma_solver(1/lam, sigma)
        arrival_times = truncnorm.rvs(alpha, beta, loc=loc_param,
            scale=scale_param, size=n)
    elif type_arrival == 'lognormal':
        new_lam = np.log(((1/lam)**2) / np.sqrt(sigma**2 +
            (1/lam)**2))
        new_sigma = np.sqrt(np.log((sigma**2) / ((1/lam)**2) + 1))
        arrival_times = np.random.lognormal(mean = new_lam, sigma
            = new_sigma, size=n)
        # print(f'mean is {np.mean(arrival_times)} and sigma is
            {np.std(arrival_times)}')
    elif type_arrival == 'weibull':
        scale_param = 1/(lam * gamma(1 + 1/shape_weibull))
        arrival_times = weibull_min.rvs(shape_weibull,
            scale=scale_param, size=n)
    elif type_arrival == 'deterministic':
        arrival_times = [1/lam]*n
        # print(f'yes {arrival_times}')
    elif type_arrival == 'geometric':
        arrival_times = np.random.geometric(p=lam, size=n)
    elif type_arrival == 'hyperexponential':
        new_lam = 0
        for i in range(d):
            # a = np.random.exponential(scale=1/random_mu[i],
                size=1)
            new_lam += random_lam[i] * prob_hyperexponential[i]
        arrival_times = np.random.exponential(scale=1/new_lam,
            size=n)

    # service times
    if type_service == 'exponential' or type_service == 'poisson':
        service_times = np.random.exponential(scale=1/mu, size=n)
    elif type_service == 'uniform':
        service_times = uniform.rvs(loc=a_uniform,
            scale=b_uniform-a_uniform, size=n)
    elif type_service == 'gamma':
```

```
        service_times = np.random.gamma(shape = shape_gamma,
            scale = (1/mu)/shape_gamma, size=n) # mean = shape *
            scale
    elif type_service == 'normal':
        loc_param, scale_param, alpha, beta =
            truncatednormsolver.mu_sigma_solver(1/mu, sigma)
        service_times = truncnorm.rvs(alpha, beta, loc=loc_param,
            scale=scale_param, size=n)
    elif type_service == 'lognormal':
        new_mu = np.log(((1/mu)**2) / np.sqrt(sigma**2 +
            (1/mu)**2))
        new_sigma = np.sqrt(np.log((sigma**2) / ((1/mu)**2) + 1))
        service_times = np.random.lognormal(mean = new_mu, sigma
            = new_sigma, size=n)
        # print(f'mean is {np.mean(service_times)} and sigma is
            {np.std(service_times)}')
    elif type_service == 'weibull':
        scale_param = 1/(mu * gamma(1 + 1/shape_weibull))
        service_times = weibull_min.rvs(shape_weibull,
            scale=scale_param, size=n)
    elif type_service == 'deterministic':
        service_times = [1/mu]*n
    elif type_service == 'geometric':
        service_times = np.random.geometric(p=mu, size=n)
    elif type_service == 'hyperexponential':
        # service_times = np.zeros(n)
        new_mu = 0
        for i in range(d):
            # a = np.random.exponential(scale=1/random_mu[i],
                size=1)
            new_mu += random_mu[i] * prob_hyperexponential[i]
        service_times = np.random.exponential(scale=1/new_mu,
            size=n)

    arrival_times = np.cumsum(arrival_times)
    return arrival_times, service_times
arrival_times, service_times = arrivals_services(type_arrival,
    lam, mu, n)
print(f'arrival times: {arrival_times}')
print(f'service times: {service_times}')
print(f'mean service times: {np.mean(service_times)}')
print(f'mean is {np.mean(service_times)}, and sigma is
    {np.std(service_times)}')
```

Simulation:

```
def simulating(n=n, K=K, arrival_times=arrival_times,
    service_times=service_times):
    service_start_times = np.zeros(n) # store the service
        starting time for each job
    job_departures_times = np.zeros(n) # store the service ending
        time for each job
    job_waiting_times = np.zeros(n) # store the waiting time for
        each job
    blocked_list = [] # record whether the job is accepted or not
    queue_lengths_arrival = np.zeros(n) # record the queue size
        before each arrival
    system_list = [] # to store the number of jobs in the system
        in current iteration

    for i in range(n):
        current_arrival_time = arrival_times[i] # job's arrival
            time in current iteration
        system_list = [j for j in system_list if
            current_arrival_time < j] # check to see if there're
            any jobs left before current arrival
        queue_lengths_arrival[i] = len(system_list) # store the
            number of jobs before each arrival

        if queue_lengths_arrival[i] < K: # accept only if the
            queue is not full
            blocked_list.append(False)
            if queue_lengths_arrival[i] == 0: # no waiting time
                if the queue is empty
                current_start_time = arrival_times[i]
            else:
                current_start_time = max(current_arrival_time,
                    system_list[-1]) # the current service starts
                    after the previous job departs

            # update the information
            service_start_times[i] = current_start_time
            job_departures_times[i] = current_start_time +
                service_times[i]
            job_waiting_times[i] = current_start_time -
                current_arrival_time
            system_list.append(job_departures_times[i]) # add the
                current job to the queue list

        else:
            blocked_list.append(True) # block if the queue is full
            # update the information
            service_start_times[i] = current_arrival_time
            job_departures_times[i] = current_arrival_time
```

```
            job_waiting_times[i] = 0

    return service_start_times, job_departures_times,
        job_waiting_times, blocked_list, queue_lengths_arrival

service_start_times, job_departures_times, job_waiting_times,
    blocked_list, queue_lengths_arrival= simulating(n, K,
    arrival_times, service_times)
df = pd.DataFrame({
    'job id': np.arange(1, n+1),
    'Arrival Time': arrival_times,
    'Service Starting Time': service_start_times,
    'Service Time': service_times,
    'Departure Time': job_departures_times,
    'Waiting Time': job_waiting_times,
    'Blocked Status': blocked_list,
    'Queue Length Upon Arrival': queue_lengths_arrival,
})
pd.set_option('display.max_columns', None)
pd.set_option('display.width', n)
# print(df)
print(df.head(20))
count_True = blocked_list.count(True)
count_False = blocked_list.count(False)
total = count_True + count_False
prob_blocking = count_True / total
print(f'blocking probability: {prob_blocking}')

def Pk_probabilities(K=K):
    _, count = np.unique(queue_lengths_arrival,
        return_counts=True)
    total = sum(count)
    count = count/total
    return count
print(Pk_probabilities())
```

Visualization:

```
def visulization(rho_low, rho_high):
    print(f'{type_arrival} & {type_service}')
    mu = 1
    lam = rho_low
    rho_values = []
    average_waiting_time = []
    average_queue_length = []
    blocking_probabilities = []
```

```
    while lam/mu < rho_high - 0.1:
        lam += 0.1
        rho_values.append(lam/mu)
        # print(lam/mu)
        arrival_times, service_times =
            arrivals_services(type_arrival, lam, mu, n)
        _, _, job_waiting_times, blocked_list,
            queue_lengths_arrival= simulating(n, K,
            arrival_times, service_times)
        average_waiting_time.append(np.mean(job_waiting_times))
        average_queue_length.append(np.mean(queue_lengths_arrival))
        count_True = blocked_list.count(True)
        count_False = blocked_list.count(False)
        total = count_True + count_False
        prob_blocking = count_True / total
        blocking_probabilities.append(prob_blocking)

    # varying K
    K_high = 20
    blocking_probabilities_new = []
    K_values = list(range(1, K_high + 1))
    for K_value in K_values:
        lam = 1
        mu = 1
        arrival_times, service_times =
            arrivals_services(type_arrival, lam, mu, n)
        _, _, _, blocked_list, _= simulating(n, K_value,
            arrival_times, service_times)
        count_True = blocked_list.count(True)
        count_False = blocked_list.count(False)
        total = count_True + count_False
        prob_blocking = count_True / total
        blocking_probabilities_new.append(prob_blocking)

    plt.figure(figsize=(10, 5))

    plt.subplot(2, 2, 1)
    plt.plot(rho_values, average_waiting_time, marker='o',
       color='darkblue', linewidth=2.5)
    plt.xlabel(r'$\rho$')
    plt.ylabel('Average Waiting Time')
    plt.xticks(np.arange(0.1, 1.1, 0.1))

    plt.subplot(2, 2, 2)
    plt.plot(rho_values, average_queue_length, marker='o',
       color='darkblue', linewidth=2.5)
    plt.xlabel(r'$\rho$')
    plt.ylabel('Average Queue Length')
```

```
    plt.xticks(np.arange(0.1, 1.1, 0.1))

    plt.subplot(2, 2, 3)
    plt.plot(rho_values, blocking_probabilities, marker='o',
        color='darkblue', linewidth=2.5)
    plt.xlabel(r'$\rho$')
    plt.ylabel('Blocking Probability')
    plt.xticks(np.arange(0.1, 1.1, 0.1))

    plt.subplot(2, 2, 4)
    plt.plot(K_values, blocking_probabilities_new, marker='o',
        color='darkblue', linewidth=2.5)
    plt.xlabel('Buffer Size')
    plt.ylabel('Blocking Probability')
    plt.xticks(np.arange(1, K_high + 1, 1))
    plt.yticks(np.arange(0, 0.7, 0.1))

    plt.tight_layout()
    plt.show()

visulization(0, 1)
```