Lab 1: Analysis of M/M/1/k Queueing System

SCHOOL OF ADVANCED TECHNOLOGY

CAN405: Data Communication and Communication Networks

Mengfan Li

ID number：2032048

2020 Fall

Abstract

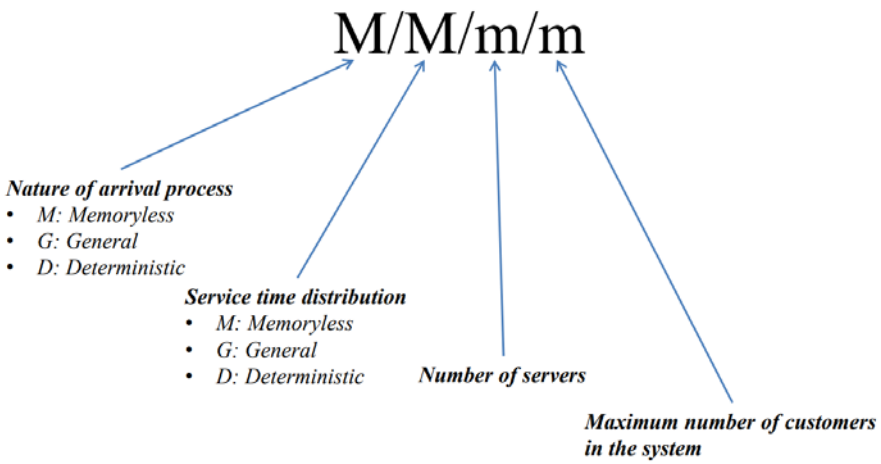
In this assignment, we are to analyze the blocking probability—also called loss rate—of M/M/1/K finite- buffer queueing system based on both queueing theory and simulation.

Declaration

I confirm that I have read and understood the University's definitions of plagiarism and collusion from the Code of Practice on Assessment. I confirm that I have neither committed plagiarism in the completion of this work nor have I colluded with any other party in the preparation and production of this work. The work presented here is my own and in my own words except where I have clearly indicated and acknowledged that I have quoted or used figures from published or unpublished sources (including the web). I understand the consequences of engaging in plagiarism and collusion as described in the Code of Practice on Assessment.

Content

1 Introduction

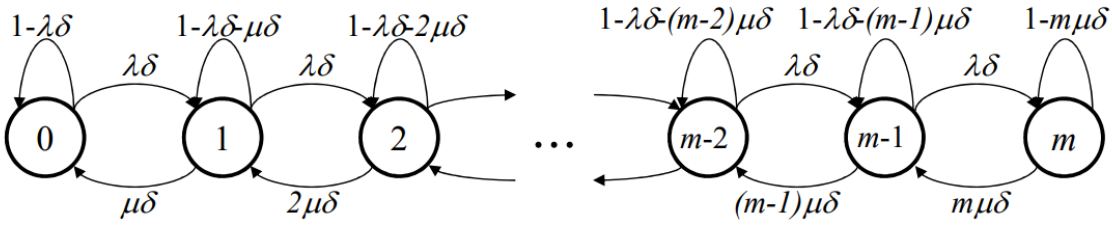


In this Lab, we use the M/M/1/K model, it means that:

Number of servers: 1

The maximum number of customers in the system is: K

The M/M/m/m model is this:



M/M/m/m – The *m*-Server Loss System

Based on the original M/M/1, it is extended to M/M/1/K, that is the length of the queue is changed from infinite to a finite value K.

2 Materials

Pycharm software was used to compile the code, and all the code versions were python3.8. The simulation used simpy to carry out the process-based discrete-event simulation

3 Methods, result, and discussion

Task

CALCULATION OF THE BLOCKING PROBABILITY OF M/M/1/K QUEUEING SYSTEM

#1

Take the sample code of the M/M/1 queueing system, extend it to M/M/1/K queueing system, carry out a series of simulations for the parameter values given below, and measure the blocking probability.

- Arrival rate: 5, 10, 15, . . . , 95
- Service rate: 100 (fixed)
- System capacity (K): 10, 20, 50

Introduction of Simpy

SimPy is a discrete event-driven simulation library. All customers (the first M in this lab) can be simulated using Process. These processes are stored in the environment. All interactions between processes and the environment are conducted through events.

The process expression is one time higher than the trailing edge, constructing an event(event) and throwing an event via a yield statement.

When a process throws an event, the process is suspended until the event is triggered. Multiple processes can wait for the same event. SimPy restores processes in order of activation of the events

thrown by these processes.

In fact, the most important type of event is Timeout, which allows a period of time to be activated, and is used to express a specified period of time for a process to sleep or keep its current state. This type of event is called the environment.timeout.

Environment determines the starting/ending point of the simulation, manages the relationships between simulation elements, and the main API is:

Simpy.Environment.	add the simulation process
Simpy.Environment.event	Create the event
Simpy. Environment.timeout	provide delay (timeout) events
Simpy.Environment.until	Conditions (time or event) for the end of the simulation
Simpy.Environment.run	simulation starts

Extend to M/M/1/K queueing system

The core idea is to use a K to represent System Capacity to control the length of the queue

Step 1: First we find the main function and add a control statement to the main function

```
1.   # run a simulation
2.   system_capacity_list = [10, 20, 50]
3.   for i in range(len(system_capacity_list)):
4.       for arrival_rate in np.linspace(5, 95, 19):
5.
6.           blocking_probability = run_simulation(np.reciprocal(arrival_rate),
7.                       mean_srv_time, packet_number, system_capacity_list[i],
8.                                                 random_seed, trace=True)
9.
10.          blocking_probability_list[i].append((blocking_probability))
11.          loss_number = 0
```

We created an array to save K, based on the list of arrival rates given by the documentation, we write a cyclic tone with one arrival rate per call.

And it creates a list of System capacity, changing the value of System capacity with each call

In the parameter run_simulation, the parameters are as follows:

np.reciprocal(arrival_rate) : The reciprocal of arrive_rate is the mean packet interarrival time

mean_srv_time : mean packet service time

num_packets : number of packets to generate

system_capacity_list[i] : system capacity number [10, 20, 50]

Step 2: run the simulation

```
1.    def run_simulation(mean_ia_time, mean_srv_time, packet_number, system_capacity,
2.                       random_seed, trace=True):
3.        """
4.        Runs a simulation and returns statistics
5.        """
6.
7.        print('M/M/1/K queue\n')
8.        random.seed(random_seed)
9.        env = simpy.Environment()
10.
11.       # start processes and run
12.
13.       server = simpy.Resource(env, capacity=1)
14.       env.process(source(env, mean_ia_time, mean_srv_time, server, packet_number,
15.                                  system_capacity, trace))
16.       env.run()
17.
18.       # return statistics
19.       return loss_number / packet_number
```

When the simulation is triggered, the data is passed into the parameter of run_simulation.

print statement to print a starting string:

```
(base) C:\Users\123\Documents\CAN405>python mm1k.py
5
M/M/1 queue
```

The simpy generator is then fired for simulation

```
1.    env = simpy.Environment()
```

This process is then triggered and the parameters are passed to the source.

Step 3:

```
1.   def source(env, mean_ia_time, mean_srv_time, server, packet_number,
2.                                       system_capacity, trace):
3.       """Generates packets with exponential interarrival time."""
4.
5.       for i in range(packet_number):
6.           ia_time = random.expovariate(1.0 / mean_ia_time)
7.           srv_time = random.expovariate(1.0 / mean_srv_time)
8.
9.           pkt = packet(env, 'Packet-%d' % i, server, srv_time, system_capacity, trace)

10.          env.process(pkt)
11.
12.          yield env.timeout(ia_time)
```

There's a loop here, simulating it in terms of the number of packets

The interval at which an exponential distribution is generated

```
1.       ia_time = random.expovariate(1.0 / mean_ia_time)
2.       srv_time = random.expovariate(1.0 / mean_srv_time)
```

Then, the parameters are passed into the packet

Step 4:

a) Parameters are passed through the source to packet

source code (Part: Extend to M/M/1/K queueing system step 3)

packet code:

```
def packet(env, name, server, service_time, system_capacity, trace):
```

b) We take the strategy of simulating entry and then judging, so each trigger increases queue number by 1

Code:

```
"""
queue_number will automatically increment by default
every time one packet is transferred
"""
queue_number += 1
```

c) Judge strategy:

When the number of services in the queue plus the number of waits is :

if less than K can also enter the queue

if equal to K, the queue will be blocked;

if than K, the queue will no longer be allowed to enter, and the loss number plus 1.

When the queue is blocked, because it is M/M/1/k queue, the blocking time is equal to the service time of the customer being served. After the service is completed, the number of queues will be reduced by one, and the state will change

Code:

```
def packet(env, name, server, service_time, system_capacity, trace):

    # Requests a server, is served for a given service_time, and leaves the server
    arrv_time = env.now

    global queue_number, loss_number

    # queue_number automatically increment  every time one packet is transfered
```

```
        queue_number += 1

        """
        Part K of MM1K, K is System capacity,
        If queue number is less than or equal to system capacity, the queue is not block
ed
        """
        if queue_number <= system_capacity:
            if trace:
                print('t=%.4Es: %s arrived, ' % (arrv_time, name))

            with server.request() as request:
                yield request

                yield env.timeout(service_time)

                # When the service is complete, the queue releases one
                queue_number -= 1

                if trace:
                    print('t=%.4Es:%s served for %.4Es' % (env.now, name, service_time))

        # queue number > system capacity, loss number plus 1
        else:

            queue_number -= 1
            loss_number += 1

            if trace:
                print('t=%.4Es: %s loss packet num:  ' % (env.now, name))
```

At this point, M/M/1 expands to M/M/1/k

Calculate the blocking probability

Step1:

In packet, we recorded the loss number when the system blocked:

```
    # queue number > system capacity, loss number plus 1
        else:

            queue_number -= 1
            loss_number += 1
```

Packet part full code (Part: Extend to M/M/1/K queueing system step 4 c))

Step2:

In run_simulation (Part: Extend to M/M/1/K queueing system step 2), we will take the final argument of the blocking rate and return it to blocking_probabilities.

```
    blocking_probability = run_simulation(np.reciprocal(arrival_rate), mean_srv_time,
                                  packet_number, system_capacity_list[i],
                                                 random_seed, trace=True)
```

In run_simulation, the return value is loss_number / packet_number , and according to formula (1) , it is the blocking probability

$$P_B = \frac{p_K}{\sum_{n=0}^{n=K} p_n} \tag{1}$$

Code:

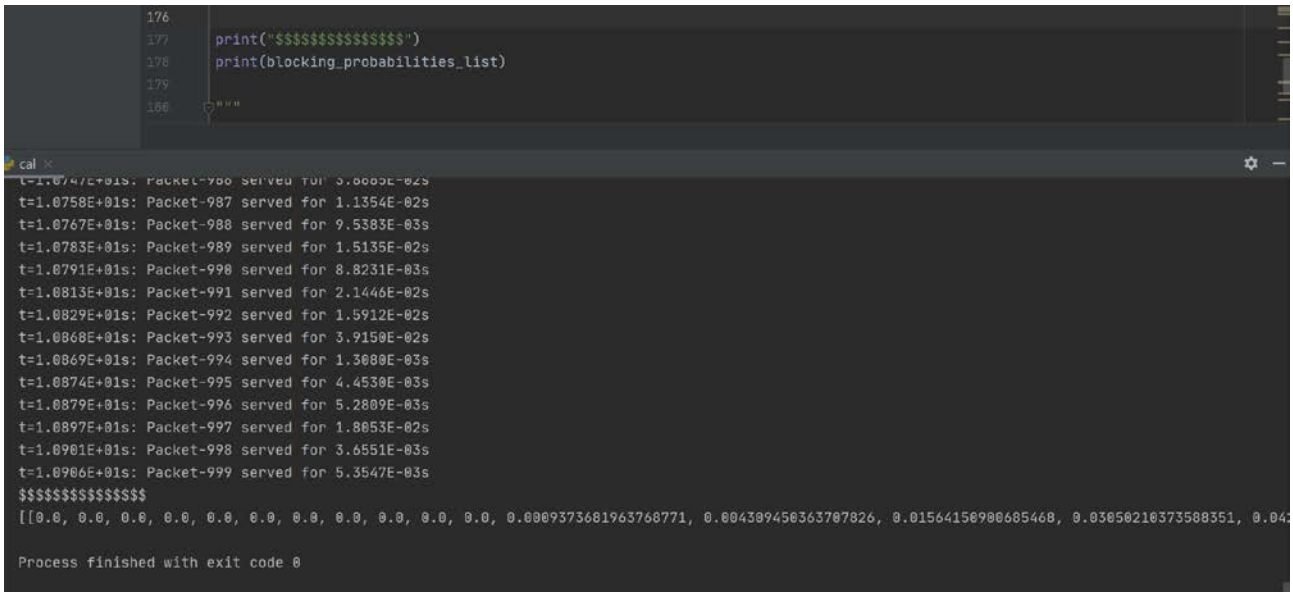
```
    # return statistics
    return loss_number / packet_number
```

Step3:

each blocking probability append passed into a list is saved as the final result.

```
    blocking_probabilities_list[i].append((blocking_probabilities))
```

Result:



After printing blocking_probabilities_list, you can see that each result is saved, and each result corresponds to a System capacity and an Arrival rate.

#2

Plot 3 charts (i.e., for K=10, 20, 50) comparing the results of the blocking probabilities obtained from the simulations and the analysis based on the queueing theory (i.e., (1) given in Section II). The charts should show clearly the blocking probabilities for both cases with the common arrival rate as x axis; you need to set the y range properly.

Step1:

According to the formula (2), calculate the theoretical value

$$P_N = \frac{1-\rho}{1-\rho^{N+1}}\rho^N \tag{2}$$

Code:

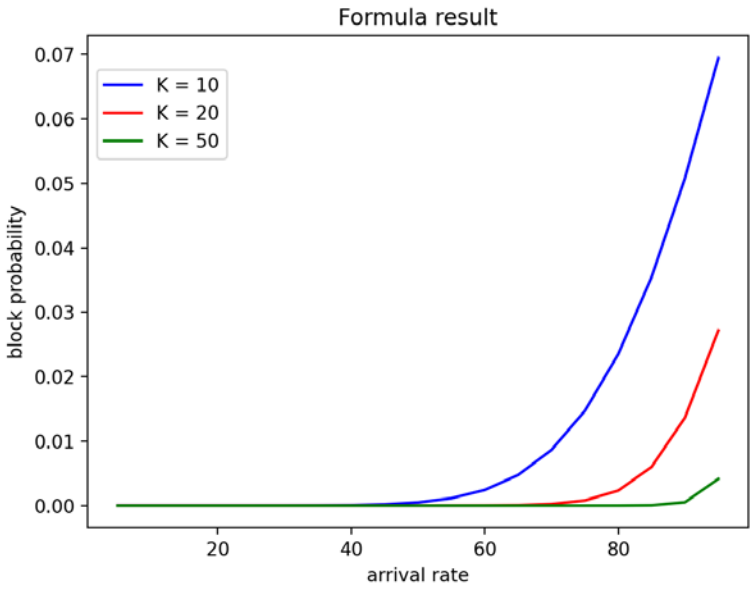
```
temp_array = [[], [], []]
system_capacity_list = [10, 20, 50]
counter = 0

for N in system_capacity_list:
    for Rho in np.linspace(0.05, 0.95, 19):
        P = ((np.power(Rho, N)) - (np.power(Rho, N + 1))) / (1 - (np.power(Rho, N + 1)))
```

```
8.
9.             temp_array[counter].append(P)
10.        counter += 1
```

drawing

```
1.    plt.figure()
2.    x_axis = np.linspace(5, 95, 19)
3.    plt.plot(x_axis, temp_array[0], color='blue', label='K = 10')
4.    plt.plot(x_axis, temp_array[1], color='red', label='K = 20')
5.    plt.plot(x_axis, temp_array[2], color='green', label='K = 50')
6.    plt.legend(loc='upper left', bbox_to_anchor=(0, 0.95))
7.    plt.title('Formula result')
8.    plt.xlabel("arrival rate")
9.    plt.ylabel('block probability')
10.   plt.show()
```

Result:



Step2:

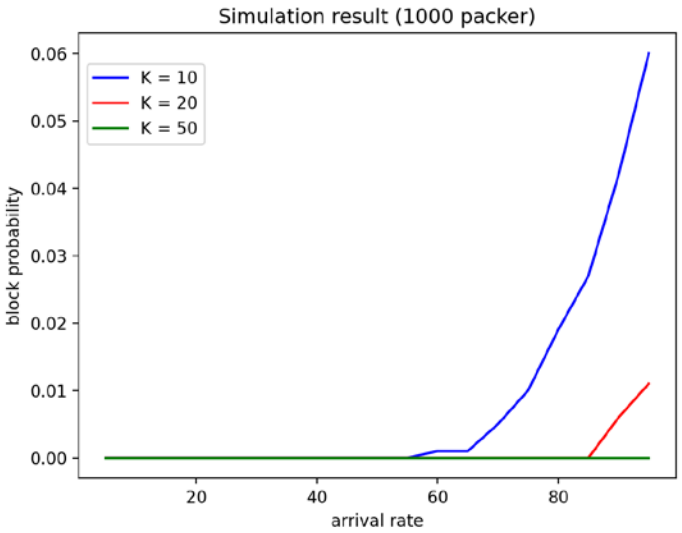
Draw the simulated image

Code:

```
1.    plt.figure()
2.    x_axis = np.linspace(5, 95, 19)
3.    plt.plot(x_axis, blocking_probabilities_list[0], color='blue', label='K = 10')
```

```
4.    plt.plot(x_axis, blocking_probabilities_list[1], color='red', label='K = 20')
5.    plt.plot(x_axis, blocking_probabilities_list[2], color='green', label='K = 50')
6.    plt.legend(loc='upper left', bbox_to_anchor=(0, 0.95))
7.    plt.title('Simulation result')
8.    plt.xlabel("arrival rate")
9.    plt.ylabel('block probability')
```

Result:



Step3:

Compare

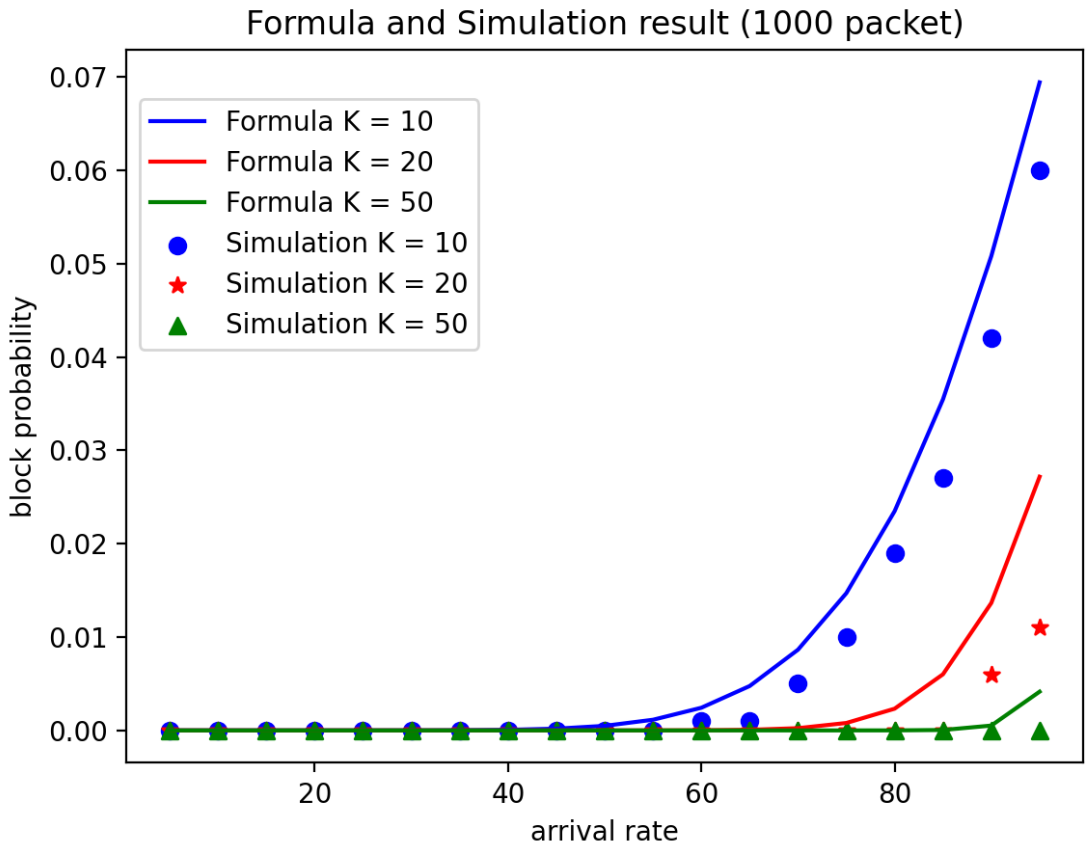
Here we use a line chart as a contrast to a scatter chart

Code:

```
1.    plt.figure()
2.    x_axis = np.linspace(5, 95, 19)
3.    plt.plot(x_axis, temp_array[0], color='blue', label='Formula K = 10')
4.    plt.plot(x_axis, temp_array[1], color='red', label='Formula K = 20')
5.    plt.plot(x_axis, temp_array[2], color='green', label='Formula K = 50')
6.    plt.scatter(x_axis,blocking_probability_list[0], marker='o',color='blue', label='Sim
ulation K = 10')
7.    plt.scatter(x_axis,blocking_probability_list[1], marker='*',color='red', label='Simu
lation K = 20')
8.    plt.scatter(x_axis,blocking_probability_list[2], marker='^',color='green', label='Si
mulation K = 50')
9.    plt.legend(loc='upper left', bbox_to_anchor=(0, 0.95))
10.   plt.title('Formula and Simulation result (1000 packet)')
```

```
11.  plt.xlabel("arrival rate")
12.  plt.ylabel('block probability')
13.  plt.show()
```

Result:



#3

Find any differences between the results from simulation and queueing analysis, if any, and provide your own explanations for them in the report.

Code:

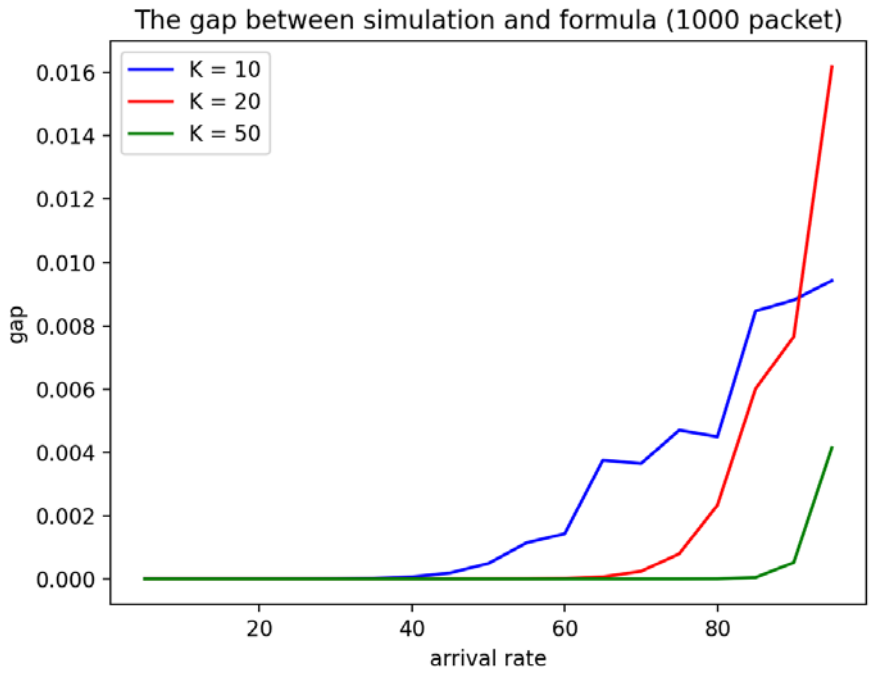
Because of the difference, it doesn't make sense to use the sign, so we take the absolute value here

```
1.  temp_array = np.array(temp_array)
2.  blocking_probability_list = np.array(blocking_probability_list)
```

```
3.    gap = abs(temp_array - blocking_probability_list)
```

```
1.    plt.figure()
2.    x_axis = np.linspace(5, 95, 19)
3.    plt.plot(x_axis, gap[0], color='blue', label='K = 10')
4.    plt.plot(x_axis, gap[1], color='red', label='K = 20')
5.    plt.plot(x_axis, gap[2], color='green', label='K = 50')
6.    plt.legend(loc='upper left', bbox_to_anchor=(0, 0.95))
7.    plt.title('The gap between simulation and formula (1000 packet)')
8.    plt.xlabel("arrival rate")
9.    plt.ylabel('gap')
10.   plt.legend()
11.   plt.show()
```

Result:



As can be seen, there is still a gap between the two, but it is already very small, and the reasons for this gap may be due to the following points:

- Because the simulation is a random process, there will inevitably be some errors, but the formula calculation is pure value, there will be no such situation.

- packet_number
- random_seed

Specific issues will be discussed further in 4 Extension.

#4

Argparse is added to the program, so here we show how to use CMD to change the parameters.

The parameters changed are as follows:

- arrival_rate
- mean_srv_time
- packet_number
- random_seed
- system_capacity

Step1: We can view the parameters with the help of CMD

```
(base) C:\Users\123\Documents\CAN405\Code\test>python mm1k.py -h
usage: mm1k.py [-h] [-A MEAN_IA_TIME] [-S MEAN_SRV_TIME] [-N NUM_PACKETS] [-R RANDOM_SEED] [-K SYSTEM_CAPACITY]
               [--trace] [--no-trace]

optional arguments:
  -h, --help            show this help message and exit
  -A MEAN_IA_TIME, --mean_ia_time MEAN_IA_TIME
                        mean packet interarrival time [s]; default is 1.0
  -S MEAN_SRV_TIME, --mean_srv_time MEAN_SRV_TIME
                        mean packet service time [s]; default is 0.1
  -N NUM_PACKETS, --num_packets NUM_PACKETS
                        number of packets to generate; default is 1000
  -R RANDOM_SEED, --random_seed RANDOM_SEED
                        seed for random number generation; default is 1234
  -K SYSTEM_CAPACITY, --system_capacity SYSTEM_CAPACITY
                        number of System capacity; default is 1
  --trace
  --no-trace
```

Step2: We can change parameters use CMD

- Case 1: arrival_rate = 5, system_capacity = 10, packet_number = 1000, random_seed 1234

```
(base) C:\Users\123\Documents\CAN405\Code\test>python mm1k.py -A 5 -K 10
M/M/1/K queue
```

Result:

```
t=2.0112E+02s: Packet-995 arrived,
t=2.0112E+02s: Packet-995 served for 4.4530E-03s
t=2.0119E+02s: Packet-996 arrived,
t=2.0120E+02s: Packet-996 served for 5.2809E-03s
t=2.0154E+02s: Packet-997 arrived,
t=2.0156E+02s: Packet-997 served for 1.8053E-02s
t=2.0219E+02s: Packet-998 arrived,
t=2.0219E+02s: Packet-998 served for 3.6551E-03s
t=2.0219E+02s: Packet-999 arrived,
t=2.0220E+02s: Packet-999 served for 5.3547E-03s
##############################
0.0
```

Case 2: arrival_rate = 80, system_capacity = 10, packet_number = 1000, random_seed 1234

```
(base) C:\Users\123\Documents\CAN405\Code\test>python mm1k.py -A 80 -K 10
M/M/1/K queue
```

Result:

```
t=1.2665E+01s: Packet-995 served for 4.4530E-03s
t=1.2670E+01s: Packet-996 served for 5.2809E-03s
t=1.2688E+01s: Packet-997 served for 1.8053E-02s
t=1.2692E+01s: Packet-998 served for 3.6551E-03s
t=1.2697E+01s: Packet-999 served for 5.3547E-03s
##############################
0.019
```

Case 3: arrival_rate = 90, system_capacity = 20, packet_number = 5000, random_seed 1200

```
(base) C:\Users\123\Documents\CAN405\Code\test>python mm1k.py -A 90 -K 20 -N 5000 -R 1200
M/M/1/K queue
```

Result:

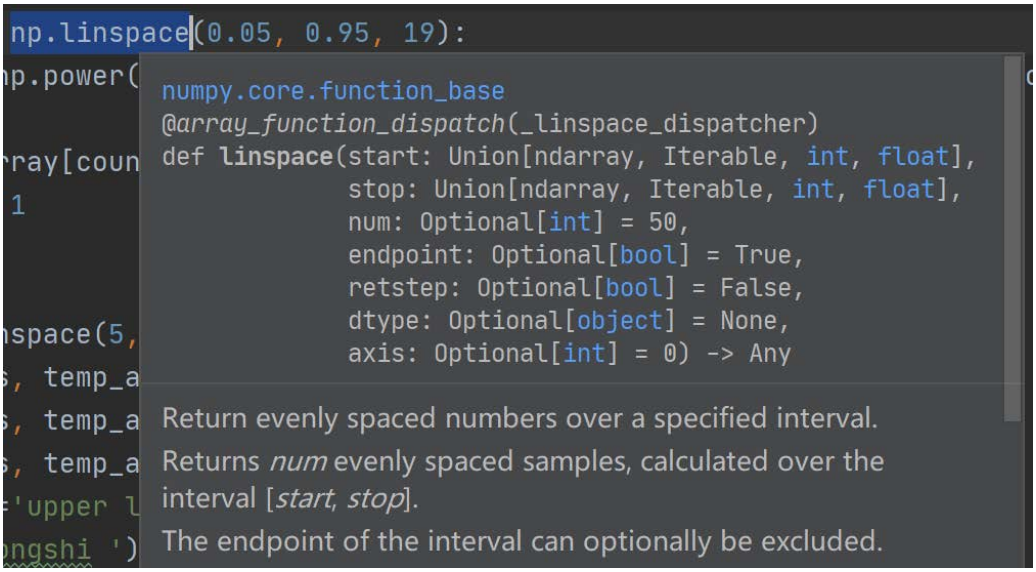
```
t=5.5462E+01s: Packet-4997 served for 7.8108E-03s
t=5.5469E+01s: Packet-4998 served for 6.3222E-03s
t=5.5481E+01s: Packet-4999 served for 1.1798E-02s
##############################
0.0138
```

4 Extension

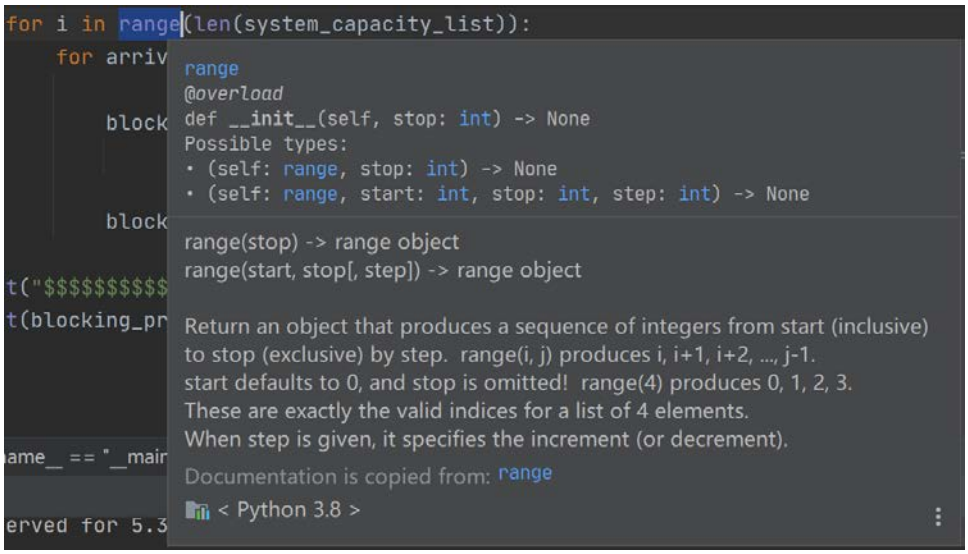
4.1 linespace vs range

We can view the numpy respectively about the definition of np.linspace as well as in python definition of the range

np.linespace:

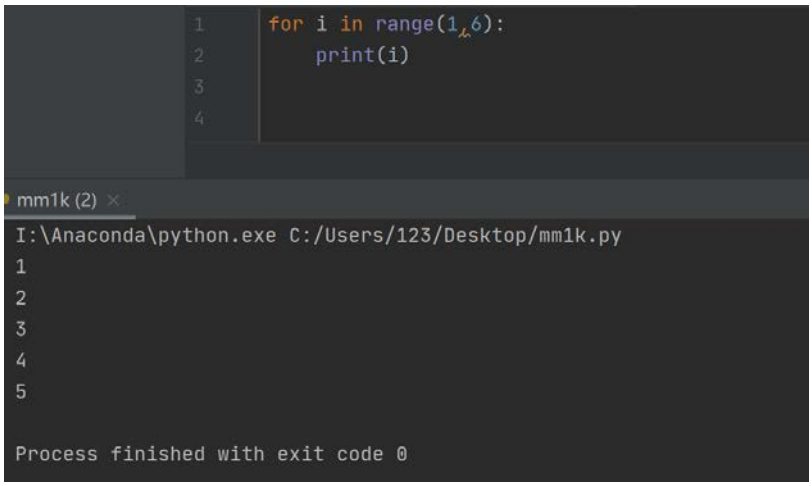


range:

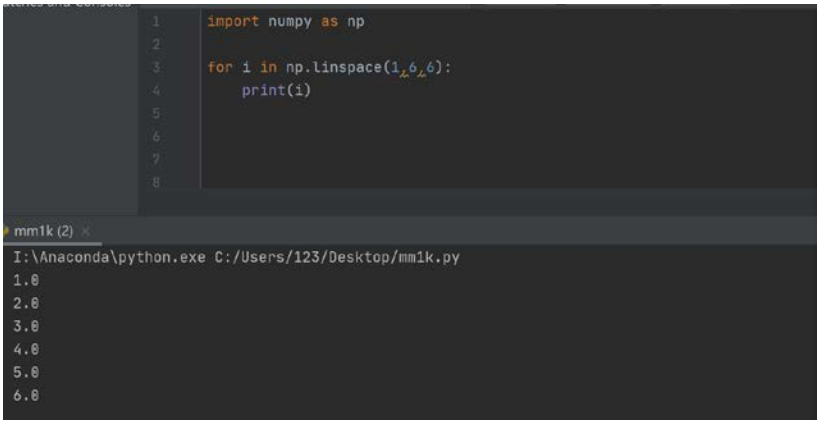


We can see the obvious difference is the value type, if we use np.linspace we can just use floating-point type, but if we use range we have to do the data again.

The other problem is the boundary of the range, in the range, it does not contain the ending boundary:



So when we use range, it's easy to lose the boundary, but the medium problem doesn't happen in np.linspace.



Also, it is clear that the default data type in K is a floating-point row, and the appropriate system is more robust

4.2 Decimal point

In order to enhance the robustness of the system, the system will not crash due to different data types.

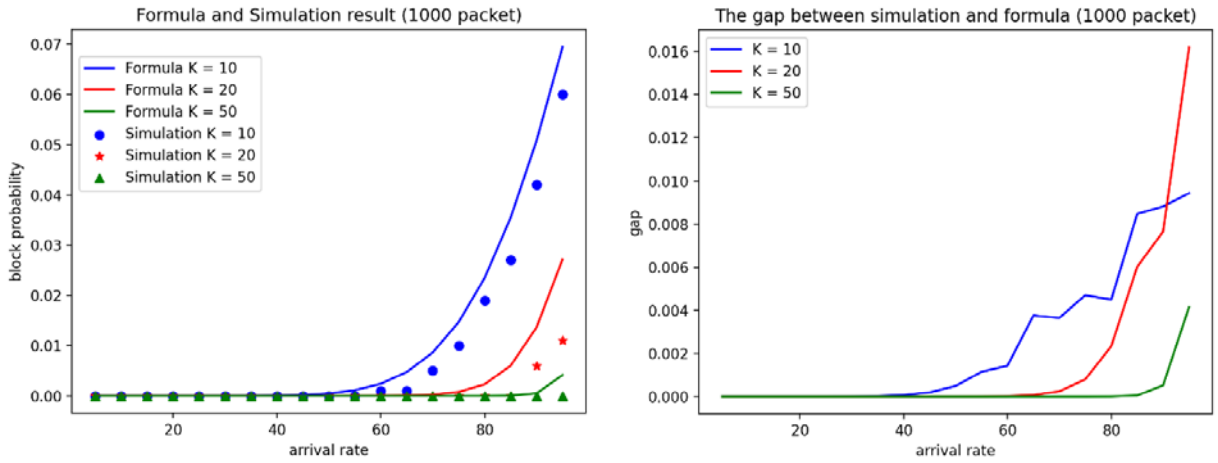
```
parser = argparse.ArgumentParser()
    parser.add_argument(
        "-A",
        "--mean_ia_time",
        help="mean packet interarrival time [s]; default is 1.0",
        default=1.0,
        type=float)
```

Because mean_ia_time is a float type of data in the program, when defined, it is defined as float

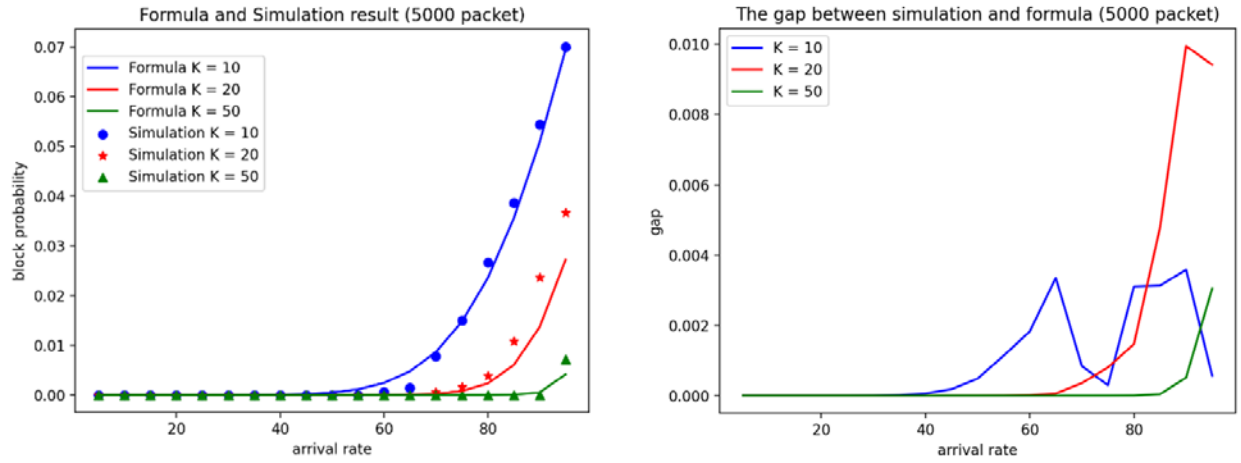
```
1.    ia_time = random.expovariate(1.0 / mean_ia_time)
```

4.3 gap

Here, we use the control variable method to discuss the influence of different packet_number, on the simulation value and formula calculation value



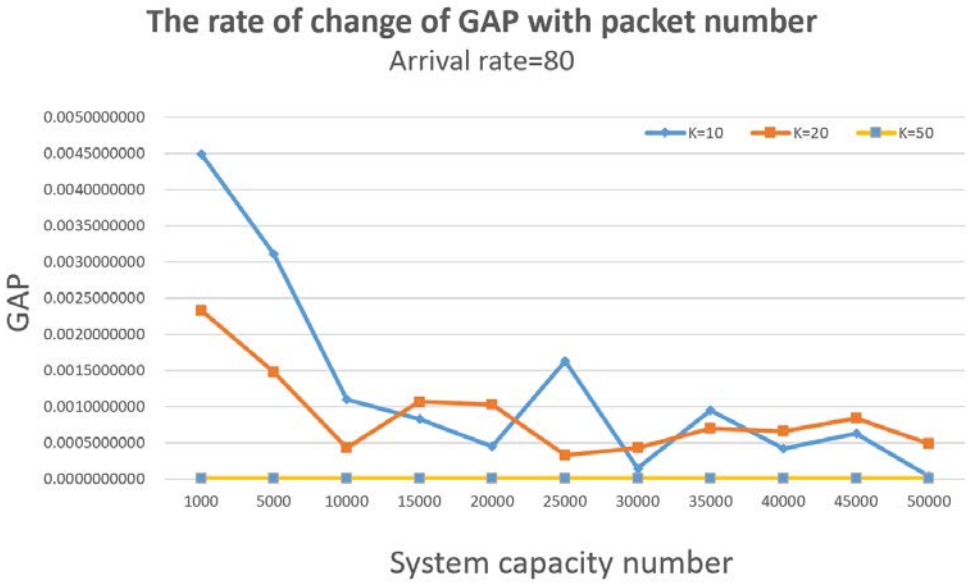
Packet number = 1000



Packet number = 2000

It can be seen that with the increase of packet number, the fitting degree of simulated value and the calculated value is improved, and the range of error is also reduced

So let's calculate the change in error as packet number goes from 1,000 to 50,000



We can see that with the increase of packer number, the variation trend of the gap decreases under the condition of the same System capacity.

5 conclusion

In this experiment, Simpy Generator is used to simulate the M/M/1/ K queuing model. Based on M/M/1, we expand the blocking rate into M/M/1/k by adding K constraints into packets and calculate the blocking rate under different arrival time and packet service time conditions by taking blocking time to total time as the blocking rate. The simulation results are compared with the formula results. Under the same preconditions, changing the amount of packet number can effectively reduce the gap between the simulated value and the calculated value.

REFERENCES

[1] Graduate course on computer networks: M/M/1/k queueing system. Accessed: 2 November 2020. [Online]. Available:http://www.ece.virginia.edu/mv/edu/7457/lectures/packet-switching/queueing-theory/MM1k.pdf