Xi'an Jiaotong-Liverpool University
西交利物浦大学

DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING

EEE413(18/19) Data Communication and Communication Networks

Lab 1 - Experimental Verfication of Little's Theorem EEE413

Lab Report

Student Name	:	Enge Xu
Student ID	:	1821635
Date	:	2018/11/28
Professor	:	Kyeong Soo (Joseph) Kim

ABSTRACT

This paper mainly focuses on introducing how to use python to realize the queueing system via two different methods. Through two tasks to basically illustrate the feasibility of using the average queue length and Little’s theorem and compare it with another way provided. Mainly in order to practice the python programming ability of rewriting code and displaying images and better understanding of queueing theory.

BACKGROUND

1. Python

Python is an interpreted high-level programming language for general-purpose programming. Created by Guido van Rossum and first released in 1991, Python has a design philosophy that emphasizes code readability, notably using significant whitespace. It provides constructs that enable clear programming on both small and large scales. In July 2018, Van Rossum stepped down as the leader in the language community after 30 years.

Python features a dynamic type system and automatic memory management. It supports multiple programming paradigms, including object-oriented, imperative, functional and procedural, and has a large and comprehensive standard library.

Python interpreters are available for many operating systems. CPython, the reference implementation of Python, is open source software and has a community-based development model, as do nearly all of Python's other implementations.

Python's developers strive to avoid premature optimization, and reject patches to non-critical parts of CPython that would offer marginal increases in speed at the cost of clarity. When speed is important, a Python programmer can move time-critical functions to extension modules written in languages such as C, or use PyPy, a just-in-time compiler. Cython is also available, which translates a Python script into C and makes direct C-level API calls into the Python interpreter.

An important goal of Python's developers is keeping it fun to use. This is reflected in the language's name—a tribute to the British comedy group Monty Python—and in occasionally playful approaches to tutorials and reference materials, such as examples that refer to spam and eggs (from a famous Monty Python sketch) instead of the standard foo and bar.

2. Queueing theory

queueing theory, the theory of stochastic service system, has its origins in research by Agner Krarup Erlang when he created models to describe the Copenhagen telephone exchange. The ideas have since seen applications including telecommunication, traffic engineering, computing and, particularly in industrial engineering, in the design of factories, shops, offices and hospitals, as well as in project management.

According to the regularity of the structure, it can better to improve the service system or reorganize the service object, making the service system to meet the needs of the service object, and make the cost of the organization the most economic and some indicators optimal.

It is a branch of mathematics operations research. It is also a subject that studies the stochastic law of queueing phenomenon in service system. It is now widely used in computer network, production, transportation, inventory and other resources sharing random service system.

queueing theory has three aspects: statistical inference, which means model based on data; system behavior, that is, the probability regularity of the quantity index related to queueing; optimization of the system, whose purpose is to correctly design and effectively run each service system, so that it can give full play to the best benefits.

TASK INTRODUCTION

First, it needs to create a SimPy simulation program based on the sample one, where you implement the suggested algorithm for the computation of the average queue length and measure the current simulation time.

the average waiting time using Little ’ s theorem. Then you need to run a series of simulations for the parameter values given below and compare the results for the average waiting time with those from the sample program by plotting a chart.

Below are the parameter values for the simulation of M/M/1 queueing system:

- Arrival rate: 5, 10, 15, ...95
- Service rate: 100 (fixed)

The chart should show clearly (* you may need to set y range properly in this regard *) the average waiting times from both measurements.

Second, you need to modify the simulation program from the subtask #1 and measure the average service time based on the average number of packets under service and Little ’ s theorem. Again, you need to run a series of simulations for the same parameter values as in the subtask #1 and compare the results with the inverse of the service rate (i.e., the given average service time) by plotting a chart; unlike the subtask #1, you don’ t have to run another set of simulations for direct measurements this time.

METHODOLOGY

There are two ways to measure the average waiting time:
One way is a direct measurement where the waiting times of packets are accumulated as they finish waiting in the queue; when the simulation completes, this sum is divided by the number of packets to obtain the average waiting time. Note that this approach is already implemented in the sample M/M/1 simulation program based on SimPy (SimPy is a process-based discrete-event simulation framework based on standard Python).

The other way is an indirect measurement using the average queue length (i.e., the average number of packets in the queue) and Little’s theorem. The definition of state variables in Little’s theorem is shown as below;

N(t): Number of customers in the system at time t.
α (t): Number of customers arrived during the interval [0, t].
Ti: Time spent in the system by the its customer.

The average queue length is the area under the second graph divided by the period of simulation,which is shown in Figure 1 as below;

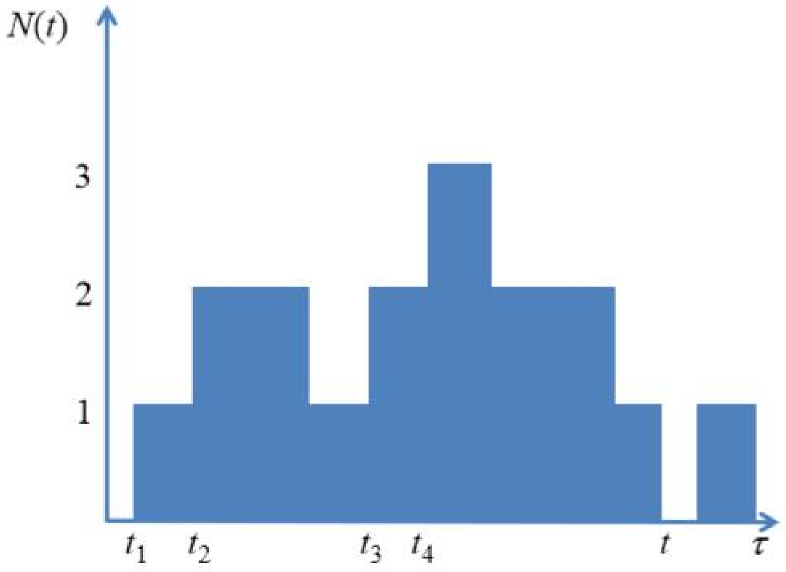

Figure 1

This area is the sum of the rectangles; the height of each rectangle is the number of packets in the queue, and the width is the time interval between changes in this number. Note that the sum does not contain the area under N(1). Because the area represents the service time which can not recognized as waiting time. Thus the waiting time are can be represented as below in Figure 2;

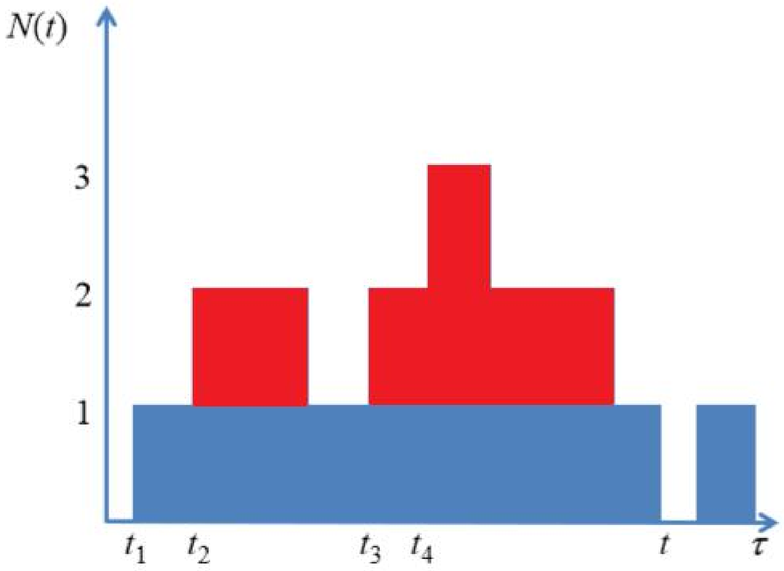

Figure 2

Therefore, we can definite these added variables sum, queue_length, and status_change_time.

At the beginning of the simulation, initialise them, i.e.,

- sum → 0
- queue_length → 0
- status_change_time → 0

These variables are updated as follows during the simulation:

- At each packet arrival:

* sum → sum + queue_length×(env.now-status_change_time)
* queue_length → queue_length + 1
* status_change_time → env.now

Note that this part will calculate the waiting time when the next customer comes while the previous customer has not finished and is waiting in line. It can be represent as 2 part of area in Figure 3 as below;

- At the end of each packet’s waiting in the queue:

* sum → sum + queue_length×(env.now-status_change_time)
* queue_length → queue_length - 1
* status_change_time → env.now

Note that this part will calculate the waiting time when the previous customer finishes the service time and the queue length minus one. It can be represent as 1&3&4 part of area in Figure 3 as below;

Figure 3 is shown as below;

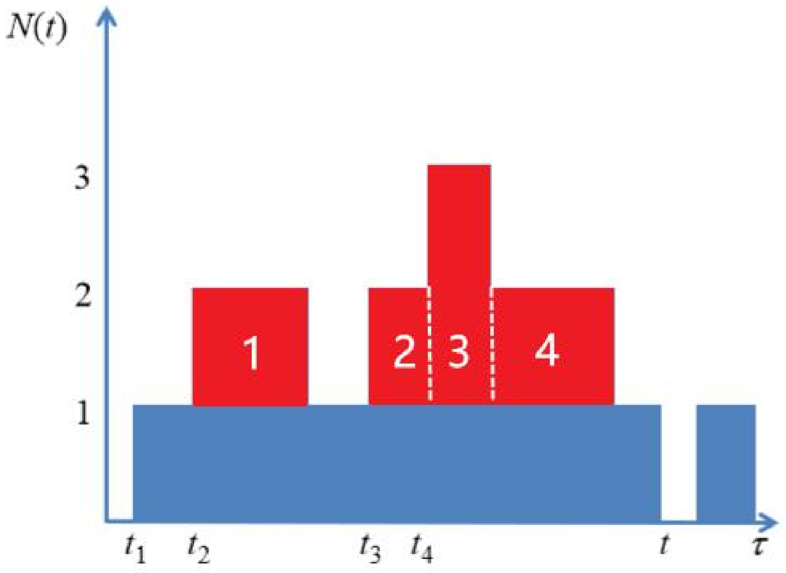

Figure 3

These variables can be created and added into the original mm1.py file combining with the basic Little's theorem shown before. Therefore, the code of each packet arrival is shown as below:

```
# record arrival time
    arrv_time = env.now
    if trace:

        change_time= env.now - status_change_time

        # update at each packet arrival
        if queue_length > 0:
            # only waiting people will be calculate
            summ=summ+(queue_length-1)*(change_time)

        queue_length = queue_length + 1

        status_change_time=env.now
```

The system runs the if statement which only calculates the waiting time area during the time when the next one comes while the previous one has not finished. And the another part of area will be calculated after the service time later.

The code record the arrival time for the first method use firstly. Then save the change_time (representing env.now-status_change_time).

In the summ variable(sum can not be directly used because of the original function ‘sum’), the queue_length will minus one to cancel the effect of the service one.

The code of the average waiting time and service time for different arrival rates in task 1 & 2 is shown as below:

```
arrival_rate = 0.05
# The while loop can calculate the average waiting time and service time for different arrival rates
while arrival_rate < 1 :
    mean_ia_time = 1 / arrival_rate
    # the first way
    mean_waiting_time = run_simulation(mean_ia_time, mean_srv_time, num_packets, random_seed, trace)
    mean_waiting_times.append(mean_waiting_time)
    # the second way
    average_length = summ / status_change_time
    al_waiting_time = average_length / arrival_rate
    al_waiting_times.append(al_waiting_time)
    servtime = np.mean(service_times)
    servtimes.append(servtime)

    #start the next loop
    arrival_rate = arrival_rate + 0.05
    summ=0
    status_change_time=0
    queue_length=0
```

Firstly set the initial arrival rate value as 0.05, which represents the first value in Arrival rate: 5, 10, 15, ...95. Then use the run_simulation function to generate 19 mean_waiting_time values and combine them as a matrix by using append function, which is useful in the first module. in a similar way, generate the ‘al_waiting_time’ , which represents average waiting time matrix, and ‘servtimes’, which represents average service time matrix, variables which is significant in the second module. Finally, the while loop should be continued by resetting the arrival_rate, summ, status_change_time, queue_length(summ, status_change_time, queue_length are set as global variables). It breaks when arrival_time equals 1.

RESULTS

The result is displayed in Figure 4,5 as below;

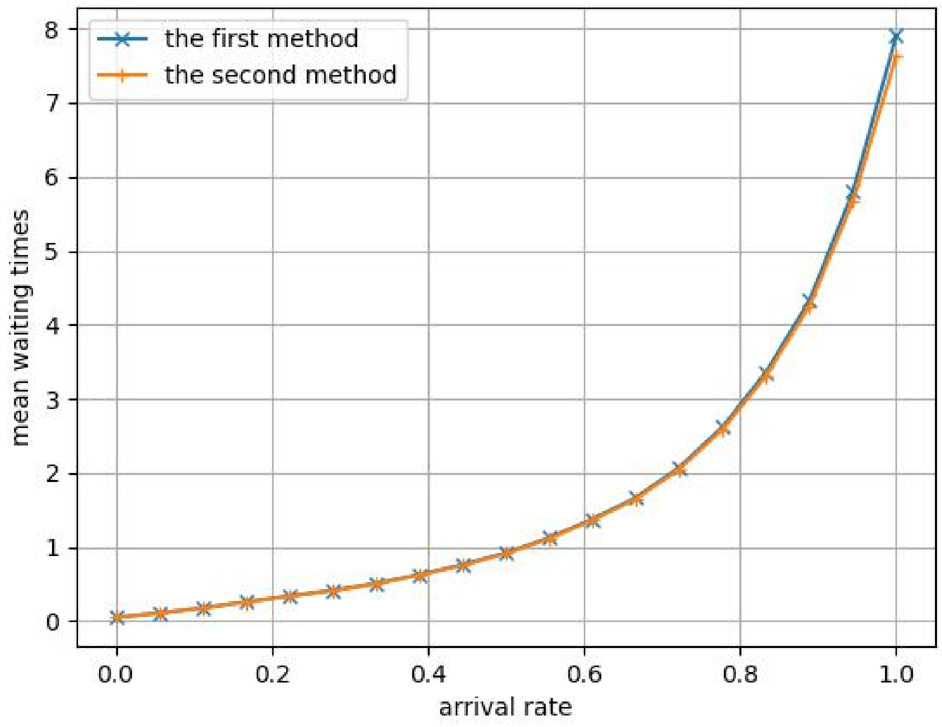

Figure 4

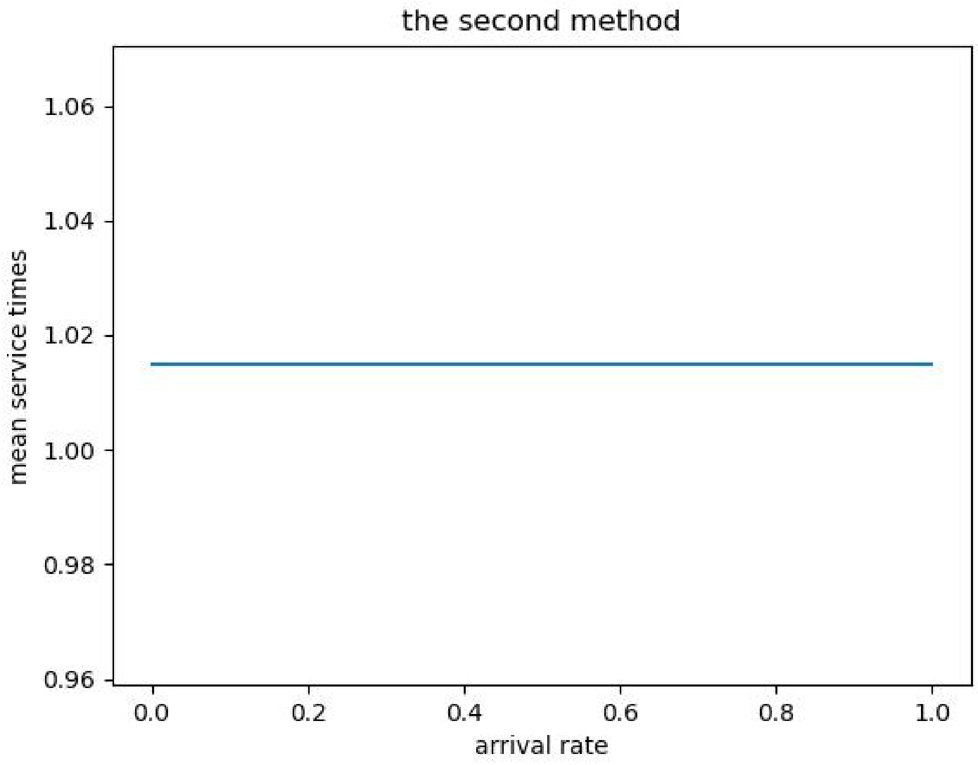

Figure 5

It can be seen from Figure 4 that with the arrival rate becoming faster, the mean waiting time is shown as exponential growth both in two methods. And the mean waiting times of two methods are similar in low frequency arrival rate, while the mean waiting times of the first method becomes a bit larger than that in the second method when the arrival rate is faster than 0.8.

In Figure 5, the mean service times does not change with the variation of the arrival rate. It will be the same when it is in the first method. It can be explained that only when the customers arrival rate goes fast and the mean service time remains the same, the mean waiting time can exponential increase. It conforms to the common sense of life.

It can be speculated that due to the inaccuracy of the first method(The algorithm does not count the number of people without waiting time and just use the average waiting time). So it shows up at high frequency.

CONCLUSION

Through this experiment, I have mastered the basic method for simple python programming, and for related functions also have a deep understanding, and can skilled application. The most important, is to master the internal principle of queueing theory and Little's theorem. Fundamentally find the changes between different code and function, instead of limit to the surface of function application

.
At the same time, in the experimental process, it also appeared many problems, such as the influence of parameters on the results, similar functions, differences, etc.. But through reading related books, search information, constantly test, I solved problems one by one. It is a good exercise of finding the problem, the ability to solve the problem.

Finally, thanks to the teacher giving us such an opportunity to learn, exercise, guidance. In the learning process in the future, I will make persistent efforts, fight for better complete of the task.

CODE

The codes are displayed as below;

```
# -*- coding: utf-8 -*-
"""
# @file         mm1_lab1.py
# @author       Enge.Xu
# @date         2018-11-29
#
```

```
# @brief        Simulate M/M/1 queueing system
#
"""

import argparse
import numpy as np
import random
import simpy
import sys
import matplotlib.pyplot as plt



def source(env, mean_ia_time, mean_srv_time, server, wait_times, number, trace):
    """Generates packets with exponential interarrival time."""
    for i in range(number):
        ia_time = random.expovariate(1.0 / mean_ia_time)
        srv_time = random.expovariate(1.0 / mean_srv_time)
        pkt = packet(env, 'Packet-%d' % i, server, srv_time, wait_times, trace)
        env.process(pkt)
        yield env.timeout(ia_time)


def packet(env, name, server, service_time, wait_times, trace):
    """Requests a server, is served for a given service_time, and leaves the server."""
    global summ,queue_length,status_change_time

    # record arrival time
    arrv_time = env.now
    if trace:

        change_time= env.now - status_change_time

        # update at each packet arrival
        if queue_length > 0:
            # only waiting people will be calculate
            summ=summ+(queue_length-1)*(change_time)

        queue_length = queue_length + 1

        status_change_time=env.now

    with server.request() as request:
        yield request
        # save wait time in a matrix
        wait_time = env.now - arrv_time
        wait_times.append(wait_time)

        yield env.timeout(service_time)
        # update at the end of each packet’s waiting in the queue
        if trace:

            change_time = env.now - status_change_time

            if queue_length > 1:
                # only waiting people will be calculate
                summ = summ+(queue_length - 1)*(change_time)

            queue_length = queue_length - 1
            service_times.append(service_time)
            status_change_time = env.now


def run_simulation(mean_ia_time, mean_srv_time, num_packets=1000,
random_seed=1234, trace=True):
    """Runs a simulation and returns statistics."""
    random.seed(random_seed)
    env = simpy.Environment()
    # start processes and run
    server = simpy.Resource(env, capacity=1)
    wait_times = []
    env.process(source(env, mean_ia_time,
                       mean_srv_time, server, wait_times,
number=num_packets, trace=trace))
    env.run()

    # return mean waiting time
    return np.mean(wait_times)

# Initialize all variables
if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument(
        "-A",
        "--mean_ia_time",
        help="mean packet interarrival time [s]; default is 1",
        default=1,
        type=float)
```

```
parser.add_argument(
        "-S",
        "--mean_srv_time",
        help="mean packet service time [s]; default is 1",
        default=1,
        type=float)
parser.add_argument(
        "-N",
        "--num_packets",
        help="number of packets to generate; default is 1000",
        default=1000,
        type=int)
parser.add_argument(
        "-R",
        "--random_seed",
        help="seed for random number generation; default is 1234",
        default=1234,
        type=int)
parser.add_argument(
        "-SUMM",
        "--summ",
        help="the sum of the rectangles' area; default is 0",
        default=0,
        type=float)
parser.add_argument(
        "-QL",
        "--queue_length",
        help="the number of packets in the queue; default is 0",
        default=0,
        type=float)
parser.add_argument(
        "-SCT",
        "--status_change_time",
        help="the time interval between changes in this number; default is 0",
        default=0,
        type=float)

parser.add_argument('--trace', dest='trace', action='store_true')
parser.add_argument('--no-trace', dest='trace', action='store_false')
parser.set_defaults(trace=True)

args = parser.parse_args()
# set variables using command-line arguments
mean_ia_time = args.mean_ia_time
```

```
    mean_srv_time = args.mean_srv_time
    num_packets = args.num_packets
    random_seed = args.random_seed
    trace = args.trace
    summ=args.summ
    queue_length=args.queue_length
    status_change_time=args.status_change_time

    mean_waiting_times=[]
    al_waiting_times=[]
    service_times=[]
    servtime=[]
    servtimes=[]

    arrival_rate = 0.05
    # The while loop can calculate the average waiting time and service time for different arrival rates
    while arrival_rate < 1 :
        mean_ia_time = 1 / arrival_rate
        # the first way
        mean_waiting_time = run_simulation(mean_ia_time, mean_srv_time, num_packets, random_seed, trace)
        mean_waiting_times.append(mean_waiting_time)
        # the second way
        average_length = summ / status_change_time
        al_waiting_time = average_length / arrival_rate
        al_waiting_times.append(al_waiting_time)
        servtime = np.mean(service_times)
        servtimes.append(servtime)

    #start the next loop
        arrival_rate = arrival_rate + 0.05
        summ=0
        status_change_time=0
        queue_length=0

    # plot the task1 figure
    plt.figure(1)
    y1=[]
    y1=np.squeeze(mean_waiting_times)
    x = np.linspace(0,1,19)
    plt.plot(x,y1, "x-", label="the first method")
    plt.ylabel('mean waiting times')
    plt.xlabel('arrival rate')
    y2=[]
    y2=np.squeeze(al_waiting_times)
    plt.plot(x,y2, "+-", label="the second method")
    plt.grid(True)

    plt.legend(loc = 0)

    plt.show()

    # plot the task2 figure
    plt.figure(2)
    y2=[]
    y2=np.squeeze(servtimes)
    x = np.linspace(0,1,19)
    plt.plot(x,y2)
    plt.ylabel('mean service times')
    plt.xlabel('arrival rate')
    plt.title("the second method")
    plt.show()


print("Average waiting time = %.4Es\n" % mean_waiting_time)
```

USE INSTRUCTION

Firstly, download and install WinPython distribution is a requirement. Secondly, open the lab1.py file and the two subtask is combined in this file. Spyder or Ipython is recommended.