In this project, some external code is referred, information as following:

Author/Origin: COMP9334 M/M/m Simulation code

Date: Week 4/5 Lecture

URL:https://webcms3.cse.unsw.edu.au/files/789d478ced0923d768b6740e8e5cd65fc552a68869ebe7c5d97a49847269e57f/attachment

Changes:

1) new mode included as 'trace' while the original code refers to mode 'random'.

2) To better calculate the response time, response time cumulative is separated into two variables:
   response_time_cumulative_0 and response_time_cumulative_1,
   respectfully, num_customer_served_0 and num_customer_served_1,
   Queue0 and queue1,
   Queue0_length and queue1_length.

3) In random, service time is generated randomly according to Probability Density Function,
   Arrival time is generated randomly according to input parameters.

4) Two new initialised np array to indicate if the job is reprocessed and the service time of a to be killed job,
   A new initialised list to indicate if a job in queue1 (queue for server 1) is a reprocessed job (killed in group 0).

5) When job passed to group 0 from arrival or queue0, check if fit the time limit for group 0, and update departure time, reprocessed status accordingly.

6) New returned variable: record which is a list of three tuples consist of information about each departed job from the system. (arrive time, departing time, from group0 or group 1 or reprocessed by group 1)
   For instance: record = [(2.0000,7.0000,1),...]

7) While 'random' mode terminates when next event time is over the assigned end time, 'trace' mode will terminate when all jobs are finished and departed from the system.

8) This program main function takes in all parameters, and according to the mode input, the parameters of unselected mode will all set to default None. No invalid input handling is embedded, as indicated only valid input will be passed.

9) Verification parts are initially comment out in the code, remove # to implement verification code.

Verification of inter-arrival probability distribution:

Using the approached discussed in Week 4 & 5.

In the following program, develop 10000 random next_arrival_time (using test 4 parameters).

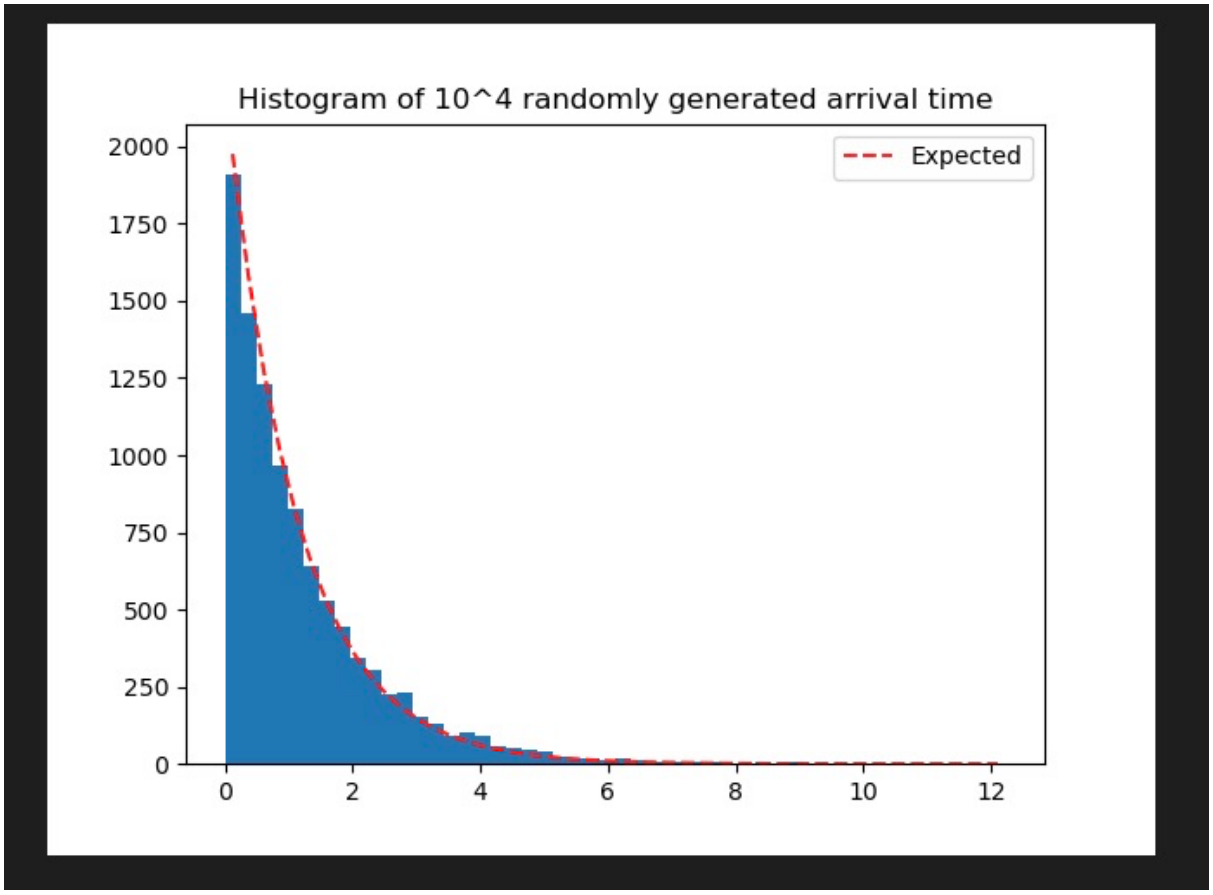
Plot them into a histogram to see if they are exponentially distributed.

Adjustments made according to this case from week 4 lecture code:

```
import numpy as np
import matplotlib.pyplot as plt
import random

#from test 4
lamb = 0.9
a2l = 0.91
a2u = 1.27

# Generate 10,000 random arrival times according to request
n = 10000
num = 0
u = []
while num < n :
    next_arrival_time = random.expovariate(lamb) * random.uniform(a2l,a2u)
    u.append(next_arrival_time)
    num = num+1

u = np.array(u)

# To check the numbers are really exponentially distributed
# Plot an histogram of the number
nb = 50 # Number of bins in histogram
freq, bin_edges = np.histogram(u, bins = nb, range=(0,np.max(u)))

# Lower and upper limits of the bins
bin_lower = bin_edges[:-1]
bin_upper = bin_edges[1:]
# expected number of exponentially distributed numbers in each bin
y_expected = n*(np.exp(-lamb*bin_lower)-np.exp(-lamb*bin_upper))
```

```
bin_center = (bin_lower+bin_upper)/2
bin_width = bin_edges[1]-bin_edges[0]

plt.bar(bin_center,freq,width=bin_width)
plt.plot(bin_center,y_expected,'r--',label = 'Expected')
plt.legend()
plt.title('Histogram of 10^4 randomly generated number')
plt.savefig('hist_expon.png')
```

the generated histogram as follow:



Similar tests on different test cases with different parameters, similar results yield indicating arrival times generated are exponentially distributed.

Verification of probability of sending job to group 0:

Following code is in code to store the number of jobs generated and the number of job assigned to group 0, tested on case 6(mostly) as it has a large end time:

```
        #for testing of probability of sending job to 1/0
        job_numbers = 0
        job_sent_to_0 = 0
        #
```

Every time a job is assigned to group 0:

```
job_sent_to_0 = job_sent_to_0 + 1
```

After simulation, print out information:

```
    #
    #test if the probability align with given parameter p0
    print(f'p0 = {p0}, and the actual percentage in this run is
{job_sent_to_0/job_numbers}')
    #
```

Run the test 10^4 times, printed as follow (fraction of print out):

```
(base) qz@QZs-MacBook-Air project_sample_files % python3 main.py 6
p0 = 0.89, and the actual percentage in this run is 0.8803376365441906
(base) qz@QZs-MacBook-Air project_sample_files % python3 main.py 6
p0 = 0.89, and the actual percentage in this run is 0.8845965770171149
(base) qz@QZs-MacBook-Air project_sample_files % python3 main.py 6
p0 = 0.89, and the actual percentage in this run is 0.8876739562624254
(base) qz@QZs-MacBook-Air project_sample_files % python3 main.py 6
p0 = 0.89, and the actual percentage in this run is 0.8914313534566699
(base) qz@QZs-MacBook-Air project_sample_files % python3 main.py 6
p0 = 0.89, and the actual percentage in this run is 0.8936484490398818
(base) qz@QZs-MacBook-Air project_sample_files % python3 main.py 6
p0 = 0.89, and the actual percentage in this run is 0.8849424712356178
(base) qz@QZs-MacBook-Air project_sample_files % python3 main.py 6
p0 = 0.89, and the actual percentage in this run is 0.8807385229540918
(base) qz@QZs-MacBook-Air project_sample_files % python3 main.py 6
p0 = 0.89, and the actual percentage in this run is 0.8872672370407649
(base) qz@QZs-MacBook-Air project_sample_files % python3 main.py 6
p0 = 0.89, and the actual percentage in this run is 0.8998005982053838
(base) qz@QZs-MacBook-Air project_sample_files % python3 main.py 6
p0 = 0.89, and the actual percentage in this run is 0.8885557221389305
(base) qz@QZs-MacBook-Air project_sample_files % python3 main.py 6
p0 = 0.89, and the actual percentage in this run is 0.8956478733926805
(base) qz@QZs-MacBook-Air project_sample_files % python3 main.py 5
p0 = 0.82, and the actual percentage in this run is 0.8262032085561497
(base) qz@QZs-MacBook-Air project_sample_files % python3 main.py 5
p0 = 0.82, and the actual percentage in this run is 0.8236057068741893
(base) qz@QZs-MacBook-Air project_sample_files % python3 main.py 5
p0 = 0.82, and the actual percentage in this run is 0.8151595744680851
(base) qz@QZs-MacBook-Air project_sample_files % python3 main.py 5
p0 = 0.82, and the actual percentage in this run is 0.8177842565597667
(base) qz@QZs-MacBook-Air project_sample_files % python3 main.py 5
p0 = 0.82, and the actual percentage in this run is 0.7951977401129944
(base) qz@QZs-MacBook-Air project_sample_files % python3 main.py 5
p0 = 0.82, and the actual percentage in this run is 0.816711590296496
(base) qz@QZs-MacBook-Air project_sample_files % python3 main.py 4
p0 = 0.7, and the actual percentage in this run is 0.7515151515151515
(base) qz@QZs-MacBook-Air project_sample_files % python3 main.py 4
p0 = 0.7, and the actual percentage in this run is 0.734375
(base) qz@QZs-MacBook-Air project_sample_files % python3 main.py 4
p0 = 0.7, and the actual percentage in this run is 0.7368421052631579
(base) qz@QZs-MacBook-Air project_sample_files % python3 main.py 4
p0 = 0.7, and the actual percentage in this run is 0.7298850574712644
(base) qz@QZs-MacBook-Air project_sample_files % python3 main.py 4
p0 = 0.7, and the actual percentage in this run is 0.75
```

The actual percentage is within tolerate(normal) range in all tests.

Verification of service time distribution:

Using the approached discussed in Week 4 & 5.

In the following program, develop 10000 random service_time (using test 5 parameters).

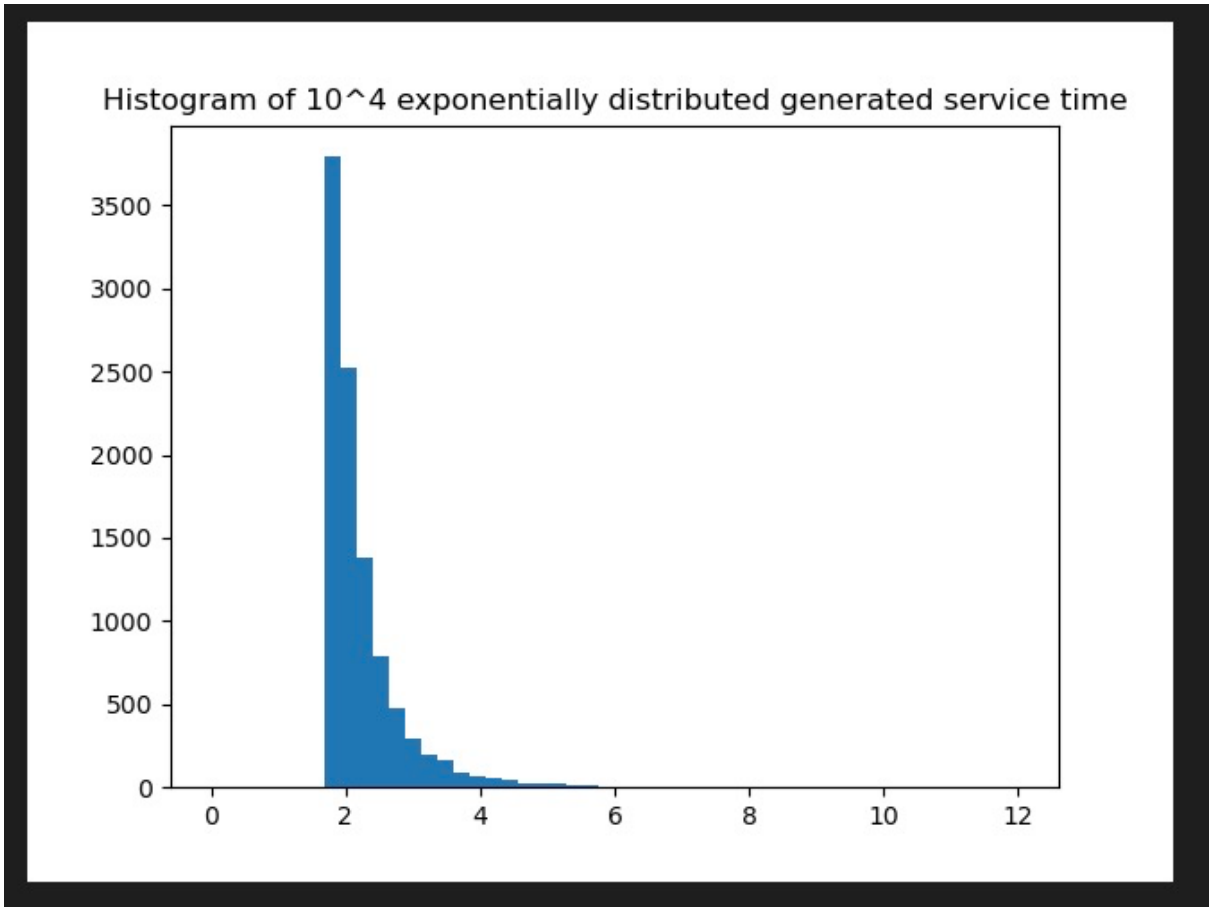
Plot them into a histogram to see if they are exponentially distributed.

Adjustments made according to this case from week 4 lecture code:

```
import numpy as np
import matplotlib.pyplot as plt
import random

#from test 5
alpha0 = 1.4
alpha1 = 3.2
eta0 = 3.4
eta1 = 3.7
beta0 = 4.1
sent = 0 # or 1 in test for group 1 servers

# Generate 10,000 random service times according to PDF
n = 10000
num = 0
x = []
while num < n :
    u = random.random()
    if sent:
        tmp = u * (alpha1**(-eta1))
        service_time_next_arrival = (eta1/tmp) ** (1/(eta1+1))
    else:
        tmp = u*(alpha0**(-eta0) - beta0**(-eta0))
        service_time_next_arrival = (eta0/tmp) ** (1/(eta0+1))

    x.append(service_time_next_arrival)

    num = num+1

x = np.array(x)

# To check the numbers are really exponentially distributed
# Plot an histogram of the number
nb = 50 # Number of bins in histogram
freq, bin_edges = np.histogram(x, bins = nb, range=(0,np.max(x)))

# Lower and upper limits of the bins
bin_lower = bin_edges[:-1]
bin_upper = bin_edges[1:]
```

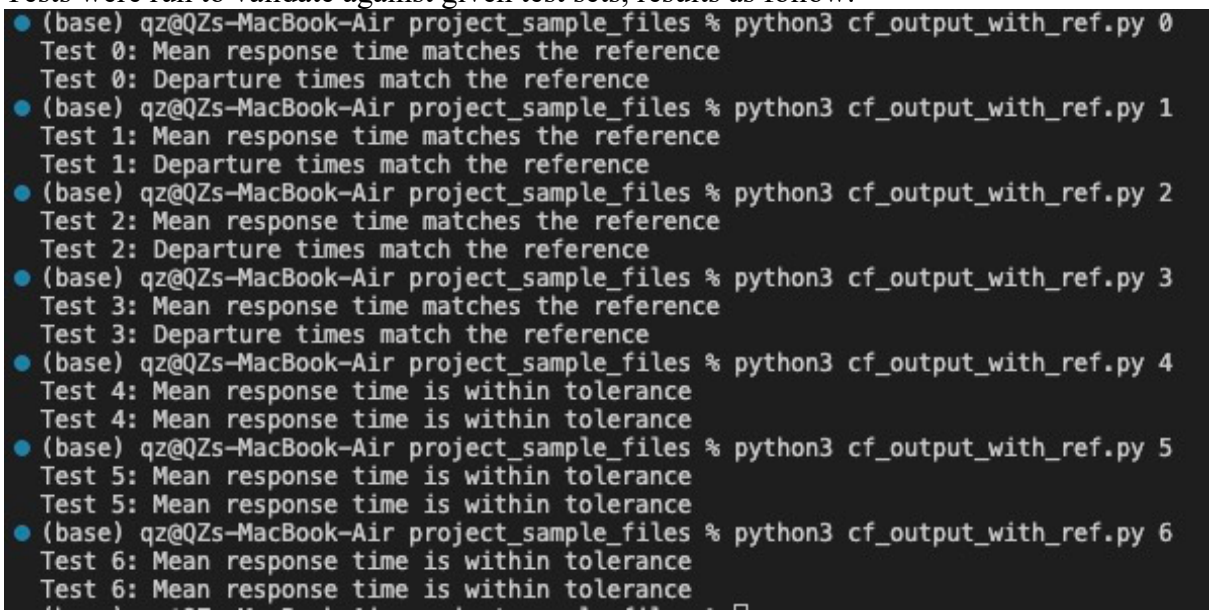
```
bin_center = (bin_lower+bin_upper)/2
bin_width = bin_edges[1]-bin_edges[0]

plt.bar(bin_center,freq,width=bin_width)
plt.title('Histogram of 10^4 exponentially distributed generated service time')
plt.savefig('hist_expon.png')
```

the generated histogram as follow:



When switch sent = 1 to generate service time for group 1 servers, similar results and histogram were yield. Thus finished verification of generated service times.

Verification of simulation code:
Tests were run to validate against given test sets, results as follow:



Other test code was run parameters as follow:

Interarrival: 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000.
Mode: trace
Para: m = 2, n0 = 1, time_limit = 5
Service: 5.000 0, 6.000 0, 6.000 0, 6.000 0, 6.000 0, 6.000 0, 6.000 0, 6.000 0, 6.000 1.

Test if the simulation code handles the queue of job sent to group 0 correctly, as in this test, all jobs except last one will be sent to group 0,

If the jobs are killed correctly, as all jobs are over time limit for group 0 except first one,

If the jobs are reprocessed correctly by group 1, and the queuing behaviour of queue 1 is correct,
as all jobs will need to be handled by group 1 except first one (reprocessed or passed straight from server).

Results are correct as follow:

Mrt.txt:
5.0000 6.0000

Depart.txt:
1.0000 6.0000 0
9.0000 15.0000 1
2.0000 21.0000 r0
3.0000 27.0000 r0
4.0000 33.0000 r0
5.0000 39.0000 r0

6.0000 45.0000 r0
7.0000 51.0000 r0
8.0000 57.0000 r0

Reproducibility:

When in 'random' mode,
Included following code to save the random state:

```
import pickle
import random
rand_state = random.getstate()
pickle.dump( rand_state, open( "simulation_currernt_state.p", "wb" ) )
```

mean response time for group 0 and group 1 in this simulation are:
2.2375 4.5191

and run the random program again with the same parameters and include the following code to access the same random state:

```
import pickle
import random
rand_state = pickle.load( open( "simulation_current_state.p", "rb" ) )
random.setstate(rand_state)
```

same result from this program run:
2.2375 4.5191

Suitable n0 value in system:

Given the following testing parameters:

- Total number of servers:

  $n = 10$
- The service time limit for Group 0 servers:

  $T_{limit} = 3.3$.
- For inter-arrival times:

  $\lambda = 3.1$, $a_{2l} = 0.85$, $a_{2u} = 1.21$
- The probability that a job is indicated for a Group 0 server:

  $p_0 = 0.74$.
- The service time for a job which is indicated for Group 0:

  $\alpha_0 = 0.5$, $\beta_0 = 5.7$, $\eta_0 = 1.9$.

• The service time for a job which is indicated for Group 1:

$\alpha_1 = 2.7$ and $\eta_1 = 2.5$.

• weighting for group 0 and group 1 service time:

w0 = 0.83 w1 = 0.059

Choose the time_end = 1000 as a longer simulation length.

At the initialisation of simulation, assign two lists to store the service time for group 0 and group 1 respectively.

```
    #
    #For transient part removal
    traces_0 = []
    traces_1 = []
    #
```

Every time a group 0 job is finished:

```
            #
            # Store response time for transient removal
            traces_0. append(master_clock –
arrival_time_next_departure[first_departure_server])
            #
```

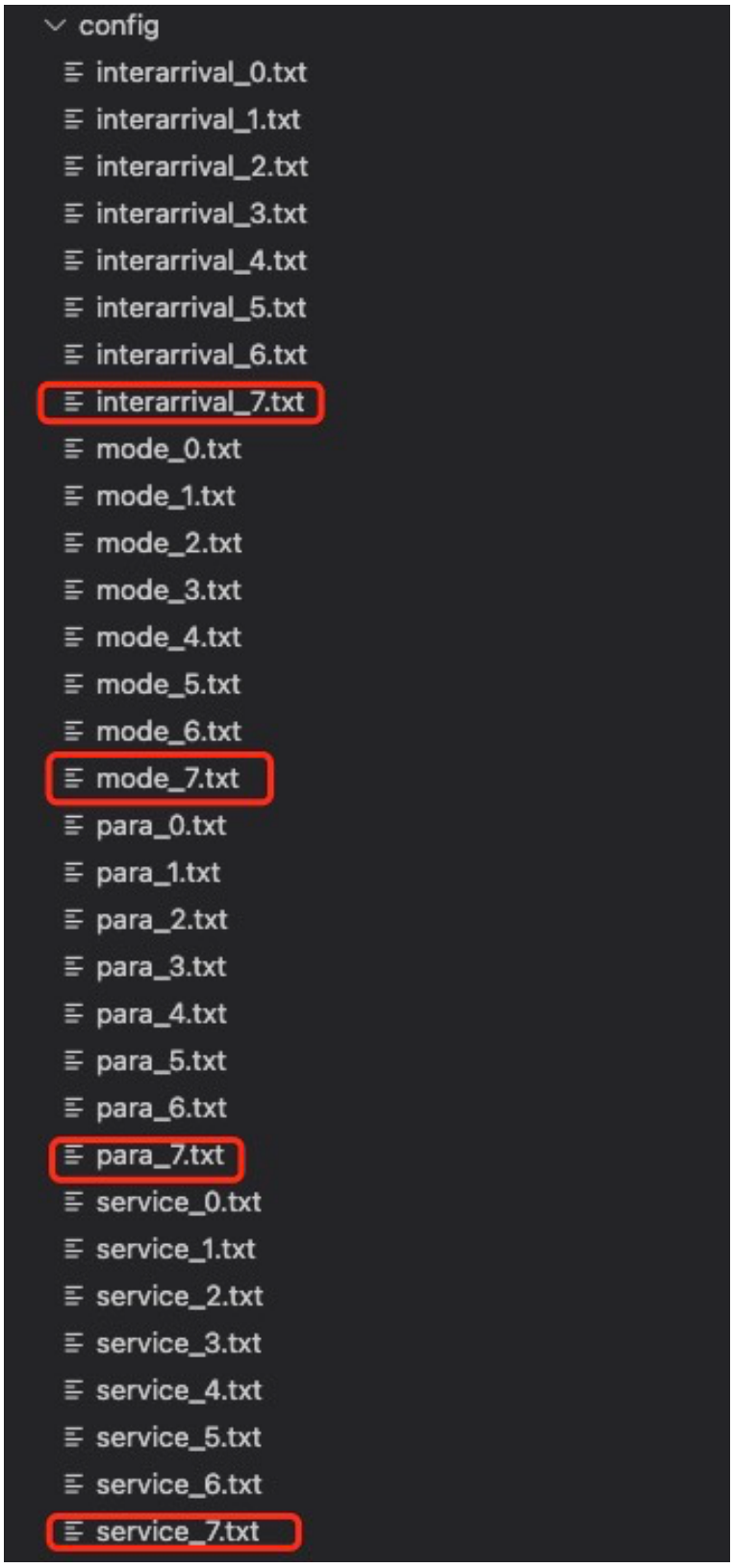
Every time a group 1 job is finished:

```
            #
            # Store response time for transient removal
            traces_1. append(master_clock –
arrival_time_next_departure[first_departure_server])
            #
```

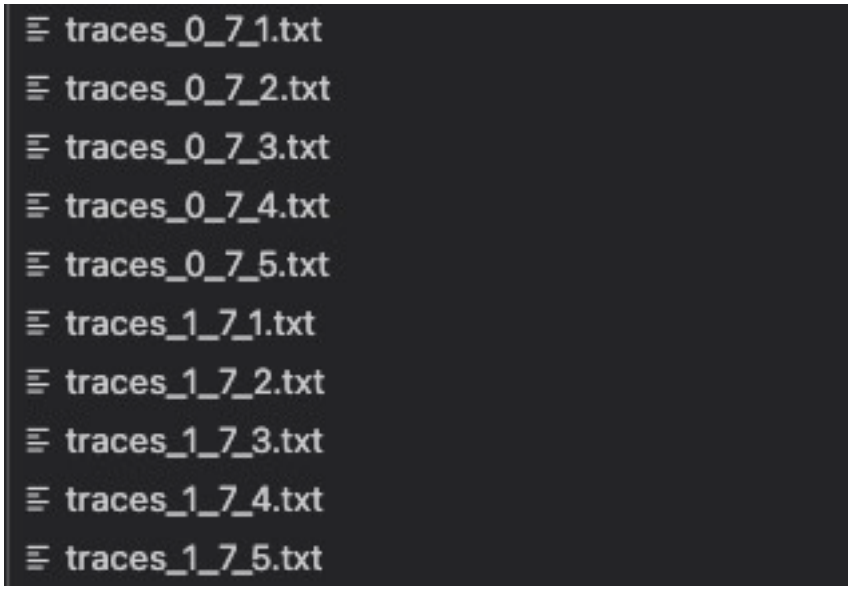
At the end of the simulation out put two txt files with service time for group 0 and group 1 respectively:

```
    #
    #For transient removal
    traces_0_file = os.path.join(out_folder,'traces_0_'+s+'.txt')
    traces_1_file = os.path.join(out_folder,'traces_1_'+s+'.txt')
    with open(traces_0_file,'w') as file:
        for trace in traces_0:
            file.writelines(str('{:.4f}'.format(trace))+'\n')
    with open(traces_1_file,'w') as file:
        for trace in traces_1:
            file.writelines(str('{:.4f}'.format(trace))+'\n')
    #
```

Create corresponding txt files with required parameters indicated, numbered as test 7 in config directory. Later in the simulation, only n0 parameter will change from 1 -9 while everything else stays the same.



Run the simulation 5 times with last digit of output txt file indicate simulation number as following:
assume: n0 = 5 in this transient remove part
Record the first 2000 results from traces 0
Similar test can be run on results from traces 1 too.

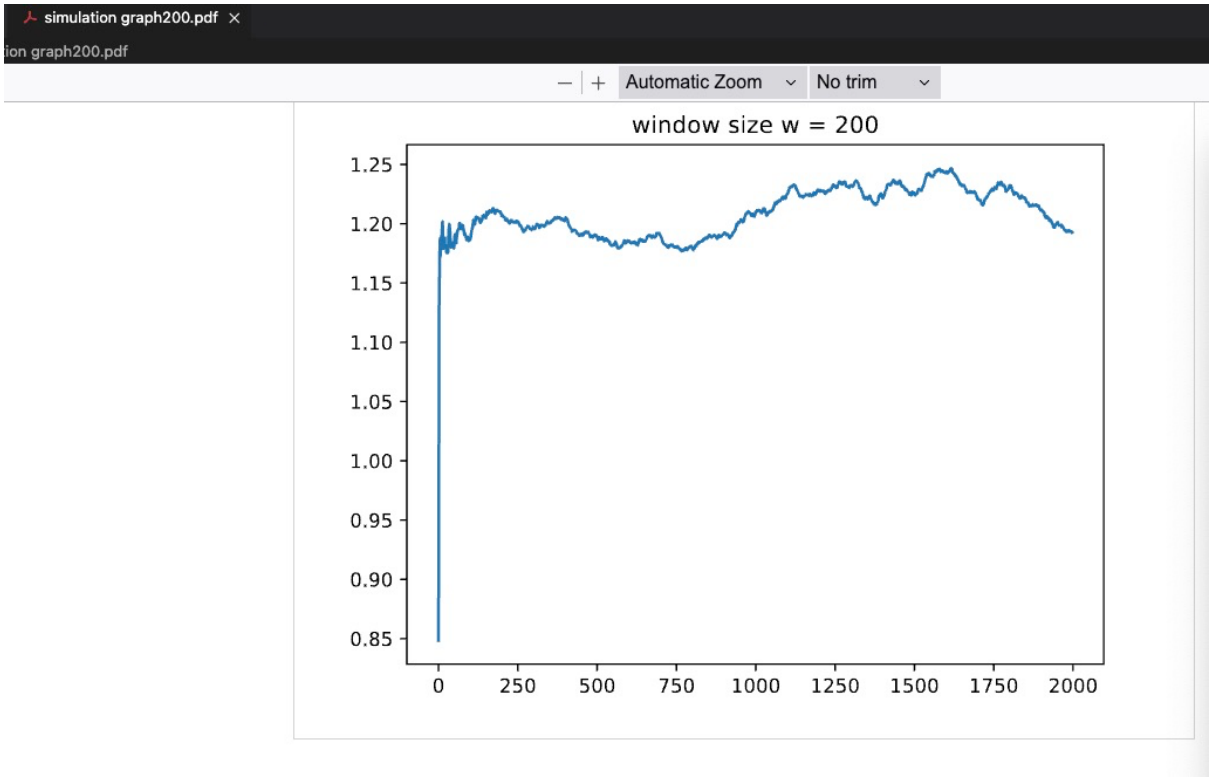


And run the following program to generate a pdf file:

```
import numpy as np
import matplotlib.pyplot as plt

# Put the traces in a numpy array
nsim = 5     # number of simulation
m = 2198   # number of data points in each simulation
response_time_traces = np.zeros((nsim,m))


# load the traces
for i in range(5):
    response_time_traces[i,:] = np.loadtxt('traces_0_7_'+str(i+1)+'.txt')

# Compute the mean over the 5 replications
mt = np.mean(response_time_traces,axis = 0)

# smooth it out with different values of w
# vary the value of w here
w = 200
mt_smooth = np.zeros((m-w,))

for i in range(m-w):
    if (i < w):
        mt_smooth[i] = np.mean(mt[:(2*i+1)])
    else:
        mt_smooth[i] = np.mean(mt[(i-w):(i+w)])
```

```
plt.plot(np.arange(m-w),mt_smooth)
plt.title('window size w = ' + str(w))
plt.savefig('week05B_q1_a_'+str(w)+'.pdf')
```

PDF as follow:



In a 2000 sample, remove the first 100 as first 5% of the data to conclude relatively stable state.

Same transient removal applied to group 1 service time.

And recalculate the mean response time for both (traces 0 and traces 1):

```
# Drop the first 100 points as the transient
# Compute the mean from data points 101 to 2000
mt = np.mean(response_time_traces[:,100:],axis=1)
```

Now transient removal is implemented, start simulation for different n0 8 times each with different random state and implement following code for print out system service time (w0*T0 + w1*T1):

```
    #For the design Probelm
    #
    w0 = 0.83
    w1 = 0.059
    print(mean_response_time_0 * w0 + mean_response_time_1 * w1)
    #
```

Record printed out value each simulation.

Results are:

n0 = 1	255.4681	256.9938	267.5571	277.7778	261.8503	247.3947	257.7656	243.0247
n0 = 2	115.9830	116.9345	109.3999	115.0480	106.8980	106.7058	104.1080	114.0377
n0 = 3	2.0378	2.7114	2.2161	2.3571	2.0405	2.3168	2.0971	2.5007
n0 = 4	1.3385	1.3456	1.3725	1.3042	1.3622	1.3354	1.3721	1.3512
n0 = 5	1.2385	1.2182	1.2405	1.2166	1.2363	1.2293	1.2267	1.2393
n0 = 6	1.2514	1.2761	1.2463	1.2299	1.2626	1.2584	1.2380	1.2589
n0 = 7	3.5463	1.6443	2.2030	2.4591	2.2128	1.9963	3.2049	1.7564
n0 = 8	11.2547	12.3897	13.7304	13.5199	12.2453	11.7397	11.3209	12.7926
n0 = 9	22.0430	21.1709	21.2049	20.0498	19.1446	20.9306	20.7089	19.9358

Run the following program, to compare only adjacent systems first according to its 95% confidence interval of system service time.

```
# import
import numpy as np
from scipy.stats import t

#%% The given data
# store the mean response times in 2-D numpy arrays
# Each column is for a system
mrt_sys = np.array([ [255.4681, 115.9830, 2.0378, 1.3385, 1.2385, 1.2514, 3.5463,
11.2547, 22.0430],
                     [256.9938, 116.9345, 2.7114, 1.3456, 1.2182, 1.2761, 1.6443,
12.3897, 21.1709],
                     [267.5571, 109.3999, 2.2161, 1.3725, 1.2405, 1.2463, 2.2030,
13.7304, 21.2049],
                     [277.7778, 115.0480, 2.3571, 1.3042, 1.2166, 1.2299, 2.4591,
13.5199, 20.0498],
                     [261.8503, 106.8980, 2.0405, 1.3622, 1.2363, 1.2626, 2.2128,
12.2453, 19.1446],
```

```
                    [247.3947, 106.7058, 2.3168, 1.3354, 1.2293, 1.2584, 1.9963,
11.7397, 20.9306],
                    [257.7656, 104.1080, 2.0971, 1.3721, 1.2267, 1.2380, 3.2049,
11.3209, 20.7089],
                    [243.0247, 114.0377, 2.5007, 1.3512, 1.2393, 1.2589, 1.7564,
12.7926, 19.9358]])
#the significance
p = 0.95

#%% Solution
# Compute the following differenes
# System n0=1 - System n0=2
dt12 = mrt_sys[:,0]- mrt_sys[:,1]
# System n0=2 - System n0=3
dt23 = mrt_sys[:,1]- mrt_sys[:,2]
# System n0=3 - System n0=4
dt34 = mrt_sys[:,2]- mrt_sys[:,3]
# System n0=4 - System n0=5
dt45 = mrt_sys[:,3]- mrt_sys[:,4]
# System n0=5 - System n0=6
dt56 = mrt_sys[:,4]- mrt_sys[:,5]
# System n0=6 - System n0=7
dt67 = mrt_sys[:,5]- mrt_sys[:,6]
# System n0=7 - System n0=8
dt78 = mrt_sys[:,6]- mrt_sys[:,7]
# System n0=8 - System n0=9
dt89 = mrt_sys[:,7]- mrt_sys[:,8]

# multiplier for confidence interval
num_tests = mrt_sys.shape[0]
mf = t.ppf(1-(1-p)/2,num_tests-1)/np.sqrt(num_tests)

# To compute the confidence interval
pm1 = np.array([-1,1])

# confidence interval for dt12
ci12 = np.mean(dt12) + pm1 * np.std(dt12, ddof=1) * mf

# confidence interval for dt23
ci23 = np.mean(dt23) + pm1 * np.std(dt23, ddof=1) * mf

# confidence interval for dt34
ci34 = np.mean(dt34) + pm1 * np.std(dt34, ddof=1) * mf

# confidence interval for dt45
ci45 = np.mean(dt45) + pm1 * np.std(dt45, ddof=1) * mf

# confidence interval for dt56
ci56 = np.mean(dt56) + pm1 * np.std(dt56, ddof=1) * mf

# confidence interval for dt67
```

```
ci67 = np.mean(dt67) + pm1 * np.std(dt67, ddof=1) * mf

# confidence interval for dt78
ci78 = np.mean(dt78) + pm1 * np.std(dt78, ddof=1) * mf

# confidence interval for dt89
ci89 = np.mean(dt89) + pm1 * np.std(dt89, ddof=1) * mf

print(ci12,ci23,ci34,ci45,ci56,ci67,ci78,ci89)
```

printed results are:

[137.61315145 157.06614855]
[104.83042395 112.87892605]
[0.73182124 1.14212876]
[0.10118425 0.13289075]
[-0.03579451 -0.00825549]
[-1.69602581 -0.55434919]
[-11.17668368 -8.81584132]
[-9.51886272 -7.02996228]

All positive number indicate one system is slower than the other,

we can draw conclusion that:

n0 = 1 require more systematic service time (w0*T0 + w1*T1) than n0 = 2,

respectively, no = 2 require more systematic service time than n0 = 3,

no = 3 require more systematic service time than n0 = 4,

no = 4 require more systematic service time than n0 = 5,

All negative number indicate one system is faster than the other,

we can draw conclusion that:

n0 = 5 require less systematic service time (w0*T0 + w1*T1) than n0 = 6,

respectively, n0 = 6 require less systematic service time than n0 = 7,

n0 = 7 require less systematic service time than n0 = 8,

n0 = 8 require less systematic service time than n0 = 9.

Thus, when n0 = 5, the system have a minimised system service time (w0*T0 + w1*T1).