1.Introduction

The report comprises the following details:

----simulation function of the fog-cloud system with processor sharing----

----explain for each part of the simulation function

----determine the “fogtimelimit” and analysis for itself

----the scripts for plotting

2 output examples

I choose 3 different seeds to do the simulation and get the total different outcomes, which is what I expect to.the seeds and mean response time are shown below:

1.seed = 1225 mean response time is 0.9503

2.seed = 1226 mean response time is 1.0002

3. seed = 1 mean response time is 1.0868

3.sepecification of simulation

The simulation function in the program is

```
man(expected_num,arrival_list,service_list,network_la
tency_list,fogTimeLimit,TimeToCloud,time_end)
```

the expected _num is a parameter of the input, which represents how many cluster of request of jobs we need to read and process.

Arrival_list is the time array of the job arrivals, and it will follow the rule of poison distribution while the mode is “random”.

Service_list is the time array of the service time that jobs required. The parameter will be calculated in accordance with the arriving function when it goes to “random” mode(the function can be found in the project sheet v1.0)

Network_latency is the network latency for the jobs that will be processed, in the circumstance of the “random” mode, the parameter of latency will be fetched from
an array of the random number[v1, v2], and the details can be found in the project sheet
v1.0

fogTimeLimit means the time limit for each job in the cloud. Once the job can not be finished in the fog, it will be delivered in the cloud via the network.

The differences between the latencylist and TimeToCloud is that TimeToCloud is an input number, and this parameter will be useful when we calculate the arguments in from fog to cloud via network

Only by calling the “random” mode can the parameter “Time_end” available. While in the
situation that the mater clock is equal to time end , the system will stop processing and reject inputs .

4. the details about probability distribution of inter-arrival time and service time

(a) arrival time

In accordance with our lecture in the week 4B, the inter-arrival time can be estimated by the formulae in our lecture slides.
There are 2 main steps:
1, the Uk, which is the random arrry of numbers, it are uniformly distributed in (0,1)
2, the sequence -log(1 – Uk)/lambda is exponential distributed with the rate lambda
so I write the code for acquiring as follows:

```
def random_arrival(job_number, Lambda):
    Uk = numpy.random.uniform(0, 1, job_number)
    arrivalTime = -numpy.log(1-Uk)/Lambda
    for i in range(1, job_number):
        arrivalTime[i] = arrivalTime[i] +
arrivalTime[i-1]
    return arrivalTime
```

(b) service time

When is comes to the service time distribution, we have the following density function.

$$g(t) = \begin{cases} 0 & for\ t \le \alpha_1 \\ \frac{\gamma}{t^\beta} & for\ \alpha_1 \le t \le \alpha_2 \\ 0 & for\ t \ge \alpha_2 \end{cases}$$

Where α_1, α_2 $and\ \beta\ given\ in\ the\ input\ files\ and\ \gamma = \frac{1-\beta}{\alpha_2^{1-\beta} - \alpha_1^{1-\beta}}$

If we get the CDF of the service time by $CDF = \int_{-\infty}^{t} g(x)dx$, we can get the CDF as G(t) as below.

$$G(t) = \begin{cases} 0 & for\ t \le \alpha_1 \\ \frac{\gamma}{1-\beta}(t^{1-\beta} - \alpha_1^{1-\beta}) & for\ \alpha_1 \le t \le \alpha_2 \\ 1 & for\ t \ge \alpha_2 \end{cases}$$

The inverse function of G(t), i.e. invG(u) is calculated as below.

$$invG(\mu) = [\frac{1-\beta}{\gamma}\left(\mu + \frac{\gamma}{1-\beta}\alpha_1^{1-\beta}\right)]^{\frac{1}{1-\beta}} \quad where\ \mu\ \epsilon\ [0,1]$$

μ is uniformly distributed between [0, 1]. So we use a uniform distribution to create a distribution based on the inversed function invG(u) above.

In accordance with the above calculation, I write the code as follows:

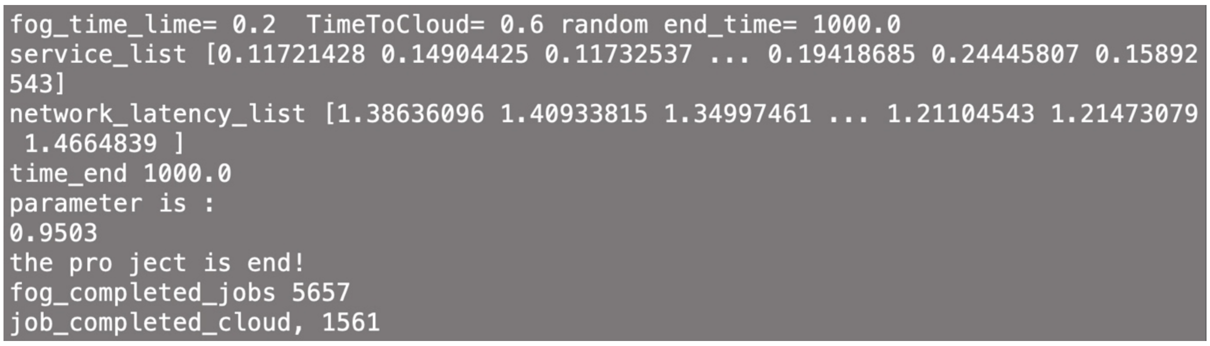
```
def random_service(Beta, Alpha_1, Alpha_2,
job_number):
    Prob = numpy.random.uniform(0, 1, job_number)
    gama = (1 - Beta) / ((Alpha_2 ** (1 - Beta)) -
(Alpha_1 ** (1 - Beta)))
    service_time = (Prob * (1 - Beta) / gama +
Alpha_1 ** (1 - Beta)) ** (1 / (1 - Beta))
    return service_time
```

(c) network latency

The network latency is uniformly distributed in [v1, v2]
So I write the code as follows:

```
def random_latency(v1, v2, job_number):
    latency_list =
numpy.random.uniform(v1,v2,job_number)
    return latency_list
```

In my test, when the seed is deployed as seed = 1225, the number o mean response time is close to the example, which is 0.9503



```
fog_time_lime= 0.2  TimeToCloud= 0.6 random end_time= 1000.0
service_list [0.11721428 0.14904425 0.11732537 ... 0.19418685 0.24445807 0.15892
543]
network_latency_list [1.38636096 1.40933815 1.34997461 ... 1.21104543 1.21473079
 1.4664839 ]
time_end 1000.0
parameter is :
0.9503
the pro ject is end!
fog_completed_jobs 5657
job_completed_cloud, 1561
```

5.Impletation and Justification

While doing the simulation, the most significant thing is to cope with the processor sharing mechanism. In the lecture , we talked about the principle of it , so I set fogRemain, netRemain and cloudRemain to represent the remain service time one job need on the particular component of the system. At the same , three array : fogList, netlist, and cloudList will append the arriving jobs which are satisfied with the place it should be . Once there are nothing in these lists, it means that now all jobs have been processed and we do not have to call the simulation function any more, then the loop will stop and the program will create the files of the parameters.

Furthermore, each job of index will also be marked as the process will do the job follow the order of their indexes.

The following is one of the request of jobs:

```
class Request:
    def __init__(self, jobOrder, fogArrival,
serviceTime, netLatency):
        self.fogRemain,
self.fogService ,self.jobOrder,  self.fogArrival,
self.netLatency = serviceTime, serviceTime, jobOrder,
fogArrival, netLatency
        self.fogPushed, self.netReceived,
self.netPushed, self.cloudReceived, self.cloudPushed
= 99999, 99999,99999,99999,99999
        self.jobCloudProcessed, self.jobFinished,
self.inLimit = False, False, True
        self.cloudRemain, self.jobRT = 0, 0
```

each job will be processed by its own parameters which have been given

For justifying my work, I choose the first trace mode to illuminate my calculation and output so that we can see the procedure of my thinking.

In the first file, there are 6 job arriving in total:

1.100
6.200
7.400
8.300
9.100
10.100

While the departure times in fog are:

3.1000
9.4333
11.2111
14.6611
15.1944
15.5000

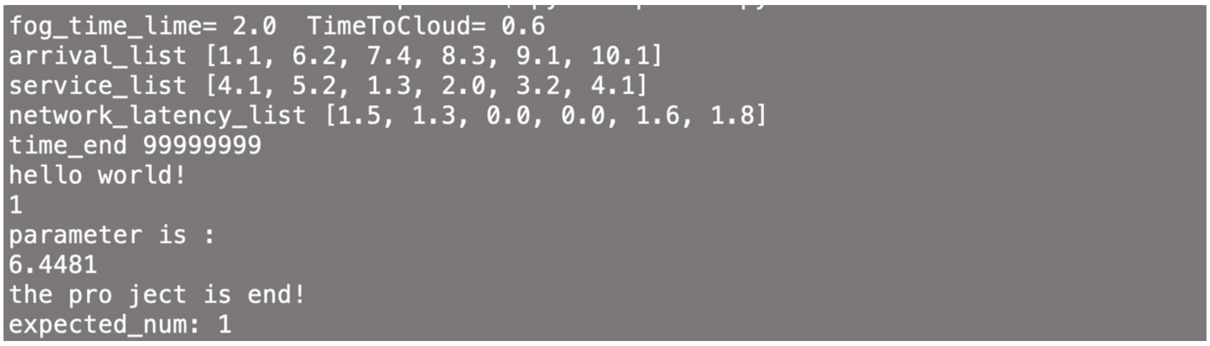
And the departure times in cloud are

5.8600
12.6533
17.7289
18.7744

According to the files, the mean response time should be :

$$mean\ response\ time$$
$$= \frac{(5.86 - 1.1) + (12.6533 - 6.2) + (11.2111 - 7.4) + (14.6611 - 8.3) + (17.7289 - 9 \quad (18.7744 - 10.1)}{6}$$
$$= 6.44813333\ seconds/request$$

Here is the output of the code:


```
fog_time_lime= 2.0  TimeToCloud= 0.6
arrival_list [1.1, 6.2, 7.4, 8.3, 9.1, 10.1]
service_list [4.1, 5.2, 1.3, 2.0, 3.2, 4.1]
network_latency_list [1.5, 1.3, 0.0, 0.0, 1.6, 1.8]
time_end 99999999
hello world!
1
parameter is :
6.4481
the pro ject is end!
expected_num: 1
```

Besides, the net departure and cloud departure also meet the reference.

So I make the conclusion that the simulation has finish the job successfully.

Here are the output of fog departure files:

```
1.1000      3.1000
6.2000      9.4333
7.4000      11.2111
8.3000      14.6611
9.1000      15.1944
10.100015.5000
```

Here are the output of net departure files:

```
1.1000     4.6000
6.2000     10.7333
9.1000     16.7944
10.1000 17.3000
```

Here are the output of cloud departure files:

```
1.1000		5.8600
6.2000		12.6533
9.1000		17.7289
10.100018.7744
```

6. Transient State selection

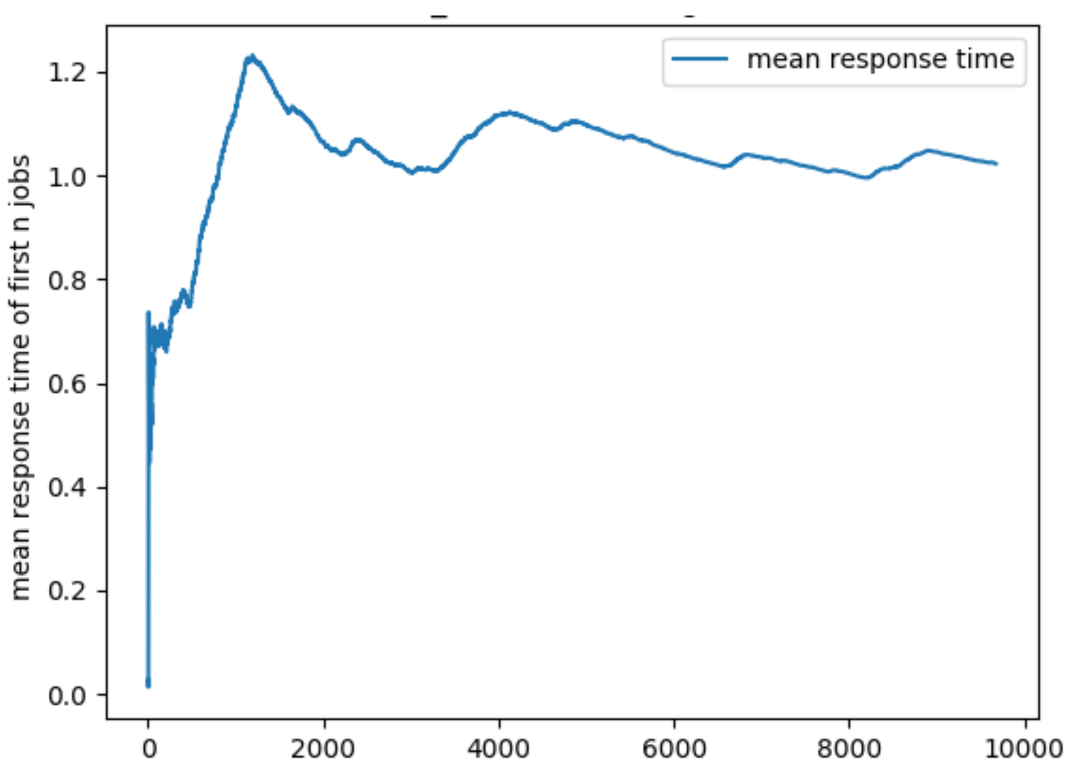
After checking the simulation function, ensuring its availability, I then start to determine the
optimal fogLimitTime . In this part, 3 steps are necessary:

1. pick the observation time, omitting the transient state

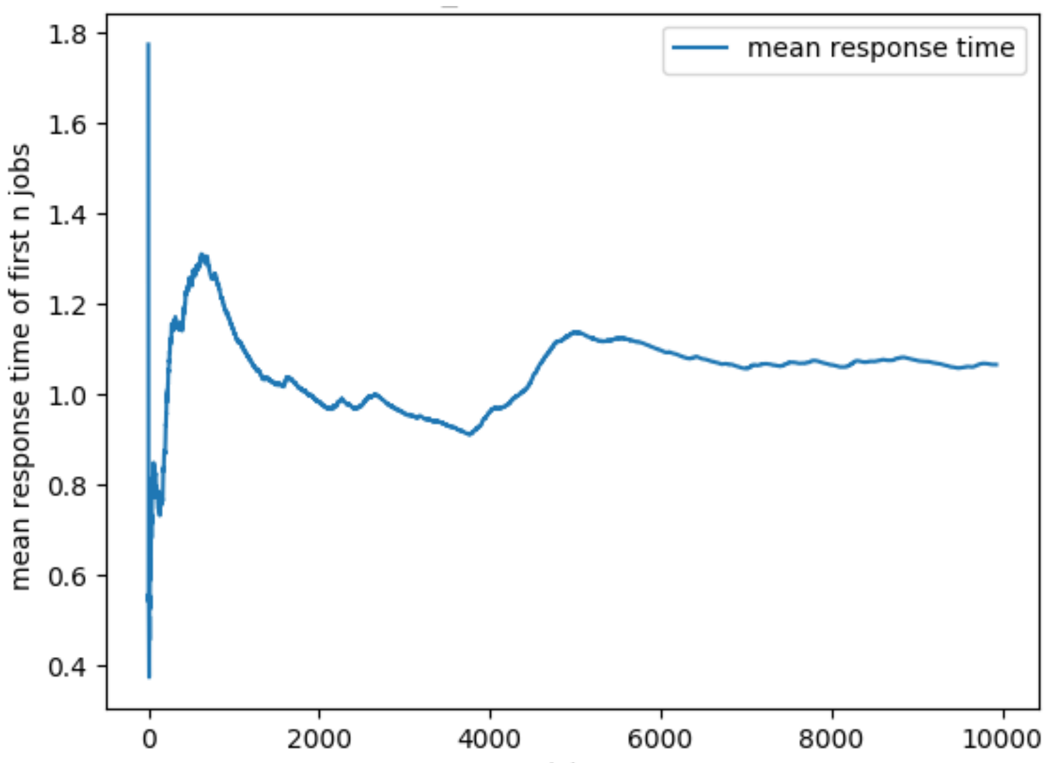
2. pick the possible range that the optimal fogLimitTime might be

3. pick the specific fogLimittime from the possible range via checking confidence interval

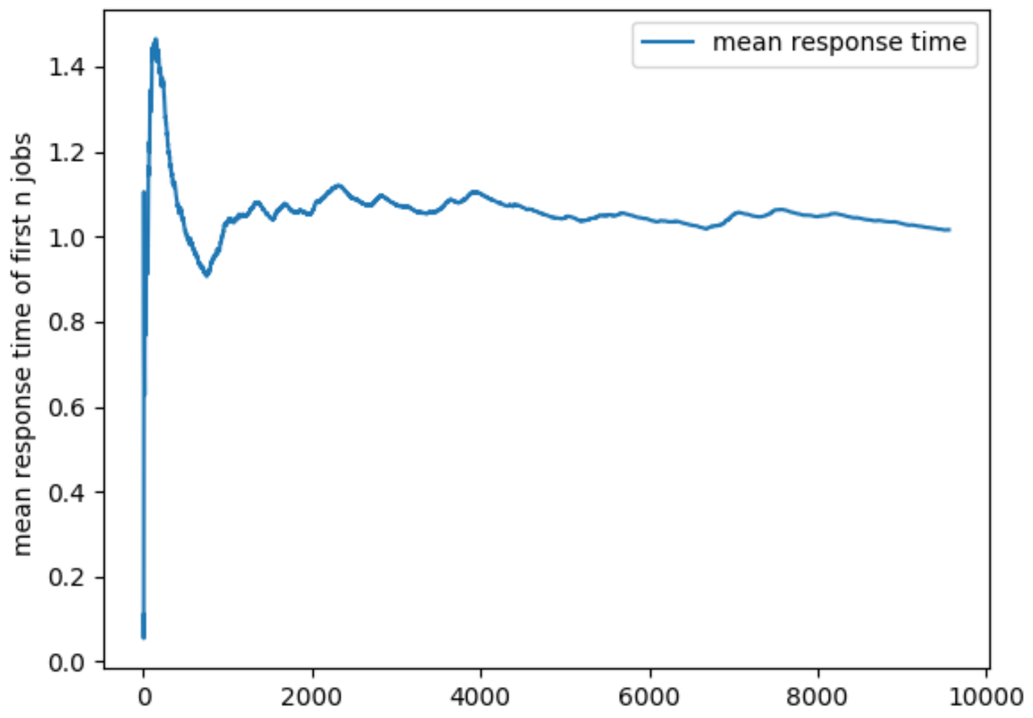
For starters , I set the fogLimitTime = 2, and I only variate the seed to make the request of
Job different. Then I run the function and draw the picture to find in which time the system will be steady.
I picked seed = 5, time end = 1000 , and by this seed, I get the mean response time is 1.0225



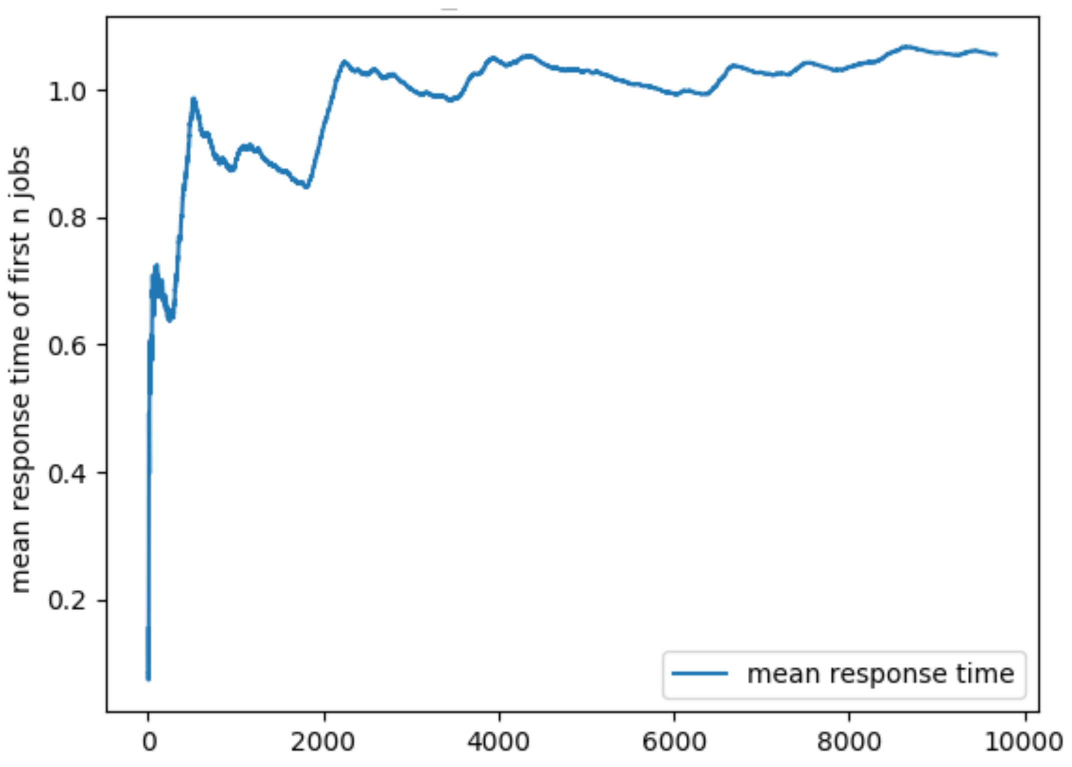
Then I picked seed = 10, time end = 1000 , and by this seed, I get the mean response time is
that the mean response time is 1.0666



After that, I picked seed = 20, time end = 1000 , and by this seed, I get the mean response time is 1.0162



And then I picked seed = 50, time end = 1000 , and by this seed, I get the mean response time is 1.0551

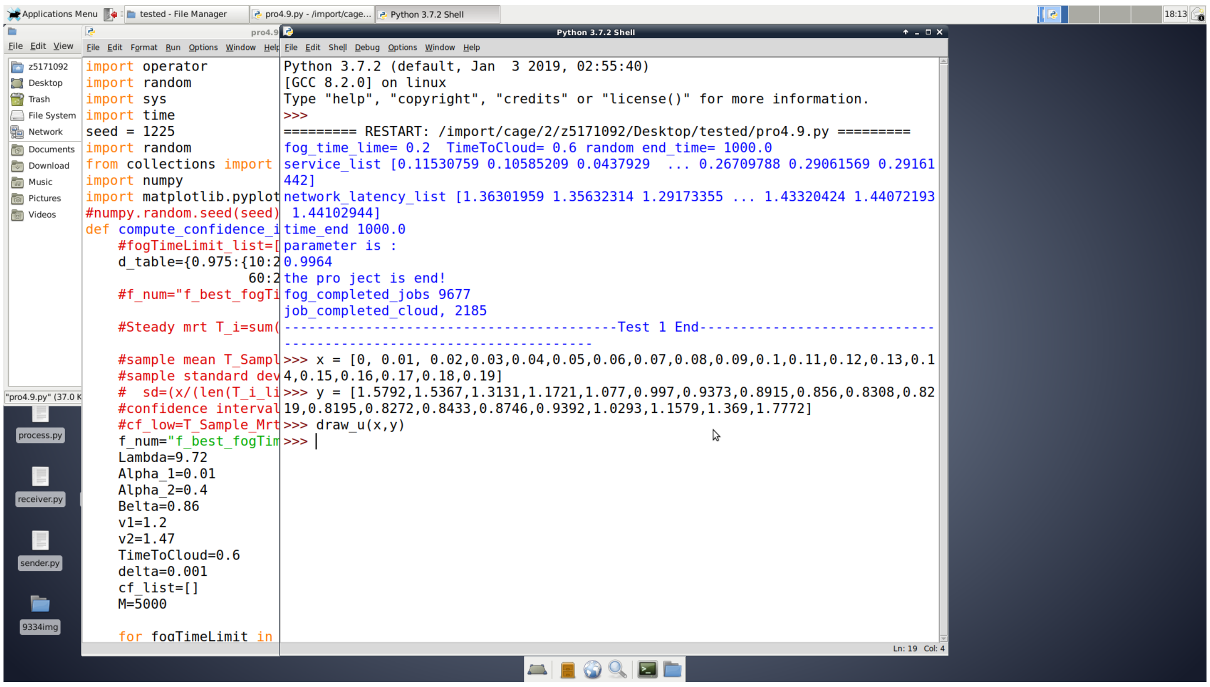


X-axis id number of jobs and Y-axis is mean response time of first n jobs

Besides, I still run many times of the plot function to draw the picture and observe to figure out the selection. At last, in accordance with my observation, I can assert that the conclusion of the images illustrate the after the about the 5000th , the system will be stable
Based on that, I pick the after 5000 request of jobs as the observation time.

Once the observation time , I begin to find out the possible range that the optimal fogLimitTime should be.

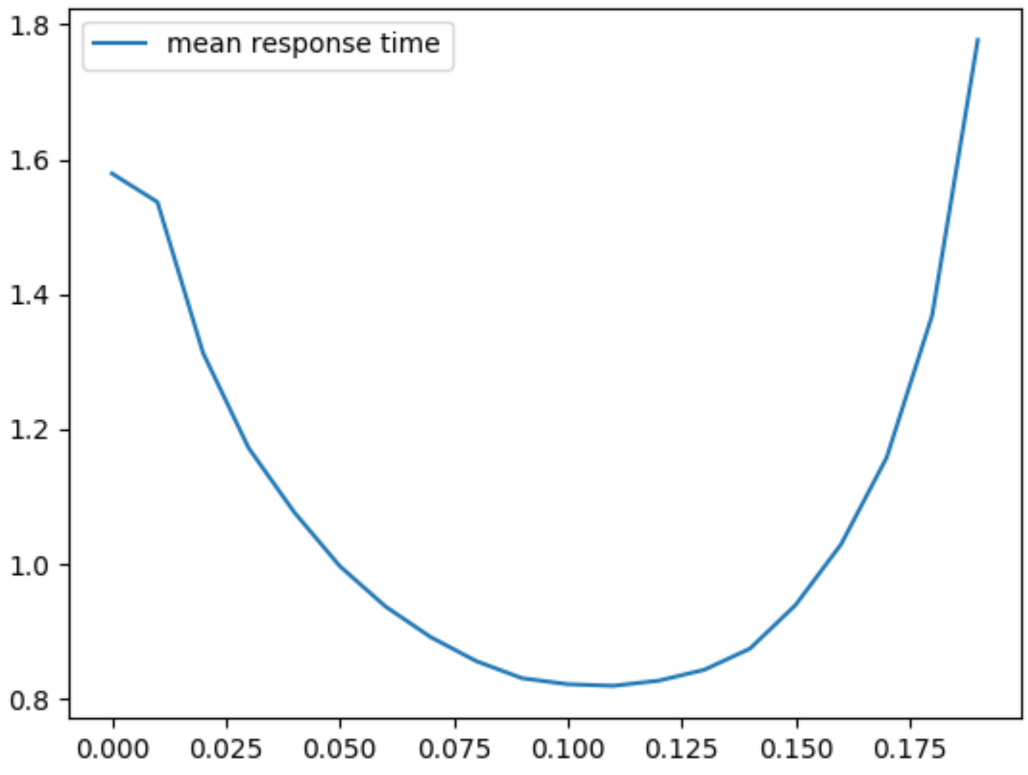
At this time, I set the time gap between 0.01, and draw the picture to see the fluctuation of the mean response time .I use the plot in python, once I find a point, I mark it on the blank, and I do this turn by turn ,until I can see that the mean response time decrease at former time and it crease at later time.



```
import operator
import random
import sys
import time
seed = 1225
import random
from collections import
import numpy
import matplotlib.pyplot
#numpy.random.seed(seed)
def compute_confidence_i
    #fogTimeLimit_list=[
    d_table={0.975:{10:2
                    60:2
    #f_num="f_best_fogTi

    #Steady mrt T_i=sum(

    #sample mean T_Sampl
    #sample standard dev
    #  sd=(x/(len(T_i_li
    #confidence interval
    #cf_low=T_Sample_Mrt
    f_num="f_best_fogTim
    Lambda=9.72
    Alpha_1=0.01
    Alpha_2=0.4
    Belta=0.86
    v1=1.2
    v2=1.47
    TimeToCloud=0.6
    delta=0.001
    cf_list=[]
    M=5000

    for fogTimeLimit in
```

```
Python 3.7.2 (default, Jan  3 2019, 02:55:40)
[GCC 8.2.0] on linux
Type "help", "copyright", "credits" or "license()" for more information.
>>>
========= RESTART: /import/cage/2/z5171092/Desktop/tested/pro4.9.py =========
fog_time_lime= 0.2  TimeToCloud= 0.6 random end_time= 1000.0
service_list [0.11530759 0.10585209 0.0437929  ... 0.26709788 0.29061569 0.29161
442]
network_latency_list [1.36301959 1.35632314 1.29173355 ... 1.43320424 1.44072193
 1.44102944]
time_end 1000.0
parameter is :
0.9964
the pro ject is end!
fog_completed_jobs 9677
job_completed_cloud, 2185
----------------------------------------Test 1 End-----------------------------
-------------------------------------
>>> x = [0, 0.01, 0.02,0.03,0.04,0.05,0.06,0.07,0.08,0.09,0.1,0.11,0.12,0.13,0.1
4,0.15,0.16,0.17,0.18,0.19]
>>> y = [1.5792,1.5367,1.3131,1.1721,1.077,0.997,0.9373,0.8915,0.856,0.8308,0.82
19,0.8195,0.8272,0.8433,0.8746,0.9392,1.0293,1.1579,1.369,1.7772]
>>> draw_u(x,y)
>>> |
```

Once I get these points, I connect them into a continuous line. The outcome Is the following one:



X-axis is fogLimitTime and Y-axis is mean response time of first n jobs

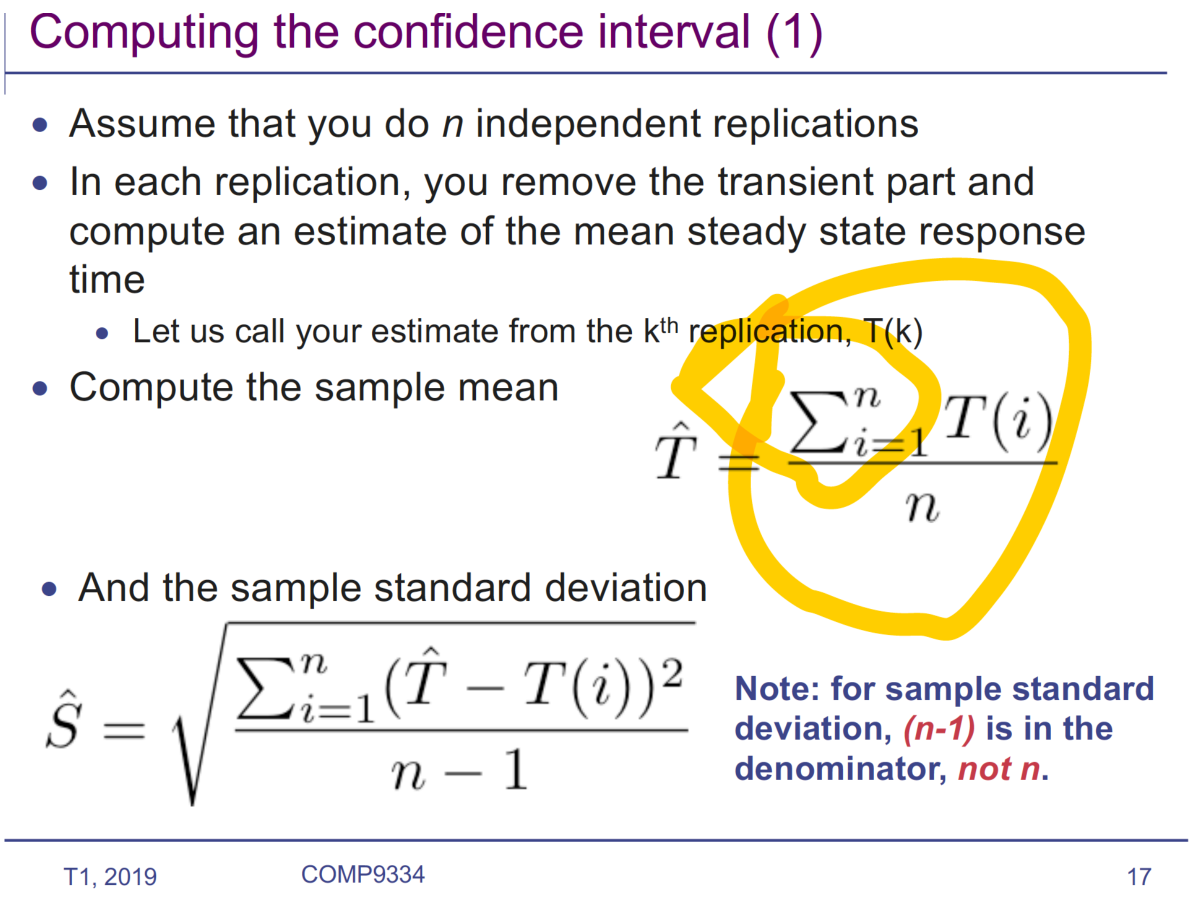
According to the above picture, I can guarantee the optimal fogLimitTime is

Between 0.09 to 0.12, after 0.12, the mean response time of first n time will rise again.
After that , I can compute the confidence interval

To make sure the accuracy, I find the confidence by using different seeds and each time I change the seed and establish the calculating code.

7.confidence interval

The most important thing in this report is to determine the optimal fogLimitTime via finding the confidence interval.
In accordance with the lecture in week 6A, we can easily master the progress to get the confidence interval.



Computing the confidence interval (1)

- Assume that you do *n* independent replications
- In each replication, you remove the transient part and compute an estimate of the mean steady state response time
  - Let us call your estimate from the kth replication, T(k)
- Compute the sample mean

$$\hat{T} = \frac{\sum_{i=1}^{n} T(i)}{n}$$

- And the sample standard deviation

$$\hat{S} = \sqrt{\frac{\sum_{i=1}^{n} (\hat{T} - T(i))^2}{n-1}}$$

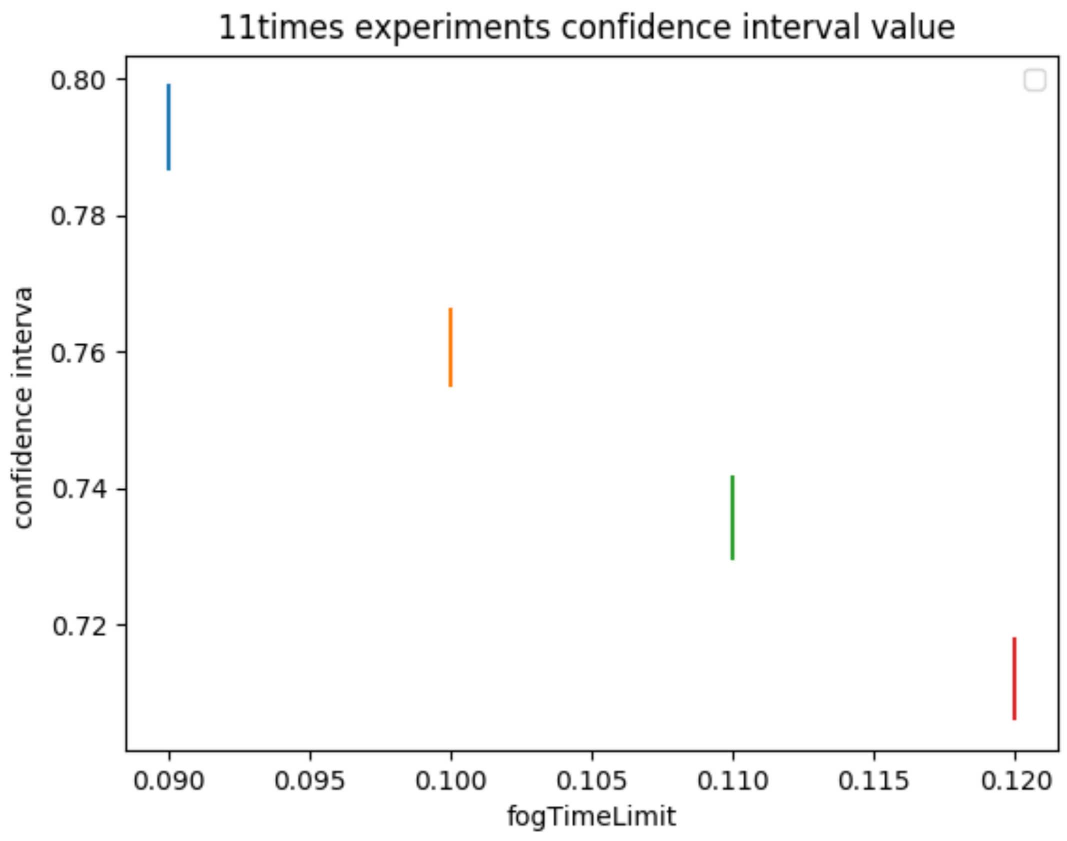
Note: for sample standard deviation, *(n-1)* is in the denominator, *not n*.

T1, 2019 COMP9334 17

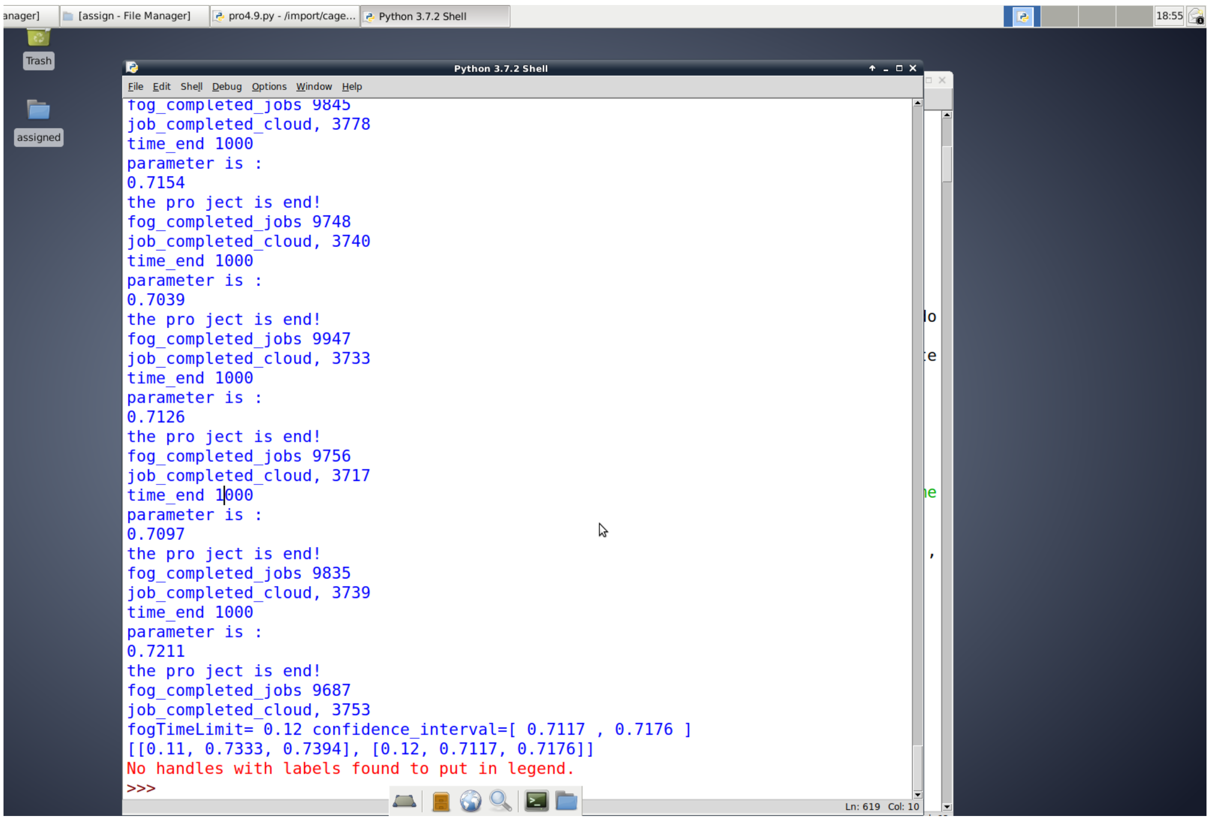
Then I use different seed to separate the confidence interval between 0.09 to 0.12.

Then I start to split them. I firstly set the gap as 0.01, the I test in the range of 0.09, 0.10,0.11,0.12 ,which one is the optimal one .

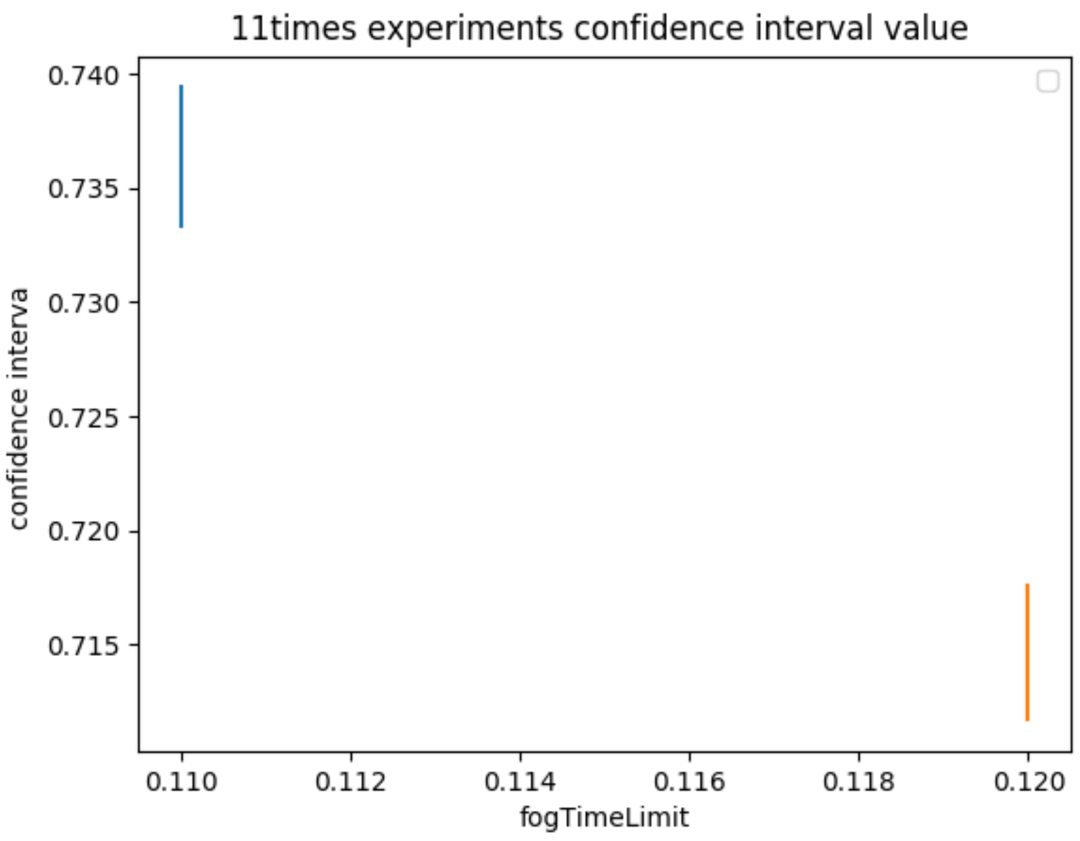
The first time I omit the 0.09 and 0.10. But there is an overlapping part in 0.11 and 0.12



Then I try to compare the 0.11 and 0.12



```
fog_completed_jobs 9845
job_completed_cloud, 3778
time_end 1000
parameter is :
0.7154
the pro ject is end!
fog_completed_jobs 9748
job_completed_cloud, 3740
time_end 1000
parameter is :
0.7039
the pro ject is end!
fog_completed_jobs 9947
job_completed_cloud, 3733
time_end 1000
parameter is :
0.7126
the pro ject is end!
fog_completed_jobs 9756
job_completed_cloud, 3717
time_end 1000
parameter is :
0.7097
the pro ject is end!
fog_completed_jobs 9835
job_completed_cloud, 3739
time_end 1000
parameter is :
0.7211
the pro ject is end!
fog_completed_jobs 9687
job_completed_cloud, 3753
fogTimeLimit= 0.12 confidence_interval=[ 0.7117 , 0.7176 ]
[[0.11, 0.7333, 0.7394], [0.12, 0.7117, 0.7176]]
No handles with labels found to put in legend.
>>>
```



Finally, I get the parameter that the interval of 0.11 is [0.7333, 0.7394] and the interval of 0.12 is [0.7117, 0.7176]. but I deploy the gap to 0.1, this might have the small difference, so I can only assert that the optimal fogLimitTime is not a specific number, it should be in the range of (0.11, 0.12]

In conclusion, I have 95% confidence to assert that the optimal range is (0.11, 0.12]

Plot script:
I use the plot script in python, here is the code.
Each time I draw anything , just input the parameter of x-axis and y-axis, then then the expected outcome will be shown

```
def draw_u(aix_x,aix_y):
    fig = plt.figure()
    ax = fig.add_subplot(parameter)
    title_s = "seed=" + str(seed) + "  Time_End="
+ str(time_end) + " fogTimeLimit=" +
str(fogTimeLimit)
    ax.legend()
    plt.show()
    cf_list=[]
```