COMP9334 - Project

Hao Chen 5102446

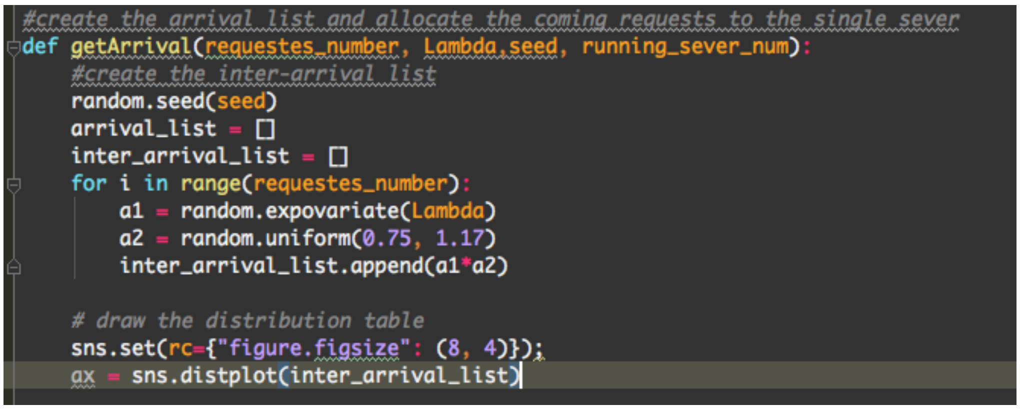
Part 1 - Probability Distributions

1. The Probability Distribution of arrival time

To begin with, we need to generate the arrival time correctly, then we can have the right number to go on the following steps.

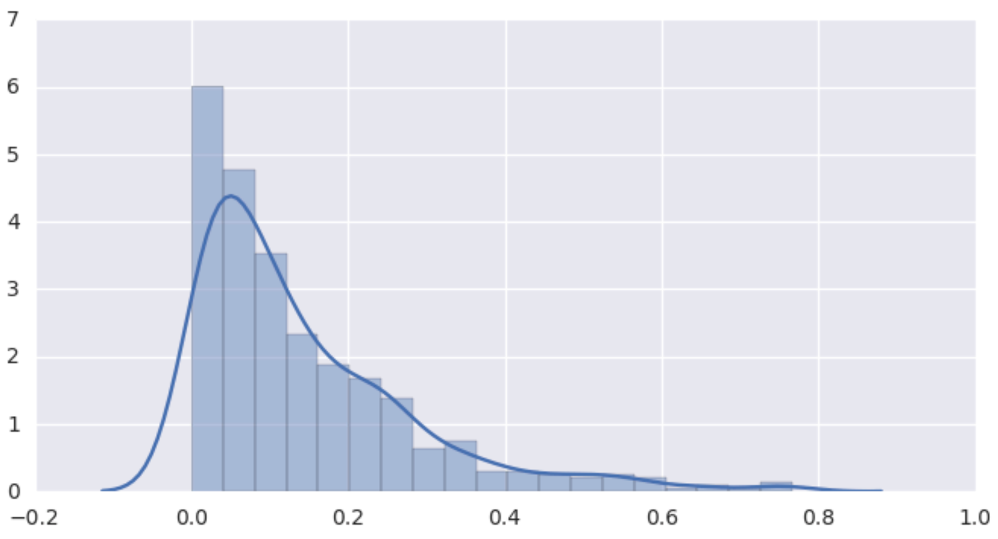
Based on the document, the inter-arrival time of the requests arriving is ak = a1k * a2k, while the a1k stands for the exponentially distributed with a mean arrival rate 7.2 req/s. In the same time, a2k is uniformly distributed in the interval [0.75, 1.17].

In the program, I write the program like below:



```
#create the arrival list and allocate the coming requests to the single sever
def getArrival(requestes_number, Lambda,seed, running_sever_num):
    #create the inter-arrival list
    random.seed(seed)
    arrival_list = []
    inter_arrival_list = []
    for i in range(requestes_number):
        a1 = random.expovariate(Lambda)
        a2 = random.uniform(0.75, 1.17)
        inter_arrival_list.append(a1*a2)

    # draw the distribution table
    sns.set(rc={"figure.figsize": (8, 4)});
    ax = sns.distplot(inter_arrival_list)
```

Note that I sum this list up to create the arrival list and allocate requests in the program, and I just show the part of it to indicate how I generate the inter_arrival_list.



2. The Probability Distribution of the service time

For the probability distribution of the service time can be deduced by the inverse transform method.

The given CDF of probability density function g(t) is:

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

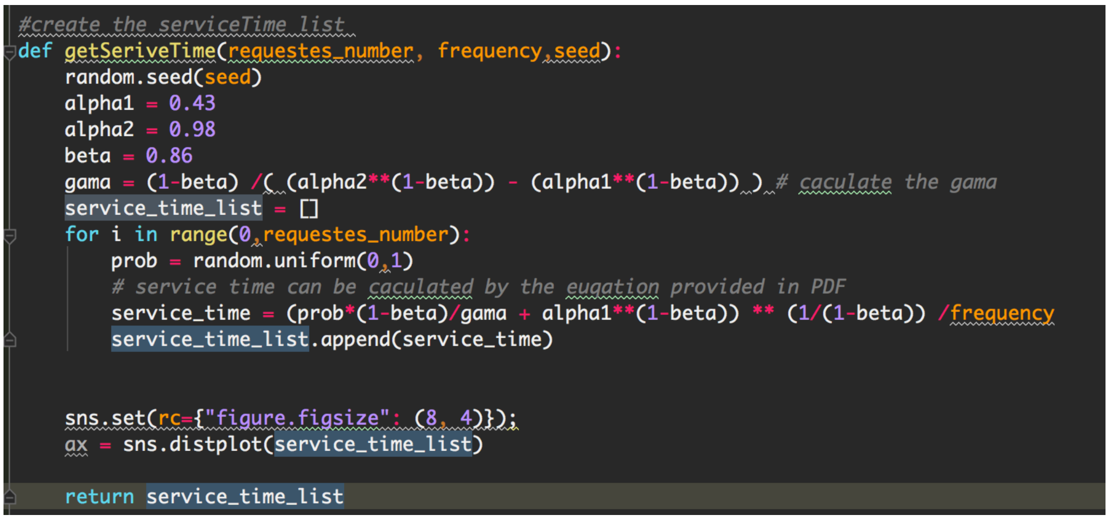
So:

$$G(t) = \int_{\alpha_1}^{t} g(t)\, dt = \frac{\gamma}{1-\beta} t^{1-\beta} - \frac{\gamma}{1-\beta} \alpha_1^{1-\beta}$$

The inverse function of G(t) can be obtained:

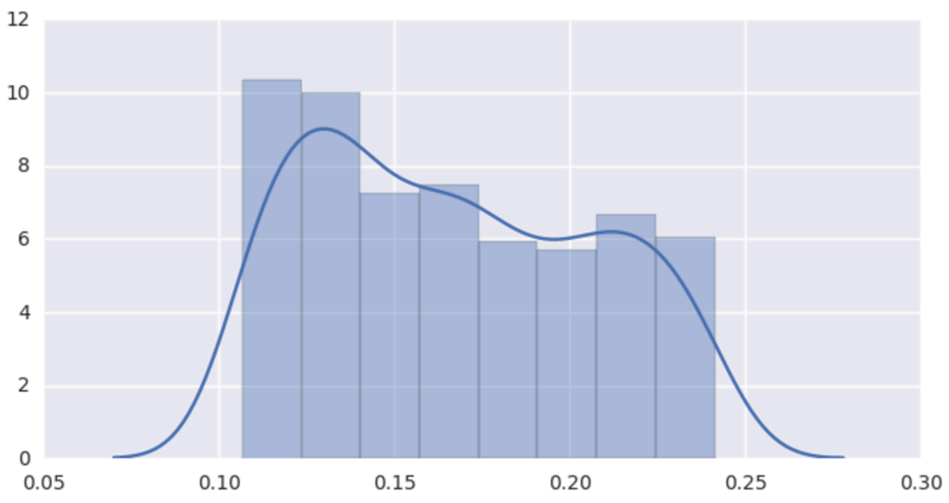
$$y^{-1}(t) = \left[\frac{1-\beta}{\gamma}\left(t + \frac{\gamma}{1-\beta}\alpha_1^{1-\beta}\right)\right]^{\frac{1}{1-\beta}}$$

Based on the equation obtained from above, I write the program like below:



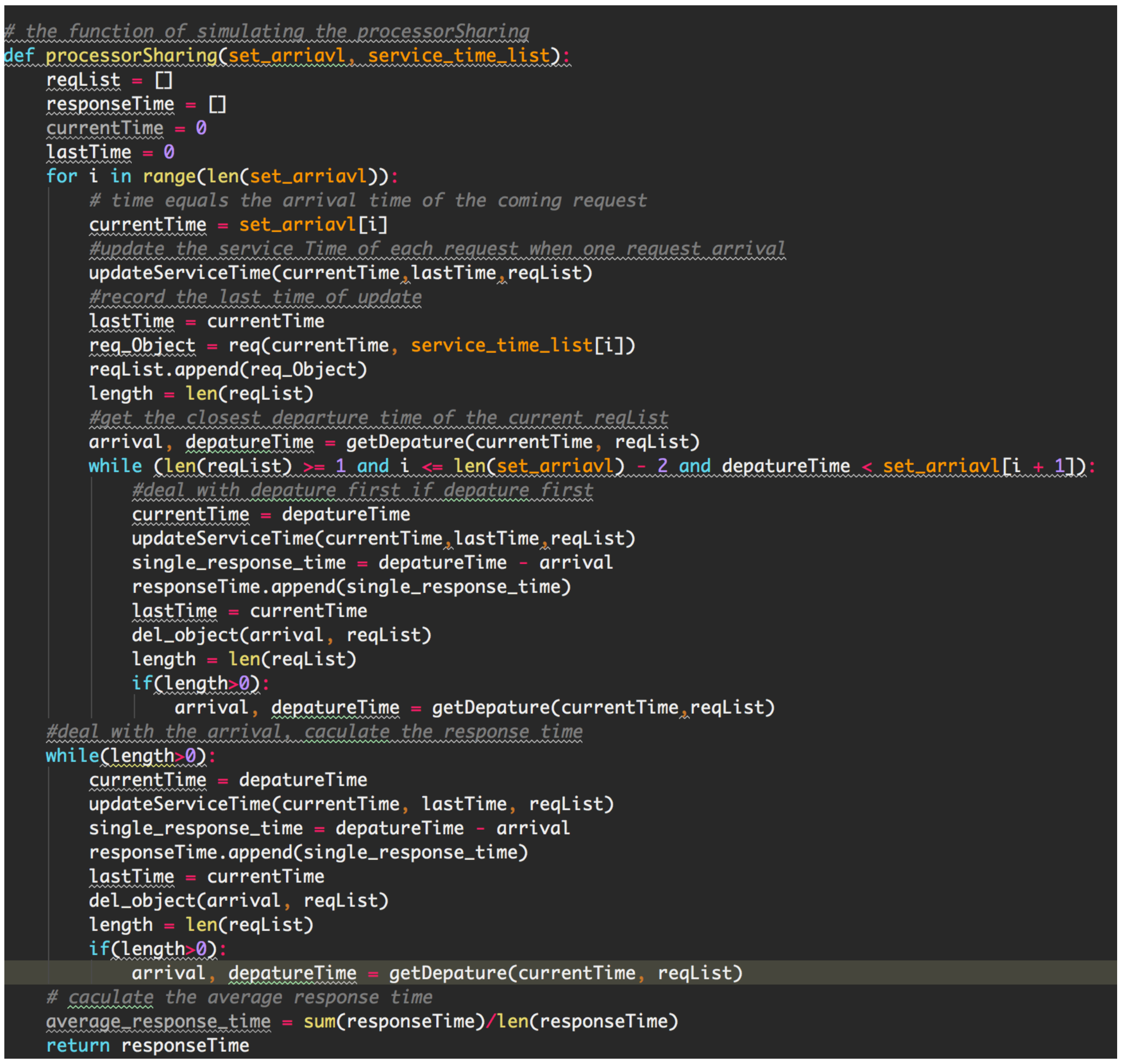
```
#create the serviceTime list
def getSeriveTime(requestes_number, frequency,seed):
    random.seed(seed)
    alpha1 = 0.43
    alpha2 = 0.98
    beta = 0.86
    gama = (1-beta) /( (alpha2**(1-beta)) - (alpha1**(1-beta)) ) # caculate the gama
    service_time_list = []
    for i in range(0,requestes_number):
        prob = random.uniform(0,1)
        # service time can be caculated by the euqation provided in PDF
        service_time = (prob*(1-beta)/gama + alpha1**(1-beta)) ** (1/(1-beta)) /frequency
        service_time_list.append(service_time)


    sns.set(rc={"figure.figsize": (8, 4)});
    ax = sns.distplot(service_time_list)

    return service_time_list
```

The figure below can illustrate the service time generated by the program:



Part 2 - Verification of the simulation program

1. Description about the program



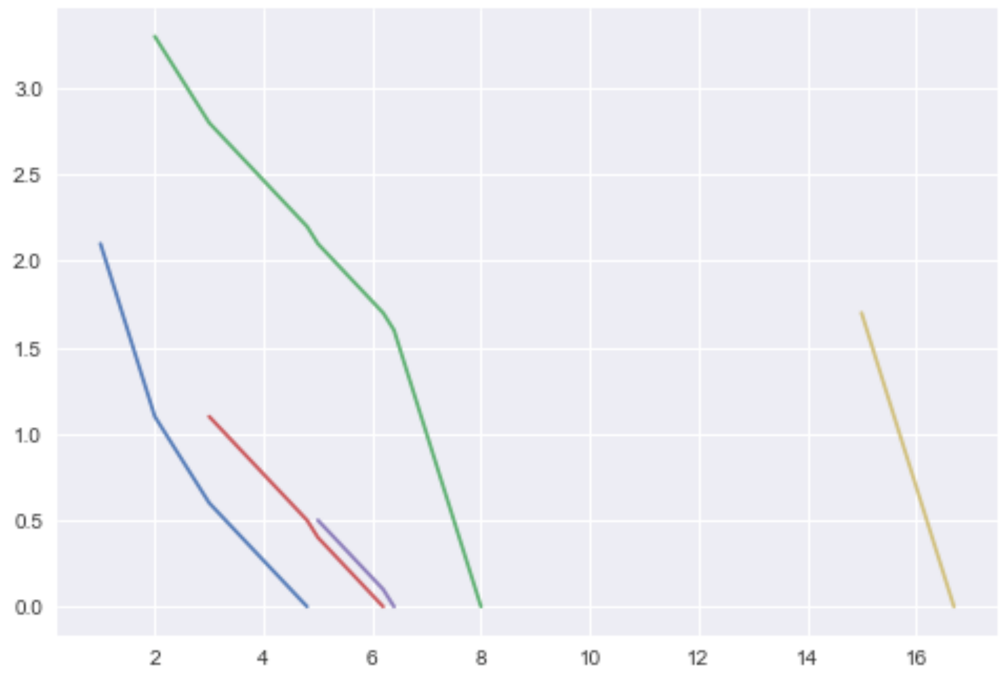
```
# the function of simulating the processorSharing
def processorSharing(set_arriavl, service_time_list):
    reqList = []
    responseTime = []
    currentTime = 0
    lastTime = 0
    for i in range(len(set_arriavl)):
        # time equals the arrival time of the coming request
        currentTime = set_arriavl[i]
        #update the service Time of each request when one request arrival
        updateServiceTime(currentTime,lastTime,reqList)
        #record the last time of update
        lastTime = currentTime
        req_Object = req(currentTime, service_time_list[i])
        reqList.append(req_Object)
        length = len(reqList)
        #get the closest departure time of the current reqList
        arrival, depatureTime = getDepature(currentTime, reqList)
        while (len(reqList) >= 1 and i <= len(set_arriavl) - 2 and depatureTime < set_arriavl[i + 1]):
            #deal with depature first if depature first
            currentTime = depatureTime
            updateServiceTime(currentTime,lastTime,reqList)
            single_response_time = depatureTime - arrival
            responseTime.append(single_response_time)
            lastTime = currentTime
            del_object(arrival, reqList)
            length = len(reqList)
            if(length>0):
                arrival, depatureTime = getDepature(currentTime,reqList)
    #deal with the arrival, caculate the response time
    while(length>0):
        currentTime = depatureTime
        updateServiceTime(currentTime, lastTime, reqList)
        single_response_time = depatureTime - arrival
        responseTime.append(single_response_time)
        lastTime = currentTime
        del_object(arrival, reqList)
        length = len(reqList)
        if(length>0):
            arrival, depatureTime = getDepature(currentTime, reqList)
    # caculate the average response time
    average_response_time = sum(responseTime)/len(responseTime)
    return responseTime
```

The whole program is available in the appendix, and I will show the main part of it briefly here (PS function).
The concept is to set the current time as the arrival list, and then compare the next arrival time and the next departure time. If arrival fist, keep adding, if departure fist, calculate the response time and del the departure one. When no more arrival, calculate the left reqs.

2. Run the simulation case proved by the PDF
Data can be illustrated from the PDF like below :

Arrival Time	Service Time	Depature Time	Index
1	2.1	4.8	1
2	3.3	8	2
3	1.1	6.2	3
5	0.5	6.4	4
15	1.7	16.7	5

The graph can be obtained based on the simulation of my program, and this can be a very simple way to verify whether the program is running well. (The code for drawing this graph is shown in the program.)

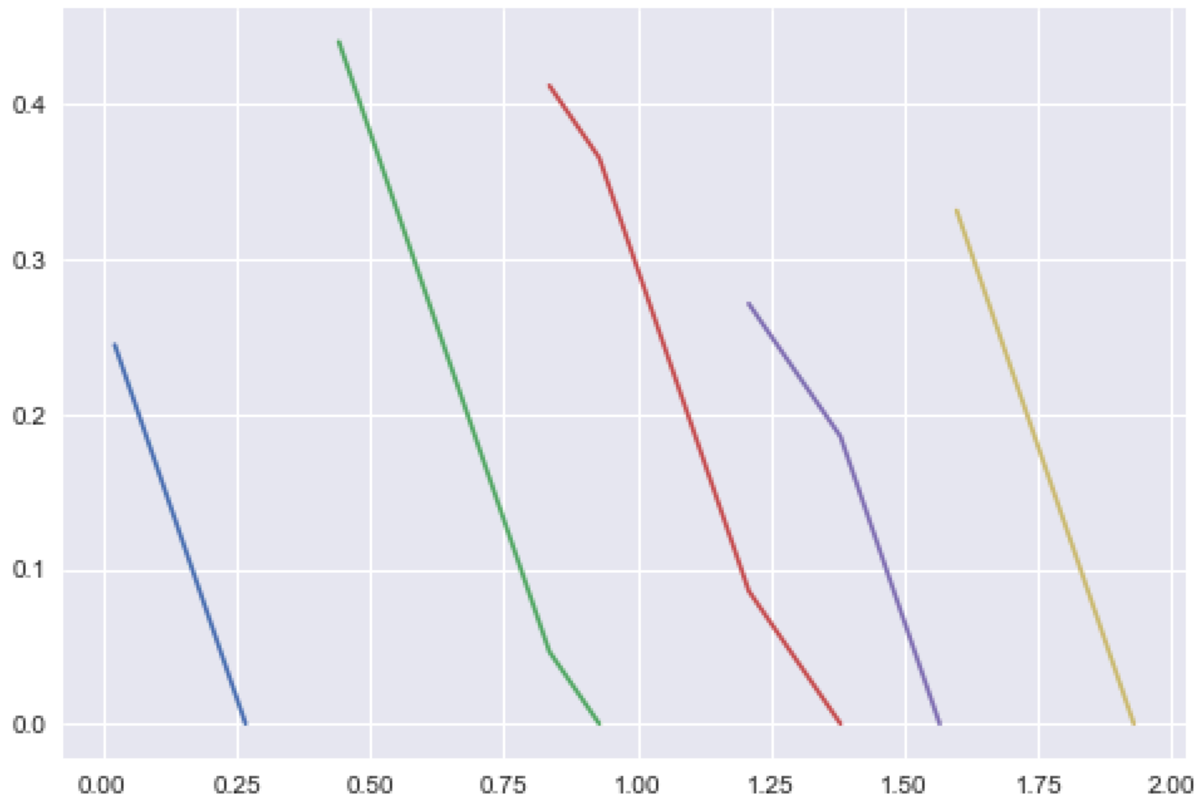


3. Run the simulation case generated by my program

To test my program further, I made another simulation, and the data is shown below, which is generated by my program. Using seed equals to 1, we can simulate this process repetitively.

Arrival Time	Service Time	Depature Time	Index
0.022163150925197196	0.244796675922363	0.2669598268475602	1
0.44147739881186837	0.44059597178127663	0.9286345613023667	2
0.8355121798839233	0.4122498302337077	1.3800520376277237	3
1.2085943640259817	0.2713400167493471	1.56566321757619986.4	4
1.5973777209566886	0.3316420665226282	1.9290197874793167	5

The result can be shown in the graph below.



Note that those slopes are changing according to process of Processor Sharing(PS). The PS process means that when N jobs in the sever, the speed of processing will only be 1/N of the total jobs.

Part 3 - Collection of the simulation process

We can start from the single sever first, and record the response time of each job. I pick the sample number of job equals to 200, in which case PS will allocate this single sever with all these jobs.

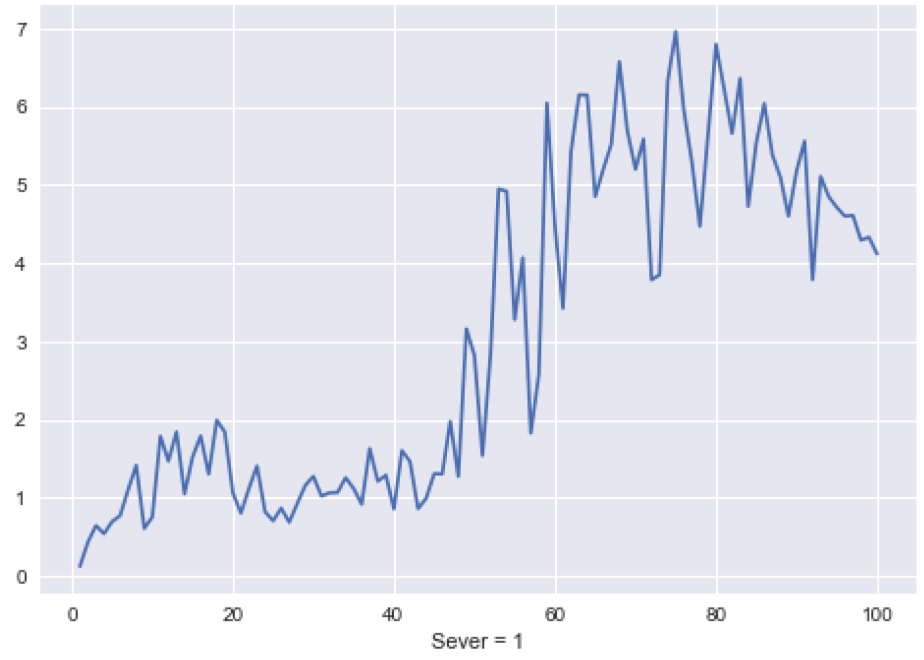


Fig.1 Single sever with 100 reqs

We can only see that it fluctuates all the time and the curve keeps going up, and it still has the trend to go up high in the end. It is hard to draw conclusion, so we increase the job number as below.

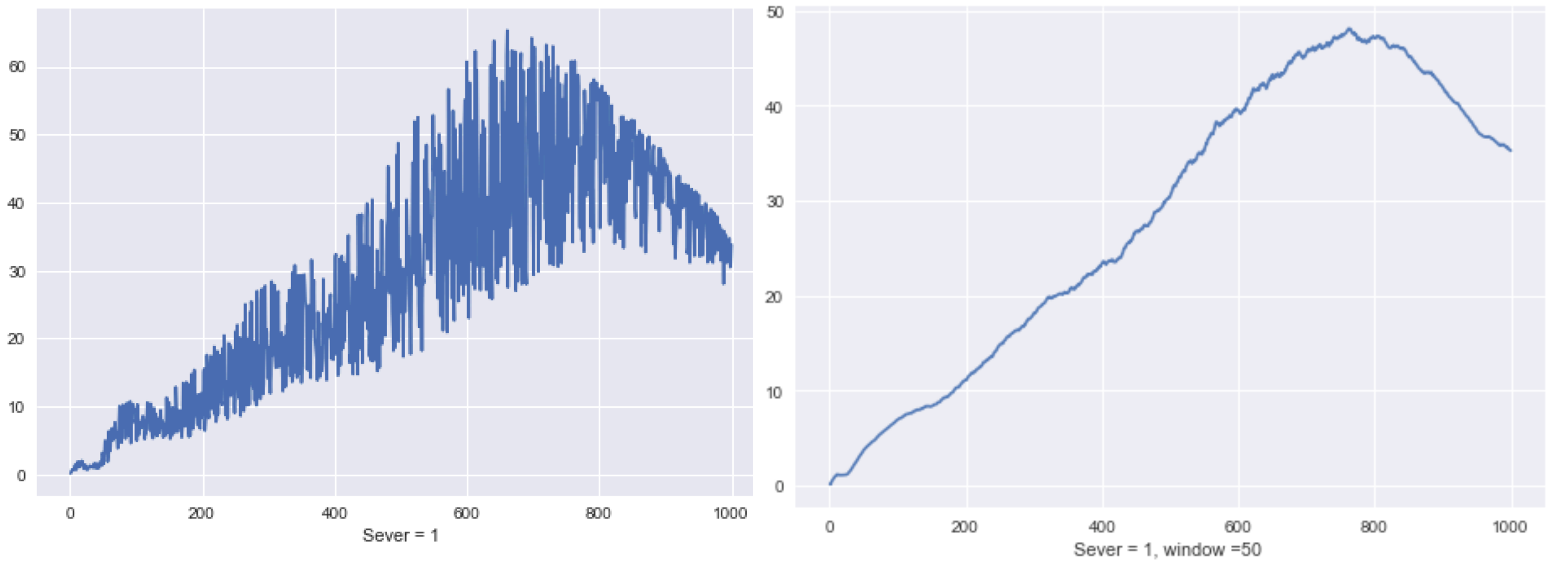
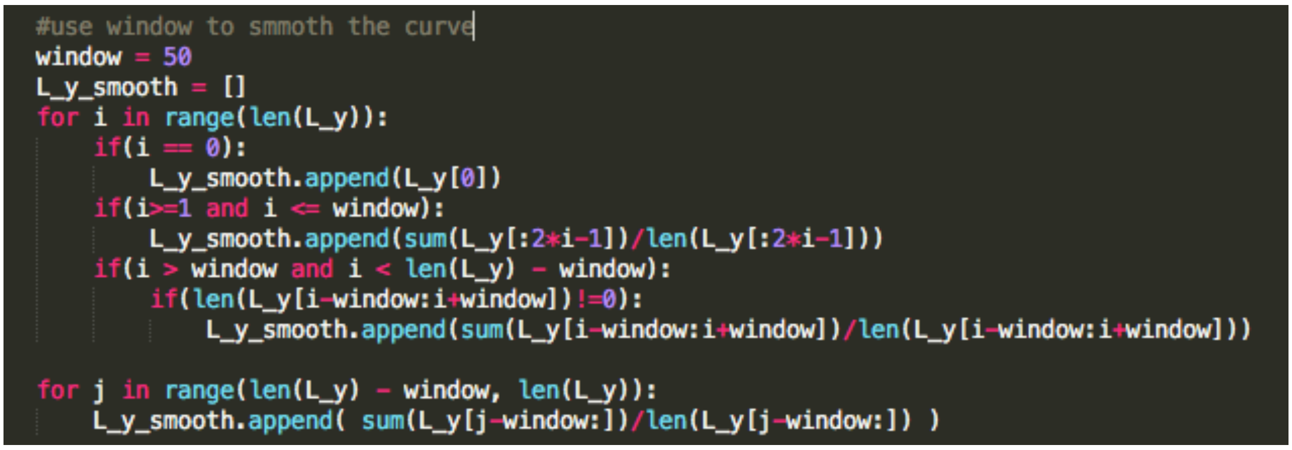


Fig.2 Single sever with 1000 reqs

For the right graph, I smooth it out with the window size equals to 50. The code I used is from teacher's week7's file, and I transfer it to the Python code, which is shown as below.



```
#use window to smmoth the curve
window = 50
L_y_smooth = []
for i in range(len(L_y)):
    if(i == 0):
        L_y_smooth.append(L_y[0])
    if(i>=1 and i <= window):
        L_y_smooth.append(sum(L_y[:2*i-1])/len(L_y[:2*i-1]))
    if(i > window and i < len(L_y) - window):
        if(len(L_y[i-window:i+window])!=0):
            L_y_smooth.append(sum(L_y[i-window:i+window])/len(L_y[i-window:i+window]))

for j in range(len(L_y) - window, len(L_y)):
    L_y_smooth.append( sum(L_y[j-window:])/len(L_y[j-window:]) )
```

From the graph, we can see that the curve is still growing rapidly, which indicates that one sever is not enough to deal with all coming jobs, so we keep adding severs and try to see any differences. The graph below is 1000 requests with 2 severs, and the right one is with smooth processed.

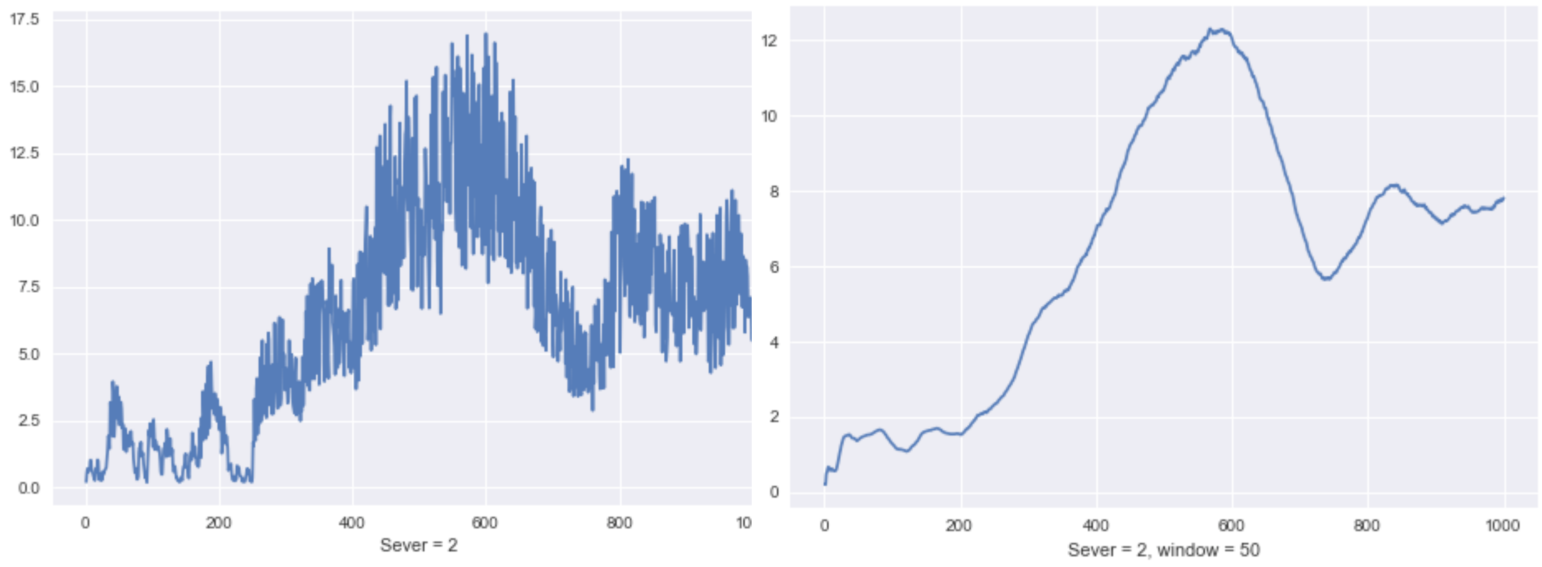
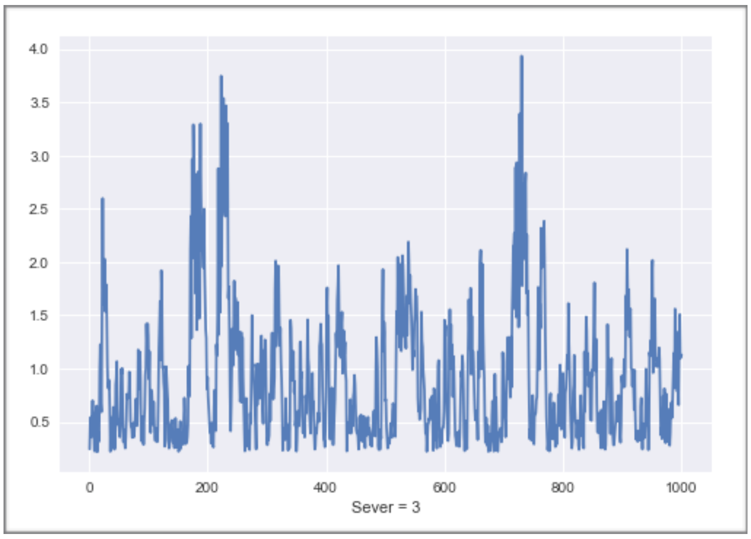


Fig.3 sever = 2 with 1000 reqs

From the graph above, we can see that the curve is still growing rapidly, and it dose not have a trend of settle down in the first 800 requests. However, the curve is reaching a basic stability of the response time when it reaching the end. So, We can conclude that the sever numbers must beyond this number, and can hold stable at some place. Then we keep raising sever number all the way to 6, and show their graphs below.



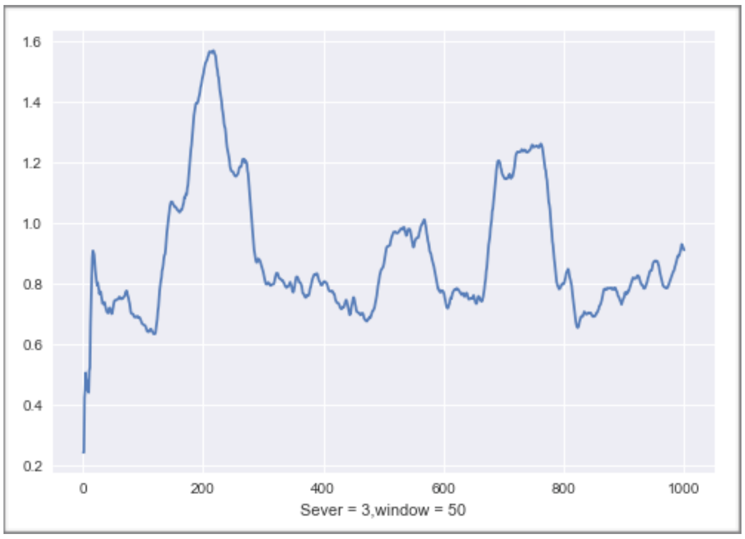
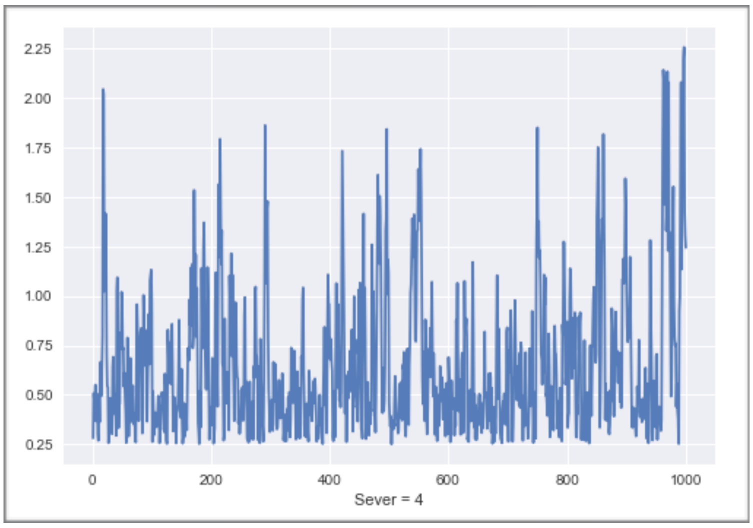


Fig.4 sever = 3 with 1000 reqs



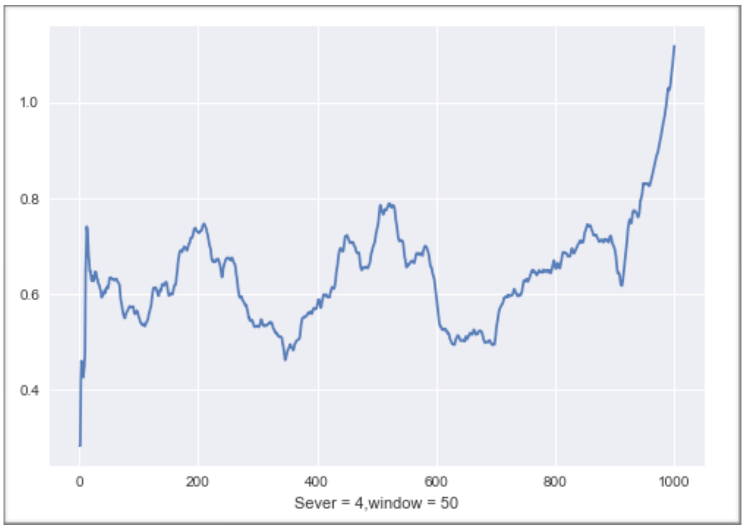
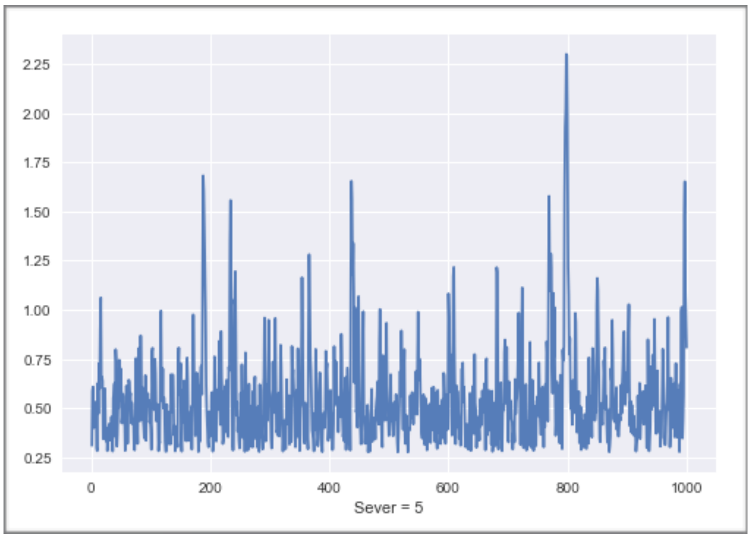


Fig.5 sever = 4 with 1000 reqs



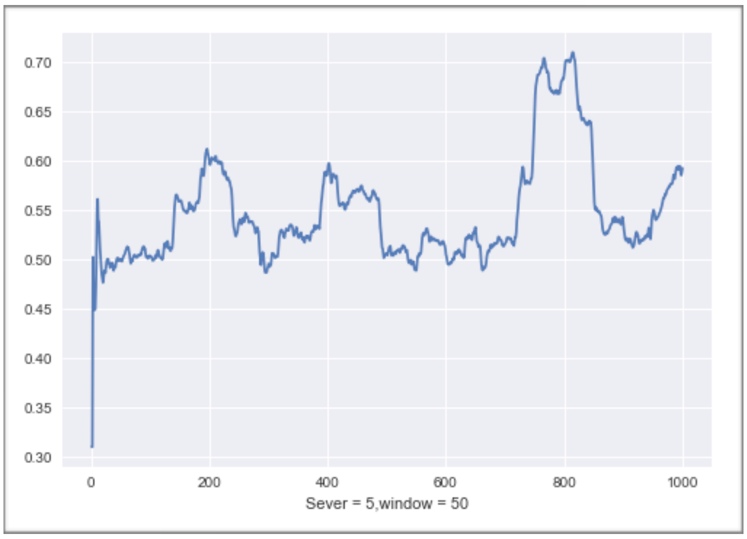


Fig.5 sever = 5 with 1000 reqs

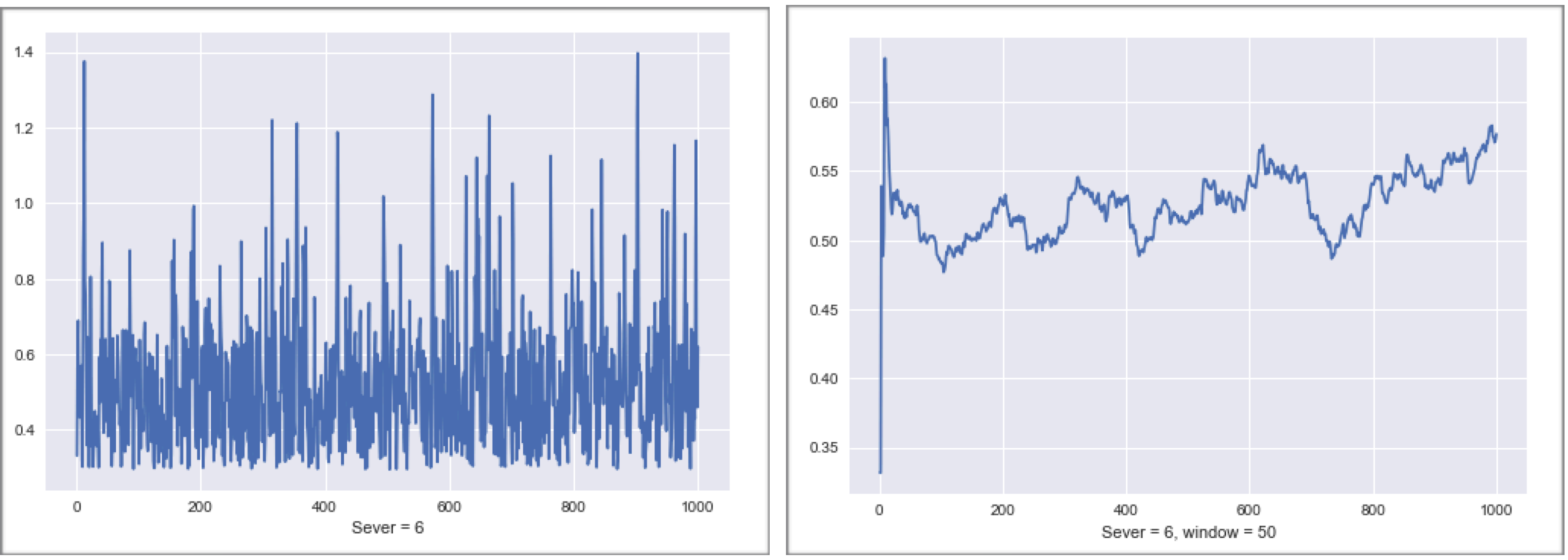


Fig.6 sever = 6 with 1000 reqs

From the figures when sever is 3, 4, 5 and 6, the response time fluctuates with the growing of the job numbers, and their response time also does not keep increasing. The fluctuation ends at around 150 jobs.

However, I notice that the window size chosen for smoothing can be a important parameter, so I will change it in following graph and try to get a better size when jobs increasing. The graph below shows when coming request equal to 5000, the influence from the window size.

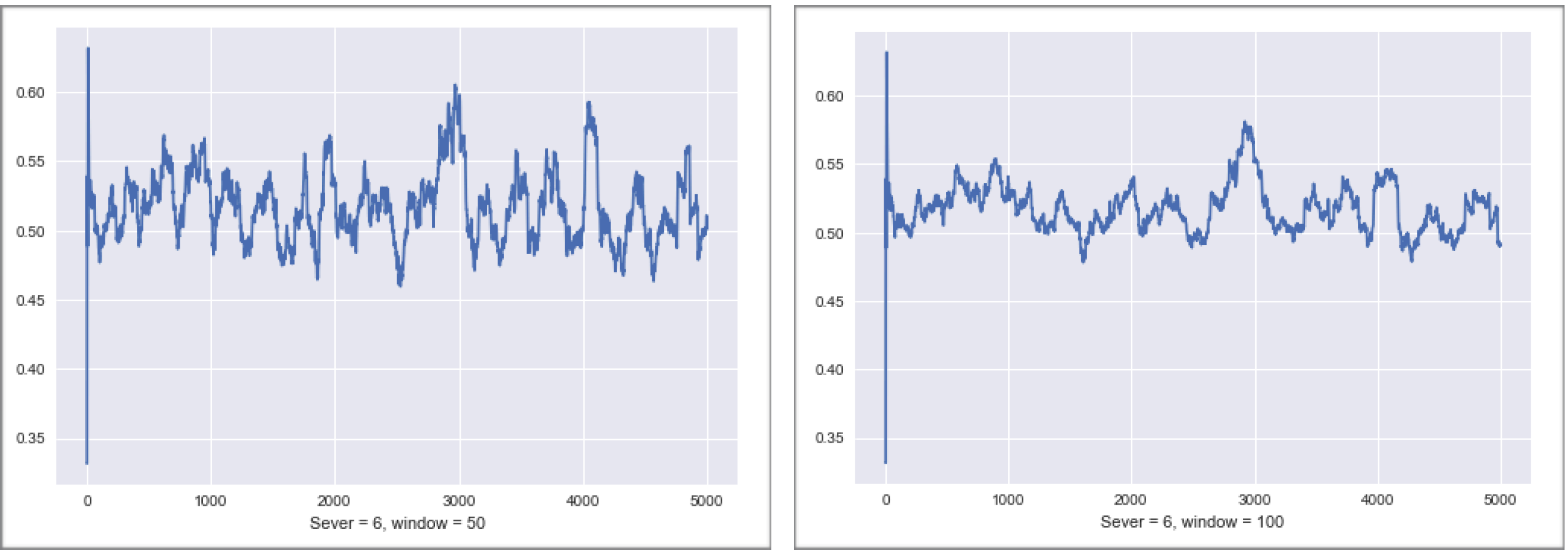


Fig.7 sever = 6, w = 50 & 100 # 5000 reqs

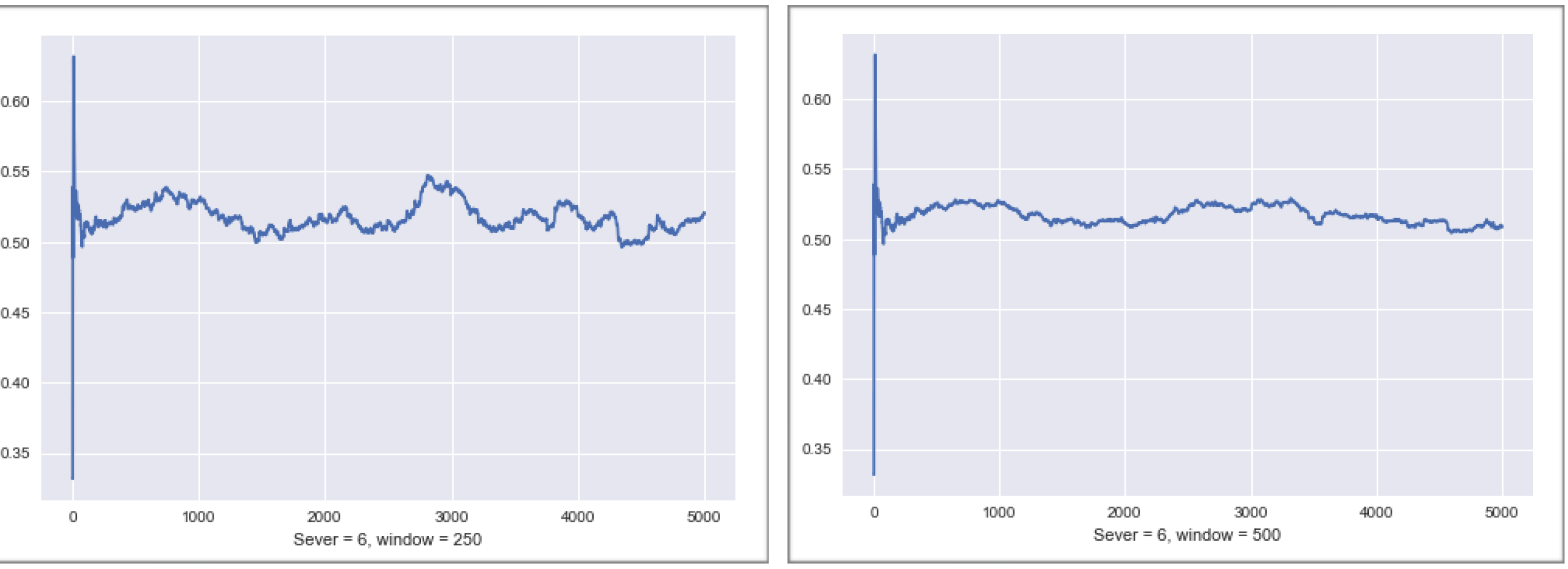


Fig.8 sever = 6, w = 250 & 500 # 5000 reqs

From the graphs shown above, it is clear that the curves are getting smoother with the growing of the value window. When the window value is 500, the curves becoming the most smooth, however since we have to distinguish the transient from the the steady period, we could not chose the smoothest one. Finally I set w = #jobs/10 as the value to do the smooth deal.

Next, I change the length of total requests from 500, 1000, 2000 to 5000, using the w = #jobs/10, and try to obtain the steady data for response time.

The figures are shown below.

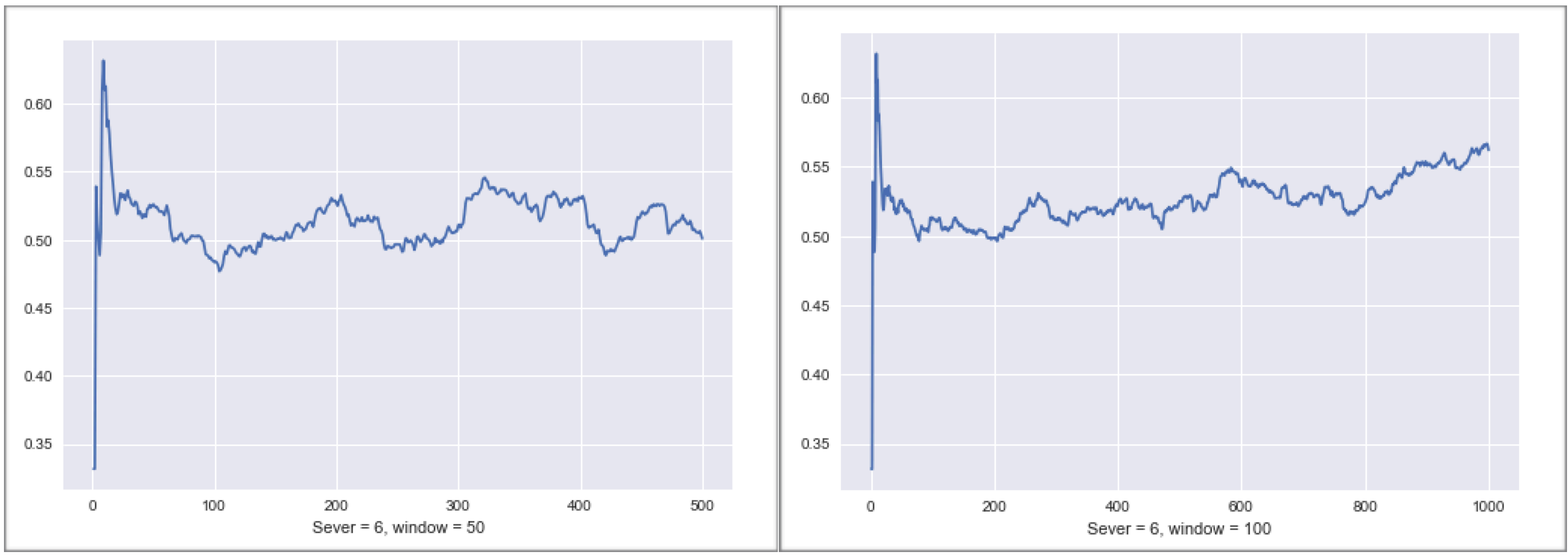


Fig.9 sever = 6, 500 & 1000 jobs

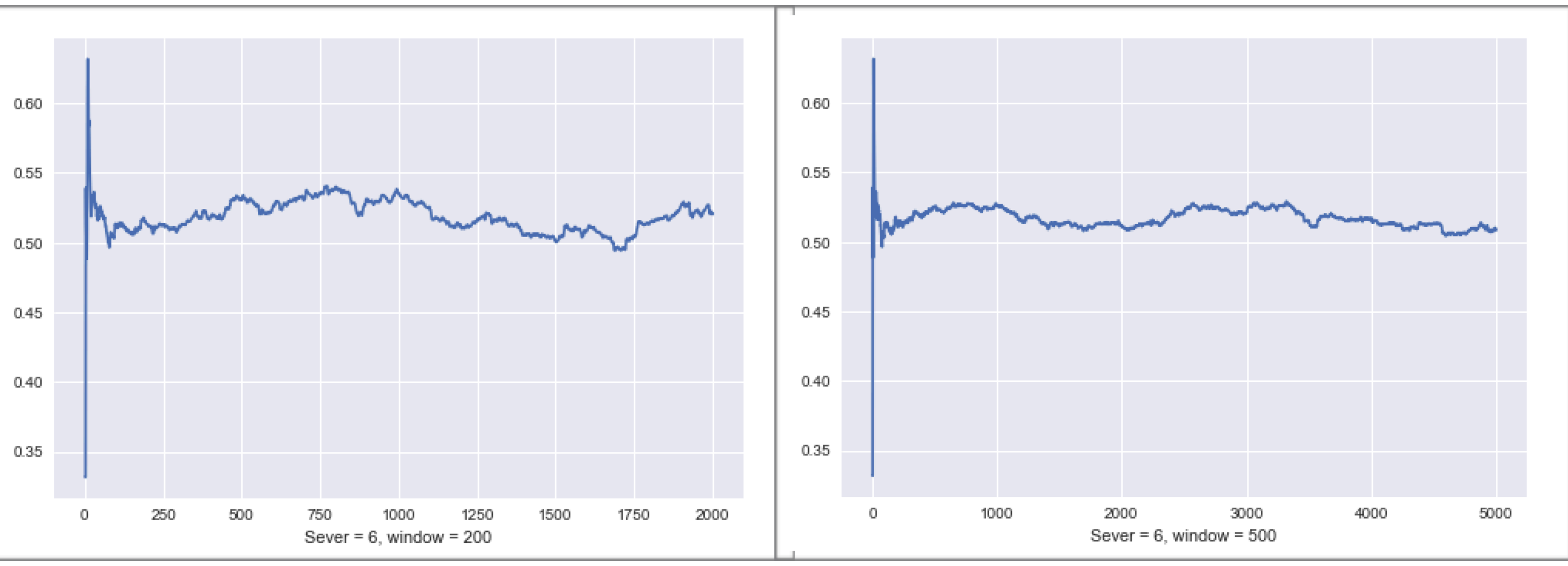


Fig.10 sever = 6, 2000 & 5000 jobs

From the results from these above graphs, we can find out that the transient ends at 150, so the jobs before the 150 can be ignored. To be more accuracy, we only consider the data after 500 jobs. To let the effective data make up the majority of our experiment , we just consider the jobs after 500 when the total job is 5000. In that way, 90% of the data are effective to us.

Apply the rule we set above to other left severs, which is 7, 8, 9,10. The figures will be shown below.

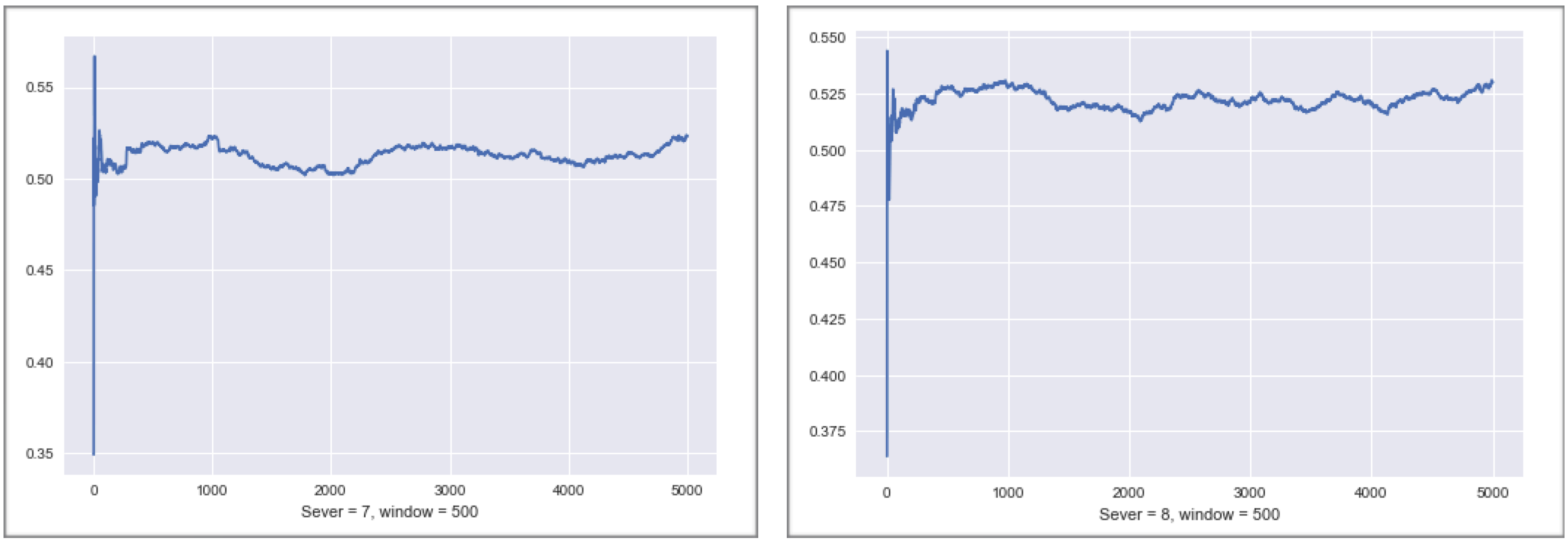
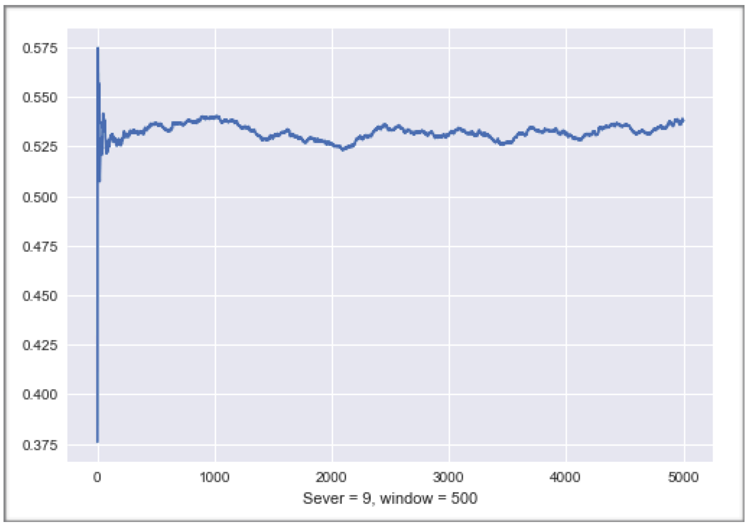


Fig.11 sever = 6 & 7, 5000 jobs



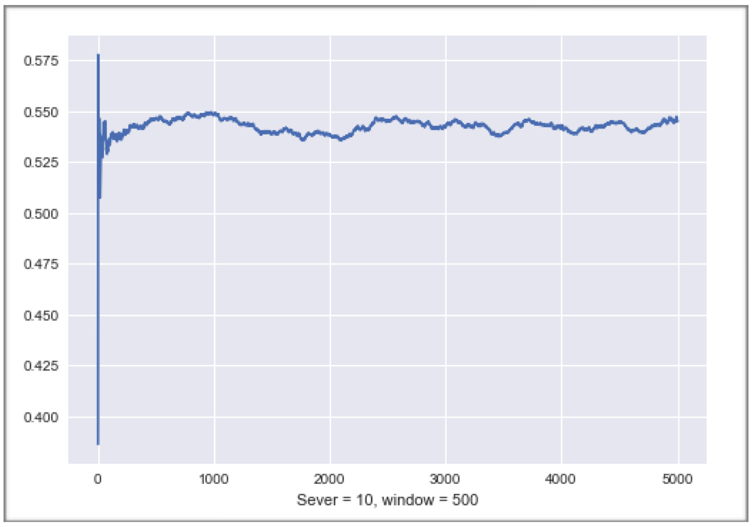


Fig.11 sever = 9 & 10, 5000 jobs

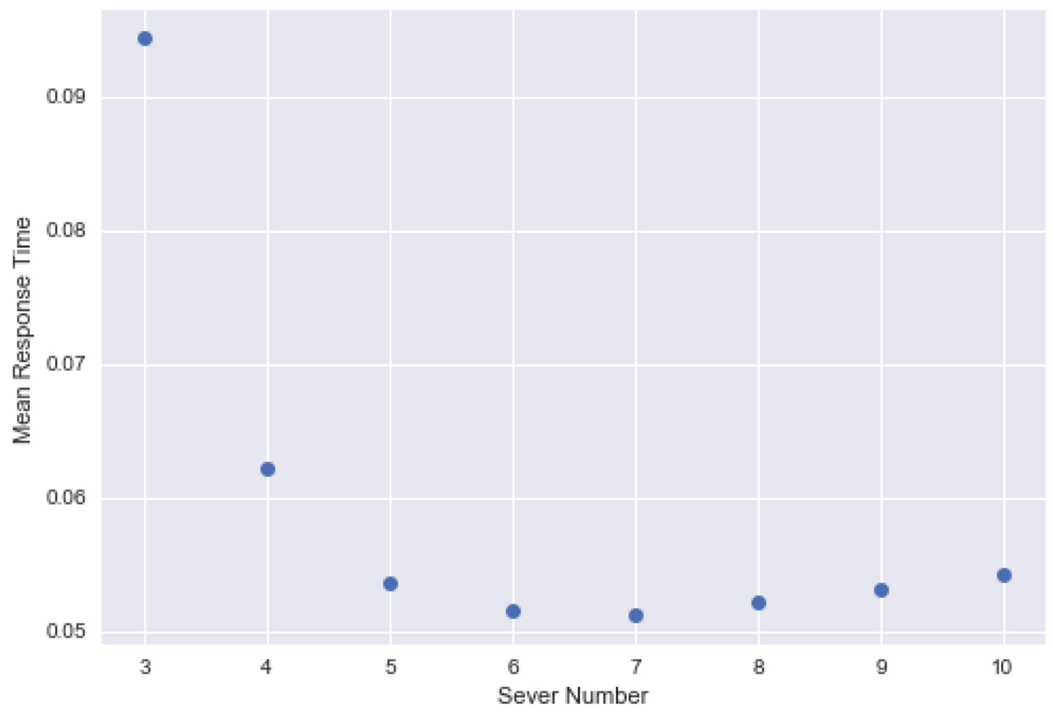
Until now, all figures of the 10 severs are drawn and the results can be compared in the following parts.

Part 4 - The analysis of the simulation

We run the program from sever 3 to 10, and calculate their mean response time separately (using the last 4000 job only). First I will show their table result and then the graph result.

Sever Number	Mean Response Time
3	0.9431998741692614
4	0.6214193964195144
5	0.5361260669998087
6	0.5151407559829395
7	0.5116321231010089
8	0.5213357484235728
9	0.531650211151551
10	0.5419250518302015

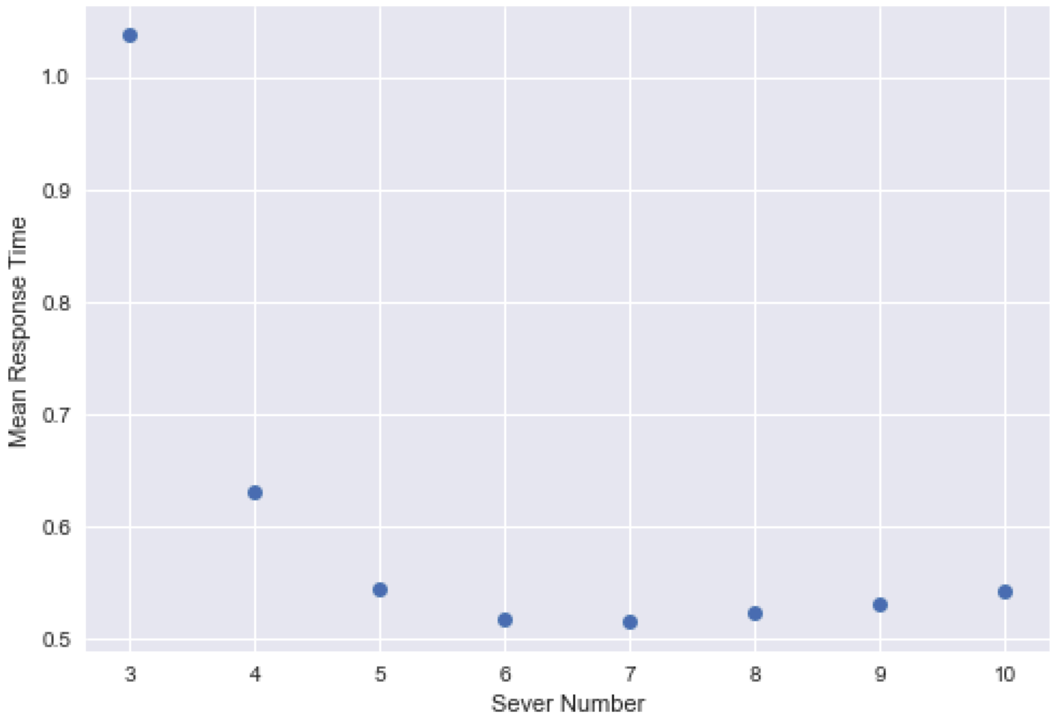
We can see that the mean response time decrease rapidly first, and then it has a slight increase later, which can be clearly seen from the graph below.



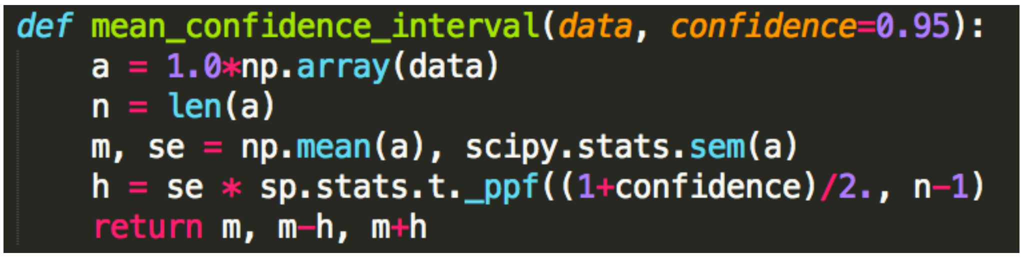
However, we are still unable to say which one is better, because based on the table above, sever 6, 7, 8 have really close mean response time. To test it further, I change the seed (run ten times) and record their mean response time and then find the mean response time of this ten test. Repeat for ten times and we can get the table below.

Sever Number	**Mean Response Time (10)**
3	1.0368352951807593
4	0.6297607263580354
5	0.544264074775374
6	0.5172267383077338
7	0.5148796927373593
8	0.5226833031518507
9	0.531285047555801
10	0.542347000737079

If we see from the graph, then we can get the figure as below:



Unfortunately, sever 6, 7, 8 are still very close to each other, it's still hard to say which one is the best for dealing with the coming jobs. We can only simply say that the sever 8 has the biggest time while sever 7 has the smallest one. Then, we introduce paired-t test to see which one is better exactly by applying the code below.



```
def mean_confidence_interval(data, confidence=0.95):
    a = 1.0*np.array(data)
    n = len(a)
    m, se = np.mean(a), scipy.stats.sem(a)
    h = se * sp.stats.t._ppf((1+confidence)/2., n-1)
    return m, m-h, m+h
```

The results is shown below.

REPLICATION	EMRT S2 - EMRT S1	Mean Value	95% CONF INTERVAL
5	8 - 7	0.00889966084391	[0.00592357616854, 0.0118757455193]
10	8 - 7	0.00780361041449	[0.00632153542224, 0.00928568540674]
20	8 - 7	0.00691984589635	[0.00580026945627, 0.00803942233642]

From this table, it can be see that the interval with 7 severs are always better than 8, since the 95% confidant interval is always greater then 0, which mean s that the mean response time of 7 is always shorter than 8.

Then we compare the difference of 6 and 7, which has a smaller gap in between. So I raise the replication to 100 times, and the 95% confidence interval can be shown as below.

REPLICATION	EMRT S2 - EMRT S1	Mean Value	95% CONF INTERVAL
5	6 - 7	0.00173285518233	[-0.000578425985028, 0.00404413634968]
10	6 - 7	0.00234704557037	[0.00129136818261, 0.00340272295814]
20	6 - 7	0.00234279492735	[0.00112893508152, 0.00355665477319]
50	6 - 7	0.0022263578389	[0.00140300928973, 0.00304970638808]
100	6 - 7	0.00260740059425	[0.00197894397017, 0.00323585721833]

Although with the growing replication of running times, the confidence interval narrows down, the boundary is also higher than 0. So, we could draw the conclusion that the 7 sever will has a shorter mean response time compared with other number of severs/

Part 5 - Conclusion

Based on the simulation and the analyse all above, we could get the conclusion that 7 severs is the best choice of this farm.

Note that the program is written by Python and it is available in the appendix, named Project.py.