COMP9334 Project

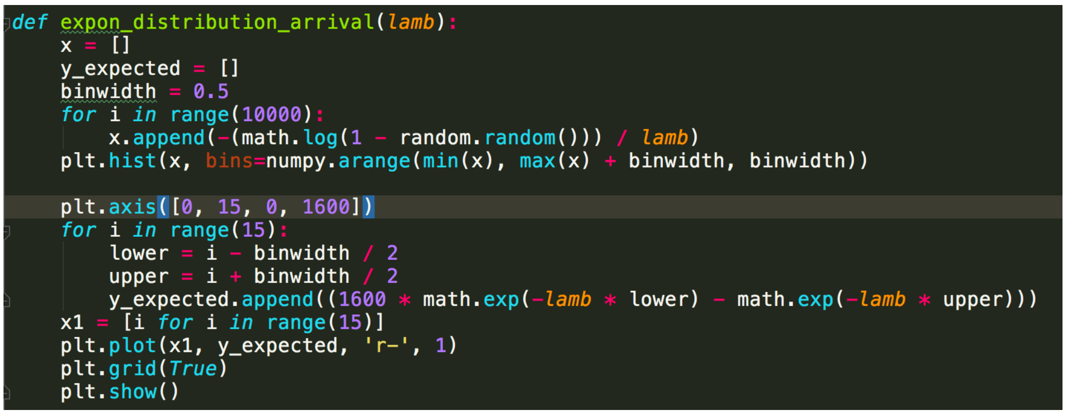
Session 1, 2018

Z5132086 Jingxuan Li - 17 May 2018

1. Probability distributions

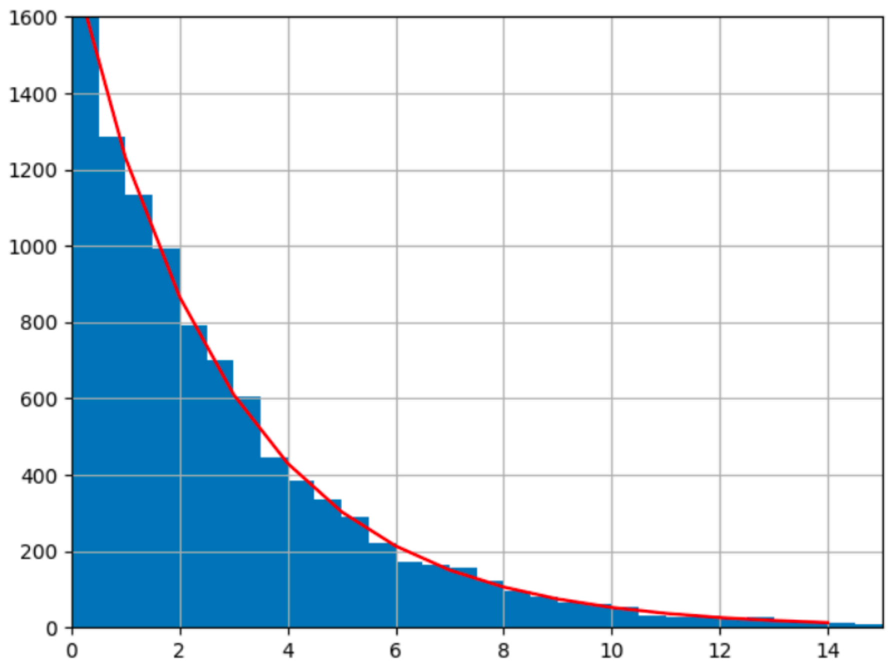
1.1.The probability distribution of arrival time:

First, we need to make sure that the arrival time is correct, which is exponential distribution. So, we generate a sequence random number {U_1, U_2, U_3 …} which is uniformly distributed in (0, 1), and the sequence -log(1-U_k) / λ should be exponential distribution with rate λ.



```
def expon_distribution_arrival(lamb):
    x = []
    y_expected = []
    binwidth = 0.5
    for i in range(10000):
        x.append(-(math.log(1 - random.random())) / lamb)
    plt.hist(x, bins=numpy.arange(min(x), max(x) + binwidth, binwidth))

    plt.axis([0, 15, 0, 1600])
    for i in range(15):
        lower = i - binwidth / 2
        upper = i + binwidth / 2
        y_expected.append((1600 * math.exp(-lamb * lower) - math.exp(-lamb * upper)))
    x1 = [i for i in range(15)]
    plt.plot(x1, y_expected, 'r-', 1)
    plt.grid(True)
    plt.show()
```

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



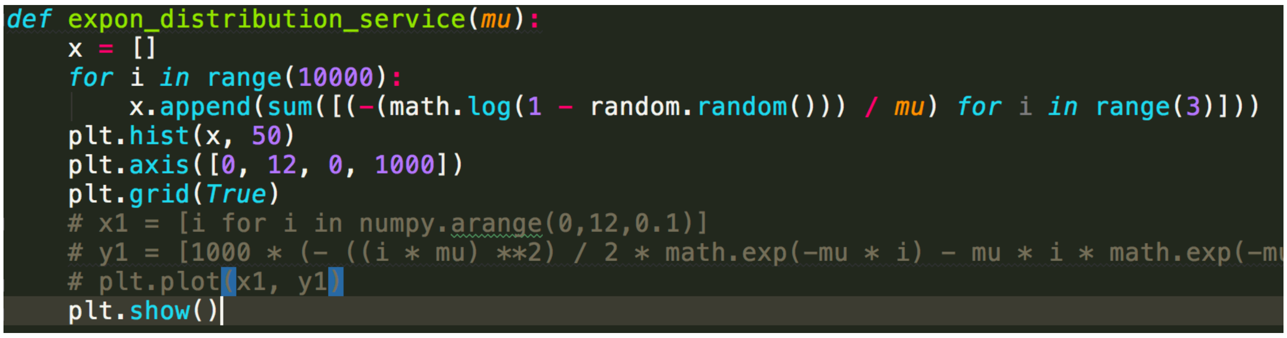
The plots shows:

1. The histogram of the numbers generated by -log(1-U_k) / λ in bins.

2.The red line shows the expected number of exponential distributed numbers in each bin.

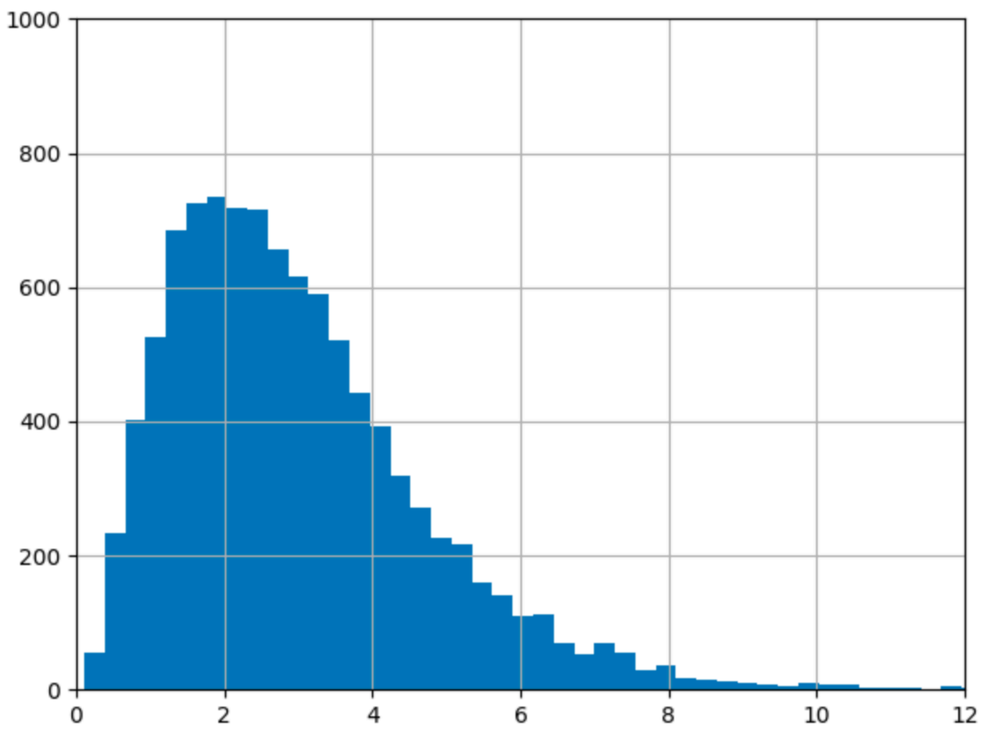
1.2.The probability distribution of service time:

Each Service time s_k is the sum of three random numbers s_{1k}, s_{2k} and s_{3k}, i.e $s_k = s_{1k} + s_{2k} + s_{3k}$ $\forall k = 1, 2, \ldots$ where s_{1k}, s_{2k} and s_{3k} are exponentially distributed random numbers with parameter μ. So, we generate a sequence random number $\{U_{1k}, U_{2k}, U_{3k} \ldots\}$ which is uniformly distributed in (0, 1), and the sequence $s_{1k} = -\log(1-U_{1k}) / \mu$, $s_{2k} = -\log(1-U_{2k}) / \mu$, $s_{3k} = -\log(1-U_{3k}) / \mu$, which s_{1k}, s_{2k} and s_{3k} are exponentially distributed random numbers, thus, $s_k = -\log(1-U_{1k})(1-U_{2k})(1-U_{3k}) / \mu$.



```
def expon_distribution_service(mu):
    x = []
    for i in range(10000):
        x.append(sum([(-(math.log(1 - random.random())) / mu) for i in range(3)]))
    plt.hist(x, 50)
    plt.axis([0, 12, 0, 1000])
    plt.grid(True)
    # x1 = [i for i in numpy.arange(0,12,0.1)]
    # y1 = [1000 * (- ((i * mu) **2) / 2 * math.exp(-mu * i) - mu * i * math.exp(-mu
    # plt.plot(x1, y1)
    plt.show()
```

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



2. Verification of simulation program

2.1. Description about the program

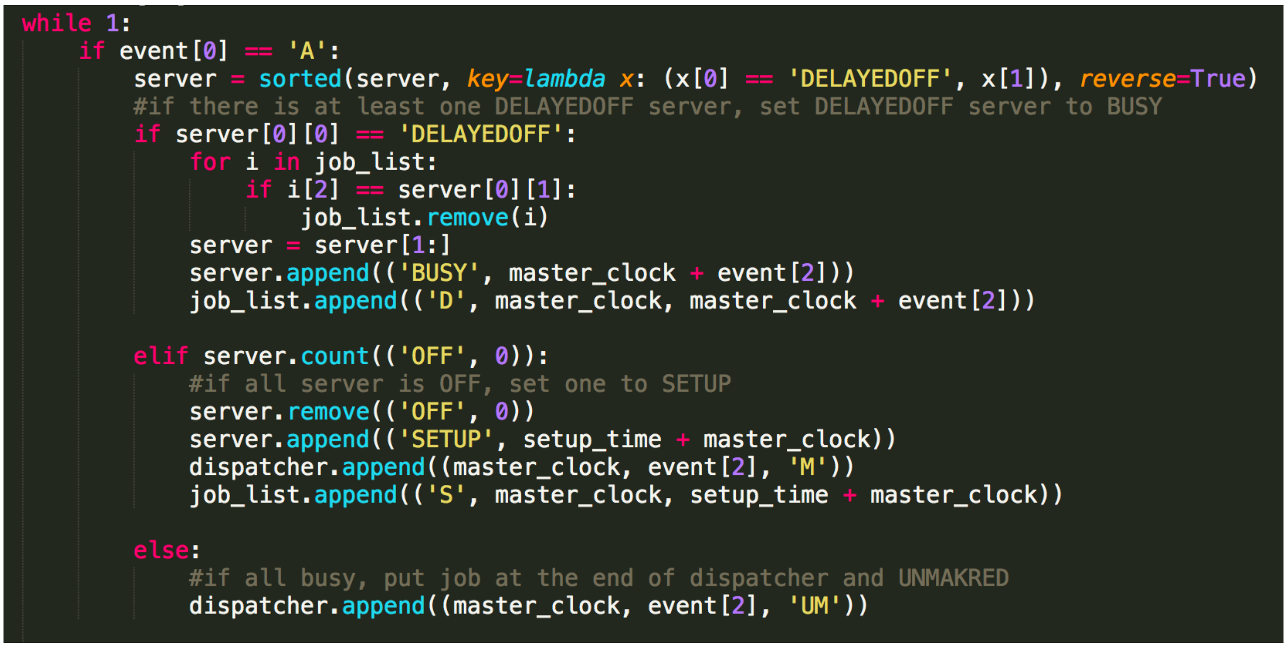
There are 4 events , ‘A’ stands for arrival, ‘D’ stands for departure, ’S’ stands for set up and ‘E’ stands for delayed off.

There are 4 status in server: OFF, SETUP, BUSY, DELAYEDOFF. The server infos were generated in a list like [('SETUP', 82.0), ('SETUP', 70.0), ('BUSY', 61.0)] ,where(‘SETUP', 82.0) stands for it will 82 seconds to finish setting up the server

All the jobs in queue were generated in a list named dispatcher.

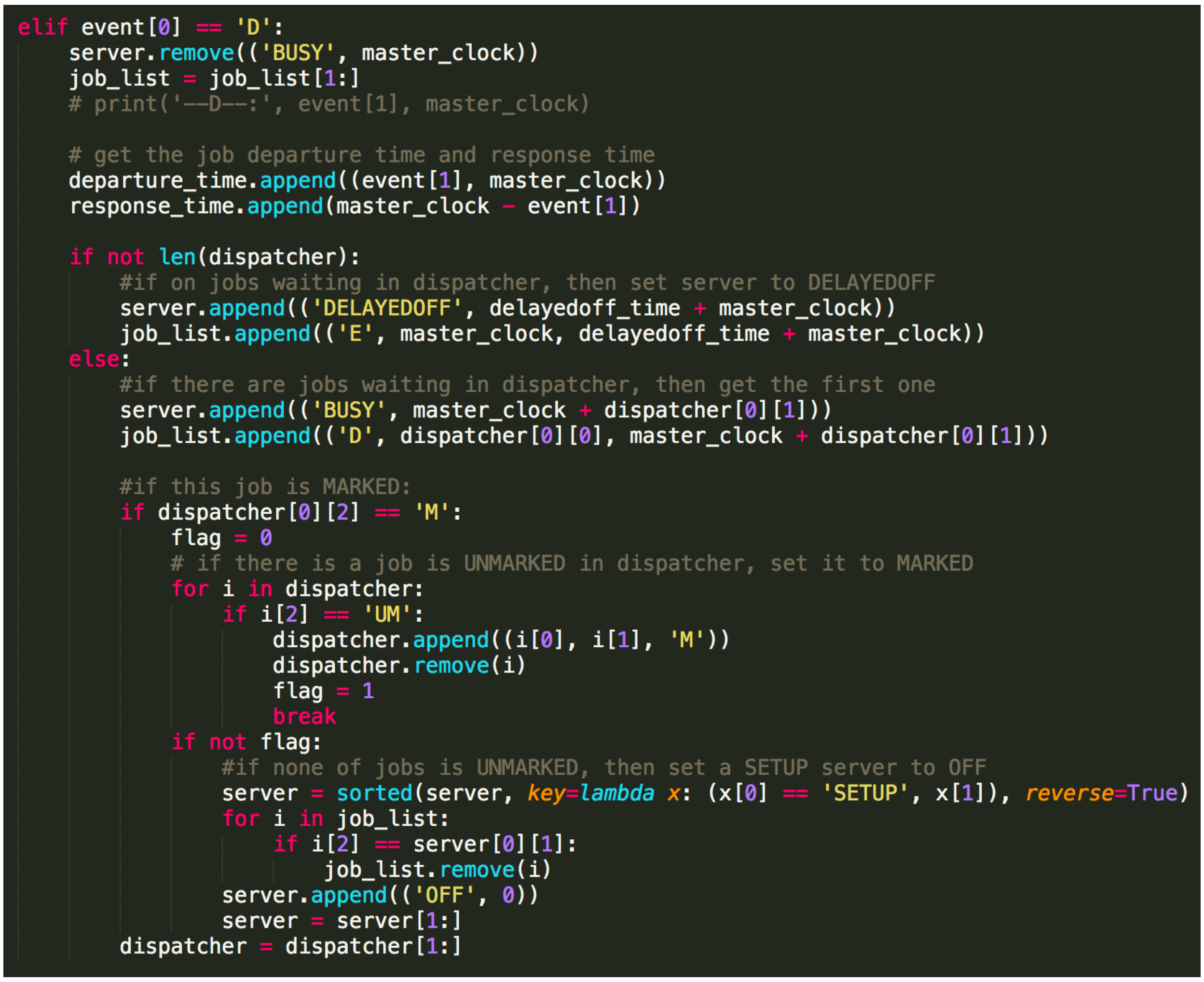
All the event to be triggered were generated in a list named job_list.

If the event type is ‘arrival’, first check whether there is a server in DELAYEDOFF status. If there is at least one, let it process the job and set it to busy status. If there all servers are in OFF status, then set one to SETUP. And put the job into dispatcher and set it to MARKED. If all the servers are in ‘busy’ status, then put the job into dispatcher and set it to UNMARKED’.



```
while 1:
    if event[0] == 'A':
        server = sorted(server, key=lambda x: (x[0] == 'DELAYEDOFF', x[1]), reverse=True)
        #if there is at least one DELAYEDOFF server, set DELAYEDOFF server to BUSY
        if server[0][0] == 'DELAYEDOFF':
            for i in job_list:
                if i[2] == server[0][1]:
                    job_list.remove(i)
            server = server[1:]
            server.append(('BUSY', master_clock + event[2]))
            job_list.append(('D', master_clock, master_clock + event[2]))

        elif server.count(('OFF', 0)):
            #if all server is OFF, set one to SETUP
            server.remove(('OFF', 0))
            server.append(('SETUP', setup_time + master_clock))
            dispatcher.append((master_clock, event[2], 'M'))
            job_list.append(('S', master_clock, setup_time + master_clock))

        else:
            #if all busy, put job at the end of dispatcher and UNMAKRED
            dispatcher.append((master_clock, event[2], 'UM'))
```

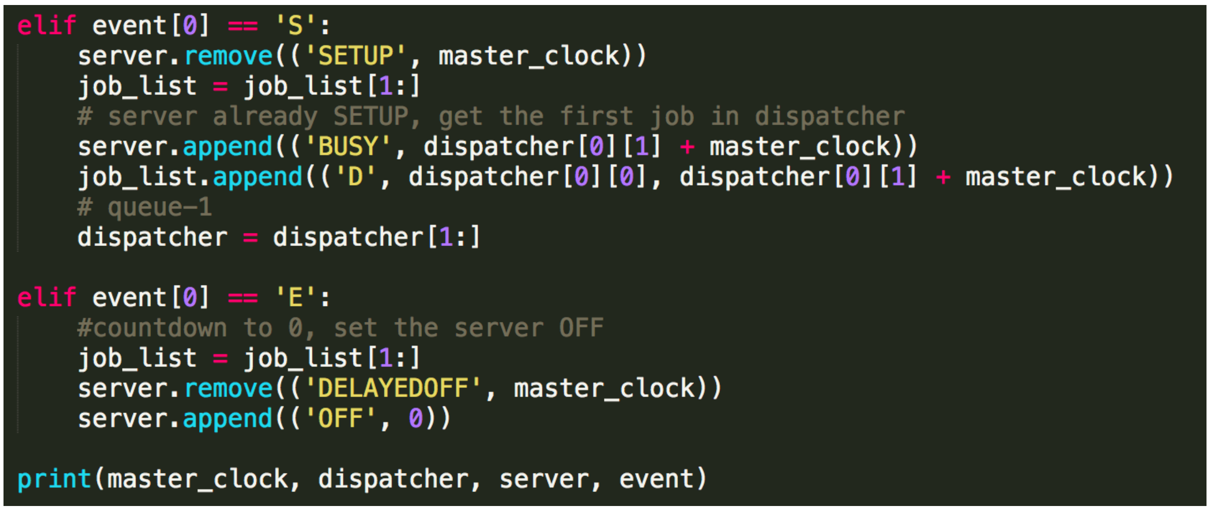
If the event type is ‘departure’, first check the dispatcher. If the dispatcher queue is empty, then the server will change its state from BUSY to DELAYEDOFF. If there are jobs waiting in queue, then get the first one. The server is still in BUSY status.We need to check the job is MARKED or not. If the job is MARKED, then check whether there is a job which is UNMARKED in queue. If there is one , then set it to MARKED. But if not, we need to set a SETUP server to OFF status.



```python
elif event[0] == 'D':
    server.remove(('BUSY', master_clock))
    job_list = job_list[1:]
    # print('--D--:', event[1], master_clock)

    # get the job departure time and response time
    departure_time.append((event[1], master_clock))
    response_time.append(master_clock - event[1])

    if not len(dispatcher):
        #if on jobs waiting in dispatcher, then set server to DELAYEDOFF
        server.append(('DELAYEDOFF', delayedoff_time + master_clock))
        job_list.append(('E', master_clock, delayedoff_time + master_clock))
    else:
        #if there are jobs waiting in dispatcher, then get the first one
        server.append(('BUSY', master_clock + dispatcher[0][1]))
        job_list.append(('D', dispatcher[0][0], master_clock + dispatcher[0][1]))

        #if this job is MARKED:
        if dispatcher[0][2] == 'M':
            flag = 0
            # if there is a job is UNMARKED in dispatcher, set it to MARKED
            for i in dispatcher:
                if i[2] == 'UM':
                    dispatcher.append((i[0], i[1], 'M'))
                    dispatcher.remove(i)
                    flag = 1
                    break
            if not flag:
                #if none of jobs is UNMARKED, then set a SETUP server to OFF
                server = sorted(server, key=lambda x: (x[0] == 'SETUP', x[1]), reverse=True)
                for i in job_list:
                    if i[2] == server[0][1]:
                        job_list.remove(i)
                server.append(('OFF', 0))
                server = server[1:]
        dispatcher = dispatcher[1:]
```

If the event type is 'setup', it stands for the server finished setting up. Then we get the first job in queue. And set the server to BUSY.

If the event type is 'delayedoff', it stands for the server finished delaying off. Then we set the server to OFF.



```python
elif event[0] == 'S':
    server.remove(('SETUP', master_clock))
    job_list = job_list[1:]
    # server already SETUP, get the first job in dispatcher
    server.append(('BUSY', dispatcher[0][1] + master_clock))
    job_list.append(('D', dispatcher[0][0], dispatcher[0][1] + master_clock))
    # queue-1
    dispatcher = dispatcher[1:]

elif event[0] == 'E':
    #countdown to 0, set the server OFF
    job_list = job_list[1:]
    server.remove(('DELAYEDOFF', master_clock))
    server.append(('OFF', 0))

print(master_clock, dispatcher, server, event)
```

2.2. Run the simulation case given by the PDF (Example 1)

The results are shown below, which is generated by my simulation. The results are exactly same to the Example 1 given by PDF.

Master Clock	Dispatcher	Server	Event
0	[]	[('OFF', 0), ('OFF', 0), ('OFF', 0)]	['']
10.0	[(10.0, 1.0, 'M')]	[('OFF', 0), ('OFF', 0), ('SETUP', 60.0)]	('A', 10.0, 1.0)
20.0	[(10.0, 1.0, 'M'), (20.0, 2.0, 'M')]	[('SETUP', 60.0), ('OFF', 0), ('SETUP', 70.0)]	('A', 20.0, 2.0)
32.0	[(10.0, 1.0, 'M'), (20.0, 2.0, 'M'), (32.0, 3.0, 'M')]	[('SETUP', 70.0), ('SETUP', 60.0), ('SETUP', 82.0)]	('A', 32.0, 3.0)
33.0	[(10.0, 1.0, 'M'), (20.0, 2.0, 'M'), (32.0, 3.0, 'M'), (33.0, 4.0, 'UM')]	[('SETUP', 82.0), ('SETUP', 70.0), ('SETUP', 60.0)]	('A', 33.0, 4.0)
60.0	[(20.0, 2.0, 'M'), (32.0, 3.0, 'M'), (33.0, 4.0, 'UM')]	[('SETUP', 82.0), ('SETUP', 70.0), ('BUSY', 61.0)]	('S', 10.0, 60.0)
61.0	[(32.0, 3.0, 'M'), (33.0, 4.0, 'M')]	[('SETUP', 82.0), ('SETUP', 70.0), ('BUSY', 63.0)]	('D', 10.0, 61.0)
63.0	[(33.0, 4.0, 'M')]	[('SETUP', 70.0), ('BUSY', 66.0), ('OFF', 0)]	('D', 20.0, 63.0)
66.0	[]	[('BUSY', 70.0), ('OFF', 0), ('OFF', 0)]	('D', 32.0, 66.0)
70.0	[]	[('OFF', 0), ('OFF', 0), ('DELAYEDOFF', 170.0)]	('D', 33.0, 70.0)
170.0	[]	[('OFF', 0), ('OFF', 0), ('OFF', 0)]	('E', 70.0, 170.0)

departure_1.txt is show below

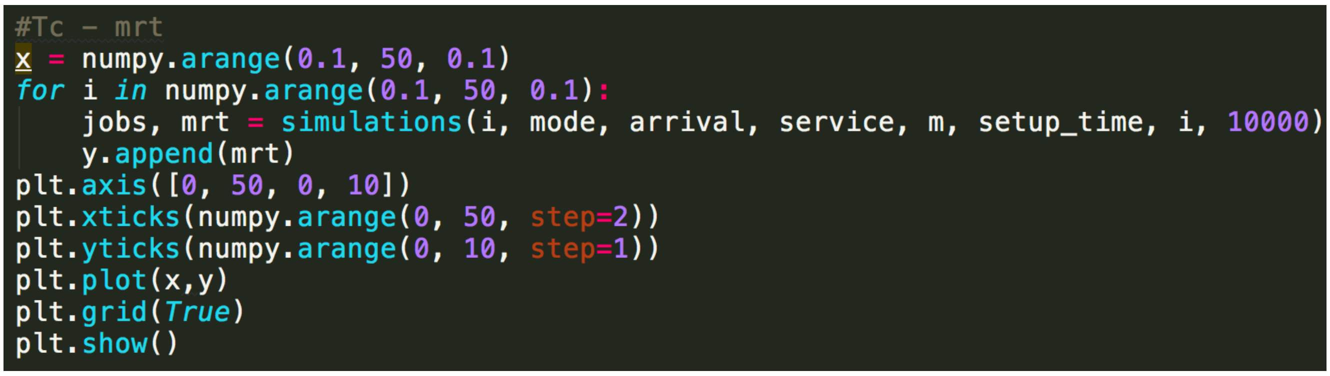
	Arrvial time	Departure time
Job1	10.000	61.000
Job2	20.000	63.000
Job3	32.000	66.000
Job4	33.000	70.000

Mean response time is 41.250

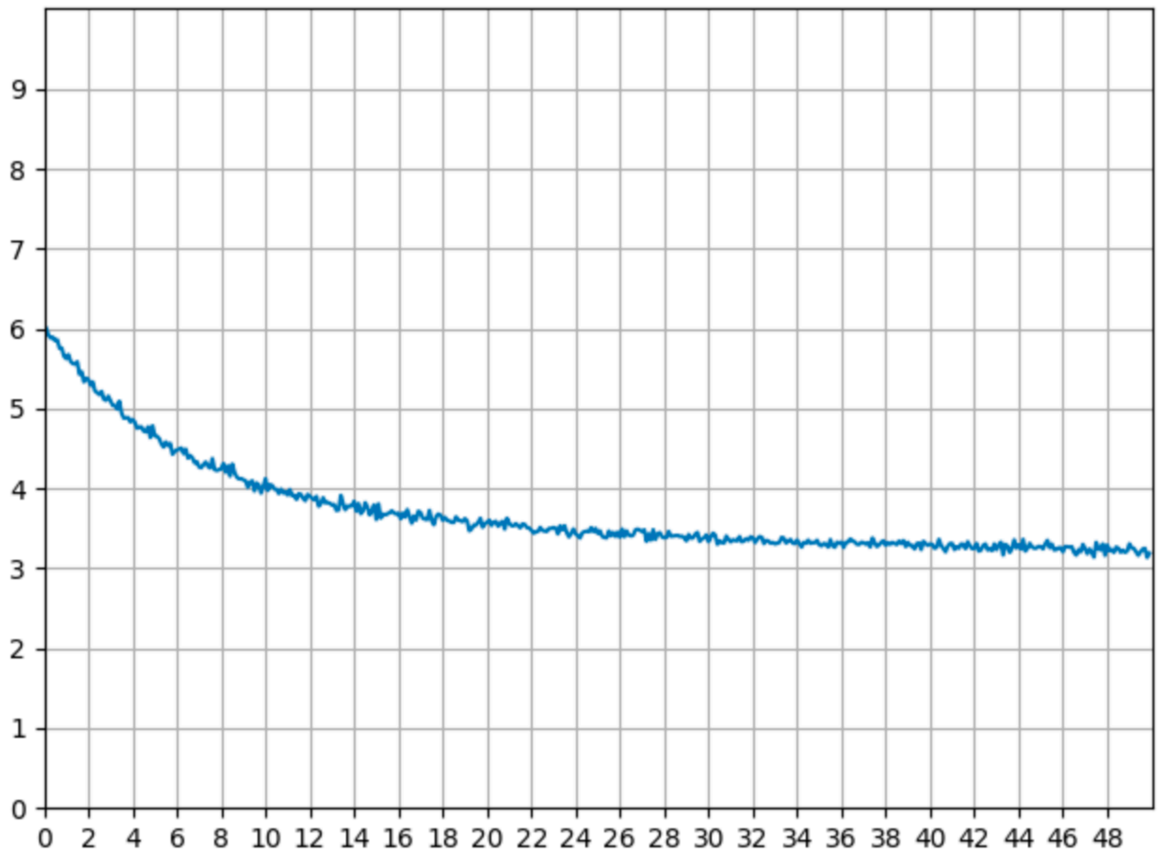
3. Analysis of the simulation

3.1. Find improved system Tc

We assume the end time is 10000 seconds, then get the plot of the mean response time with Tc from 0 to 50 seconds. (random seed)



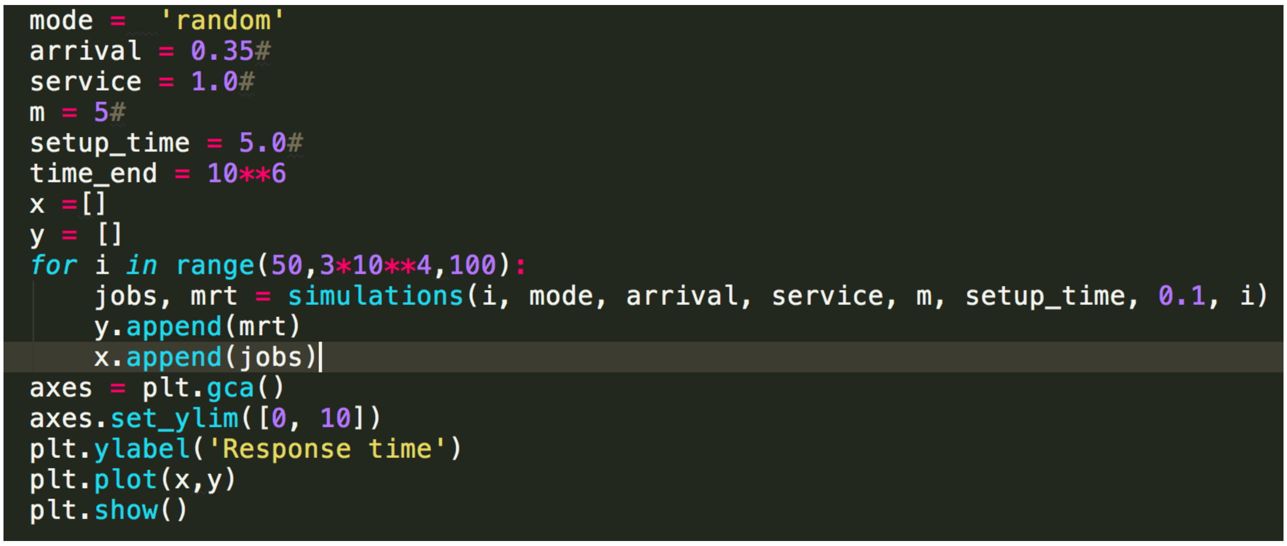
```
#Tc - mrt
x = numpy.arange(0.1, 50, 0.1)
for i in numpy.arange(0.1, 50, 0.1):
    jobs, mrt = simulations(i, mode, arrival, service, m, setup_time, i, 10000)
    y.append(mrt)
plt.axis([0, 50, 0, 10])
plt.xticks(numpy.arange(0, 50, step=2))
plt.yticks(numpy.arange(0, 10, step=1))
plt.plot(x,y)
plt.grid(True)
plt.show()
```



It shows that the mean response time of baseline system is around 6 seconds, therefore we must improve it to around 4 seconds, where the Tc of the improved system is approximately about 10 seconds. We need more analysis to prove it.

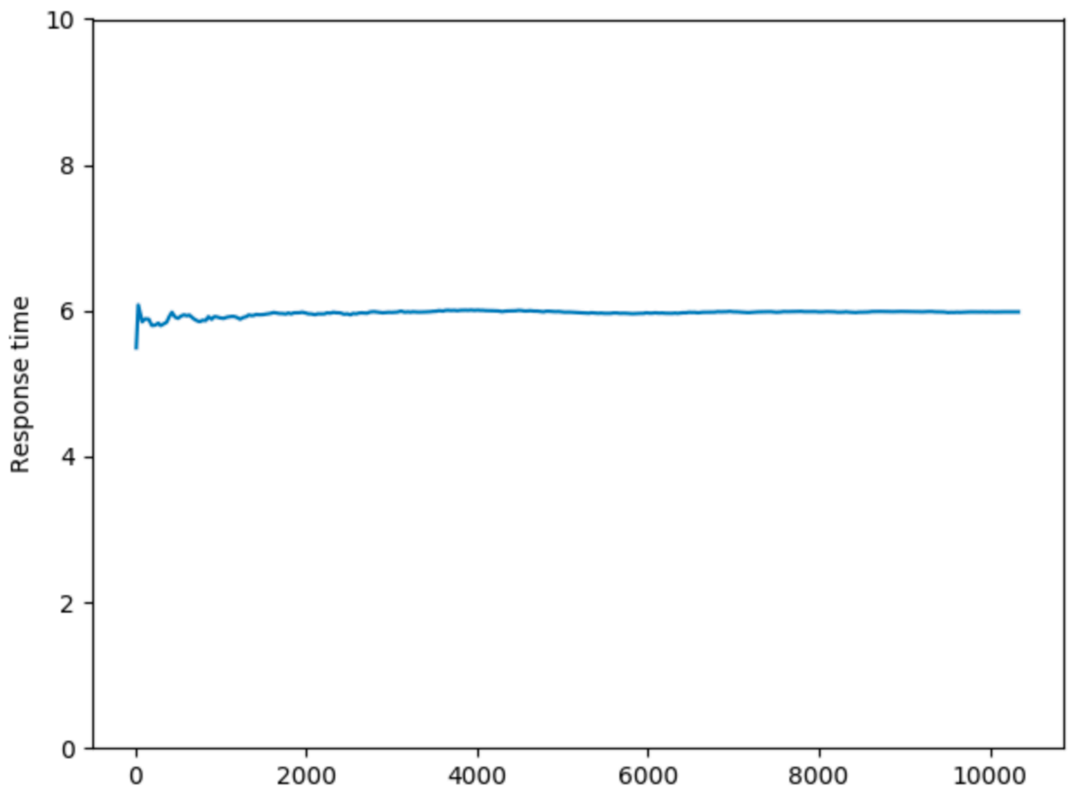
3.2. Transient removal

When end_time = 30000, jobs are around 10000, we generate the plot of the mean response time of first k jobs.



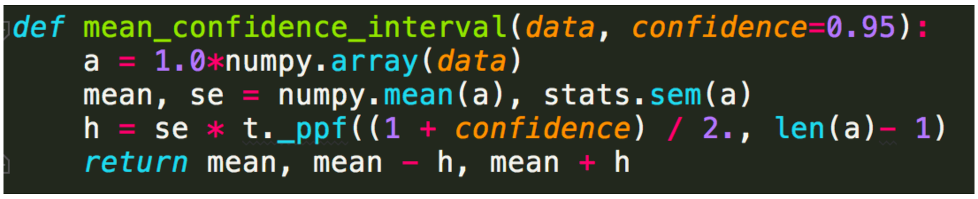
```
mode =  'random'
arrival = 0.35#
service = 1.0#
m = 5#
setup_time = 5.0#
time_end = 10**6
x =[]
y = []
for i in range(50,3*10**4,100):
    jobs, mrt = simulations(i, mode, arrival, service, m, setup_time, 0.1, i)
    y.append(mrt)
    x.append(jobs)
axes = plt.gca()
axes.set_ylim([0, 10])
plt.ylabel('Response time')
plt.plot(x,y)
plt.show()
```

The plot generate by program is shown below:



It shows that the transient part of data ends approximately after 2000 jobs. And the end_time is around 5000s.

3.3. confidence interval



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

We generate the mean response time of the Tc from 10.0 s to 10.5 s. As the table shown below, if we want to improve the response time 2 units less than that of the baseline system, the Tc should at least be 10.2 seconds. We will do more experiments to compare Tc =10.2 and Tc = 10.3 with the baseline system.

Tc	95% confidence interval of EMRT System	Mean response time	Mean response time - EMRT of Baseline system
0.1	[6.011930001396703, 6.036600773581776]	6.02426538748924	
10.0	[4.015738057318413, 4.048592851694539]	4.032165454506476	1.9920999329827636
10.1	[4.005926574176925, 4.034699006103677]	4.020312790140301	1.9949776124787295
10.2	[3.9999839123259497, 4.0334463207380535]	4.016715116532001	2.000951766970794
10.3	[3.9961200088728037, 4.025729831204359]	4.0109249200385815	2.0037219061455973
10.4	[3.983622829550288, 4.014139907140294]	3.998881368345291	2.0164676574896667
10.5	[3.9609451049891753, 3.992288458696895]	3.976616781843035	2.045307804322938

3.4. Compare two systems

3.4.1. Tc = 10.2 s

System 1: #servers = 5, setup time = 5, $\lambda = 0.35$, $\mu = 1$, Tc = 0.1 s

System 2: #servers = 5, setup time = 5, $\lambda = 0.35$, $\mu = 1$, Tc = 10.2 s

EMRT = estimated mean response time

	EMRT System 1	EMRT System 2	EMRT System2 -System 1
Rep.1	6.026826606955027	4.002171405952159	2.0246552010028687
Rep.2	6.023128870769336	4.023069800560054	2.0000590702092813
Rep.3	6.020388655366779	4.0230121567661845	1.9973764986005946
Rep.4	6.02326346837982	4.013528359842377	2.009735108537443

As the table shown above, EMRT of system 2 are 2 units less than system 1 in 3 of the replications. But in replication 3, the mean response time did not improve 2 unit time. Hence, Tc = 10.2s is no long satisfied the requirement.

3.4.2. Tc = 10.3 s

System 1: #servers = 5, setup time = 5, λ = 0.35, μ = 1, Tc = 0.1 s

System 2: #servers = 5, setup time = 5, λ = 0.35, μ = 1, Tc = 10.3 s

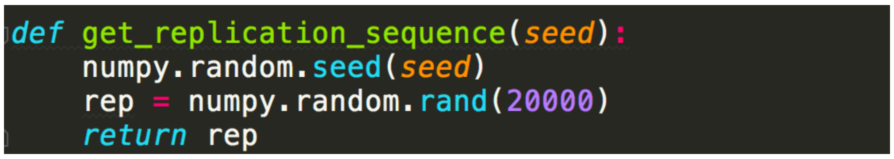
We do the same experiments for the system of Tc = 10.3 s

	EMRT System 1	EMRT System 2	EMRT System2 -System 1
Rep.1	6.02096130768923	4.006969259697315	2.0139920479919153
Rep.2	6.032851709420109	4.0137092348470915	2.0191424745730178
Rep.3	6.021035377696807	4.003860861101397	2.01717451659541
Rep.4	6.028581509620203	4.009857442055167	2.018724067565036

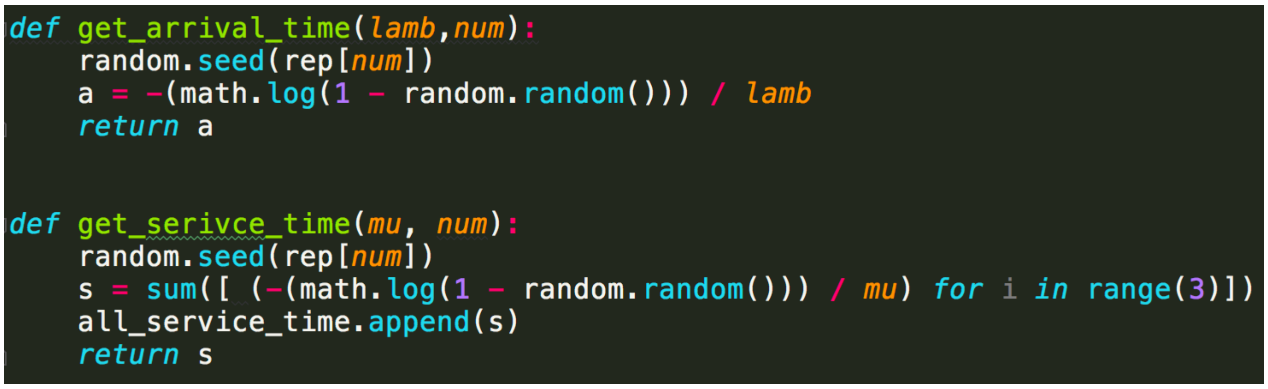
As the table shown above, EMRT of system 2 are 2 units less than system 1 in all 4 replications.

Using common random numbers method(CNR):

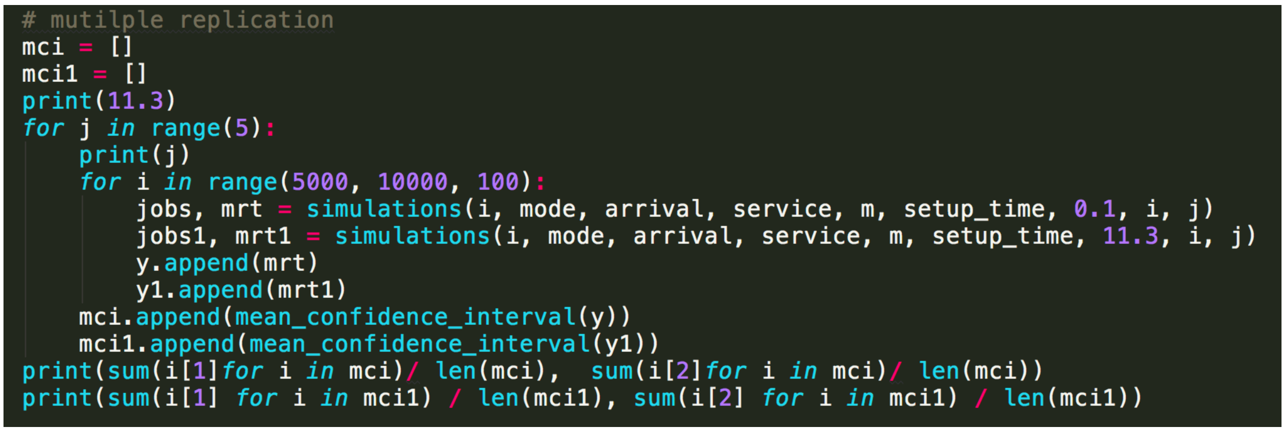
In each replication, we need to generate one arrival and one service time sequence. And we use a seed to generate a sequence. Then we use the each element in sequence as random seed to generate arrival and service time . Thus, with the same seed, the sequence will be the same. In next replication, we change the seed which use to generate the sequence.



```
def get_replication_sequence(seed):
    numpy.random.seed(seed)
    rep = numpy.random.rand(20000)
    return rep
```



```python
def get_arrival_time(lamb,num):
    random.seed(rep[num])
    a = -(math.log(1 - random.random())) / lamb
    return a


def get_serivce_time(mu, num):
    random.seed(rep[num])
    s = sum([ (-(math.log(1 - random.random())) / mu) for i in range(3)])
    all_service_time.append(s)
    return s
```



```python
# mutilple replication
mci = []
mci1 = []
print(11.3)
for j in range(5):
    print(j)
    for i in range(5000, 10000, 100):
        jobs, mrt = simulations(i, mode, arrival, service, m, setup_time, 0.1, i, j)
        jobs1, mrt1 = simulations(i, mode, arrival, service, m, setup_time, 11.3, i, j)
        y.append(mrt)
        y1.append(mrt1)
    mci.append(mean_confidence_interval(y))
    mci1.append(mean_confidence_interval(y1))
print(sum(i[1]for i in mci)/ len(mci),  sum(i[2]for i in mci)/ len(mci))
print(sum(i[1] for i in mci1) / len(mci1), sum(i[2] for i in mci1) / len(mci1))
```

Run the experiments by replications, we got the result below:

#independent replicaitons	95% confidence interval of EMRT System 1	95% confidence interval of EMRT System 2	95% confidence interval of EMRT System2 -System 1
5	[6.009454088248466, 6.028646129889344]	[3.9941788101263165, 4.013774618854423]	[2.0152752781221497 2.0148715110349205]
10	[6.017727401113873, 6.032298030545635]	[3.998110491065417, 4.011824953199932]	[2.019616910048456, 2.020473077345703]
15	[6.016720186428303, 6.02711139061133]	[3.996313439937218, 4.0078294353032335]	[2.0204067464910853, 2.019281955308096]
20	[6.017741717442282, 6.027825235813683]	[4.002328035354526, 4.012654014584063]	[2.0154136820877557, 2.01517122122962]

By using CNR, we can observe that the confidence interval of EMRT System2 are 2 units less than system 1 in all replications.

5. Conclusion

Based the simulation and analysis above, we can prove that Tc =10.3 s can improve the response time of system 2 units less then baseline system.