2018

Server setup in data centres

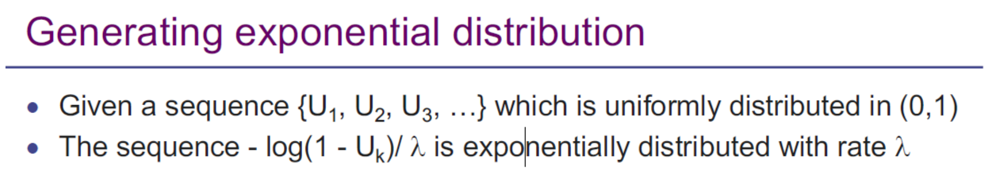
COMP9334 Project

YUYOU XU Z5143390

Content

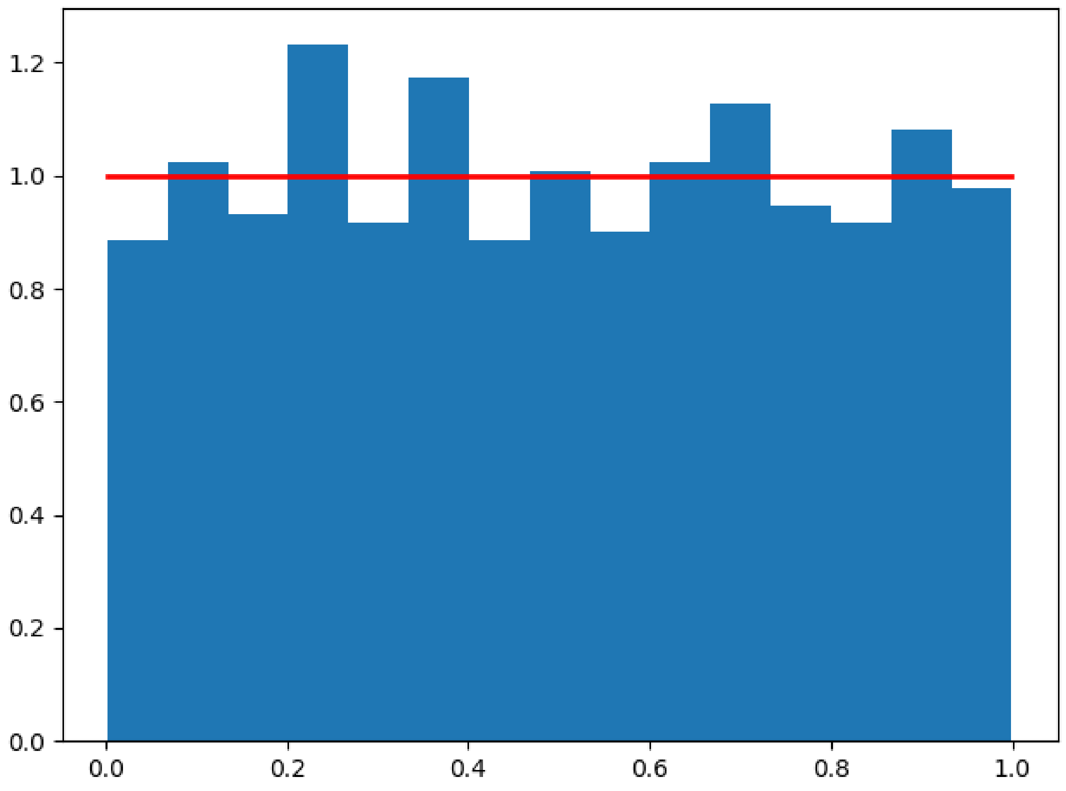
Correctness of the inter-arrival probability distribution and service time distribution

To generate exponentially distributed random numbers with parameter mu, the rule is followed:

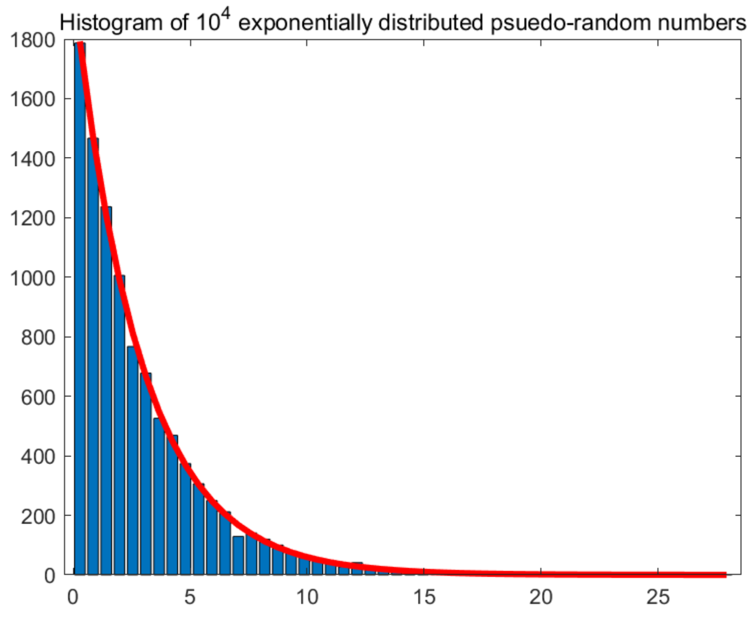


Generating exponential distribution

- Given a sequence $\{U_1, U_2, U_3, \ldots\}$ which is uniformly distributed in (0,1)
- The sequence $-\log(1 - U_k)/\lambda$ is exponentially distributed with rate λ

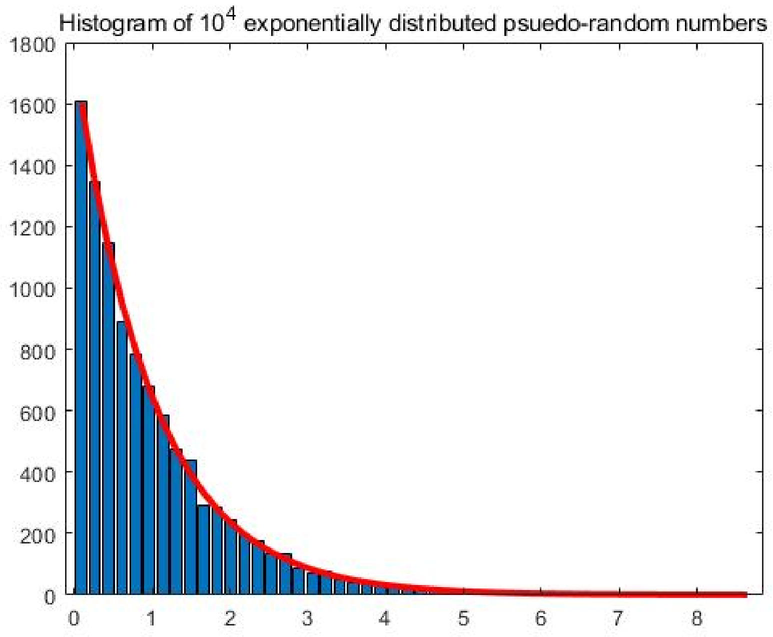


Uniform distribution in (0, 1)
Generated by support.py



Histogram of the numbers generated by "- math.log(1 - uniform(0,1)) / lmd" in 50 bins when **lambda = 0.35** (red as expected number of exponential distributed numbers in each bin)

Generated by hist_expon.m

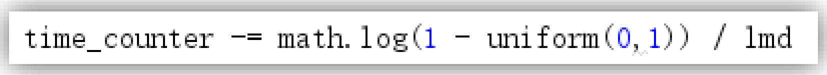


Histogram of the numbers generated by "- math.log(1 - uniform(0,1)) / miu" in 50 bins when **miu = 1** (red as expected number of exponential distributed numbers in each bin)

Generated by hist_expon.m

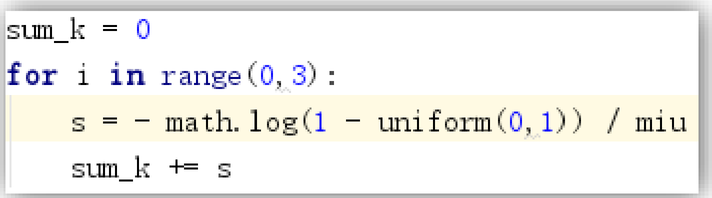
Specifically:

Simulation code 1 (inter-arrival probability distribution): arrival rate of the jobs is lambda,



```
time_counter -= math.log(1 - uniform(0,1)) / lmd
```

Simulation code 2 (service time distribution): each sk is the sum of three random numbers s1k, s2k and s3k



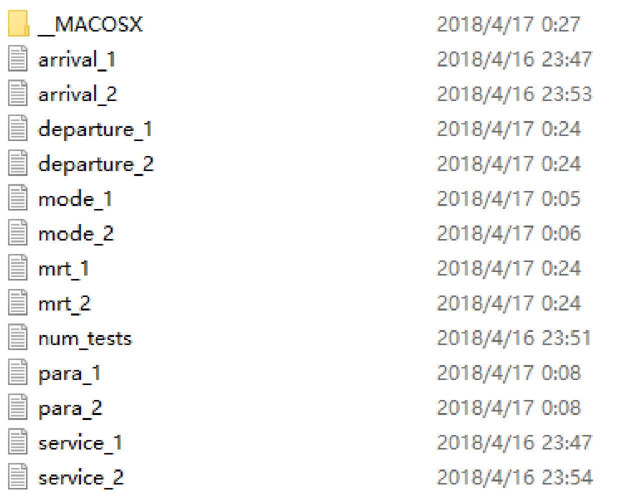
```
sum_k = 0
for i in range(0,3):
    s = - math.log(1 - uniform(0,1)) / miu
    sum_k += s
```

It is shown in last two graphics that they obey the rule of inter-arrival probability distribution and service time distribution.

Correctness of simulation code

There are two types of verification for the correctness of the code: visual inspection and consistency check.

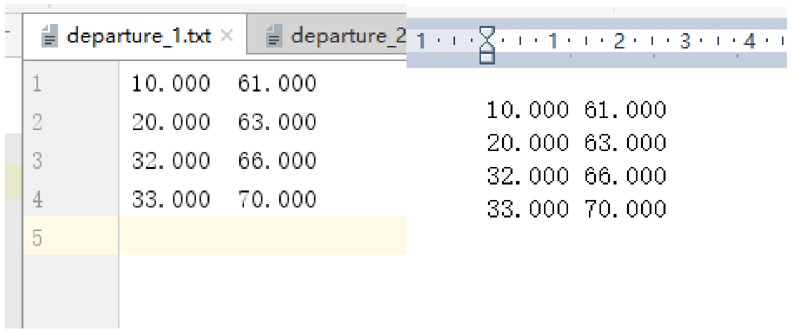
Firstly, the sample_1 file provided by professor is used:



__MACOSX	2018/4/17 0:27
arrival_1	2018/4/16 23:47
arrival_2	2018/4/16 23:53
departure_1	2018/4/17 0:24
departure_2	2018/4/17 0:24
mode_1	2018/4/17 0:05
mode_2	2018/4/17 0:06
mrt_1	2018/4/17 0:24
mrt_2	2018/4/17 0:24
num_tests	2018/4/16 23:51
para_1	2018/4/17 0:08
para_2	2018/4/17 0:08
service_1	2018/4/16 23:47
service_2	2018/4/16 23:54

Run the simulation code by using this file and comparing the output mrt and departure file with the relevant sample file to check whether the simulation.py work correctly.

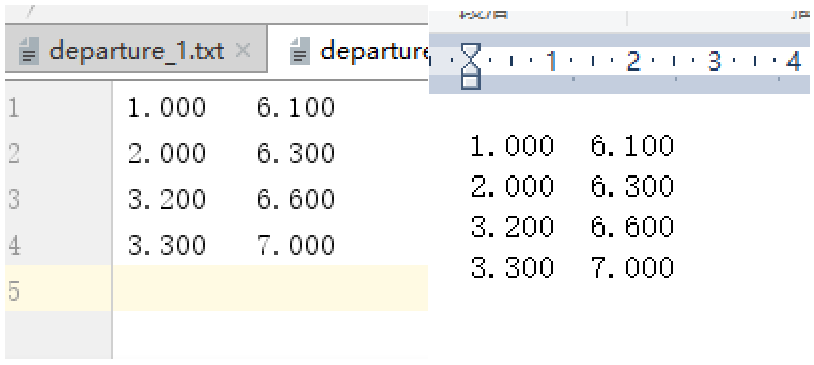
After simulation (**visual inspection**):



```
10.000   61.000
20.000   63.000
32.000   66.000
33.000   70.000
```

```
10.000 61.000
20.000 63.000
32.000 66.000
33.000 70.000
```

Comparison between wrapper.py file departure_1.txt (left) and sample file (right)



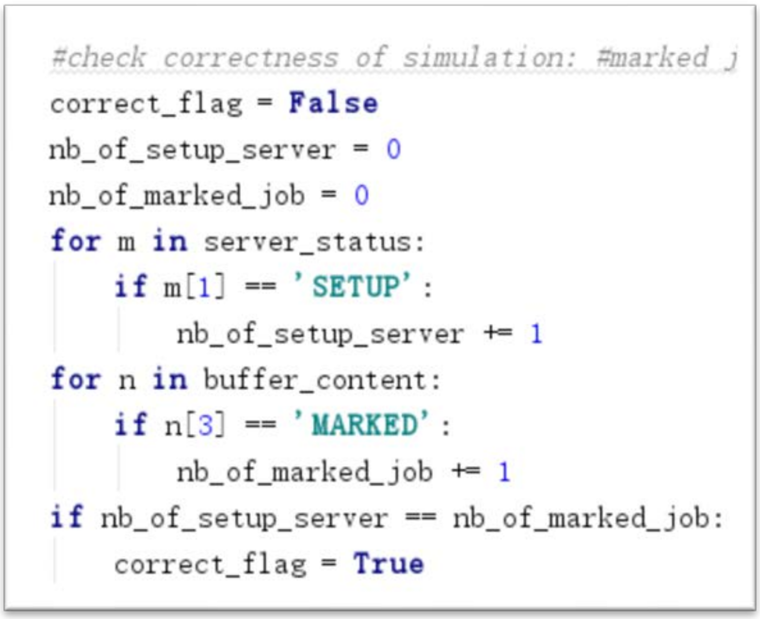
```
1.000   6.100
2.000   6.300
3.200   6.600
3.300   7.000
```

```
1.000   6.100
2.000   6.300
3.200   6.600
3.300   7.000
```

Comparison between wrapper.py output departure_2.txt and sample file

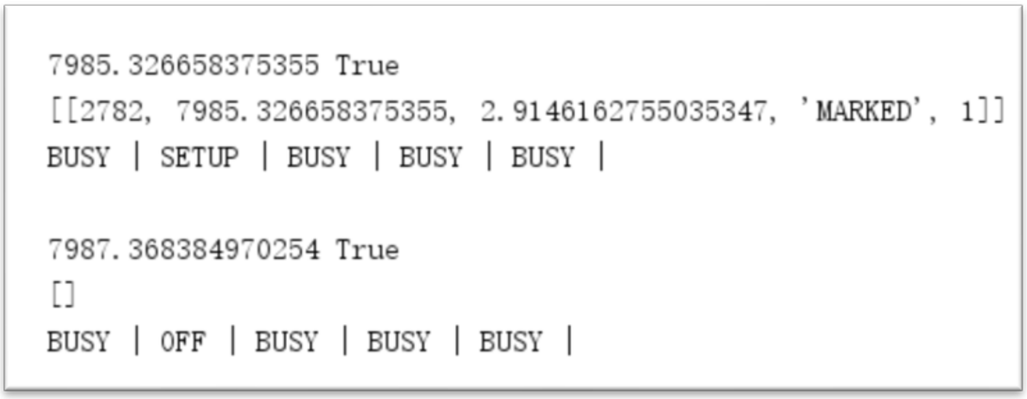
On the other hand, each time a ‘DEPARTURE’ event is coming, the number of ‘SETUP’ server and the number of ‘MARKED’ job is checked for verification, since the number of MARKED jobs in the queue should equal to the number of servers in the SETUP state:

Code in simulation.py (**consistency check**):



```
#check correctness of simulation: #marked j
correct_flag = False
nb_of_setup_server = 0
nb_of_marked_job = 0
for m in server_status:
    if m[1] == 'SETUP':
        nb_of_setup_server += 1
for n in buffer_content:
    if n[3] == 'MARKED':
        nb_of_marked_job += 1
if nb_of_setup_server == nb_of_marked_job:
    correct_flag = True
```

Sample output (‘True’ refers to the check result):



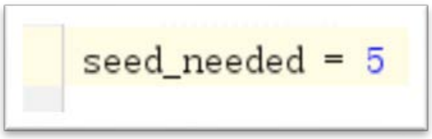
```
7985.326658375355 True
[[2782, 7985.326658375355, 2.9146162755035347, 'MARKED', 1]]
BUSY | SETUP | BUSY | BUSY | BUSY |

7987.368384970254 True
[]
BUSY | OFF | BUSY | BUSY | BUSY |
```

Thus, the correctness of the simulation is guaranteed.

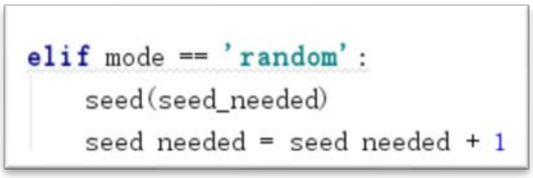
Reproducible verification

To simulate reproducibly, which means getting the same mean response time and departure information in a random-mode simulation, the seed should be configured. A **fixed seed** for the random number generator is used for this purpose:

Code in wrapper.py:



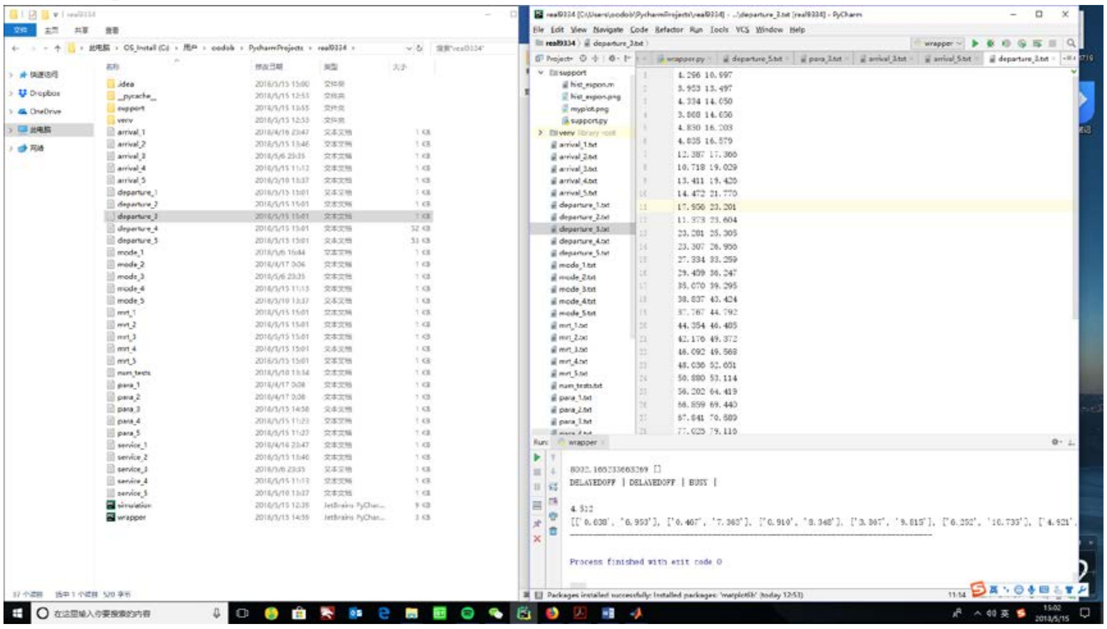
```python
seed_needed = 5
```



```python
elif mode == 'random':
    seed(seed_needed)
    seed_needed = seed_needed + 1
```

The seed is specified with a specific number “5”, ensuring that the same sequence of random numbers is given whenever the simulation process is turned on.

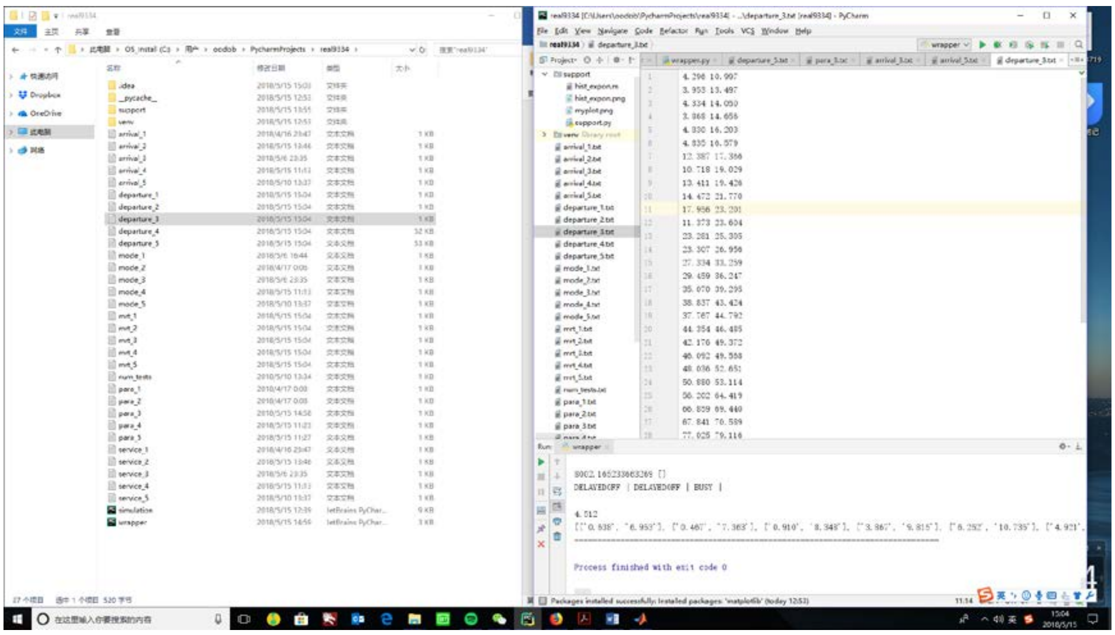
Verification (**visual inspection**):
Run the simulation 3 times to check if the same result is shown (take departure_3.txt as an example):



```
4.296 10.997
3.953 13.497
4.334 14.050
3.868 14.050
4.830 16.203
4.835 16.579
12.387 17.366
10.718 19.029
13.411 19.420
14.472 21.770
17.956 23.201
11.373 23.604
23.201 25.305
23.307 26.956
27.334 33.259
29.409 36.247
35.070 39.295
38.837 43.424
37.767 44.792
44.354 46.405
42.176 49.372
46.092 49.568
48.036 52.651
50.880 53.114
56.202 64.419
66.859 69.440
67.841 70.889
77.025 79.116
```

```
8002.165233663269 []
DELAYEDOFF | DELAYEDOFF | BUSY |

4.512
[['0.038', '6.953'], ['0.407', '7.363'], ['0.910', '8.348'], ['3.807', '9.815'], ['6.252', '10.735'], ['4.921',
------------------------------------------------------------------------------------------------

Process finished with exit code 0
```

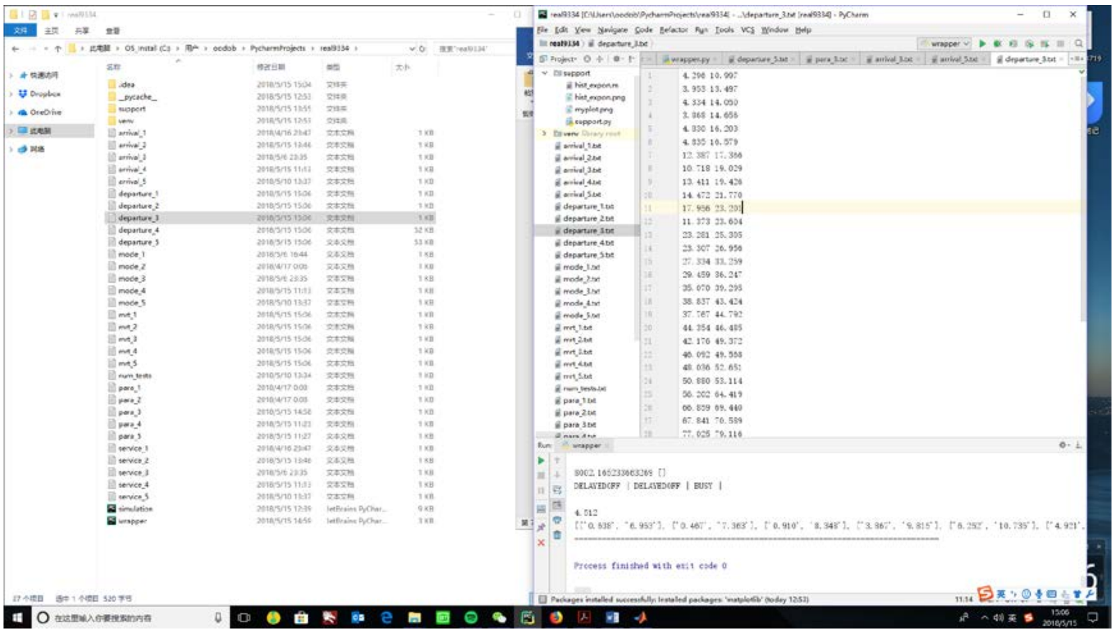
15:01 pm



```
4.298 10.997
3.953 13.497
4.334 14.050
3.868 14.656
4.330 16.203
4.835 16.579
12.387 17.386
10.718 19.029
13.411 19.426
14.472 21.770
17.956 23.201
11.373 23.604
23.281 25.305
23.307 26.956
27.334 33.259
29.459 36.247
35.070 39.295
38.837 43.424
37.767 44.792
44.354 46.485
42.176 49.372
46.092 49.558
48.036 52.651
50.880 53.114
56.202 64.419
66.859 69.440
67.841 70.589
77.025 79.116
```

```
8002.165233063269 []
DELAYEDOFF | DELAYEDOFF | BUSY |

4.512
[['0.638', '6.953'], ['0.467', '7.363'], ['0.910', '8.348'], ['3.867', '9.815'], ['6.252', '10.735'], ['4.921',
--------------------------------------------------------------------------------

Process finished with exit code 0
```

15:04 pm



```
4.298 10.997
3.953 13.497
4.334 14.050
3.868 14.656
4.330 16.203
4.835 16.579
12.387 17.386
10.718 19.029
13.411 19.426
14.472 21.770
17.956 23.201
11.373 23.604
23.281 25.305
23.307 26.956
27.334 33.259
29.459 36.247
35.070 39.295
38.837 43.424
37.767 44.792
44.354 46.485
42.176 49.372
46.092 49.558
48.036 52.651
50.880 53.114
56.202 64.419
66.859 69.440
67.841 70.589
77.025 79.116
```

```
8002.165233063269 []
DELAYEDOFF | DELAYEDOFF | BUSY |

4.512
[['0.638', '6.953'], ['0.467', '7.363'], ['0.910', '8.348'], ['3.867', '9.815'], ['6.252', '10.735'], ['4.921',
--------------------------------------------------------------------------------

Process finished with exit code 0
```

15:06 pm

It is shown the same departure information could be generated for each run of the code. Thus, the random-mode simulation process is reproducible.

Determining a suitable value of Tc

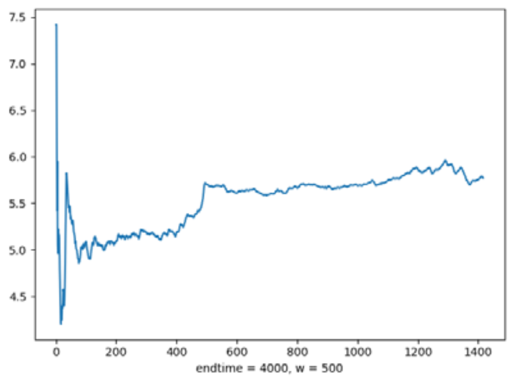
Baseline system uses **Tc = 0.1**, with poor response time because the servers are powered up again frequently (seed = 1):

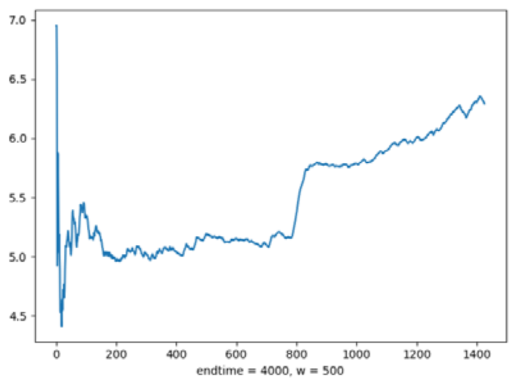
Para.txt	Mrt.txt
5	5.344
5	
0.1	
8000	

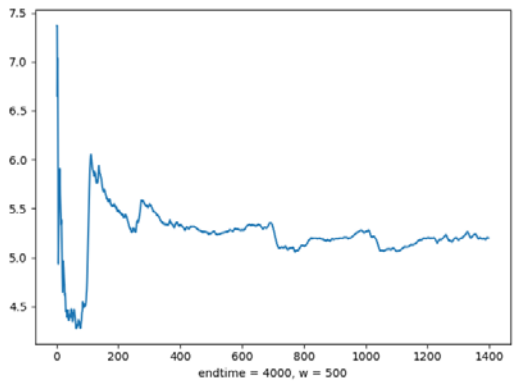
Goal: **Improved system** uses **higher Tc** to reduce response time, specifically, be **2 units less** than that of the baseline system.

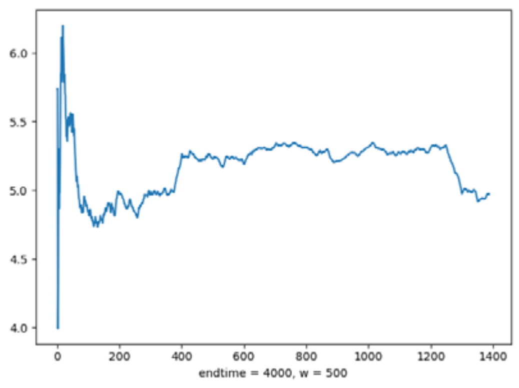
- **Analysis of Transient behavior versus steady state behavior of the baseline system** (5 independent replications with different seed):
  Graphs generated by a modified python version of week07_q1_a.m

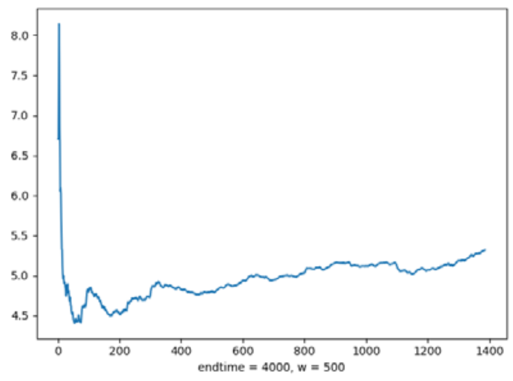
1. <u>W= 500, end time = 4000</u>







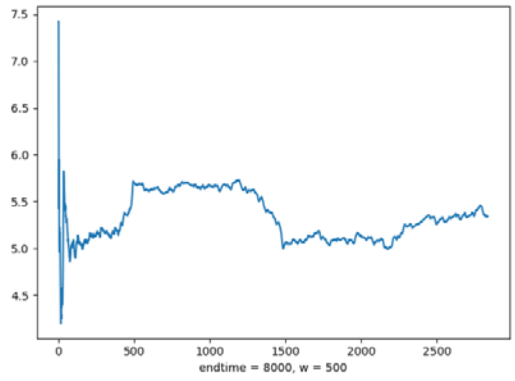


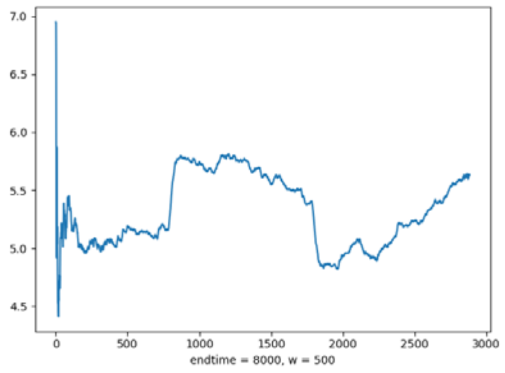


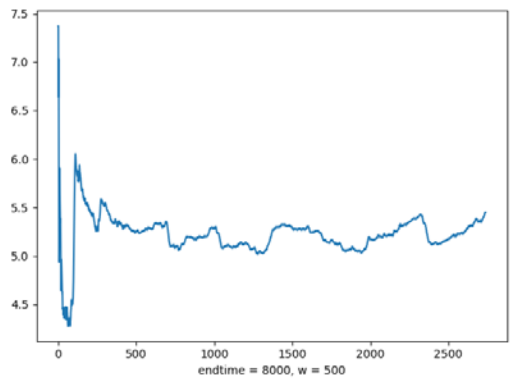
For choosing length of simulation and number of replications, no strict rules could be followed. They are chosen by doing some trial and error. The first guess of end time is 4000 with ‘w’ as 500.

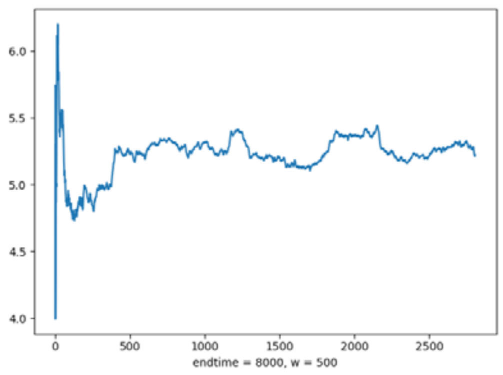
As shown above, some cases oscillate dramatically such as the second the picture, which is not ideal since end time might be shorter than the transient. To help find the steady state, the end time is doubled to 8000 in next phase.

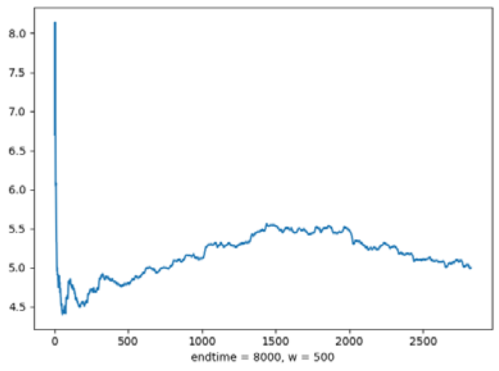
2. <u>W = 500, end time = 8000</u>





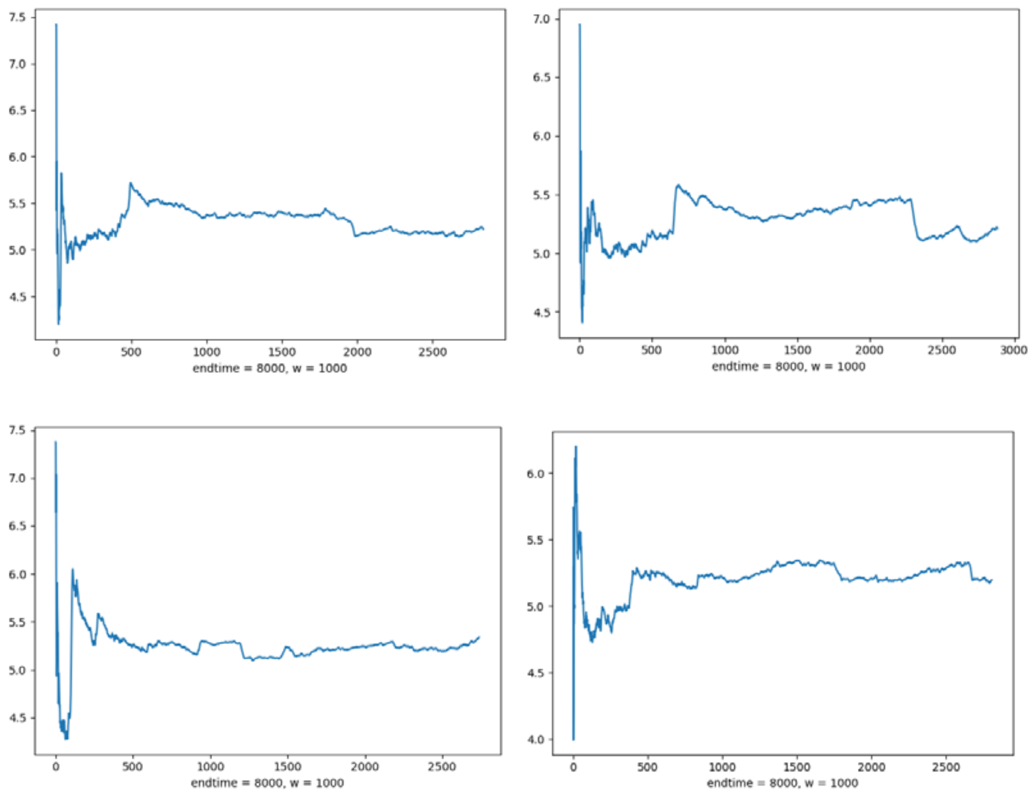


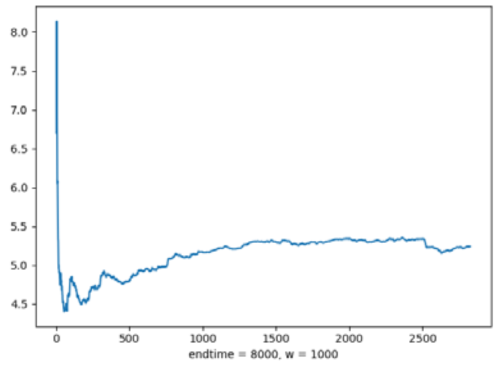




Since a safer end time is chosen, it is faintly reflected that the transient is between 500 and 1500. However, case 2 still oscillates intensely. More effects are needed to capture the real transient. Maybe the change of the value of ‘w’ could be helpful for smoothing the plot, thus it is doubled to 1000 in next phase.

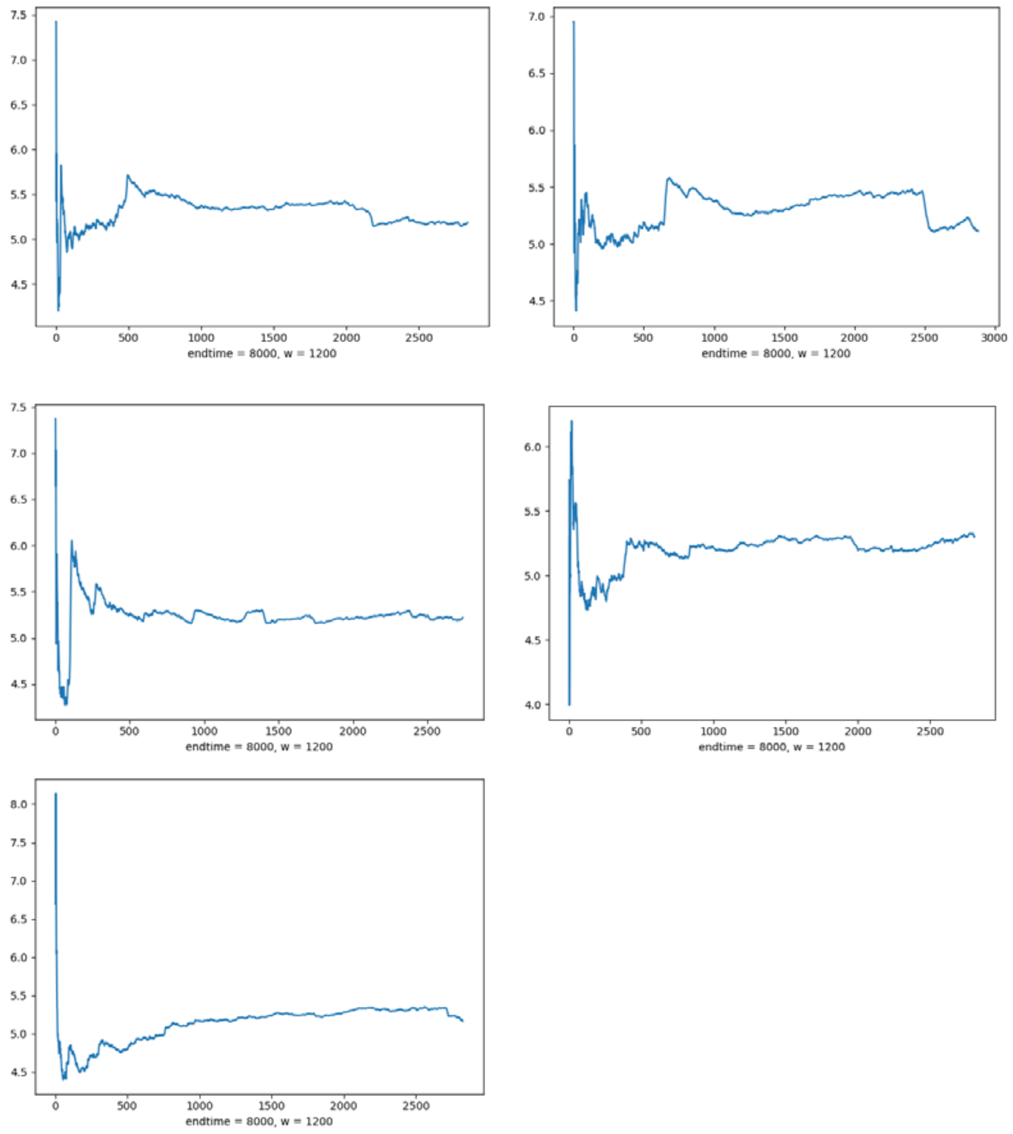
3. W = 1000, end time = 8000





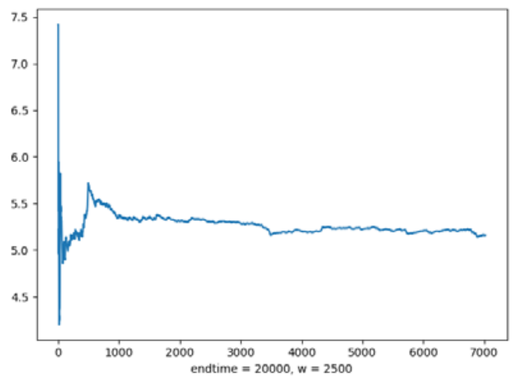
All plots are much smoother when the value of 'w' is 1000. Then, by visual inspection, steady state might be after nearly 600 jobs in this condition.

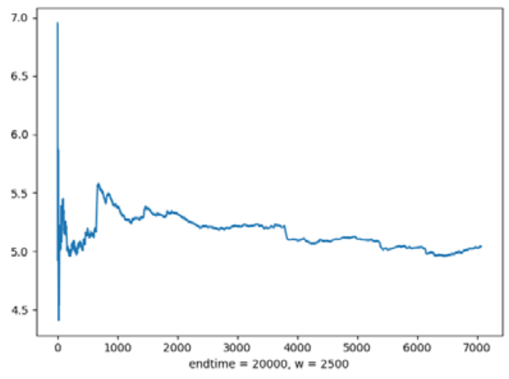
4. <u>W = 1200, end time = 8000</u>

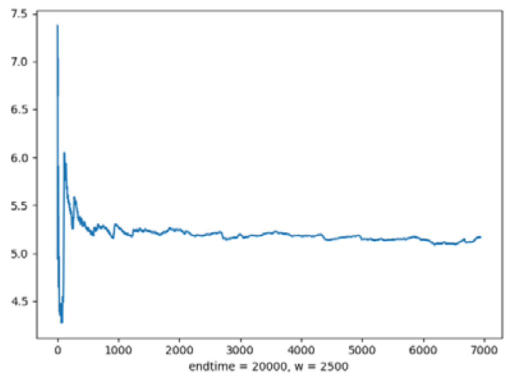


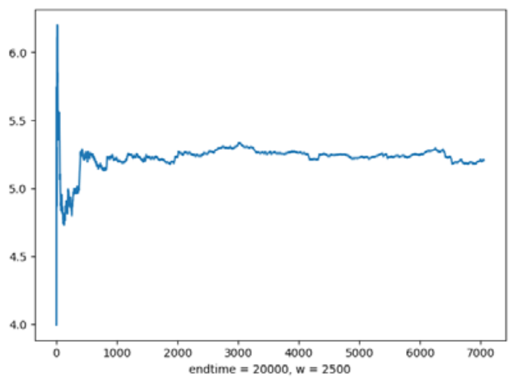
Increasing the ‘w’ slightly to 1200, the plots are like last phase. To evident the conclusion so far, the climbing of the value of those parameters are required in next phase.

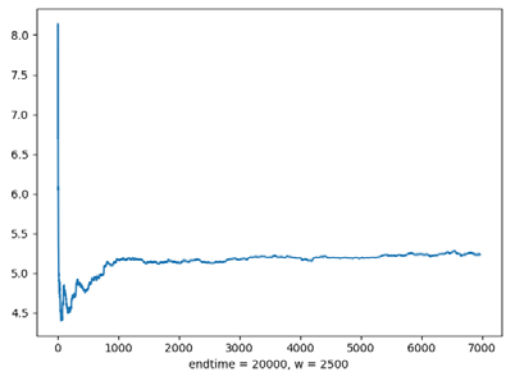
5. <u>W = 2500, end time = 20000</u>











To prove the conclusion that the transient is nearly 600 jobs, the end time is increased to 20000 and the value of ‘w’ increased to 2500.

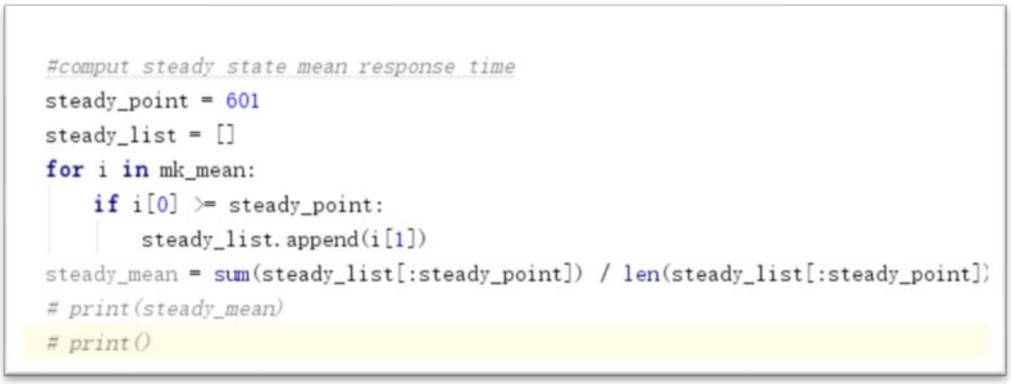
After the dramatical boost of the value of time end and ‘w’, the plots stay smooth and are consistent with our conclusion that **end of transient** is nearly **600 jobs**. Also, it tells that **end time as 8000** is enough for simulating because the result is not too different when it equals to 20000.

- **Choose of a suitable value of Tc**

Firstly, the **steady state mean response time** of **baseline system** are calculated by dropping the first 600 jobs:

Replications	*Mean response time*	*Steady state mean response time*
Seed = 1	5.344	5.828507487520803
Seed = 2	5.331	5.252321131447595
Seed = 3	5.246	4.981790349417643
Seed = 4	5.217	5.515306156405986
Seed = 5	5.151	5.0593128119800355

Code for Steady state mean response time:



```
#comput steady state mean response time
steady_point = 601
steady_list = []
for i in mk_mean:
    if i[0] >= steady_point:
        steady_list.append(i[1])
steady_mean = sum(steady_list[:steady_point]) / len(steady_list[:steady_point])
# print(steady_mean)
# print()
```

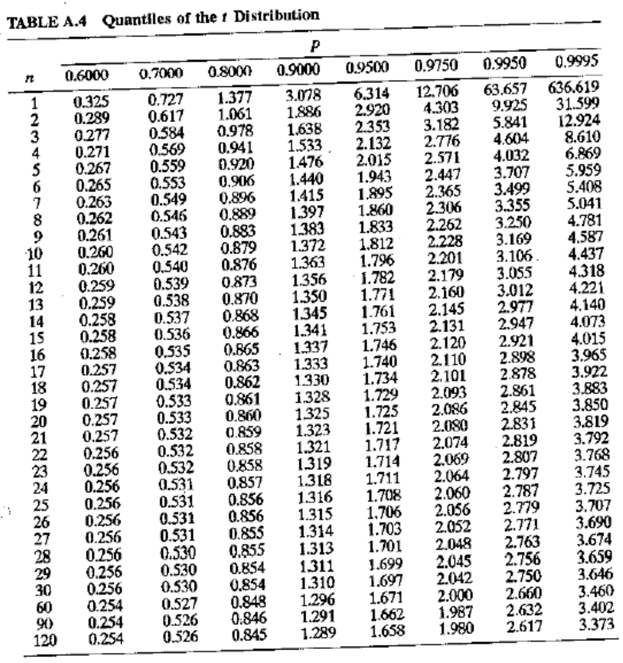
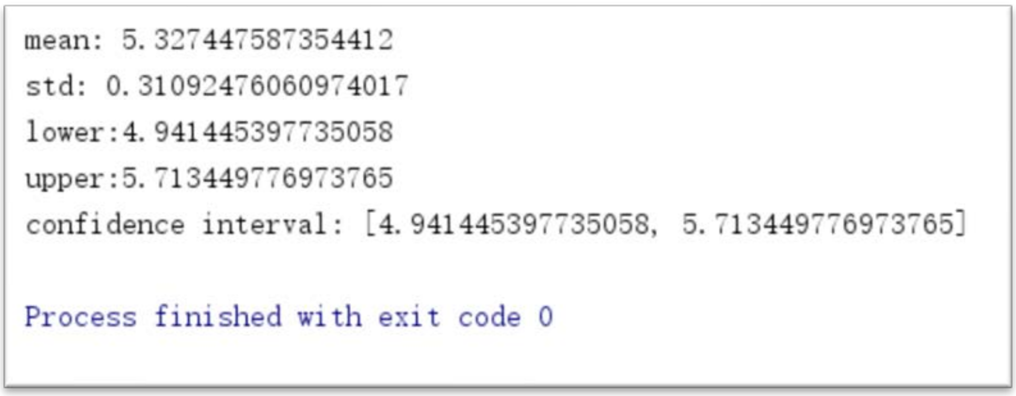


TABLE A.4 Quantiles of the *t* Distribution

	p							
n	0.6000	0.7000	0.8000	0.9000	0.9500	0.9750	0.9950	0.9995
1	0.325	0.727	1.377	3.078	6.314	12.706	63.657	636.619
2	0.289	0.617	1.061	1.886	2.920	4.303	9.925	31.599
3	0.277	0.584	0.978	1.638	2.353	3.182	5.841	12.924
4	0.271	0.569	0.941	1.533	2.132	2.776	4.604	8.610
5	0.267	0.559	0.920	1.476	2.015	2.571	4.032	6.869
6	0.265	0.553	0.906	1.440	1.943	2.447	3.707	5.959
7	0.263	0.549	0.896	1.415	1.895	2.365	3.499	5.408
8	0.262	0.546	0.889	1.397	1.860	2.306	3.355	5.041
9	0.261	0.543	0.883	1.383	1.833	2.262	3.250	4.781
10	0.260	0.542	0.879	1.372	1.812	2.228	3.169	4.587
11	0.260	0.540	0.876	1.363	1.796	2.201	3.106	4.437
12	0.259	0.539	0.873	1.356	1.782	2.179	3.055	4.318
13	0.259	0.538	0.870	1.350	1.771	2.160	3.012	4.221
14	0.258	0.537	0.868	1.345	1.761	2.145	2.977	4.140
15	0.258	0.536	0.866	1.341	1.753	2.131	2.947	4.073
16	0.258	0.535	0.865	1.337	1.746	2.120	2.921	4.015
17	0.257	0.534	0.863	1.333	1.740	2.110	2.898	3.965
18	0.257	0.534	0.862	1.330	1.734	2.101	2.878	3.922
19	0.257	0.533	0.861	1.328	1.729	2.093	2.861	3.883
20	0.257	0.533	0.860	1.325	1.725	2.086	2.845	3.850
21	0.257	0.532	0.859	1.323	1.721	2.080	2.831	3.819
22	0.256	0.532	0.858	1.321	1.717	2.074	2.819	3.792
23	0.256	0.532	0.858	1.319	1.714	2.069	2.807	3.768
24	0.256	0.531	0.857	1.318	1.711	2.064	2.797	3.745
25	0.256	0.531	0.856	1.316	1.708	2.060	2.787	3.725
26	0.256	0.531	0.856	1.315	1.706	2.056	2.779	3.707
27	0.256	0.531	0.855	1.314	1.703	2.052	2.771	3.690
28	0.256	0.530	0.855	1.313	1.701	2.048	2.763	3.674
29	0.256	0.530	0.854	1.311	1.699	2.045	2.756	3.659
30	0.256	0.530	0.854	1.310	1.697	2.042	2.750	3.646
60	0.254	0.527	0.848	1.296	1.671	2.000	2.660	3.460
90	0.254	0.526	0.846	1.291	1.662	1.987	2.632	3.402
120	0.254	0.526	0.845	1.289	1.658	1.980	2.617	3.373

Since there are 5 replications, to compute 95% confidence interval, t(n-1,1-alpha/2) = t(4,1-0.05/2) = **t(4,0.975) = 2.776**, then the confidence interval is given:



```
mean: 5.327447587354412
std: 0.31092476060974017
lower:4.941445397735058
upper:5.713449776973765
confidence interval: [4.941445397735058, 5.713449776973765]

Process finished with exit code 0
```

Calculated by support.py

After that, an improved system using higher Tc should be evaluated and compared with this baseline system. For a better performance, the response time should be 2 units less than that of the baseline system.

To choose a suitable value of Tc, some guesses are taken and then the comparation between new system and baseline system are done by statistically sound methods such as steady state mean response time comparison and Paired-t test.

<u>Guess 1: When Tc = 5</u>

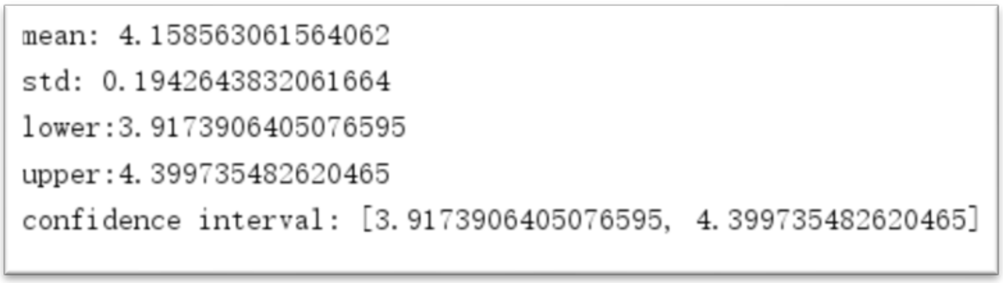
Replications	*Mean response time*	*Steady state mean response time*
Seed = 1	4.972	5.4686322795341145
Seed = 2	4.948	5.10555574043262
Seed = 3	4.924	4.77418302828619
Seed = 4	4.738	4.917113144758728
Seed = 5	4.823	4.830081530782034

It is clearly shown that the mean response time decreases slightly. However, it unsatisfied the desired goal. The Tc value should be much bigger in next guess.

<u>Guess 2: When Tc = 20</u>

Replications	*Mean response time*	*Steady state mean response time*
Seed = 1	3.980	4.492497504159735
Seed = 2	4.032	3.963129783693854
Seed = 3	3.974	4.016835274542437
Seed = 4	3.988	4.257374376039931
Seed = 5	3.967	4.062978369384358

After calculation of confidence interval and mean response time:

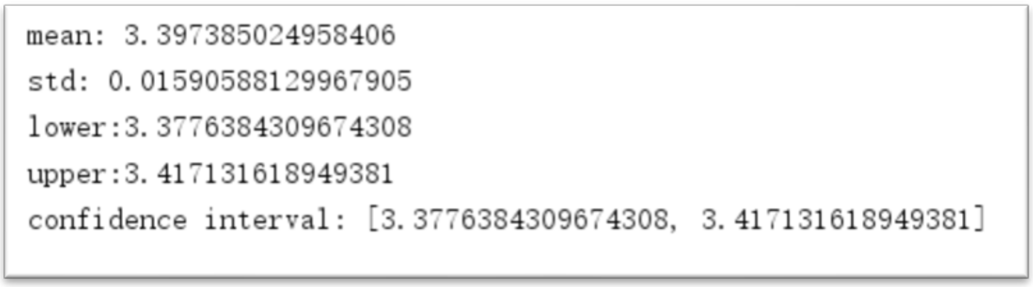


```
mean: 4.158563061564062
std: 0.1942643832061664
lower:3.9173906405076595
upper:4.399735482620465
confidence interval: [3.9173906405076595, 4.399735482620465]
```

Apparently, this value of Tc still fails to reach the goal of reducing 2 units of response time. The value should be heightened in next guess.

Guess 3: When Tc = 40

Replications	*Mean response time*	*Steady state mean response time*
Seed = 1	3.299	3.3892179700499185
Seed = 2	3.363	3.3723094841930252
Seed = 3	3.382	3.418089850249592
Seed = 4	3.275	3.4092312811980054
Seed = 5	3.344	3.3980765391014884

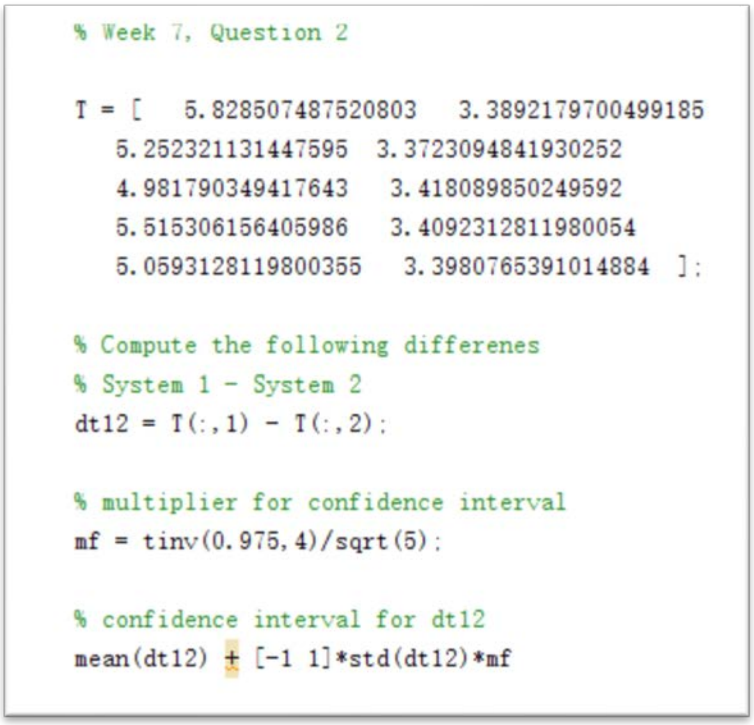


```
mean: 3.397385024958406
std: 0.01590588129967905
lower:3.3776384309674308
upper:3.417131618949381
confidence interval: [3.3776384309674308, 3.417131618949381]
```

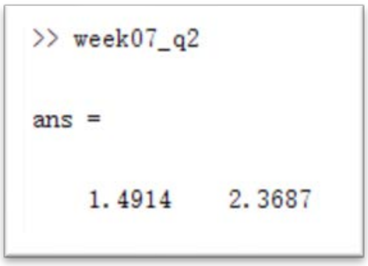
Here, the mean response time seems to satisfy the goal situation. For further proving, **a paired-t test** is given then:

	Baseline system	*Improved system With Tc = 40*	*Base sys. – Improved sys.*
Rep.1	5.828507487520803	3.3892179700499185	2.439289517
Rep.2	5.252321131447595	3.3723094841930252	1.880011647
Rep.3	4.981790349417643	3.418089850249592	1.563700499
Rep.4	5.515306156405986	3.4092312811980054	2.106074875
Rep.5	5.0593128119800355	3.3980765391014884	1.661236273

Calculated by week07_q2.m



```
% Week 7, Question 2

T = [   5.828507487520803   3.3892179700499185
   5.252321131447595  3.3723094841930252
   4.981790349417643   3.418089850249592
   5.515306156405986   3.409231281198054
   5.0593128119800355   3.3980765391014884  ];

% Compute the following differenes
% System 1 - System 2
dt12 = T(:,1) - T(:,2);

% multiplier for confidence interval
mf = tinv(0.975,4)/sqrt(5);

% confidence interval for dt12
mean(dt12) + [-1 1]*std(dt12)*mf
```



```
>> week07_q2

ans =

    1.4914    2.3687
```

After calculating by the mean response time of baseline system minus the response time of improved system, the 95% confidence interval for this difference is **[1.4914, 2.3687]**, which is approximately 2 units, thus it could be described that the improved system with Tc = 40 is better than the baseline system with Tc = 0.1.

However, the improved response time of improved system may oscillate around 2 with different given seed. Maybe a range of the suitable Tc is suggested. Then, try some trial and error to find the bound of the range of Tc.

Guess upper bound	*Confidence interval of the difference between baseline system and improved system (95%, 5 replications)*
Tc = 50	[1.5198, 2.5119]
Tc = 100	[1.7723, 2.6025]
Tc = 500	[1.8417, 2.6575]
Tc = 30	[1.1895, 2.2492]

As shown in the graph above, when the value of Tc set between 50 and 100, the response time confidence interval of the difference between improved and baseline system are all around 2, which means they are suitable. However, the dramatical delayed off time might cost too much power supply. On the other hand, the Confidence interval is [1.1895, 2.2492] when Tc = 30, which might possibly fail to reach the goal in many cases. To conclude, the suitable Tc value for this simulation system is around 40 in random mode, with end time equals to 8000, w equals to 1000 and the end of transient is 600 jobs.