COMP9334 Project Report

This report is written by Bingxin Tong (z5093617) to explain the codes of the project.

For simulation work:

There are two main files for simulation: simulation.py and wrapper.py.

All codes are written by using Python 3. You can run the simulation by command below:

python3 wrapper.py

All you need to do is to put these two files into a same folder with all configuration files, such as: num_tests.txt, mode_*.txt, para_*.txt, arrival_*.txt, service_*.txt.

For evaluation work:

There are also 4 files for evaluation work:

- arrival_distribution.m: verify the correctness of the interarrival probability distribution in mode random
- service_distribution.m: verify the correctness of the service time distribution in mode random
- end_time.py: each time we need to run wrapper.py first with different end time and then run end_time.py to get plot
- replication.py: each time we need to run wrapper.py with different seed first and then run replication.py to get mean response time which is removing the transient

1. Idea of Simulation Code

The main idea of this project:

- develop a simulation program for the setup/delayedoff system
- use statistically sound methods to analyse simulation outputs

First, we need to complete the simulation algorithm based on the instructions.

This computer system consists of a dispatcher with sufficient memory and m servers.

Mode of Program:

- Random: the inter-arrival probability distribution is exponentially distributed with λ. $s_k = s_{1k} + s_{2k} + s_{3k}$, where s_{1k}, s_{2k} and s_{3k} are exponentially distributed random numbers with μ.
- Trace: read the list of arrival times and list of service times from two ASCII files.

States of Server:

- OFF: the server is powered off and cannot process a job.
- SETUP: a server in the OFF state is powered on and cannot process a job.
- BUSY: the server is processing a job.

- DELAYEDOFF: during a countdown timer, the server is still powered on, after that without any job come in, turn to OFF, otherwise, turn to BUSY.

<u>Types of Events</u>:

- An Arrival event: a new job arrives at the system.
- A Departure event: a job departs from a server.
- A SETUP finished event: a server has finished its setup.
- A DELAYEDOFF expired event: the expiry of the countdown timer of a server in DELAYEDOFF state.

<u>Input Parameters</u>:

- mode: to control whether this program will run simulation using randomly generated arrival and service times, or in trace driven mode.
- arrival: supplying arrival information to the program.
- service: supplying service time information to the program.
- m: the number of servers.
- setup_time: setup time.
- delayedoff_time: the initial value of the countdown timer Tc.
- time_end: stops the simulation if the master clock exceeds this value, only relevant when mode is random, a positive floating point number.
- test_index: the index of the input test file, in order to name the output files.
- random_seed: the random seed of random function, only relevant when mode is random.

<u>Algorithm Structure</u>:

```
While master clock is smaller than final time:
    Find the next event
    Update master clock
    Distinguish the type of the next event
    If next event is Arrival:
        Do something
    If next event is Departure:
        Do something
    If next event is Setup_finished:
        Do something
    If next event is Delayedoff_expired:
        Do something
    Record response time and departure time
```

2. Correctness of Simulation Code

(a) Verify the correctness of the interarrival probability distribution and service time distribution

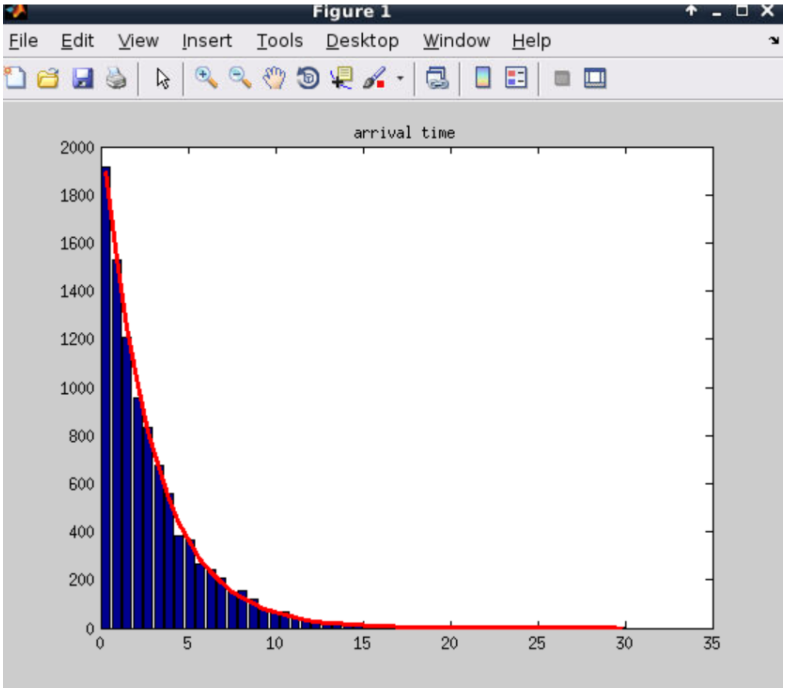
In terms of mode random:

Arrival Time:

- Each time, I generate a random number which is uniformly distributed in (0,1).
- $arrival\_time = -\log(1.0 - random\_number)/\lambda$
- Thus, the arrival time will be exponentially distributed with rate λ.

```
mean_arrival_rate = arrival
next_arrival_time = -log(1.0-uniform(0,1))/mean_arrival_rate
```

- I plot the arrival time distribution in MATLAB when $\lambda = 0.35$.
- (*code is in arrival_distribution.m*)

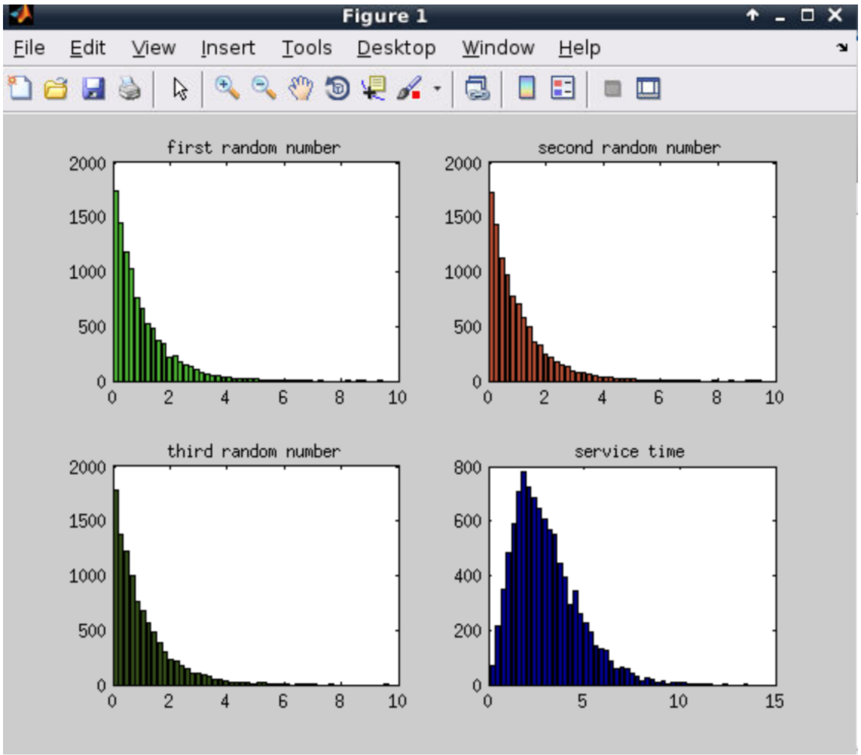


Service Time:

- Each time, I generate three random number which are uniformly distributed in (0,1).
- $service\_time = -\log(1.0 - random\_number\_1)/\mu - \log(1.0 - random\_number\_2)/\mu - \log(1.0 - random\_number\_3)/\mu$

```
mean_service_rate = service
s_1 = -log(1.0-uniform(0,1))/mean_service_rate
s_2 = -log(1.0-uniform(0,1))/mean_service_rate
s_3 = -log(1.0-uniform(0,1))/mean_service_rate
service_time_next_arrival = s_1 + s_2 + s_3
```

- I plot the service time distribution in MATLAB when $\mu = 1$.
- (*code is in service_distribution.m*)



(b) Verify the correctness of my simulation code.

In terms of <u>trace</u>:

Example 1:

In this example, there are $m = 3$ servers. The arrival and service times of the jobs are shown in Table 1. We assume all servers are in the OFF state at time zero. The setup time is assumed to be 50. The initial value of the countdown timer is $T_c = 100$. Table 2 shows the on-paper simulation with explanatory comments.

Arrival time	Service time
10	1
20	2
32 ~~30~~	3
33	4

Table 1: Example 1: Job arrival and service times.

My output trace of this example is shown in the screenshot below, same with the example results.

(In Servers, 0 represents OFF, 1 represents SETUP, 2 represents BUSY, 3 represents DELAYEDOFF)

```
bianca@ubuntu:~/Desktop/9334/trace_input$ python35 wrapper.py
Master clock      Dispatcher                            Servers
t=0               []                                    [0, 0, 0]
t=10.0            [10.0, 1.0, 'MARKED']                 [1, 0, 0]
t=20.0            [10.0, 1.0, 'MARKED']                 [1, 1, 0]
                  [20.0, 2.0, 'MARKED']
t=32.0            [10.0, 1.0, 'MARKED']                 [1, 1, 1]
                  [20.0, 2.0, 'MARKED']
                  [32.0, 3.0, 'MARKED']
t=33.0            [10.0, 1.0, 'MARKED']                 [1, 1, 1]
                  [20.0, 2.0, 'MARKED']
                  [32.0, 3.0, 'MARKED']
                  [33.0, 4.0, 'UNMARKED']
t=60.0            [20.0, 2.0, 'MARKED']                 [2, 1, 1]
                  [32.0, 3.0, 'MARKED']
                  [33.0, 4.0, 'UNMARKED']
t=61.0            [32.0, 3.0, 'MARKED']                 [2, 1, 1]
                  [33.0, 4.0, 'MARKED']
t=63.0            [33.0, 4.0, 'MARKED']                 [2, 1, 0]
t=66.0            []                                    [2, 0, 0]
t=70.0            []                                    [3, 0, 0]
```

Example 2:

In this example, there are $m = 3$ servers. In order to shorten the description, we will start from time 10 and the state of the system at this time is shown in Table 4. The arrival and service times of the job after time 10 are shown in Table 3. The setup time is assumed to be 5. The initial value of the countdown timer is $T_c = 10$. Table 4 shows the on-paper simulation with explanatory comments.

Arrival time	Service time
11	1
11.2	1.4
11.3	5
13	1

Table 3: Example 2: Job arrival and service times.

My output trace of this example is shown in the screenshot below, same with the example results.

(In Servers, 0 represents OFF, 1 represents SETUP, 2 represents BUSY, 3 represents DELAYEDOFF)

```
bianca@ubuntu:~/Desktop/9334/trace_input$ python35 wrapper.py
Master clock      Dispatcher                            Servers
t=10.0            []                                    [3, 3, 0]
t=11.0            []                                    [2, 3, 0]
t=11.2            []                                    [2, 2, 0]
t=11.3            [11.3, 5.0, 'MARKED']                 [2, 2, 1]
t=12.0            []                                    [2, 2, 0]
t=12.6            []                                    [2, 3, 0]
t=13.0            []                                    [2, 2, 0]
t=14.0            []                                    [2, 3, 0]
t=17.0            []                                    [3, 3, 0]
```

3. Reproducible of Results

In order to make my results reproducible, I added an input parameter “random_seed” for function simulation. If the seed is constant, then the results of random mode will be same.

```
### Initialising the seed number
seed(random_seed)
```

In this project, I submitted 3 sets of output files:

(1) $seed = 0, m = 3, setup\ time = 5, delayed\ off\ time = 0.1, time\ end = 10000, \lambda = 0.35, \mu = 1$

departure_1.txt (part of the whole file):

```
5.316	12.580
7.362	14.774
9.211	16.545
10.157	16.873
19.915	26.535
18.090	27.842
17.029	30.077
27.755	32.522
26.892	33.894
31.389	35.876
31.392	36.876
32.516	38.970
33.864	39.696
33.534	40.187
```

mrt_1.txt:

```
6.059
```

(2) $seed = 10, m = 3, setup\ time = 5, delayed\ off\ time = 0.1, time\ end = 10000, \lambda = 0.35, \mu = 1$

departure_1.txt (part of the whole file):

```
2.421	9.075
7.216	12.044
9.317	15.783
25.521	33.458
26.894	33.962
30.203	36.145
41.759	50.094
45.184	50.918
41.747	51.229
43.062	51.520
44.698	52.502
47.145	53.723
52.665	55.191
55.292	62.007
```

mrt_1.txt:

```
6.047
```

(3) $seed = 100, m = 3, setup\ time = 5, delayed\ off\ time = 0.1, time\ end = 10000, \lambda = 0.35, \mu = 1$

departure_1.txt (part of the whole file):

```
0.450   8.752
4.212   11.691
4.450   12.572
13.499    19.330
12.847    20.087
18.627    24.280
21.615    24.921
16.311    25.998
25.858    28.887
32.860    39.741
39.506    44.142
42.190    47.604
43.168    49.254
48.669    51.800
```

mrt_1.txt:

```
6.038
```

4. Solving Design Problem – determine a suitable value of Tc

According to the problem, the number of servers is 5, setup time is 5, $\lambda = 0.35$, $\mu = 1$.

We need to decide parameters, such as length of simulation, number of replications, transient removals.

Aim to get smaller mean response time by increasing Tc.

Baseline System: Tc=0.1

(a) Length of simulation and Transient removals

Length of simulation should be longer than the transient and have a good number of data point in the steady state part.

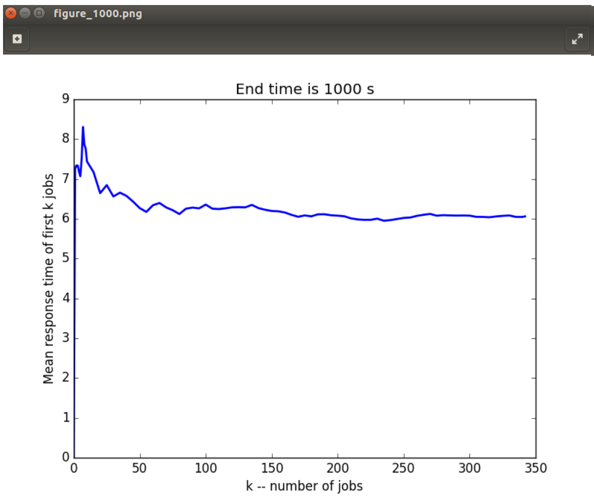
Thus, I tried different length of simulation to observe the transient part and steady part of this simulation. (seed = 0)

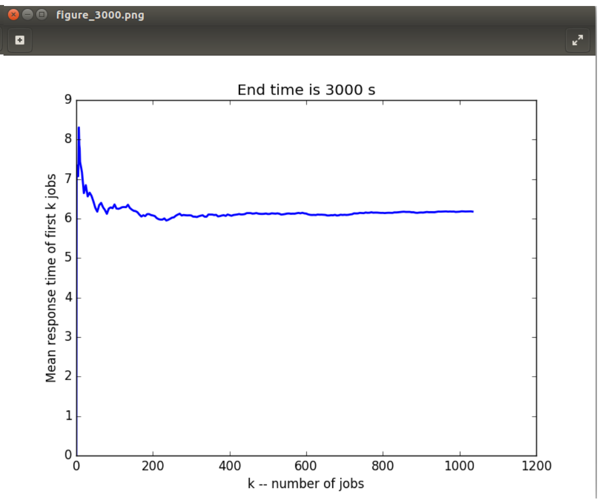
(*code is in end_time.py, this file using two extra output files from simulation.py, which are rf_*.txt and kf_*.txt.*

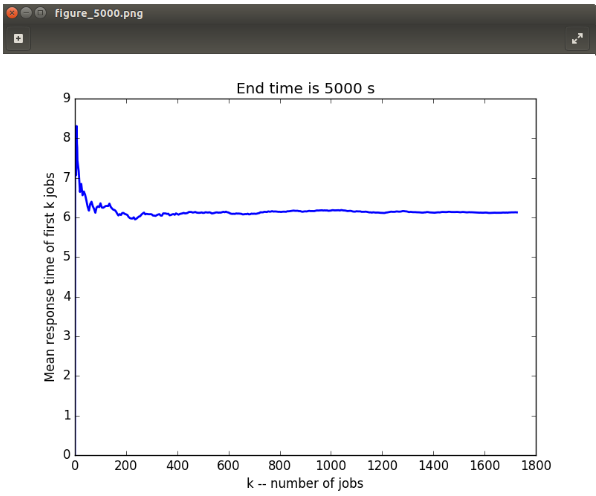
rf_*.txt records the mean response time of first k jobs and kf_*.txt records the number of jobs.

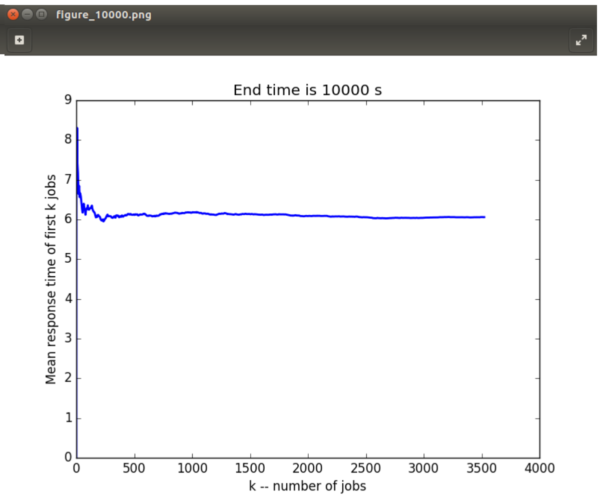
each time we need to run wrapper.py first and then run end_time.py to get plot.)

The result figures are shown below:









According to these figures, we can conclude that about first 400 jobs constitute the transient part.

In addition, in order to get a good number of data points in the steady part, end time is better to set up as 10000s, which has 3523-400=3123 jobs.

(b) Independent replications

I set 20 replications with different seeds: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29.

For each independent experiment, I record the response time of all the jobs and remove the transient part, then compute the mean response time using the steady state section.

(*code is in replication.py, each time we need to run wrapper.py with different seed first and then run replication.py to get mean response time which is removing the transient.*)

```
bianca@ubuntu:~/Desktop/9334/random_input$ python35 replication.py
seed = 0, mean_response_time = 6.055208069987525
```

```
bianca@ubuntu:~/Desktop/9334/random_input$ python35 replication.py
seed = 1, mean_response_time = 6.017601313086881
```

```
bianca@ubuntu:~/Desktop/9334/random_input$ python35 replication.py
seed = 2, mean_response_time = 6.062015154017947
```

```
bianca@ubuntu:~/Desktop/9334/random_input$ python35 replication.py
seed = 3, mean_response_time = 6.125782726332055
```

```
bianca@ubuntu:~/Desktop/9334/random_input$ python35 replication.py
seed = 4, mean_response_time = 6.0239622477247625
```

```
bianca@ubuntu:~/Desktop/9334/random_input$ python35 replication.py
seed = 5, mean_response_time = 6.087394164064148
```

```
bianca@ubuntu:~/Desktop/9334/random_input$ python35 replication.py
seed = 6, mean_response_time = 6.022380095741066
```

```
bianca@ubuntu:~/Desktop/9334/random_input$ python35 replication.py
seed = 7, mean_response_time = 6.070525630307876
```

```
bianca@ubuntu:~/Desktop/9334/random_input$ python35 replication.py
seed = 8, mean_response_time = 5.993788582954848
```

```
bianca@ubuntu:~/Desktop/9334/random_input$ python35 replication.py
seed = 9, mean_response_time = 6.0004440668005845
```

```
bianca@ubuntu:~/Desktop/9334/random_input$ python35 replication.py
seed = 20, mean_response_time = 6.057894983845244
```

```
bianca@ubuntu:~/Desktop/9334/random_input$ python35 replication.py
seed = 21, mean_response_time = 6.102278824202902
```

```
bianca@ubuntu:~/Desktop/9334/random_input$ python35 replication.py
seed = 22, mean_response_time = 6.148198629052257
```

```
bianca@ubuntu:~/Desktop/9334/random_input$ python35 replication.py
seed = 23, mean_response_time = 6.094695666637555
```

```
bianca@ubuntu:~/Desktop/9334/random_input$ python35 replication.py
seed = 24, mean_response_time = 5.913518143125085
```

```
bianca@ubuntu:~/Desktop/9334/random_input$ python35 replication.py
seed = 25, mean_response_time = 6.104013595879374
```

```
bianca@ubuntu:~/Desktop/9334/random_input$ python35 replication.py
seed = 26, mean_response_time = 6.044230217052884
```

```
bianca@ubuntu:~/Desktop/9334/random_input$ python35 replication.py
seed = 27, mean_response_time = 6.031232139677548
```

```
bianca@ubuntu:~/Desktop/9334/random_input$ python35 replication.py
seed = 28, mean_response_time = 6.0313609540897195
```

```
bianca@ubuntu:~/Desktop/9334/random_input$ python35 replication.py
seed = 29, mean_response_time = 6.125727931677117
```

Sample mean:

$$\mathrm{T} = \frac{\sum_{i=1}^{n} T(i)}{n} = \frac{121.112}{20} = 6.056$$

Sample standard deviation:

$$\mathrm{S} = \sqrt{\frac{\sum_{i=1}^{n}(T - T(i))^2}{n-1}} = 0.055$$

(c) Confident interval -- using statistically sound methods to analyse simulation results

After removing the transient, compute the confidence interval for this estimate.

I want to compute the 95% confidence interval, $\alpha = 0.05$.

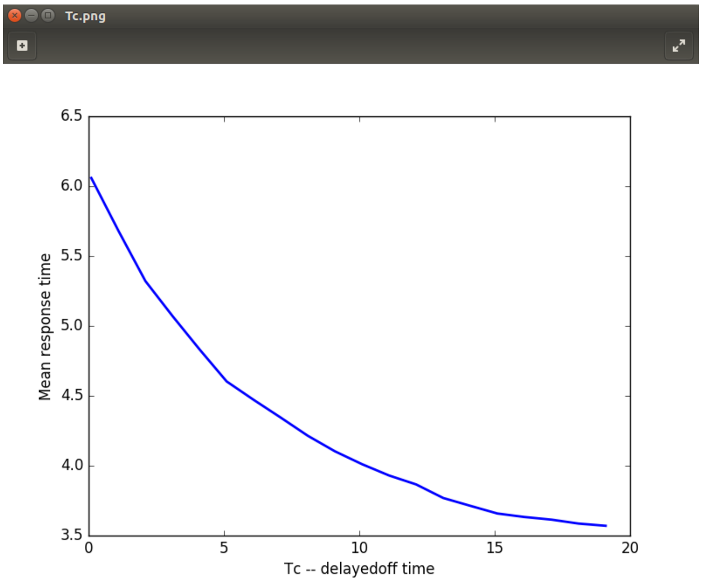
$$\left[\mathrm{T} - t_{n-1,1-\frac{\alpha}{2}}\frac{S}{\sqrt{n}}, \mathrm{T} + t_{n-1,1-\frac{\alpha}{2}}\frac{S}{\sqrt{n}}\right] = \left[6.056 - 2.093 \times \frac{0.055}{\sqrt{20}}, 6.056 + 2.093 \times \frac{0.055}{\sqrt{20}}\right]$$

$$95\% \text{ confidence interval} = [6.030, 6.082]$$

Choose Tc

The aim is to determine a value of Tc so that the improved system's response time must be 2 units less than that of the baseline system.

I plotted a figure to observe the change of response time by changing Tc.



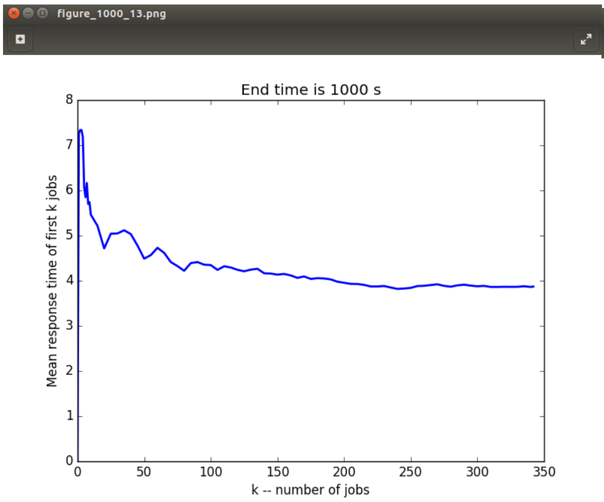
According to this figure, we can conclude that, Tc should be bigger than 11 to make the mean response time of improved system 2 units less than baseline system.

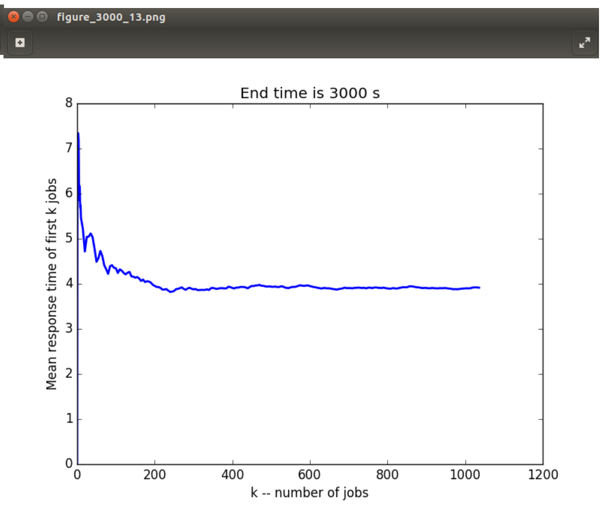
Thus, I choose Tc = 13.

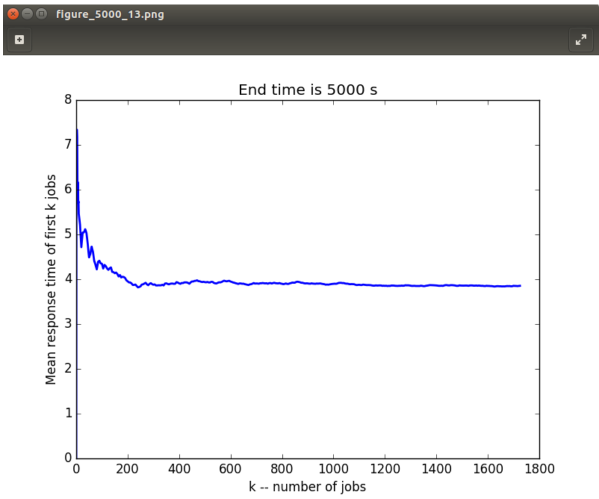
Improved System: Tc=13

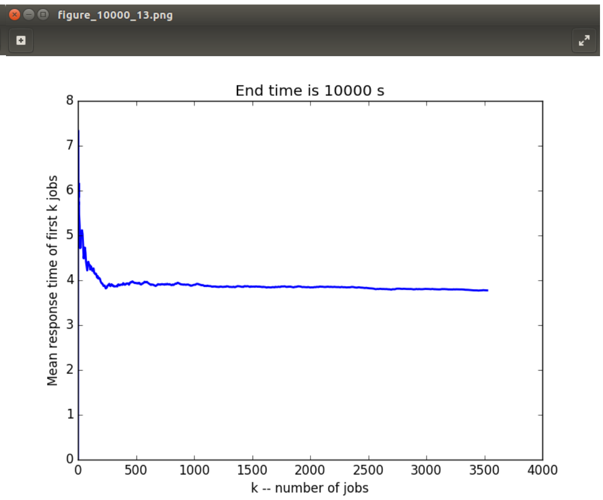
(a) Length of simulation and Transient removals (seed = 0)

The result figures are shown below:









According to these figures, we can conclude that about first 400 jobs constitute the transient part.

In addition, in order to get a good number of data points in the steady part, end time is better to set up as 10000s, which has 3525-400=3125 jobs.

(b) Independent replications

I set 20 replications with different seeds: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29.

```
bianca@ubuntu:~/Desktop/9334/random_input$ python35 replication.py
seed = 0, mean_response_time = 3.7573709358684813
```

```
bianca@ubuntu:~/Desktop/9334/random_input$ python35 replication.py
seed = 1, mean_response_time = 3.752187746955881
```

```
bianca@ubuntu:~/Desktop/9334/random_input$ python35 replication.py
seed = 2, mean_response_time = 3.8224562079461295
```

```
bianca@ubuntu:~/Desktop/9334/random_input$ python35 replication.py
seed = 3, mean_response_time = 3.8609541802489806
```

```
bianca@ubuntu:~/Desktop/9334/random_input$ python35 replication.py
seed = 4, mean_response_time = 3.8053731573420584
```

```
bianca@ubuntu:~/Desktop/9334/random_input$ python35 replication.py
seed = 5, mean_response_time = 3.9164195805867568
```

```
bianca@ubuntu:~/Desktop/9334/random_input$ python35 replication.py
seed = 6, mean_response_time = 3.8529686963929657
```

```
bianca@ubuntu:~/Desktop/9334/random_input$ python35 replication.py
seed = 7, mean_response_time = 3.841153303075565
```

```
bianca@ubuntu:~/Desktop/9334/random_input$ python35 replication.py
seed = 8, mean_response_time = 3.7202550931441802
```

```
bianca@ubuntu:~/Desktop/9334/random_input$ python35 replication.py
seed = 9, mean_response_time = 3.8196093138521348
```

```
bianca@ubuntu:~/Desktop/9334/random_input$ python35 replication.py
seed = 20, mean_response_time = 3.8508359530099625
```

```
bianca@ubuntu:~/Desktop/9334/random_input$ python35 replication.py
seed = 21, mean_response_time = 3.8132360987999157
```

```
bianca@ubuntu:~/Desktop/9334/random_input$ python35 replication.py
seed = 22, mean_response_time = 3.874267729592238
```

```
bianca@ubuntu:~/Desktop/9334/random_input$ python35 replication.py
seed = 23, mean_response_time = 3.8568259680086534
```

```
bianca@ubuntu:~/Desktop/9334/random_input$ python35 replication.py
seed = 24, mean_response_time = 3.7356855472276513
```

```
bianca@ubuntu:~/Desktop/9334/random_input$ python35 replication.py
seed = 25, mean_response_time = 3.888312124180831
```

```
bianca@ubuntu:~/Desktop/9334/random_input$ python35 replication.py
seed = 26, mean_response_time = 3.834905681498628
```

```
bianca@ubuntu:~/Desktop/9334/random_input$ python35 replication.py
seed = 27, mean_response_time = 3.8733937160245016
```

```
bianca@ubuntu:~/Desktop/9334/random_input$ python35 replication.py
seed = 28, mean response time = 3.844978285837163
```

```
bianca@ubuntu:~/Desktop/9334/random_input$ python35 replication.py
seed = 29, mean_response_time = 3.769776096633569
```

Sample mean:

$$\mathrm{T} = \frac{\sum_{i=1}^{n} T(i)}{n} = \frac{76.488}{20} = 3.824$$

Sample standard deviation:

$$\mathrm{S} = \sqrt{\frac{\sum_{i=1}^{n} (T - T(i))^2}{n-1}} = 0.053$$

(c) Confident interval -- using statistically sound methods to analyse simulation results

After removing the transient, compute the confidence interval for this estimate.

I want to compute the 95% confidence interval, $\alpha = 0.05$.

$$\left[\mathrm{T} - t_{n-1,1-\frac{\alpha}{2}} \frac{S}{\sqrt{n}}, \mathrm{T} + t_{n-1,1-\frac{\alpha}{2}} \frac{S}{\sqrt{n}}\right] = \left[3.824 - 2.093 \times \frac{0.053}{\sqrt{20}}, 3.824 + 2.093 \times \frac{0.053}{\sqrt{20}}\right]$$

$$95\% \text{ confidence interval} = [3.799, 3.849]$$

Compare Baseline system with Improved system

Replication	EMRT Baseline System	EMRT Improved System	EMRT Baseline System- EMRT Improved System
1	6.055	3.757	2.298
2	6.018	3.752	2.266
3	6.062	3.822	2.240
4	6.126	3.861	2.265
5	6.024	3.805	2.219

6	6.087	3.916	2.171
7	6.022	3.853	2.169
8	6.071	3.841	2.230
9	5.994	3.720	2.274
10	6.000	3.820	2.180
11	6.058	3.850	2.208
12	6.102	3.813	2.289
13	6.148	3.874	2.274
14	6.095	3.857	2.238
15	5.914	3.736	2.178
16	6.104	3.888	2.216
17	6.044	3.835	2.209
18	6.031	3.873	2.158
19	6.031	3.845	2.186
20	6.126	3.770	2.356

Sample mean:

$$\mathrm{T} = \frac{\sum_{i=1}^{n} T(i)}{n} = \frac{44.624}{20} = 2.231$$

Sample standard deviation:

$$\mathrm{S} = \sqrt{\frac{\sum_{i=1}^{n} (T - T(i))^2}{n-1}} = 0.052$$

95% Confidence interval of EMRT Baseline System- EMRT Improved System:

$$\left[\mathrm{T} - t_{n-1,1-\frac{\alpha}{2}} \frac{S}{\sqrt{n}}, \mathrm{T} + t_{n-1,1-\frac{\alpha}{2}} \frac{S}{\sqrt{n}}\right] = \left[2.231 - 2.093 \times \frac{0.052}{\sqrt{20}}, 2.231 + 2.093 \times \frac{0.052}{\sqrt{20}}\right]$$

$$95\% \text{ confidence interval} = [2.207, 2.255]$$

Thus, we can conclude that there is 95% probability that the improved system is better than baseline system.

In addition, there is 95% probability that the mean response time of improved system is 2 units less than baseline system at least.