

COMP9334 Project, Term 1, 2024:

Computing clusters

1 Simulation reproducible

To ensure the reproducibility of my simulation experiment, I used a fixed random number seed method. Because Python calculates pseudo-random numbers through seeds, the same seeds can result in the same sequence of random numbers, which ensures that my experimental data is repeatable. I wrote a program as shown in Figure 1.1 to control it.

```
config.py > generate_decimal  
1 random_seed = 9334  
  
random.seed(cfg.random_seed)
```

Figure 1.1

I use test case 4 given by lecture ($p_0 = 0.7, \alpha_0 = 1.2, \beta_0 = 3.6, \eta_0 = 2.1, \alpha_1 = 2.8, \eta_1 = 4.1, \lambda = 0.9, \alpha_{2l} = 0.91, \alpha_{2u} = 1.27, T_{limit} = 3.1, time_end = 200$), seed=9334, run 100 times and compare the result in mrt_4.txt.

As the Figure 1.2 show below, the mean response times are identical over 100 simulations. Hence, with the same seed, my simulation program will generate same random number sequence, and provide the same output.

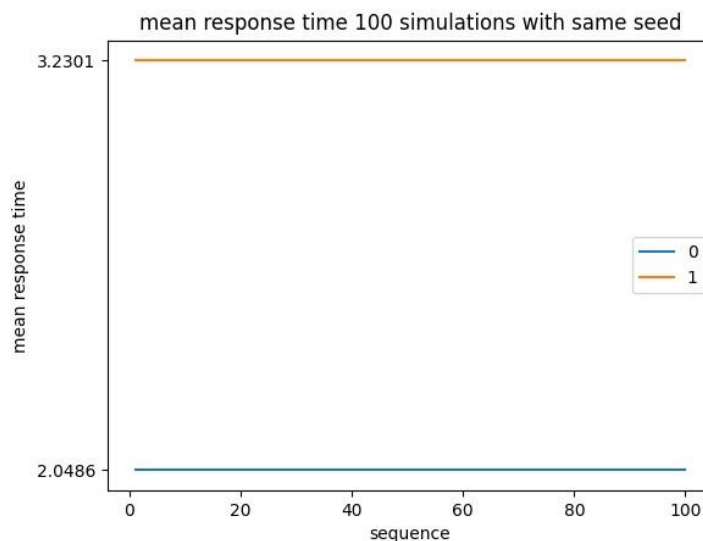


Figure 1.2

2 Probability distribution

In all random generated, I write the code that is “import random” to import the library in Python3.

2.1 The probability distribution of inter-arrival time

At first, we should make sure the arrival time is correct. Each inter-arrival time is the product of two random numbers a_{1k} and a_{2k} , i.e $a_k = a_{1k}a_{2k} \forall k = 1, 2, \dots$. The sequence a_{1k} is exponentially distributed with a mean arrival rate λ requests/s. The sequence a_{2k} is uniformly distributed in the interval $[a_{2l}, a_{2u}]$.

As shown in the code in Figure 2.1, first generate a sequence $a1_sequence$ with an exponential distribution, and then generate a range within the α_{2l} and α_{2u} uniformly distributed sequence $a2_sequence$. Finally, multiply the two sequences to obtain inter-arrival time sequence.

```
def generate_interarrival_time(self, size):  
    a1_sequence = [random.expovariate(self.lambd) for _ in range(size)]  
    a2_sequence = [random.uniform(self.alpha2l, self.alpha2u) for _ in range(size)]  
    return [a1 * a2 for a1, a2 in zip(a1_sequence, a2_sequence)]
```

Figure 2.1

The Figure 2.2 below can illustrate the inter-arrival time generated by the program:

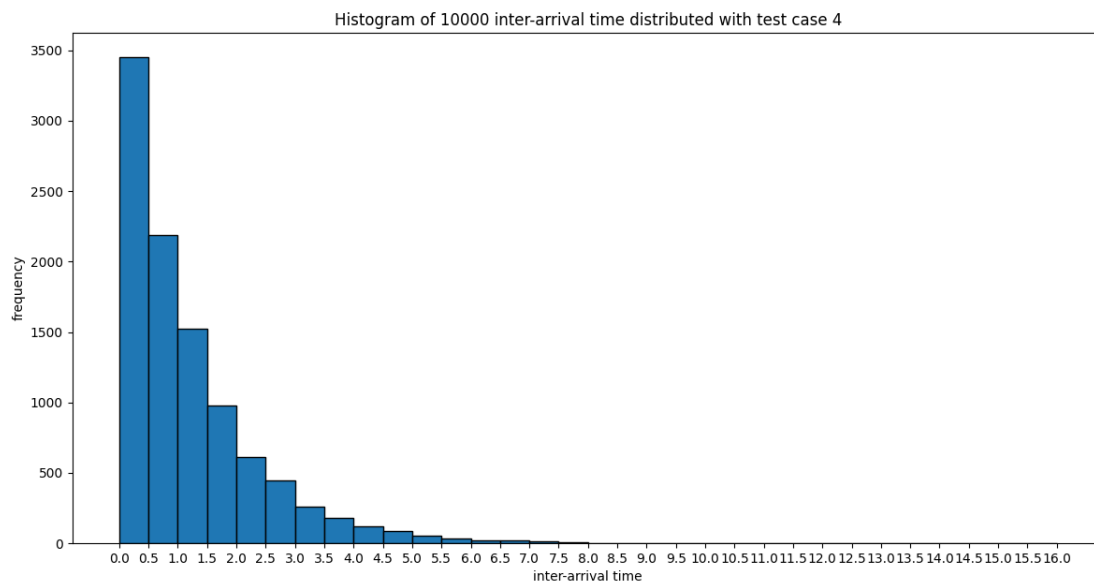
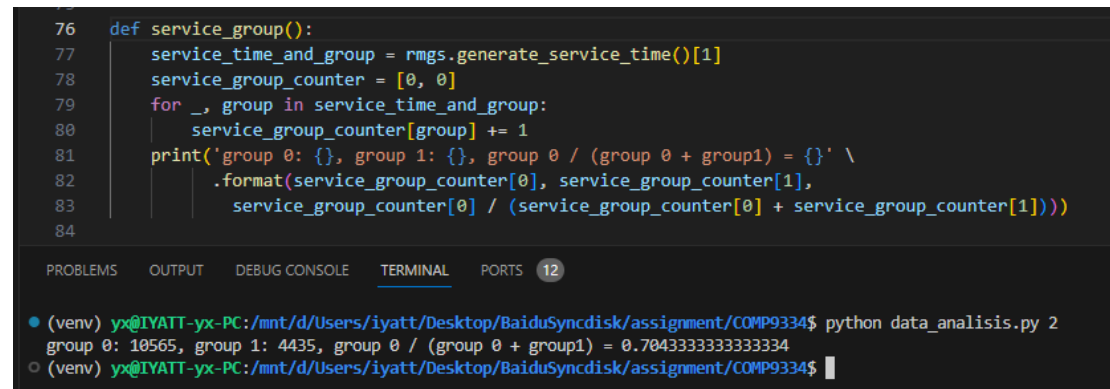


Figure 2.2

2.2 The probability of service group

The parameter p_0 specifies the generation probability of group 0, so the generation probability of group 1 is $(1-p_0)$. Here, the p_0 parameter in test case 4 is used for testing, with a total of 15000 generation attempts. Group 0 was generated 1056 times, while group 1 was

generated 4435 times. The proportion of group 0 to the total generation attempts is approximately 0.704333, which meets the 0.7 specified in test case 4.



```

76 def service_group():
77     service_time_and_group = rmgs.generate_service_time()[1]
78     service_group_counter = [0, 0]
79     for _, group in service_time_and_group:
80         service_group_counter[group] += 1
81     print('group 0: {}, group 1: {}, group 0 / (group 0 + group1) = {}' \
82           .format(service_group_counter[0], service_group_counter[1],
83                 service_group_counter[0] / (service_group_counter[0] + service_group_counter[1])))
84

```

PROBLEMS OUTPUT DEBUG CONSOLE **TERMINAL** PORTS 12

```

• (venv) yx@IYATT-yx-PC:/mnt/d/Users/iyatt/Desktop/BaiduSyncdisk/assignment/COMP9334$ python data_analysis.py 2
group 0: 10565, group 1: 4435, group 0 / (group 0 + group1) = 0.7043333333333334
○ (venv) yx@IYATT-yx-PC:/mnt/d/Users/iyatt/Desktop/BaiduSyncdisk/assignment/COMP9334$ █

```

Figure 2.3

2.3 The probability distribution of service time

If a job is indicated to go to a Group 0 server, its service time has the probability density function (PDF) $g_0(t)$:

$$g_0(t) = \begin{cases} 0 & \text{for } 0 \leq t \leq \alpha_0 \\ \frac{\eta_0}{\gamma_0 t^{\eta_0+1}} & \text{for } \alpha_0 < t < \beta_0 \\ 0 & \text{for } t \geq \beta_0 \end{cases}$$

where

$$\gamma_0 = \alpha_0^{-\eta_0} - \beta_0^{-\eta_0}$$

If a job is indicated to go to a Group 1 server, its service time has PDF:

$$g_1(t) = \begin{cases} 0 & \text{for } 0 \leq t \leq \alpha_1 \\ \frac{\eta_1}{\gamma_1 t^{\eta_1+1}} & \text{for } \alpha_1 < t \end{cases}$$

where

$$\gamma_1 = \alpha_1^{-\eta_1}$$

I used the Acceptance-Rejection Method to generate service time that conforms to the probability density function. For group 1, generate a uniformly distributed random number t from α_0 to β_0 , and then generate a uniformly distributed random number g . Substituting the value of t into the probability density function to calculate the value is greater than the value of g , then the value of t is The service time value conforms to the probability density function. If the conditions are not met, re-execute the above steps until the conditions are met. The value of the probability density function of group 1 is an open interval, so it is different. The only difference is that the exponential distribution is used to generate t . The corresponding Python3 code implementation is shown in Figure 2.4,

```

def generate_service_time(self):
    interarrival_time_list = self.generate_interarrival_time(3 * self.time_end)

    gamma0 = self.alpha0**(-self.eta0) - self.beta0*(-self.eta0)
    gamma1 = self.alpha1**(-self.eta1)

    service_time_and_group_list = list()

    g0 = lambda x: self.eta0 / (gamma0 * x**(self.eta0 + 1))
    g1 = lambda x: self.eta1 / (gamma1 * x**(self.eta1 + 1))

    for _ in interarrival_time_list:
        service_group = random.choices(self.choice, self.weights)[0]

        if service_group == 0:
            while True:
                random_t0 = (self.beta0 - self.alpha0) * random.uniform(0, 1) + self.alpha0
                random_g0 = random.uniform(0, 1)
                if random_g0 <= g0(random_t0):
                    service_time_and_group_list.append((random_t0, service_group))
                    break
        elif service_group == 1:
            while True:
                random_t1 = self.alpha1 + random.expovariate(1)
                random_g1 = random.uniform(0, 1)
                if random_g1 <= g1(random_t1):
                    service_time_and_group_list.append((random_t1, service_group))
                    break
    return (interarrival_time_list, service_time_and_group_list)

```

Figure 2.4

The Figure 2.5 below show the sequence time generated by the program. Figures (a) and (c) show the distribution of service time for group 0 and group 1 generated based on probability density functions, while Figures (b) and (d) show the scatter plots of probability density functions g_0 and g_1 , respectively.

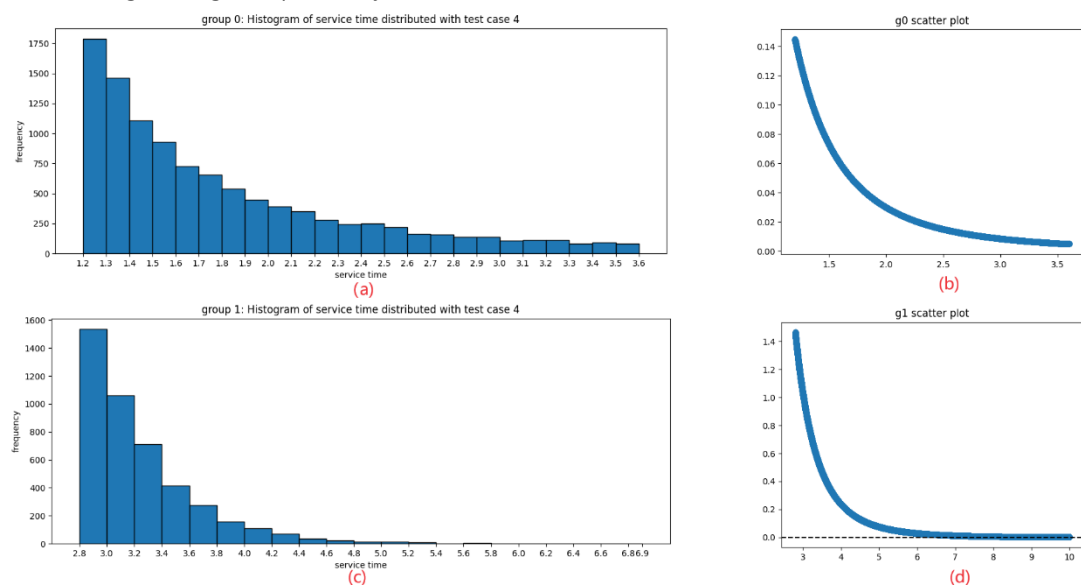


Figure 2.5

3 Verifying the correctness of the simulation program

To verify the correctness of my simulation program, I used the test cases provided by the lecturer. Running test_all.sh will test all cases, and then call cf_output_with_ref.py provided by the lecturer for verification.

In my testing environment, the Pandas version is 2.2.1, but a deprecated parameter is used in cf_output_with_ref.py. Running this script will print a warning. So I modified it by changing "delim_whitespace=True" at line 28 to "sep='\s+'" in Figure 3.1.

```
35 import pandas as pd
36
37 def read_dep_file(file_path):
38     output_df = pd.read_csv(file_path, header=None, sep='\s+',
39                             names=['arr_time', 'dep_time', 'ser_class'],
40                             dtype={'arr_time':float, 'dep_time':float, 'ser_class':str})
41
42     output_class = output_df['ser_class'].str.strip().values.astype(str)
43
44     output_times = output_df[['arr_time', 'dep_time']].values
45
46     return output_times, output_class
```

Figure 3.1

Figure 3.2 shows the test results for all cases, all of which meet the expected output given by the lecturer. Therefore, the simulation program can run normally.

```
[info] mode=trace, n=3, n0=1, T_limit=3.0
[info] mode=trace, n=4, n0=2, T_limit=3.5
[info] mode=trace, n=4, n0=1, T_limit=3.5
[info] mode=trace, n=5, n0=3, T_limit=3.4
[info] mode=random, n=5, n0=2, T_limit=3.1, time_end=200, lambda=0.9, alpha2l=0.91, alpha2u=1.27, p0=0.7, alpha0=1.2, beta0=3
.6, eta0=2.1, alpha1=2.8, eta1=4.1, seed=29
[info] mode=random, n=8, n0=5, T_limit=3.5, time_end=500, lambda=1.6, alpha2l=0.86, alpha2u=1.34, p0=0.82, alpha0=1.4, beta0=
4.1, eta0=3.4, alpha1=3.2, eta1=3.7, seed=29
[info] mode=random, n=10, n0=6, T_limit=3.2, time_end=1000, lambda=2.1, alpha2l=0.97, alpha2u=1.14, p0=0.89, alpha0=1.9, beta
0=4.5, eta0=2.6, alpha1=2.0, eta1=4.1, seed=29
Test 0: Mean response time matches the reference
Test 0: Departure times match the reference
Test 1: Mean response time matches the reference
Test 1: Departure times match the reference
Test 2: Mean response time matches the reference
Test 2: Departure times match the reference
Test 3: Mean response time matches the reference
Test 3: Departure times match the reference
Test 4: Mean response time is within tolerance
Test 4: Mean response time is within tolerance
Test 5: Mean response time is within tolerance
Test 5: Mean response time is within tolerance
Test 6: Mean response time is within tolerance
Test 6: Mean response time is within tolerance
```

Figure 3.2

4 Determine a suitable value of n_0

For this design problem, I will assume the following parameter values: $p_0 = 0.7$, $\alpha_0 = 1.2$, $\beta_0 = 3.6$, $\eta_0 = 2.1$, $\alpha_1 = 2.8$, $\eta_1 = 4.1$, $\lambda = 0.9$, $\alpha_{2l} = 0.91$, $\alpha_{2u} = 1.27$, $T_{limit} = 3.1$ (test case 4), $n = 10$. In this part, we will use sound statistical analysis on the simulation results obtained.

4.1 Running the simulation once and present the results

(1) Choose number of simulation and simulation time.

First, We choose `time_end = 15000` and `seed=9334` to run simulation function temporarily. Use the `draw_mrtofk` function in `data_analysis.py`, we can generate the figure 4.1 below,

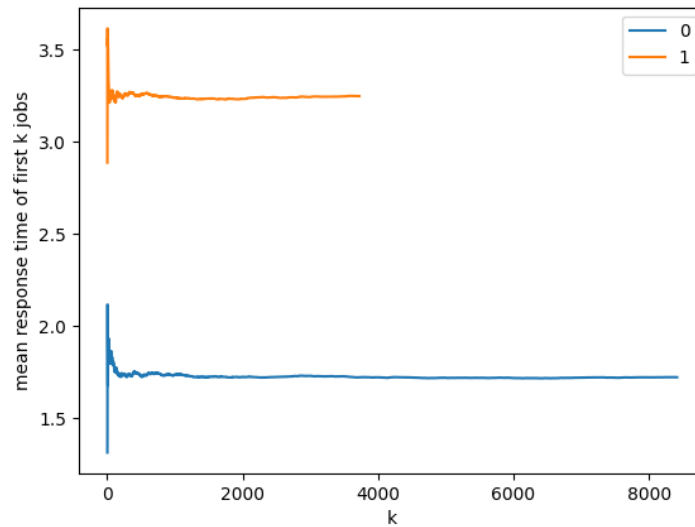


Figure 4.1

So now we see that the figure 4.1 shows response time of the first over 15000 jobs. Mean response time of first k jobs is

$$M(k) = \frac{X(1) + X(2) + \dots + X(k)}{k}$$

, among that $X(k)$ represent the response time of k-th job.

(2) Transient removal

Next, we know from generated mrt file that before transient removal, the mean response time of group 0 and group 1 is approximately 1.7211 and 3.2490. Then, analyzing the figure 4.1, distinguish transient part and steady part, as graph shows below,

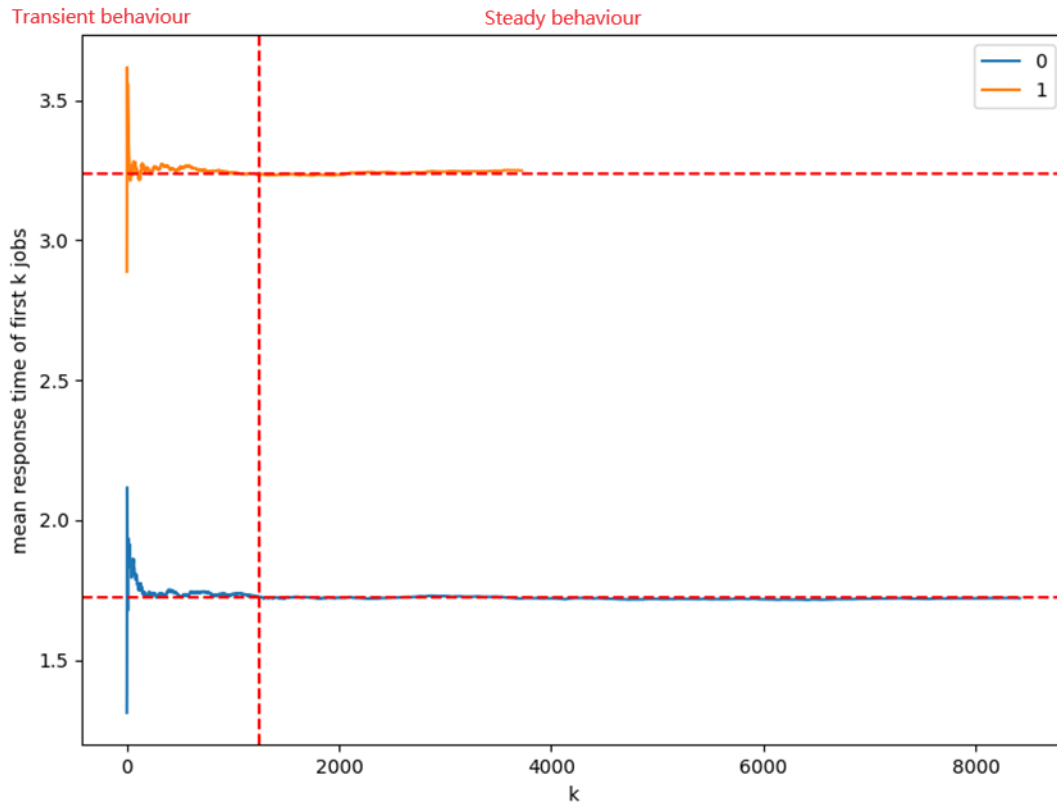


Figure 4.2

The early part of the simulation displays transient (= non-steady state behaviour). In contrast, the later part of the simulation converges or fluctuates around the steady state value. Since we are interested in the steady state value, we should not use the transient part of the data to compute the steady state value. We should remove the transient part and only use the steady state part to compute the mean response time. That means that we should only use data after $k=1250$ to calculate mean response time. Now, we should use the formula,

$$\frac{X(m+1) + X(m+2) + \dots + X(N)}{N - m}$$

among that $X(m)$ is response time in m -th job. According to this formula, the mean response time after transient part removal of the system in figure 4.2 is about 1.7203 and 3.2558 which is generated by `data_analysis.py` in attach files. In addition, now we can also know that `time_end = 15000` is big enough to enter the steady part in this situation. If the selected `time_end` does not show a steady part, then we should increase the simulation time (`time_end`).

4.2 Take different n_0 value

After being able to remove the transient part of the simulation, we can now proceed to determining a suitable value of n_0 .

We will now compare systems of different n_0 with the baseline, by calculating the confidence interval of the difference in estimated mean response time of the new system with the baseline. By conducting 30 different experiments for each system, we can ensure that the data obtained is fairly accurate.

In order to calculate the confidence interval, we need to use some formulas. The formula for

calculating the sample mean is as follows,

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

The code is as shown in the figure 4.3,

```
mean = lambda x: sum(x) / len(x)
```

Figure 4.3

The formula for sample standard deviation is as follows,

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

The code is as shown in the figure 4.4,

```
def sample_standard_deviation(x, mean_value):
    S = 0
    for i in range(len(x)):
        S += (mean_value - x[i])**2
    return math.sqrt(S / (len(x) - 1))
```

Figure 4.4

The mean response time with 95% probability is in the interval

$$\left[\hat{T} - t_{n-1, 1-\frac{\alpha}{2}} \frac{\hat{S}}{\sqrt{n}}, \hat{T} + t_{n-1, 1-\frac{\alpha}{2}} \frac{\hat{S}}{\sqrt{n}} \right]$$

The code is as shown in the figure 4.5,

```
def confidence_interval(ssd, mv):
    lower_value = cfg.decimal_float_str(mv - 2 * ssd / math.sqrt(sampling_frequency))
    upper_value = cfg.decimal_float_str(mv + 2 * ssd / math.sqrt(sampling_frequency))
```

Figure 4.5

The upper and lower limits in the results returned by the confidence_interval function will retain four decimal places and be presented as strings.

Since the response times of two groups, group 0 and group 1, exist in the experiment, the overall response time cannot be well estimated. The lecturer gave a formula for calculating weighted mean response time,

$$\omega_0 T_0 + \omega_1 T_1$$

where $T_i (i=0,1)$ is the average impact time of each of the two groups of servers. $\omega_i (i=0,1)$ is calculated from two parts. The first part is the proportion of the number of tasks in the current group to the total number of tasks in the two groups. The second part is the average service time of the current group. Then divide the first part by the second part to get The weight of this group. After each simulation experiment, ω_0 , ω_1 , T_0 and T_1 need to be calculated.

The parameter values are based on test case 4, and on this basis, time_end=10000 and n=10 are set. Test the values of n_0 from 0 to 9, with each n_0 value tested 30 times, and set the seed value to be tested sequentially from 0 to 29. Ensure sufficient and reproducible samples for the experiment.

The specific implementation of the Python 3 code is located in data_analysis.py. By executing "Python data_analysis.py 5", the simulation can be run and the data can be calculated after

the simulation is completed. The result is shown in Table 4.1.

n_0	Confidence Interval
1	(88.4888,88.9469)
2	(1.1695,1.0261)
3	(1.0257,1.0261)
4	(1.0044,1.0045)
5	(1.0008,1.0008)
6	(1.0007,1.0008)
7	(1.0041,1.0043)
8	(1.0287,1.0297)
9	(1.9432,1.9450)

Table 4.1

We can see that when $n_0=6$, the weighted mean response time achieves the minimum value. It can also be found that there is not much difference from when $n_0=5$.