EECE5640 – SIMULATION AND PERFORMANCE EVALUATION: HOMEWORK 3

ANNA DEVRIES[1]

8 March 2021

CONTENTS

LIST OF FIGURES

[1] *Department of Electrical and Computer Engineering, Northeastern University, Boston, United States*

LIST OF TABLES

1 QUESTION 1

Description: Ex. 3.1.1

- Modify program ssq2 to use Exponential (1.5) service times.
- Process a relatively large number of jobs, say 100000, and report what changes this produces relative to the statistics in Example 3.1.3.
- Explain (or conjecture) why some statistics change and others do not.

1.1 Solution

First, I modified line 64 in the double GetService(void) function. It was originally Uniform(1.0,2.0) and I changed it to Exponential(1.5). Figure 1 shows the results of the original ssq2 output and the new output.

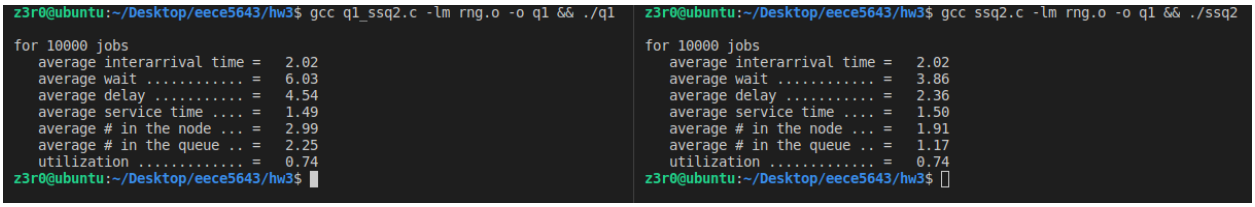

Figure 1: Question 1a modified output.

Next, I changed the number of jobs from 10000 to 100000. Figure 2 shows the results of the original ssq2 output and the new output. Comparing the statistics of Example 3.1.3 and the new output, the average interarrival time, average service time and utilization are the same; however, the other statistics are not. Average wait time increases from 3.83 to 6.04, average delay time increases from 2.33 to 4.53, average number of jobs in the node increases from 1.92 to 3.02, and average number of jobs in the queue increases from 1.17 to 2.27.

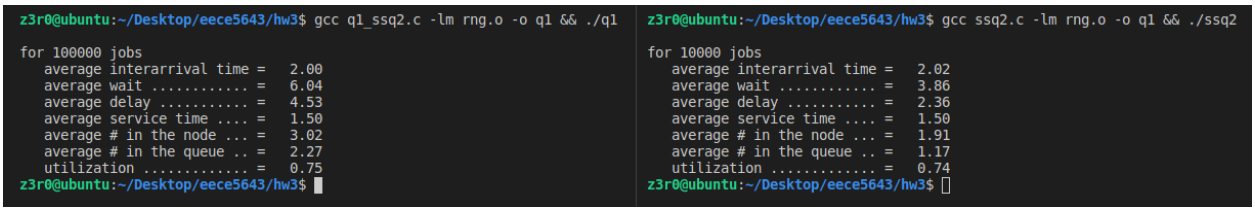

Figure 2: Question 1b modified output.

The changed service time distribution creates an exponential distribution rather than a fairly linear distribution created from Uniform(a,b). Uniform(a,b) gives an equal likely chance that all values between a and b will occur at random, whereas, Exponential(μ) is a nonlinear distribution that maps 0.0 -> 1.0 to 0.0 -> ∞. The μ value refers to the mean random sample selection. Due to the mean remaining at 1.5, the edited program still produces a 1.5 average service time and 0.75 utilization (with average interarrival time remaining the same as well since the arrival function was not changed at all). However, the other averages increased because of the new distribution to reflect the graph in Figure 3.

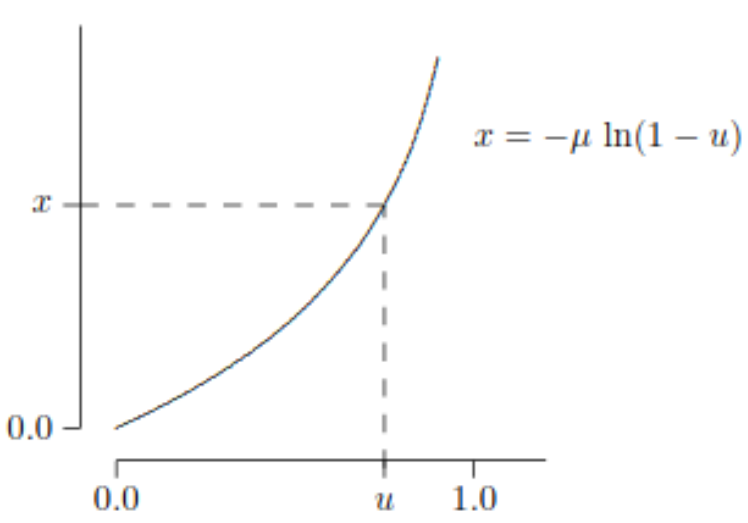

Figure 3: Exponential Distribution.

2 QUESTION 2

Description: Ex. 3.1.5.

- Verify that the mean service time in Example 3.1.4 is 1.5.
- Verify that the steady-state statistics in Example 3.1.4 seem to be correct.
- Note that the arrival rate, service rate, and utilization are the same as those in Example 3.1.3, yet all the other statistics are larger than those in Example 3.1.3. Explain (or conjecture) why this is so. Be specific.

2.1 Solution

2.1.1 A)

$$\text{Mean of Service tasks} = 1 + \text{Geometric}(0.9) = 1 + \frac{0.9}{1-0.9} = 10$$
$$\text{Mean time per task is Uniform}(0.1, 0.2) = \frac{0.1+0.2}{2} = 0.15$$
$$10 \text{ tasks} * 0.15 \text{ min/task} = 1.5 \text{ min aka mean service time}$$

2.1.2 B)

To verify this, I utilized ssq2 and edited the GetService function such that it reflected the example in 3.1.4. Figure 4 shows the new GetService function.

After editing the code's service function, I added logic to run the tests for three different seeds (123456789, -123456789, 12345) and pulled the average interarrival rate, wait, delay, service time, number of tasks in the node, number of tasks in the queue, and utilization at each task index number. I appended these values to a csv file and graphed the results to observe the steady-state rate and verify the steady-state statistics provided in Example 3.1.4. The graphs are shown in Figure 5 to Figure 11.

The final steady-state values are shown in Figure 12. This was gathered after running 100,000 jobs and closely matched the values provided in Example 3.1.4.

The Figure 5 to Figure 11 graphs plot the accumulated average for each characteristic and seed at a particular task number. This shows how the characteristic statistics change as the number of tasks in the

```
double GetService(void)
/* ------------------------------
 * generate the next service time
 * ------------------------------
 */
{
  long k;
  double sum = 0.0;
  long tasks = 1 + Geometric(0.9);
  for (k = 0; k < tasks; k++)
    sum += Uniform(0.1, 0.2);
  return (sum);
}
```

Figure 4: Geometric and Uniform Distribution in GetService.

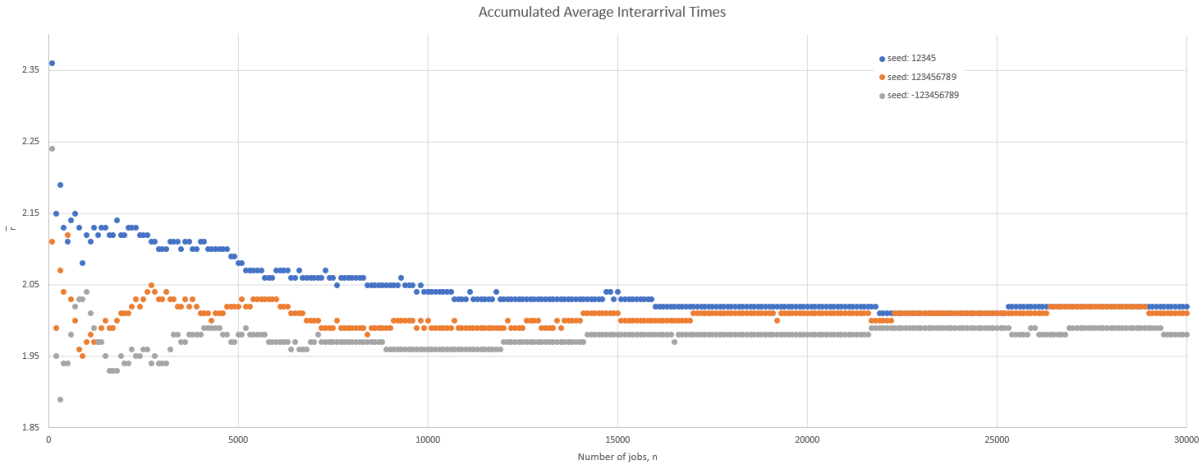

Figure 5: Accumulated Average Interarrival Time.

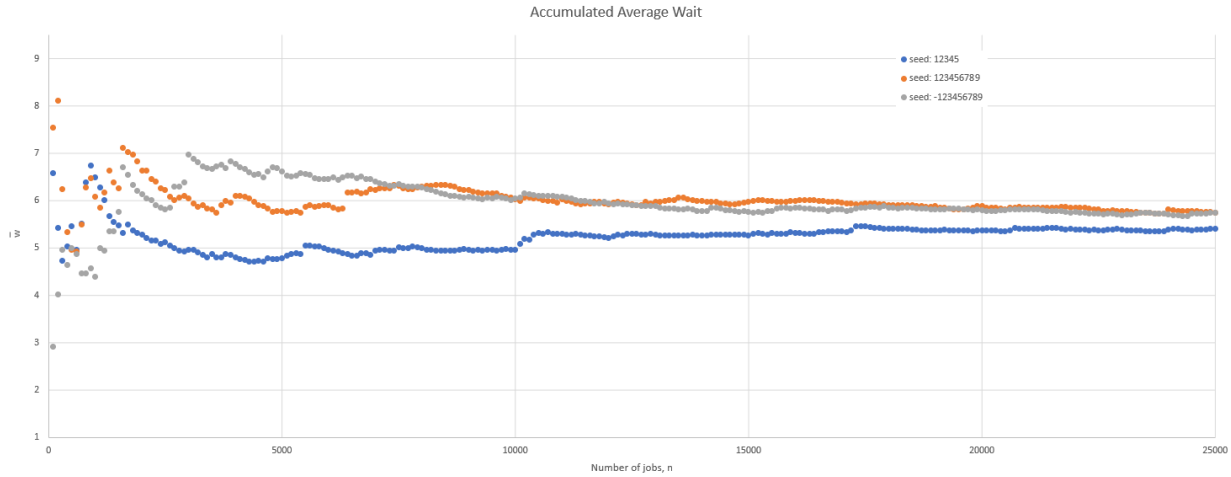

Figure 6: Accumulated Average Wait.

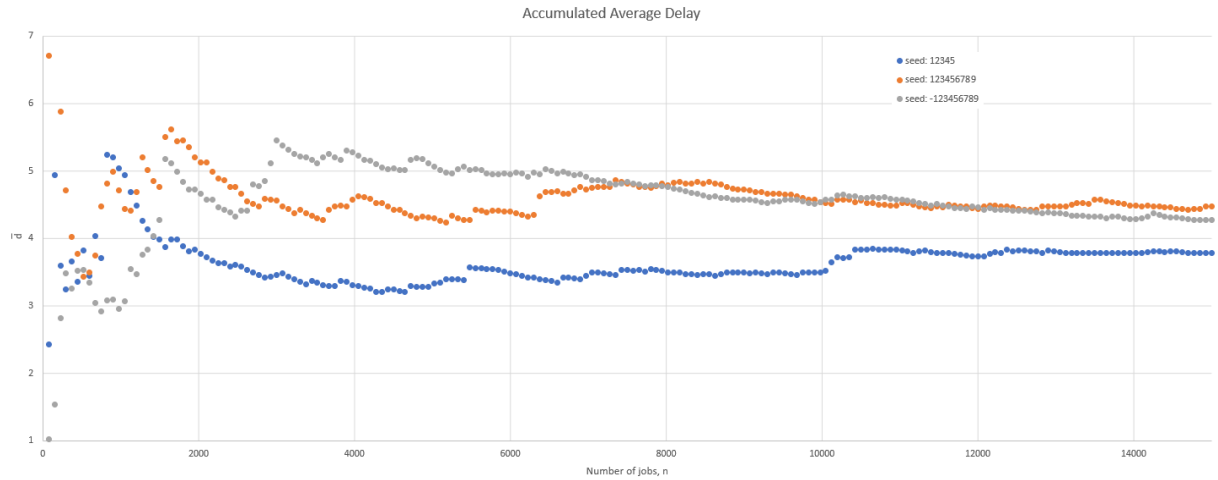

Figure 7: Accumulated Average Delay.

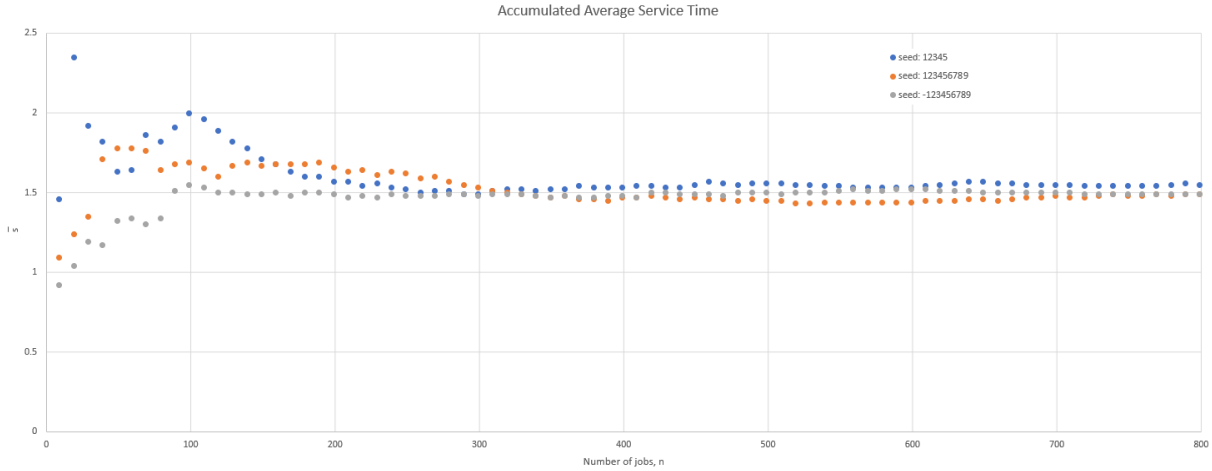

Figure 8: Accumulated Average Service Times.

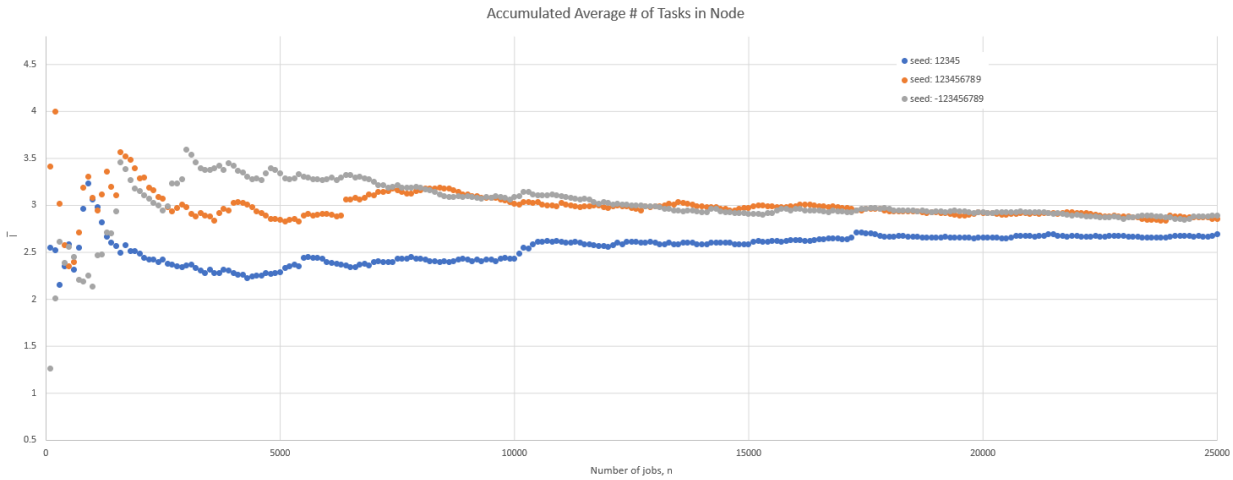

Figure 9: Accumulated Average of Tasks in Node.

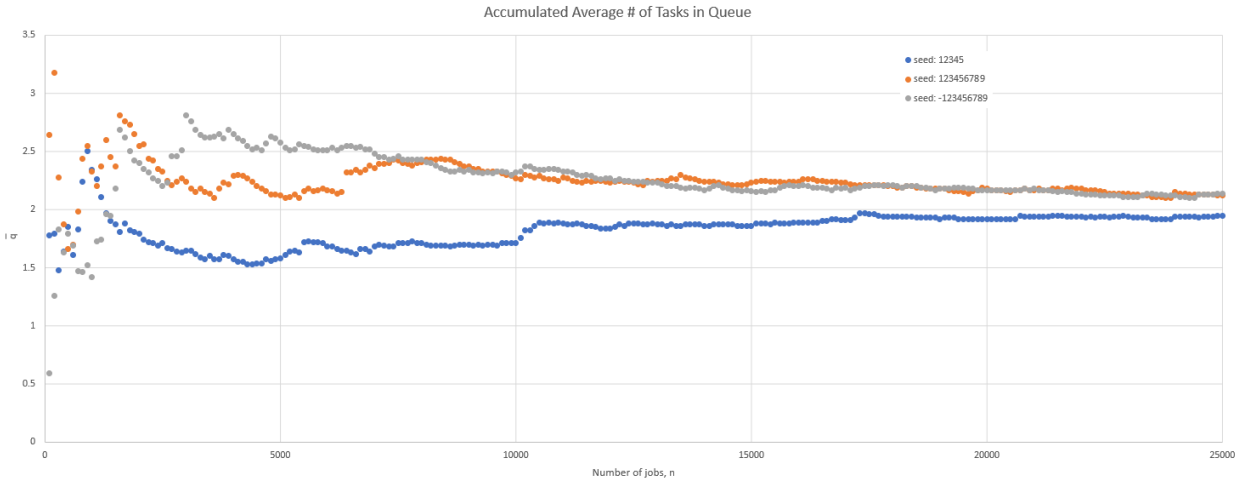

Figure 10: Accumulated Average of Tasks in Queue.

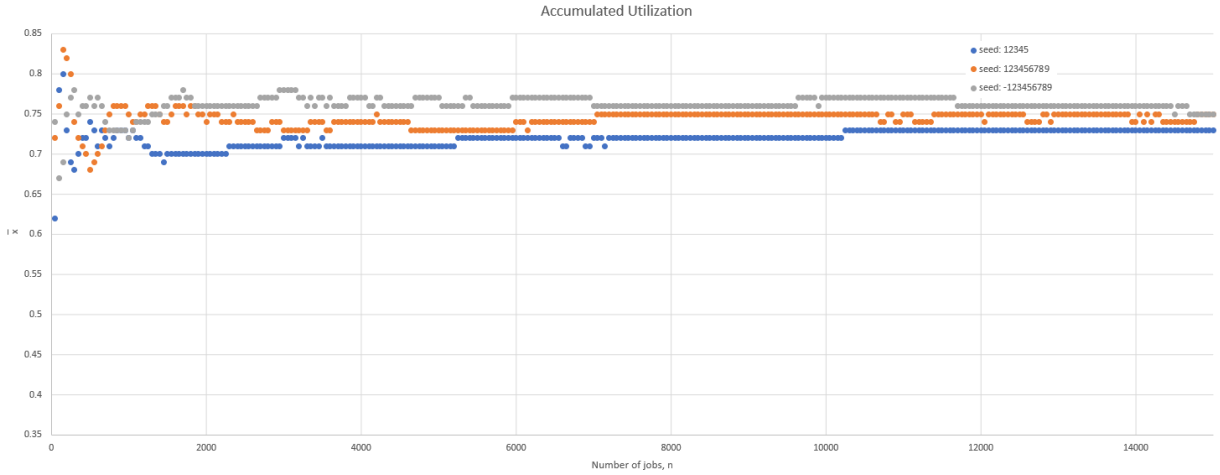

Figure 11: Accumulated Average Utilization.

```
for 10000000 jobs and seed 123456789
   average interarrival time =   2.00
   average wait ............ =   5.79
   average delay ........... =   4.29
   average service time .... =   1.50
   average # in the node ... =   2.90
   average # in the queue .. =   2.15
   utilization ............. =   0.75
```

```
for 10000000 jobs and seed -123456789
   average interarrival time =   2.00
   average wait ............ =   5.79
   average delay ........... =   4.29
   average service time .... =   1.50
   average # in the node ... =   2.90
   average # in the queue .. =   2.15
   utilization ............. =   0.75
```

```
for 10000000 jobs and seed 12345
   average interarrival time =   2.00
   average wait ............ =   5.78
   average delay ........... =   4.28
   average service time .... =   1.50
   average # in the node ... =   2.89
   average # in the queue .. =   2.14
   utilization ............. =   0.75
```

Figure 12: Steady-State Results after 100,000 jobs.

system increases. As shown in each plot, initially, the graphs are in a transient state. This refers to the beginning event where the graph behaves erratically. Eventually, however, the graph reaches a steady-state and the characteristic statistics among all the seed values are very similar. This is shown by a fairly linear line in each graph where all seeds reach similar values.

2.1.3 C)

The average interarrival rate, average service rate, and utilization are similar in both Example 3.1.3 and Example 3.1.4; however, the other characteristics are different. Example 3.1.3 utilizes Uniform(1.0, 2.0) for service times rather than the multi-task service model of Example 3.1.4. Since the GetArrival() function is unmodified between the two examples (both utilize Exponential(2.0), the arrival rates are the same. Furthermore, the service rates are the same as well because the mean service time remains at 1.5 in Example 3.1.4 as proved in question 2a above which is the mean of Uniform(1.0, 2.0) ($\mathtt{Uniform}(1.0, 2.0) = \frac{1.0+2.0}{2} = 1.5$). Since service rate and interarrival rate are the same between the examples, the utilization is also the same because it depends on these rates.

However, the other statistics are quite different. Example 3.1.4 values are higher than Example 3.1.3. The average wait is 3.83 in Example 3.1.3 but 5.77 in Example 3.1.4; likewise, the other statistics are 2.33 and 4.27 for average delay, 1.92 and 2.89 for average number of tasks in the node, 1.17 and 2.14 for average number of tasks in the queue.

These differences owe to the method of which Example 3.1.4 processes tasks. Example 3.1.3 processes tasks one at a time, each with a Uniform() distribution of service time. Example 3.1.4, however, processes tasks once a certain number of tasks have arrived in the queue (number of tasks required to start processing the multiple tasks is determined in a Geometric() distribution fashion with service time of each task being a Uniform() distribution). Since multiple tasks must arrive to begin processing, the delay time increases because tasks must wait to be processed till the node has accumulated enough tasks to begin multi-task processing. Since delay time increases and service time remains the same, the overall wait time also increases (wait = delay + service). Furthermore, the number of tasks in the queue increases since they must be accumulated till the number to be processed is reached; and then the multi-tasks are brought from the queue to the server. This increase in number of tasks held in the queue increases the overall number of tasks in the node because the node consists of the queue and server.

3 QUESTION 3

Description: Ex.3.3.1: Let β be the probability of feedback and let the integer-valued random variable X be the number of times a job feeds back.

- For x = 0, 1, 2, ... what is $P_r(X = x)$?
- How does this relate to the discussion of acceptance/rejection in Section 2.3 (i.e., Example 2.3.8)?

3.1 Solution

3.1.1 A)

$P_r(1 - B)$ refers to the departure of a task and $P_r(B)$ refers to the feedback. Since X=x is the number of times a job is fed back, the probability of feedback would be as follows:
$P_r(X = 0) = Pr(1 - B)$
$P_r(X = 1) = Pr(1 - B) * Pr(B)$
$P_r(X = 2) = Pr(1 - B) * Pr(B) * Pr(B)$
and so on.
Therefore the general form would be:
$P_r(X = x) = Pr(1 - B) * (Pr(B)^x)$

3.1.2 B)

Example 2.3.8 utilized the Acceptance/Rejection method to find random points within the circle. It worked by finding random points within the square and then either accepting or rejecting the point based on if it was inside the circumference of the circle. The reason for this was utilizing the circle's dimensions to find a random point inside the circle did not lead to uniform distribution, rather the points tended towards the center of the circle.
Similar idea, the feedback allows the task to return to the node rather than output. This feedback could be to ensure the output values are under a certain error value for example. Any task output that contains a certain amount of error or over a value that could be extreme for the simulation would be removed from the output. This can also be seen in control theory when the beta value is utilized in a system to feedback into the system and improve the output values - attempting to find an equilibrium. Another example can be to simulate real life systems. If attempting to simulate a network queuing system to analyze the best method to create a network, this could be used to find the number of jobs that would be rejected and not able to be outputted due to too many jobs arriving at once to the network.

4 QUESTION 4

Description: Ex. 3.3.4: Modify program ssq2 to account for a finite queue capacity.

- For the queue capacities 1,2,3,4,5, and 6, construct a table of the estimated steady-state probability of rejection.
- Also, construct a similar table if the service-time distribution is changed to be Uniform(1.0, 3.0).
- Comment on how the probability of rejection depends on the service process.
- How did you convince yourself these tables are correct?

4.1 Solution

4.1.1 A)

For part a, I added the following logic to ssq2 (Figure 13). This code works by creating an array to hold the departure from the queue times of tasks in the queue (calculated by delay + arrival since after that moment, the task will leave the queue and be in the server for processing). Once a new task arrives after the queue departure time (delay + arrival) of another task, the old task is purged from the array and replaced with a 0. The assumption is that no task will have a queue departure time of 0. The array is then counted after purging already departed tasks, if the array is full then the new task is rejected. However, if the array is not completely full, then the task's queue departure time is added to the queue at an empty entry.

```
// Initializes queue counter
queue_count = 0;

// Removes any tasks whose departure < new arriv
for(int i = 0; i < QUEUE_CAP; i++){
  if(queue_arr[i] < arrival)
    queue_arr[i] = 0;
}

// Counts the number of active tasks in queue
for(int i = 0; i < QUEUE_CAP; i++){
  if( queue_arr[i] != 0){
    queue_count++;
  }
}

// Finds if queue is full
if(queue_count == QUEUE_CAP){
  rejection++;
}
else{
  int first_zero = 0;
  for(int i = 0; i < QUEUE_CAP; i++){
    if( queue_arr[i] == 0 && first_zero == 0){
      queue_arr[i] = arrival + delay;
      first_zero = 1;
    }
  }
}
```

Figure 13: Finite Queue Capacity Logic.

From these results, I constructed Table 1 with the estimated steady-state probability of rejection.

Table 1: Steady-State Estimated Probability of Rejection.

Queue Cap.	1,000,000 tasks	10,000,000 tasks	100,000,000 tasks
1	0.34870	0.35026	0.35021
2	0.17328	0.17466	0.17465
3	0.08934	0.09056	0.09061
4	0.04720	0.04829	0.04836
5	0.02529	0.02625	0.02630
6	0.01371	0.01446	0.01449

4.1.2 B)

For part b, the code only changed Uniform(1.0, 2.0) to Uniform(1.0, 3.0) in the GetService() function. The new estimated steady-state probability of rejection values are in Table 2.

Table 2: Steady-State Estimated Probability of Rejection with Uniform(1.0, 3.0).

Queue Cap.	10,000,000 tasks	100,000,000 tasks	500,000,000 tasks
1	0.99730	0.99913	0.99911
2	0.99502	0.99836	0.99834
3	0.99289	0.99762	0.99760
4	0.99086	0.99690	0.99688
5	0.98891	0.99620	0.99619
6	0.98703	0.99552	0.99551

4.1.3 C)

Probability of rejection depends on the number of tasks in the queue waiting to be serviced. The GetService() function determines how long a task takes inside the service node. However, the service node only performs one task at a time. This means that a greater service time results in increase delay time as the tasks in the queue must wait for their time to be serviced. Uniform(1.0, 3.0) gives a mean service time of $\frac{1.0+3.0}{2} = 2.0$. Therefore, tasks take a mean service time of 2.0 but before they had a mean service time of 1.5. This increase in service time affects the delay time of tasks in the queue. Since tasks will wait in the queue longer, more tasks are rejected that arrive to the node.

4.1.4 D)

To convince myself that these tables are accurate, I verified my script by printing the queue, task arrival, and task departure information for each task on 20 total tasks for queue capacity size of 1, 2 and 3. I went task by task ensuring that the queue was updated appropriately and the task rejection occurred when the task ought to be rejected. When I was sure that the rejection was occurring correctly, I added the total rejections and divided this by the total number of tasks to get the estimated probability of rejection.
Next, I performed the script with different numbers of tasks until a steady-state was occurring by the rejection probability being fairly consistently among the tests. To achieve this, I performed the experiment with 1,000 tasks, 10,000 tasks, 100,000 tasks, 1,000,000 tasks, 10,000,000 tasks, 100,000,000 tasks, and 500,000,000 tasks. In the last two task levels, the probability remained fairly consistent (within 0.01% of each other). This showed the script had reached steady-state for the rejection probability.

REFERENCES