Airport Queues Simulation

Talmai B. Oliveira
oliveitb@email.uc.edu

1. INTRODUCTION

The purpose of this project is to simulate the queues that an airline customer will have to face in order to be able to board a plane. We assume that there is only one air line company in the airport. Furthermore, we assume that passengers' interarrival times are independent and identically distributed (IID) random variables with an exponential distribution with mean $1/\lambda$. The service times are also assumed to be IID and exponentially distributed random variables with mean $1/\mu$.

Initially the passenger will have to pass through the check-in gate (M/M/1) where they will have to present a copy of their flight confirmation at the reception desk. Once inside the terminal building, due to security measures, all passengers must pass through security check. Passengers are split up into three different queues (M/M/1), where they will be throughly searched and checked. There is an airport security officer that directs the passengers to each queue, guaranteeing that each queue gets the same amount of passengers. Finally, once inside the airport lounge, passengers can head towards their plane, where a quick check of identification documents and baggage x-ray (M/M/1) will be followed by boarding of the aircraft. A diagram of the queues can be seen in Figure 1.

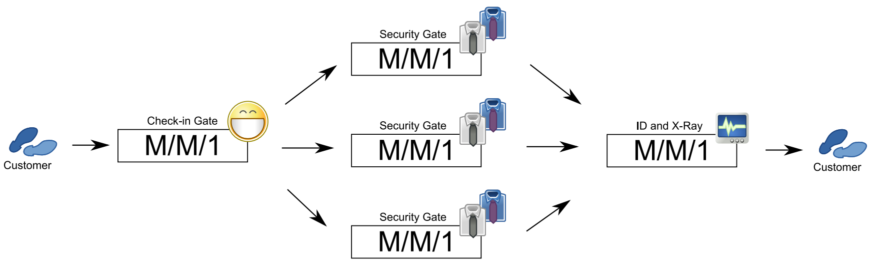

Figure 1. Diagram of Airport Queue Placement.

In this report, we analyze, specify and show the results of our implementation of the simulation. In the next section, we specify how our random number and variate generator works. Then, in section 3, we describe the flow chart of our implementation. In section 4, we evaluate the simulation results, and finally in section 5, conclusions are stated.

2. RANDOM NUMBER AND VARIATE GENERATOR

For the **random number generator**, we used a multiplicative LCG, based on an equivalent expression of Lehmer's original proposal.[1] It takes the following form:

$$x_n = (8 * 8192 + 3)\ x_{n-1}\ mod\ (2^{31}) \tag{1}$$

For efficiency reasons, the modulus m takes the form 2^k. To guarantee a full period, the multiplier a is in the form $8i \pm 3$. The period is then equal to $2^{31-2} = 2^{29} = 536870912$, as long as the initial seed is odd. For the seed selection, we wrote a program that would generate a number of seeds for intervals of 100.000. By doing this, we can guarantee that our simulations will use nonoverlapping streams. Although we did not test our random number generator, we created another program to generate 1000 random normalized numbers between 0.0 and 1.0 with different seeds, and compared plotted results with the *rand()* function of C++. Results shown in Figure 2 indicate that our generator is uniformly distributed and shows results similar to that of the C++ *rand()* function.

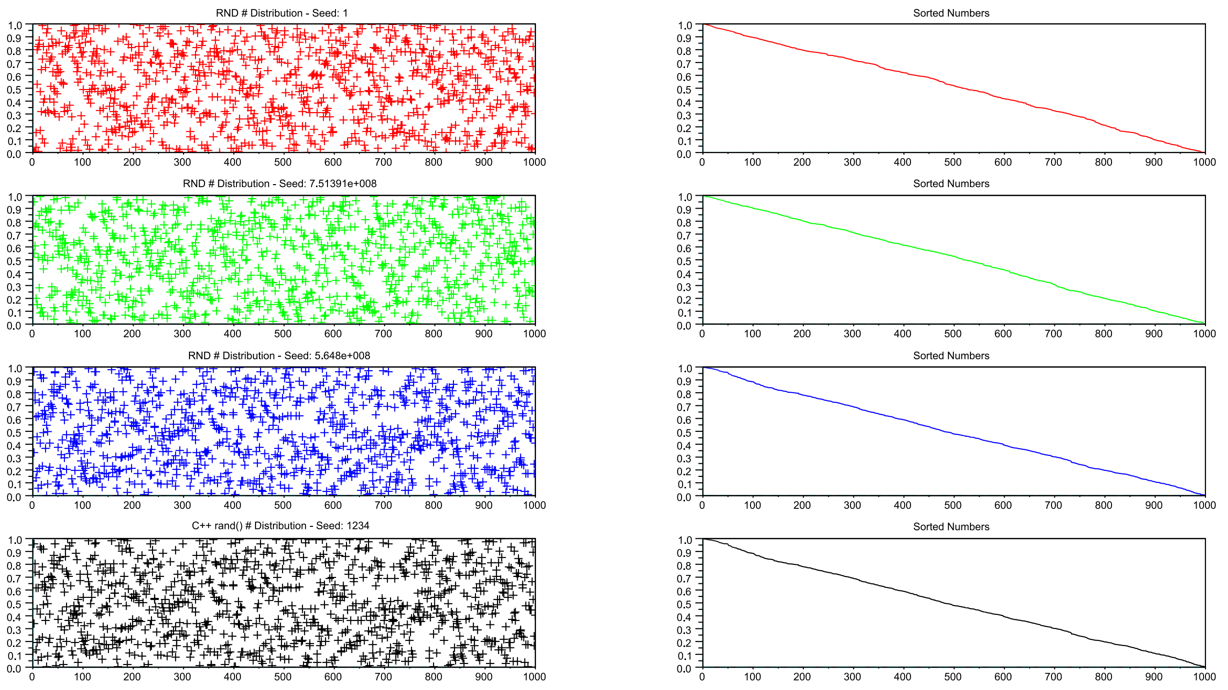

Figure 2. Plots of Normalized Random Numbers Generated.

For the exponential **random variate generator**, we relied on the inverse transformation. Since the *pdf* is $f(x) = \lambda e^{-\lambda x}$, we can generate exponential random values from an uniformly distributed random number u through the CDF $F(x) = 1 - e^{-\lambda x}$, and discover that $x = \dfrac{ln(1-u)}{-\lambda}$. To test our generator, we generated random exponential values with different seeds, and in a similar manner, plotted the results as can be seen in Figure 3.

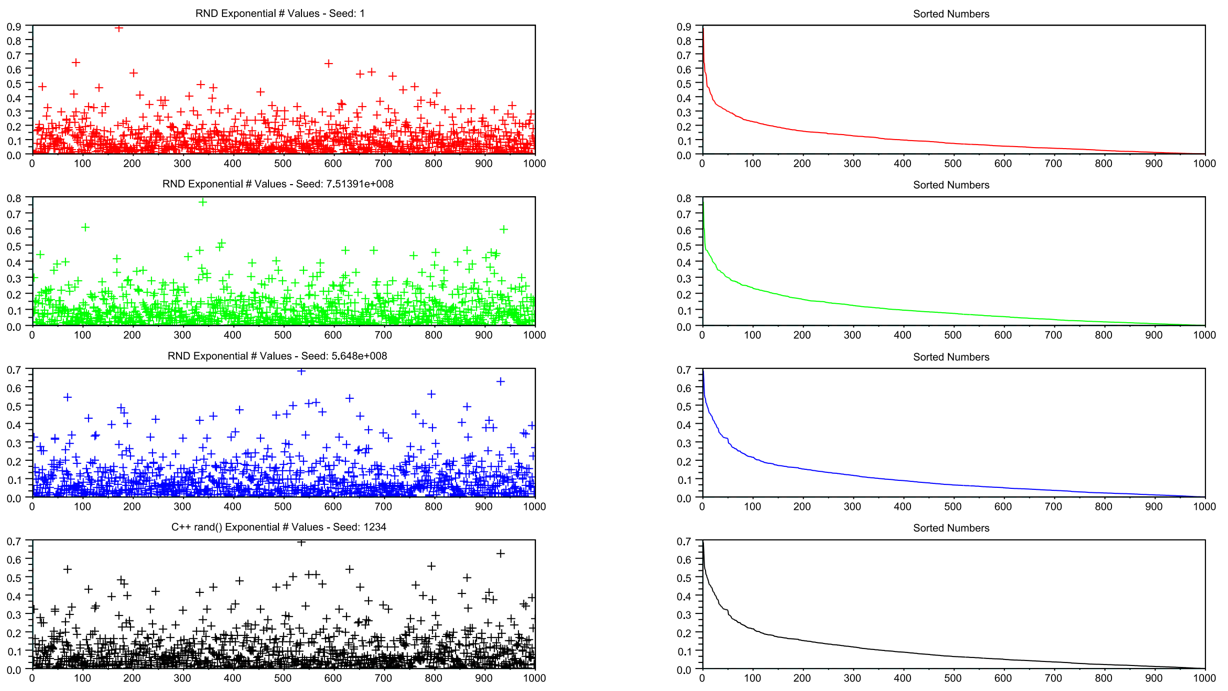

Figure 3. Plots of Normalized Exponential Random Generated Values.

3. PROGRAMMING DESCRIPTION AND FLOW CHART

We implemented our simulator in C++. Using an object-oriented approach, a simplified view of our modeling can be seen in Figure 4. In the *initialize()* method of the MM1_Queue object, the expected values (utilization, idle, queue length, number of customers, waiting time and response time) are calculated, and at each simulation step, the mean values are calculated to determine the variance and error range.

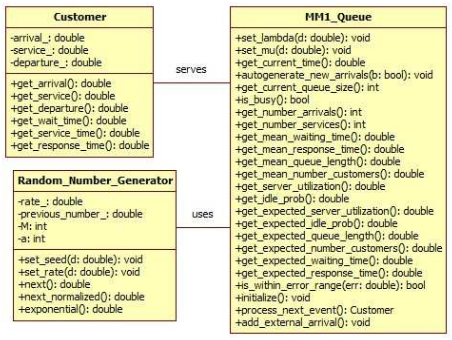
Customer
-arrival_: double
-service_: double
-departure_: double
+get_arrival(): double
+get_service(): double
+get_departure(): double
+get_wait_time(): double
+get_service_time(): double
+get_response_time(): double
serves
MM1_Queue
+set_lambda(d: double): void
+set_mu(d: double): void
+get_current_time(): double
+autogenerate_new_arrivals(b: bool): void
+get_current_queue_size(): int
+is_busy(): bool
+get_number_arrivals(): int
+get_number_services(): int
+get_mean_waiting_time(): double
+get_mean_response_time(): double
+get_mean_queue_length(): double
+get_mean_number_customers(): double
+get_server_utilization(): double
+get_idle_prob(): double
+get_expected_server_utilization(): double
+get_expected_idle_prob(): double
+get_expected_queue_length(): double
+get_expected_number_customers(): double
+get_expected_waiting_time(): double
+get_expected_response_time(): double
+is_within_error_range(err: double): bool
+initialize(): void
+process_next_event(): Customer
+add_external_arrival(): void
Random_Number_Generator
-rate_: double
-previous_number_: double
-M: int
-a: int
+set_seed(d: double): void
+set_rate(d: double): void
+next(): double
+next_normalized(): double
+exponential(): double
uses

Figure 4. Modeling of Classes.

It is the method *process_next_event()* that handles all of the actual simulation code, and the next simplified flow chart describes it's inner workings. We refer the reader to the source code for more details.

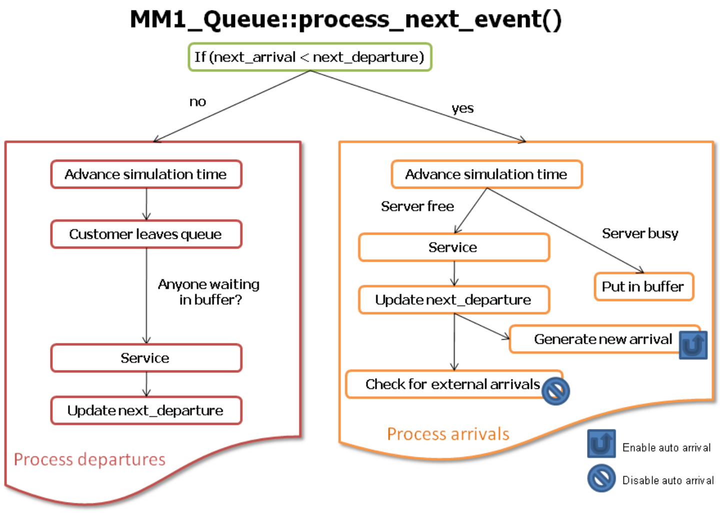

Figure 5. Flow chart of the *process_next_event()* Method.

The initialization of the objects occurs in our *main()* function. As can be seen in the following flow chart, we create and initialize all the queues and while the error is above a pre-determined threshold, we continue simulating. Each queue has 2 random variate generators - one used to generate values for the arrival times and the other for the service times. The first queue (check-in) is the only queue that generates new arrivals. All the other queues only process customers that have been served. That is why we need to disable the auto-arrival generation property of these. Only when a customer leaves one queue is the arrival event created on the subsequent queue.

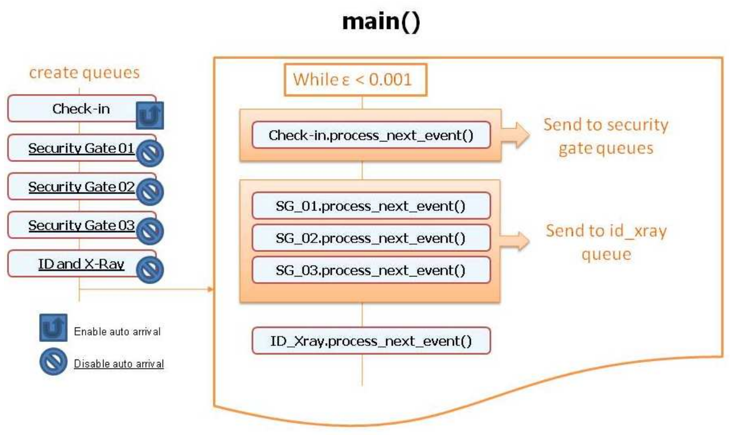
main()
create queues
Check-in
Security Gate 01
Security Gate 02
Security Gate 03
ID and X-Ray
Enable auto arrival
Disable auto arrival
While ε < 0.001
Check-in.process_next_event()
Send to security gate queues
SG_01.process_next_event()
SG_02.process_next_event()
SG_03.process_next_event()
Send to id_xray queue
ID_Xray.process_next_event()

Figure 6. Flow chart of the *main()* Function.

4. ANALYSIS OF SIMULATION RESULTS

In this section we present the analysis of our simulation results. Initially we will describe how we identify the conditions to terminate the simulation. Then the trace data from various runs are plotted and validated.

4.1 Stopping Criteria

During the simulation runs, we collect data to calculate the mean waiting time, mean response time, mean queue length, mean number of customers in the system, server utilization and idle probability. The mean number of customers and the mean queue length consider the time average, as specified in the section 25.4 of the textbook.[1] Once again, we refer the reader to the source code (file: *mm1_queue.cpp*, method: *update_evaluation_values()*, line number: 28 and file: *mm1_queue.hpp*, line number: 109–114).

Based on the accumulation of this data, we have implemented two different stop criteria: 90% confidence interval (file: *mm1_queue.hpp*, method: *is_within_confidence_interval()*, line number: 126) or error range (file: *mm1_queue.hpp*, method: *is_within_error_range()*, line number: 141). Using the 90% confidence interval, the simulation time would advance to 8.12 seconds, but the error of the mean response time would be in the range of 0.05. However, when running the simulation until the mean response time was within an error range of 10^{-3} required 23, 723.6 seconds of simulation time. **All performance analysis results are then based on the error range stopping criteria.**

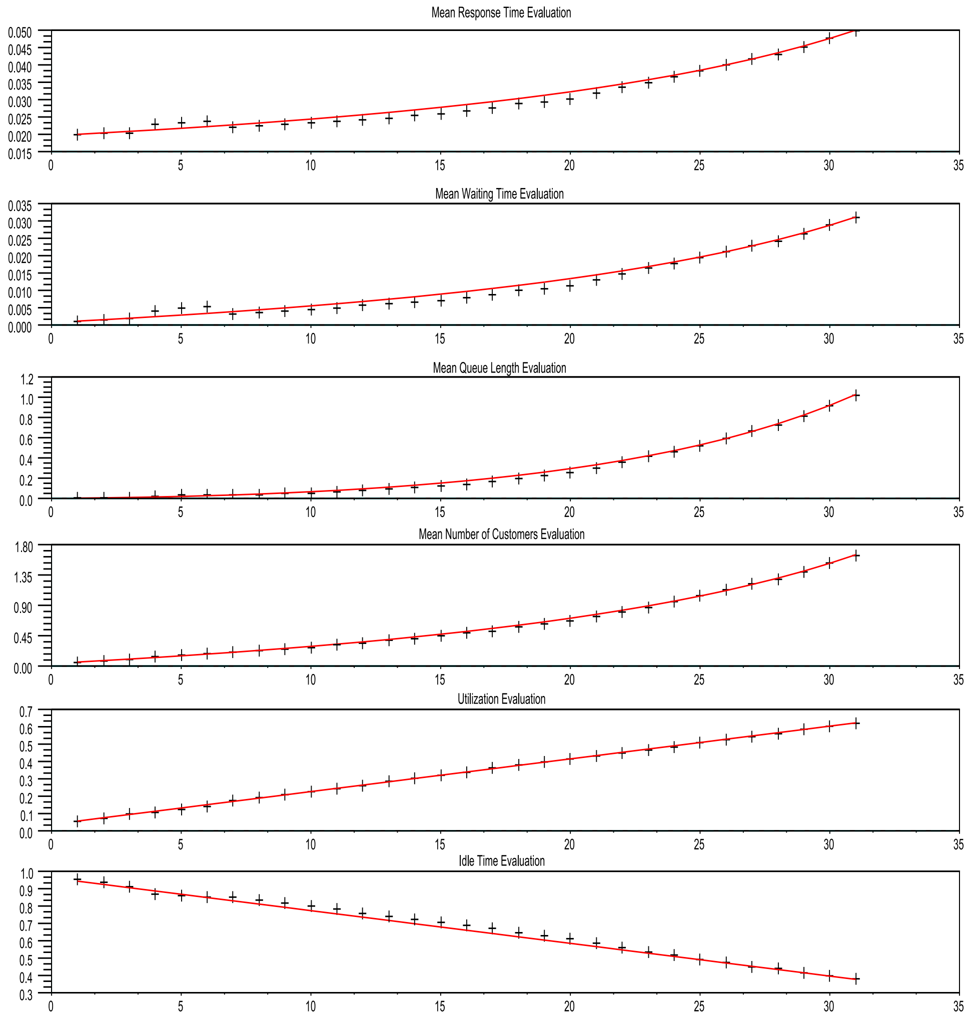

Figure 7. Plots to Evaluate Simulation: Theory vs. Practice.

4.2 Simulation Results

The next results that will be presented were obtained from simulation runs with the following parameters:

→ Check-in queue with $\mu = 53$ and $\lambda = 33$, and random number generators with seeds 1 and $7.51391e+008$;

→ Security Gate 01, 02 and 03 queues with $\mu = 20$ and expected $\lambda = 11$ and seeds $5.648e + 008$ and $1.42815e + 009$, $1.4853e + 009$ and $1.77402e + 009$, and $4.49834e + 008$ and $2.35495e + 008$ respectively.

$\rightarrow$ ID and X-Ray queue with $\mu = 80$ and expected $\lambda = 33$, and seeds $6.92328e + 008$ and $1.15022e + 008$.

$\rightarrow$ Mean response rate of the check-in and security gate queues with error $\epsilon < 10^{-3}$ (Simulation time of $23,723.6$ seconds).

For initial simulation evaluation, we first plot the theoretical results versus the actual results obtained from simulation runs where the λ is varied from 3 to 33. This can be seen in Figure 7. The line plot is from theoretical results, while the points are the values obtained from calculated results.

A full simulation run, obtained the following results:

$\rightarrow$ For the **check-in queue**, the mean response time was of 0.0500609 [expected: 0.05], the mean waiting time: 0.031202 [expected: 0.0311321], the mean queue length: 1.02999 [expected: 1.02736], the mean number of customers: 1.65254 [expected: 1.65], the server utilization: 0.622543 [expected: 0.622642] and the server idle: 0.376265 [expected: 0.377358].

$\rightarrow$ For the **security gate 01 queue**, the mean response time was of 0.110112 [expected: 0.111111], the mean waiting time: 0.0602689 [expected: 0.0611111], the mean queue length: 0.664374 [expected: 0.672222], the mean number of customers: 1.21377 [expected: 1.22222], the server utilization: 0.549397 [expected: 0.55] and the server idle: 0.451598 [expected: 0.45].

$\rightarrow$ For the **security gate 02 queue**, the mean response time was of 0.110129 [expected: 0.111111], the mean waiting time: 0.0602298 [expected: 0.0611111], the mean queue length: 0.664422 [expected: 0.672222], the mean number of customers: 1.21489 [expected: 1.22222], the server utilization: 0.550467 [expected: 0.55] and the server idle: 0.449756 [expected: 0.45].

$\rightarrow$ For the **security gate 03 queue**, the mean response time was of 0.110217 [expected: 0.111111], the mean waiting time: 0.0601764 [expected: 0.0611111], the mean queue length: 0.659968 [expected: 0.672222], the mean number of customers: 1.20878 [expected: 1.22222], the server utilization: 0.548811 [expected: 0.55] and the server idle: 0.450967 [expected: 0.45].

$\rightarrow$ For the **id and x-ray queue**, the mean response time was of 0.0229146 [expected: 0.0212766], the mean waiting time: 0.0104183 [expected: 0.0087766], the mean queue length: 0.344425 [expected: 0.289628], the mean number of customers: 0.757549 [expected: 0.702128], the server utilization: 0.413124 [expected: 0.4125] and the server idle: 0.5143 [expected: 0.5875].

Finally, the sorted plot of the trace files generated from the first 4000 serviced passengers can be seen in Figure 8. On the left side we show the service time distribution, while on the right the waiting time distribution. Results indicate that our implementation is correct.

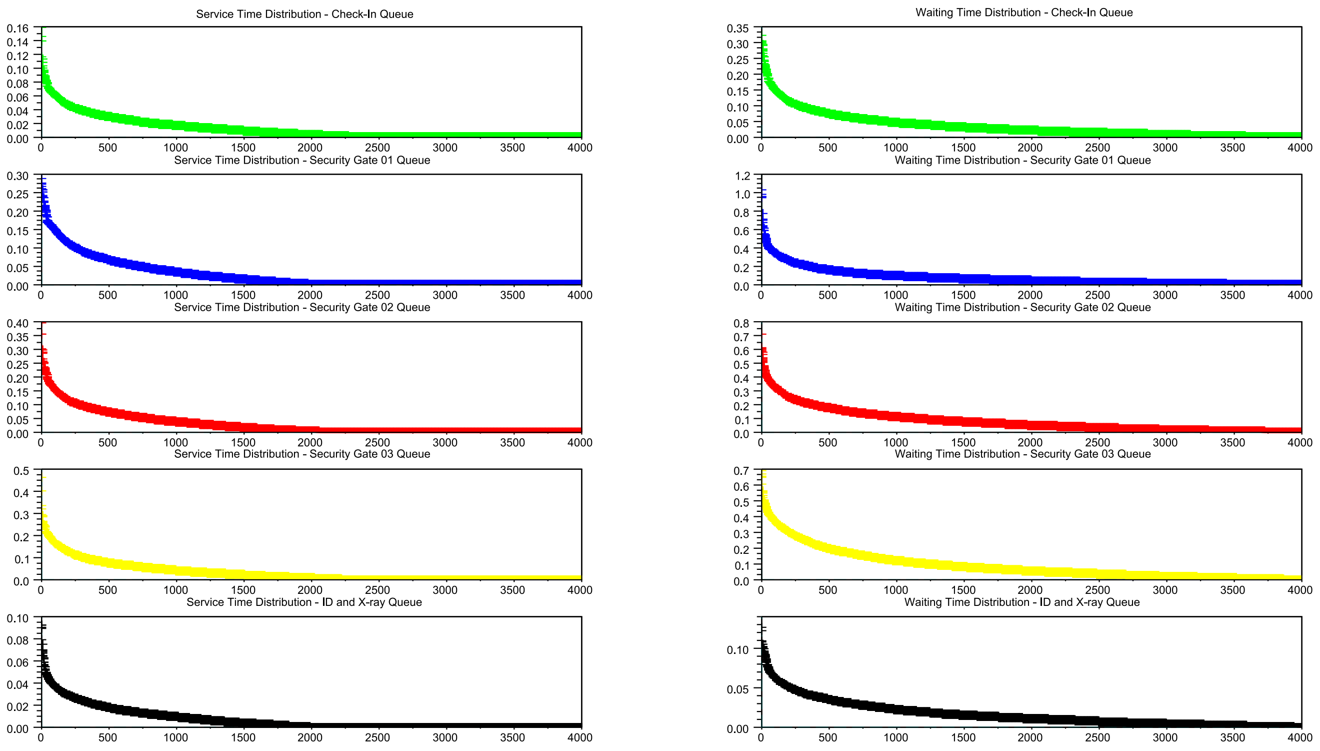

Figure 8. Plots of Distribution of the Response and Waiting Time of the Queues .

5. CONCLUSIONS

In this report, we specified and analyzed results of our implementation of 5 queues inside an airport. We described how we implemented our uniform random number and exponential random variate generators and how results indicate that they are implemented correctly. We also explained how our implementation functions and how the queues are sequentially used. Finally, we indicated what stopping criteria we used and analyzed our results which indicates that our implementation is indeed correct. Through the process of this assignment we have learned not only how to implement a simulation scenario with multiple M/M/1 queues, but also on how to analyze practical situations that we might face during our PhD.

REFERENCES

[1] Jain, R., [*The Art of Computer Systems Performance Analysis: techniques for experimental design, measurement, simulation, and modeling*], Wiley (1991).