

OPERATION RESEARCH

# QUEUEING SIMULATOR

## AIRPORT SECURITY BOARDING QUEUE

EXPERIENCE THE SIMULATOR (<https://simulator.saqlain1020.com/>)

SOURCE CODE (<https://github.com/saqlain1020/airport-queue-simulation>)

---



### INTRODUCTION

Welcome to our airport security boarding queue simulator! In this simulation, we will be monitoring the efficiency and performance of the boarding queue at an airport security checkpoint. We have implemented the M/M/C queueing model, a mathematical representation of a queue system, to analyze and understand the behavior of the boarding process. By using this model, we aim to provide insights into the potential bottlenecks and areas for improvement in the queue management process.

---

---

## DATA SOURCING

### Jinnah International Airport

Data collection was started on 7th of January 2023 and continued to record for around 95 minutes. During this interval we observed almost 20 customers with random arrivals and partially distinct service times. Original model was running on a single server and was able to expand and shrink w.r.t demand and congestion.

During this monitoring, the observed mean inter arrival time was 4.75 minutes with a mean service time of 5.4 minutes. This may vary based on the policies of regulatory authorities or flight details or customer demands and other factors as well.

### System Significance

<i>Parameter</i>	<i>symbol</i>	<i>Calculated</i>	<i>Simulated</i>
Average inter arrival time	Lambda $\lambda$	4.75	4.4
Average service time	Meu $\mu$	5.4	5.4
Average number of customers in the system	L	1.675	1.969
Average number of customers in the queue	Lq	0.543	0.741
Average number of waiting customers in the system	W	7.978	8.661
Average number of waiting customers in the queue	Wq	2.578	3.261
Average idle time of server	Idle	0.432	0.386

To compute the system significance, we need to compare the values of the performance metrics obtained from the simulated model and the calculated model. The system significance gives us an idea of how closely the simulated model approximates the real system.

---

In general, a system significance of less than 5% is considered good, indicating that the simulated model is a good approximation of the real system. A significance value between 5% and 10% is considered acceptable, while a value greater than 10% suggests that the simulated model is not a good approximation of the real system.

---

To calculate the system significance, we can use the formula:

$$\text{System Significance} = | (\text{Simulated Value} - \text{Calculated Value}) / \text{Calculated Value} | \times 100$$

Using the values given, we can calculate the system significance for each performance metric as follows. Using the above mentioned formula of significance, we now calculate all of the below listed parameters.

<i>Parameter</i>	<i>symbol</i>	<i>Freedom Of Error</i>
Average number of customers in the system	L	17.32%
Average number of customers in the queue	Lq	36.43%
Average number of waiting customers in the system	W	8.54%
Average number of waiting customers in the queue	Wq	26.46%
Average idle time of server	Idle	10.65%
Average server utilization	Rho ρ	8.10%

Based on these calculations, we can see that the system significance for the Average Customers in System and Average Customers in Queue metrics is higher than the acceptable range of 5% to 10%, indicating that the simulated model may not be a good approximation of the real system for these metrics. However, the system significance for the other metrics is within an acceptable range, suggesting that the simulated model is a reasonable approximation of the real system for those metrics.

## APPLICATION VIEW

Step 1: Open application

Airport Security Boarding Queue Simulation

Chi Square Distributions

**Input Parameters**

Speed  
Instant

Number of Servers  
1

Run Simulation

**Performance Measures**

L: 10.581  
Average Customers in System

Lq: 9.445  
Average Customers in Queue

W: 50.262 (min)  
Average Waiting Customers in System

Wq: 44.862 (min)  
Average Waiting Customers in Queue

**Server Utilization Graphs**

Step 2: Enter parameters for number of customers and number of servers

Step 3: Click run simulation button

Airport Security Boarding Queue Simulation

Chi Square Distributions

**Input Parameters**

Speed  
Instant

Number of Servers  
2

Run Simulation

C#	Arrival (min)	Interarrival (min)	Service (min)	Start (min)	End (min)	Wait (min)	TT (min)	Server
1	0	0	11	0	11	0	11	1
2	10	10	7	10	17	0	7	2
3	13	3	8	13	21	0	8	1
4	18	5	13	18	31	0	13	2
5	26	8	22	26	48	0	22	1
6	30	4	9	31	40	1	10	2
7	38	8	2	40	42	2	4	2
8	39	1	3	42	45	3	6	2
9	45	6	2	45	47	0	2	2
10	49	4	14	49	63	0	14	2
11	55	6	1	55	56	0	1	1
12	61	6	2	61	63	0	2	1

**Performance Measures**

*Arrivals and inter arrivals are generated with relevant measures.*



*Performance measures with graphs are created as well*

## Graphs

*Following graphs are created in the application*

- *Server Utilization for each server*
- *Queue length graphs for each server*
- *Wait time graph for each queue*
- *Turnaround time graph for each queue*

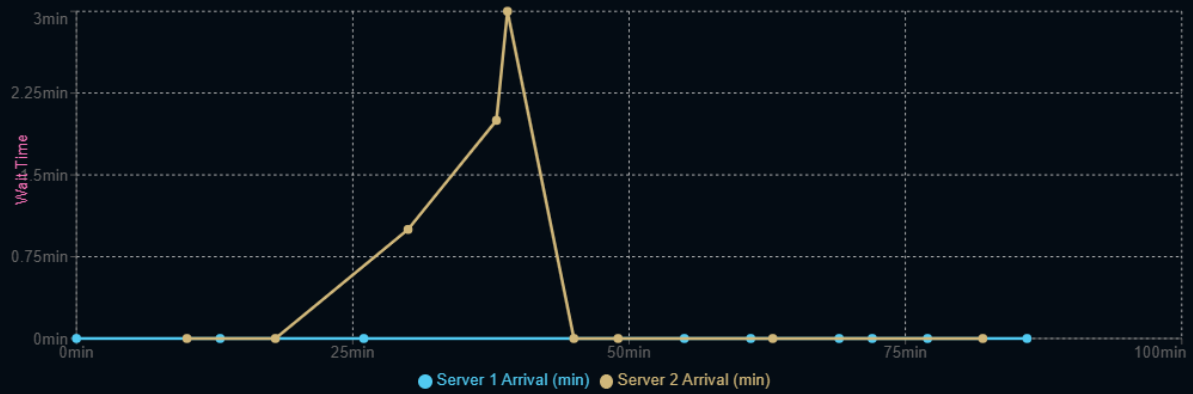
## Performance Measures

Following performance measures are calculated in the application

- Average customers in the system ( $L$ )
- Average customers in the queue ( $L_q$ )
- Average waiting customers in system ( $W$ )
- Average waiting customers in queue ( $W_q$ )
- Average idle time of server (idle)
- Average utilization of server ( $\rho$ )

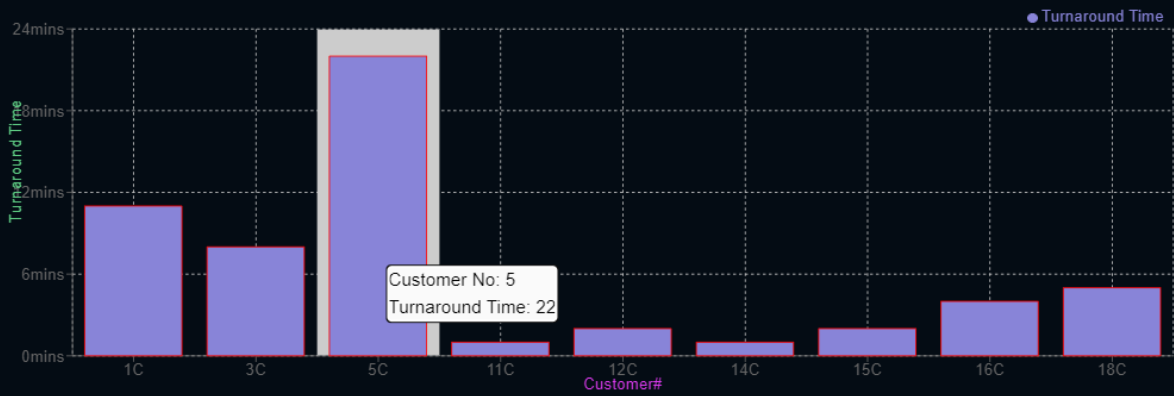


## Wait Time Graphs

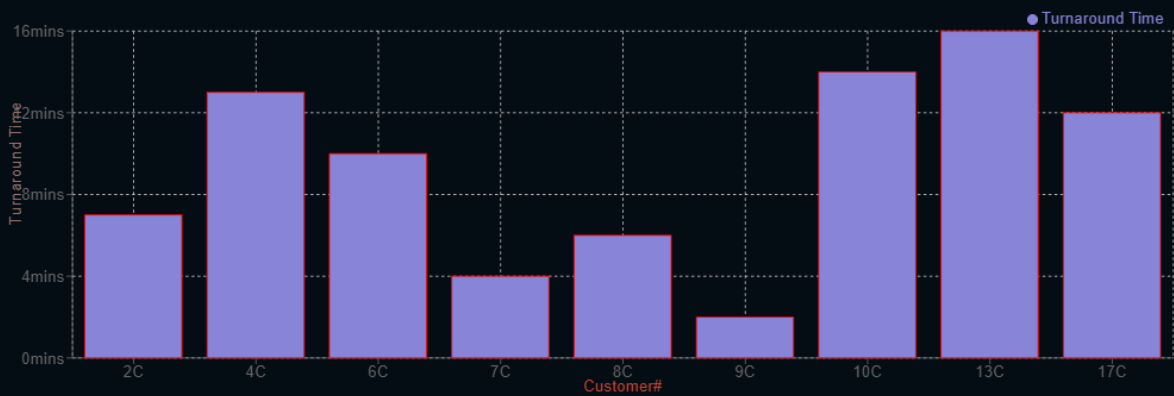


## Turnaround Time Graphs

### Server 1



### Server 2



## Custom distributions

Users can use any other custom distribution to get performance measures, just follow these steps.

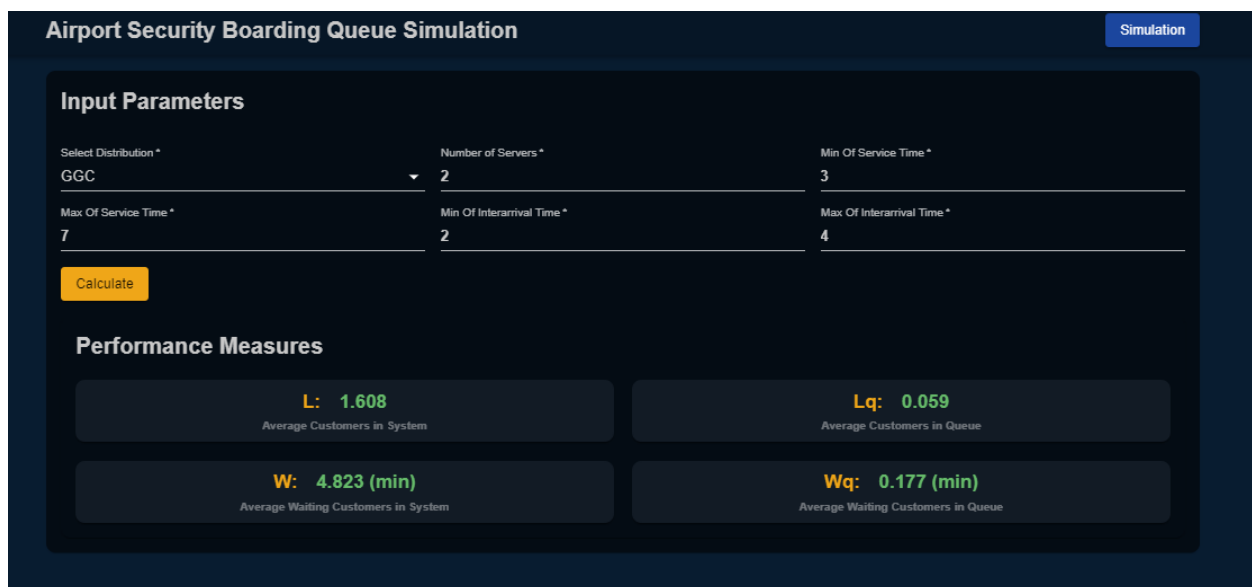
**Step 1:** Click distributions button on top right corner



**Step 2:** Click custom distribution button on top right corner

**Step 3:** Input parameters and select distribution from **MMC, MGC, GGC**

**Step 4:** Click Calculate to get the performance measures

A screenshot of the simulation interface. The title 'Airport Security Boarding Queue Simulation' is at the top left, and a 'Simulation' button is at the top right. The 'Input Parameters' section contains six input fields: 'Select Distribution \*' (dropdown menu with 'GGC' selected), 'Number of Servers \*' (text input '2'), 'Min Of Service Time \*' (text input '3'), 'Max Of Service Time \*' (text input '7'), 'Min Of Interarrival Time \*' (text input '2'), and 'Max Of Interarrival Time \*' (text input '4'). Below these fields is a yellow 'Calculate' button. The 'Performance Measures' section displays four results in a 2x2 grid: 'L: 1.608' (Average Customers in System), 'Lq: 0.059' (Average Customers in Queue), 'W: 4.823 (min)' (Average Waiting Customers in System), and 'Wq: 0.177 (min)' (Average Waiting Customers in Queue).



---

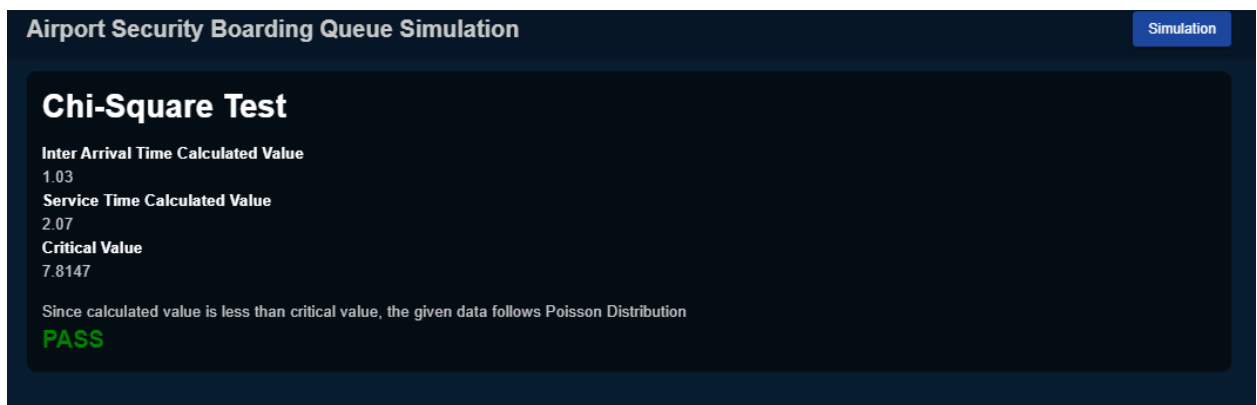
## Chi Square

User can see the chi square result of the dataset used

- Click on the chi square tab in the header



Chi square result as we can see our dataset passes the chi square test



---

## BRAIN CELL

```
const generate = (interArrivals: number[], serviceTimes: number[]) => {
  const arrivals = calculateArrivalsFromInterArrivals(interArrivals);
  let servers: Server[] = new Array(numberOfServers).fill({ endTime: 0 });
  servers = servers.map((server, index) => ({
    ...server,
    serverNum: index + 1,
    customers: []
  }));
  const customers: Customer[] = [];

  arrivals.forEach((_, i) => {
    let startTime = -1;
    let serverNum = 0;
    for (let j = 0; j < servers.length; j++) {
      const server = servers[j];
      if (server.endTime <= arrivals[i]) {
        startTime = arrivals[i];
        serverNum = server.serverNum;
        server.endTime = startTime + serviceTimes[i];
        break;
      }
    }
    if (startTime === -1) {
      let sorted = servers.sort((a, b) => a.endTime - b.endTime);
      serverNum = sorted[0].serverNum;
      startTime = sorted[0].endTime;
      sorted[0].endTime = startTime + serviceTimes[i];
    }
    let endTime = startTime + serviceTimes[i];
    let arrival = arrivals[i];
    let waitTime = startTime - arrival;
    let turnaroundTime = endTime - arrival;
    let obj: Customer = {
      arrival,
      interArrival: interArrivals[i],
      serviceTime: serviceTimes[i],
      server: serverNum,
      startTime,
      endTime,
      waitTime,
      turnaroundTime,
    };
    customers.push(obj);
    servers.find((item) => item.serverNum === serverNum)?.customers.push(obj);
  });
  setCustomerRecords(customers);
  return servers;
};
```

---

This function is the heart of the project. It is responsible for simulating the arrivals and service time for the given number of customers with respect to the server count. It assigns server number to each customer as well. Similarly calculates start, end, interArrival, waiting, turnaround for every customer. And at the last it returns the sorted array server wise of customers.

```
const MeanInterArival = 4.75;
const MeanServiceTime = 5.4;

const generateArrivals = () => {

  const serviceTimes: number[] = [];
  const interArrivals: number[] = [];
  for (let i = 0; i < numberOfCustomers; i++) {
    interArrivals.push(generateRandomExponential(MeanInterArival));
    serviceTimes.push(generateRandomExponential(MeanServiceTime));
  }

  interArrivals[0] = 0;
  return generate(interArrivals, serviceTimes);
};

export function generateRandomExponential(u: number) {
  return Math.floor(u * Math.log(Math.random()) * -1);
}

export function calcProbabilityPoisson(lambda: number, x: number) {
  let sum = 0;
  for (let i = 0; i <= x; i++) {
    sum += (Math.E ** -lambda * lambda ** i) / factorial(i);
  }
  return sum;
}
```

There comes your most awaited part where the simulation formulas were used. The mean inter arrival and mean service time are hard coded and recorded from the live observation at the airport security boarding queue. That is why these values are being used directly to simulate the arrivals and service time.

---

## GRAPHS

This function separates customer records based on the server they are assigned to. It does this by first finding the maximum server number among all the customer records, then using a loop to sort each customer record into an array indexed by their server number. Finally, the function filters out any empty arrays and returns the final result, which is an array of arrays, where each sub-array contains the customer records for a specific server.

```
export const separateCustomerServerWise = (customerRecords: Customer[]) => {
  let servers: Customer[][] = [];
  let maxServerNumber = 0;

  //get max server number
  customerRecords.forEach((c: Customer) => (
    maxServerNumber = c.server! > maxServerNumber
      ? c.server!
      : maxServerNumber
  ));

  //array of array. each index contains customers for that server
  for (let i = 0; i < customerRecords.length; i++) {
    let server = customerRecords[i].server;
    if (server) {
      if (!servers[server]) {
        servers[server] = [];
      }
      servers[server].push(customerRecords[i]);
    }
  }
  //remove empty arrays and return
  return servers.filter((item) => item !== undefined || item !== null);
};
```

---

## Server Utilization

The plotting of server utilization on the graph was not easy though, but we came up with an idea of why not show the utilization through the simulation like, at every arrival what was the utilization so far. For this we need mean arrival rate and mean service time rate at every arrival of the customer, because later the utilization will be the ratio of mean arrival rate and mean service time rate.

Below functions serve these two purposes solely.

```
const addArrivalAverageAndServiceTimeAverageAtEveryArrival = (servers: Customer[][]) => {

  if (servers.length > 0) {
    servers.forEach((server) => {
      server.forEach((customer, index) => {
        if (index === 0) {
          server[index] = { ...server[index], arrivalAverage: customer.arrival };
          server[index] = {
            ...server[index],
            serviceTimeAverage: customer.serviceTime
          };
          return;
        }
        const previousCustomer = server[index - 1];
        const previousArrivalAverage = previousCustomer.arrivalAverage;

        const previousServiceTimeAverage = previousCustomer.serviceTimeAverage;

        const newArrivalAverage = (
          (previousArrivalAverage! * index + customer.arrival!)
          / (index + 1)
        );

        const newServiceTimeAverage = (
          (previousServiceTimeAverage! * index + customer.serviceTime!)
          / (index + 1)
        );

        server[index] = {
          ...server[index],
          arrivalAverage: newArrivalAverage
        };

        server[index] = {
```

---

```
        ...server[index],
        serviceTimeAverage: newServiceTimeAverage
    });
});
});
}
```

//for the first arrival, the utilization will be zero because the customer has been served yet, it has just arrived

```
const addUtilizationAtEveryArrival = (servers: Customer[][]) => {
    servers.forEach((server) => {
        server.map((customer, index) => {
            return (server[index] = {
                ...server[index],
                utilizationAtArrival: customer.serviceTimeAverage === 0
                    ? 0 : customer.arrivalAverage! / customer.serviceTimeAverage!,
            });
        });
    });
    setServerUtilization(servers);
};
```

---

## Queue length

In order to plot the graph of Queue length, we need to see if the previous customer's turn has come or not and if not then what is the current queue length so far. For the first customer at every server, there will be no queue and for the customer coming next to him we look, if the previous customer's service got done and if not, there will be a queue.

```
const getQueueLengthAt = () => {
  if (customerRecords.length > 0) {
    //add new attribute to each server i.e. queueLength
    const servers = separateCustomerServerWise(customerRecords);
    servers.forEach((server, i) => {
      server.forEach((c: any, i) => {
        if (i === 0) {
          server[i] = { ...server[i], queueLength: 0 };
          return;
        }
        const previousCustomer = server[i - 1];
        //if customer has arrived but previous customer has not left yet
        if (c.arrival < previousCustomer.endTime!) {
          server[i] = {
            ...server[i],
            queueLength: previousCustomer.queueLength ?
              previousCustomer.queueLength + 1 : 1,
          };
          return;
        } else {
          server[i] = { ...server[i], queueLength: 0 };
        }
      });
    });
    setQueueLengthServers(servers);
  }
};
```

---

## Wait Time

For the wait time we have done the calculation while simulating the record of every customer. It will be the difference of arrival from the start time. Using this we will know how long the customer has waited for his turn.

```
const { customerRecords, waitingInTheQueueServers, setWaitingInTheQueueServers } = useApp();
React.useEffect(() => {
  if (customerRecords.length > 0) {
    const servers = separateCustomerServerWise(customerRecords);
    setWaitingInTheQueueServers(servers);
  }
}, [customerRecords]);
```

## Turnaround Time

The same work is done for turnaround time. We use the formula *endtime - arrival*. With the help of this we now know how long a customer has spent the time in the system.

```
const TurnaroundTimeGraph = () => {
  const { customerRecords, waitingInTheQueueServers } = useApp();

  React.useEffect(() => {
    if (customerRecords.length > 0) {
      const servers = separateCustomerServerWise(customerRecords);
      setWaitingInTheQueueServers(servers);
    }
  }, [customerRecords]);
}
```



---

## TEAM

Hafiz Muhammad Mutahhir Khan EB19103027

Syed Shair Ali Rashid EB19103127

Syed Saqlain Riaz EB19103119

Sumaiya Arif EB19103117

Submitted To Miss Shaista Rais

Class BSSE - B, DCS, UBIT, University Of Karachi