M/M/1 Queue Implementation and Analysis in Python

Author: Jalen Moore

MATH 4310 Project 3

Table of Contents

Introduction

The problem of which this project aims to solve is the simulation of an M/M/1 queue with an interarrival rate of 3 units per hour and a service rate of 4 units per hour. How does such a queue behave when simulated over 500 hours and how do these results compare to the theoretical output parameters of the simulation and the output parameters of another simulation (using AnyLogic). The simulation implementation is constructed using the Python multi-paradigm programming language. Additional dependencies include:

- **NumPy:** A dependency for managing and manipulating arrays, vectors, matrices, and tensors on the machine's CPU. This dependency is utilized post-simulation, to linearly compute the expect values and averages of varying output parameters.
- **Pandas:** A dependency for managing and manipulating tabled information like an Excel spreadsheet but utilized programmatically. This dependency is utilized to track the history of each simulation generated, as well as the output parameters of simulations and replications.
- **TQDM:** While not actually utilized in the simulation or analysis, this dependency provides a convenient solution for measuring the execution and progress of iterative loops.

The benefit of simulating a M/M/1 queue in this manor, as in contrast to an existing solution is for educational purposes. The implementation proposed and analyzed in the following report is not optimal with an average execution time of 2 replications per second on the target machine, but it exhibits an understanding of how to generate, track, and manipulate the information that can be gathered at each simulation iteration and each replication of a larger statistical experiment. The target machine for the following implementation is a MacBook Pro with a M3 Max processor and any execution times mentioned throughout the report are dependent on this target machine. The simulation source code is stored within a repository and is publicly available (Moore, 2024).

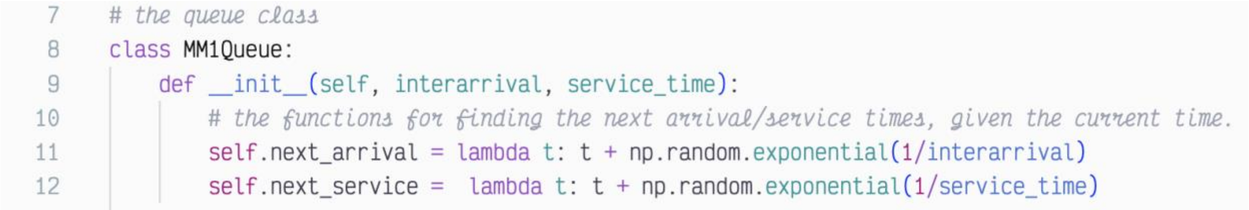
Assumptions

For the purposes of describing the implementation queue, the following assumptions and definitions were made. Each count in the queue will be regarded as a *unit*. To illustrate this, if the queue is of length 3, then there are 3 *units* in the queue. This is done to keep the simulation unit agnostic (which could be replaced by users, customers, or some sort of item).

During the simulation, it is required to calculate the next time that a unit either enters or leaves the queue. For this purpose, let λ be the inter-arrival rate in *units per hour*, and let μ be the service rate in *units per hour*. Let X be a random variable such that $X \sim Exp(1/\lambda)$ and let Y be a random variable such that $Y \sim Exp(1/\mu)$. Each random variable is distributed exponentially with the reciprocal λ and μ. The reason for this is that X and Y represent the next time delta until a new unit enters or leaves the queue. The random variables are measured in *hours per unit*. The reason for this implementation decision is discussed appropriately in the *Implementation* section.

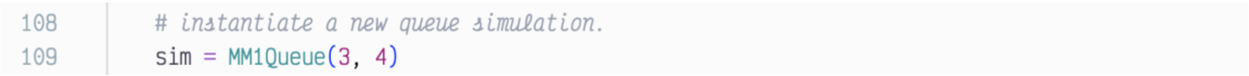
Simulation Implementation

The programmatic implementation of one replication of the M/M/1 queue simulation contains three major portions; There is the setup of initial parameters for the simulation, the simulation loop, and the replication parameter calculations. The setup of the initial simulation parameters includes retrieving the inter-arrival rate and the service rate. This is done by instantiating a *MM1Queue* object with the desired rate parameters. Figure # shows what occurs when a new *MM1Queue* is instantiated.



```
# the queue class
class MM1Queue:
    def __init__(self, interarrival, service_time):
        # the functions for finding the next arrival/service times, given the current time.
        self.next_arrival = lambda t: t + np.random.exponential(1/interarrival)
        self.next_service =  lambda t: t + np.random.exponential(1/service_time)
```

Figure #: Initialization script for the *MM1Queue* class. The *MM1Queue* class takes an inter-arrival rate and a service rate and stores them as functions which take a time input and return the next arrival or service time.

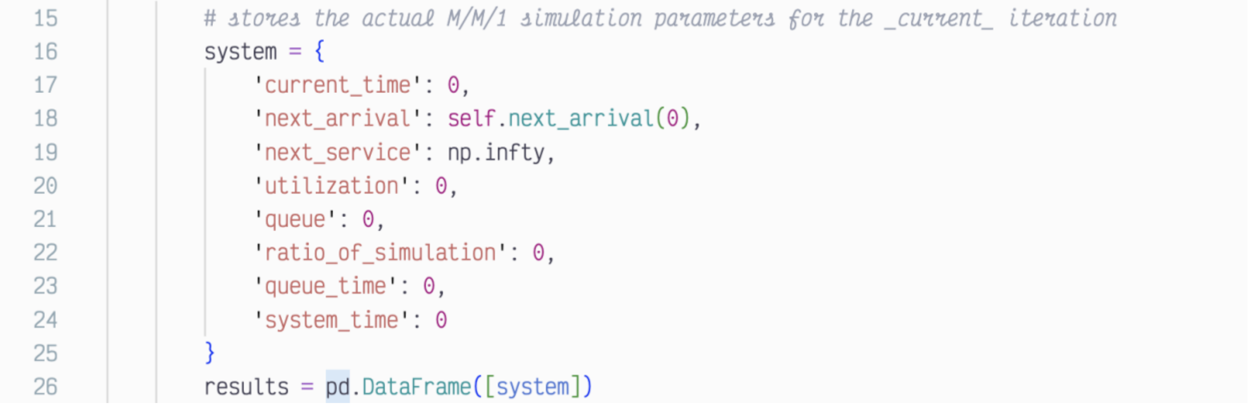


```
    # instantiate a new queue simulation.
    sim = MM1Queue(3, 4)
```

Figure #: The instantiation of a *MM1Queue* object with an inter-arrival rate of 3 and a service rate of 4.

Also included in the simulation instantiation is a definition for how the next arrival or service times are to be calculated. Doing so requires the reciprocal rates distributed exponentially. The next arrival/service times are calculated by taking the current time t and adding the arrival/service time delta. In other words, the next arrival time is $t + X$ and the next service time is $t + Y$. This is a valid implementation, as X and Y are measured in $hours\ per\ unit$ which is a direct consequence of using the reciprocal rates. Therefore, X and Y can be utilized as a time delta, given each generation of the next arrival/service time is with respect to 1 unit.

Beyond instantiation, the simulation requires several different states to be tracked at each iteration until termination. Figure # shows the state dictionary that stores the current state of the simulation system, along with a *results* table that stores each a history of the system state.



```
# stores the actual M/M/1 simulation parameters for the _current_ iteration
system = {
    'current_time': 0,
    'next_arrival': self.next_arrival(0),
    'next_service': np.infty,
    'utilization': 0,
    'queue': 0,
    'ratio_of_simulation': 0,
    'queue_time': 0,
    'system_time': 0
}
results = pd.DataFrame([system])
```

Figure #: The system initialization. The *system* variable stores the most up-to-date state of the simulation. The *results* table is a variable which stores the state of the system after each iteration as a new row and can be historically referenced during analysis of the simulation.

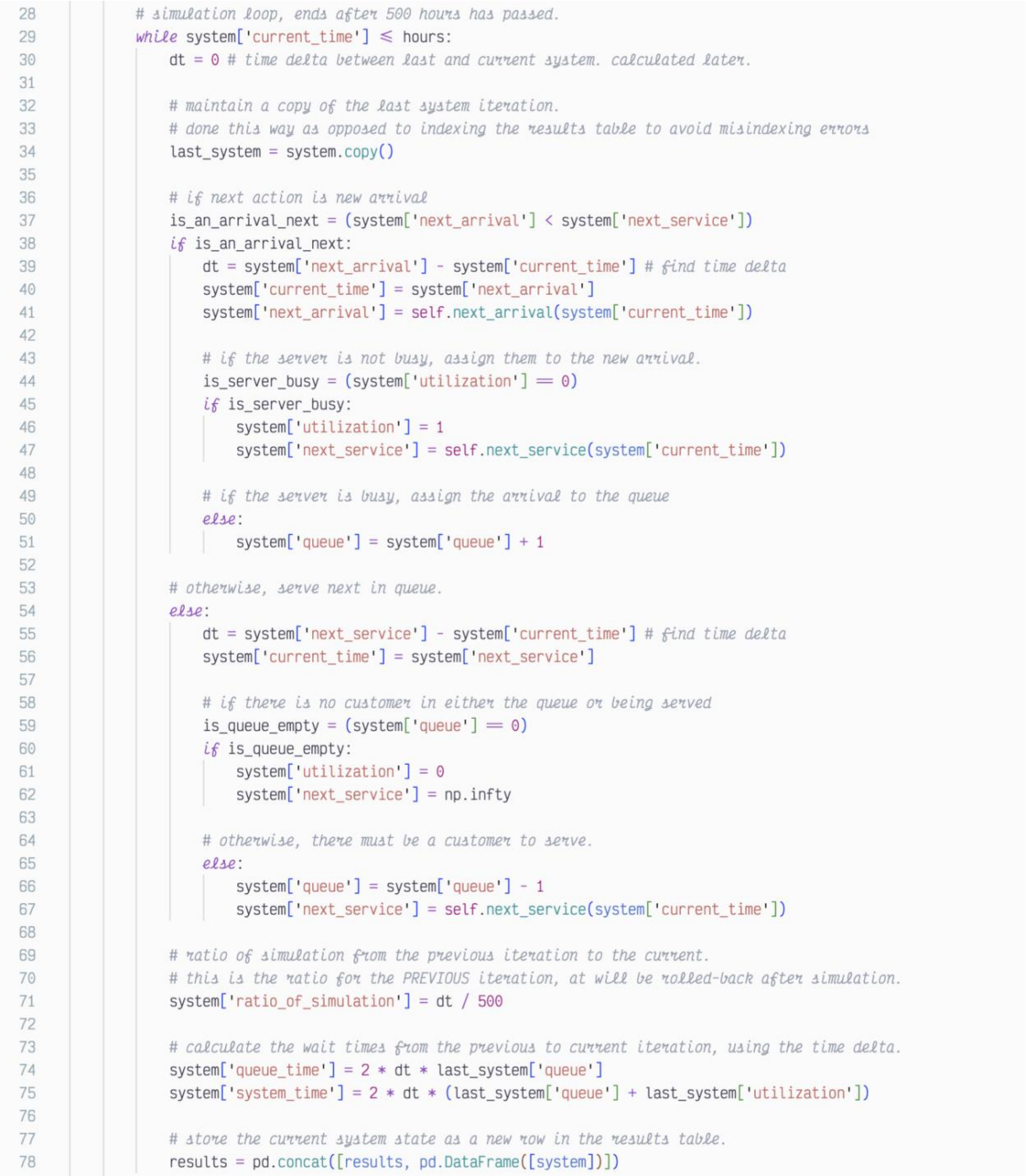
Given the parameters shown in Figure #, here is an explanation for each.

- **Current Time**: The current time of the system. The current time is calculated by finding the minimum between the next arrival and next service times from the previous simulation iteration.
- **Next Arrival**: The next scheduled time for a new unit in the queue. When a new unit enters the queue, this value is updated with $next\_arrival(current\ time)$.
- **Next Service**: The next scheduled time for a unit to leave the queue and be served. The next service time can be interpreted as *both* the time it takes for the server to finish serving the current unit and the time it takes for the next unit in queue to leave the queue; both events are equivalent and happen at the same time. When this happens, this value is updated with $next\_service(current\ time)$.

- **Utilization**: A marker for how many servers are utilized at the current time. The implementation includes only one server, so the utilization is binary. Zero utilization signifies that the server is not busy at the current iteration. One utilization signifies that the server is busy at the current iteration.
- **Queue**: The length of the queue at the current iteration. Given utilization, this parameter implies that the length of the *system* is the queue length plus the utilization.
- **Ratio Of Simulation**: A proportion constant for calculating the average parameters during analysis. This ratio is calculated by the time delta from the current time and the previous iteration time, over the total time of the simulation. This ratio signifies that the *previous* iteration makes up the state of the simulation for the ratio's proportion of the simulated time. So, a ratio of 0.1 would signify that the state of *previous* iteration is true for 10% of the simulation. After the simulation loop, the ratio of simulation will be rolled back within the *results* table history to accurately reflect the ratio for a given iteration/row.
- **Queue Time**: The wait time experienced by each member of the queue from the last iteration to the current iteration. Calculated as the time delta of the last iteration and current iteration times the current length of the queue.
- **System Time**: The wait time experience by each member of the system from the last iteration to the current iteration. Calculated as the time delta of the last iteration and current iteration, multiplied by the current length of the queue plus the current utilization.

Using these parameters, the simulation can begin with the simulation loop as shown in Figure #. This is a long loop and as such will be explained using line numbers and plaintext. The loop in Figure # can be split into two main parts, the action at *current time* and the parameter calculation. The two possible actions at *current time* are adding a unit to the queue or removing a unit from the queue. Adding a unit occurs when the next arrival time is less than the next service time. During this possible branch, the following plain-text procedure occurs:

- **Line 39.** The time delta is calculated as the next arrival time minus the previous iteration time. This delta is necessary for calculating all of the simulation outputs.
- **Line 40.** The current time is assigned the time value of the next arrival time.
- **Line 41.** A *new* next arrival time must be generated. This is done with $next\_arrival(current\ time)$ as defined during instantiation (See Figure #).
- **Lines 44 to 47.** If the server is *not busy*, then mark them as busy by setting the utilization to one. Then, assign a new next service time with $next\_service(current\ time)$.
- **Lines 50 to 51.** If the server is *busy*, then simply increment the queue size.



```
28          # simulation loop, ends after 500 hours has passed.
29          while system['current_time'] <= hours:
30              dt = 0 # time delta between last and current system. calculated later.
31
32              # maintain a copy of the last system iteration.
33              # done this way as opposed to indexing the results table to avoid misindexing errors
34              last_system = system.copy()
35
36              # if next action is new arrival
37              is_an_arrival_next = (system['next_arrival'] < system['next_service'])
38              if is_an_arrival_next:
39                  dt = system['next_arrival'] - system['current_time'] # find time delta
40                  system['current_time'] = system['next_arrival']
41                  system['next_arrival'] = self.next_arrival(system['current_time'])
42
43                  # if the server is not busy, assign them to the new arrival.
44                  is_server_busy = (system['utilization'] == 0)
45                  if is_server_busy:
46                      system['utilization'] = 1
47                      system['next_service'] = self.next_service(system['current_time'])
48
49                  # if the server is busy, assign the arrival to the queue
50                  else:
51                      system['queue'] = system['queue'] + 1
52
53              # otherwise, serve next in queue.
54              else:
55                  dt = system['next_service'] - system['current_time'] # find time delta
56                  system['current_time'] = system['next_service']
57
58                  # if there is no customer in either the queue or being served
59                  is_queue_empty = (system['queue'] == 0)
60                  if is_queue_empty:
61                      system['utilization'] = 0
62                      system['next_service'] = np.infty
63
64                  # otherwise, there must be a customer to serve.
65                  else:
66                      system['queue'] = system['queue'] - 1
67                      system['next_service'] = self.next_service(system['current_time'])
68
69              # ratio of simulation from the previous iteration to the current.
70              # this is the ratio for the PREVIOUS iteration, at will be rolled-back after simulation.
71              system['ratio_of_simulation'] = dt / 500
72
73              # calculate the wait times from the previous to current iteration, using the time delta.
74              system['queue_time'] = 2 * dt * last_system['queue']
75              system['system_time'] = 2 * dt * (last_system['queue'] + last_system['utilization'])
76
77              # store the current system state as a new row in the results table.
78              results = pd.concat([results, pd.DataFrame([system])])
```

Figure #: The simulation loop for the M/M/1 queue. The loop is split to simulate both arrival and service behavior and then calculates analysis parameters that result from these actions.

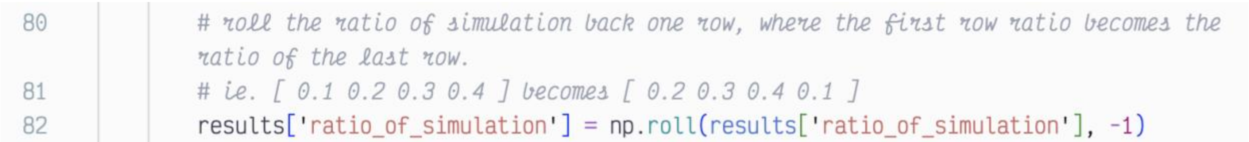
The other possible branch occurs if the first branch condition was unfulfilled. During the ‘else’ branch at starting from line 54, the following plaintext procedure occurs:

- **Line 55.** Calculate the time delta. In this case, the time delta is the next service time minus the previous iteration time.
- **Line 56.** Assign the current time with the value of the next service time.
- **Lines 59 to 62.** If the queue is empty, mark the server as *not busy* by setting the utilization to zero. Then, mark the next service time to never (set it to infinity so the next iteration is forced to be a new queue arrival).
- **Lines 65 to 67.** If the queue is *not empty,* decrement the queue length and find the next service time. This procedure represents the server retrieving the next unit for service.

After completing the iteration and executing one of the above procedures, the current iteration outputs can be calculated. The following procedure is executed after each iteration:

- **Line 71.** Calculate the ratio of simulation for the *previous* iteration by taking the time delta over the simulation period (For example, 500 hours for this project). The *previous* iteration ratio is found instead of the current, as the time the next iteration is semi-unknown. The next time *could* be found by finding the minimum of the next arrival and the next service times, but this would also require calculating more time deltas. Rolling the ratio vector back one element in the *results* table after simulation is completed seemed the better alternative with less work involved.
- **Lines 74 to 75.** The queue and system wait times for the current iteration are calculated. This is done using by multiplying the time delta by the lengths of the queue and system respectfully. The factor of 2 comes from the fact that during implementation, the results were *always* half the theoretical values, so the factor is a correction in an implementation error.
- **Line 78.** Store the results of the current iteration in the *results* table. At the end of the simulation, the *results* table will have the entire history of the simulation as a direct result of this line.

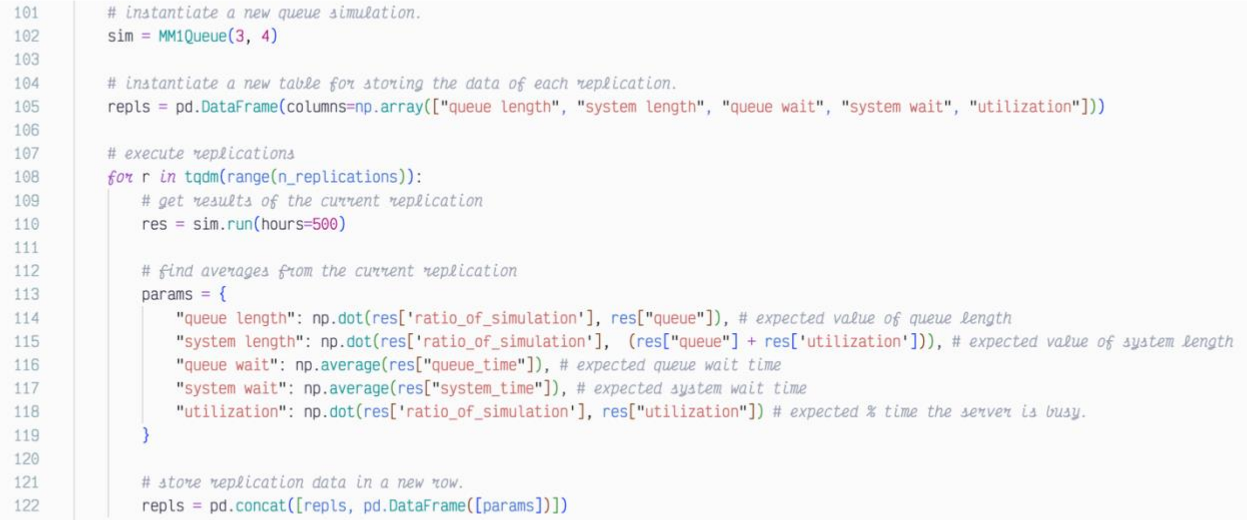
After this sequence, a new iteration begins. The simulation continues until the condition at line 29 is false; the simulation ends when the current time is greater than the desired maximum time. Once the simulation loop terminates, the code in Figure # occurs to properly align the ratios of simulation.



```
80          # roll the ratio of simulation back one row, where the first row ratio becomes the
            ratio of the last row.
81          # ie. [ 0.1 0.2 0.3 0.4 ] becomes [ 0.2 0.3 0.4 0.1 ]
82          results['ratio_of_simulation'] = np.roll(results['ratio_of_simulation'], -1)
```

Figure #: Code that performs a circular roll of the ratio of simulation column vector within the *results* table. Included in the comments is a visual of what occurs.

The simulation loop is wrapped within a $run()$ class function within the *MM1Queue* class. Therefore, multiple simulations can take place. For the purposes of simulating many replications of the simulation for an accurate representation of the queue, refer to Figure #.



```
101    # instantiate a new queue simulation.
102    sim = MM1Queue(3, 4)
103
104    # instantiate a new table for storing the data of each replication.
105    repls = pd.DataFrame(columns=np.array(["queue length", "system length", "queue wait", "system wait", "utilization"]))
106
107    # execute replications
108    for r in tqdm(range(n_replications)):
109        # get results of the current replication
110        res = sim.run(hours=500)
111
112        # find averages from the current replication
113        params = {
114            "queue length": np.dot(res['ratio_of_simulation'], res["queue"]), # expected value of queue length
115            "system length": np.dot(res['ratio_of_simulation'],  (res["queue"] + res['utilization'])), # expected value of system length
116            "queue wait": np.average(res["queue_time"]), # expected queue wait time
117            "system wait": np.average(res["system_time"]), # expected system wait time
118            "utilization": np.dot(res['ratio_of_simulation'], res["utilization"]) # expected % time the server is busy.
119        }
120
121        # store replication data in a new row.
122        repls = pd.concat([repls, pd.DataFrame([params])])
```

Figure #: Code for the replication loop of numerous simulations. In each loop iteration, a new simulation occurs, and the parameter averages are calculated. Each simulation's resulting averages are stored in the *repls* table.

From Figure #, the simulation object is instantiated as reflected by Figure #. Then, a new table variable named *repls* is created to store the averages found in each replication. At this point a loop begins and repeats for the number of desired replications. The variable *n_replications* is an integer constant for the number of desired replications, and is user inputted via the command line. The replication loop adheres to the following structure:

- **Line 110.** The next simulation begins by running the *MM1Queue.run()* function. This function contains the simulation loop code. The function returns the *results* table and is aliased as the *res* variable.
- **Line 113.** A new dictionary variable *params* is created, and the next five lines are the keys and their values that are stored within the dictionary.
- **Line 114.** The average queue length is calculated as the dot product between the ratio of simulation column vector and the queue length column vector. This is equivalent to finding the expected value of the queue length, by defining a random variable representing the queue length and distributing it by a discrete probability mass function returning the respective ratio of simulation.
- **Line 115.** The average system length is calculated as the dot product between the ratio of simulation column vector and the quantity resulting from the element-wise vector sum between the queue length column vector and the utilization column vector.
- **Lines 116 to 117.** The average queue and system wait time is calculated as the *uniform* average of the queue wait time column vector. This average must be

calculated uniformly instead of with the ratio of simulation because the average wait time is dependent on varying time deltas, and the ratio of simulation would remove this.

- **Line 118.** The average utilization is calculated as the dot product between the ratio of simulation column vector and the utilization column vector.
- **Line 122.** Store the current replication average parameters within the *repls* table, so further analysis can be performed after all replications are complete.

With all the preceding code discussed, an M/M/1 queue can be simulated for any number of replications. The more analysis-oriented code will be discussed in the Analysis portion of the report, but for now the discussion will move to the AnyLogic implementation. There is not too much to discuss regarding the AnyLogic implementation of the simulation, as it is nearly identical to what was demoed in class. Figure # shows the node sequence utilized for simulating an M/M/1 queue. Furthermore, the experiment in Figure # takes the outputs from Figure # and replicates that experiment for 30, 100, and 1000 replications to find more accurate output averages. The outputs of the experiments will be compared to the Python implementation in the Analysis section.

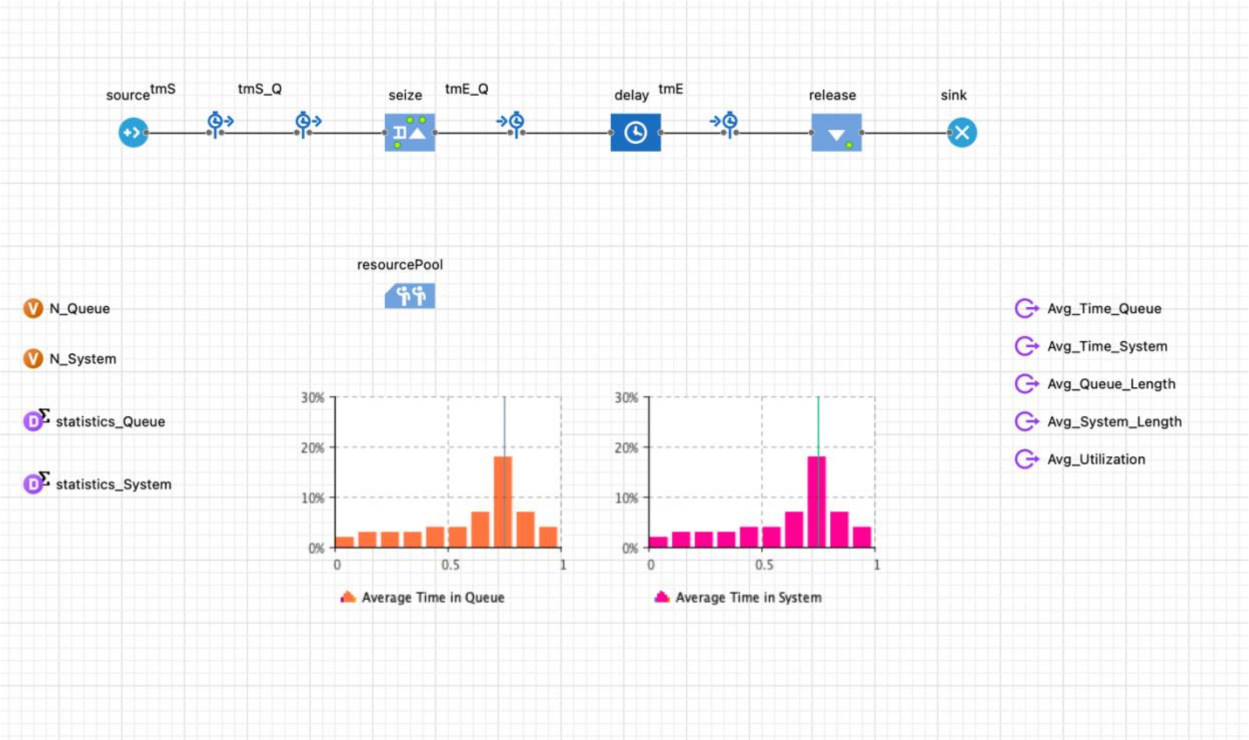


Figure #: The AnyLogic simulation experiment for a M/M/1 queue. The source has an interarrival rate of 3 units per hour and the delay represents the service time with a rate of 4 units per hour. The resource pool contains one server, and is seized by the seize node. This experiment tracks the size of the queue/system, the wait times in the queue/system, and outputs the average wait times and lengths of the queue/system and the average utilization of the server.

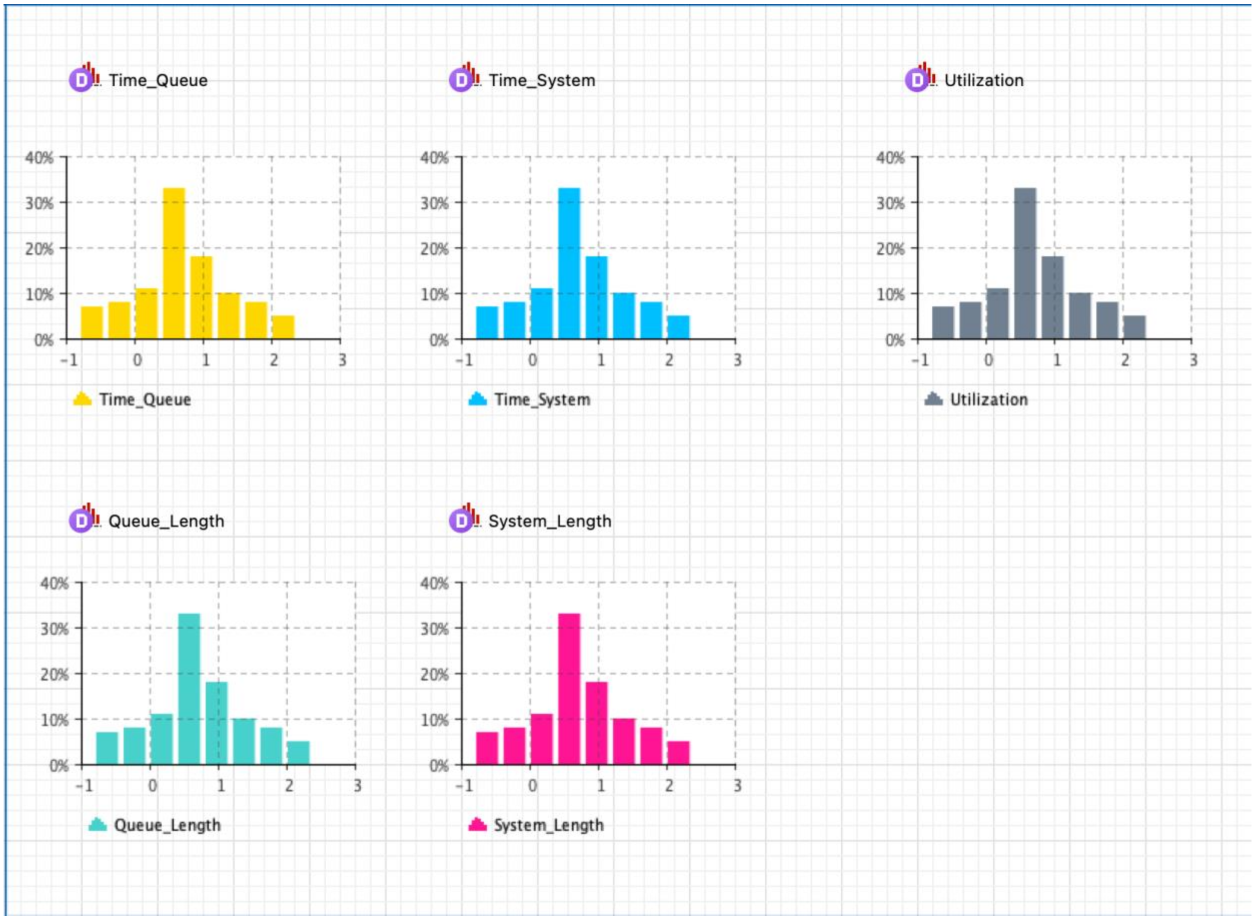


Figure #: The AnyLogic replication experiment for numerous M/M/1 queue simulations. This experiment takes as input the outputs of the experiment in Figure # and distributes them along a histogram and calculates the averages of the outputs for many replications.

Analysis Implementation

To properly analyze the results of the Python implementation, the theoretical queue parameters must be found. Given an interarrival time $\lambda = 3$ and a service time $\mu = 4$, the average queue length L_q, average system length L_s, average queue wait time w_q, average system wait time w_s, and average server utilization ρ can be found utilizing the M/M/1 queue parameter formulas defined in (Hillier, 2021, pp. 751-753).

$$L_q = \frac{3^2}{4(4-3)} = \frac{9}{4} = 2.25;$$

$$L_s = \frac{3}{4-3} = 3;$$

$$w_q = \frac{3}{4(4-3)} = \frac{3}{4} = 0.75;$$

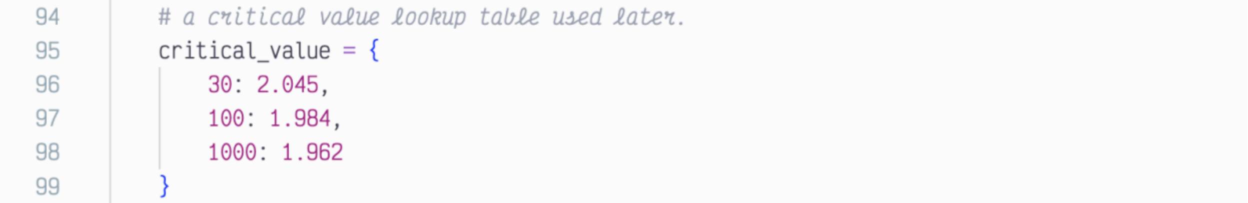
$$w_s = \frac{1}{4-3} = 1;$$

$$\rho = \frac{3}{1(4)} = 0.75.$$

To statistically compare these theoretical parameters to the Python parameters, the confidence intervals of each must be found. Let $\vec{m}$ be a vector corresponding to each parameter average of all replications from the Python implementation, where each element is the average of a different parameter. Let $\vec{v}$ be a vector corresponding to the variance of each different parameter from all replications. The unconventional notation is utilized to better reflect the analysis code shown later. Let n be a scalar of the number of replications executed. Finally, let t_n be the scalar critical value as described in a t-distribution look up table for n replications. Then, the confidence interval is provided by the following equation using element-wise arithmetic.

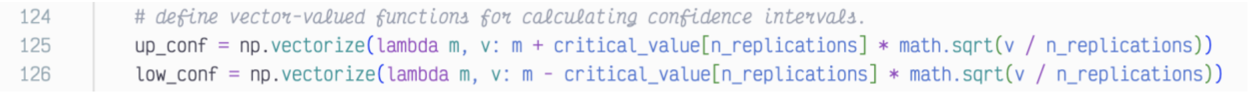
$$\vec{m} \pm t_n \sqrt{\frac{\vec{v}}{n}}.$$

For the purposes of this project, assume a 95% confidence interval such that the critical values are $t_{30} = 2.045$, $t_{100} = 1.984$, and $t_{1000} = 1.962$ for the provided subscripted number of replications. These values are then stored in Python as a lookup table in Figure #. Furthermore, functions for the lower and upper bound confidence intervals are defined in Figure #. These functions are equivalent to the formula above, where $up\_conf(m, v)$ utilizes addition and $low\_conf(m, v)$ utilizes subtraction.



```
# a critical value lookup table used later.
critical_value = {
    30: 2.045,
    100: 1.984,
    1000: 1.962
}
```

Figure #: A lookup table for the critical values for each relevant number of replications for analysis.

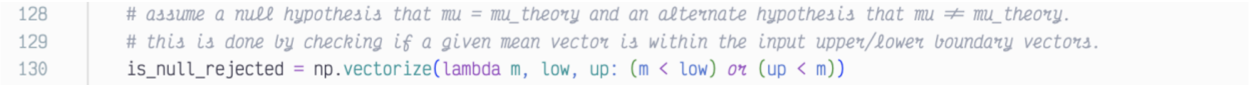


```
# define vector-valued functions for calculating confidence intervals.
up_conf = np.vectorize(lambda m, v: m + critical_value[n_replications] * math.sqrt(v / n_replications))
low_conf = np.vectorize(lambda m, v: m - critical_value[n_replications] * math.sqrt(v / n_replications))
```

Figure #: The vector-valued functions to calculate the confidence intervals for each experiment parameter average.

The resulting confidence intervals can be used to statistically compare the Python implementation means to the theoretical and AnyLogic means. If the theoretical or

AnyLogic parameters are within the confidence interval, then they must be statistically equal. Suppose the null hypothesis H_0 that the theoretical/AnyLogic mean is equal to the Python mean and let the alternative hypothesis H_A be the negation; the means are not equal. The function in Figure # attempts to reject the null hypothesis by taking a vector of either theoretical or AnyLogic means, and the vector confidence intervals of the Python results; the output of the function is a Boolean vector corresponding to whether the hypothesis was rejected for each parameter.

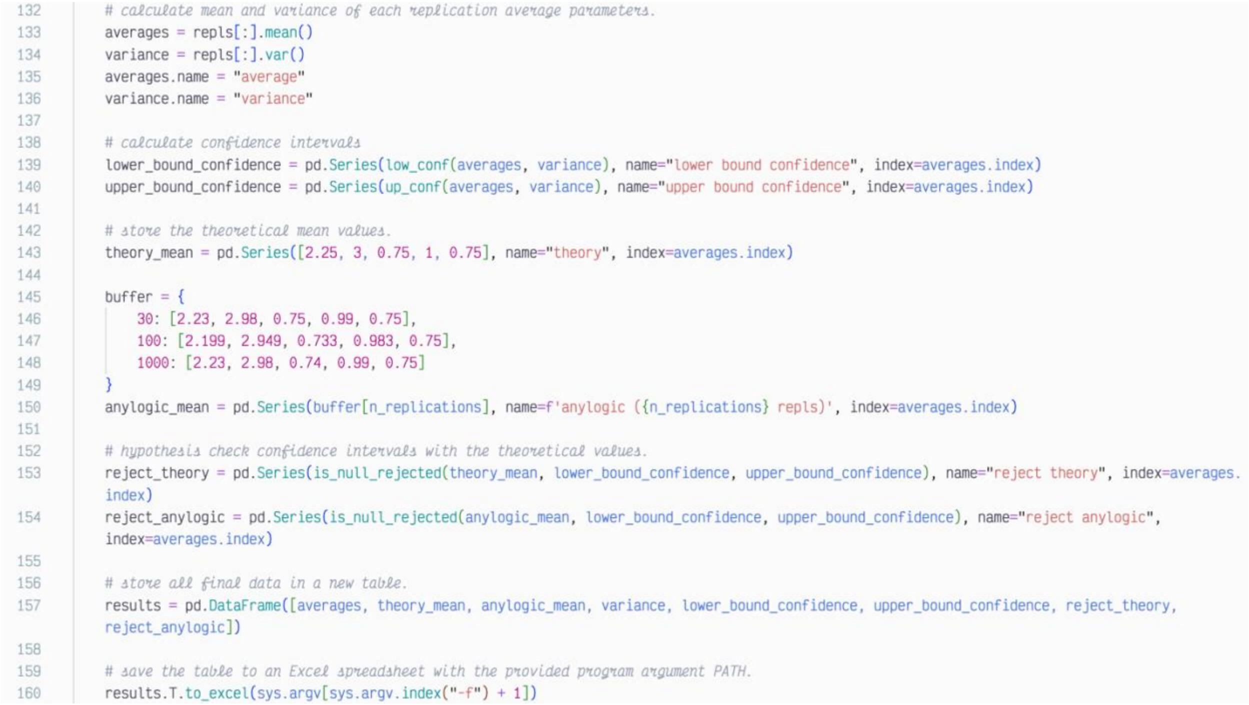


```python
# assume a null hypothesis that mu = mu_theory and an alternate hypothesis that mu ≠ mu_theory.
# this is done by checking if a given mean vector is within the input upper/lower boundary vectors.
is_null_rejected = np.vectorize(lambda m, low, up: (m < low) or (up < m))
```

Figure #: The function $\mathrm{is\_null\_rejected(m, low, up)}$ checks elementwise if the mean vector is within a given confidence interval, and returns a vector of the resulting Boolean values, where a value 0 is a failure to reject and a value 1 is a rejection of the null hypothesis.

At this point, the analysis is conducted from Figure #. The procedure follows:

- **Lines 133 to 136.** The mean and variance with respect to all replications of all parameters from each simulation is found. The *averages* and *variance* variables are vector valued and correspond to the average/variance of each simulation output parameter.
- **Lines 139 to 140.** The confidence intervals are found for each simulation output parameter.
- **Line 143.** A column vector is created to store all the theoretical queue outputs.
- **Lines 145 to 150.** A lookup table is created called *buffer* which contains the results of the AnyLogic implementation after 30, 100, and 1000 replications. Using the number of replications, this lookup table is accessed to find the proper values to statistically compare to the Python implementation.
- **Lines 153 to 154.** Statistically compare the theoretical and AnyLogic parameters with the Python implementation results.
- **Line 157 to 160.** Store all the final data into a table and output the table to an excel spreadsheet. This data includes the Python mean/variances, theoretical and AnyLogic means, the confidence intervals, and the hypothesis testing results.



```python
# calculate mean and variance of each replication average parameters.
averages = repls[:].mean()
variance = repls[:].var()
averages.name = "average"
variance.name = "variance"

# calculate confidence intervals
lower_bound_confidence = pd.Series(low_conf(averages, variance), name="lower bound confidence", index=averages.index)
upper_bound_confidence = pd.Series(up_conf(averages, variance), name="upper bound confidence", index=averages.index)

# store the theoretical mean values.
theory_mean = pd.Series([2.25, 3, 0.75, 1, 0.75], name="theory", index=averages.index)

buffer = {
    30: [2.23, 2.98, 0.75, 0.99, 0.75],
    100: [2.199, 2.949, 0.733, 0.983, 0.75],
    1000: [2.23, 2.98, 0.74, 0.99, 0.75]
}
anylogic_mean = pd.Series(buffer[n_replications], name=f'anylogic ({n_replications} repls)', index=averages.index)

# hypothesis check confidence intervals with the theoretical values.
reject_theory = pd.Series(is_null_rejected(theory_mean, lower_bound_confidence, upper_bound_confidence), name="reject theory", index=averages.index)
reject_anylogic = pd.Series(is_null_rejected(anylogic_mean, lower_bound_confidence, upper_bound_confidence), name="reject anylogic", index=averages.index)

# store all final data in a new table.
results = pd.DataFrame([averages, theory_mean, anylogic_mean, variance, lower_bound_confidence, upper_bound_confidence, reject_theory, reject_anylogic])

# save the table to an Excel spreadsheet with the provided program argument PATH.
results.T.to_excel(sys.argv[sys.argv.index("-f") + 1])
```

Figure #: The analysis code and construction of a table output of the resulting hypothesis testing and actual mean/variance of each parameter.

Analysis

The following page contains three tables that are generated from the Python analysis implementation. Table 1 shows the output results of 30 replications. Table 2 shows the output results of 100 replications. Finally, Table 3 shows the output results of 1000 replications. All analysis and discussion in this section relates to these three tables.

From the statistical analysis, for 30 and 100 replications, the simulated queues are equivalent to the theoretical and AnyLogic queues. The confidence intervals at 30 and 100 replications is wide enough to consistently contain the theoretical and AnyLogic parameters. The problem arises at 1000 replications. Despite the tests in Table 3 showing a failure to reject any null hypothesis, that is not always the case. When the number of replications increases, the confidence interval decreases. From Table 3 the lower and upper bound of the confidence intervals produce a very narrow interval for the hypothesis test. This is equivalent to saying there is more opportunity for the null hypothesis to be rejected. For example, the average system wait could be rejected with respect to the theoretical null hypothesis if there were less significant figures or if there was more floating-point error. There were many attempts when generating Table 3 that resulted in the theoretical and AnyLogic utilization test being rejected due to a narrow confidence interval. The same also occurred for the theoretical queue/system wait tests.

	Average	Theory	AnyLogic (30 repls)	Variance	Lower Bound Confidence	Upper Bound Confidence	Reject Theory	Reject AnyLogic
Queue Length	2.357953744	2.25	2.23	0.729383	2.039085985	2.676821504	0	0
System Length	3.1054032	3	2.98	0.756547	2.780652028	3.430154372	0	0
Queue Wait	0.781419106	0.75	0.75	0.069333	0.683107833	0.87973038	0	0
System Wait	1.030335093	1	0.99	0.07	0.931552359	1.129117826	0	0
Utilization	0.747449456	0.75	0.75	0.000679	0.737720497	0.757178414	0	0

Table 1: The results of running the Python M/M/1 queue for 30 replications. The experiment provides statistically equivalent results to the theoretical and AnyLogic outputs.

	Average	Theory	AnyLogic (100 repls)	Variance	Lower Bound Confidence	Upper Bound Confidence	Reject Theory	Reject AnyLogic
Queue Length	2.273900297	2.25	2.199	0.241368	2.17642793	2.371372663	0	0
System Length	3.021298916	3	2.949	0.262922	2.919567539	3.123030293	0	0
Queue Wait	0.757236921	0.75	0.733	0.023319	0.726940058	0.787533783	0	0
System Wait	1.006740667	1	0.983	0.024462	0.975710431	1.037770903	0	0
Utilization	0.747398619	0.75	0.75	0.000789	0.741825586	0.752971653	0	0

Table 2: The results of running the Python M/M/1 queue for 100 replications. The experiment provides statistically equivalent results to the theoretical and AnyLogic outputs.

	Average	Theory	AnyLogic (1000 repls)	Variance	Lower Bound Confidence	Upper Bound Confidence	Reject Theory	Reject AnyLogic
Queue Length	2.228835	2.25	2.23	0.262823	2.197028	2.260643	0	0
System Length	2.978006	3	2.98	0.284219	2.944929	3.011083	0	0
Queue Wait	0.741052	0.75	0.74	0.025823	0.731082	0.751022	0	0
System Wait	0.990756	1	0.99	0.026998	0.980561	1.00095	0	0
Utilization	0.749171	0.75	0.75	0.000715	0.747512	0.75083	0	0

Table 3: The results of running the Python M/M/1 queue for 1000 replications. The experiment provides statistically equivalent results to the theoretical and AnyLogic outputs.

It is interesting to note that the increasing probability of a null hypothesis rejection at high replications, cannot completely erased due to the random nature of the results. There must be some replication quantity such that if α is the theoretical expected parameter value, then the confidence interval $I = \{\alpha\}$. A failure to reject the null hypothesis with such a confidence interval would require the simulated mean to be numerically equal to α. Based on the nature of computational simulation, it is impossible for this to be the case, due to floating point error. Therefore, as the number of replications approaches infinity, there is a point where the null hypothesis will *always* be rejected. Due to this property, a properly implemented simulation would be best suited for a “moderate” number of replications. This “moderate” number of replications may vary depending on the application and the level of accuracy to the theoretical values that is required. For the purposes of the current implementation, 100 replications appear as the best balance between accuracy for the current application and the time to execute all replications. While times may vary based on the machine utilized, the target machine for this code runs 30 replications in 13 seconds, 100 replications in 44 seconds, and 1000 replications in 7 minutes and 30 seconds.

Conclusion

The statistical experiment of comparing the Python queue implementation with theoretical and industry standard simulation software show both how a queue simulation works at a procedural level and exhibits why a hand-written implementation is not practical. The implemented Python simulation consistently fails to reject the average parameter equivalence null hypotheses for 30 and 100 replications. In contrast, 1000 replications does reject the null hypothesis at decent amount of the time. This shows that the Python simulation is a valid implementation if large replication quantities are not required by a given application. The Python implementation is a good educational tool for a glimpse into how more professional tools may be implemented. But, for all practical applications, an AnyLogic simulation is the more efficient and accurate simulation to utilize.

References

Moore, J. (2024, April 28). A M/M/1 queue simulation implementation in python using Numpy/Pandas. GitHub. https://github.com/mjalen/mm1-queue

Hillier, F. S. & Lieberman, G. J. (2021). Introduction to operations research (11th edition). New York, NY: McGraw-Hill.