Queueing in the Grocery Store

Minerva University

CS166: Modeling and Analysis of Complex Systems

Prof Ribeiro

Jan 26, 2023

Table of Contents

Problem Identification

Queuing theory studies formations and congestion of people and objects in queues. By analyzing various aspects of waiting lines, one can optimize for service time, queue length, or the number of people. One prominent application of the theory is a queueing system in grocery stores, whereby customers join a queue to be served by one of the cashiers after shopping. Compared to traditional queuing theory, context-specific assumptions and rules govern the system.

This report is guided by the question of what the optimal number of cashiers in a grocery store is by considering both the mathematical side of queuing theory, building a simulation to model the context and comparing the results. The report explores a system of $M/G/1 * n$, where n is the number of queues.

Variables

The following variables are part of the model:

- Time of day (measured in minutes since 9 am).
- Number of customers served at each cashier.
- Number of customers per queue.
- Number of customers served by the store manager.
- Number of customers at the store manager queue.
- Total number of customers who finished shopping and left the grocery store.
- Number of queues and cashiers available at each time stamp.

Update Rules

The model follows the following update rules:

1. The store opens at 9 am and closes at 8 pm. After 8 pm, no new customers can join the queue. Any customer already in the queues is served until all queues are empty.
2. Customers choose to join the shortest queue and cannot switch queues.
3. When a cashier is done serving a customer, the customer leaves and the cashier starts serving the next customer.
4. After being served, a customer may join a single manager queue with a 5% probability as an extended service.
5. Arrival times are independent and follow the same exponential probability distribution.

Assumptions

To model this system, we have made the following assumptions:

1. All queues operate according to the first in, first out principle.
2. Cashiers serve a single customer at a time.
3. There are a maximum of 10 cashier queues and 1 manager queue.
4. The arrival times to queues are exponentially distributed $Exp(\lambda = 1\, per\, minute)$ and independent.
    a. An important property of the exponential distribution is that states are memoryless, as only the previous state and no history of past transition states are needed to derive the successive states of the system.
5. Cashier serving times are normally distributed $N(\mu_1 = 3\, minutes,\ \sigma_1 = 1\, minute)$.

a. Servers' times to serve customers are independent since the sample of serving times are drawn from the same distribution, assuming constant productivity and no break time between customers.

6. The time the manager takes to handle requests is normally distributed $N(\mu_2 = 5\,minutes,\ \sigma_2 = 2\,minutes)$.

The final results of the report are supported by the theoretical derivation of key metrics, such as average customer waiting and service times and queue lengths, empirical simulation of the model, and critical evaluation of its accuracy and feasibility relative to the theoretical values.

Analysis

In this section of the paper, we analyze the theoretical implications of queuing theory by applying and modifying some of the formulas.

It is important to make a side note here about the theoretical analysis. Given that these formulas only hold under certain conditions, there are certain regimes in the parameter space that we should simulate. Firstly, the formulas only hold when utilization, ρ, is less than 1 such that the average arrival time is less than the average service time, and the queue doesn't keep growing. Since the utilization is $3/n$, n must be greater than 3 (more than 3 servers) for the calculations to be valid. We would thus expect the queues to grow on average when we have 1, 2, or 3 servers. We must also have a mean service time of greater than 0, since this would result in errors in our calculations (division by 0).

Most theoretical results in this section are directly compared with empirical results obtained by running our simulation (see code in Appendix A). Something important to note for the empirical results is that we only ran 100 trials per condition in each experiment simply because the time to produce them was too long. Ideally, each experiment would have at least 500 trials to represent the average behavior of the system more accurately but given the constraint, running only 100 trials was sufficient for this report.

Average Waiting Time for Cashier Queues

Though the arrival rate to the overall system is 1 customer/minute, we must taken into account that we are modeling each queue as an M/G/1 system, and thus we require a parameter λ for the arrival to each queue. Though the customers actually join the shortest queue that they arrive to, which is a violation of the assumptions of queueing theory, we will model the arrival rate to each queue here as $1/n$, where n is the number of servers. The average service time for a cashier, τ, is 3, the variance of service time for a cashier, δ^2, is 1, we can calculate the utilization, ρ, of the cashiers by multiplying the average arrival rate and service time to get $3/n$.

Using these variables, we can then calculate the average waiting time for a cashier, which is given by the following formula for $M/G/1$ queues according to queuing theory:

$$Average\ waiting\ time\ = \frac{\rho\tau(1+\delta^2/\tau^2)}{2(1-\rho)}$$

Substituting in the variables for our system, we get

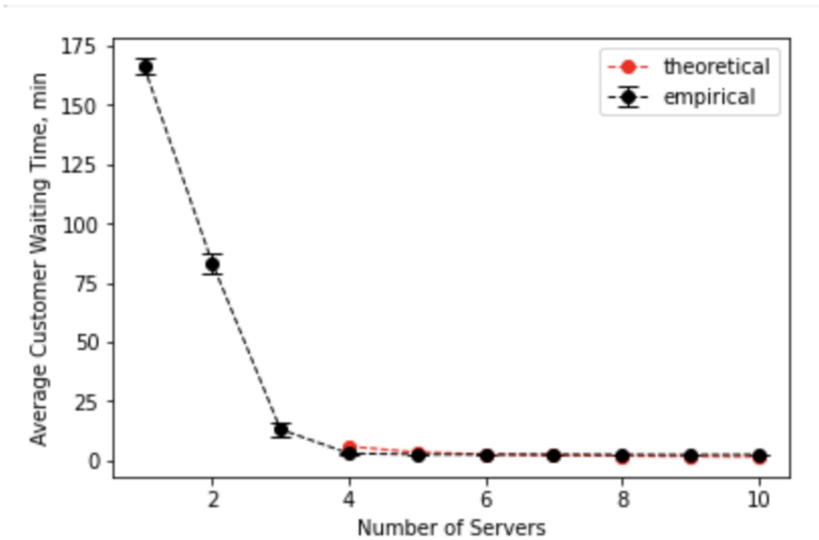
$$\frac{(3/n)(3)(1+1/3^2)}{2(1-3/n)} =$$

$$= \frac{5}{n-3}\ \text{minutes}$$

Thus, we expect the average waiting time for the cashiers to be dependent on the number of servers n in the system.

We obtained data by running our simulation (reference code in Appendix A) and visualized it in the plot below.

Figure 1. Number of servers vs Average waiting time.



Note: The line plot with error bars shows how the average customer waiting time decreases as the number of cashiers is increased.

As we can see from Figure 1, the empirical results from the simulation are close to those of the theoretical formula for the average waiting time as we vary the number of servers. The theoretical predictions (red line) start at n=4 because this is the point where the mathematical formulas become valid so that the utilization value is smaller than 1. The error bars, representing the 95% confidence interval, are incredibly narrow, which makes sense because there were 100 trials and each trial ran for 500 minutes. Even though the simulation is stochastic since we run them for so long, the result represents the average behavior of the system, so the data points are close together. This, together with the fact that we can see that the empirical and theoretical results closely match, are both proofs verifying that our model works as expected and accurately models the grocery store's system.

Let's analyze Figure 1 closely. If we look at the trend, having fewer than 3 servers is not ideal because the average waiting customer time would be more than 75 minutes. The point at 3 servers points to approximately 12 minute waiting time on average, which is acceptable. Even better would be having 4 as the point is closer to 0. After the number of servers reaches 4, the trend remains stable at approximately the same value. This implies that employing more than 4 cashiers would be counterproductive and a waste of resources for the grocery store.

Average Waiting Time for Store Manager

We can also calculate the average waiting time for the store manager. Here, we make the assumption that the arrival rate to the manager queue is 5% of the arrival rate to the cashier queues and is governed by the exponential distribution, in line with all queueing theory models. In reality, this assumption might be incorrect since customers have already queued and been served at the cashier desk before arriving at the manager queue. Given that the arrival rate, λ_M, is 0.05, the average service time for the manager, τ_M, is 5, and the variance of service time for the manager, δ_M^2, is $2^2 = 4$, we can calculate the utilization of the store manager by multiplying the average arrival rate and service time to get 0.25. Using the same formula as above, we can calculate the average waiting time for the store manager as:

$$Average\ manager\ waiting\ time\ =\ \frac{(0.25)(5n)(1+4/5^2)}{2(1-0.25)} =$$

$$= \frac{29}{30} = 0.97 \text{ minutes}$$

Average Response Time for Cashier

Using the average waiting times, we can also calculate several other useful metrics for determining the efficiency of the system. These theoretical results will serve as a comparison for our empirical results from the simulation in the next part of the report. Given that the assumptions made here and the assumptions made by our simulation do not perfectly align, we might expect our estimates to differ from the simulated values, however they will give us a good ballpark estimate to ensure that both our theoretical and experimental values are reasonable, and to give us a good idea of regimes in the parameter space to test on the simulation.

The average response time, the time it takes for a customer to wait in line and to be served by a cashier (not including any time waiting or being served by the manager), would be given by:

$$Average\ response\ time\ = \tau + average\ wait\ time\ =$$

$$= 3 + \frac{5}{n-3} \text{ minutes}$$

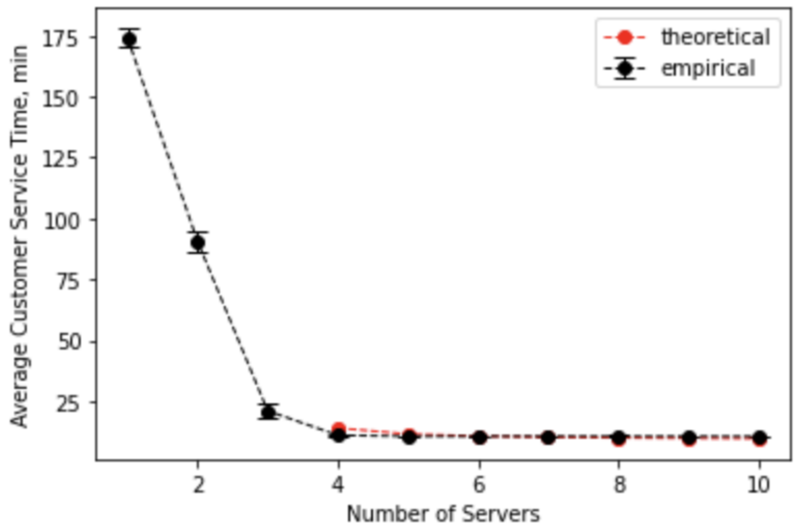
Average Response Time Considering Manager

In reality, however, the customer has a 0.05 chance of seeing the store manager as well, meaning the average time a customer spends waiting and being served in the system will likely be greater than the theoretical result above.

We can thus calculate the average response time for the manager. This is given by the same formula, $\tau_M + average\ wait\ time\ for\ manager$, which is equal to 5.97 minutes.

Now let's compare the theoretical results of the average response time with empirical data obtained by running the simulation.

Figure 2. Number of Servers vs Average Customer Response Time of Cashier and Manager Servers.



Note: The line plot with error bars shows how the average time of the servers decreases as the number of cashiers is increased.

Similar to Figure 1, in Figure 2 the 95% confidence intervals are narrow due to the same reasons mentioned in the analysis of Figure 1 (high runtime and number of trials). In Figure 2, the empirical results also match theoretical ones. When calculating the service times, we included the service time of both the cashiers and the manager rather than leaving the manager out. We chose to include the manager, because it better represents the whole grocery store system and, therefore, more accurately captures the scenario.

As we look further into what Figure 1 suggests, having fewer than 3 cashiers seems unrealistic as the average service time is approximately at least 100 minutes. In real life, no customer would wait for more than an hour and a half to pay. Having 3 cashiers is significantly better (25 minute server time) but having 4 is even better. After the number of servers reaches 4 we can see the trend plateauing at around 12 minutes. This suggests that employing more than 4 would have a minimal impact.

Average Queue Length

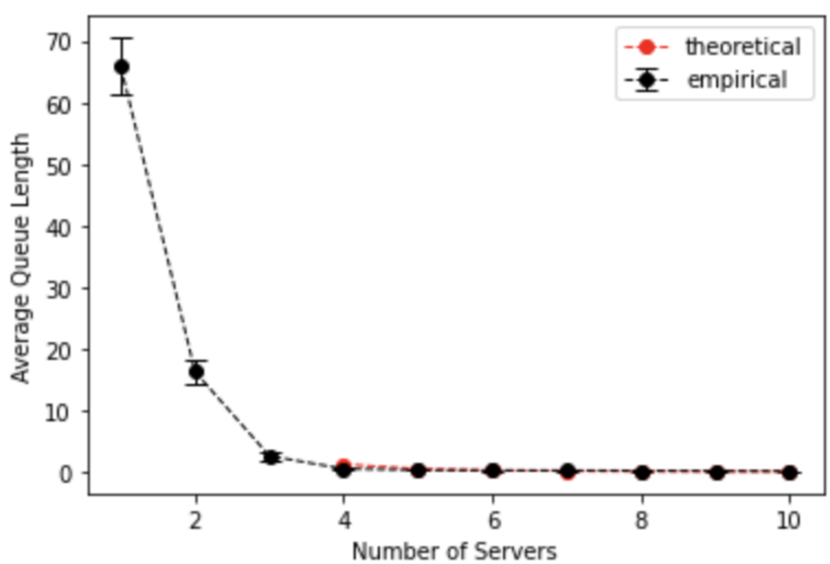
We can also calculate an estimate for the average queue length (per cashier queue) for the system:

$$Average\ queue\ length\ = \lambda .\ average\ wait\ time\ =$$

$$= \frac{5}{n(n-3)} \text{ people}$$

Let's directly compare this result with data generated by the Python implementation.

Figure 3. Number of servers vs Average queue length.



Note: The line graph demonstrates how the average queue length decreases as the number of cashiers is increased.

Figure 3 describes the relationship between increasing the number of servers, which, when increased, leads to a smaller average queue length. The results of the simulation and the graph point to a similar result as the previous graphs – that an optimal number of servers would be again 4 cashiers.

Average Number of Customers per Cashier Queue and Service Desk

Finally, the average number of customers per cashier queue and service desk is given by:

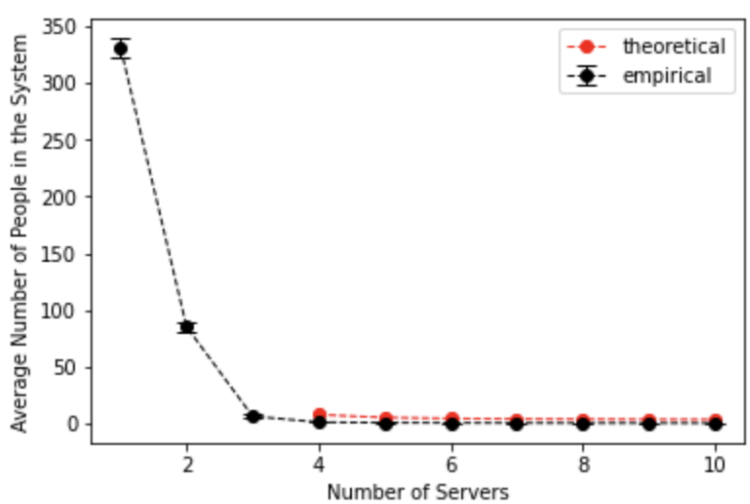
$$\lambda .\ average\ response\ time\ =$$

$$= \frac{3}{n} + \frac{5}{n(n-3)}$$

Similar to response time, this value does not include waiting or being served by the store manager, since this is not a part of queueing theory. Thus, in reality we would expect to have a greater average number of people in the system than $3 + 5/(n-3)$, since there will be individuals waiting for and being served by the manager.

Let's compare these results with the simulation.

Figure 4. Number of servers vs Average number of people in the system.



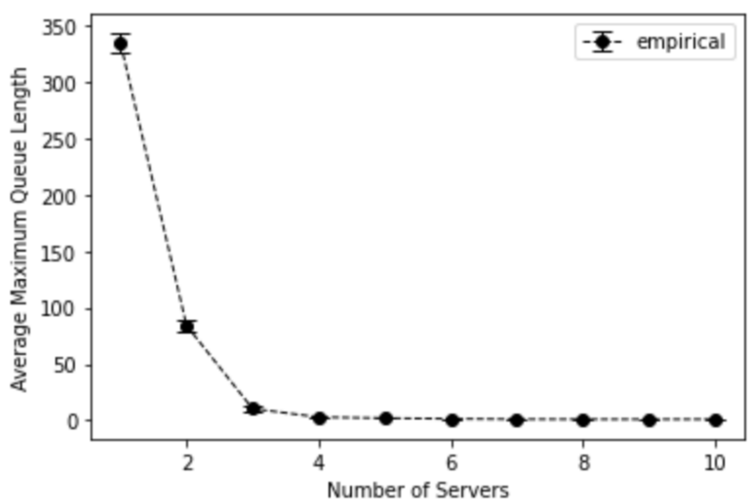
Note: The line graph indicates that increasing the number of servers leads to a decrease in the average number of people in total in the system.

As shown in Figure 4, the trend between the number of servers (independent variable) and the dependent variable (average number of people in the system) remains the same as in the previous plots. The empirical results also closely match those of the theoretical formulas, which is one more indicator that the model of the grocery store works as intended. Looking at the graph trend, similar to before, we can notice that 3 cashiers are a good enough total number of servers. Here, 4 servers are still slightly better but not significantly.

Average Maximum Queue Length

In this section, we considered an additional metric, average maximum queue length, not accounted for by the theoretical formulas. It is an important metric because it can pinpoint if there are any regimes according to the number of servers where bottlenecks form for the queues.

Figure 5. Number of Servers vs Average maximum queue length.



Note: The line plot with error bars shows how the maximum queue length decreases as the number of cashiers is increased.

As shown in Figure 3, as the number of cashiers increases, the maximum queue length decreases. Each trial ran for 500 minutes representing almost the whole day the grocery store would operate, and then the maximum queue length considering all queues was taken. As we can see, having fewer than 3 cashiers is not optimal because that makes the maximum queue length close to 100, which is unreasonable to maintain in real life. Therefore, the plot suggests that at least 3 or better 4 cashiers should be employed. After the servers reach 4, the graph plateaus at about a length of 1 customer. So the plot also suggests that employing more than 4 cashiers would be a waste of resources as it would not impact the customer experience much.

Conclusion

After the theoretical and empirical analysis, both point out that the optimal number of servers would be 4 in total. Having 3 cashiers would also be permissible if the grocery store would like to save up on some resources (space, staff, money), but having 4 would optimize the customer experience. Having more than 4 cashiers might lead to a minimal improvement, but the trend after 4 plateaus shows that hiring more cashiers would be useless and less profitable. The paper suggests employing 3 cashiers. However, it has the preceding limitations. These are our suggestions to improve the analysis in the future.

Furthermore, when there are 4 servers, the average waiting time is approximately 10 minutes which is not too far off from 10 servers, about a 2 minute difference; the response time with 4 servers is 13 minutes and 10 minutes with 10 servers; and the average queue length is approximately 3 customers with 4 servers and 1.5 with 10 servers. Ultimately, having 4 servers is the optimal point that significantly improves the average customer experience compared to having fewer servers while also maintaining a realistic expected value for the metrics discussed above and not wasting unnecessary resources, including money, space, and additional staff members.

Word count: 2643

Bibliography

Martelli, (2022). How do I make a flat list out of a list of lists?. *StackOverflow*. Retrieved from: https://stackoverflow.com/questions/952914/how-do-i-make-a-flat-list-out-of-a-list-of-lists

Ribeiro, (2022). Session 3 - Implementing Simulations. *Forum*. Retrieved from: https://forum.minerva.edu/app/courses/2571/sections/10241/classes/70754

LO Applications

#EmpiricalAnalysis

The data for the empirical analysis was obtained from the simulation that we implemented in Python. The results are visualized using appropriate plots (e.g., an error plot, a histogram with the confidence interval, and mean). We effectively compared the empirical and theoretical results by discussing the related ones closely in the report and also plotting the theoretical results in the same graphs as the empirical to make the comparison easier to follow. We considered 5 variables for the empirical analysis to answer best the explanatory challenge posed from the beginning. We also justified why each of these variables is most useful to explore to answer the question.

Word count: 108

#CodeReadability

We have used appropriate variable names that describe what they are for, docstrings for all methods and functions, and in-line code comments where necessary. We have also organized the code into classes that perform distinct functions (as explained in the code implementation section of the paper). We have used neither magic numbers nor non-constant global variables in our work to increase the understanding of the code and to ensure that a tiny change in the code doesn’t lead to unexpected results. The code is structured in OOP, which makes it easy to follow. Even the analysis is performed by its class, namely the EmpiricalAnalyst. We have also added different sections and titles in the Jupyter Notebook to make it easy to follow.

Word count: 122

#Modeling

We created a rigorous model based on queuing theory that we applied by programming a simulation to the context of a grocery store to answer an explanatory challenge: what is the optimal number of cashiers. We explain and justify all assumptions of the model in the first part of the paper (in the bullet points) as well as throughout the analysis, where we point out why some of the theoretical formulas are not perfect approximations of the behavior since our model breaks some of their assumptions (i.e. in cases where the utilization is smaller than 1, when we have to account for the manager). Nevertheless, the model is a good approximation of the grocery store's overall behavior, allowing us to explore the initial question. We also explain all relevant rules (i.e., store working times), variables (i.e., customers, cashiers), and parameters (i.e., the parameters of the service distributions). We also clearly related queuing theory to the context of queues in a grocery store. After implementing the model, we used the simulation to generate data, plot it, and explore the graphs, comparing them with the theoretical analysis to more effectively answer the explanatory challenge.

Word count: 196

#PythonImplementation

We implemented a working simulation of the grocery store model that follows all assumptions, variables, and rules described in our report's introduction. We used appropriate data structures throughout. For example, we used a heap as the priority queue for each cashier and manager queue to improve efficiency with the help of the in-built Python heappop and heappush methods. The state of the simulation is currently only visualized with the help of print statements. They are sometimes hard to read when there are many of them, which is a possible improvement. We also implemented simple test cases (in the Test Cases subsection of the Jupyter notebook) to verify that the main parts of the code work as expected. All required materials to run the code are submitted as a separate file to this report and can be recreated by anyone.

Word count: 139

#TheoreticalAnalysis

M/G/1 * n model was chosen to explain queue dynamics in the grocery store. We acknowledged the model's assumptions and derived several metrics to explain how the simulation is expected to behave. The validity of those initial assumptions and conditions was also considered while comparing theoretical and empirical analyses.

Word count: 50

#Professionalism

We organized the report with relevant titles and subsections. We also included a table of contents. The report was run through Grammarly to optimize the sentence structure and fix any lingering typos. We also used the math conventions when writing formulas and included them as equations in Google Docs. We have provided a well-formatted bibliography and appendix.

Word count: 57

Appendix

Appendix A: Python Code

```
import heapq

class Event:
    '''
    Store the properties of one event in the Schedule class defined below.
Each
    event has a time at which it needs to run, a function to call when
running
    the event, along with the arguments and keyword arguments to pass to
that
    function.

    Attributes
    ----------
    timestamp: int
        The time in minutes that has passed since the opening of the
grocery store.
    function: func
    *args
        The first argument to any event function is always the schedule in
which
        events are being tracked.
    **kwargs
        Keyword arguments [what is this used for?]
    '''
    def __init__(self, timestamp, function, *args, **kwargs):
        self.timestamp = timestamp
        self.function = function
        self.args = args
        self.kwargs = kwargs

    def __lt__(self, other):
        '''
        This overloads the less-than operator in Python. We need it so the
```

```
        priority queue knows how to compare two events. We want events
with
        earlier (smaller) times to go first.

        Parameters
        ----------
        other: Event class instance
            An instance of the same class to compare with.

        Returns
        -------
        bool
            True if the first event's timestamp is smaller than the second
event's
            ('other'). False, otherwise.
        '''
        return self.timestamp < other.timestamp

    def run(self, schedule):
        '''
        Run an event by calling the function with its arguments and
keyword
        arguments. The first argument to any event function is always the
        schedule in which events are being tracked. The schedule object
can be
        used to add new events to the priority queue.

        Parameters
        ----------
        schedule
            An instance of the Schedule class.
        '''
        self.function(schedule, *self.args, **self.kwargs)

    def __str__(self):
        return f'Event happened at {self.timestamp} with function
{self.function.__name__}'


class Schedule:
```

```
    '''
    Implement an event schedule using a priority queue. You can add events
and
    run the next event.

    The `now` attribute contains the time at which the last event was run.

    Attributes
    ----------
    now: int
        The time (in minutes) that has passed since the opening of the
store
        (at 9 a.m.)
    priority_queue: list
        A list that is used as a priority queue of the events based on
their
        order in terms of time.
    '''

    def __init__(self):
        self.now = 0
        self.priority_queue = []

    def add_event_at(self, timestamp, function, *args, **kwargs):
        '''
        This method is used to add events (Even class instance) to the
priority
        queue.

        Parameters
        ----------
        timestamp: int
            The time at which the event occurs.
        function: function
            The function applied to the event (e.g. start serving
sustomer,
            finish serving customer).

        Returns
        -------
```

```
        Doesn't return any values. Pushes a new event to the priority
queue of
        the Schedule instance.
        '''
        heapq.heappush(
            self.priority_queue,
            Event(timestamp, function, *args, **kwargs))


    def add_event_after(self, interval, function, *args, **kwargs):
        '''
        This method is used to add events (Event class instance) to the
priority
        queue after a specified time interval from the current time, which
is
        expressed with the now attribute of the Schedule instance.

        Parameters
        ----------
        interval: int
            The time interval after which the event occurs.
        function: function
            The function applied to the event (e.g. start serving
sustomer,
            finish serving customer).

        Returns
        -------
        Doesn't return any values. Pushes a new event to the priority
queue of
        the Schedule instance.
        '''
        self.add_event_at(self.now + interval, function, *args, **kwargs)

    def next_event_time(self):
        '''
        This method returns the time of the next event in the priority
queue.

        Parameters
```

```
        ----------
        No parameters.

        Returns
        -------
        timestamp: int
            Returns the timestamp of the next event in the queue.
        '''
        return self.priority_queue[0].timestamp

    def run_next_event(self):
        '''
        This method runs the next event in the priority queue (the one at
the
        top of the queue with highest priority), updates the current time
(now
        attribute) and runs the event instance by invoking the Event.run()
method.

        Parameters
        ----------
        No parameters.

        Returns
        -------
        No returns.
        '''
        # Get next event in line.
        event = heapq.heappop(self.priority_queue)

        # Update the current time.
        self.now = event.timestamp

        # Run the event.
        event.run(self)

    def __repr__(self):
        '''
        This method changes the representation of an object when printed.
```

```
        Parameters
        ----------
        No parameters.

        Returns
        -------
        str
            The representation of the object including the timestamp and
number
            of events in the priority queue.
        '''
        return (
            f'Schedule() at time {self.now}min ' +
            f'with {len(self.priority_queue)} events in the queue')

    def print_events(self):
        '''
        Prints all the events in the priority queue.

        Parameters
        ----------
        No parameters.

        Returns
        -------
        str
            Each event, including its timestamp and the function name
associated
            with it.
        '''
        print(repr(self))
        for event in sorted(self.priority_queue):
            print(f'  ⏱ {event.timestamp}min: {event.function.__name__}')


    def __str__(self):
        '''
        This method changes the representation of an object when printed.

        Parameters
```

```
        ----------
        No parameters.

        Returns
        -------
        str
            The representation of the the events in the priority queue as
a
            string.
        '''
        for i in self.priority_queue:
            return str(i)
import scipy.stats as sts
class Queue_MG1:
    '''
    This class implements an MG1 queue. Each queue is linked with a single
server.
    All cashier queues have information about the manager queue (linked as
an
    instance of this class).

    Attributes
    ----------
    service_distribution: scipy.stats distribution
        The service distribution of the queue.
    number: int
        The number of the queue from 1 to 10 (since there is a maximum of
10
        cashiers in total).
    service_type: str
        The type of queue: 'CASHIER' is it is a normal cashier queue or
'MANAGER'
        if it is the queue of the (single) manager.
    manager_queue: instance of the Queue_MG1 class.
        Points toward the manager queue. If the instance is of sevice_type
        'MANAGER' this attribute is None.
    '''

    def __init__(self, service_distribution, number, service_type =
"CASHIER", manager_queue = None):
```

```
        ### CHANGED - each queue has a 'name' which is expressed as an
integer number
        # In grocery stores cashiers are often linked with numbers so we
felt
        # comforable adding this attribute so it is easier to follow the
behavior
        # of the system (i.e. when printing out the timeline).
        self.number = number

        # Store the deterministic service time for an M/D/1 queue
        self.service_distribution = service_distribution

        # We start with an empty queue and the server not busy
        self.people_in_queue = 0
        self.people_being_served = 0

        ### CHANGED - added type of queue 'MANAGER' or 'CASHIER'
        self.service_type = service_type

        ### CHANGED - each cashier queue points to the queue for the
manager
        self.manager_queue = manager_queue

        ### CHANGED - add a list to store the arrival, serving, and
departure
        # times for each customer (FIFO so they match)
        self.arrival_times = []
        self.serving_start_times = []
        self.departure_times = []

        ### CHANGED - keep track of queue length
        self.max_queue_length = 0

    def add_customer(self, schedule):
        '''
        This method adds a customer as an event to the priority queue. It
also
        checks if there is currently a customer being served at the
cashier. If
```

```
        not, the customer being added can be served immediately and an
event is
        added to the schedule to start serving this customer.

        Parameters
        ----------
        schedule: lst
            A priority queue with events.

        Returns
        -------
        str
            Prints an overview of the schedule at this timestamp including
the length
            of the queue, the type of queue (cashier or manager), the
queue ID,
            and the time.
        '''

        # Add the customer to the queue.
        self.people_in_queue += 1


        ### CHANGED - keep track of maximum queue length
        self.max_queue_length = max(self.max_queue_length,
self.people_in_queue)


        ### CHANGED - keep track of customer arrival time to the queue
        self.arrival_times.append(schedule.now)

        print(
            f'⏱{schedule.now:5.2f}min: Add customer to
{self.service_type.lower()} queue.     '
            f' 🧍🏾‍♀️People in the queue #{self.number}:
{self.people_in_queue}')

        if self.people_being_served < 1:
            # This customer can be served immediately.
            schedule.add_event_after(0, self.start_serving_customer)
```

```
    def start_serving_customer(self, schedule):
        '''
        This method starts the service of the customer at the front of the
queue.
        It also add the finish serving customer event to the queue by
generating
        a service time from the N(3, 1) distribution.

        Parameters
        ----------
        schedule: lst
            A priority queue with events.

        Returns
        -------
        str
            Prints an overview of the schedule at this timestamp including
the length
            of the queue, the type of queue (cashier or manager), the
queue ID,
            and the time.
        '''
        # Move the customer from the queue to a server.
        self.people_in_queue -= 1
        self.people_being_served += 1
        print(
            f'⏱{schedule.now:5.2f}min: Start serving customer at
{self.service_type.lower()}. '
            f' 🧍People in the queue #{self.number}:
{self.people_in_queue}')

        ### CHANGED - keep track of customer serving time to the queue
        self.serving_start_times.append(schedule.now)

        # Schedule when the server will be done with the customer
        schedule.add_event_after(
            self.service_distribution.rvs(),
            self.finish_serving_customer)
```

```
    def finish_serving_customer(self, schedule):
        '''
        This method ends the service of a customer. A random probability
is
        generated (if it is a cashier queue), if this is smaller than
0.05,
        the customer is added to the queue for the manager. It also checks
in there
        are any other customers in this specific queue for the cashier,
and begins
        the service of a new customer if so, adding this as an event to
the schedule.

        Parameters
        ----------
        schedule: lst
            A priority queue with events.

        Returns
        -------
        str
            Prints an overview of the schedule at this timestamp including
the length
            of the queue, the type of queue (cashier or manager), the
queue ID,
            and the time.
        '''
        # Remove the customer from the server
        self.people_being_served -= 1

        print(
            f'⏱{ schedule.now:5.2f}min: Stop serving customer at
{self.service_type.lower()}.  '
            f' 🧍🏾‍♀️People in the queue #{self.number}:
{self.people_in_queue}')

        # If at cashier queue, check if customer moves to manager or
leaves
        if self.service_type == 'CASHIER':
            if np.random.random() < 0.05:
```

```
                self.manager_queue.add_customer(schedule)

        ### CHANGED - keep track of customer departure time from the queue
        self.departure_times.append(schedule.now)

        if self.people_in_queue > 0:
            # There are more people in the queue so serve the next
customer
            schedule.add_event_after(0, self.start_serving_customer)


class GroceryStore_MG1:
    '''
    This class implements an MG1 queue. Each queue is linked with a single
server.
    All cashier queues have information about the manager queue (linked as
an
    instance of this class).

    Attributes
    ----------
    num_queues: int
        The number of queues in the store (1-10).
    arrival_distribution: scipy.stats distribution
        The arrival distribution of the customers to the queues.
    service_distribution: scipy.stats distribution
        The service distribution of the cashier queues.
    manager_distribution: scipy.stats distribution
        The service distribution of the manager queues.
    run_until: int
        The time in minutes of the time the store opens (9 a.m.) until the
closing
        time (8 p.m.).
    '''

    def __init__(self, num_queues, arrival_distribution,
service_distribution, manager_distribution, run_until):
        ### CHANGED
        # Add a manager (queue) to the grocery store. There is a single
manager.
```

```
        self.manager_queue = Queue_MG1(manager_distribution, 0,
service_type = "MANAGER")

        ### CHANGED
        # Initialize the queues as a list of Queue objects.
        self.queues = []
        for i in range(num_queues):
            self.queues.append(Queue_MG1(service_distribution =
service_distribution,
                                         number = i+1,
                                         manager_queue =
self.manager_queue))

        self.arrival_distribution = arrival_distribution
        self.run_until = run_until

        self.total_customers = 0

    def add_customer(self, schedule):
        '''
        Adds a customer to the shortest queue in self.queues (only if it
is before
        the stores closing time). The customer's arrival is added as an
event
        to the schedule by generating their arrival time from Exp(1).

        Parameters
        ----------
        schedule: lst
            A priority queue with events.

        Returns
        -------
        No returns.
        '''

        ### CHANGED
        if schedule.now <= self.run_until:
            # Add this customer to the shortest queue
            # Intialized current shortest queue.
```

```
            shortest_queue = self.queues[0]
            min_people_in_queue = self.queues[0].people_in_queue

            self.total_customers += 1

            # Traverse all queues and find the one with the min number of
people.
            for queue in self.queues[1:]:
                if queue.people_in_queue < min_people_in_queue:
                    min_people_in_queue = queue.people_in_queue
                    shortest_queue = queue

            shortest_queue.add_customer(schedule)
            # Schedule when to add another customer
            schedule.add_event_after(
                self.arrival_distribution.rvs(),
                self.add_customer)

    def run(self, schedule):
        '''
        Adds the arrival of the first customer to the schedule as an event
by
        generating an arrival time from Exp(1).

        Parameters
        ----------
        schedule: lst
            A priority queue with events.

        Returns
        -------
        No returns.
        '''
        # Schedule when the first customer arrives
        schedule.add_event_after(
            self.arrival_distribution.rvs(),
            self.add_customer)

### CHANGED
```

```
def run_simulation_MG1(num_queues, arrival_distribution,
service_distribution, run_until, manager_distribution):
    """
    The main function that is used to run the M/G/1 queue simulation with
the
    specified average arrival and cashier and manager service rates, as
well as the
    total time for which to run the simulation and the number of queues in
the store.

    Parameters
    ----------
    num_queues: int
        The number of queues in the store (1-10).
    arrival_distribution: scipy.stats distribution
        The arrival distribution of the customers to the queues.
    service_distribution: scipy.stats distribution
        The service distribution of the cashier queues.
    manager_distribution: scipy.stats distribution
        The service distribution of the manager queues.
    run_until: int
        The time in minutes of the time the store opens (9 a.m.) until the
closing
        time (8 p.m.).

    Returns
    -------
    grocery_store: obj
        Schedule for the day, including when customers arrive, are served,
and depart,
        as well as other attributes of the grocery_store class.
    """
    schedule = Schedule()
    grocery_store = GroceryStore_MG1(num_queues, arrival_distribution,
                                     service_distribution,
manager_distribution, run_until)
    grocery_store.run(schedule)

    # runs while there are events in the priority queue to serve all
customers
```

```
    while schedule.priority_queue:
        schedule.run_next_event()

    return grocery_store

### CHANGED
def run_simulation(num_queues, arrival_distribution, service_distribution,
run_until, manager_distribution):
    """
    * Function is specifically for the average queue length simulations.
    The main function that is used to run the M/G/1 queue simulation with
the
    specified average arrival and cashier and manager service rates, as
well as the
    total time for which to run the simulation and the number of queues in
the store.

    Parameters
    ----------
    num_queues: int
        The number of queues in the store (1-10).
    arrival_distribution: scipy.stats distribution
        The arrival distribution of the customers to the queues.
    service_distribution: scipy.stats distribution
        The service distribution of the cashier queues.
    manager_distribution: scipy.stats distribution
        The service distribution of the manager queues.
    run_until: int
        The time in minutes of the time the store opens (9 a.m.) until the
closing
        time (8 p.m.).

    Returns
    -------
    grocery_store: obj
        Schedule for the day, including when customers arrive, are served,
and depart,
        as well as other attributes of the grocery_store class.
    """
    schedule = Schedule()
```

```
    grocery_store = GroceryStore_MG1(num_queues, arrival_distribution,
                                     service_distribution,
manager_distribution, run_until)
    grocery_store.run(schedule)

    # stops at the end of the duration time (8p.m.)
    while schedule.next_event_time() < run_until:
        schedule.run_next_event()

    return grocery_store

import numpy as np
class EmpiricalAnalyst:
    """
    This class calculates the values of different metrics used to
determine
    how many servers should be in the grocery store.

    Attributes
    ----------
    No attributes.
    """

    def calculate_cashier_waiting_time(self, grocery_store):
        """
        This method calculates the average waiting time of customers in
one
        grocery store simulation, specifically for the cashier queues and
not the
        manager queue.

        Parameters
        ----------
        grocery_store: obj
            Schedule for the day, including when customers arrive, are
served, and depart,
            as well as other attributes of the grocery_store class.

        Returns
        -------
```

```
        queue_waiting_times: lst
            A list of waiting times for each customer in the simulation.
        """

        # Only waiting time for the cashier queues
        queue_waiting_times = []

        for queue in grocery_store.queues:
            waiting_time = [service_start_i - arrival_i for arrival_i,
service_start_i
                            in zip(queue.arrival_times,
queue.serving_start_times)]
            queue_waiting_times.append(waiting_time)

        return self.flatten_list(queue_waiting_times)

    def calculate_manager_waiting_time(self, grocery_store):
        """
        This method calculates the waiting times of customers in one
        grocery store simulation, specifically for the manager queue and
not the
        cashier queues.

        Parameters
        ----------
        grocery_store: obj
            Schedule for the day, including when customers arrive, are
served, and depart,
            as well as other attributes of the grocery_store class.

        Returns
        -------
        queue_waiting_times: lst
            A list of waiting times for each customer that goes to the
manager
            in the simulation.
        """

        # Only waiting time for the manager queue
```

```
        queue_waiting_times = [service_start_i - arrival_i for arrival_i,
service_start_i
                                in
zip(grocery_store.manager_queue.arrival_times,

grocery_store.manager_queue.serving_start_times)]

        return queue_waiting_times

    def calculate_cashier_response_time(self, grocery_store):
        """
        This method calculates the response times of customers in one
        grocery store simulation, specifically for the cashier queues and
not the
        manager queue. The calculation is simplified as:

        service time = departure - service start
        waiting time = service start - arrival time
        waiting time + service time = departure - arrival

        Parameters
        ----------
        grocery_store: obj
            Schedule for the day, including when customers arrive, are
served, and depart,
            as well as other attributes of the grocery_store class.

        Returns
        -------
        response_times: lst
            A list of response times for each customer in the simulation.
        """

        response_times = []

        # Response time of each cashier queue.
        for queue in grocery_store.queues:
            response_time = [departure_i - arrival_i for arrival_i,
departure_i in
```

```
                              zip(queue.arrival_times,
queue.departure_times)]

            response_times.append(response_time)

        return self.flatten_list(response_times)

    def calculate_manager_response_time(self, grocery_store):
        """
        This method calculates the response times of customers in one
        grocery store simulation, specifically for the manager queue and
not the
        cashier queues. The calculation is simplified as:

        service time = departure - service start
        waiting time = service start - arrival time
        waiting time + service time = departure - arrival

        Parameters
        ----------
        grocery_store: obj
            Schedule for the day, including when customers arrive, are
served, and depart,
            as well as other attributes of the grocery_store class.

        Returns
        -------
        response_times: lst
            A list of response times for each customer in the simulation
that goes
            to the manager.
        """

        # Response time of manager queue.
        response_times = [departure_i - arrival_i for arrival_i,
departure_i in

zip(grocery_store.manager_queue.arrival_times,

grocery_store.manager_queue.departure_times)]
```

```
        return response_times

    def calculate_max_queue(self, grocery_store):
        """
        This method calculates the maximum queue length for one simulation
        across all cashier queues.

        Parameters
        ----------
        grocery_store: obj
            Schedule for the day, including when customers arrive, are
served, and depart,
            as well as other attributes of the grocery_store class.

        Returns
        -------
        max_queue_length: int
            The maximum queue length for the simulation.
        """

        max_queue_length = 0

        for queue in grocery_store.queues:
            max_queue_length = max(max_queue_length,
queue.max_queue_length)

        return max_queue_length

    def calculate_cashier_utilization(self, grocery_store):
        """
        This method returns the average utilisation across all cashier
queues in
        a single simulation, where utilization = arrival rate * average
service time.

        Parameters
        ----------
        grocery_store: obj
```

```
            Schedule for the day, including when customers arrive, are
served, and depart,
            as well as other attributes of the grocery_store class.

        Returns
        -------
        utilization: float
            The average utilization of the cashier queues in the
simulation.
        """
        # Arrival rate
        arrival_rate = grocery_store.total_customers /
grocery_store.run_until

        # Average service time
        service_times = []

        for queue in grocery_store.queues:
            service_time = [departure_i - service_start_i for departure_i,
service_start_i in
                            zip(queue.departure_times,
queue.serving_start_times)]
            service_times.append(service_time)

        service_times = self.flatten_list(service_times)
        average_service_time = np.mean(service_times)
        utilization = arrival_rate*average_service_time

        return utilization

    def calculate_average_queue_length(self, grocery_store):
        """
        This method calculates the average queue length across all n
cashier
        queues in the store (per queue).

        Parameters
        ----------
        grocery_store: obj
```

```
            Schedule for the day, including when customers arrive, are
served, and depart,
            as well as other attributes of the grocery_store class.

        Returns
        -------
        queue_length: float
            The number of average number of customers across all n queues
for each
            queue.
        """
        customers = 0

        # Sum customers left in cashier queues
        for queue in grocery_store.queues:
            customers += queue.people_in_queue

        queue_length = customers/len(grocery_store.queues)

        return queue_length

    def calculate_average_system_cashier_customers(self, grocery_store):
        """
        This method calculates the average number of customers in the
system
        (those waiting and being served) across all n cashier queues in
the
        store (per queue).

        Parameters
        ----------
        grocery_store: obj
            Schedule for the day, including when customers arrive, are
served, and depart,
            as well as other attributes of the grocery_store class.

        Returns
        -------
        system_size: float
```

```
            The number of average number of customers across all n queues
(per
            queue).
        """
        customers = 0

        # Sum customers left in cashier queues and being served
        for queue in grocery_store.queues:
            customers += queue.people_in_queue
            customers += queue.people_being_served

        system_size = customers/len(grocery_store.queues)

        return system_size

    def calculate_average_customer_waiting_time(self, grocery_store):

        """
        This method calculates the total waiting time for customer in the
system
        (for both cashiers and the manager).

        Parameters
        ----------
        grocery_store: obj
            Schedule for the day, including when customers arrive, are
served, and depart,
            as well as other attributes of the grocery_store class.

        Returns
        -------
        total_waiting_time: float
            The mean of all waiting times of customers who went to
cashiers and the manager.
        """

        average_cashier_waiting_time =
np.mean(self.calculate_cashier_waiting_time(grocery_store))

        if len(self.calculate_manager_waiting_time(grocery_store)) != 0:
```

```
          average_manager_waiting_time =
np.mean(self.calculate_manager_waiting_time(grocery_store))
        else:
          average_manager_waiting_time = 0

        total_waiting_time = average_cashier_waiting_time +
average_manager_waiting_time

        return total_waiting_time

    def calculate_average_customer_response_time(self, grocery_store):
        """
        This method calculates the total service time for customer in the
system
        (for both cashiers and the manager).

        Parameters
        ----------
        grocery_store: obj
            Schedule for the day, including when customers arrive, are
served, and depart,
            as well as other attributes of the grocery_store class.

        Returns
        -------
        total_service_time: float
            The mean of all service times of customers who were served by
the cashiers and the manager.
        """
        average_cashier_service_time =
np.mean(self.calculate_cashier_response_time(grocery_store))

        if len(self.calculate_manager_waiting_time(grocery_store)) != 0:
          average_manager_service_time =
np.mean(self.calculate_manager_response_time(grocery_store))
        else:
          average_manager_service_time = 0

        total_serving_time = average_cashier_service_time +
average_manager_service_time
```

```
        return total_serving_time

    def flatten_list(self, lst):
        """
        This method flattens a nested list.

        Parameters
        ----------
        lst: lst
            A nested list (eg. [[], []]).

        Returns
        -------
        flat_list: lst
            A non-nested list (eg. []).
        """
        # Taken from
https://stackoverflow.com/questions/952914/how-do-i-make-a-flat-list-out-o
f-a-list-of-lists
        flat_list = [item for sublist in lst for item in sublist]
        return flat_list

# Run the simulation to check it works as expected for a shorter duration
and one queue.
import numpy as np

# Exponential with lambda = 1
arrival_distribution = sts.expon(scale = 1)

# Normal Distribution with mu = 3 and sigma = 1 (standard deviation)
service_time_mean = 3
service_time_variance  = 1**2
service_distribution = sts.truncnorm(loc = service_time_mean,
                               scale = np.sqrt(service_time_variance),
                                a = -3, b = float('inf'))

### CHANGED - Add the parameters for the distribution for the manager
# Normal Distribution with mu = 5 and sigma = 2 (standard deviation)
manager_time_mean = 5
```

```
manager_time_variance  = 2**2
manager_distribution = sts.truncnorm(loc = manager_time_mean,
                               scale = np.sqrt(manager_time_variance),
                               a = -2.5, b = float('inf'))

duration = 1*5 # in minutes
num_queues = 1
## run the actual simulation
grocery_store = run_simulation_MG1(num_queues, arrival_distribution,
service_distribution,
                                   duration, manager_distribution)

# Visualize (by printing) number of individuals in each queue
# To check that all customers are served
for queue in grocery_store.queues:
    print(f'\n🔚 At closing time, there are {queue.people_in_queue} people
in queue #{queue.number}.')

    # check that all customers are served, even after the store closes
    assert queue.people_in_queue == 0

# check that there are the right number of queues implemented
assert len(grocery_store.queues) == num_queues

# Run the simulation to check it works as expected for a shorter duration
and one queue.
import numpy as np

# Exponential with lambda = 1
arrival_distribution = sts.expon(scale = 1)

# Normal Distribution with mu = 3 and sigma = 1 (standard deviation)
service_time_mean = 3
service_time_variance  = 1**2
service_distribution = sts.truncnorm(loc = service_time_mean,
                               scale = np.sqrt(service_time_variance),
                                a = -3, b = float('inf'))

### CHANGED - Add the parameters for the distribution for the manager
# Normal Distribution with mu = 5 and sigma = 2 (standard deviation)
```

```
manager_time_mean = 5
manager_time_variance  = 2**2
manager_distribution = sts.truncnorm(loc = manager_time_mean,
                                scale = np.sqrt(manager_time_variance),
                                a = -2.5, b = float('inf'))

duration = 1*5 # in minutes
num_queues = 1
## run the actual simulation
grocery_store = run_simulation_MG1(num_queues, arrival_distribution,
service_distribution,
                                    duration, manager_distribution)

# Visualize (by printing) number of individuals in each queue
# To check that all customers are served
for queue in grocery_store.queues:
    print(f'\n🔚 At closing time, there are {queue.people_in_queue} people
in queue #{queue.number}.')

    # check that all customers are served, even after the store closes
    assert queue.people_in_queue == 0

# check that there are the right number of queues implemented
assert len(grocery_store.queues) == num_queues

# Run the simulation to check it works as expected for a shorter duration
and five queues.

# Exponential with lambda = 1
arrival_distribution = sts.expon(scale = 1)

# Normal Distribution with mu = 3 and sigma = 1 (standard deviation)
service_time_mean = 3
service_time_variance  = 1**2
service_distribution = sts.truncnorm(loc = service_time_mean,
                                scale = np.sqrt(service_time_variance),
                                 a = -3, b = float('inf'))

### CHANGED - Add the parameters for the distribution for the manager
# Normal Distribution with mu = 5 and sigma = 2 (standard deviation)
```

```
manager_time_mean = 5
manager_time_variance  = 2**2
manager_distribution = sts.truncnorm(loc = manager_time_mean,
                               scale = np.sqrt(manager_time_variance),
                               a = -2.5, b = float('inf'))

duration = 5 # in minutes
num_queues = 5
## run the actual simulation
grocery_store = run_simulation_MG1(num_queues, arrival_distribution,
service_distribution,
                                   duration, manager_distribution)

# Visualize (by printing) number of individuals in each queue
# To check that all customers are served
for queue in grocery_store.queues:
    print(f'\n🔚 At closing time, there are {queue.people_in_queue} people
in queue #{queue.number}.')

    # check that all customers are served, even after the store closes
    assert queue.people_in_queue == 0

# check that there are the right number of queues implemented
assert len(grocery_store.queues) == num_queues

# Run the simulation to check it works as expected for the store opening
times
# and five queues.

# Exponential with lambda = 1
arrival_distribution = sts.expon(scale = 1)

# Normal Distribution with mu = 3 and sigma = 1 (standard deviation)
service_time_mean = 3
service_time_variance  = 1**2
service_distribution = sts.truncnorm(loc = service_time_mean,
                               scale = np.sqrt(service_time_variance),
                                a = -3, b = float('inf'))

### CHANGED - Add the parameters for the distribution for the manager
```

```
# Normal Distribution with mu = 5 and sigma = 2 (standard deviation)
manager_time_mean = 5
manager_time_variance  = 2**2
manager_distribution = sts.truncnorm(loc = manager_time_mean,
                               scale = np.sqrt(manager_time_variance),
                               a = -2.5, b = float('inf'))

duration = 60*11 # in minutes
num_queues = 5
## run the actual simulation
grocery_store = run_simulation_MG1(num_queues, arrival_distribution,
service_distribution,
                                   duration, manager_distribution)

# Visualize (by printing) number of individuals in each queue
# To check that all customers are served
for queue in grocery_store.queues:
    print(f'\n🔚 At closing time, there are {queue.people_in_queue} people
in queue #{queue.number}.')

    # check that all customers are served, even after the store closes
    assert queue.people_in_queue == 0

# check that there are the right number of queues implemented
assert len(grocery_store.queues) == num_queues

# Check that the number of people that are served in the grocery store in
total
# increases as we increase the arrival rate (linearly).
import matplotlib.pyplot as plt

duration = 5 # in minutes
num_queues = 1
arrival_rates = [0.5, 1, 1.5, 2, 2.5, 3]
trials = 50
avg_total_customers = []

for rate in arrival_rates:
    arrival_distribution = sts.expon(scale = 1/rate)
    total_customers = []
```

```
    for i in range(trials):
        grocery_store = run_simulation_MG1(num_queues,
arrival_distribution, service_distribution,
                                   duration, manager_distribution)
        total_customers.append(grocery_store.total_customers)

    avg_total_customers.append(np.mean(total_customers))

plt.plot(arrival_rates, avg_total_customers)
plt.xlabel('Arrival rate')
plt.ylabel('Average total customers')
plt.title('Average Total Customers in the System with Different Arrival
Rates')
plt.show()

# Check empirical analysis class and methods
analyst = EmpiricalAnalyst()

# Exponential with lambda = 1
arrival_distribution = sts.expon(scale = 1)

# Normal Distribution with mu = 3 and sigma = 1 (standard deviation)
service_time_mean = 3
service_time_variance  = 1**2
service_distribution = sts.truncnorm(loc = service_time_mean,
                               scale = np.sqrt(service_time_variance),
                               a = -3, b = float('inf'))

### CHANGED - Add the parameters for the distribution for the manager
# Normal Distribution with mu = 5 and sigma = 2 (standard deviation)
manager_time_mean = 5
manager_time_variance  = 2**2
manager_distribution = sts.truncnorm(loc = manager_time_mean,
                               scale = np.sqrt(manager_time_variance),
                                a = -2.5, b = float('inf'))

duration = 5 # in minutes
num_queues = 7
```

```
grocery_store = run_simulation_MG1(num_queues, arrival_distribution,
service_distribution,
                                   duration, manager_distribution)
print(analyst.calculate_cashier_waiting_time(grocery_store))
print(analyst.calculate_manager_waiting_time(grocery_store))
print(analyst.calculate_cashier_response_time(grocery_store))
print(analyst.calculate_manager_response_time(grocery_store))
print(analyst.calculate_max_queue(grocery_store))
print(analyst.calculate_cashier_utilization(grocery_store))

grocery_store = run_simulation(num_queues, arrival_distribution,
service_distribution,
                                   duration, manager_distribution)
print(analyst.calculate_average_queue_length(grocery_store))
print(analyst.calculate_average_system_cashier_customers(grocery_store))
print(analyst.calculate_average_customer_waiting_time(grocery_store))
print(analyst.calculate_average_customer_response_time(grocery_store))
```

```
import random

# adding a seed to get similar results
random.seed(32)

# a list storing all possible number of cashiers
num_cashiers = [1,2,3,4,5,6,7,8,9,10]

trials = 100
duration = 500 # 500/60 is over 8 hours

# a list storing average waiting time for different number of cashiers
average_waiting_time_per_cashier = []
# a list storing errors bars for each of the averages
errors = []

# run the simulation for each number of cashiers
for cashier in num_cashiers:
  average_waiting_times = []
```

```
  for i in range(trials): # run the simulation 100 times and estimate the
average waiting time
    grocery_store = run_simulation(cashier, arrival_distribution,
service_distribution, duration, manager_distribution)
    all_waiting_times =
analyst.calculate_average_customer_waiting_time(grocery_store) # waiting
times of all customers in that simulation
    average_waiting_times.append(np.mean(all_waiting_times)) # stroring
100 average waiting times

  # conducting a t-test
  t = sts.sem(average_waiting_times)
  errors.append(2 * 1.96 * t) # distance between lower and higher bound of
intervals gives the error value

  average_waiting_time_per_cashier.append(np.mean(average_waiting_times))

print(average_waiting_time_per_cashier)
print(errors)


# import necessary libraries
import matplotlib.pyplot as plt
import scipy.stats as sts
import numpy as np

# calculating theoretical values for each server amount
def theoretical_average_waiting_time(num_cashiers):
  values = []
  for cashier in num_cashiers:
    values.append(5/(cashier - 3) + 0.97)
  return values

# plotting two results and error bars
def draw_plot(num_cashiers, average_values, errors, label):
    plt.errorbar(num_cashiers, average_values, errors,
                 color='black', marker='o', capsize=5, linestyle='--',
                 linewidth=1, label='empirical')
    # no theoretical values exist when cashier number is either 1 or 2 or
3
```

```
    theoretical_average_values =
theoretical_average_waiting_time(num_cashiers[3:])
    plt.plot(num_cashiers[3:], theoretical_average_values,
             color='red', marker='o', linestyle='--', linewidth=1,
label='theoretical')
    plt.xlabel('Number of Servers')
    plt.ylabel(label)
    plt.legend()
    plt.show()

print(average_waiting_time_per_cashier)


draw_plot(num_cashiers, average_waiting_time_per_cashier, errors, label =
"Average Customer Waiting Time, min")


random.seed(88)

num_cashiers = [1,2,3,4,5,6,7,8,9,10]
trials = 100
duration = 500

average_response_time_per_cashier = []
errors = []

for cashier in num_cashiers:
  average_response_times = []

  for i in range(trials):
    grocery_store = run_simulation(cashier, arrival_distribution,
service_distribution, duration, manager_distribution)
    all_response_times =
analyst.calculate_average_customer_response_time(grocery_store) # all
serving times for all customers
    average_response_times.append(np.mean(all_response_times))

  # error bars based on the mean distribution between trials
  t = sts.sem(average_response_times)
  errors.append(2 * 1.96 * t)
```

```
average_response_time_per_cashier.append(np.mean(average_response_times))

print(average_response_time_per_cashier)
print(errors)


# the theoretical formula is derived from the report
def theoretical_average_response_time(num_cashiers):
  values = []
  for cashier in num_cashiers:
    values.append(3 + 5/(cashier - 3) + 5.97)
  return values

def draw_plot(num_cashiers, average_values, errors, label):
    plt.errorbar(num_cashiers, average_values, errors,
                 color='black', marker='o', capsize=5, linestyle='--',
                 linewidth=1, label='empirical')
    theoretical_average_values =
theoretical_average_response_time(num_cashiers[3:])
    plt.plot(num_cashiers[3:], theoretical_average_values,
             color='red', marker='o', linestyle='--', linewidth=1,
label='theoretical')
    plt.xlabel('Number of Servers')
    plt.ylabel(label)
    plt.legend()
    plt.show()
draw_plot(num_cashiers, average_response_time_per_cashier, errors, label =
"Average Customer Service Time, min")


random.seed(78)

num_cashiers = [1,2,3,4,5,6,7,8,9,10]
trials = 100
duration = 500

average_maximum_queue_length = []
errors = []

for cashier in num_cashiers:
```

```
  maximum_queue_lengths = []

  for i in range(trials):
    grocery_store = run_simulation(cashier, arrival_distribution,
service_distribution, duration, manager_distribution)
    one_max_length = analyst.calculate_max_queue(grocery_store) # maximum
queue length for each of the runs
    maximum_queue_lengths.append(one_max_length)

  t = sts.sem(maximum_queue_lengths)
  errors.append(2 * 1.96 * t)

  average_maximum_queue_length.append(np.mean(maximum_queue_lengths))

print(average_maximum_queue_length)
print(errors)


# no theoretical formulas exist for maximum queue length
def draw_plot(num_cashiers, average_values, errors, label):
    plt.errorbar(num_cashiers, average_values, errors,
                 color='black', marker='o', capsize=5, linestyle='--',
                 linewidth=1, label='empirical')
    # plt.axhline(1, color='red', linestyle='--')
    plt.xlabel('Number of Servers')
    plt.ylabel(label)
    plt.legend()
    plt.show()


draw_plot(num_cashiers, average_maximum_queue_length, errors, label =
"Average Maximum Queue Length")


random.seed(711)

num_cashiers = [1,2,3,4,5,6,7,8,9,10]
trials = 100
duration = 500

average_number_people_insystem = []
```

```
errors = []

for cashier in num_cashiers:
  number_people_insystem = []

  for i in range(trials):
    grocery_store = run_simulation(cashier, arrival_distribution,
service_distribution, duration, manager_distribution)
    one_system =
analyst.calculate_average_system_cashier_customers(grocery_store)
    number_people_insystem.append(one_system)

  t = sts.sem(number_people_insystem)
  errors.append(2 * 1.96 * t)

  average_number_people_insystem.append(np.mean(number_people_insystem))

print(average_number_people_insystem)
print(errors)

def theoretical_average_people_insystem(num_cashiers):
  values = []
  for cashier in num_cashiers:
    values.append(3 + 5/(cashier - 3))
  return values

def draw_plot(num_cashiers, average_values, errors, label):
    plt.errorbar(num_cashiers, average_values, errors,
                 color='black', marker='o', capsize=5, linestyle='--',
                 linewidth=1, label='empirical')
    theoretical_average_values =
theoretical_average_people_insystem(num_cashiers[3:])
    plt.plot(num_cashiers[3:], theoretical_average_values,
             color='red', marker='o', linestyle='--', linewidth=1,
label='theoretical')
    plt.xlabel('Number of Servers')
    plt.ylabel(label)
    plt.legend()
    plt.show()
```

```
draw_plot(num_cashiers, average_number_people_insystem, errors, label =
"Average Number of People in the System")


random.seed(666)
num_cashiers = [1,2,3,4,5,6,7,8,9,10]
trials = 100
duration = 100

average_queue_length_per_cashier_size = []
errors = []

for cashier in num_cashiers:
  average_queue_length = []

  for i in range(trials):
    grocery_store = run_simulation(cashier, arrival_distribution,
service_distribution, duration, manager_distribution)
    one_average= analyst.calculate_average_queue_length(grocery_store)
    average_queue_length.append(one_average)

  t = sts.sem(average_queue_length)
  errors.append(2 * 1.96 * t)

  # append the average value for the specific number of cashiers in the
system

average_queue_length_per_cashier_size.append(np.mean(average_queue_length)
)

print(average_queue_length_per_cashier_size)
print(errors)


def theoretical_average_queue_length(num_cashiers):
  values = []
  for cashier in num_cashiers:
    values.append(5/(cashier*(cashier - 3)))
  return values

def draw_plot(num_cashiers, average_values, errors, label):
```

```
    plt.errorbar(num_cashiers, average_values, errors,
                 color='black', marker='o', capsize=5, linestyle='--',
                 linewidth=1, label='empirical')
    theoretical_average_values =
theoretical_average_queue_length(num_cashiers[3:])
    plt.plot(num_cashiers[3:], theoretical_average_values,
             color='red', marker='o', linestyle='--', linewidth=1,
label='theoretical')
    plt.xlabel('Number of Servers')
    plt.ylabel(label)
    plt.legend()
    plt.show()


draw_plot(num_cashiers, average_queue_length_per_cashier_size, errors,
label = "Average Queue Length")
```