Assignment 1 Project Report

Introduction

In this project, we apply the modeling, simulation, and analysis process to implement a grocery queue simulation. We implement the modeling and simulation process using object-oriented programming in Python with the support of NumPy and Matplotlib functionalities. The goal of this project is to explore the staffing problem of ***how many cashiers should the grocery store employ.*** We will do so by modeling the system, implementing the simulation, and analyzing the results both empirically and theoretically with relevant metrics in mind to reach a plausible suggestion.

System Description

The system can be described as below:

- A grocery store opens at 9 am and closes at 8 pm with several checkout queues. The number of queues cannot exceed 10.
  - After closing time, customers can no longer join the queues, and customers in the queues will be served until the queues are empty.
- Customers arrive at the store, join the queue with the shortest line, and get served by the cashier. 5% of customers need the manager for extended help, others leave the store.
  - The time interval between customers arriving at the store follows an exponential distribution with $\lambda = 1$ customer per minute.
  - The time it takes for a cashier to serve a customer is normally distributed with mean $\mu_1 = 3$ minutes and standard deviation of $\sigma_1 = 1$ minute.

- The time it takes for the manager to handle a request is normally distributed with mean $\mu_2 = 5$ minutes and standard deviation of $\sigma_2 = 2$ minutes. There is only one manager.

Desired Feedback

We would like feedback primarily on the elegance of our code (part of it felt repetitive) and our discussion of empirical vs theoretical results. Specifically, we would like to know if our discussion felt sufficiently detailed and aware of assumptions in the model and approaches. We also noted a difference in the graph for average queue length (while the lines for average wait time and response time matched perfectly) in Figure 2 and provided a theory as to why that is. It would be great to get feedback on whether our theory makes sense or if the difference exists due to bugs in our modeling or analysis.

Model Implementation

In order to implement the model, we use the Event and Schedule classes from the pre-class workbook for a simple M/D/1 model. These two classes implement a priority queue for managing and running events (such as customers joining a queue, being served, and leaving) based on their scheduled time.

The model requires a few other classes: GroceryStore, Queue, ManagerQueue, and Customer. We can see that through a single iteration of the simulation, all the classes are used together:

1. Initiate the grocery store in the GroceryStore class. We use the get_customer_till_closing method to generate all the customers arriving until the store's closing time based on the time intervals sampled from the arrival time distribution. We add these arrivals as events to the schedule.
2. The arrival of a customer. The customer arrives following a specified arrival distribution, which is done using the arrival method inside the GroceryStore class. Now the customer needs to decide to join the shortest queue. This particular action is handled again within the GroceryStore class using the find_shortest_line method.
3. The cashier serves the customer. The Queue class handles the cashier's interactions with the customer. In the class, we can specifically look at the serve_next method which checks if the cashier for that particular queue is available for serving, if so then the customer in front of the queue is popped out of the queue and assigned to the cashier. The time for the service is dependent on the service distribution which we specify at the start of the simulation, so we schedule an event to take place after that amount of time elapses.
4. The cashier finishes serving the customer. Once the service has been completed we can move on to the next customer and the previous customer can leave the system, leaving the cashier available (done in the done_serving method of the same class). If there are more customers in the line, the cashier immediately starts serving the next person.
5. The customer who was just served might need the help of the manager. In 5% of all cases, we add the customer to a manager queue implemented using the ManagerQueue class. The manager queue and handling system can be thought of as a simple M/G/1 model as the arrival of the customers to the original system are still exponential and it carries forward, the service distribution for the manager is different from the cashiers but it is still a normal distribution so it is a general distribution, and there is only 1 queue for the manager.

6. The manager helps the customer. This uses the done_serving method in the ManagerQueue class. Overall, the structure of the manager line is similar to that of the cashier line.

Another important class is the Customer class. This class handles the information for the customer like their ID and arrival time, while also handling information like the starting and ending of service times for both the cashier and the manager. After the simulation is complete, we can use these timestamps stored in all Customer instances that have visited the store that day to find the metrics we are measuring, as discussed in the following section.

Test Cases

#1: Check whether the model is able to take into account that no more customers will be added to the queue after the grocery store has been closed, and all existing customers get served till completion. This is reflected by the queue lengths after the shop has been closed. We see that as long as run_until is equal to or bigger than closing_time (implying that we are running the simulation till completion), we can see that the customers in the queue(s) become 0 for all the different numbers of queues.

#2: Checks whether the code implements the arrival times of the customer. We increased the arrival rate to see how the queue lengths are affected. We confirmed that this parameter was working well since when the arrival rate grows significantly, the grocery store ends up with extremely long lines that built up over the day.

#3: Checks whether basic functionalities of the model (such as adding customers to queues, serving customers, etc) perform as expected. We do this by setting the number of queues to 1 so customers needing to join the shortest queue no longer change our theoretical analysis. We ran 100 trials and confirmed that the results aligned with theoretical analysis.

Simulation Approaches

We chose to explore cases where the number of queues ranges from 4 to 9. We are not dealing with queue length = 3 because it cannot be verified using theoretical analysis (see the section below). For each of the different queues, we run the simulation till completion for 100 trials in order to have a significantly large sample. From these samples, we can find relevant metrics to evaluate the outcomes.

Results

To explore the question of *how many cashiers should the grocery store employ,* we chose four metrics to explore given the different numbers of cashiers employed:

1. **Average customer waiting time:** the time interval between a customer joining a queue and when the customer starts being served; if the customer sees the manager, this is the sum of their waiting time in the cashier queue and the manager queue.
2. **Average customer response time:** time interval between a customer joining a queue and when the customer finishes being served; if the customer sees the manager, this is between the customer joining the cashier queue and the customer finishing being helped by the manager.
3. **Max queue length:** maximum number of people in a queue in the grocery store at any point in time.
4. **Average queue length:** average number of people in each queue during the day.

We chose *average queue length* as the fourth metric because of two reasons. Firstly, it is a metric that we can use theoretical analysis to compute (as shown below), whereas we cannot do so for maximum queue length since it's largely random. Secondly, we believe it's a metric that is significant for the customers and the store. The average queue length describes what the customer typically sees when they join a queue and long queues are not ideal. While only a few unlucky customers might encounter the maximum queue length, many will encounter queue lengths around the average queue length. Thus it's a metric that we want to keep in mind when deciding how many cashiers to employ (more cashiers will lead to shorter average queue length).

Theoretical Calculations

Our scenario can be roughly modeled using an M/G/1 model.

M: the arrival rate of customers follows an exponential distribution. This assumes that events are entirely independent and that states in the system can be modeled by updating the previous state only.

G: the service rate follows a normal distribution, so we are using a general distribution model.

1: although we have multiple queues in this model, we can see them as separate systems. **This is a key assumption that we are making for the theoretical analysis that does not reflect the empirical model.**

In order to use an M/G/1 model, we need to see each queue independently. This is equivalent to customers randomly choosing a queue to join — in which case, the arrival rate to each queue is simply $\frac{1}{n}$ of the arrival rate to the system, with n being the number of queues. Other parameters remain the same. However, this is not the same as the customer always choosing the shortest queue, since in that case each of the queues are dependent on each other. Additionally, we repeat the calculations for the manager queue (which is an M/G/1 queue) and add the results while accounting for the 5% probability of a customer joining the manager queue.

As we are limited in our capacities to reflect that detail, we will use n M/G/1 models to approximate the cashier queues:

Variables

- Number of queues n
- Average system arrival rate $\lambda_{sys} = 1$ customer per minute
- Average queue arrival rate $\lambda = \frac{1}{n}$ customer per minute
- Average service time $\tau = 3$ minutes
- Variance of service time $\sigma = 1$ minute
- Utilization $\rho = \lambda \times \tau = \frac{3}{n}$

Formulas

- Average waiting time at equilibrium:

$$\frac{\rho\tau}{2(1-\rho)}(1+\frac{\sigma^2}{\tau^2})$$

- Average response time at equilibrium:

$$\text{average service time } + \text{average waiting time}$$
$$= \tau + \frac{\rho\tau}{2(1-\rho)}(1 + \frac{\sigma^2}{\tau^2})$$

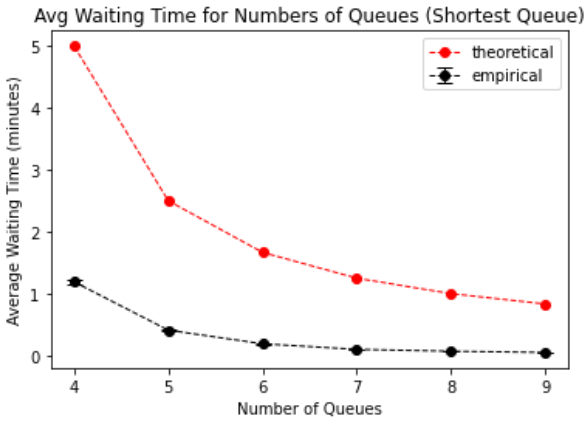
- Average queue length at equilibrium (Little's Law):

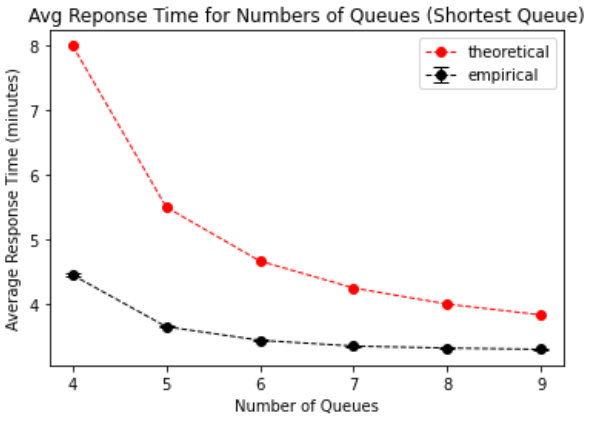
$$\text{average queue length} = \lambda \times \text{average waiting time}$$

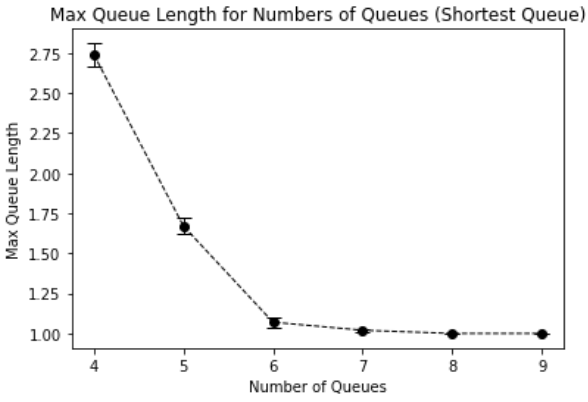
Note that when $n = 3$, $1 - \rho = 0$, making the denominator 0. Therefore, the theoretical analysis does not apply when the queue length is 3, and we start our analysis when there are 4 cashiers.

Empirical & Theoretical Analysis

The graphs in Figure 1 show the four metrics in our empirical simulation compared to the theoretical analysis. Clearly, the empirical results have far better performance — customers spend significantly less time waiting in line (for $n = 4$, the time decreases from 5 min to slightly over 1 min) and the average queue length roughly halves for $n = 4$. The reason our empirical analysis and theoretical analysis do not align is due to the key assumption in the theoretical analysis that *customers join queues randomly.* This assumption is unrealistic and sub-optimal since queues will grow unnecessarily longer and thus lead to longer waits.







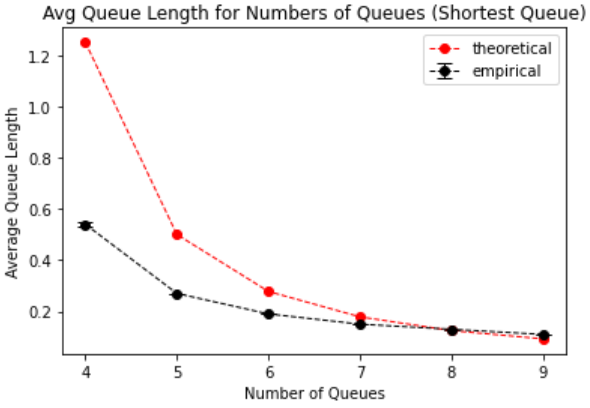


Figure 1. Results from modeling of customers joining the shortest queue (trials = 100, confidence interval of 95%).

To confirm our hypothesis that this assumption is the key driver behind the mismatch, we tweaked the model to produce results with customers choosing to join lines randomly instead of finding the shortest queue. The number of trials remains at 100. In the graphs in Figure 2, the empirical and theoretical results for average waiting time and average response time almost perfectly line up, which lends support to our theoretical analysis and modeling implementation being correct. Empirically, the average queue lengths seem to be slightly longer than the theoretical values. This might be attributable to when the queue length is measured — in our implementation, the queue length is measured at almost all timestamps when an action is performed, whereas, for the theoretical analysis, it is taken at some stable timestamp. This could be an explanation for the consistent difference in queue length across all numbers of queues.

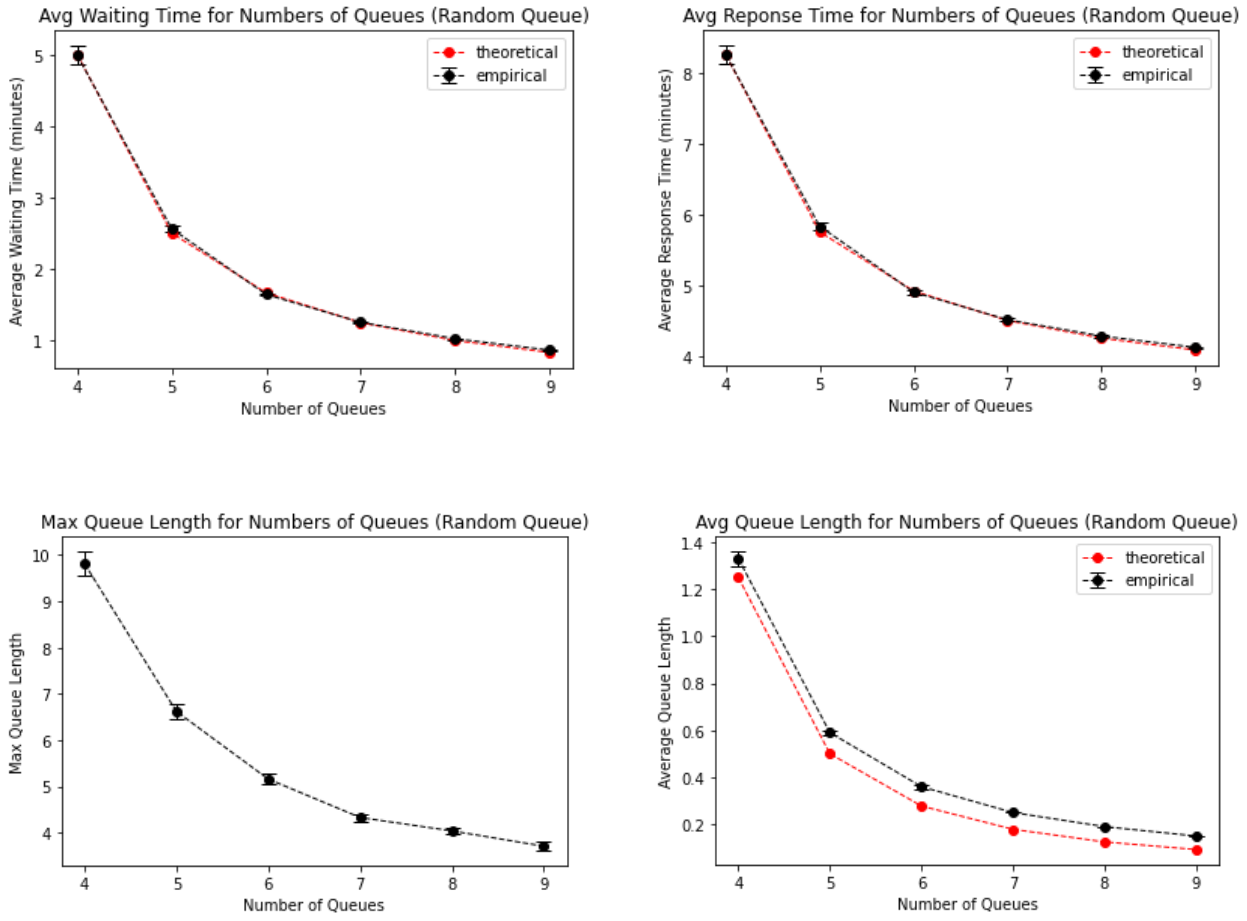


Figure 2. Results from modeling of customers joining random queue (trials = 100, confidence interval of 95%).

Despite the limitations of our theoretical analysis in reflecting the scenario at hand, we were able to use it to confirm that our implementation is indeed correct by changing the model to conform with the assumption that customers join queues randomly. It's intuitive that when customers join the shortest queue, it would significantly increase the efficiency of the system, thus decreasing metrics including average wait time, average response time, and average queue length. Given this improvement, we found that having 4 cashiers

is more than sufficient to provide customers with a satisfactory experience. At $n = 4$, we can see that the average wait time is 1.19 minutes, which is only 24% of that of the random queue (Figure 3). The average queue length is 0.54, meaning that almost half of the customers can immediately be served when they arrive, which is a significant improvement from the average queue length of 1.48 in the random queue model. Although even more cashiers can decrease these metrics even further, it would incur additional costs for the grocery store that seems unnecessary given the efficiency of the model using 4 cashiers.

	Wait Time (minutes)				Average Queue Length			
	Shortest Queue	Random Queue	Theoretical	Shortest vs Random	Shortest Queue	Random Queue	Theoretical	Shortest vs Random
4	1.19	5	5.05	23.80%	0.54	1.48	1.25	36.49%
5	0.41	2.57	2.55	15.95%	0.27	0.7	0.5	38.57%
6	0.19	1.65	1.72	11.52%	0.19	0.43	0.28	44.19%
7	0.1	1.26	1.3	7.94%	0.15	0.32	0.18	46.88%
8	0.07	1.03	1.05	6.80%	0.13	0.25	0.13	52.00%
9	0.05	0.87	0.88	5.75%	0.11	0.2	0.09	55.00%

Figure 3. Summary of results from empirical analysis (shortest queue approach and random queue approach) and theoretical analysis.

Word Count: 2,223

Appendices

A. LO & HC Applications

LOs

#EmpiricalAnalysis:

We ran the simulation multiple time and visualized the results of the simulation to see how it changes with different numbers of queues. We also calculated the confidence interval to check how confident we were in our empirical results and how they could carry. The multiple runs of the simulations were compared among the different queue lengths and the results were interpreted and also compared with the theoretical results.

Word Count: 66 words

#TheoreticalAnalysis:

We used proper mathematical formulas in order to make predictions of how the queue lengths will be affected for different numbers of queues. We also highlighted the key assumption that we had to make when doing theoretical analysis (queues joined randomly) and later compared the results of theoretical results with the empirical results to show that it was true.

Word Count: 57 words

#Professionalism:

We followed the instructions of the project and focused on the results of our report so that it only included parts that provided the most important and relevant information regarding the problem at hand. The visualizations included in the report are succinct and express the results we want clearly to the audience.

Word Count: 51 words.

#CodeReadability:

The code has been well documented and has a docstring for all the classes, methods, and functions, thus making it easy for the reader to understand what different parts of the code are doing and specifying why these different things are used there. We also made sure the choice of variable made sense and the structure for the code stayed consistent.

Word Count: 57 words

#Modeling:

We implemented all the required conditions to our model that were described in the project instructions, they can be seen in the code. The model was made keeping the queueing theory in mind while also taking into consideration the additional assumptions that came with the project instructions. The explanation for the model is also done in a manner that is easily understandable by our audience.

Word Count: 63 words

#PythonImplementation:

We implemented the simulation as it was specified to us in the project instructions. This involved using data structures like priority queue, which helped in keeping the time steps in order and also overall with the simulation. We also ran some test cases like checking that the customers stopped coming to the queue after the closing time and checking if the customers' arrival was properly affected by the arrival rate, to make sure the code was working well.

Word Count: 75 words

HCs

#dataviz:

We generated data visualizations that were suitable for our data and helped express information to our audience in a well-formatted and detailed manner. The visualizations highlighted all the necessary information from our simulations and also explained the results of our theoretical and empirical analysis, in an easy-to-digest manner.

Word Count: 49 words.

#modeling:

We made a model based on the conditions that were provided to us while also building off of a pre-existing model, queueing model, and seeing how they connect. We also ran simulations to generate empirical results and then identified the patterns found in these results via a visualization. We also ran two different scenarios to explain why the empirical results were different in the two different scenarios for our model, and why was it important to check that.

Word Count: 74 words

#algorithms:

We implemented effective code to generate our model and simulation. We followed an algorithmic approach when going about writing the code thus giving it proper structure and ensuring that it works properly. We also made sure that the code was bug-free and ran test cases to check for edge cases.

Word Count: 50 words

#organization:

We organized the whole report in a sophisticated and well-formatted manner. This organization can be seen in our codebook as well as our writeup where everything was divided into sections and the sections were ordered in a sensible manner allowing the reader to go transition smoothly between the different sections when reading.

Word Count: 53 words

B. Work Division

Favour Okeke: write most of the modeling and simulation code, work with Junran on producing empirical analysis results, theoretical results writeup

Junran Shi: edit code and add docstrings and comments, work with Favour to produce empirical analysis results, introduction writeup, empirical results writeup, and overall project management

Yousaf: write test cases, modeling writeup, HC & LO tags

C. Code

CS166_Grocery_Store_Simulation

January 26, 2023

0.1 Modeling

[1]:
```
import heapq
from collections import deque
import numpy as np
import matplotlib.pyplot as plt
import scipy.stats as sts
```

[2]:
```
class Event:
    """
    Represent each event in the schedule. Used for priority queue.

    Attributes:
        timestamp (float): time for the event to take place
        function (function): function to call when the event takes place
    """
    def __init__(self, timestamp, function, *args, **kwargs):
        self.timestamp = timestamp
        self.function = function
        self.args = args
        self.kwargs = kwargs

    def __lt__(self, other):
        """
        Compare whether the timestamp of this event is earlier than another one.

        Inputs:
          other (Event): another Event instance to be compared with.

        Returns:
          bool: True if this event is earlier than the other one, False␣
↪otherwise.
        """
        return self.timestamp < other.timestamp

    def run(self, schedule):
        """
        Run the function associated with the Event when the it's the timestamp.
```

```
        Inputs:
          schedule (Schedule): the schedule with the current event.
        """
        self.function(schedule, *self.args, **self.kwargs)

class Schedule:
    """
    Class to manage events in a schedule.

    Attributes:
        now (float): current timestamp
        priority_queue (list): priority queue of events, comparison by their␣
↪timestamps.
    """
    def __init__(self):
        self.now = 0
        self.priority_queue = []

    def add_event_at(self, timestamp, function, *args, **kwargs):
        """
        Add event to the priority queue to take place at a specified timestamp.

        Inputs:
            timestamp (float): timestamp at which to execute the event.
            function (function): function to execute when the event takes place.
        """
        heapq.heappush(
            self.priority_queue,
            Event(timestamp, function, *args, **kwargs))

    def add_event_after(self, interval, function, *args, **kwargs):
        """
        Add event to the priority queue to take place a certain time interval␣
↪from now.

        Inputs:
            interval (float): time to wait from current time until executing␣
↪the event.
            function (function): function to execute when the event takes place.
        """
        self.add_event_at(self.now + interval, function, *args, **kwargs)

    def next_event_time(self):
        """
        Return the timestamp of the next event to be executed in the priority␣
↪queue.
```

```
        """
        return self.priority_queue[0].timestamp

    def run_next_event(self):
        """
        Execute the next event in the priority queue and update the current␣
↪time.
        """
        event = heapq.heappop(self.priority_queue)
        self.now = event.timestamp
        event.run(self)

    def __repr__(self):
        """
        Return information about the overall schedule.
        """
        return (
            f'Schedule() at time {self.now} ' +
            f'with {len(self.priority_queue)} events in the queue')

    def print_events(self):
        """
        Print information about the schedule and each event in the schedule.
        """
        print(repr(self))
        for event in sorted(self.priority_queue):
            print(f'   {event.timestamp}: {event.function.__name__}')
```

[3]:
```
class Customer:
    """
    Class to represent each customer and store information about their activity.

    Attributes:
        id (int): customer's unique ID.
        arrival_time (float): the time the customer joins the queue.
        cashier_start_time (float): the time the cashier starts serving the␣
↪customer.
        cashier_end_time (float): the time the cashier finishes serving the␣
↪customer.
        needed_assistance (bool): whether the customer needs assistance from␣
↪manager.
        manager_queue_arrival_time (float): the time the customer joins the␣
↪manager queue.
        manager_start_time (float): the time the manager starts helping the␣
↪customer.
        manager_end_time (float): the time the manager finishes helping the␣
↪customer.
```

```
    """
    def __init__(self, customer_id, arrival_time):
        self.id = customer_id
        self.arrival_time = arrival_time
        self.cashier_start_time = None
        self.cashier_end_time = None
        self.needed_assistance = False
        self.manager_queue_arrival_time = None
        self.manager_start_time = None
        self.manager_end_time = None

    def __str__(self):
        """
        Return all information about a customer's activity.

        Returns:
            string: contains information to be displayed about the customer's␣
↪activity.
        """
        str_ = f"#{self.id}:\nArrival: {round(self.arrival_time, 2)}"
        if self.cashier_start_time:
            str_ += f"\nCashier wait time: {round(self.cashier_start_time -␣
↪self.arrival_time, 2)}"
        if self.cashier_end_time:
            str_ += f"\nService time: {round(self.cashier_end_time - self.
↪cashier_start_time, 2)}\n"
        if self.needed_assistance:
            str_ += f"\nManager arrival time: {round(self.
↪manager_queue_arrival_time, 2)}"
        if self.manager_start_time:
            str_ += f"\nManager wait time: {round(self.manager_start_time -␣
↪self.manager_queue_arrival_time, 2)}"
        if self.manager_end_time:
            str_ += f"\nManager service time: {round(self.manager_end_time -␣
↪self.manager_start_time, 2)}\n"
        return str_
```

[4]:
```
class Queue:
    """
    Represents a queue in the grocery store, includes methods for the cashier␣
↪to serve the customer.

    Attributes:
        grocery_store (GroceryStore object): the grocery store instance.
        serving (bool): whether the cashier is currently serving a customer or␣
↪not.
```

```
        line (deque): a deque (double ended queue) representing the line of␣
↪this queue.
        service_distribution (scipy stats distribution): distribution of serve␣
↪times used to generate samples of service time.
        track_timestamps (list): list of the times that a customer starts being␣
↪served.
        track_queue_lens (list): list tracking the lengths of the line.
    """
    def __init__(self, service_distribution, grocery_store):
        self.grocery_store = grocery_store
        self.serving = False
        self.line = deque([])
        self.service_distribution = service_distribution
        self.track_timestamps = []
        self.track_queue_lens = []

    def done_serving(self, schedule, customer):
        """
        Method to call when a cashier is done serving a customer.

        Inputs:
            schedule (Schedule object): the schedule containing all events.
            customer (Customer object): the customer being served.
        """
        if self.serving:
            # update queue's serving status
            self.serving = False
            # update customer's cashier_end_time to current time
            customer.cashier_end_time = schedule.now

            #print(f"Done serving Customer {customer.id} at time␣
↪{round(schedule.now, 3)}")
            #print_status(self.grocery_store, schedule)

            # if there's a line in the queue, immediately start serving next␣
↪customer
            if self.line:
                schedule.add_event_after(0, self.serve_next)

            # 5% chance that customer needs assistance
            if np.random.rand() < 0.05:
                customer.needed_assistance = True
                customer.manager_queue_arrival_time = schedule.now
                # add event of customer joining the manager queue
                schedule.add_event_after(0, self.grocery_store.manager_arrival,␣
↪customer)
```

```
    def serve_next(self, schedule):
        """
        Method to serve next customer in line.

        Inputs:
            schedule (Schedule object): the schedule containing all events.
        """
        # check that cashier is available and there is a line
        if not self.serving and self.line:
            self.serving = True
            # get first customer in line
            customer = self.line.popleft()
            # update tracking lists: current time and line length
            self.track_timestamps.append(schedule.now)
            self.track_queue_lens.append(len(self.line))

            # update tracking list for
            prev_avg = self.grocery_store.track_avg_queue_lens[-1]
            n_queues = len(self.grocery_store.queues)
            new_avg = (prev_avg*n_queues - 1)/n_queues
            self.grocery_store.track_timestamps.append(schedule.now)
            self.grocery_store.track_avg_queue_lens.append(new_avg)

            # update the customer's cashier_start_time
            customer.cashier_start_time = schedule.now

            # print(f"Started serving Customer {customer.id} at time␣
 ↪{round(schedule.now, 3)}")
            # print_status(self.grocery_store, schedule)

            # use max() to avoid negative values from the distribution -- if␣
 ↪negative, then service time = 0
            # add event to finish serving the customer after certain service␣
 ↪time drawn from distribution
            schedule.add_event_after(max(self.service_distribution.rvs(), 0),␣
 ↪self.done_serving, customer)


# overall the structure of the ManagerQueue class is very similar to the Queue␣
 ↪class, so there are less detailed comments here.
class ManagerQueue:
    """
    Represents a manager queue, includes methods for the manager to help the␣
 ↪customer.
```

```
    Attributes:
        serving (bool): whether the manager is currently helping a customer or␣
↪not.
        line (deque): a deque (double ended queue) representing the line of␣
↪manager queue.
        manager_service_distribution (scipy stats distribution): distribution␣
↪of serve times used to generate samples of service time.
        track_timestamps (list): list of the times that a customer starts being␣
↪helped.
        track_queue_lens (list): list tracking the lengths of the line.
    """
    def __init__(self, manager_service_distribution):
        self.serving = False
        self.line = deque([])
        self.manager_service_distribution = manager_service_distribution
        self.track_timestamps = []
        self.track_queue_lens = []

    def done_serving(self, schedule, customer):
        """
        Method to call when the manager is done helping a customer.
        """
        if self.serving:
            self.serving = False
            customer.manager_end_time = schedule.now

            # print(f"Manager done serving Customer {customer.id} at time␣
↪{round(schedule.now, 3)}\n")

            if self.line:
                schedule.add_event_after(0, self.serve_next)

    def serve_next(self, schedule):
        """
        Method to call for the manager to help the next customer in line.
        """
        if not self.serving and self.line:
            self.serving = True
            customer = self.line.popleft()
            self.track_timestamps.append(schedule.now)
            self.track_queue_lens.append(len(self.line))
            customer.manager_start_time = schedule.now

            # print(f"Manager started serving Customer {customer.id} at time␣
↪{round(schedule.now, 3)}")
```

```
            schedule.add_event_after(max(self.manager_service_distribution.
↪rvs(), 0), self.done_serving, customer)
```

[5]:
```
class GroceryStore:
    """
    Represents the grocery store being modeled.

    Attributes:
        queues (list): list of Queue instances for each queue in the store.
        manager_queue (ManagerQueue object): manager queue.
        closing_time (int): number of minutes since opening.
        arrival_distribution (scipy stats distribution): distribution of␣
↪arrival time intervals.
        customer_id (int): current customer ID. increases by 1 for each new␣
↪customer.
        last_arrival (float): the arrival time of the last customer. when this␣
↪exceeds the closing time, we stop letting customers join.
        all_customers (list): list of all customers who have visited the store.
        track_timestamps (list): list of timestamps for tracking average queue␣
↪lengths.
        track_avg_queue_lens (list): list of average queue lengths of all␣
↪queues whenever a customer joins.
        rand_queue (bool): whether to randomly assign a customer to a queue.␣
↪default set to False. this is only used for testing.
    """
    def __init__(self, closing_time, n_queues, arrival_distribution,␣
↪service_distribution, manager_service_distribution, rand_queue = False):
        self.queues = [Queue(service_distribution, self) for _ in␣
↪range(n_queues)]
        self.manager_queue = ManagerQueue(manager_service_distribution)
        self.closing_time = closing_time
        self.arrival_distribution = arrival_distribution
        self.customer_id = 0
        self.last_arrival = 0
        self.all_customers = []
        self.track_timestamps = []
        self.track_avg_queue_lens = []
        self.rand_queue = rand_queue

    def find_shortest_line(self):
        """
        Find the index of the queue of shortest line currently.

        Returns:
            ind (int): index of shortest line.
        """
```

```
        ind = 0
        min_len = len(self.queues[0].line)
        # find queue with no line or queue with shortest line
        for i in range(1, len(self.queues)):
            if not self.queues[i].serving or len(self.queues[i].line) < min_len:
                ind = i
                min_len = len(self.queues[i].line)
        return ind

    # favour
    def random_gen_ind(self):
        n = len(self.queues)
        return np.random.randint(n)

    def arrival(self, schedule, customer):
        """
        Method to call when a customer arrives and joins a queue.

        Inputs:
            schedule (Schedule object): schedule of events of this grocery␣
↪store.
            customer (Customer object): customer who arrived.
        """
        # add customer to the queue with shortest line
        # add to random queue if self.rand_queue == True
        if self.rand_queue == False:
            ind = self.find_shortest_line()
        else:
            ind = self.random_gen_ind()

        self.queues[ind].line.append(customer)

        #print(f'Customer {customer.id} joined line {ind + 1}')
        #print_status(self, schedule)

        sum_queue_len = 0
        # update each queue's tracking with current time and line length
        for i in range(len(self.queues)):
            self.queues[i].track_timestamps.append(schedule.now)
            self.queues[i].track_queue_lens.append(len(self.queues[i].line))
            sum_queue_len += len(self.queues[i].line)
        avg_queue_len = sum_queue_len/len(self.queues)
        # track average queue length at current time
        self.track_timestamps.append(schedule.now)
        self.track_avg_queue_lens.append(avg_queue_len)

        self.queues[ind].serve_next(schedule)
```

```
    def manager_arrival(self, schedule, customer):
        """
        Method to call when a customer joins the manager queue.
        """
        self.manager_queue.line.append(customer)
        # track the current time and length of the manager queue.
        self.manager_queue.track_timestamps.append(schedule.now)
        self.manager_queue.track_queue_lens.append(len(self.manager_queue.line))
        # manager starts serving the customer if available, otherwise the␣
↪customer waits in line.
        self.manager_queue.serve_next(schedule)

    def get_customers_till_closing(self, schedule):
        """
        Generate all customer arrivals until closing time.
        """
        # only allow customers to arrive before the store's closing time
        while self.last_arrival < self.closing_time:
            if self.last_arrival > self.closing_time:
                break
            # draw arrival time interval from distribution
            arrival_interval = self.arrival_distribution.rvs()
            self.customer_id += 1
            # add interval to arrival time tracker
            self.last_arrival += arrival_interval
            customer = Customer(self.customer_id, self.last_arrival)
            self.all_customers.append(customer)
            # schedule event for customer to arrive at the store
            schedule.add_event_at(self.last_arrival, self.arrival, customer)

    def run(self, schedule):
        """
        Start running the similation by generating all customer arrivals.
        """
        self.get_customers_till_closing(schedule)


def print_status(grocery_store, schedule):
    """
    Print the current status of the grocery store. Used for checking the model.
    """
    print(f"At timestamp = {schedule.now}, the grocery store queues:")
    for i in range(len(grocery_store.queues)):
        print(f"Queue #{i+1}: {len(grocery_store.queues[i].line)}   in the␣
↪queue")
    print()
```

0.2 Simulation

[6]:
```
def run_simulation(closing_time, arrival_distribution, service_distribution,␣
↪manager_service_distribution, run_until, n_queues, print_queue = True,␣
↪rand_queue = False):
    """
    Function to run the simulation given inputs and give status of the store␣
↪after the simulation ends.

    Inputs:
        closing_time (int): number of minutes from opening
        arrival_distribution (scipy stats distribution): distribution of␣
↪arrival time intervals
        service_distribution (scipy stats distribution): distribution of␣
↪service times
        manager_service_distribution (scipy stats distribution): distribution␣
↪of manager service times
        run_until: (int) length of the simulation
        n_queues (int): number of queues in the store
        print (bool): whether to print information about the queues after␣
↪simulation ends

    Returns:
        store (GroceryStore object)
    """
    schedule = Schedule()
    store = GroceryStore(closing_time, n_queues, arrival_distribution,
                         service_distribution, manager_service_distribution,␣
↪rand_queue)
    # initialize simulation by scheduling arrivals
    store.run(schedule)
    # if the simulation is scheduled to end before closing time, keep running␣
↪next event until hitting run_until
    if run_until < closing_time:
        while schedule.now <= run_until:
            schedule.run_next_event()
    # if the simulation is scheduled to end after closing time, run simulation␣
↪until completion (clearing the schedule)
    else:
        while schedule.priority_queue:
            schedule.run_next_event()
    if print_queue == True:
        print("At the end of the simulation:")
        for i in range(len(store.queues)):
            print(f"Queue #{i+1}: {len(store.queues[i].line)}   in the queue")
        print()
    return store
```

[7]:
```
# DO NOT RERUN THIS CELL
# this output is use to check that customers join the shortest queue. simulation for 5 timestamps.

arrival_distribution = sts.expon(scale=1/1)
service_distribution = sts.norm(loc=3, scale=1)
manager_service_distribution = sts.norm(loc=5, scale=2)

# closing_time = 660 (minutes) -- that is 11 hours (8pm is 11 hours from opening time of 9am)
# run for 500 minutes
# 3 queues
grocery_store = run_simulation(660, arrival_distribution, service_distribution,
                               manager_service_distribution, 5, 3)
```

```
At the end of the simulation:
Queue #1: 0   in the queue
Queue #2: 0   in the queue
Queue #3: 0   in the queue
```

[8]:
```
arrival_distribution = sts.expon(scale=1/1)
service_distribution = sts.norm(loc=3, scale=1)
manager_service_distribution = sts.norm(loc=5, scale=2)

# closing_time = 660 (minutes) -- that is 11 hours (8pm is 11 hours from opening time of 9am)
# run for 500 timestamps -- not till completion
# 4 queues
grocery_store = run_simulation(660, arrival_distribution, service_distribution,
                               manager_service_distribution, 500, 4)
```

```
At the end of the simulation:
Queue #1: 1   in the queue
Queue #2: 0   in the queue
Queue #3: 0   in the queue
Queue #4: 0   in the queue
```

0.3 Theoretical Analysis

[9]:
```
theo_avg_wait_time = []
theo_avg_response_time = []
theo_avg_queue_length = []

sigma_cashier = 1
tau_cashier = 3
sigma_manager = 2
```

```
tau_manager = 5

for n in range(4, 10):
    lambda_cashier = 1/n
    lambda_manager = 0.05
    rho_cashier = lambda_cashier * tau_cashier
    rho_manager = lambda_manager * tau_manager

    avg_wait_time = ((rho_cashier * tau_cashier)/(2 * (1-rho_cashier))) * (1 +␣
↪(sigma_cashier**2)/(tau_cashier**2))
    # include wait time in manager line
    avg_manager_wait_time = 0.05 * (((rho_manager * tau_manager)/(2 *␣
↪(1-rho_manager))) * (1 + (sigma_manager**2)/(tau_manager**2)))
    theo_avg_wait_time.append(avg_wait_time + avg_manager_wait_time)

    avg_response_time = tau_cashier + tau_manager * 0.05 + avg_wait_time
    theo_avg_response_time.append(avg_response_time)

    avg_queue_length = avg_wait_time * lambda_cashier
    theo_avg_queue_length.append(avg_queue_length)

    print(f'Number of queues = {n}')
    print(f"---Average wait time: {round(avg_wait_time + avg_manager_wait_time,␣
↪2)}")
    print(f"---Average response time: {round(avg_response_time, 2)}")
    print(f"---Average queue length: {round(avg_queue_length, 2)}")
    print()
```

```
Number of queues = 4
---Average wait time: 5.05
---Average response time: 8.25
---Average queue length: 1.25

Number of queues = 5
---Average wait time: 2.55
---Average response time: 5.75
---Average queue length: 0.5

Number of queues = 6
---Average wait time: 1.72
---Average response time: 4.92
---Average queue length: 0.28

Number of queues = 7
---Average wait time: 1.3
---Average response time: 4.5
---Average queue length: 0.18
```

```
Number of queues = 8
---Average wait time: 1.05
---Average response time: 4.25
---Average queue length: 0.12

Number of queues = 9
---Average wait time: 0.88
---Average response time: 4.08
---Average queue length: 0.09
```

0.4 Empirical Analysis

[10]:
```python
def find_metrics(grocery_store):
    """
    Find the four metrics for the grocery store after the simulation. Store as attributes.
    """
    grocery_store.wait_times = []
    grocery_store.service_times = []
    grocery_store.response_times = []

    for customer in grocery_store.all_customers:
        if customer.cashier_start_time and customer.cashier_end_time:
            wait_time = round(customer.cashier_start_time - customer.arrival_time, 2)
            service_time = round(customer.cashier_end_time - customer.cashier_start_time, 2)
            if customer.manager_start_time and customer.manager_end_time:
                wait_time += round(customer.manager_start_time - customer.manager_queue_arrival_time, 2)
                service_time += round(customer.manager_end_time - customer.manager_start_time, 2)
            grocery_store.wait_times.append(wait_time)
            grocery_store.service_times.append(service_time)
            grocery_store.response_times.append(wait_time + service_time)

    # metric: avg waiting time
    grocery_store.avg_wait_time = np.average(grocery_store.wait_times)

    # metric: avg response time
    grocery_store.avg_response_time = np.average(grocery_store.response_times)

    # metric: max queue length
    m = 0
    for i in range(len(grocery_store.queues)):
```

```
        if max(grocery_store.queues[i].track_queue_lens) > m:
            m = max(grocery_store.queues[i].track_queue_lens)
    grocery_store.max_queue_length = m

    # metric: average queue length
    grocery_store.avg_queue_length = np.average(grocery_store.
↪track_avg_queue_lens)


def compute_95_interval(data):
    """
    Compute the average queue length, standard error, and 95% confidence␣
↪interval.
    """
    m = np.mean(data)
    m_approx = round(m, 2)
    t = sts.sem(data)
    t_approx = round(t, 2)
    return m_approx, t_approx
```

[11]:
```
# save experiment data
shortest_queue_data = {}
random_queue_data = {}

def trials(n_trials, n_queues):
    shortest_queue_data[n_queues] = [],[],[],[]
    random_queue_data[n_queues] = [],[],[],[]

    for _ in range(n_trials):
        # shortst queue approach
        grocery_store = run_simulation(660, arrival_distribution,␣
↪service_distribution,
                                       manager_service_distribution, 700,␣
↪n_queues, print_queue = False)
        find_metrics(grocery_store)
        # save metrics
        shortest_queue_data[n_queues][0].append(grocery_store.avg_wait_time)
        shortest_queue_data[n_queues][1].append(grocery_store.avg_response_time)
        shortest_queue_data[n_queues][2].append(grocery_store.max_queue_length)
        shortest_queue_data[n_queues][3].append(grocery_store.avg_queue_length)

        # random queue approach (note rand_queue = True)
        grocery_store = run_simulation(660, arrival_distribution,␣
↪service_distribution,
                                       manager_service_distribution, 700,␣
↪n_queues, print_queue = False, rand_queue = True)
        find_metrics(grocery_store)
```

```
        random_queue_data[n_queues][0].append(grocery_store.avg_wait_time)
        random_queue_data[n_queues][1].append(grocery_store.avg_response_time)
        random_queue_data[n_queues][2].append(grocery_store.max_queue_length)
        random_queue_data[n_queues][3].append(grocery_store.avg_queue_length)
```

[12]:
```
# run 100 trials for each n_queues
for n_queues in range(4,10):
    trials(100, n_queues)
```

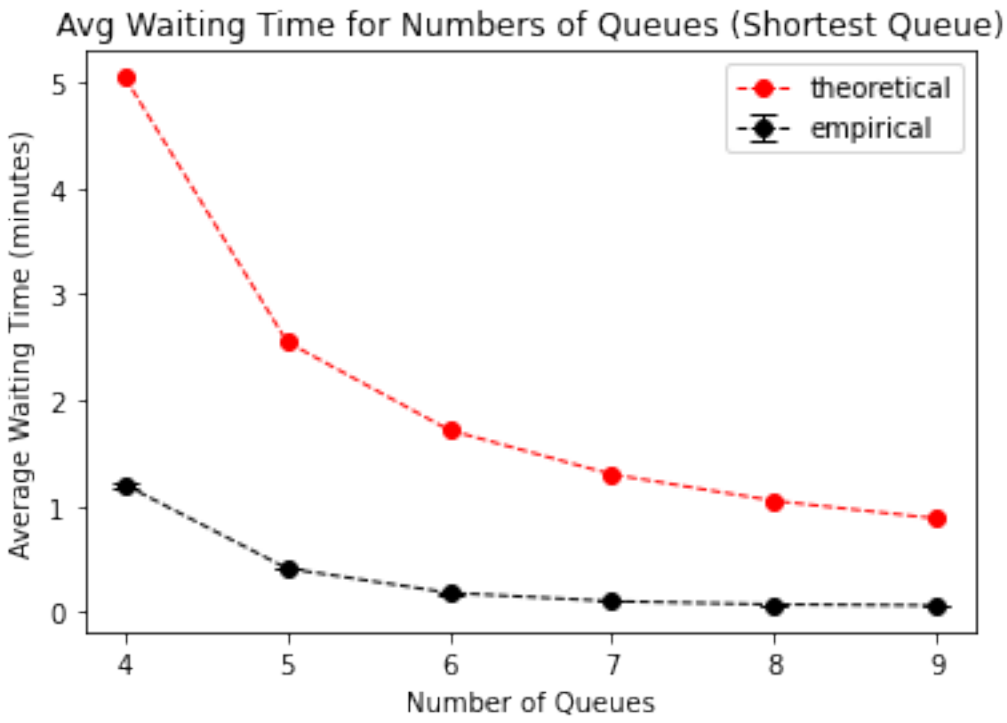
[13]:
```
shortest_queue_plot_data = {}
random_queue_plot_data = {}

# find mean and standard deviation for metrics for plotting
for metric in range(0,4):
    shortest_queue_plot_data[metric] = [[],[]]
    random_queue_plot_data[metric] = [[],[]]
    for n_queues in range(4,10):
        short_mean, short_se =␣
↪compute_95_interval(shortest_queue_data[n_queues][metric])
        rand_mean, rand_se =␣
↪compute_95_interval(random_queue_data[n_queues][metric])
        shortest_queue_plot_data[metric][0].append(short_mean)
        shortest_queue_plot_data[metric][1].append(short_se)
        random_queue_plot_data[metric][0].append(rand_mean)
        random_queue_plot_data[metric][1].append(rand_se)
```

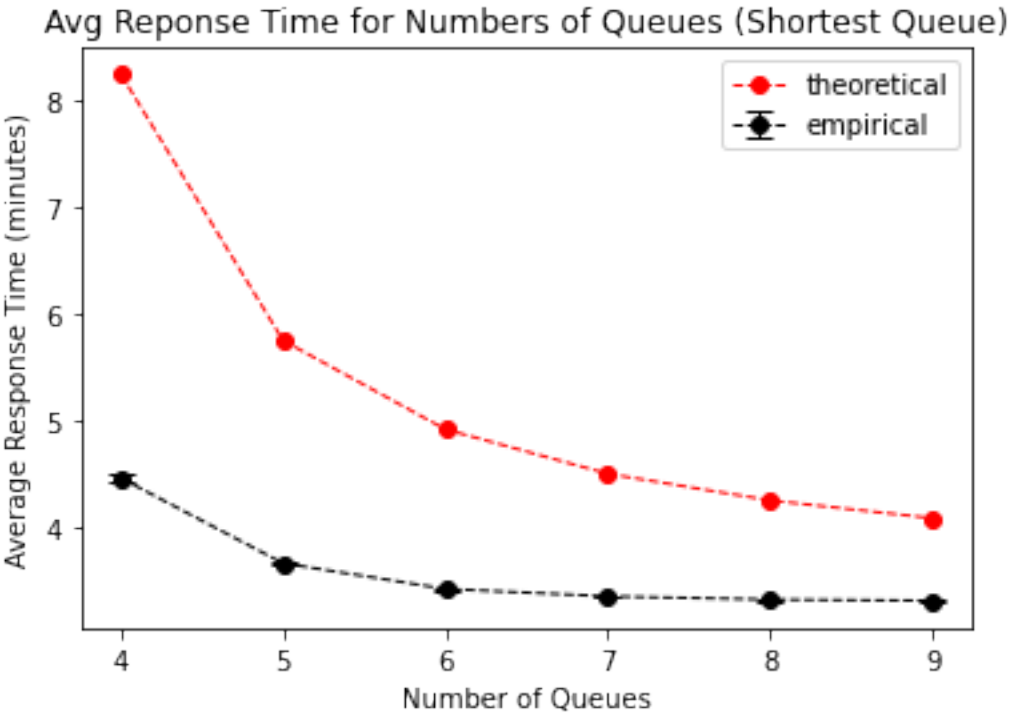
[14]:
```
def error_plot_theo(y, y_error, theoretical, xlabel, ylabel, title):
    """
    Show error plot with 95% confidence interval, including theoretical values.
    """
    plt.figure()
    plt.errorbar(range(4,10), y, y_error,
                 color='black', marker='o', capsize=5, linestyle='--',␣
↪linewidth=1, label='empirical')
    plt.plot(range(4,10), theoretical,
             color='red', marker='o',  linestyle='--', linewidth=1,␣
↪label='theoretical')
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    plt.legend()
    plt.title(title)
    plt.show()

def error_plot(y, y_error, xlabel, ylabel, title):
    """
    Show error plot with 95% confidence interval.
```

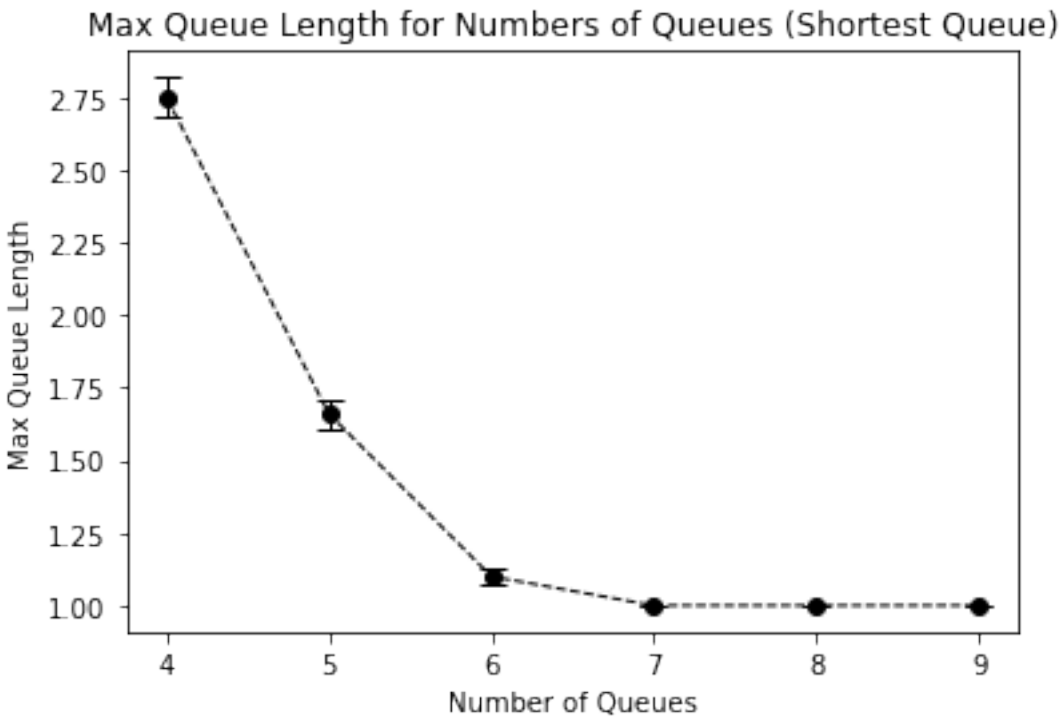
```
    """
    plt.figure()
    plt.errorbar(range(4,10), y, y_error,
                 color='black', marker='o', capsize=5, linestyle='--',␣
↪linewidth=1)
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    plt.title(title)
    plt.show()
```

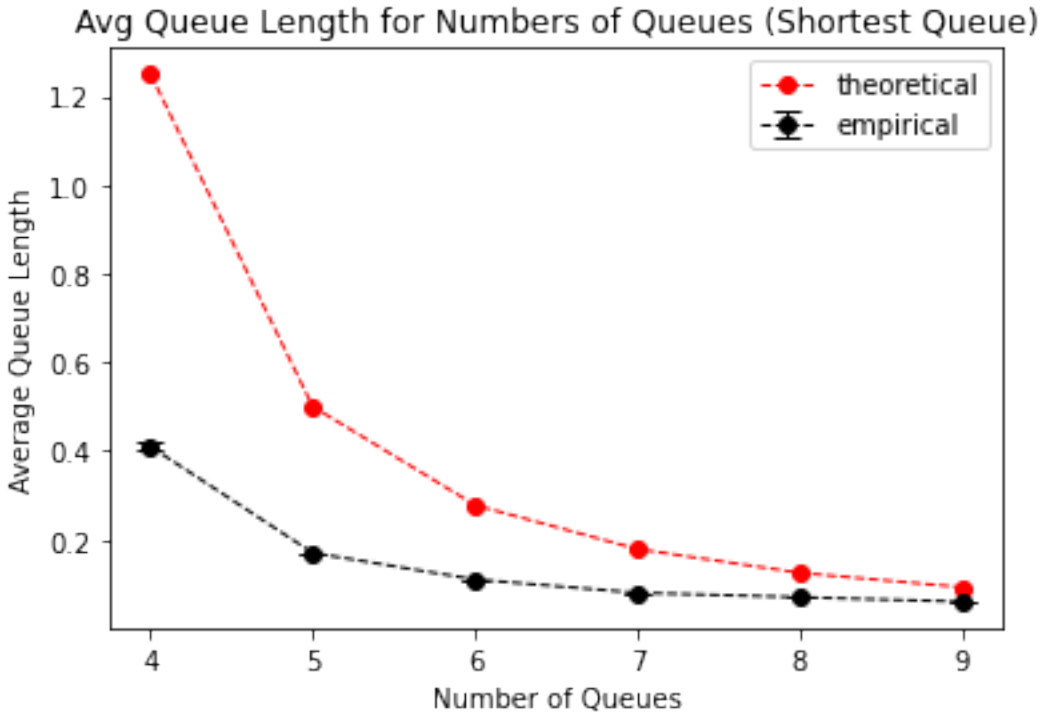
0.4.1 Shortest Queue

[15]:
```
error_plot_theo(shortest_queue_plot_data[0][0],␣
↪shortest_queue_plot_data[0][1],theo_avg_wait_time,'Number of Queues',␣
↪'Average Waiting Time (minutes)', 'Avg Waiting Time for Numbers of Queues␣
↪(Shortest Queue)')
error_plot_theo(shortest_queue_plot_data[1][0],␣
↪shortest_queue_plot_data[1][1],theo_avg_response_time,'Number of Queues',␣
↪'Average Response Time (minutes)', 'Avg Reponse Time for Numbers of Queues␣
↪(Shortest Queue)')
error_plot(shortest_queue_plot_data[2][0],␣
↪shortest_queue_plot_data[2][1],'Number of Queues', 'Max Queue Length', 'Max␣
↪Queue Length for Numbers of Queues (Shortest Queue)')
error_plot_theo(shortest_queue_plot_data[3][0],␣
↪shortest_queue_plot_data[3][1],theo_avg_queue_length,'Number of Queues',␣
↪'Average Queue Length', 'Avg Queue Length for Numbers of Queues (Shortest␣
↪Queue)')
```


Avg Waiting Time for Numbers of Queues (Shortest Queue)
theoretical
empirical
Average Waiting Time (minutes)
0
1
2
3
4
5
4
5
6
7
8
9
Number of Queues


Avg Reponse Time for Numbers of Queues (Shortest Queue)
theoretical
empirical
Average Response Time (minutes)
4
5
6
7
8
4
5
6
7
8
9
Number of Queues


Max Queue Length for Numbers of Queues (Shortest Queue)
Max Queue Length
2.75
2.50
2.25
2.00
1.75
1.50
1.25
1.00
4
5
6
7
8
9
Number of Queues

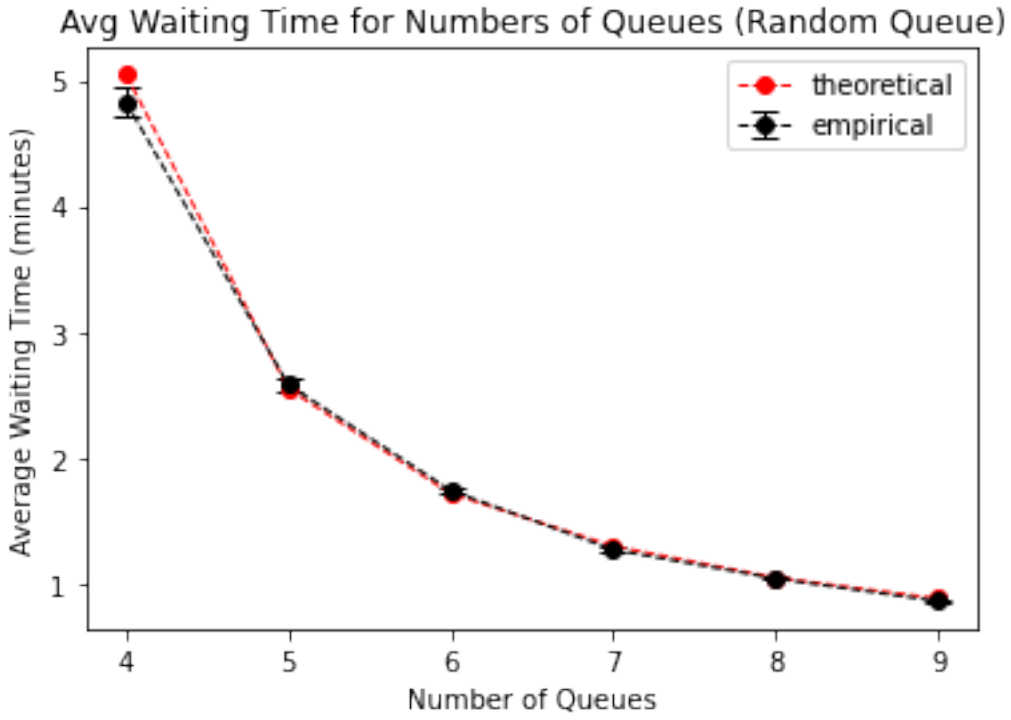

Avg Queue Length for Numbers of Queues (Shortest Queue)
theoretical
empirical
Average Queue Length
1.2
1.0
0.8
0.6
0.4
0.2
4
5
6
7
8
9
Number of Queues

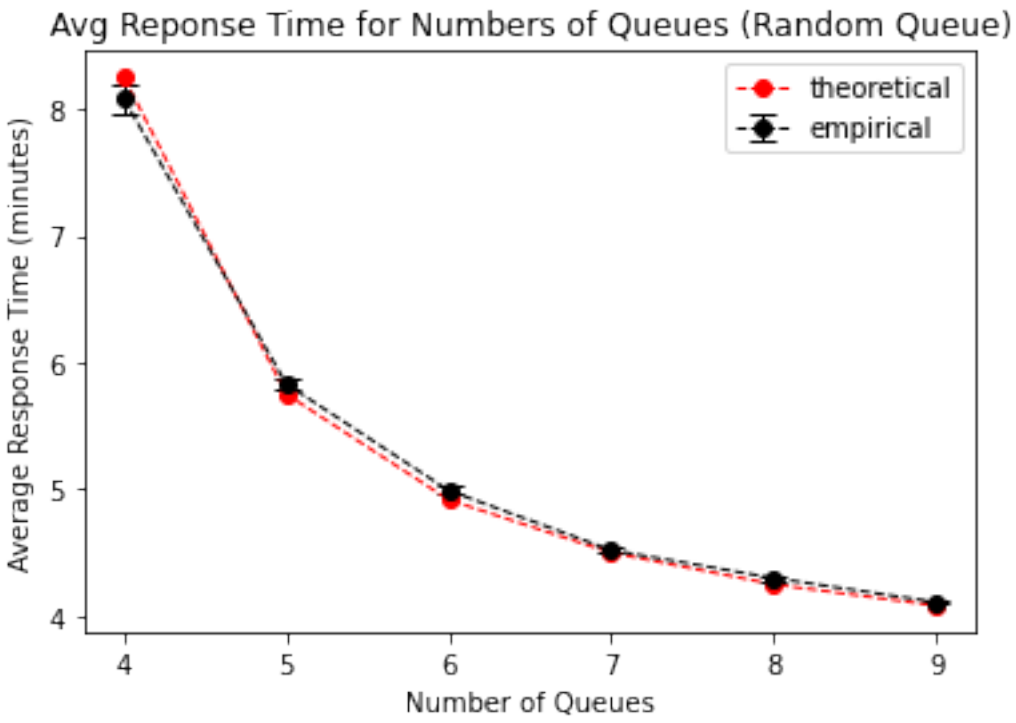
[16]:
```
for n in range(4,10):
    print(f'Number of queues = {n}')
    print(f"---Average wait time: {shortest_queue_plot_data[0][0][n-4]}")
    print(f"---Average response time: {shortest_queue_plot_data[1][0][n-4]}")
    print(f"---Maximum queue length: {shortest_queue_plot_data[2][0][n-4]}")
    print(f"---Average queue length: {shortest_queue_plot_data[3][0][n-4]}")
    print()
```

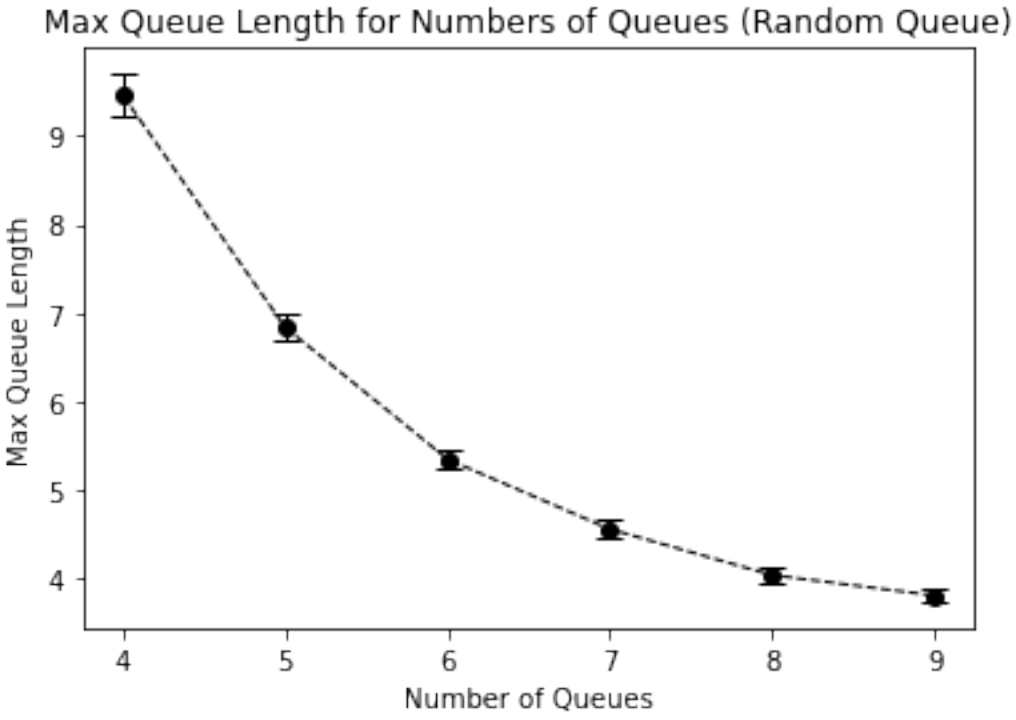
```
Number of queues = 4
---Average wait time: 1.2
---Average response time: 4.46
---Maximum queue length: 2.75
---Average queue length: 0.41

Number of queues = 5
---Average wait time: 0.41
---Average response time: 3.66
---Maximum queue length: 1.66
---Average queue length: 0.17

Number of queues = 6
---Average wait time: 0.18
---Average response time: 3.42
---Maximum queue length: 1.1
---Average queue length: 0.11

Number of queues = 7
---Average wait time: 0.1
---Average response time: 3.35
---Maximum queue length: 1.0
---Average queue length: 0.08

Number of queues = 8
---Average wait time: 0.07
---Average response time: 3.32
---Maximum queue length: 1.0
---Average queue length: 0.07

Number of queues = 9
---Average wait time: 0.06
---Average response time: 3.31
---Maximum queue length: 1.0
---Average queue length: 0.06
```

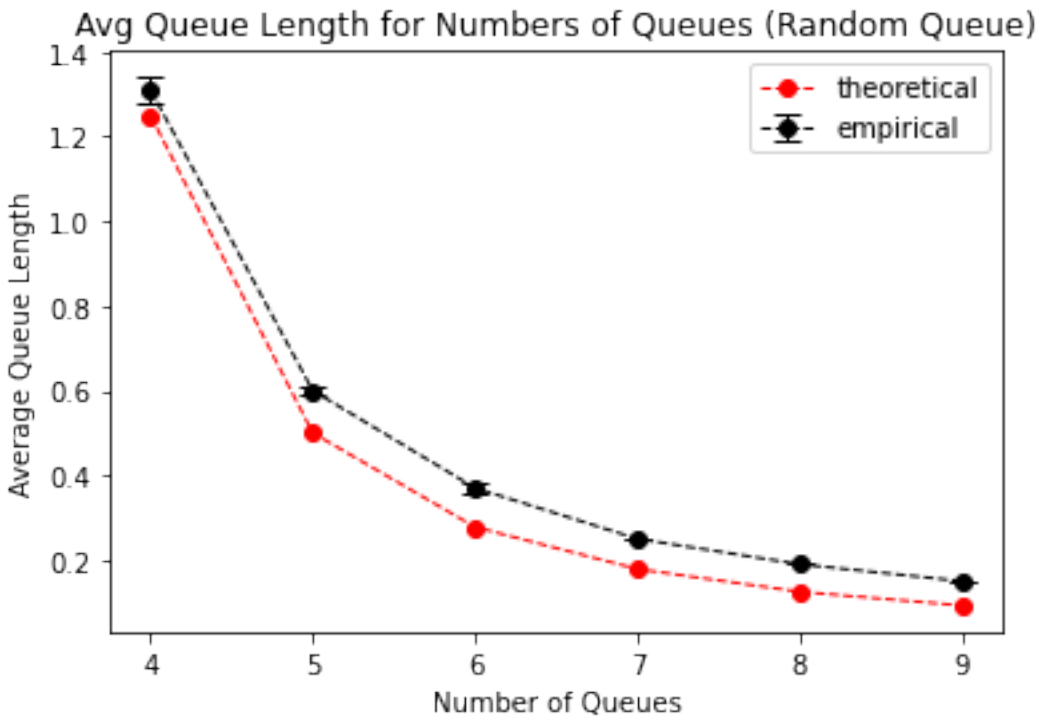
0.4.2 Random Queue

[17]:
```
error_plot_theo(random_queue_plot_data[0][0],␣
↪random_queue_plot_data[0][1],theo_avg_wait_time,'Number of Queues', 'Average␣
↪Waiting Time (minutes)', 'Avg Waiting Time for Numbers of Queues (Random␣
↪Queue)')
error_plot_theo(random_queue_plot_data[1][0],␣
↪random_queue_plot_data[1][1],theo_avg_response_time,'Number of Queues',␣
↪'Average Response Time (minutes)', 'Avg Reponse Time for Numbers of Queues␣
↪(Random Queue)')
error_plot(random_queue_plot_data[2][0], random_queue_plot_data[2][1],'Number␣
↪of Queues', 'Max Queue Length', 'Max Queue Length for Numbers of Queues␣
↪(Random Queue)')
error_plot_theo(random_queue_plot_data[3][0],␣
↪random_queue_plot_data[3][1],theo_avg_queue_length,'Number of Queues',␣
↪'Average Queue Length', 'Avg Queue Length for Numbers of Queues (Random␣
↪Queue)')
```




Avg Reponse Time for Numbers of Queues (Random Queue)
theoretical
empirical
Average Response Time (minutes)
8
7
6
5
4
4
5
6
7
8
9
Number of Queues


Max Queue Length for Numbers of Queues (Random Queue)
Max Queue Length
9
8
7
6
5
4
4
5
6
7
8
9
Number of Queues



[18]:
```
for n in range(4,10):
    print(f'Number of queues = {n}')
    print(f"---Average wait time: {random_queue_plot_data[0][0][n-4]}")
    print(f"---Average response time: {random_queue_plot_data[1][0][n-4]}")
    print(f"---Maximum queue length: {random_queue_plot_data[2][0][n-4]}")
    print(f"---Average queue length: {random_queue_plot_data[3][0][n-4]}")
    print()
```

```
Number of queues = 4
---Average wait time: 4.83
---Average response time: 8.08
---Maximum queue length: 9.46
---Average queue length: 1.31

Number of queues = 5
---Average wait time: 2.58
---Average response time: 5.83
---Maximum queue length: 6.84
---Average queue length: 0.6

Number of queues = 6
---Average wait time: 1.74
---Average response time: 4.99
---Maximum queue length: 5.35
```

```
---Average queue length: 0.37

Number of queues = 7
---Average wait time: 1.27
---Average response time: 4.52
---Maximum queue length: 4.56
---Average queue length: 0.25

Number of queues = 8
---Average wait time: 1.04
---Average response time: 4.3
---Maximum queue length: 4.04
---Average queue length: 0.19

Number of queues = 9
---Average wait time: 0.86
---Average response time: 4.11
---Maximum queue length: 3.81
---Average queue length: 0.15
```

0.5 Test Cases

We are doing this at the end of the notebook because we need some of the function defined previously.

[19]:
```
# Test Case 1

# Do queues reach 0 after the closing time has been achieved
# meaning no further customers are taken in and all customers in the queue are␣
 ↪served?

arrival_distribution = sts.expon(scale=1/1)
service_distribution = sts.norm(loc=3, scale=1)
manager_service_distribution = sts.norm(loc=5, scale=2)

# run_until = 660 -- as long as we hit closing hour, the simulation will be run␣
 ↪till completion
for i in range(1, 8):
    print(f'Number of queues = {i}')
    grocery_store = run_simulation(660, arrival_distribution,␣
 ↪service_distribution,
                                   manager_service_distribution, 660, i)
```

```
Number of queues = 1
At the end of the simulation:
Queue #1: 0   in the queue
```

```
Number of queues = 2
At the end of the simulation:
Queue #1: 0   in the queue
Queue #2: 0   in the queue

Number of queues = 3
At the end of the simulation:
Queue #1: 0   in the queue
Queue #2: 0   in the queue
Queue #3: 0   in the queue

Number of queues = 4
At the end of the simulation:
Queue #1: 0   in the queue
Queue #2: 0   in the queue
Queue #3: 0   in the queue
Queue #4: 0   in the queue

Number of queues = 5
At the end of the simulation:
Queue #1: 0   in the queue
Queue #2: 0   in the queue
Queue #3: 0   in the queue
Queue #4: 0   in the queue
Queue #5: 0   in the queue

Number of queues = 6
At the end of the simulation:
Queue #1: 0   in the queue
Queue #2: 0   in the queue
Queue #3: 0   in the queue
Queue #4: 0   in the queue
Queue #5: 0   in the queue
Queue #6: 0   in the queue

Number of queues = 7
At the end of the simulation:
Queue #1: 0   in the queue
Queue #2: 0   in the queue
Queue #3: 0   in the queue
Queue #4: 0   in the queue
Queue #5: 0   in the queue
Queue #6: 0   in the queue
Queue #7: 0   in the queue
```

[20]:
```
# Test Case 2

# Does increasing the rate of arrival makes sure that the queue length keeps␣
 ↪increasing?

# arrival rate is 100 customers per minute
arrival_distribution = sts.expon(scale=1/100)
service_distribution = sts.norm(loc=3, scale=1)
manager_service_distribution = sts.norm(loc=5, scale=2)

grocery_store = run_simulation(660, arrival_distribution, service_distribution,
                               manager_service_distribution, 500, 10)
```

```
At the end of the simulation:
Queue #1: 4821   in the queue
Queue #2: 4821   in the queue
Queue #3: 4821   in the queue
Queue #4: 4821   in the queue
Queue #5: 4821   in the queue
Queue #6: 4820   in the queue
Queue #7: 4820   in the queue
Queue #8: 4820   in the queue
Queue #9: 4820   in the queue
Queue #10: 4820   in the queue
```

[21]:
```
# Test Case 3

# checking that when we only have one queue, the output metrics align with␣
 ↪theoretical results

# arrival rate of 1 customer per minute
# 0.8 minutes per customer service time
arrival_distribution = sts.expon(scale=1/1)
service_distribution = sts.norm(loc=0.8, scale=0.2)
manager_service_distribution = sts.norm(loc=2, scale=1)

waiting_times = []
avg_queue_lengths = []
for i in range(100):
    grocery_store = run_simulation(660, arrival_distribution,␣
 ↪service_distribution,
                                   manager_service_distribution, 660, 1,␣
 ↪print_queue = False)
    find_metrics(grocery_store)
    waiting_times.append(grocery_store.avg_wait_time)
    avg_queue_lengths.append(grocery_store.avg_queue_length)
```

```
print(f'Empirical results: average waiting time = {np.average(waiting_times)},␣
 ↪average queue length = {np.average(avg_queue_lengths)}')
```

```
Empirical results: average waiting time = 1.6842107885397448, average queue
length = 2.175221556937642
```

[22]:
```
sigma_cashier = 0.2
tau_cashier = 0.8
sigma_manager = 1
tau_manager = 2

n=1
lambda_cashier = 1/n
lambda_manager = 0.05
rho_cashier = lambda_cashier * tau_cashier
rho_manager = lambda_manager * tau_manager

avg_wait_time = ((rho_cashier * tau_cashier)/(2 * (1-rho_cashier))) * (1 +␣
 ↪(sigma_cashier**2)/(tau_cashier**2))
avg_manager_wait_time = 0.05 * (((rho_manager * tau_manager)/(2 *␣
 ↪(1-rho_manager))) * (1 + (sigma_manager**2)/(tau_manager**2)))
theo_avg_wait_time.append(avg_wait_time + avg_manager_wait_time)

avg_queue_length = avg_wait_time * lambda_cashier
theo_avg_queue_length.append(avg_queue_length)

print(f'Number of queues = {n}')
print(f"---Average wait time: {round(avg_wait_time + avg_manager_wait_time,␣
 ↪2)}")
print(f"---Average queue length: {round(avg_queue_length, 2)}")
```

```
Number of queues = 1
---Average wait time: 1.71
---Average queue length: 1.7
```

Test Case 3: average wait time aligns with the theoretical analysis. There is a gap in average queue length, but we obsere the same difference in all of our experiments.