Everyone's a Waiter: A Data-Driven Queuing Simulation Model of Mike's Clam Shack

Natalie Robinson

Follow this and additional works at: https://digitalcommons.library.umaine.edu/honors

Part of the Numerical Analysis and Computation Commons

This Honors Thesis is brought to you for free and open access by DigitalCommons@UMaine. It has been accepted for inclusion in Honors College by an authorized administrator of DigitalCommons@UMaine. For more information, please contact um.library.technical.services@maine.edu.
EVERYONE’S A WAITER:
A DATA-DRIVEN QUEUING SIMULATION MODEL OF MIKE’S CLAM SHACK

by
Natalie Robinson

A Thesis Submitted in Partial Fulfillment
of the Requirements for a Degree with Honors
(Mathematics)

The Honors College
University of Maine

May 2023

Advisory Committee:
David Hiebeler, Professor of Mathematics, Advisor
Jane Wang, Assistant Professor of Mathematics
Aden Forrow, Assistant Professor of Mathematics
This thesis seeks to understand the mathematical foundation of several prominent concepts in queuing theory and apply them to gain a better understanding of nightly business levels and dining room queue behavior during the summer tourist season at Mike’s Clam Shack, which is a restaurant located in Wells, Maine. To do so, a variety of queue and server section data has been collected from Mike’s and analyzed to determine probability distributions for interarrival and service times. In addition, a queuing simulation model has been constructed in the R Programming Language, which uses this data to generate dining room and queue activity on a given month and week day. Suggestions for improving the model’s authenticity are also provided.
ACKNOWLEDGEMENTS

So many thank-you’s are in order for this thesis! This project would not have been possible without the help of these wonderful people, and I’m so grateful for each and every one of them.

First, to Mike, Mary, Max, and the entire staff at Mike’s Clam Shack – thank you for not only allowing me to collect data from the restaurant for this thesis, but for having me as an employee for the past four years as well. I have loved being a part of the “clamily!”

To my advisor, David Hiebeler – this thesis would not have existed without you. Thank you so much for supporting and guiding me through the ups and downs of this process, as well as my college career in general!

To my committee members, Jane Wang and Aden Forrow – thank you for your enthusiasm, support, and advice throughout the thesis process! It has been a privilege to work with and get to know you both.

And finally, to Mom, Dad, Ryan, and the rest of my family and friends – thank you for your constant love and encouragement along the way! I’m here today because of you, and I love you all so much.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Figures</td>
<td>v</td>
</tr>
<tr>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td>Chapter I: Mathematical Background</td>
<td>2</td>
</tr>
<tr>
<td>Random Variables</td>
<td>2</td>
</tr>
<tr>
<td>The Exponential Distribution</td>
<td>3</td>
</tr>
<tr>
<td>Poisson Processes</td>
<td>5</td>
</tr>
<tr>
<td>Chapter II: Queuing Theory</td>
<td>7</td>
</tr>
<tr>
<td>Overview</td>
<td>7</td>
</tr>
<tr>
<td>The M/G/s Queuing Model</td>
<td>8</td>
</tr>
<tr>
<td>Chapter III: Data Collection &amp; Estimation</td>
<td>10</td>
</tr>
<tr>
<td>Restaurant Background</td>
<td>10</td>
</tr>
<tr>
<td>Data Collection</td>
<td>11</td>
</tr>
<tr>
<td>Data Limitations</td>
<td>12</td>
</tr>
<tr>
<td>Goodness-of-Fit Testing for the Interarrival Time Distribution</td>
<td>12</td>
</tr>
<tr>
<td>Estimating the Service Time Distribution</td>
<td>16</td>
</tr>
<tr>
<td>Chapter IV: Queuing Simulation Model</td>
<td>20</td>
</tr>
<tr>
<td>Simulation Code</td>
<td>20</td>
</tr>
<tr>
<td>Simulation Outcomes</td>
<td>25</td>
</tr>
<tr>
<td>Chapter V: Conclusions &amp; Recommendations for Future Study</td>
<td>29</td>
</tr>
<tr>
<td>References</td>
<td>32</td>
</tr>
<tr>
<td>Appendix: R Code for Queuing Simulation Model</td>
<td>33</td>
</tr>
<tr>
<td>Author’s Biography</td>
<td>47</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

Figure 1 – General Shape of the Exponential Distribution 4
Table 1 – Bin Division of Interarrival Times for Chi-Square Test 13
Figure 2 – Observed and Expected Counts for Interarrival Times 14
Table 2 – Parameters and Results of Chi-Square Test 15
Table 3 – Estimated Service Times of Parties 17
Table 4 – Valid Inputs for the Main Simulation Function 23
Table 5 – Comparison of Average Party Counts 26
INTRODUCTION

In today’s competitive restaurant industry, utilizing queuing theory in business decision-making is more important than ever. Queuing analysis is particularly useful in helping restaurants predict daily business levels, which in turn can help them determine methods of minimizing wait times and providing more efficient service to maximize profits. This thesis project seeks to use queuing theory and its mathematical background to analyze nightly business levels and queue behavior at Mike’s Clam Shack. Business levels at Mike’s are highly influenced by summer tourism, making it all the more important for the restaurant to use every available resource to prepare for high customer volumes in the summer months.

To do so, we discuss queuing theory and some of its most prominent mathematical concepts, and use this foundation to examine queue and service data collected from Mike’s Clam Shack. We then attempt to construct a queuing simulation model of the restaurant in the R Programming Language to repeatedly generate nightly dining room and queue activity. The goal of the simulation is to provide the management at Mike’s with a method of exploring “what-if” scenarios involving adjustments to staffing levels, restaurant capacities, and customer volumes at different times during the summer season. Exploring these types of scenarios can aid management in determining the ideal number of servers and tables needed on different months and days of the week to efficiently handle the amount of parties dining in the restaurant while minimizing queue lengths and wait times. A discussion of the results of the simulation and proposed methods for improvement follows.
Randomness is an inherent property of many processes that we experience or interact with on an everyday basis. Examples of this can be found in nearly every aspect of life, ranging from weather forecasts to estimated restaurant waits, and everything in between. Many worldly processes like these have the ability to generate outcomes that cannot be predicted exactly [2]. We refer to a specific desired outcome as a random event, which is defined as an event that occurs with a consistent relative frequency when a process is repeated many times [2]. While it is impossible to predict the occurrence of such events with total certainty, we can use probability concepts to better understand them by approximating the likelihood of different possible outcomes of a process. Oftentimes, we do so by describing the outcomes of the process as a random variable.

A random variable is a real-valued function that is defined over the set of all possible outcomes of the experiment being performed [6]. There are two main types of random variables: discrete and continuous. A discrete random variable can take on a finite or countably infinite number of distinct values [2]. For this type of random variable, we examine the probability of observing a particular outcome value it can take on. A continuous random variable can take on an uncountably infinite number of values in a given interval [2]. For this type of random variable, the probability of it equaling any exact particular value is zero due to its infinite nature, so we examine the probability that
its outcome value is within a given range. We use probability distributions to approximate the likelihood of observing different outcomes of random variables.

**The Exponential Distribution**

A continuous random variable \( X \) is said follow an exponential distribution with parameter \( \lambda > 0 \) if its cumulative distribution function is given by

\[
F(x) = P(X \leq x) = \begin{cases} 1 - e^{-\lambda x}, & x \geq 0 \\ 0, & \text{otherwise} \end{cases},
\]

or, equivalently, if its probability density function is given by

\[
f(x) = \begin{cases} \lambda e^{-\lambda x}, & x \geq 0 \\ 0, & \text{otherwise} \end{cases}.
\]

Furthermore, it is defined that

\[
F(x) = \int_0^x f(s) \, ds,
\]

from which we can obtain

\[
P(a \leq X \leq b) = \int_a^b f(x) \, dx.
\]

The exponential distribution is often used to model random variables that represent lifetime distributions [4]. Common examples of lifetime distributions include arrival times and failure rates. The parameter \( \lambda \) is thus interpreted as the constant average occurrence rate of events per unit time [1]. The mean, or theoretical expected value, of the exponential distribution is given by

\[
E[X] = 1/\lambda,
\]
which can be interpreted as the average lifetime of an individual being observed [1]. The mean and standard deviation of the distribution are equal. The general shape of the distribution’s probability density function is right-skewed and has a negative slope, indicating that smaller values of $X$ have a higher probability of occurrence.

![Figure 1](image.png)

**Figure 1.** The general shape of an exponentially distributed random variable $X$ with parameter $\lambda$.

In addition, the distribution is memoryless, and it is known to be the only continuous probability distribution property that possesses this property [6]. Mathematically, the memoryless property is stated as follows:

$$P(X > s + t \mid X > t) = P(X > s), \quad \text{for all } s, t \geq 0.$$  

This means that the future states of a stochastic process whose inter-event times are exponential random variables $\{X_n, n = 1, 2, \ldots\}$ are independent of its previous states, conditioned on knowing the current state of the process. Therefore, knowing that $t$ time units have passed since the last event does not affect the likelihood of another event occurring in the next $s$ time units.
Poisson Processes

A stochastic process \( \{N(t), t \geq 0\} \) is known as a counting process if the function \( N(t) \) represents the total number of events that have occurred by time \( t \). This type of process is subsequently known as a Poisson process with rate \( \lambda > 0 \) if it meets the following conditions [6]:

1. \( N(0) = 0 \).
2. The counting process possesses stationary increments, meaning that the distribution of the number of events that occur in a given time interval is solely dependent on the length of the interval, and not the time at which the interval begins.
3. The counting process possesses independent increments, meaning that the number of events that occur in disjoint time intervals are independent of each other.
4. The number of events \( n \) that occur in a given interval of length \( t \) follow a Poisson distribution with mean \( \lambda t \). Thus, for all \( s, t > 0 \), we have that
   \[
   P\{N(s + t) - N(s) = n\} = e^{-\lambda t} \frac{(\lambda t)^n}{n!}, \quad n = 0, 1, ...
   \]
   and that
   \[
   E[N(t)] = \lambda t.
   \]

The assumption that the process has stationary and independent increments directly implies that the behavior of the process is the same at any given point in time, regardless of its previous states. Therefore, the process \( \{N(t), t \geq 0\} \) is memoryless, and we can conclude that its sequence of inter-event times \( \{T_n, n = 1, 2, \ldots\} \) must be exponentially
distributed with mean $1/\lambda$. This result is particularly useful in modeling arrival and waiting time distributions, which will be explored in the following sections of this study.
CHAPTER II

QUEUING THEORY

Overview

Queuing theory is a branch of mathematics that studies the formation and function of waiting lines, or queues, and their associated properties. Queues and queuing systems are found in many aspects of everyday life, from concert ticket sales to grocery checkouts to restaurant dining, which is the focus of this study. A basic queuing system is comprised of (1) customers who arrive in some random manner to receive a service, (2) at least one queue, where the customers wait to receive service, and (3) at least one server, which provides the desired service to the customer. Other important components of the system include customer interarrival times, which are defined as the elapsed times between successive arrivals to the queuing system, and service times, which are defined as the total times elapsed from the beginning to the end of each customer’s service [3]. In addition, basic queuing models typically assume a “first-come, first-served” discipline, where customers are served in the order in which they arrive [3].

There are many types of queuing models that can be used to approximate the behavior of different queuing system arrangements. These models assume various probability distributions for customer interarrival times and service times, and incorporate their properties to analyze the activity and performance of the queuing system. These distributional assumptions generate an accurate representation of the variability in the given system’s interarrival and service times. The models also specify
the number of servers available to provide service to customers. In general, queuing models are symbolically denoted by the following convention:

\[\text{Interarrival Time Distribution / Service Time Distribution / Number of Servers}\]

Each queuing model also follows a given set of assumptions about other aspects of the nature of the queuing system it represents, including the limiting capacity of the queue, its discipline, and independence of the interarrival and service times [3].

Some basic models have associated formulas that use the parameters of their interarrival time and service time distributions to measure the performance of the queuing system. The four most common performance measures are (1) the expected number of customers in the queue, \(L_q\), (2) the expected number of customers in the system, \(L\), which includes both those waiting in the queue and those receiving service, (3) the expected waiting time for customers in the queue, \(W_q\), and (4) the expected waiting time for customers in the system, \(W\), which includes both the queue waiting time and the service time.

**The M/G/s Queuing Model**

The M/G/s queuing model is one of the most well-known basic queuing models. The letter \(M\) stands for “Markovian,” and is used to indicate an exponential interarrival time distribution, since Markovian models are memoryless. The letter \(G\) stands for “General,” and is used to indicate an arbitrary service time distribution. It is not necessary to specify the form of the service time distribution, only its mean and standard deviation [3]. The letter \(s\) denotes the number of servers in the model. In addition to its
distributional guidelines, the M/G/s model adheres to the following general assumptions that are common among most basic queuing models [3]:

1. The queue capacity is infinite, and all arriving customers remain in the queue until they have received service.

2. The discipline of the queue is first-come, first-served, and customers are served individually.

3. Each server is capable of serving any customer, and a server becomes available immediately after the completion of a service.

Formulas to measure performance are available for the M/G/1 model. However, performance measures for this model are meaningless in the case where $s > 1$, due to the combined complications of including multiple servers while allowing any choice for the service time distribution [3].

Some aspects of this study’s constructed queuing simulation model are partially based on the basic structure of the M/G/s model, which will be discussed further in later sections.
CHAPTER III

DATA COLLECTION & ESTIMATION

Restaurant Background

Mike’s Clam Shack is a popular family restaurant located in Wells, Maine, owned by Mike McDermott. Wells is a coastal beach town that maintains a very high tourist volume during the summer months, and the annual influx of visitors marks Mike’s busiest season as well. The restaurant is beloved by both locals and newcomers. I have worked in takeout there every summer for the past four years and built a wonderful relationship with Mike, the managers, and the staff. They were generous enough to allow me to collect data from the restaurant and simulate its activity.

Mike’s Clam Shack boasts three separate dining areas, along with a large overall seating capacity. The dining room follows a traditional restaurant seating model, where the host assigns parties to specific tables as they arrive and a queue is accumulated as the seating area reaches its full capacity. It is split into multiple seating sections designated by color, the number and size of which are dependent on the number of servers working in the dining room. In turn, the number of seating sections open in the dining room on any particular night is determined by the relative busyness of the given month and day of the week.

In addition, there is a separate sports pub with a full bar and dance floor, which offers the same menu and service as the regular dining room. In the summer months, there is also an outdoor tent area available that offers the same services, and local musicians perform out there on weekends. The sports pub and outdoor tent areas are “seat
yourself,” where parties are allowed to find an open table to sit at without the direction of a host. Oftentimes, parties waiting in the dining room queue will leave to sit in one of the self-seating areas instead if more tables are readily available there. Finally, there is also a separate takeout area with an outdoor window and picnic tables, where parties can place orders in-person or over the phone and take them to go. Sometimes on busy nights, a party will opt for takeout instead of indoor service when faced with a long wait for the dining room.

Mike’s Clam Shack is open Wednesday – Monday each week. In the summer, the restaurant opens at 11:30 AM, and closes at 8:30 PM during the week and on Sundays. On Fridays and Saturdays, Mike’s remains open until 9:00 PM. There are two serving shifts: morning and night. The morning shift begins at 11:30 AM and ends at 4:00 PM, and the night shift lasts from 4:00 PM until the restaurant closes. In particular, this study focuses on activity in the dining room during the night shift.

Data Collection

The data used in this study was collected from May 30, 2022, to August 21, 2022, for a total of 71 days. This timeframe spans the majority of the summer season at Mike’s, when the tourism industry in Wells is in full swing and business levels are at their highest. I worked directly with the staff at Mike’s to obtain paper records of nightly server counts and queue lists, as well as seating section layouts, capacities, and party counts. These records were then transcribed into Excel spreadsheets for use in the study. As the data collection did not involve subjective research on customers or the staff at Mike’s, IRB approval was not necessary for this study.
Data Limitations

It is important to note that the data used in this study is limited. First, Mike’s does not record data about party sizes or arrival times when there is no queue. This means that on days where a queue never forms, no data is recorded. This also means that even if there is a queue, no data is recorded during the shift before it forms, nor after it dies out. The departure times of parties who have finished dining are also not recorded, and are estimated in this study based off of general table reservation timeframes.

While I observed the restaurant’s activity for a total of 71 days, only 50 of those days had a queue accumulate, and the records I obtained only contain information about the periods of the night shifts where a queue is actively accumulating. Thus, the interarrival times of parties and the proportions of party sizes entering the restaurant come from specific portions of the night shifts on those 50 days. However, this study focuses on the peak business periods for each month and week day, and it is reasonable to assume that during these periods the interarrival times of unrecorded parties are consistent with those of recorded parties, and the results of the study can be generalized accordingly.

Goodness-of-Fit Testing for the Interarrival Time Distribution

To verify whether party arrivals at Mike’s Clam Shack are described by a Poisson counting process, it is sufficient to determine whether the party interarrival times follow an exponential distribution.

Chi-square goodness-of-fit testing is utilized to compare the distribution of observed interarrival times at Mike’s to an exponential distribution. In total, the dataset
contains 3,086 interarrival time observations. Any observations with missing interarrival times are excluded when performing the chi-square test. Observations with an interarrival time of at least 10 minutes are excluded from the chi-square test, since they span a very wide range of values while only making up 2.5% of the total observations. These adjustments result in a total of 2,959 observations that are used to examine the distribution of interarrival times.

The chi-square goodness-of-fit testing is performed according to the following set of hypotheses:

\[ H_0: \text{The data arise from an exponential distribution whose mean is unknown.} \]

\[ H_a: \text{The data do not arise from an exponential distribution.} \]

Additionally, the interarrival time observations are divided into \( k = 9 \) discrete bins for the test:

<table>
<thead>
<tr>
<th>Bin</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interarrival Times ( t ) Contained (Minutes)</td>
<td>( t &lt; 2 )</td>
<td>( t \in [2,3) )</td>
<td>( t \in [3,4) )</td>
<td>( t \in [4,5) )</td>
<td>( t \in [5,6) )</td>
<td>( t \in [6,7) )</td>
<td>( t \in [7,8) )</td>
<td>( t \in [8,9) )</td>
<td>( t \in [9,10) )</td>
</tr>
</tbody>
</table>

Table 1. Division of interarrival time bins for exponential chi-square goodness-of-fit testing.

While eight of the nine bins are of equal length, interarrival times less than two minutes are combined into a larger single bin to account for potential variability in those observations. At Mike’s, the arrival time of a given party is recorded as the time the party is initially greeted by the host, but it is common for parties to have to wait a few extra seconds after entering the restaurant if the host is still attending to another party when they arrive at the host desk. This implies that some of the smaller interarrival times may
actually be slightly different than recorded, so binning them together controls for any inaccuracies at the one-minute margin.

Expected observation counts for each bin are calculated assuming an exponential distribution with mean $1/\lambda$, where $\lambda$ is the average observed interarrival time. For this dataset, we have $1/\lambda = 2.1753$ minutes.

![Observed & Expected Counts for Interarrival Times](image)

**Figure 2.** Comparison of the distribution of observed and expected counts of the interarrival observations. The observed and expected counts for each bin appear to be very similar, indicating that the data has potential to be a good fit for an exponential distribution.

A chi-square goodness-of-fit test has been performed in R using the observed and expected counts to assess the data’s quality of fit for an exponential distribution. While a chi-square test of this nature typically uses $k - 1$ degrees of freedom, an additional degree is subtracted since the test utilizes our data-driven estimate of the parameter $\lambda$. 
### Table 2. Parameters and results of the chi-square goodness-of-fit test.

<table>
<thead>
<tr>
<th>Degrees of Freedom</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chi-Square Test Statistic</td>
<td>9.2674</td>
</tr>
<tr>
<td>p-Value</td>
<td>0.2340</td>
</tr>
</tbody>
</table>

The p-value for the chi-square test is defined as the probability of observing a test statistic at least this extreme from the dataset, given that $H_0$ is true. Larger p-values indicate a better model fit. According to standard guidelines, the p-value obtained from the test signifies that the observed interarrival data has a good quality of fit for an exponential distribution. Therefore, we fail to reject $H_0$, and assume exponentiality of the interarrival times. This result is consistent with the nature of the interarrival data. We can reasonably assume that a given party’s arrival to the restaurant system is not dependent on the previous party’s arrival to the system, and thus we can infer that the interarrival times for each party are independent, which is a critical characteristic of an exponentially distributed random variable. Since we have shown that the observed interarrival times of the parties can be described by an exponential distribution, we can further conclude that the arrival of parties to Mike’s can be described by a Poisson counting process.

Furthermore, this result makes sense because of the very nature of parties choosing to whether to dine at Mike’s. There is a very large number of potential parties in the local area that could come to the restaurant during a given period of time, with a small probability that any one of these parties actually chooses to do so. This type of system is very similar to a binomial distribution with a large number of events, $n$, and a small probability of success, $p$, (where each “event” is a potential party that may come dine at Mike’s, and the probability of success is the probability that the party actually does), and
is known to be well approximated by a Poisson distribution. Since the number of potential parties that may come to the restaurant in a given time interval follows a Poisson distribution, there must be independence between these parties. Therefore, there must also be independence between the counts of parties who do choose to come to Mike’s, as well as the elapsed times between their arrivals. This further indicates that the interarrival times of parties must be exponentially distributed.

It should be noted that a version of the chi-square goodness-of-fit test that includes the interarrival times larger than 10 minutes has also been performed (77 observations fall into this upper range, so this version of the test uses a total of 3,036 observations). Comparing its results to the truncated version of the test detailed above, the p-value for the test drops to virtually zero when including these times, indicating that the data is not following an exponential distribution. Currently the cause of this phenomenon is unknown, and may be attributable to missing interarrival time data or some other unobserved factor that is influencing the interarrival times that fall in that upper range. However, the chi-square tests reveal that the vast majority of the interarrival time data collected do appear to be exponentially distributed, which is sufficient for the purposes of this study.

**Estimating the Service Time Distribution**

Service times for parties at Mike’s are estimated based on the typical table reservation timeframe allotted for a certain party size, and are displayed in Table 3. As a note, a single customer of the restaurant queuing system is defined as a single party, regardless of the number of people in the party. It is reasonable to assume that one party’s
service time is does not depend on another’s, and thus we can assume independence of the service times.

<table>
<thead>
<tr>
<th>Party Size (People)</th>
<th>Service Time (Minutes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 - 4</td>
<td>45 - 60</td>
</tr>
<tr>
<td>5 - 6</td>
<td>90</td>
</tr>
<tr>
<td>7 or more</td>
<td>120</td>
</tr>
</tbody>
</table>

Table 3. Estimated service time of a given party based on its size.

As stated in the previous chapter, while it is unnecessary to determine the form of the service time distribution, it is necessary to determine its mean (expected value) and standard deviation. We use standard methods to approximate the expected value and standard deviation of the observed distribution, which is discrete [2].

In general, for a discrete random variable $X$, the theoretical expected value of $X$ is defined as

$$E[X] = \sum_x x p(x),$$

where $p(x)$ is defined as the probability function of $X$. Since three distinct party size categories make up the distribution, the expected value is

$$E[X] = x_1 P_1 + x_2 P_2 + x_3 P_3,$$

where the $x_n$ denote the mean service time for each category and the $P_n$ denote the probability of observing a party size that falls in that category. The $P_n$ are calculated from the observed data as a simple proportion; for example, parties of size 1-4 people make up 72.44% of the dataset. Thus, the expected value of the service time distribution is
\[ E[X] = (52.5 \times 0.7244) + (90 \times 0.1773) + (120 \times 0.0983) \]

\[ E[X] = 65.7840 \text{ minutes.} \]

The standard deviation of the distribution, \( S \), is equivalent to the square root of the variance of the distribution. For a discrete random variable \( X \), the variance of \( X \) is defined as

\[ Var[X] = E[X^2] - E[X]^2. \]

Substituting in the definition of \( E[X] \), we obtain

\[ Var[X] = \sum_x x^2 p(x) - \left[ \sum_x x p(x) \right]^2 \]

\[ Var[X] = (x_1^2 P_1 + x_2^2 P_2 + x_3^2 P_3) - (x_1 P_1 + x_2 P_2 + x_3 P_3)^2. \]

Inserting the calculated values from the data, we have

\[ Var[X] = (52.5^2 \times 0.7244) + (90^2 \times 0.1773) + (120^2 \times 0.0983) - 65.7840^2 \]

\[ Var[X] = (2756.25 \times 0.7244) + (8100 \times 0.1773) + (14400 \times 0.0983) - 4327.5347 \]

\[ Var[X] = 520.7428 \text{ minutes.} \]

Finally, we take the square root of the variance to calculate the standard deviation of the distribution:

\[ S = \sqrt{Var[X]} \]

\[ S = \sqrt{520.7428} \]
\[ S = 22.8198 \text{ minutes}. \]

Although it is not necessary to determine the specific form of the service time distribution, we compare its calculated mean and standard deviation to that of an exponential distribution, since the simulation model built in the following chapter relies on the assumption of exponentiality, and we can reasonably assume that the service times of parties are independent. As stated previously, the mean and standard deviation of an exponential distribution are equal. The large discrepancy between the calculated mean of 65.7840 and the standard deviation of 22.8198 for the observed service time distribution indicates that service times in the restaurant are likely not exponentially distributed. However, the simulation model will still retain its assumption of exponentiality in order to make its structure more feasible.

It is also important to note that the observed service time distribution given here has limited variability. While the frequencies of different party sizes used to obtain the mean and standard deviation of the service time distribution are taken directly from the observed data, the service times for parties of various sizes are means that have been estimated based on the experience of managers at Mike’s Clam Shack (because finishing/service times of individual parties are not recorded), and thus there is no variability included around those means. The mean and variance of the service time distribution obtained here from the available data could be used to build and explore the behavior of an M/G/1 queuing model of the restaurant, for which analytical results are possible to obtain, and compare it to the current multi-server model, although we do not do so here.
CHAPTER IV

QUEUING SIMULATION MODEL

Simulation Code

To model the nightly dining room activity and queue at Mike’s Clam Shack, a queuing simulation model has been constructed in R. The components of the simulation program incorporate the observed data and rates to comprehensively model the dining room’s activity on a given month and day of the week. The full R program can be found in the Appendix.

As mentioned previously, the simulation model partially bases some of its structure on that of a typical M/G/s queuing model. First, through chi-square goodness-of-fit testing, it has been shown that in general, the interarrival times of parties arriving to Mike’s can be described by an exponential distribution. Furthermore, the simulation model has an infinite queue capacity and a partially first-come, first-served queue discipline (detailed further below). It also assumes that tables immediately become available after a party’s service is finished. The model considers an individual table to be a server, so the number of servers $s$ is equal to the number of tables in the dining room, which is determined by the month and day of week being simulated. The tables serve each party individually.

However, the simulation model does violate some of the foundational assumptions of the M/G/s model in the interest of creating a more accurate depiction of the behavior of the dining room and queue at Mike’s. First, the restaurant’s structure is rather unique since it is comprised of multiple self-seating areas in addition to a
traditional dining room and takeout area. As a result, parties in the queue have a high tendency to leave the queue for one of the other areas of the restaurant, and thus the simulation incorporates an early leave probability for parties in the queue. In addition, not every server is capable of serving any arriving party, since parties are assigned to tables based on their size and some tables are too small to fit larger parties. This is why the queue discipline is said to be only partially first-come, first-serve, as mentioned above. While parties are typically served in the order in which they arrive, due to the limited table capacities, it is common for a smaller party to arrive after a larger party but get seated before the larger party since there are more small tables than large tables, and small tables tend to become available more quickly than large ones. Finally, since real-time business levels tend to vary from day to day and month to month, the simulation model calculates the arrival rate of parties based on the portion of interarrival time data from a specified month and day of the week, which has the potential to affect the validity of its exponential fit, although this is very unlikely.

For simplification purposes, some of the data incorporated into the simulation model has also been limited. To start, any entries in the dataset without a recorded interarrival time are excluded from the model. In addition, any entries with a party size larger than 10 are excluded, since this is the maximum capacity of a single table at Mike’s. In practice, if a party size exceeds this capacity, the hosts and servers accommodate the party by temporarily rearranging several smaller tables to create a single table that is large enough for the entire party. However, accounting for this is slightly beyond the scope of this study, and proposed methods for handling this particular obstacle will be discussed in detail in the next section.
The simulation program is comprised of commands to load and format the Excel files that make up the dataset, followed by seven function definitions. The first six functions are preliminary functions that are called in the seventh function, which is the main simulation function. The six preliminary functions are described in further detail below:

1. \textit{numSectionFunction} determines the number of seating sections open in the dining room, given a month and day of the week. The numbers of open seating sections assigned to each month and week day have been previously determined by taking the rounded average number of seating sections open for each combination of month and week day as observed in the data.

2. \textit{numTablesFunction} determines the number of tables available in each seating section, given the number of seating sections open in the dining room.

3. \textit{arrivalRateFunction} calculates the arrival rate of parties to the restaurant from the observed data, given a month and day of the week.

4. \textit{serviceRateFunction} determines and stores the service rate for a given party.

5. \textit{tableSearchFunction} searches for available tables of appropriate size when seating a party in the dining room.

6. \textit{biggestPartyIndexFunction} searches for parties in the queue to seat when a party in the dining room completes its service.

The main function, \textit{restaurantSim}, takes a month and a day of the week as input, and uses this information to set the party arrival rate to the restaurant, the number of tables available in the dining room, and the end time for the simulation, which is synonymous with the closing time of the restaurant. Along with these components, the function begins
by initializing a number of R data structures that store rate information, keep track of the status of the dining room, and capture the system data that is generated by the simulation.

<table>
<thead>
<tr>
<th>Valid Months</th>
<th>June, July, August</th>
</tr>
</thead>
<tbody>
<tr>
<td>Valid Week Days</td>
<td>Monday, Wednesday, Thursday, Friday, Saturday, Sunday</td>
</tr>
</tbody>
</table>

Table 4. Valid string inputs for the main simulation function, which are based on the timeframe of the data collected and the restaurant’s summer schedule.

After defining the necessary data structures, the function enters a loop that repeatedly generates event times and types until the restaurant system reaches its specified closing time. All time units are in minutes. The simulation time starts at 0, which represents 4:00 PM, and the ending time of the simulation is either 270 (representing 8:30 PM) or 300 (representing 9:00 PM) depending on the specified week day. Inter-event times are randomly generated from an exponential distribution with rate $\lambda$, where $\lambda$ is based on the rate of each event type and is continuously updated to reflect the current state of the system. There are three types of events that the simulation can generate:

1. An *arrival*, which is defined as the arrival of a party to the system.
2. A *finish*, which is defined as the completion of a party’s service, i.e., the party leaves the system after being assigned to a table in the dining room.
3. An *early leave*, which is defined as a party in the queue departing from the system before being assigned to a table in the dining room.

At each event time, the simulation randomly chooses one of the three possible events to occur using weighted probabilities based on their corresponding rates and the current state of the system.
The design of the simulation model is greatly simplified by the Poisson nature of the system. More specifically, the memoryless nature of the Poisson process assumed for this model is the main facilitator of the methods used to generate event times and types. This assumption allows the event times to be generated exponentially using a combined rate based on the rates of the three event types. If the system did not follow a Poisson process, the methods used to generate event times and event types would become much more complex to implement, as the simulation would be required to associate each event time with its chosen event type and repeatedly check, update, and sort the corresponding data structures that would be needed to keep track of them.

After the event type is determined, the function updates its data structures accordingly to generate and seat parties, add or remove them from the queue, and store relevant information about party and queue counts, finish and queue leave rates, and dining room availability. The size of an arriving party is randomly selected from the dataset.

The function returns a list of seven data structures that contain information generated by the simulation. In particular, the list is made up of the following structures:

1. `queueSize.v`, which is a vector containing the length of the queue at each event time.
2. `eventTimes.v`, which is a vector containing each event time of the simulation.
3. `waitTimes.v`, which is a vector containing the total wait time for each party that waited in the queue.
4. `totalEnteredCount`, which is a variable containing the total number of parties that arrived to the restaurant.
5. *totalQueueCount*, which is a variable containing the total number of parties that waited in the queue at any point while they were in the restaurant system.

6. *totalLeaveCount*, which is a variable containing the total number of parties that left the queue early and did not receive service.

7. *totalDinedCount*, which is a variable containing the total number of parties that received service in the dining room. It is calculated by subtracting

   \[ \text{totalLeaveCount} \] from \[ \text{totalEnteredCount} \].

---

**Simulation Outcomes**

To analyze the overall behavior of the queuing system and business levels for each month and day of the week, we perform a Monte Carlo experiment by repeatedly running the simulation function and aggregating the results of all the function calls. For this experiment, the simulation has been repeated 1,000 times for each combination of month and week day. In particular, we are interested in analyzing the following outcomes of the experiment: party counts, queue sizes, and wait times.

First, we examine the four party-related counts that are compiled and returned by the simulation function and compare them to the observed data. Average values of these counts for each combination of month and week day are calculated from the data and the results of the simulation repetitions, and are displayed in Table 5. Due to the limited nature of the data, actual counts of party arrivals and parties who received service are unobserved, but actual counts of parties who waited in the queue and left the queue early are observed and their averages are able to be calculated.
| Month | Day of Week | Observed Data | | | Simulation Results | | | |
|-------|-------------|---------------|---------------|---------------|---------------|---------------|---------------|
|       |             | Average Waited in Queue | Average Left Queue Early | Average Entered System | Average Received Service | Average Waited in Queue | Average Left Queue Early |
| June  | Monday      | 6              | 3             | 99            | 99            | 0             | 0             |
|       | Wednesday   | 7              | 1             | 130           | 128           | 6             | 2             |
|       | Thursday    | 0              | 0             | 130           | 127           | 6             | 3             |
|       | Friday      | 12             | 3             | 67            | 67            | 0             | 0             |
|       | Saturday    | 48             | 9             | 142           | 140           | 6             | 3             |
|       | Sunday      | 16             | 3             | 133           | 129           | 8             | 3             |
| July  | Monday      | 75             | 28            | 115           | 114           | 1             | 0             |
|       | Wednesday   | 74             | 21            | 130           | 127           | 6             | 3             |
|       | Thursday    | 34             | 12            | 99            | 99            | 0             | 0             |
|       | Friday      | 19             | 9             | 140           | 138           | 6             | 2             |
|       | Saturday    | 60             | 26            | 128           | 127           | 1             | 1             |
|       | Sunday      | 30             | 11            | 95            | 95            | 0             | 0             |
| August| Monday      | 111            | 35            | 157           | 153           | 11            | 5             |
|       | Wednesday   | 93             | 30            | 137           | 128           | 21            | 10            |
|       | Thursday    | 47             | 18            | 129           | 127           | 5             | 2             |
|       | Friday      | 66             | 18            | 120           | 120           | 1             | 0             |
|       | Saturday    | 71             | 20            | 128           | 128           | 1             | 1             |
|       | Sunday      | 96             | 27            | 137           | 125           | 27            | 12            |

Table 5. Average counts of parties who enter the system, receive service, wait in the queue, and leave the queue early for each combination of month and week day, which are calculated from the observed data and the results of the Monte Carlo experiment. Values have been rounded to the nearest whole number.

Comparing the observed counts to those generated by the Monte Carlo experiment, we observe some rather surprising results: the simulated counts of parties who have waited in the queue are much lower than their real-life counterparts for nearly every month and week day. Simulated counts of parties who leave the queue early are also much lower than the actual counts for July and August. Upon examining the
simulated finishing times of parties, it was discovered that parties finished their service in
the dining room at a much faster rate than expected. Because of this, the queue does not
have time to accumulate, and wait times for parties who do make it into the queue are
only a few minutes at most.

The cause of this strange phenomenon was initially rather difficult to identify, as
the program’s methods of generating event times and types appeared to be
mathematically correct in their construction. However, it has been determined that the
memoryless nature of the program, while a crucial part of its structure, is actually the
reason this phenomenon is occurring. The memoryless aspect of exponentially distributed
random variables gives them the following property: for a set of independent exponential
random variables \( \{X_i, \ i = 1, 2, \ldots n\} \), each with rate \( \lambda_i \), the minimum \( X_i \) is exponentially
distributed, and its rate is equal to the sum of the \( \lambda_i \) [6]. Formally, this property is written
as

\[
P\{\min(X_1, X_2, \ldots, X_n) > x\} = P\{X_i > x, i = 1, 2, \ldots, n\}
\]

\[
= \prod_{i=1}^{n} P\{X_i > x\}
\]

Substituting in the definition of \( P\{X_i > x\} \), we obtain

\[
P\{\min(X_1, X_2, \ldots, X_n) > x\} = \prod_{i=1}^{n} e^{-\lambda_i x}
\]

\[
= \exp \left\{ -x \sum_{i=1}^{n} \lambda_i \right\}.
\]
To further illustrate this concept in the context of this study’s simulation, consider an example: suppose four parties, each with a service time of 60 minutes, arrive to the restaurant at the same time and are all seated. Because the simulation function generates event times and types exponentially, we consider the service times of the four parties to be a set of independent exponential random variables, each with rate 1/60. Then, according to the property detailed above, the first party will finish its service with rate

\[ \lambda = \sum_{i=1}^{4} \frac{1}{60} = \frac{4}{60} = \frac{1}{15}, \]

meaning that on average, the first party will finish its service after 15 minutes. This is what is occurring in the simulation, and what is causing the finish times of parties to be uncharacteristically rapid.

Despite this mathematical complication, we still observe some outcomes of the simulation that are consistent with the behavior of the actual queuing scenario at Mike’s. In general, higher average counts of parties in the queue are accompanied by higher average counts of parties who leave the queue before receiving service. Additionally, Mondays, Wednesdays, and Sundays in August return the highest average counts of parties who wait in the queue throughout the night.

The Monte Carlo experiment has also been performed with several faster party arrival rates to determine what arrival rate would produce simulated queue size averages similar to the observed averages, given that the simulation currently causes parties to finish at a very quick rate. It was found that to do so, the arrival rate needs to be accelerated to between 1.4 and 1.5 times its normal rate, although the simulated averages for some month and week day combinations are still very inaccurate.
CHAPTER V

CONCLUSIONS & RECOMMENDATIONS FOR FUTURE STUDY

Overall, this study has sought to understand the some of the most prominent mathematical concepts associated with queuing theory and apply them to construct a simulation model of a real queuing scenario at Mike’s Clam Shack in order to investigate dining room activity and queue behavior during the summer season. While no clear guidance for determining optimal dining room capacities or staffing resources for the summer has jumped out from the study at this time, making adjustments to the current queuing simulation model would provide more effective methods of exploring these “what-if” scenarios with changing parameters for different months and week days. In this section, we propose several additional ideas and procedures that could be implemented in the future to improve the accuracy of the simulation model and remedy some of its current eccentricities.

First, the most important aspect of the simulation that should be focused on for improvement is the way the model handles service times and event generation. Currently, event times and types are generated exponentially based on a continuously updated total rate. While this method greatly simplifies the overall structure of the model, the memoryless nature of the exponential distribution is directly causing finish events to occur too quickly. In the future, it may be more appropriate to use another probability distribution to generate events and thus avoid running into this problem, even though this process would complicate the R code.
Secondly, the incorporation of additional data would allow for a more authentic model of the behavior of the queuing system. For example, obtaining data on party arrivals even when there is no queue would allow us to paint a more accurate picture of the interarrival time distribution, as well as the proportions of party sizes entering the restaurant. Furthermore, specific information about the individual service times for parties would allow us to construct a more thorough service time distribution and incorporate an additional level of randomness into the model. More data on individual service times would allow us to determine a range of service times for parties that fall within each category of party sizes. We could use this additional information to estimate the mean service times more accurately, as well as the standard deviation of the service time distribution. This would allow us generate more correct and variable finish times of parties dining in the restaurant.

Another metric of the restaurant scenario that could be included in an updated model is bussing times. In practice, tables do not become immediately available after a party is finished dining – the table must be bussed (cleared, cleaned, and reset) first. If data were collected on table bussing times, this process could be included in the model as part of the service time.

Another limitation of the current simulation model is its exclusion of parties that are too large to fit at a single table. To handle this complication, the simulation function could include additional data structures or trackers that are capable of identifying multiple tables in the same seating section with a combined capacity large enough to fit the party. The function could then link the tables together with a unique ID to assign the
party to these tables, and all the linked tables would become available again when the party finished its service.

Finally, to further improve the simulation’s accuracy, we could consider modeling the arrival times as a nonhomogeneous Poisson process. This generalization of the Poisson process would allow the arrival rate at time $t$ to be a function of $t$, $\lambda(t)$, meaning that the arrival rate could change throughout the timeframe of the simulation [6]. Implementing this method would be particularly useful, as the arrival rate of parties does not remain constant throughout the night in a real restaurant scenario. Typically, the arrival rate near the beginning of the night is a bit slower. Arrivals gradually become more frequent towards the middle of the shift, and remain fairly constant for a period before gradually slowing again as the restaurant nears its closing time. Using a nonhomogeneous Poisson process to describe party arrivals would allow us to account for these changes in behavior near the beginning and end of the night.
REFERENCES


====== Queuing Simulation Model of Mike's Clam Shack =====

# Load Data & Packages

# Clear environment
rm(list = ls())

# Load packages
library(readxl)
library(tidyverse)
library(vtable)

# Set working directory
setwd(wait_datafolder)

# Make list of file names
wait_files <- list.files(path = wait_datafolder)
# Read in files & create data frame
for (i in 1:length(wait_files)) {
  daily_data <- read_excel(wait_files[i], range = cell_cols("A:M"))
  wait_data <- rbind(wait_data, daily_data)
}

# Exclude observations with missing interarrival times
wait_data <- na.omit(wait_data)
# Exclude observations with party size > 10
wait_data <- wait_data %>% filter(Party.Size <= 10)

# Create vector of party sizes
partySizes.v <- c(wait_data$Party.Size)
# Create vector of queue leave status: TRUE = left queue early, FALSE = stayed in queue
leaveStatus.v <- c(wait_data$Left) != "No"
Preliminary Functions

# Determine number of seating sections open based on month & day
numSectionFunction <- function(month, dayOfWeek) {
  # Filter data by month
  if (month == "June") {
    # Determine number of sections based on day of week
    if (dayOfWeek == "Monday") {numSections <- 5}
    else if (dayOfWeek == "Wednesday") {numSections <- 6}
    else if (dayOfWeek == "Thursday") {numSections <- 5}
    else if (dayOfWeek == "Friday") {numSections <- 7}
    else if (dayOfWeek == "Saturday") {numSections <- 7}
    else if (dayOfWeek == "Sunday") {numSections <- 7}
    else if (dayOfWeek == "Tuesday") {stop("The restaurant is closed on Tuesdays. Please choose a different day of the week.")}
    else {stop("Error! Please enter a valid weekday")}
  }
  else if (month == "July") {
    # Determine number of sections based on day of week
    if (dayOfWeek == "Monday") {numSections <- 6}
    else if (dayOfWeek == "Wednesday") {numSections <- 7}
    else if (dayOfWeek == "Thursday") {numSections <- 6}
    else if (dayOfWeek == "Friday") {numSections <- 7}
    else if (dayOfWeek == "Saturday") {numSections <- 7}
    else if (dayOfWeek == "Sunday") {numSections <- 7}
    else if (dayOfWeek == "Tuesday") {stop("The restaurant is closed on Tuesdays. Please choose a different day of the week.")}
    else {stop("Error! Please enter a valid weekday")}
  }
  else if (month == "August") {
    # Determine number of sections based on day of week
    if (dayOfWeek == "Monday") {numSections <- 8}
    else if (dayOfWeek == "Wednesday") {numSections <- 7}
    else if (dayOfWeek == "Thursday") {numSections <- 7}
    else if (dayOfWeek == "Friday") {numSections <- 7}
    else if (dayOfWeek == "Saturday") {numSections <- 7}
    else if (dayOfWeek == "Sunday") {numSections <- 7}
    else if (dayOfWeek == "Tuesday") {stop("The restaurant is closed on Tuesdays. Please choose a different day of the week.")}
    else {stop("Error! Please enter a valid weekday.")}
  }
}
else {stop("Error! Please enter a valid month.")}

# Return the number of seating sections open
return(numSections)
}

# DETERMINE THE TABLES IN EACH SECTION BASED ON MONTH & DAY
numTablesFunction <- function(numberOfSections) {
  if (numberOfSections == 5) {
    # Vectors of table capacities and ID numbers for each section
    brownCapacity.v <- c(4, 4, 4, 4, 4, 6, 6, 6, 6)
    brownIDs.v <- c(1, 2, 3, 4, 5, 12, 13, 14, 15, 16)
    redCapacity.v <- c(4, 4, 4, 6, 6, 6)
    redIDs.v <- c(6, 7, 8, 9, 10, 11)
    tanCapacity.v <- c(4, 4, 4, 4, 4, 8, 10)
    tanIDs.v <- c(17, 18, 19, 35, 36, 21, 20)
    yellowCapacity.v <- c(4, 4, 4, 4, 4, 8, 10)
    yellowIDs.v <- c(24, 25, 26, 27, 28, 23, 29)
    pinkCapacity.v <- c(4, 4, 4, 4, 4, 8, 10)
    pinkIDs.v <- c(37, 38, 39, 40, 41, 33, 31)

    # Create lists for table capacities, ID numbers, availability status
    tableCapacities.l <- list(brownCapacity.v, redCapacity.v, tanCapacity.v, yellowCapacity.v, pinkCapacity.v)
    tableIDs.l <- list(brownIDs.v, redIDs.v, tanIDs.v, yellowIDs.v, pinkIDs.v)
    tableAvailability.l <- list(c(rep(TRUE, 10)), c(rep(TRUE, 6)), c(rep(TRUE, 7)), c(rep(TRUE, 7)), c(rep(TRUE, 7)))

    # Determine the total number of tables in the dining room
    totalTables <- length(brownIDs.v) + length(redIDs.v) + length(tanIDs.v) + length(yellowIDs.v) + length(pinkIDs.v)
  }

  else if (numberOfSections == 6) {
    # Vectors of table capacities and ID numbers for each section
    brownCapacity.v <- c(4, 4, 4, 4, 6, 6, 6)
    brownIDs.v <- c(1, 2, 3, 14, 15, 16)
    greenCapacity.v <- c(4, 4, 4, 15, 16)
    greenIDs.v <- c(14, 15, 16)
    redCapacity.v <- c(4, 4, 6, 6, 6)
    redIDs.v <- c(7, 8, 9, 10, 11)
    tanCapacity.v <- c(4, 4, 4, 4, 4, 8, 10)
    tanIDs.v <- c(17, 18, 19, 35, 36, 21, 20)
    yellowCapacity.v <- c(4, 4, 4, 4, 4, 8, 10)
    yellowIDs.v <- c(24, 25, 26, 27, 28, 23, 29)
  }

  # Other cases, such as 3 or 4 sections, can be handled similarly.
}
pinkCapacity.v <- c(4, 4, 4, 4, 4, 8, 10)
pinkIDs.v <- c(37, 38, 39, 40, 41, 33, 31)

# Create lists for table capacities, ID numbers, availability status

tableCapacities.l <- list(brownCapacity.v, greenCapacity.v, redCapacity.v, tanCapacity.v, yellowCapacity.v, pinkCapacity.v)
tableIDs.l <- list(brownIDs.v, greenIDs.v, redIDs.v, tanIDs.v, yellowIDs.v, pinkIDs.v)
tableAvailability.l <- list(c(rep(TRUE, 6)), c(rep(TRUE, 5)), c(rep(TRUE, 5)), c(rep(TRUE, 7)), c(rep(TRUE, 7)))

# Determine the total number of tables in the dining room
	totalTables <- length(brownIDs.v) + length(greenIDs.v) + length(redIDs.v) + length(tanIDs.v) + length(yellowIDs.v) + length(pinkIDs.v)

}{

else if (numberOfSections == 7) {

# Vectors of table capacities and ID numbers for each section
brownCapacity.v <- c(4, 4, 4, 6, 6, 6)
brownIDs.v <- c(1, 2, 3, 14, 15, 16)
greenCapacity.v <- c(4, 4, 4, 6, 6)
greenIDs.v <- c(4, 5, 6, 12, 13)
redCapacity.v <- c(4, 4, 6, 6, 6)
redIDs.v <- c(7, 8, 9, 10, 11)
tanCapacity.v <- c(4, 4, 4, 8, 10)
tanIDs.v <- c(17, 18, 19, 21, 20)
yellowCapacity.v <- c(4, 4, 4, 4, 8)
yellowIDs.v <- c(24, 25, 26, 27, 28, 23)
pinkCapacity.v <- c(4, 4, 4, 8, 10)
pinkIDs.v <- c(39, 40, 41, 29, 31)
purpleCapacity.v <- c(4, 4, 4, 4, 10)
purpleIDs.v <- c(35, 36, 37, 38, 33)

# Create lists for table capacities, ID numbers, availability status

tableCapacities.l <- list(brownCapacity.v, greenCapacity.v, redCapacity.v, tanCapacity.v, yellowCapacity.v, pinkCapacity.v, purpleCapacity.v)
tableIDs.l <- list(brownIDs.v, greenIDs.v, redIDs.v, tanIDs.v, yellowIDs.v, pinkIDs.v, purpleIDs.v)
tableAvailability.l <- list(c(rep(TRUE, 6)), c(rep(TRUE, 5)), c(rep(TRUE, 5)), c(rep(TRUE, 7)), c(rep(TRUE, 6)), c(rep(TRUE, 6)), c(rep(TRUE, 5)))

# Determine the total number of tables in the dining room
}
totalTables <- length(brownIDs.v) + length(greenIDs.v) + length(redIDs.v) + length(tanIDs.v) + length(yellowIDs.v) + length(pinkIDs.v) + length(purpleIDs.v)
}

else if (numberOfSections == 8) {
  # Vectors of table capacities and ID numbers for each section
  brownCapacity.v <- c(4, 4, 4, 6, 6, 6)
brownIDs.v <- c(1, 2, 3, 14, 15, 16)
greenCapacity.v <- c(4, 4, 4, 6, 6)
greenIDs.v <- c(4, 5, 6, 12, 13)
redCapacity.v <- c(4, 4, 6, 6, 6)
redIDs.v <- c(7, 8, 9, 10, 11)
tanCapacity.v <- c(4, 4, 4, 4, 4, 8, 10)
tanIDs.v <- c(17, 18, 19, 21, 20)
yellowCapacity.v <- c(4, 4, 4, 4, 8)
yellowIDs.v <- c(24, 25, 26, 27, 28, 23)
pinkCapacity.v <- c(4, 4, 4, 8, 10)
pinkIDs.v <- c(39, 40, 41, 29, 31)
purpleCapacity.v <- c(4, 4, 4, 4, 10)
purpleIDs.v <- c(35, 36, 37, 38, 33)
tealCapacity.v <- c(4, 4, 4, 6, 8, 8)
tealIDs.v <- c(47, 48, 49, 53, 51, 50)

  # Create lists for table capacities, ID numbers, availability status
  tableCapacities.l <- list(brownCapacity.v, greenCapacity.v, redCapacity.v, tanCapacity.v, yellowCapacity.v, pinkCapacity.v, purpleCapacity.v, tealCapacity.v)
tableIDs.l <- list(brownIDs.v, greenIDs.v, redIDs.v, tanIDs.v, yellowIDs.v, pinkIDs.v, purpleIDs.v, tealIDs.v)
tableAvailability.l <- list(c(rep(TRUE, 6)), c(rep(TRUE, 5)), c(rep(TRUE, 5)), c(rep(TRUE, 7)), c(rep(TRUE, 6)), c(rep(TRUE, 5)), c(rep(TRUE, 5)), c(rep(TRUE, 6)))

  # Determine the total number of tables in the dining room
  totalTables <- length(brownIDs.v) + length(greenIDs.v) + length(redIDs.v) + length(tanIDs.v) + length(yellowIDs.v) + length(pinkIDs.v) + length(purpleIDs.v) + length(tealIDs.v)
}
else {stop("Error!")}

# Return the 3 lists & total table count
return(list(tableCapacities.l, tableIDs.l, tableAvailability.l, totalTables))
# DETERMINE PARTY ARRIVAL RATE BASED ON MONTH & DAY

arrivalRateFunction <- function(month, dayOfWeek) {
  # Filter data by month
  if (month == "May" | month == "June") {
    # Filter data by day of the week
    if (dayOfWeek == "Monday") {
      interarrivals <- wait_data %>%
        filter((Month == "May" | Month == "June") & Day.of.Week == "Monday") %>%
        select(Interarrival.Period)
    }
    else if (dayOfWeek == "Wednesday" | dayOfWeek == "Thursday") {
      interarrivals <- wait_data %>%
        filter((Month == "May" | Month == "June") & (Day.of.Week == "Wednesday" | Day.of.Week == "Thursday") %>%
        select(Interarrival.Period))
    }
    else if (dayOfWeek == "Friday") {
      interarrivals <- wait_data %>%
        filter((Month == "May" | Month == "June") & Day.of.Week == "Friday") %>%
        select(Interarrival.Period)
    }
    else if (dayOfWeek == "Saturday") {
      interarrivals <- wait_data %>%
        filter((Month == "May" | Month == "June") & Day.of.Week == "Saturday") %>%
        select(Interarrival.Period)
    }
    else if (dayOfWeek == "Sunday") {
      interarrivals <- wait_data %>%
        filter((Month == "May" | Month == "June") & Day.of.Week == "Sunday") %>%
        select(Interarrival.Period)
    }
    else if (dayOfWeek == "Tuesday") {
      stop("The restaurant is closed on Tuesdays. Please choose a different day of the week.")
    }
    else {
      stop("Error! Please enter a valid weekday")
    }
  }
  else if (month == "July") {
    # Filter data by day of the week
    if (dayOfWeek == "Monday") {
      interarrivals <- wait_data %>%
        filter((Month == "July") & Day.of.Week == "Monday") %>%
        select(Interarrival.Period)
    }
    else if (dayOfWeek == "Wednesday") {
      interarrivals <- wait_data %>%
        filter((Month == "July") & Day.of.Week == "Wednesday") %>%
        select(Interarrival.Period)
    }
    else if (dayOfWeek == "Thursday") {
      interarrivals <- wait_data %>%
        filter((Month == "July") & Day.of.Week == "Thursday") %>%
        select(Interarrival.Period)
    }
    else if (dayOfWeek == "Friday") {
      interarrivals <- wait_data %>%
        filter((Month == "July") & Day.of.Week == "Friday") %>%
        select(Interarrival.Period)
    }
    else if (dayOfWeek == "Saturday") {
      interarrivals <- wait_data %>%
        filter((Month == "July") & Day.of.Week == "Saturday") %>%
        select(Interarrival.Period)
    }  
  }
}

# Calculate arrival rate
arrivalRate <- 1 / mean(interarrivals$Interarrival.Period)
else if (dayOfWeek == "Sunday") {interarrivals <- wait_data %>% filter((Month == "July") & Day.of.Week == "Sunday") %>% select(Interarrival.Period)
else if (dayOfWeek == "Tuesday") {stop("The restaurant is closed on Tuesdays. Please choose a different day of the week.")}
else {stop("Error! Please enter a valid weekday")}
# Calculate arrival rate
arrivalRate <- 1 / mean(interarrivals$Interarrival.Period)

else if (month == "August") {
  # Filter data by day of the week
  if (dayOfWeek == "Monday") {interarrivals <- wait_data %>% filter((Month == "August") & Day.of.Week == "Monday") %>% select(Interarrival.Period)
  else if (dayOfWeek == "Wednesday") {interarrivals <- wait_data %>% filter((Month == "August") & Day.of.Week == "Wednesday") %>% select(Interarrival.Period)
  else if (dayOfWeek == "Thursday") {interarrivals <- wait_data %>% filter((Month == "August") & Day.of.Week == "Thursday") %>% select(Interarrival.Period)
  else if (dayOfWeek == "Friday") {interarrivals <- wait_data %>% filter((Month == "August") & Day.of.Week == "Friday") %>% select(Interarrival.Period)
  else if (dayOfWeek == "Saturday") {interarrivals <- wait_data %>% filter((Month == "August") & Day.of.Week == "Saturday") %>% select(Interarrival.Period)
  else if (dayOfWeek == "Sunday") {interarrivals <- wait_data %>% filter((Month == "August") & Day.of.Week == "Sunday") %>% select(Interarrival.Period)
  else if (dayOfWeek == "Tuesday") {stop("The restaurant is closed on Tuesdays. Please choose a different day of the week.")}
else {stop("Error! Please enter a valid weekday")}
  # Calculate arrival rate
  arrivalRate <- 1 / mean(interarrivals$Interarrival.Period)

else {stop("Error! Please enter a valid month.")}

# Return the party arrival rate
return(arrivalRate)

# GENERATE SERVICE RATES FOR PARTIES
serviceRateFunction <- function(sizeOfParty, finishRateVector) {
  # Choose service rate based on party size (1/minutes)
if (sizeOfParty <= 4) {serviceRate <- 1 / runif(1, min = 45, max = 60)}
if (sizeOfParty >= 5 & sizeOfParty < 7) {serviceRate <- 1 / 90}
if (sizeOfParty >= 7) {serviceRate <- 1 / 120}

# Append the service rate to the finish rates vector to update
return(append(finishRateVector, serviceRate))

# FIND THE SMALLEST AVAILABLE TABLE TO FIT A PARTY
tableSearchFunction <- function (sizeOfParty, tableCapacityList, tableIDList, tableAvailabilityList, partiesCurrentlyDiningList) {
  # Large placeholder value for minimum table size
  minTableSize = 100
  tableFound = FALSE
  # Find the smallest available table that can fit the party
  for (sectionIndex in 1:length(tableCapacityList)) {
    for (tableIndex in 1:length(tableCapacityList[[sectionIndex]])) {
      tempTableSize = tableCapacityList[[sectionIndex]][[tableIndex]]
      if ((tempTableSize >= sizeOfParty) && (tempTableSize < minTableSize)) {
        minTableSize = tempTableSize
        tableFound = TRUE
        # Record indexes of the section and table
        savedSectionIndex = sectionIndex
        savedTableIndex = tableIndex
      }
    }
  }
  # If a table is found, seat the party & return TRUE
  if (tableFound == TRUE) {
    # Identify the table ID number & change its status to unavailable
    tableID <- tableIDList[[savedSectionIndex]][[savedTableIndex]]
    tableAvailabilityList[[savedSectionIndex]][[savedTableIndex]] <- FALSE
    # Update list of parties currently dining
    partiesCurrentlyDiningList <- c(partiesCurrentlyDiningList, c(sizeOfParty, tableID, savedSectionIndex, savedTableIndex))
    # Return TRUE and the updated data structures
    return(list(TRUE, tableAvailabilityList, partiesCurrentlyDiningList))
  }
  # Return FALSE if no table is found & do not change data structures
else {return(list(FALSE, NA, NA))}
}

# FIND THE LARGEST PARTY IN THE QUEUE THAT CAN BE SEATED AT A
# RECENTLY FINISHED TABLE
biggestPartyIndexFunction <- function(queuePartySizesVector, tableCapacityList, tableIDList, tableAvailabilityList, partiesCurrentlyDiningList, lengthOfQueue) {
  biggestParty <- 0
  biggestIndex <- 0
  biggestCanSeat <- vector(mode = "list", 3)
  # Find the largest party in the queue that can fit at the table
  for (index in 1:lengthOfQueue) {
    nextParty <- queuePartySizesVector[index]
    canSeat <- tableSearchFunction(nextParty, tableCapacityList, tableIDList, tableAvailabilityList, partiesCurrentlyDiningList)
    if (canSeat[[1]] == TRUE & nextParty > biggestParty) {
      biggestParty <- nextParty; biggestIndex <- index; biggestCanSeat <- canSeat
    }
  }
  # Return the index of the biggest party, updated lists for
table availability & parties currently dining
  return(list(biggestIndex, biggestCanSeat[[2]], biggestCanSeat[[3]]))
}

#================================================================
# Simulation Function
#================================================================

restaurantSim <- function(month, weekday) {

  # INITIALIZE DATA STRUCTURES
  # SET RATES
  # Arrival rate
  arrivalRate <- arrivalRateFunction(month, weekday)
  # Vector of finish rates of the parties
  finishRates.v <- c()
  # Average leave rate for a single party
  indivQueueLeaveRate <- sum(leaveStatus.v) / nrow(wait_data)
  # Total leave rate for all of the parties in the queue
  totalQueueLeaveRate <- 0

  # SET SIMULATION TIME & RELATED STRUCTURES
  # Current time of simulation, start time is 0 (4:00 PM in min)
  currentTime <- 0
# End time for the simulation (8:30 PM in min)
endTime <- 270
# End time for Friday & Saturday (9:00 PM in min)
if (weekday == "Friday" | weekday == "Saturday") {endTime <- 300}
# Vector keeping track of all the event times
eventTimes.v <- c()

# SET TABLE STRUCTURES
capacity <- numTablesFunction(numSectionFunction(month, weekday))
# List of vectors: each vector contains the capacities of each table in each seating section
tableCapacities.l <- capacity[[1]]
# List of vectors keeping track of the ID numbers of each table in each seating section
tableIDs.l <- capacity[[2]]
# List of vectors: each vector contains Boolean values for the tables in each seating section to indicate their availability status
tableAvailability.l <- capacity[[3]]
# Total number of tables in the dining room
totalTables <- capacity[[4]]

# SET STRUCTURES FOR PARTIES CURRENTLY DINING
# Number of parties currently being served
n <- 0
# Vector recording value of n at each event time
n.v <- c()
# Index to update appropriate vectors
nIndex <- 2
# List of vectors: each vector contains the size & table information of a party that is currently dining
currentPartiesServed.l <- list()

# SET QUEUE & WAIT TIME STRUCTURES
# Current length of the queue
queueLength <- 0
# Vector recording value of queueLength at each event time
queueSize.v <- c()
# Vector keeping track of all the sizes of parties in currently the queue
queueParties.v <- c()
# Vector recording each event time where a party enters the queue
queueArrivalTimes.v <- c()
# Vector recording the wait time for each party in the queue
waitTimes.v <- c()
# Index to record entries in waitTimes.v
waitIndex <- 1

# COUNTERS
# Count the total number of parties who enter the restaurant
totalEnteredCount <- 0
# Count the total number of parties who have had to wait in the queue at any point
totalQueueCount <- 0
# Count the number of parties who leave the queue early
totalLeaveCount <- 0

# SIMULATION LOOP
while (currentTime < endTime) {
  # Finish rate of all the parties currently dining
  finishRate <- sum(finishRates.v)
  # Total rate = arrival rate - finish rate
  totalRate <- arrivalRate + finishRate + totalQueueLeaveRate

  # Probability the event is an Arrival
  arrivalProb <- arrivalRate / totalRate
  # Probability the event is a Finish
  finishProb <- finishRate / totalRate
  # Probability the event is an Early Leave
  queueLeaveProb <- totalQueueLeaveRate / totalRate

  # TIME OF NEXT EVENT
  currentTime <- currentTime + rexp(1, totalRate)
  # Choose event type: 1 = Arrival, 2 = Finish, 3 = Early Leave
  eventType <- sample(3, 1, prob = c(arrivalProb, finishProb, queueLeaveProb))

  # IF THE EVENT IS AN ARRIVAL
  if (eventType == 1) {
    # If the restaurant is not full, the party is seated
    if (n < totalTables) {
      # Randomly generate party size
      partySize <- sample(partySizes.v, 1)
      # Update count of parties who have entered the system
      totalEnteredCount = totalEnteredCount + 1

      # ASSIGN THE PARTY TO A TABLE
      # Look for smallest possible table available that can seat the party
canSeat <- tableSearchFunction(partySize, tableCapacities.l, tableIDs.l, tableAvailability.l, currentPartiesServed.l)

# If the party can be seated immediately, update dining room data structures
if (canSeat[[1]] == TRUE) {
  tableAvailability.l <- canSeat[[2]]
  currentPartiesServed.l <- canSeat[[3]]
  n <- n + 1
  # Choose service rate for the party
  finishRates.v <- serviceRateFunction(partySize, finishRates.v)
}

# IF NO SUITABLE TABLE IS FOUND, ADD PARTY TO QUEUE
else if (canSeat[[1]] == FALSE) {
  # Update queue data structures
  queueLength <- queueLength + 1
  queueParties.v <- append(queueParties.v, partySize)
  queueArrivalTimes.v <- append(queueArrivalTimes.v, currentTime)
  totalQueueLeaveRate <- queueLength * indivQueueLeaveRate
  # Update count of total parties that have waited in the queue
  totalQueueCount = totalQueueCount + 1
}

# IMMEDIATELY ADD PARTY TO QUEUE IF RESTAURANT IS FULL
else {
  # Update queue data structures
  queueLength <- queueLength + 1
  queueParties.v <- append(queueParties.v, partySize)
  queueArrivalTimes.v <- append(queueArrivalTimes.v, currentTime)
  totalQueueLeaveRate <- queueLength * indivQueueLeaveRate
  # Update count of total parties that have waited in the queue
  totalQueueCount = totalQueueCount + 1
}

# IF THE EVENT IS A FINISH
else if (eventType == 2) {
  # Randomly determine the party that is finishing
  whichPartyLeaving <- sample(n, 1, prob = finishRates.v)
# REMOVE THE PARTY FROM THE DINING ROOM
# Update finish rate vector & current dining room party count
finishRates.v <- finishRates.v[-whichPartyLeaving]
n <- n - 1
# Find the table information for the leaving party
tableID <- currentPartiesServed.l[whichPartyLeaving][2]
sectionIndex <- whichPartyLeaving[3]
tableIndex <- whichPartyLeaving[4]
# Update dining room data structures
currentPartiesServed.l[whichPartyLeaving]
tableAvailability.l[sectionIndex][tableIndex] <- TRUE

# IF QUEUE IS NOT EMPTY, SEAT THE NEXT PARTY IN THE QUEUE
if (queueLength > 0) {
  # Find next party to be seated & update dining room data structures
  biggestPartyInfo.l <-
  biggestPartyIndexFunction(queueParties.v, tableCapacities.l, tableIDs.l, tableAvailability.l, currentPartiesServed.l, queueLength)
  qIndex <- biggestPartyInfo.l[[1]]
  if (qIndex > 0) {
    nextParty <- queueParties.v[qIndex]
tableAvailability.l <- biggestPartyInfo.l[[2]]
currentPartiesServed.l <- biggestPartyInfo.l[[3]]
  # Generate their service rate
  finishRates.v <- serviceRateFunction(nextParty, finishRates.v)
  # Update queue data structures & determine the party's wait time
  queueParties.v <- queueParties.v[-qIndex]
  waitTimes.v[waitIndex] <- currentTime - queueArrivalTimes.v[qIndex]
  waitIndex <- waitIndex + 1
  queueArrivalTimes.v <- queueArrivalTimes.v[-qIndex]
  queueLength <- queueLength - 1
  # Negative queue length error
  if (queueLength < 0) {stop('terrible error! negative queue length 1')}
  # Update queue leave rate
  totalQueueLeaveRate <- queueLength * indivQueueLeaveRate
  # Update the number of parties currently dining
  n <- n + 1
}
if (queueLength < 0) {stop('terrible error! negative queue length 2')}

# IF THE EVENT IS A PARTY LEAVING THE QUEUE EARLY
else {
  if (queueLength > 0) {
    # Choose the party that is leaving the queue
    leavingParty <- sample(length(queueParties.v), 1)
    # Update queue data structures
    queueLength <- queueLength - 1
    queueParties.v <- queueParties.v[-leavingParty]
    queueArrivalTimes.v <- queueArrivalTimes.v[-leavingParty]
    # Update the count of parties who have left the queue early
    totalLeaveCount = totalLeaveCount + 1
    # Negative queue length error
    if (queueLength < 0) {stop('terrible error! negative queue length 3')} # Update queue leave rate
    totalQueueLeaveRate <- queueLength * indivQueueLeaveRate
  }
}

# UPDATE DATA STRUCTURES AT EACH EVENT TIME
n.v[nIndex] <- n
eventTimes.v[nIndex] <- currentTime
queueSize.v[nIndex] <- queueLength
nIndex <- nIndex + 1

# COUNT THE TOTAL NUMBER OF PARTIES WHO RECEIVE SERVICE
totalDinedCount <- totalEnteredCount - totalLeaveCount

# RETURN STATEMENT
return(list(queueSize.v, eventTimes.v, waitTimes.v, totalEnteredCount, totalDinedCount, totalQueueCount, totalLeaveCount))
AUTHOR BIOGRAPHY

Natalie Alise Robinson was born on January 27, 2001. She was raised in Wells, Maine, and graduated from Wells High School in 2019. At the University of Maine, Natalie is a Mathematics major with a minor in Economics, as well as a 4+1 Data Science and Engineering Master’s student. She is also a member of Phi Kappa Phi, Phi Beta Kappa, Pi Mu Epsilon, the Maine Top Scholar Program, and the Society of Women Engineers. Outside of school, she loves spending time with family, friends, and her cat Lucy.

Upon graduation, Natalie plans to complete her Master’s degree in Data Science and Engineering at the University of Maine.