Bike-Share Toronto Stochastic Simulation

William Hazen
Department of MIE
University of Toronto
Toronto, Ontario

1 Introduction

Bike-sharing has emerged as a popular, sustainable, and affordable mode of transportation in cities worldwide. By offering a convenient solution that eliminates concerns about bike maintenance and theft, bike-sharing systems enable customers to rent a bike from one station and return it to any other station within the city in 30-minute intervals. Exceeding this time limit incurs additional costs. In Toronto, Bike-Share provides 24/7 access to 7,185 bikes and 630 stations spread across a 200 km^2 area. With a growing population and significant traffic congestion, Toronto residents are seeking reliable and efficient commuting options for work or school, and bike-sharing presents a viable solution. However, the challenge of bike overpopulation at certain stations during rush hour persists. The Toronto Parking Authority (TPA), which owns Bike-Share Toronto, reallocated bikes overnight to mitigate this issue. Nevertheless, during peak hours, bike flow can become unbalanced, resulting in customer frustration when faced with empty docks or unavailable bike return spots due to stations reaching maximum capacity. Ensuring that bike-sharing remains a reliable and convenient transportation option for Toronto's commuters is a crucial concern for both customers and the TPA. Although the destination of an individual bike is stochastic, the process of stations becoming under or overpopulated can be simulated to yield practical insights for devising an optimal solution for proper allocation. By addressing these challenges, bike-sharing systems can continue to provide a dependable and efficient mode of transportation for urban residents.

1.1 Related Work

This project is primarily based on the works of [1] and [2], which are among several research papers addressing the bike-sharing rebalancing problem within the existing literature. [1] employs Citi Bike NYC data from December 2015 as input for a discrete-event simulation model, which is an adaptation of the model proposed by

[2]. The objective of this research is to minimize the expected number of dissatisfied customers resulting from suboptimal bike allocation. [1] introduces a gradient-like heuristic approach capable of enhancing any given allocation based on the discrete-event model. Furthermore, the paper investigates the behaviour of the bike-sharing system during various times of the day, such as morning and afternoon rush hours. By utilizing simulated optimization heuristics, the study optimizes bike and dock allocations for distinct periods of the day. On the other hand, [2] primarily focuses on rebalancing challenges that consider traffic congestion during bike reallocation. During peak hours, the paper emphasizes the optimal strategies for rebalancing the system, taking into account limited resources and real-time constraints such as traffic conditions. This project incorporates key aspects from both research papers, presenting a comprehensive approach to addressing the bike-sharing rebalancing problem.

2 Problem Description

The efficient allocation of bikes across stations in bike-sharing systems is a critical issue that affects the user experience and the sustainability of the transportation system. Poor bike allocation can lead to stations being under or overpopulated, resulting in dissatisfied customers and reduced usage of the system. The Toronto BikeShare system, like many other bike-sharing systems, faces this challenge. This project will follow the methods used in [1] to simulate bike-share systems and optimize the initial bike allocation which can be directly applied to the real-time Toronto Bike-Share data. Using open-source bike-share data from the city of Toronto [3], the aim is to replicate [1] work to minimize the expected number of unhappy customers that would arrive at a station, which is defined as a station being under or overpopulated when the customer arrives. Equation (1) models one of the objectives for this project, being to minimize $E[f(x, r)]$ where x_i and r_i are bounded between their respective values. x_i detonates the level or current number of bikes at a station i at a given time and r_i detonates the capacity of station i. The aim is to create a discrete-event simulation model that can model the behaviour of the Toronto BikeShare system and optimize the initial bike and dock allocations to minimize the number of unhappy customers. This will be achieved through stochastic optimization techniques, where the objective function is the expected number of errors. To calculate the expected number of errors, we will use the 95% confidence interval within a time period. The time period will be selected based on the availability of data and the peak season of bike usage in Toronto, which is typically during the summer months.

$$
\begin{aligned}
\text{minimize} \quad & E[f(x,r)] \\
\text{subject to} \quad & \sum_i x_i = 399 \\
& \sum_i r_i = 500 \\
& 0 \le x_i \le r_i, \\
& 12 \le r_i \le 32, \\
& x_i, r_i \in \mathbb{Z}, \forall i
\end{aligned} \tag{1}
$$

The simulation model will be based on [1] version such that it should take into account the time-varying behaviour of trip duration and potential bikers'. Moreover, the probability of arriving at a certain station given the departure from a specific station should follow a multinomial distribution due to the nature of bike traffic. Using this logic, new data can be extrapolated and fitted to these distributions such that optimization heuristic methods can be performed in 'real-time'. Therefore the goal of this project is to provide a comprehensive analysis of how the Toronto BikeShare system can sustain a surge in usage during peak hours, which would allow developers to have better insight into future development. By optimizing the initial bike allocation in the system, we can reduce the number of unhappy customers arriving at a station and promote the use of bicycles as a mode of transportation, leading to a more sustainable and efficient transportation system. Thus, this project represents a valuable contribution to the field of urban transportation systems and has the potential to improve the sustainability and efficiency of transportation systems globally.

3 BikeShare Data

3.1 Input Data

Using public BikeShare data from [3], a time period was selected from real trip data containing 705150 trips from August 1st - September 1st, 2022. This time was chosen due to the peak season being in July and August in Toronto, thus proper bike allocation has a greater impact during this time. The dataset used in this study was quite large, containing information from a large number of stations scattered throughout the city. To focus the analysis and reduce complexity, a subset of 19 stations located in close proximity to each other were selected for further examination. These stations were identified based on their geographic location and the nature of the traffic flows they serviced. Additionally, to capture data from a period of high traffic volume, in terms of our analysis it was restricted to the four-hour time window between 8 am and 12 pm. This time period was selected as it is commonly considered to be rush hour with many people commuting to work or school. By focusing on a specific subset of stations during a specific time period, this study aimed to provide targeted insights into traffic patterns and allows for more controlled experimentation and easier validation of the simulation model. Figure 1 shows the locations of the selected stations.

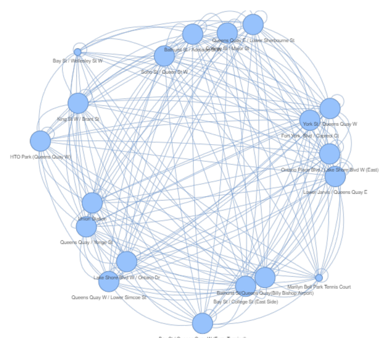


Figure 1: Graph Representation of Trips between Stations

The chosen subset of stations is located in downtown Toronto along Lakeshore, offering a diverse range of trip patterns, including short trips and longer ones across various paths. These stations are popular during the selected time period. In Equation (1), the sum of $x_i = 399$ and $r_i = 500$ are based on 19 unique stations, with the dock capacity for Toronto BikeShare ranging from 5 - 50. We selected an average of 26 spots per station. The dataset contains start and end times, station IDs, and trip durations in seconds, allowing us to derive metrics like average trip duration, busiest stations, and popular routes. Our objective includes validating demand and supply at different times using historical data and simulation-based experiments.

4 Model Description

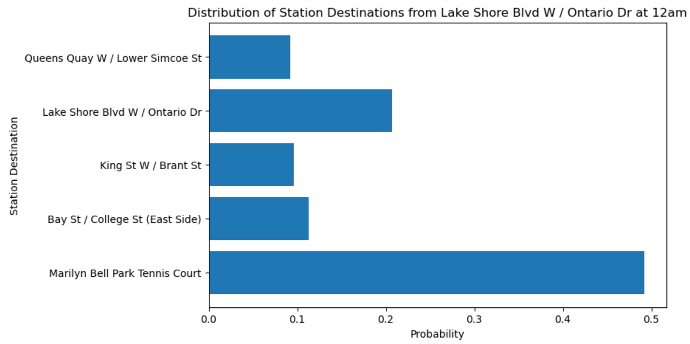
The simulation model created is a discrete-event-based model where it operates in minutes. To note, this paper is based on [1] and [2] work such that their simulations use a stationary Poisson process to model the flow of potential bikers. In a discrete-event simulation of a bike-sharing system, it may be assumed that the arrival rate of customers is constant over time and does not depend on the current state of the system. This means that the probability of a customer arriving P at a station i at a given time T is the same, regardless of the number of bikes or docks available. While this assumption may not be entirely accurate, it is a reasonable approximation for short time intervals, such as minutes or hours. Furthermore, the stationary process simplifies the modelling process and allows for easier computation of important system metrics such as the expected number of customers at each station and the expected number of bikes available. Therefore, this paper follows the same logic and uses a stationary Poisson process to model the flow of potential bikers in the Toronto BikeShare system, with the understanding that this is a simplification of the actual behaviour of the system.

4.1 Arrival Rates

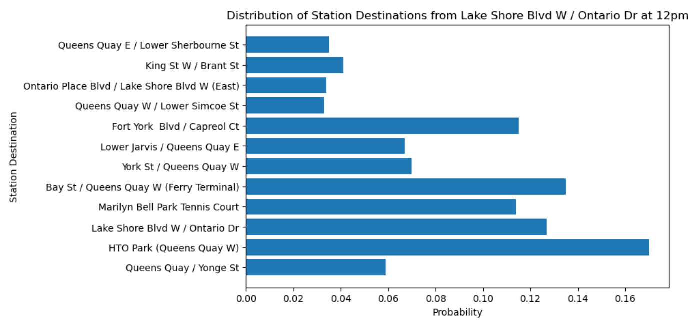
The simulation model is logically derived from [1] as it operates in discrete time ranging from $t \in [0, 47]$ - being 30-minute intervals in a 24-hour clock. Since this simulation operates in discrete time and follows a stationary poison process, the probability of a customer arriving at a station at a specific time T is derived from the BikeShare data the flow of potential bikers arriving at each station can be modelled by a time-varying Poisson process with a rate of $\mu_{t,i} = \sum_j \mu_{t,i,j}$ for time interval t, where μ is the total number of observed customers that arrive at station i in 30 minutes time intervals with the arrival times are rounded to the nearest minute. The 95% confidence interval for the expected number of customers arriving at any of the 19 stations in the subset is 22.92 ± 1.52, and for the expected number of customers per minute is 0.7641 ± 0.05071.

4.2 Probability of Destinations

The process of a customer choosing their route from station i can be determined by a multinomial distribution. When a customer arrives at station i in the model, they choose a destinations station j based on a multi-index pivot table (shown in Appendix) that selects a destination based on the time of day t and station i on the customer starting from. determines the destination of a biker leaving a station i during time interval t where the probability of going to station j is estimated by $P_{t,i,j} = \mu_{t,i,j}/\mu_{t,i}$. Figure 2 shows the different destinations that a customer is expected to travel at a certain time t.



(a) 12am



(b) 12pm

Figure 2: Difference in Destination based on t

4.3 Trip Duration

From [1], the authors investigated the duration of each trip $T_{i,j}$, which is the time it takes for a customer to travel from station i to its possible destination stations j. After considering multiple distributions, the lognormal distribution was found to be the best fit for the data. To estimate the parameters of this distribution, a regression approach was used. To perform the regression, the authors extracted estimated trip times from station i to its possible destination stations j using the Google Maps API. They then used a linear regression model on the natural logarithm of the Google trip times and the natural logarithm of the observed trip times. Figure 3 in their paper shows a scatter plot of the natural logarithm of the observed trip times against the natural logarithm of the Google trip times, with an R^2 value of 0.66.

To obtain the parameters of the lognormal distribution, the authors added an error term ϵ to the base linear regression equation, assuming that the error term follows a normal distribution with parameters obtained from the base regression. To account

Figure 3: ln(Observed Data) = 0.45 ln(Google Trips) + 3.97 + ϵ.

for any unexplained variation or randomness in the observed data that is not accounted for by the regression model. In other words, the regression model may not perfectly predict the observed data due to factors that are not included in the model, such as traffic congestion, road closures, or individual differences in travel behavior. This results in the equation $\ln(\text{Observed Data}) = 0.45 \ln(\text{Google Trips}) + 3.97 + \epsilon$. The resulting distribution was used to generate synthetic trip times for use in their simulation model and the 95% confidence interval of trip times $= 14.44 \pm 0.17413$, similarly, the trip duration is also rounded to the nearest minute. The mean and standard deviation that is required for PythonSim is given below.

$$E(X) = e^{\mu + \frac{\sigma^2}{2}} = 1.136$$
$$SD(X) = e^{\mu + \frac{\sigma^2}{2}} \sqrt{e^{\sigma^2} - 1} = 0.6135$$

To note, some of the observed data were trips that started and ended at the same station, this raises an issue in the regression as google maps deem the expected trip time between the same station as 0. Therefore, for trips that had the same start and end station, we sampled from the empirical distribution of the observed trips with the attribute

4.4 Discrete-Event Simulation

The discrete-event simulation operates as follows: at the start of time period t, initial customer arrivals are scheduled, taking into account the subset of customers who only arrive at specific stations throughout the entire period. When a customer arrives at a station, the model checks if the station is empty. If it is not empty, the customer rents a bike, and a destination and trip time are assigned. If a customer arrives at an empty station, an *Empty Error* is recorded. When a biker arrives at a fully occupied station and cannot dock their bike, a *Full Error* is recorded. In the case where a biker cannot dock, they are assigned a new destination based on their current station and the corresponding time t, before embarking on another trip. The simulation runs for a specified number of iterations, and the sample average or confidence interval of unhappy customers can be calculated as E(f(x,r)) = E(Empty Error) + E(Full Error). Now since we want to compare different starting bike allocations, common random numbers will be utilized to manage the output and facilitate validation.

5 Results

5.1 Simulation Validation

To validate our model, we have presented two figures. Figure 4 shows a comparison of the arrival rates obtained from our simulation of each station with the actual arrival rates at those stations during a given time period. In addition, Figure 5 displays a comparison of the trip durations between the simulated and real data.

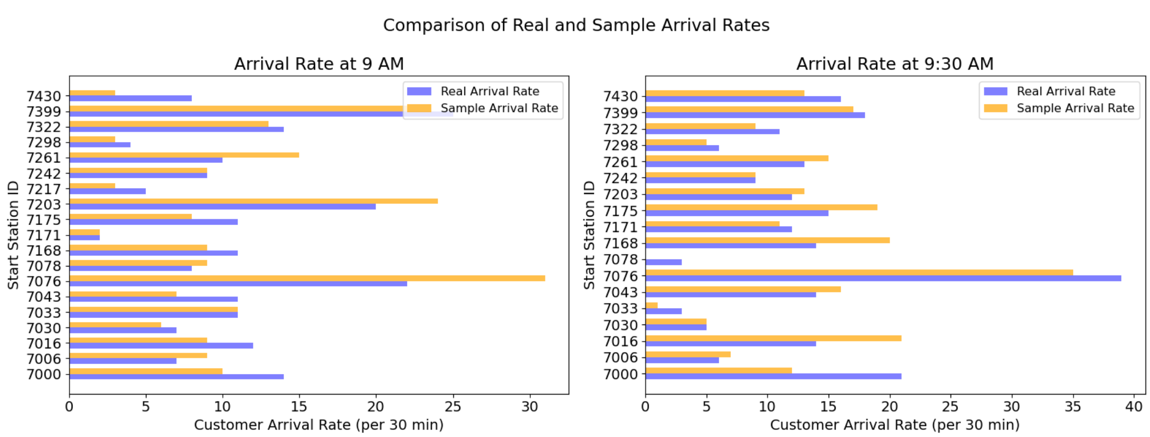


Figure 4: Comparison of Arrival Rates

One notable observation from Figure 5 is that for station 7076 (York St / Queens Quay W), there is a significant difference between the simulated and actual trip durations at 9 am. This discrepancy can be attributed to the fact that this station had a large number of leisure trips, which started and ended at the same station. Since our model samples from the empirical distribution for leisure trips, there is a possibility of a trip duration being an outlier due to the randomness and not being averaged out, which could impact the performance of our model due to the size of the subset.

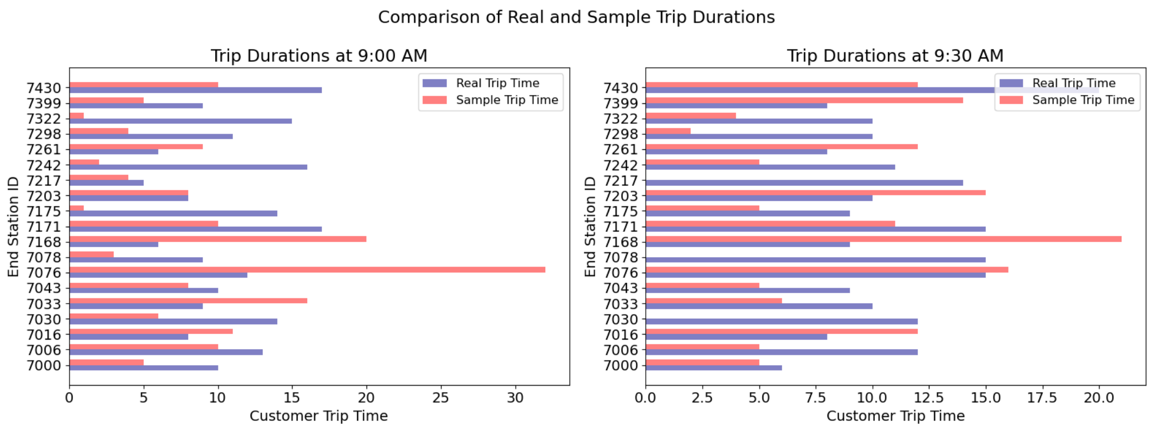


Figure 5: Comparison of Trip Times

5.2 Optimization

To minimize the expected number of unhappy customers, two different heuristics were used. Both heuristics start from an equal allocation of bikes to each station. Each method starts with an initial bike distribution and iteratively generates trial solutions by redistributing bikes between randomly selected station pairs. After evaluating the trial solutions through simulation, the algorithm updates the bike distribution if the new solution yields a lower total error. This process is repeated for a given number of iterations, and the best bike distribution found is returned as the final solution. This approach is flexible and can be adapted to different bike-sharing systems by adjusting the number of iterations and replications and parameter w being the number of bikes that can be transported from one station to another and

Heuristic 1

The first heuristic was based on [1], to which it generates trial solutions, which are new distributions of bikes across stations, based on the current bike distribution, station levels, and parameter w. The initial bike distribution to avoid modifying the original data. It then identifies two sets of stations: those that can accept w more bikes without exceeding capacity (statE) and those that can lose w bikes without going below zero bikes (statF). The function randomly selects one station from each set and redistributes w bikes between them. Afterward, it calculates the total number of bikes assigned and distributes any remaining bikes to the stations with the lowest bike levels. Finally, it creates a new dictionary containing updated instances of the Station class and returns this modified station dictionary as the trial solution. This function is integral to the iterative optimization process, as it generates trial solutions that are subsequently evaluated through simulation to find the best bike distribution, minimizing the total error in the bike-sharing system.

Heuristic 2

The second heuristic follows the same pattern as the first, but with an additional adjustment based on the flow rate of the previous trial. Specifically, we look at the flow rates of specific stations throughout our selected time period. Figure 6 provides a visual representation of the flow rates for specific stations during our selected time period. Consider Station 7016 at 11 am. The data shows that more customers started their trips from this station than ended their trips there, suggesting that it could be a source of unhappy customers. Therefore, we adjust the initial bike allocation at this station to account for the previous trial's flow rate.

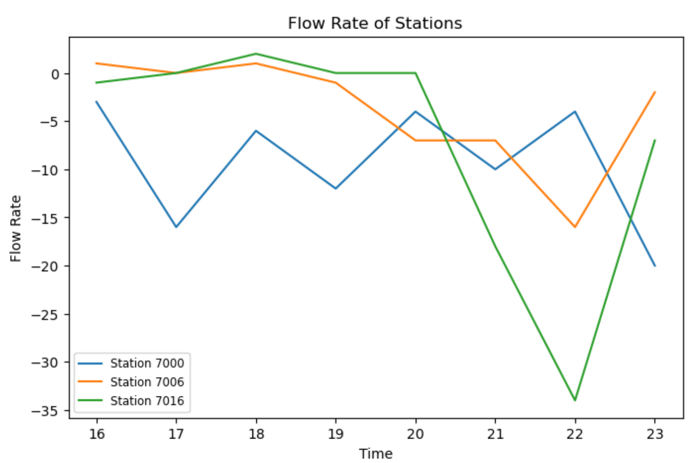


Figure 6: Flow Rates

The flow rate is calculated by the number of bikers that arrive at a station subtracted by the number of customers that leave from the station. By randomly selecting the 5 stations with the highest flow rates and the 5 stations with the lowest flow rates, we can choose a starting station and an ending station that are likely to result in unhappy customers. We then follow the rest of the algorithm as described in Heuristic 1.

5.3 Optimizing over Rush Hour

After running both heuristic models for 20 iterations, the results are illustrated in Figure 7. It is important to note that both heuristics commence with the same initial stations and share the same starting point. As the iterations progress, the objective function for both heuristics decreases, which indicates improvement in the system. However, Heuristic 2 exhibits a steeper decline, while Heuristic 1 seems to follow a linear pattern.

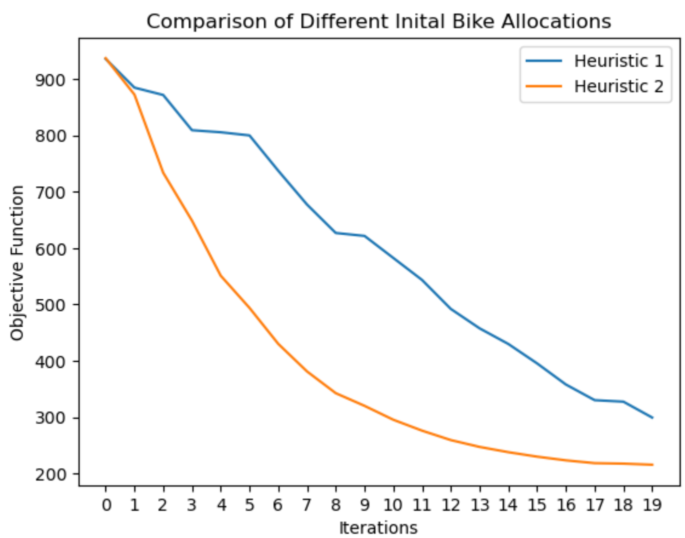


Figure 7: Optimization over 20 Iterations

The confidence interval values for the starting and ending stations are as follows:

- Heuristic 1: 935.8 ± 28.4 to 299.1 ± 22.2
- Heuristic 2: 937.7 ± 17.6 to 215.3 ± 14.4

Upon analyzing these results, it becomes evident that using the flow rate to allocate bikes is the superior heuristic. This is because Heuristic 2 takes into consideration the total flow rate for a given time period, whereas Heuristic 1 merely accounts for the number of bikes present at each station. By doing so, Heuristic 2 can more effectively pinpoint the stations that have a higher likelihood of causing dissatisfaction among customers, and allocate bikes accordingly. This approach allows Heuristic 2 to minimize the objective function more efficiently, thereby enhancing customer satisfaction. The steeper decline in the objective function for Heuristic 2 compared to the linear decline of Heuristic 1 provides further evidence of the superiority of the flow rate-based heuristic. By taking into account the dynamics of bike usage over time, Heuristic 2 can better optimize bike allocation, leading to a more effective and customer-centric solution.

6 Conclusion

This paper successfully adapted the bike-share simulation model by Jian et al. [1], achieving the primary objectives of developing a working discrete-event model and optimizing the number of unhappy customers. Through the implementation of two distinct heuristics, we were able to incrementally allocate bikes based on the empty stations and the flow rate of bike usage, ultimately leading to improved customer satisfaction. Although not replicating all results from the original study, it provides valuable insights and a foundation for future bike-share system optimization efforts. Potential future work includes adjusting dock sizes, loosening station capacity constraints, and incorporating real-time data into heuristics. Advanced machine learning algorithms can further enhance optimization. This project, utilizing stochastic simulation processes, contributes to the advancement of urban transportation systems, promoting sustainability, efficiency, and reduced environmental impact, ultimately creating more livable urban environments.

Bibliography

[1] Nanjing Jian et al. "Simulation optimization for a large-scale bike-sharing system". In: *2016 Winter Simulation Conference (WSC)*. IEEE. 2016, pp. 602–613.

[2] Eoin O'Mahony. "Smarter tools for (Citi) bike sharing". In: (2015).

[3] City of Toronto. *Open data dataset*. URL: `https://open.toronto.ca/dataset/bike-share-toronto-ridership-data/`.

Appendix

End Station Name		Bathurst St / Adelaide St W	Bathurst St/Queens Quay(Billy Bishop Airport)	Bay St / College St (East Side)	Bay St / Queens Quay W (Ferry Terminal)	Bay St / Wellesley St W	College St / Major St	Fort York Blvd / Capreol Ct	HTO Park (Queens Quay W)	King St W / Brant St
Start Station Name	**Start Time (per 30min)**									
Bathurst St / Adelaide St W	0.0	0.222222	0.000000	0.222222	0.0	0.222222	0.0	0.000000	0.000000	0.0
York St / Queens Quay W	0.0	0.083333	0.000000	0.000000	0.0	0.083333	0.0	0.083333	0.000000	0.0
Bay St / College St (East Side)	0.0	0.125000	0.000000	0.125000	0.0	0.000000	0.0	0.125000	0.000000	0.0
Union Station	0.0	0.000000	0.000000	0.000000	0.0	0.500000	0.0	0.000000	0.000000	0.0
Bay St / Queens Quay W (Ferry Terminal)	0.0	0.000000	0.111111	0.000000	0.0	0.000000	0.0	0.000000	0.111111	0.0

End Station Name		Lake Shore Blvd W / Ontario Dr	Lower Jarvis / Queens Quay E	Marilyn Bell Park Tennis Court	Ontario Place Blvd / Lake Shore Blvd W (East)	Queens Quay / Yonge St	Queens Quay E / Lower Sherbourne St	Queens Quay W / Lower Simcoe St	Soho St / Queen St W	Union Station	York St / Queens Quay W
Start Station Name	**Start Time (per 30min)**										
Bathurst St / Adelaide St W	0.0	0.0	0.000000	0.0	0.000000	0.000	0.000000	0.000000	0.111111	0.000000	0.222222
York St / Queens Quay W	0.0	0.0	0.000000	0.0	0.000000	0.250	0.000000	0.166667	0.000000	0.000000	0.333333
Bay St / College St (East Side)	0.0	0.0	0.000000	0.0	0.000000	0.125	0.000000	0.000000	0.250000	0.250000	0.000000
Union Station	0.0	0.0	0.000000	0.0	0.000000	0.000	0.500000	0.000000	0.000000	0.000000	0.000000
Bay St / Queens Quay W (Ferry Terminal)	0.0	0.0	0.111111	0.0	0.111111	0.000	0.222222	0.000000	0.000000	0.333333	0.000000

Figure 8: Probability of Destination

Table 1: Subset of 19 Stations

Start Station Name	Start Station Id	Count
York St / Queens Quay W	7076	1929
HTO Park (Queens Quay W)	7175	1810
Ontario Place Blvd / Lake Shore Blvd W (East)	7171	1628
Lake Shore Blvd W / Ontario Dr	7242	1457
Bathurst St/Queens Quay(Billy Bishop Airport)	7203	1409
Bay St / Queens Quay W (Ferry Terminal)	7016	1331
Queens Quay / Yonge St	7168	1168
Marilyn Bell Park Tennis Court	7430	1151
Lower Jarvis / Queens Quay E	7399	1101
Queens Quay W / Lower Simcoe St	7043	1084
Queens Quay E / Lower Sherbourne St	7261	1020
Fort York Blvd / Capreol Ct	7000	788
King St W / Brant St	7322	736
Union Station	7033	544
Bathurst St / Adelaide St W	7298	467
Bay St / College St (East Side)	7006	401
Bay St / Wellesley St W	7030	365
Soho St / Queen St W	7217	359
College St / Major St	7078	348

In [1]:
```python
import SimFunctions
import SimClasses
import SimRNG_Modified
import pandas as pd
import numpy as np
from sklearn.linear_model import LinearRegression
import math
import matplotlib.pyplot as plt
import seaborn as sns
import warnings
from scipy.stats import probplot, kstest, t
import pickle
from copy import deepcopy

warnings.filterwarnings("ignore")
np.random.seed(1)
ZSimRNG = SimRNG_Modified.InitializeRNSeed()
```

Data Loading & Preprocessing

In [384…
```python
#subset_df = pd.read_csv("10_station_subset.csv")
subset_df = pd.read_csv("top20_station_subset.csv")
subset_df['End Station Id'] = subset_df['End Station Id'].astype(int)
```

In [386…
```python
subset_df[["Start Station Name",
                          "Start Station Id"]].value_counts()
```

Out[386]:
```
Start Station Name                                   Start Station Id
York St / Queens Quay W                              7076                1929
HTO Park (Queens Quay W)                             7175                1810
Ontario Place Blvd / Lake Shore Blvd W (East)        7171                1628
Lake Shore Blvd W / Ontario Dr                       7242                1457
Bathurst St/Queens Quay(Billy Bishop Airport)        7203                1409
Bay St / Queens Quay W (Ferry Terminal)              7016                1331
Queens Quay / Yonge St                               7168                1168
Marilyn Bell Park Tennis Court                       7430                1151
Lower Jarvis / Queens Quay E                         7399                1101
Queens Quay W / Lower Simcoe St                      7043                1084
Queens Quay E / Lower Sherbourne St                  7261                1020
Fort York  Blvd / Capreol Ct                         7000                 788
King St W / Brant St                                 7322                 736
Union Station                                        7033                 544
Bathurst St / Adelaide St W                          7298                 467
Bay St / College St (East Side)                      7006                 401
Bay St / Wellesley St W                              7030                 365
Soho St / Queen St W                                 7217                 359
College St / Major St                                7078                 348
dtype: int64
```

In [3]:
```python
subset_df.head()
```

Out[3]:

	Unnamed: 0	Trip Id	Trip_Duration	Start Station Id	Start Time	Start Station Name	End Station Id	End Time	End Station Name	Bike Id	User Type	NumOfTrips	Start Time (per 30min)	En Tim (pe 30mir
0	0	17515475	29.566667	7203	2022-08-01 00:02:00	Bathurst St/Queens Quay(Billy Bishop Airport)	7261	2022-08-01 00:32:00	Queens Quay E / Lower Sherbourne St	6908	Casual Member	72	0.0	1.
1	18537	17961013	13.583333	7006	2022-08-18 00:14:00	Bay St / College St (East Side)	7217	2022-08-18 00:28:00	Soho St / Queen St W	5958	Casual Member	49	0.0	0.
2	18536	17961003	14.216667	7006	2022-08-18 00:13:00	Bay St / College St (East Side)	7217	2022-08-18 00:27:00	Soho St / Queen St W	2635	Annual Member	49	0.0	0.
3	11661	18045153	16.400000	7298	2022-08-21 00:24:00	Bathurst St / Adelaide St W	7076	2022-08-21 00:41:00	York St / Queens Quay W	641	Annual Member	14	0.0	1.
4	11660	18013115	17.833333	7298	2022-08-20 00:10:00	Bathurst St / Adelaide St W	7076	2022-08-20 00:28:00	York St / Queens Quay W	3264	Casual Member	14	0.0	0.

In [4]:
```python
len(subset_df["Start Station Name"].value_counts())
```

Out[4]: 19

Arrival Rates

In [5]:
```
# Convert the start time and end time to minutes
subset_df['Start Time'] = pd.to_datetime(subset_df['Start Time'])
subset_df['End Time'] = pd.to_datetime(subset_df['End Time'])
subset_df['Start Time (per 30min)'] = (subset_df['Start Time'].dt.hour * 60 + (subset_df['Start Time'].dt.minut
subset_df['End Time (per 30min)'] = (subset_df['End Time'].dt.hour * 60 + (subset_df['End Time'].dt.minute // 3

# Group the data by station and 30-minute interval, and count the number of trips that started in each group
Start_Station_HalfHour_Arrivals = subset_df.groupby([subset_df['Start Station Name'],subset_df['Start Station I
#Start_Station_HalfHour_Arrivals = subset_df.groupby([subset_df['Start Station Name'], subset_df['Start Station

# Calculate the arrival rate at each station and 30-minute interval (trips per hour)
Start_Station_HalfHour_Arrivals['ArrivalRate (per min)'] = Start_Station_HalfHour_Arrivals['ArrivalRate (per 30
arrival_df = Start_Station_HalfHour_Arrivals.sort_values(by="Start Time (per 30min)")

arrival_df
```

Out[5]:

	Start Station Name	Start Station Id	Start Time (per 30min)	ArrivalRate (per 30min)	ArrivalRate (per min)
0	Bathurst St / Adelaide St W	7298	0.0	9	0.300000
787	York St / Queens Quay W	7076	0.0	12	0.400000
92	Bay St / College St (East Side)	7006	0.0	8	0.266667
744	Union Station	7033	0.0	2	0.066667
134	Bay St / Queens Quay W (Ferry Terminal)	7016	0.0	9	0.300000
...	...	...	...	...	...
133	Bay St / College St (East Side)	7006	47.0	4	0.133333
743	Soho St / Queen St W	7217	47.0	3	0.100000
91	Bathurst St/Queens Quay(Billy Bishop Airport)	7203	47.0	27	0.900000
609	Queens Quay / Yonge St	7168	47.0	13	0.433333
832	York St / Queens Quay W	7076	47.0	35	1.166667

833 rows × 5 columns

In [472…
```
ar_30 = arrival_df["ArrivalRate (per min)"].values
CI_95(ar_30)
```

Out[472]:
```
(0.7641456582633053, '+/-', 0.050717161382507564)
```

In [469…
```
import scipy.stats as stats
desc = arrival_df.describe()[1:2].values
mean_arrival_30 = desc[0][2]
mean_arrival_min = desc[0][3]

customer_per_30min = mean_arrival_30
customer_per_min = mean_arrival_min
print(f"Customers Per 30 min = {customer_per_30min}")
print(f"Customers Per min = {customer_per_min}")


# Assuming you have a sample of arrival data and have already calculated the sample mean and standard deviation
n = len(arrival_df)
t_value = stats.t.ppf(1-0.05/2, n-1)
se_30 = mean_arrival_30 / (n ** 0.5)
se_min = mean_arrival_min / (n ** 0.5)
ci_30 = (mean_arrival_30 - t_value * se_30, mean_arrival_30 + t_value * se_30)
ci_min = (mean_arrival_min - t_value * se_min,
          mean_arrival_min + t_value * se_min)

print(
    f"Customers Per 30 min = {customer_per_30min:.2f} (95% CI: {ci_30[0]:.2f}, {ci_30[1]:.2f})")
print(
    f"Customers Per min = {customer_per_min:.2f} (95% CI: {ci_min[0]:.2f}, {ci_min[1]:.2f})")
```

```
Customers Per 30 min = 22.92436974789916
Customers Per min = 0.7641456582633053
Customers Per 30 min = 22.92 (95% CI: 21.37, 24.48)
Customers Per min = 0.76 (95% CI: 0.71, 0.82)
```

- There is approx 0.10 customer every minute in the dataset or 1 customer every 10min

Probability of Destinations

- After finding the probabilities of arriving to a destination from a specific stations, I filled any NA with 0.01 and then normalized to account for random arrivals to different destinations that was not in the data

In [7]:
```
subset_df['Start Time'] = pd.to_datetime(subset_df['Start Time'])
subset_df['End Time'] = pd.to_datetime(subset_df['End Time'])
subset_df['Start Time (per 30min)'] = (subset_df['Start Time'].dt.hour * 60 + (subset_df['Start Time'].dt.minut
subset_df['End Time (per 30min)'] = (subset_df['End Time'].dt.hour * 60 + (subset_df['End Time'].dt.minute // 3

start_end_station_halfhour_trips = subset_df.groupby([subset_df['Start Station Name'], subset_df['Start Station
total_trips = start_end_station_halfhour_trips.groupby(['Start Station Name', 'Start Time (per 30min)'])['NumOf

start_end_station_prob = pd.merge(start_end_station_halfhour_trips, total_trips, on=['Start Station Name', 'Sta
start_end_station_prob['Probability'] = start_end_station_prob['NumOfTrips'] / start_end_station_prob['TotalTri

station_vs_Dest_vs_halfhour = start_end_station_prob.pivot(index=['Start Station Name', 'Start Time (per 30min)
name_prob_df = station_vs_Dest_vs_halfhour.sort_values(by="Start Time (per 30min)")
name_prob_df.head()
```

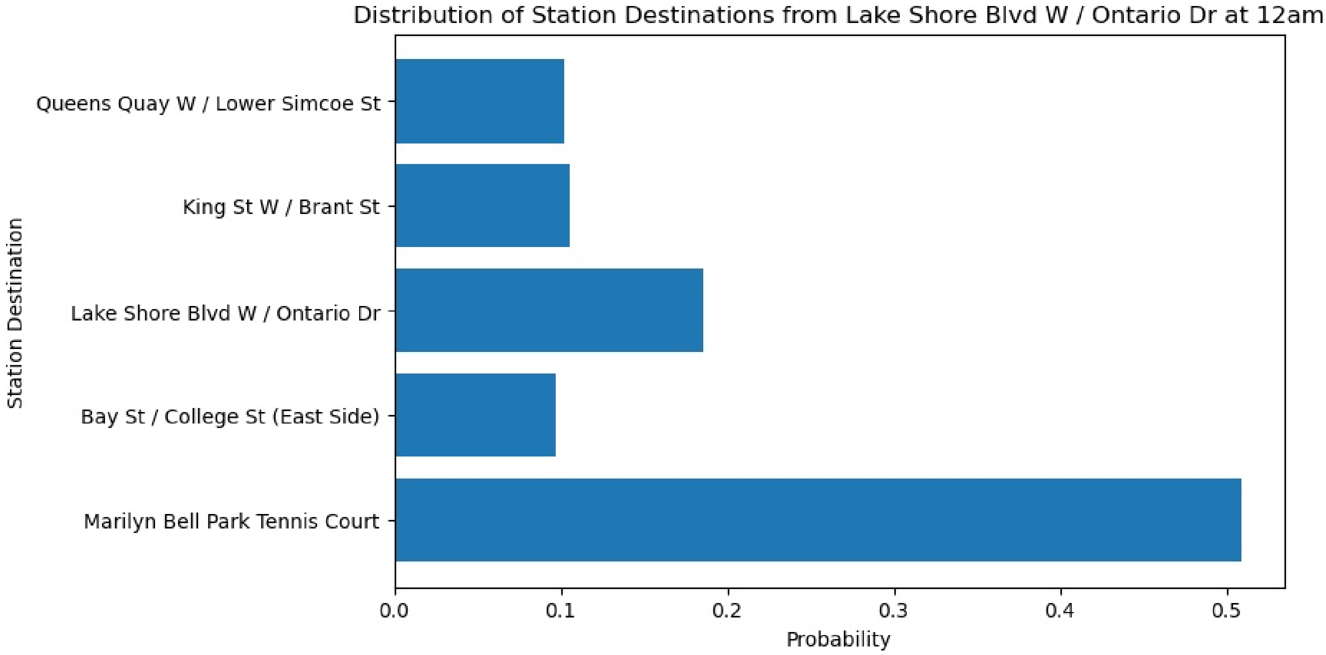
Out[7]:

Start Station Name	Start Time (per 30min) \ End Station Name	Bathurst St / Adelaide St W	Bathurst St/Queens Quay(Billy Bishop Airport)	Bay St / College St (East Side)	Bay St / Queens Quay W (Ferry Terminal)	Bay St / Wellesley St W	College St / Major St	Fort York Blvd / Capreol Ct	HTO Park (Queens Quay W)	King St W / Brant St	Lake Shore Blvd W / Ontario Dr	Lower Jarvis / Queens Quay E	Marilyn Bell Park Tennis Court	Ont Pl Bl L Sh Blv (E
Bathurst St / Adelaide St W	0.0	0.222222	0.000000	0.222222	0.0	0.222222	0.0	0.000000	0.000000	0.0	0.0	0.000000	0.0	0.000
York St / Queens Quay W	0.0	0.083333	0.000000	0.000000	0.0	0.083333	0.0	0.083333	0.000000	0.0	0.0	0.000000	0.0	0.000
Bay St / College St (East Side)	0.0	0.125000	0.000000	0.125000	0.0	0.000000	0.0	0.125000	0.000000	0.0	0.0	0.000000	0.0	0.000
Union Station	0.0	0.000000	0.000000	0.000000	0.0	0.500000	0.0	0.000000	0.000000	0.0	0.0	0.000000	0.0	0.000
Bay St / Queens Quay W (Ferry Terminal)	0.0	0.000000	0.111111	0.000000	0.0	0.000000	0.0	0.000000	0.111111	0.0	0.0	0.111111	0.0	0.111

In [8]:
```
def ChoosingRoute(prob_df, start_station, start_time):
    try:
        start_row = prob_df.loc[(start_station, start_time)]
        probs = start_row.values
        destination = np.random.choice(start_row.index, p=probs)
        return destination
    except KeyError:
        print(
            f"No data found for start station '{start_station}' and start time '{start_time}'")


monte_carlo = []
for i in range(1000):
    start_station = 'Lake Shore Blvd W / Ontario Dr'
    start_time = 0  # 30-minute interval index
    destination = ChoosingRoute(name_prob_df, start_station, start_time)
    if destination is not None:
        monte_carlo.append(destination)


prob = {}
for s in monte_carlo:
    if s in prob:
        prob[s] += 1
    else:
        prob[s] = 1
for s in prob:
    prob[s] /= len(monte_carlo)

print("Real Probability")
print(name_prob_df.loc[(start_station, start_time)])
print()
print("Simulated Probability")
print(prob)
# Plot the bar chart
plt.figure(figsize=(8, 5))
plt.barh(list(prob.keys()), list(prob.values()))
plt.xlabel('Probability')
plt.ylabel('Station Destination')
plt.title(
    f'Distribution of Station Destinations from {start_station} at 12am')
```

```
plt.show()
```

```
Real Probability
End Station Name
Bathurst St / Adelaide St W                           0.0
Bathurst St/Queens Quay(Billy Bishop Airport)         0.0
Bay St / College St (East Side)                       0.1
Bay St / Queens Quay W (Ferry Terminal)               0.0
Bay St / Wellesley St W                               0.0
College St / Major St                                 0.0
Fort York  Blvd / Capreol Ct                          0.0
HTO Park (Queens Quay W)                              0.0
King St W / Brant St                                  0.1
Lake Shore Blvd W / Ontario Dr                        0.2
Lower Jarvis / Queens Quay E                          0.0
Marilyn Bell Park Tennis Court                        0.5
Ontario Place Blvd / Lake Shore Blvd W (East)         0.0
Queens Quay / Yonge St                                0.0
Queens Quay E / Lower Sherbourne St                   0.0
Queens Quay W / Lower Simcoe St                       0.1
Soho St / Queen St W                                  0.0
Union Station                                         0.0
York St / Queens Quay W                               0.0
Name: (Lake Shore Blvd W / Ontario Dr, 0.0), dtype: float64

Simulated Probability
{'Marilyn Bell Park Tennis Court': 0.509, 'Bay St / College St (East Side)': 0.097, 'Lake Shore Blvd W / Ontari
o Dr': 0.186, 'King St W / Brant St': 0.106, 'Queens Quay W / Lower Simcoe St': 0.102}
```

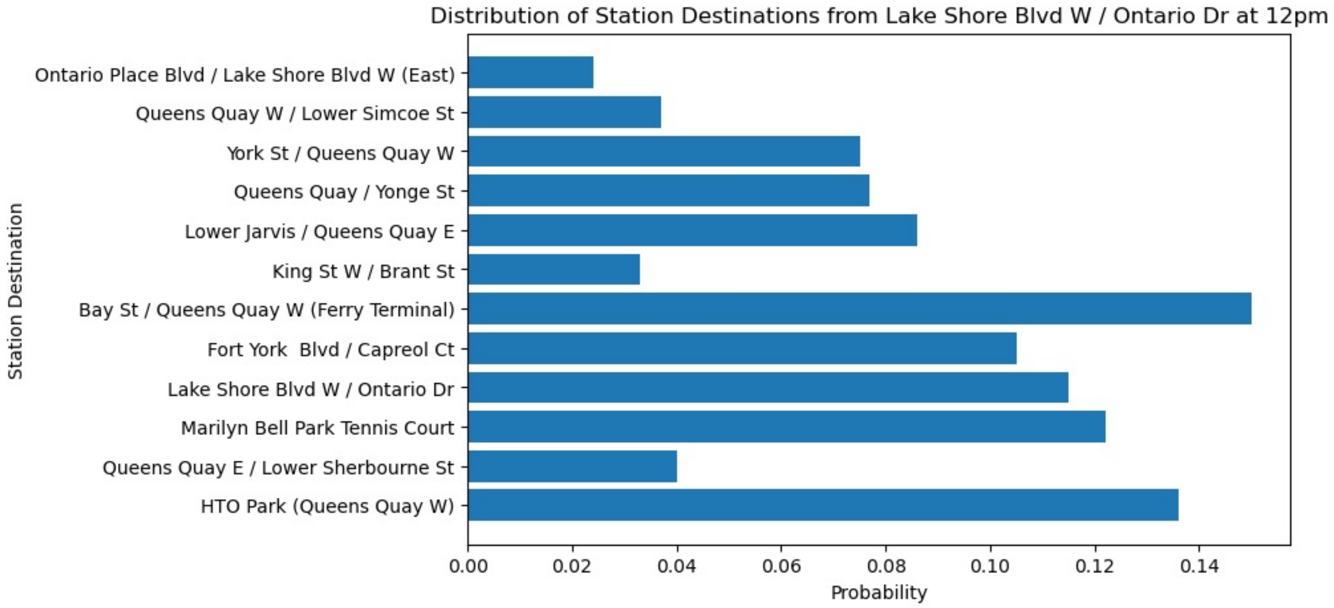


In [9]:

```
def ChoosingRoute(prob_df, start_station, start_time):
    try:
        start_row = prob_df.loc[(start_station, start_time)]
        probs = start_row.values
        destination = np.random.choice(start_row.index, p=probs)
        return destination
    except KeyError:
        print(
            f"No data found for start station '{start_station}' and start time '{start_time}'")


monte_carlo = []
for i in range(1000):
    start_station = 'Lake Shore Blvd W / Ontario Dr'
    start_time = 24  # 30-minute interval index
    destination = ChoosingRoute(name_prob_df, start_station, start_time)
    if destination is not None:
        monte_carlo.append(destination)


prob = {}
for s in monte_carlo:
    if s in prob:
        prob[s] += 1
    else:
        prob[s] = 1
for s in prob:
    prob[s] /= len(monte_carlo)

print("Real Probability")
print(name_prob_df.loc[(start_station, start_time)])
print()
```

```
print("Simulated Probability")
print(prob)
# Plot the bar chart
plt.figure(figsize=(8, 5))
plt.barh(list(prob.keys()), list(prob.values()))
plt.xlabel('Probability')
plt.ylabel('Station Destination')
plt.title(
    f'Distribution of Station Destinations from {start_station} at {start_time//2}pm')
plt.show()
```

```
Real Probability
End Station Name
Bathurst St / Adelaide St W                         0.000000
Bathurst St/Queens Quay(Billy Bishop Airport)       0.000000
Bay St / College St (East Side)                     0.000000
Bay St / Queens Quay W (Ferry Terminal)             0.161290
Bay St / Wellesley St W                             0.000000
College St / Major St                               0.000000
Fort York  Blvd / Capreol Ct                        0.096774
HTO Park (Queens Quay W)                            0.161290
King St W / Brant St                                0.032258
Lake Shore Blvd W / Ontario Dr                      0.129032
Lower Jarvis / Queens Quay E                        0.064516
Marilyn Bell Park Tennis Court                      0.129032
Ontario Place Blvd / Lake Shore Blvd W (East)       0.032258
Queens Quay / Yonge St                              0.064516
Queens Quay E / Lower Sherbourne St                 0.032258
Queens Quay W / Lower Simcoe St                     0.032258
Soho St / Queen St W                                0.000000
Union Station                                       0.000000
York St / Queens Quay W                             0.064516
Name: (Lake Shore Blvd W / Ontario Dr, 24.0), dtype: float64

Simulated Probability
{'HTO Park (Queens Quay W)': 0.136, 'Queens Quay E / Lower Sherbourne St': 0.04, 'Marilyn Bell Park Tennis Cour
t': 0.122, 'Lake Shore Blvd W / Ontario Dr': 0.115, 'Fort York  Blvd / Capreol Ct': 0.105, 'Bay St / Queens Qua
y W (Ferry Terminal)': 0.15, 'King St W / Brant St': 0.033, 'Lower Jarvis / Queens Quay E': 0.086, 'Queens Quay
/ Yonge St': 0.077, 'York St / Queens Quay W': 0.075, 'Queens Quay W / Lower Simcoe St': 0.037, 'Ontario Place
Blvd / Lake Shore Blvd W (East)': 0.024}
```



In [10]:
```
station_vs_Dest_vs_halfhour = start_end_station_prob.pivot(index=['Start Station Id', 'Start Time (per 30min)']
prob_df = station_vs_Dest_vs_halfhour.sort_values(by="Start Time (per 30min)")
prob_df
```

Out[10]:

Start Station Id	Start Time (per 30min) \ End Station Id	7000	7006	7016	7030	7033	7043	7076	7078	7168	7171	7175	7203	7217	7242	7261	
7000	**0.0**	0.000000	0.000000	0.0	0.0	1.00	0.0	0.000000	0.0	0.000000	0.0	0.0	0.000000	0.000000	0.0	0.000000	0.00
7006	**0.0**	0.125000	0.125000	0.0	0.0	0.25	0.0	0.000000	0.0	0.125000	0.0	0.0	0.000000	0.250000	0.0	0.000000	0.12
7399	**0.0**	0.000000	0.000000	0.0	0.0	0.00	0.2	0.200000	0.0	0.000000	0.0	0.0	0.000000	0.000000	0.0	0.200000	0.00
7033	**0.0**	0.000000	0.000000	0.0	0.5	0.00	0.0	0.000000	0.0	0.000000	0.0	0.0	0.000000	0.000000	0.0	0.500000	0.00
7261	**0.0**	0.000000	0.000000	0.0	0.0	0.00	0.0	0.166667	0.0	0.333333	0.0	0.0	0.000000	0.000000	0.0	0.333333	0.00
...	**...**	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
7006	**47.0**	0.500000	0.000000	0.0	0.0	0.00	0.0	0.000000	0.0	0.000000	0.0	0.0	0.000000	0.500000	0.0	0.000000	0.00
7322	**47.0**	0.285714	0.000000	0.0	0.0	0.00	0.0	0.000000	0.0	0.000000	0.0	0.0	0.000000	0.000000	0.0	0.000000	0.14
7000	**47.0**	0.333333	0.000000	0.0	0.0	0.00	0.0	0.000000	0.0	0.000000	0.0	0.0	0.000000	0.333333	0.0	0.333333	0.00
7043	**47.0**	0.000000	0.066667	0.0	0.0	0.00	0.4	0.000000	0.0	0.000000	0.0	0.0	0.266667	0.000000	0.0	0.200000	0.00
7430	**47.0**	0.000000	0.000000	0.0	0.0	0.20	0.0	0.000000	0.0	0.000000	0.0	0.0	0.200000	0.000000	0.3	0.000000	0.00

833 rows × 19 columns

Trip Durations

Different Stations Destinations

In [11]:

```
with open('top20_diff_google_bike_trip_est.pickle', 'rb') as f:
    google_bike_trip = pickle.load(f)

diff_stations_subset_df = pd.read_csv("diff_stations_subset_df.csv")
google = pd.DataFrame((np.array(google_bike_trip)), columns=["Google"])
observed = pd.DataFrame(np.array((diff_stations_subset_df["Trip_Duration"].values)), columns=["Observed"])
trip_reg_df = pd.DataFrame({"Observed": np.array((diff_stations_subset_df["Trip_Duration"].values)*60), "Google
trip_reg_df.head()
```

Out[11]:

	Observed	Google
0	1774.0	582
1	1761.0	582
2	1098.0	582
3	892.0	582
4	1310.0	582

In [12]:

```
trip_reg_df.describe()
```

Out[12]:

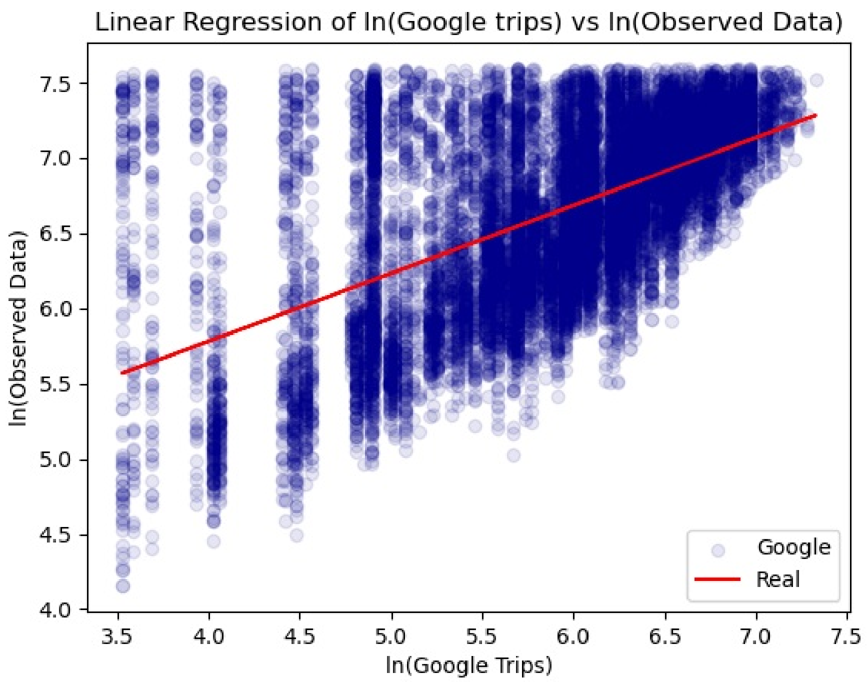
	Observed	Google
count	16202.000000	16202.000000
mean	892.046599	455.499198
std	459.080990	274.908215
min	64.000000	34.000000
25%	504.000000	249.000000
50%	841.500000	421.000000
75%	1246.000000	644.000000
max	1980.000000	1521.000000

In [13]:

```
np.random.seed(1)

X = np.log(trip_reg_df["Google"].values)
y = np.log(trip_reg_df["Observed"].values)

X = np.array(X).reshape(-1, 1)

# set regression through the origin
model = LinearRegression(fit_intercept=True)
model.fit(X, y)
predictions = model.predict(X)
score = model.score(X, y)
beta = model.coef_[0]
```

```
intercept = model.intercept_

residuals = y - predictions
residual_var = np.var(residuals)
residual_mean = np.mean(residuals)

print("Error Mean", residual_mean)
print("Residual Variance:", residual_var)
print()
print('Beta:', beta)
print('Intercept:', intercept)
print("R^2:", score)

plt.scatter(X, y, color='darkblue', alpha=0.1)
plt.plot(X, predictions, color='red')
plt.title("Linear Regression of ln(Google trips) vs ln(Observed Data)")
plt.ylabel("ln(Observed Data)")
plt.xlabel("ln(Google Trips)")
plt.legend(["Google", "Real"], loc="lower right")
plt.show()
```

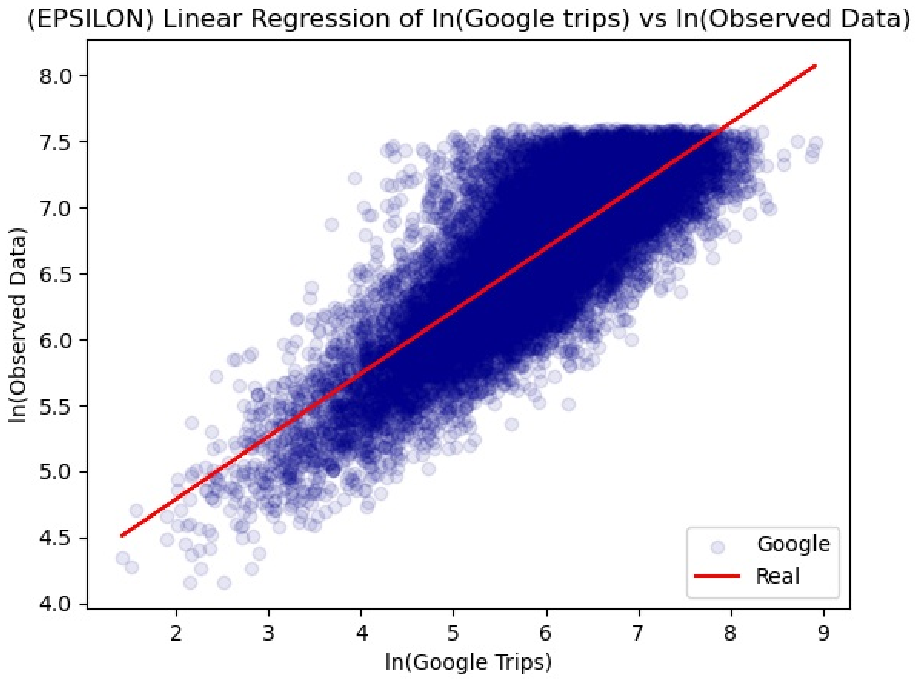
```
Error Mean -1.5261626468615904e-16
Residual Variance: 0.25580459588191384

Beta: 0.4513542745881248
Intercept: 3.974270551809947
R^2: 0.3143043103068821
```



In [14]:
```
np.random.seed(1)

X = np.log(trip_reg_df["Google"].values)
y = np.log(trip_reg_df["Observed"].values)

X_i = np.array(X).reshape(-1, 1)

# set regression through the origin
model = LinearRegression(fit_intercept=True)
model.fit(X_i, y)
predictions = model.predict(X_i)
score = model.score(X_i, y)
beta = model.coef_[0]
intercept = model.intercept_

residuals = y - predictions
residual_mean = np.mean(residuals)
residual_var = np.var(residuals)
residual_std = np.std(residuals)

error_sd = np.sqrt(residual_var)
errors = np.random.normal(loc=residual_mean, scale=residual_std, size=len(residuals))
epsilon = residuals - errors

new_X = X + epsilon
new_X_i = np.array(new_X).reshape(-1, 1)

# Fit linear regression model
model = LinearRegression(fit_intercept=True)
model.fit(new_X_i, y)
new_predictions = model.predict(new_X_i)
```

```python
# Calculate R-squared and print results
r2 = model.score(new_X_i, y)
new_beta = model.coef_[0]
new_intercept = model.intercept_

new_residuals = y - new_predictions
new_residual_mean = np.mean(new_residuals)
new_residual_var = np.var(new_residuals)
new_residual_std = np.std(new_residuals)
print("Pre Residual Mean", residual_mean)
print("Pre Residual Variance:", residual_var)
print()

print("New Resiudal Mean", new_residual_mean)
print("New Residual Variance:", new_residual_var)
print()
print('Beta:', new_beta)
print('Intercept:', new_intercept)
print("R^2:", r2)

plt.scatter(new_X_i, y, color='darkblue', alpha=0.1)
plt.plot(new_X_i, new_predictions, color='red')
plt.title("(EPSILON) Linear Regression of ln(Google trips) vs ln(Observed Data)")
plt.ylabel("ln(Observed Data)")
plt.xlabel("ln(Google Trips)")
plt.legend(["Google", "Real"], loc="lower right")
plt.show()
```

```
Pre Residual Mean -1.5261626468615904e-16
Pre Residual Variance: 0.25580459588191384

New Resiudal Mean -5.530146832449613e-16
New Residual Variance: 0.12858666740184382

Beta: 0.47504569391691565
Intercept: 3.8372405086866
R^2: 0.6553176721259963
```



Lognormal random variable and denote by μ and σ the mean and standard deviation of log(X) as estimated in R. The mean and standard deviation of X, that is required in PythonSim, are given by,

$$E(X) = e^{\mu+\frac{\sigma^2}{2}} = 0.2475798,$$

$$SD(X) = e^{\mu+\frac{\sigma^2}{2}}\sqrt{e^{\sigma^2}-1} = 0.2233498.$$

In [15]:

```python
global E_x, SD_X

u = residual_mean
std = np.sqrt(residual_var)
E_x = np.exp(u + ((std**2)/2))
SD_x = np.exp(u + ((std**2)/2)) * np.sqrt(np.exp(std**2) - 1)
print(E_x, SD_x)
```

```
1.1364419645529011 0.6135737822915353
```

Same Start and End Destination

In [16]:
```
same_stations_subset_df = pd.read_csv("same_stations_subset_df.csv")
same_stations_subset_df
```

Out[16]:

	Unnamed: 0	index	Trip Id	Trip_Duration	Start Station Id	Start Time	Start Station Name	End Station Id	End Time	End Station Name	Bike Id	User Type	NumOfTrips
0	0	88	17515597	1.250000	7261	08/01/2022 00:12	Queens Quay E / Lower Sherbourne St	7261.0	08/01/2022 00:13	Queens Quay E / Lower Sherbourne St	1815	Casual Member	156
1	1	90	17520082	20.633333	7261	08/01/2022 10:37	Queens Quay E / Lower Sherbourne St	7261.0	08/01/2022 10:58	Queens Quay E / Lower Sherbourne St	5739	Annual Member	156
2	2	91	17520873	25.416667	7261	08/01/2022 11:18	Queens Quay E / Lower Sherbourne St	7261.0	08/01/2022 11:43	Queens Quay E / Lower Sherbourne St	5090	Casual Member	156
3	3	92	17527065	1.533333	7261	08/01/2022 14:46	Queens Quay E / Lower Sherbourne St	7261.0	08/01/2022 14:48	Queens Quay E / Lower Sherbourne St	2183	Casual Member	156
4	4	93	17528090	5.766667	7261	08/01/2022 15:16	Queens Quay E / Lower Sherbourne St	7261.0	08/01/2022 15:22	Queens Quay E / Lower Sherbourne St	6707	Annual Member	156
...	...	...	...	...	...	...	...	...	...	...	...	...	...
2889	2889	20478	18195862	26.833333	7006	08/27/2022 13:32	Bay St / College St (East Side)	7006.0	08/27/2022 13:58	Bay St / College St (East Side)	4243	Casual Member	46
2890	2890	20479	18214675	32.016667	7006	08/27/2022 21:53	Bay St / College St (East Side)	7006.0	08/27/2022 22:25	Bay St / College St (East Side)	295	Casual Member	46
2891	2891	20481	18277866	25.800000	7006	08/30/2022 16:13	Bay St / College St (East Side)	7006.0	08/30/2022 16:39	Bay St / College St (East Side)	4007	Casual Member	46
2892	2892	20482	18289543	1.016667	7006	08/30/2022 23:16	Bay St / College St (East Side)	7006.0	08/30/2022 23:17	Bay St / College St (East Side)	5573	Casual Member	46
2893	2893	20483	18301399	1.433333	7006	08/31/2022 14:53	Bay St / College St (East Side)	7006.0	08/31/2022 14:54	Bay St / College St (East Side)	769	Casual Member	46

2894 rows × 13 columns

Average Trip Durations between Specific Stations

In [479…
```
CI_95(avg_trip_duration["Avg_Trip_Duration"].values)
```

Out[479]:
```
(14.446534483226277, '+/-', 0.17413841505173533)
```

In [17]:
```
# Convert the start time and end time to minutes
subset_df['Start Time'] = pd.to_datetime(subset_df['Start Time'])
subset_df['End Time'] = pd.to_datetime(subset_df['End Time'])
subset_df['Start Time (per 30min)'] = (subset_df['Start Time'].dt.hour * 60 + (subset_df['Start Time'].dt.minut
subset_df['End Time (per 30min)'] = (subset_df['End Time'].dt.hour * 60 + (subset_df['End Time'].dt.minute // 3

# Group the data by start and end station and 30-minute interval, and calculate the average trip duration in se
Station_HalfHour_AvgDuration = subset_df.groupby([subset_df['Start Station Name'], subset_df['Start Station Id'

Station_HalfHour_AvgDuration['Avg_Trip_Duration'] = Station_HalfHour_AvgDuration['Avg_Trip_Duration']

avg_trip_duration = Station_HalfHour_AvgDuration.sort_values(by="Start Time (per 30min)")
avg_trip_duration
```

Out[17]:

	Start Station Name	Start Station Id	End Station Name	End Station Id	Start Time (per 30min)	Avg_Trip_Duration
0	Bathurst St / Adelaide St W	7298	Bathurst St / Adelaide St W	7298	0.0	5.691667
4287	Queens Quay / Yonge St	7168	Queens Quay / Yonge St	7168	0.0	25.391667
4211	Queens Quay / Yonge St	7168	Lower Jarvis / Queens Quay E	7399	0.0	2.950000
758	Bay St / College St (East Side)	7006	Fort York Blvd / Capreol Ct	7000	0.0	17.850000
4068	Queens Quay / Yonge St	7168	Bathurst St/Queens Quay(Billy Bishop Airport)	7203	0.0	10.508333
...	...	...	...	...	...	...
1284	Bay St / Wellesley St W	7030	Bathurst St/Queens Quay(Billy Bishop Airport)	7203	47.0	23.950000
5379	Union Station	7033	Bathurst St/Queens Quay(Billy Bishop Airport)	7203	47.0	20.500000
2572	King St W / Brant St	7322	Lower Jarvis / Queens Quay E	7399	47.0	26.200000
2743	Lake Shore Blvd W / Ontario Dr	7242	Bay St / College St (East Side)	7006	47.0	22.466667
6123	York St / Queens Quay W	7076	York St / Queens Quay W	7076	47.0	14.178571

6124 rows × 6 columns

In [18]:

```
avg_trip_duration["Avg_Trip_Duration"].describe()
```

Out[18]:

```
count    6124.000000
mean       14.446534
std         6.952742
min         1.016667
25%         8.966667
50%        14.012500
75%        19.360417
max        33.000000
Name: Avg_Trip_Duration, dtype: float64
```

Classes & Functions

Classes

- Decide on the time units
  - Min

In [233…

```
class Station:
    def __init__(self, station_id, level, capacity):
        self.id = station_id
        self.level = level
        self.capacity = capacity
        self.bikes = {}
        self.bike_list = []
        for i in range(level):
            bike_id = f"{station_id}-{i+1}"  # create unique bike ID
            self.bikes[bike_id] = True  # mark bike as available
            self.bike_list.append(bike_id)

    def rent_bike(self):
        # Request a bike from the station
        if self.level > 0:
            if self.bike_list:
                random_index = np.random.randint(0, len(self.bike_list))
                bike_id = self.bike_list.pop(random_index)
                self.level -= 1
                return bike_id
        return None

    def return_bike(self, bike_id):
        # # Return a bike to the station
        if self.level < self.capacity:
            self.bike_list.append(bike_id)
            self.level += 1

    def Get_Bike_List(self):
        return self.bike_list


class Customer:
    def __init__(self, customer_id, start_s_id, bike=None):
        self.customer_id = customer_id
```

```
        self.start_s_id = start_s_id
        self.end_s_id = 0
        self.station_level = 0
        self.bike = bike
        self.T = 0
        self.time = 0
        self.Min = None
        self.Trip_Time = 0

    def rent_bike(self):
        station = StationDict[self.start_s_id]
        print(f"   Customer Arrives at S{station.id} with Level: {station.level}")

        if station.level > 0:
            self.bike = station.rent_bike()
            self.station_level = station.level
            print(f"    [Customer Rent Bikes] Customer ID: {self.customer_id} | Bike ID {self.bike} || Start Ti
            self.Departure()
        else:
            print(f"    -   (EMPTY) -- Customer {self.customer_id} CANNOT RENT BIKE -- EMPTY STATION {self.star

    def return_bike(self, end_station, bike):
        destination_station = StationDict[end_station]
        destination_station.return_bike(bike)
        self.station_level = destination_station.level

    def Departure(self):
        self.end_s_id = int(self.Destination())
        end_station = StationDict[self.end_s_id]

        trip_time = self.TripDuration()
        self.Trip_Time = trip_time

        print(f"        [Customer Rents Bike and Departs]: Customer ID: {self.customer_id} | Bike ID: {self.bik
        print(f"            - Start Time:{self.time}:{self.Min}")
        print(f"            - From: S{self.start_s_id} -> Level {self.station_level} | To: S{end_station.id} ->
        print(f"            - Expected Trip Time: {self.Trip_Time} min")

        SimFunctions.Schedule(Calendar, "Bike_Arrival", trip_time)
        return self.end_s_id


######
#HELPER FUNCTIONS
######
    def Destination(self):
        end_s_id = None
        while end_s_id is None:
            end_s_id = self.ChoosingRoute(prob_df, start_s_id=self.start_s_id, start_time=self.T)
        return end_s_id


    def ChoosingRoute(self, prob_df, start_s_id, start_time):
        while True:
            try:
                start_row = prob_df.loc[(start_s_id, max(0, start_time))]
                probs = start_row.values
                end_s_id = np.random.choice(start_row.index, p=probs)
                return end_s_id

            except KeyError:
                print(f"Choose Route Error NO DATA found for start station: '{start_s_id}' and start T: '{start_



    def TripDuration(self):

        if self.start_s_id == self.end_s_id: #sample from emphericaI df if same start and end station
            time_df = subset_df[subset_df["Start Time (per 30min)"] == self.T]
            same_station_subset = time_df.loc[time_df['Start Station Id'] == time_df['End Station Id']]['Trip_D
            duration_data = np.random.choice(same_station_subset)

        else:
            T = self.T
            condition = True
            while condition:
                try:
                    duration_data = avg_trip_duration[(avg_trip_duration["Start Time (per 30min)"] == T) &
                                        (avg_trip_duration["Start Station Id"] == self.start_s_id) &
                                        (avg_trip_duration["End Station Id"] == self.end_s_id)]["Avg_Trip_D
                    condition = False
                except KeyError:
                    print(f"NO DATA found for start station: '{start_s_id}' and start T: '{start_time}'")
                    T -= 1
                    if T < 0:
                        return 2
            except IndexError:
                return 2
```

```
        trip_time = duration_data * SimRNG_Modified.Lognormal(ZSimRNG, E_x, SD_x**2, 4)
        trip_time = min(35, np.round(trip_time))
        trip_time = max(2, trip_time)
        return trip_time
```

Functions

In [27]:
```
def Start():
    SimFunctions.Schedule(Calendar, "Customer_Arrival", SimRNG_Modified.Expon(ZSimRNG, 0, 1))

def NextCustomerID():
    if not hasattr(NextCustomerID, "counter"):
        NextCustomerID.counter = 0
    NextCustomerID.counter += 1
    return NextCustomerID.counter


def CI_95(data):
    a = np.array(data)
    n = len(a)
    m = np.mean(a)
    sd = np.std(a, ddof=1)
    hw = 1.96*sd / np.sqrt(n)
    return m, "+/-", hw
```

Trip Process Functions

In [28]:
```
def inital_Customer_Arrival_Rate(T):
    temp_df = arrival_df[arrival_df["Start Time (per 30min)"] == T]
    arrival_rates = temp_df["ArrivalRate (per min)"].values
    possible_station_ids = temp_df["Start Station Id"].values
    arrival_rates = arrival_df[(arrival_df["Start Time (per 30min)"] == T)]["ArrivalRate (per min)"].values
    return arrival_rates, possible_station_ids


def inital_Customer_Arrival(empty_error, CustomerList, T, minute):
    arrival_rates, multi_station_id = inital_Customer_Arrival_Rate(T)
    for i, station_id in enumerate(multi_station_id):
        arrival_rate = arrival_rates[i]
        station = StationDict[station_id]
        customer_id = NextCustomerID()
        customer = Customer(customer_id, station_id)
        customer.start_s_id = station_id
        customer.station_level = station.level
        customer.T = T
        customer.time = T//2
        customer.Min = minute

        mu = 1/arrival_rate
        inter_arrival_time = np.round(SimRNG_Modified.Expon(ZSimRNG, mu, 1))
        SimFunctions.Schedule(Calendar, "Customer_Arrival",
                              max(2, inter_arrival_time))
        # Store values in global lists
        start_time_list.append(T//2)
        inter_arrival_time_list.append(inter_arrival_time)
        arrival_time_list.append(arrival_rate)
        start_station_id_list.append(station_id)

        ################################################################
        # STATION EMPTY
        ################################################################
        if customer.station_level == 0:
            #print(f"    (EMPTY) -- Customer {customer.customer_id} CANNOT RENT BIKE | S{customer.start_s_id} -
            empty_error += 1
        else:
            CustomerList.append(customer)
            customer.rent_bike()

    return empty_error
```

In [29]:
```
def Customer_Arrival_Rate(T):
    temp_df = arrival_df[arrival_df["Start Time (per 30min)"] == T]
    arrival_rates = temp_df["ArrivalRate (per min)"].values
    possible_station_ids = temp_df["Start Station Id"].values
    arrival_rates = arrival_df[(arrival_df["Start Time (per 30min)"] == T)]["ArrivalRate (per min)"].values
    selected_station_id = np.random.choice(possible_station_ids, p=(arrival_rates / arrival_rates.sum()))
    arrival_rate = arrival_df[(arrival_df["Start Time (per 30min)"] == T) & (arrival_df["Start Station Id"] ==
    return arrival_rate, selected_station_id


def Customer_Arrival(empty_error, CustomerList, T, minute):
    global inter_arrival_time_list, arrival_time_list, start_station_id_list, start_time_list
    arrival_rate, station_id = Customer_Arrival_Rate(T)
    station = StationDict[station_id]
```

```
    customer_id = NextCustomerID()
    customer = Customer(customer_id, station_id)
    customer.start_s_id = station_id
    customer.station_level = station.level
    customer.T = T
    customer.time = T//2
    customer.Min = minute

    mu = 1/arrival_rate
    inter_arrival_time = np.round(SimRNG_Modified.Expon(ZSimRNG, mu, 1))
    SimFunctions.Schedule(Calendar, "Customer_Arrival",
                            max(2, inter_arrival_time))
    # Store values in global lists
    start_time_list.append(T)
    inter_arrival_time_list.append(inter_arrival_time)
    arrival_time_list.append(arrival_rate)
    start_station_id_list.append(station_id)

#################################################################
# STATION EMPTY
#################################################################
    if customer.station_level == 0:
        print(f"    (EMPTY) -- Customer {customer.customer_id} CANNOT RENT BIKE | S{customer.start_s_id} -> lev
        empty_error += 1
    else:
        CustomerList.append(customer)
        customer.rent_bike()

    return empty_error
```

In [30]:

```
def Bike_Arrival(Full_Error, CustomerList, T, minute):
    global end_station_id_list, end_time_list, trip_time_list

    for customer in CustomerList:
        end_s_id = customer.end_s_id
        end_station = StationDict[end_s_id]
        if end_s_id != 0:
            if end_station.level < end_station.capacity:
                end_time_minutes = (T//2) * 60 + minute
                start_time_minutes = customer.time * 60 + customer.Min
                total_trip_time = end_time_minutes - start_time_minutes
                if customer.end_s_id == end_station.id and customer.bike is not None and total_trip_time >= cus
                    customer.return_bike(customer.end_s_id, customer.bike)
                    print(f"                [BIKE RETURNED] Customer ID: {customer.customer_id} | Bike ID: {cus
                    print(f"                    - Start Time:{customer.time}:{customer.Min} - End Time:{T//2}:{
                    print(f"                    - Expected Trip Time: {customer.Trip_Time} min")
                    print(f"                    - Total Trip Time: {total_trip_time} min")
                    print(f"                    - From: S{customer.start_s_id} | To: S{end_station.id} -> Level
                    end_station_id_list.append(end_station.id)
                    end_time_list.append(T)
                    trip_time_list.append(total_trip_time)
                    CustomerList.remove(customer)

    #################################################################
    # STATION FULL
    #################################################################
            else:
                for customer in CustomerList:
                    customer_end_station = StationDict[customer.start_s_id]
                    end_time_minutes = (T//2) * 60 + minute
                    start_time_minutes = customer.time * 60 + customer.Min
                    total_trip_time = end_time_minutes - start_time_minutes
                    if customer.end_s_id == customer_end_station.id and customer_end_station.level >= customer_
                        temp_customer = customer
                        end_id = end_station.id
                        print(f"     (FULL) -- Start Time:{customer.time}:{customer.Min} - End Time:{T//2}:{min
                        Full_Error += 1
                        Retrial(temp_customer=temp_customer, end_id=end_id, T=T, minute=minute)
                        return Full_Error
    return Full_Error

def Retrial(temp_customer, end_id, T, minute):
    customer = temp_customer
    customer.start_s_id = end_id
    customer.T = T
    customer.time = T//2
    customer.Min = minute
    customer.end_s_id = customer.Destination()
    print(f"    [TRAVELS TO NEW STATION] Start Time:{customer.time}:{customer.Min} || Customer ID: {customer.cu
    trip_time = customer.TripDuration()
    SimFunctions.Schedule(Calendar, "Bike_Arrival", trip_time)
```

Simulation

In [1]:

```
NextCustomerID.counter = 0
```

```python
ZSimRNG = SimRNG_Modified.InitializeRNSeed()
np.random.seed(1)

Calendar = SimClasses.EventCalendar()
TheCTStats = []
TheDTStats = []
TheQueues = []
TheResources = []
Stations = []
CustomerList = []
CI_Full_Error_list = []
CI_Empty_Error_list = []

CI_inter_arrival_time_list = []
CI_arrival_time_list = []
CI_start_station_id_list = []
CI_end_station_id_list = []
CI_start_time_list = []
CI_end_time_list = []
CI_trip_time_list = []


CI_total_error_list = []

for days in range(0, 3, 1):
    Full_Error = 0
    Empty_Error = 0
    inital_count = 0

    inter_arrival_time_list = []
    arrival_time_list = []
    start_station_id_list = []
    end_station_id_list = []
    start_time_list = []
    end_time_list = []
    trip_time_list = []
##########################################################################################################
    # Initialize the stations and create a dictionary mapping station IDs to Station instances
    unique_stations = np.unique(subset_df["Start Station Id"].values)
    num_stations = len(unique_stations)
    total_capacity = 500
    total_bikes = 399
    Stations = {}
    level_sum = 0
    level_sum = 0
    capacity_sum = 0
    capacity_per_station = total_capacity // num_stations

    for i, station in enumerate(unique_stations):
        x_i = total_bikes //19
        r_i = capacity_per_station
        if i == num_stations - 1:
            # Allocate the remaining capacity to the last station
            r_i = total_capacity - capacity_per_station * (num_stations - 1)
        Stations[station] = Station(station_id=station, level=x_i, capacity=r_i)
        level_sum += x_i
        for station_id, station in Stations.items():
            capacity_sum += station.capacity


    StationDict = {station.id: station for station in Stations.values()}

    # Print the initial bike list for each station
    val_level_sum = []
    val_capacity_sum = []
    count = 0
    for station_id, station in Stations.items():
        count += 1
        print(f"Station {station_id} Initial Bike List: {station.Get_Bike_List()}")
        val_level_sum.append(station.level)
        val_capacity_sum.append(station.capacity)
    print("Number of Stations", count)
    print("Level Sum", sum(val_level_sum))
    print("Capacity Sum", sum(val_capacity_sum))

##########################################################################################################

    SimFunctions.SimFunctionsInit(
        Calendar, TheQueues, TheCTStats, TheDTStats, TheResources)
    SimFunctions.Schedule(Calendar, "Start", 0)

    NextEvent = Calendar.Remove()
    SimClasses.Clock = NextEvent.EventTime
    if NextEvent.EventType == "Start":
        Start()

##########################################################################################################
# SIMULATION RUN
    for T in range(15, 24):  # T = hours intervals
```

```
        inital_count += 1
        hour = T // 2
        minute = 00 if T % 2 == 0 else 30
        unit = 'PM' if hour >= 12 else 'AM'
        print()
        print("Interval:", T)
        mini = 0
        SimFunctions.Schedule(Calendar, "inital_Customer_Arrival", 0)

        if inital_count == 2:
            Full_Error = 0
            Empty_Error = 0

        while True:
            #print("Clock: {:02d}:{:02d} {:s}".format(hour, minute, unit))
            NextEvent = Calendar.Remove()
            SimClasses.Clock = NextEvent.EventTime
            minute = int((SimClasses.Clock) % 60)
            if SimClasses.Clock >= (T+1) * 30:
                break
            if NextEvent.EventType == "inital_Customer_Arrival":
                Empty_Error = inital_Customer_Arrival(
                    Empty_Error, CustomerList, T, minute)
            elif NextEvent.EventType == "Customer_Arrival":
                Empty_Error = Customer_Arrival(
                    Empty_Error, CustomerList, T, minute)
            elif NextEvent.EventType == "Bike_Arrival":
                Full_Error = Bike_Arrival(Full_Error, CustomerList, T, minute)

################################################################################################################
# OPTIMIZE
    objective_fun = Full_Error + Empty_Error
    total_error_list.append(objective_fun)

# OPTIMZE

    CI_Full_Error_list.append(Full_Error)
    CI_Empty_Error_list.append(Empty_Error)
    CI_total_error_list.append(total_error_list)

    CI_inter_arrival_time_list.append(inter_arrival_time_list)
    CI_arrival_time_list.append(arrival_time_list)
    CI_start_station_id_list.append(start_station_id_list)
    CI_end_station_id_list.append(end_station_id_list)
    CI_start_time_list.append(start_time_list)
    CI_end_time_list.append(end_time_list)
    CI_trip_time_list.append(trip_time_list)

    print(f"End of Day {days}")
    print("-----------------------------------------------------------------------------------------------------------
    print()



Errors_DF = pd.DataFrame({"Full Error": CI_Full_Error_list,
                          "Empty Error": CI_Empty_Error_list,
                          "Total Error": CI_total_error_list})

BikeSim_DF = pd.DataFrame({"Start Station ID": CI_start_station_id_list,
                           "End Station ID": CI_end_station_id_list,
                           "Arrival Rate": CI_arrival_time_list,
                           "Interarrival Rate": CI_inter_arrival_time_list,
                           "Start Time": CI_start_time_list,
                           "End Time": CI_end_time_list,
                           "Trip Time": CI_trip_time_list})
print(f"Num of Full Errors: {CI_Full_Error_list}")
print(f"Num of Empty Errors: {CI_Empty_Error_list}")
print(f"Total Errors: {CI_95(total_error_list)}")
```

Flow Rate Calculation

Total Errors

In [ ]:

```
print(f"Num of Full Errors: {CI_Full_Error_list}")
print(f"Num of Empty Errors: {CI_Empty_Error_list}")
print(f"Total Errors: {CI_95(total_error_list)}")
```

```
Num of Full Errors: [46, 48]
Num of Empty Errors: [405, 421]
Total Errors: (460.0, '+/-', 17.639999999999997)
```

In [33]:

```
print(f"Num of Full Errors: {CI_Full_Error_list}")
print(f"Num of Empty Errors: {CI_Empty_Error_list}")
```

```
Num of Full Errors: [64]
Num of Empty Errors: [387]
```

In [34]:
```
Errors_DF
```

Out[34]:

	Full Error	Empty Error	Total Error
0	64	387	[451]

Flow Rates

In [35]:
```
i = 0
station_id = BikeSim_DF["Start Station ID"].values[i]
end_station_id = BikeSim_DF["End Station ID"].values[i]
start_times = BikeSim_DF["Start Time"].values[i]
end_times = BikeSim_DF["End Time"].values[i]
trip_times = BikeSim_DF["Trip Time"].values[i]
Arrival_Rates = BikeSim_DF["Arrival Rate"].values[i]
Interarrival_Rates = BikeSim_DF["Interarrival Rate"].values[i]

print("List Lengths")
print(f"Start ID: {len(station_id)} | End ID: {len(end_station_id)}")
print(f"Start Time {len(start_times)} | End Time {len(end_times)}")
print(f"Trip Time {len(trip_times)}")
print(f"Arrival Rate {len(Arrival_Rates)} | Interarrival Rate {len(Interarrival_Rates)}")

start_flow_df = pd.DataFrame({"Start ID": station_id, "Start Time": start_times})
end_flow_df = pd.DataFrame({"End ID": end_station_id,"End Time": end_times, "Trip Time": trip_times})
arrival_flow_df = pd.DataFrame({"Start ID": station_id, "Arrival Rates": Arrival_Rates})
interarrival_flow_df = pd.DataFrame({"Start ID": station_id, "Arrival Rates": Interarrival_Rates})
```

```
List Lengths
Start ID: 2958 | End ID: 2127
Start Time 2958 | End Time 2127
Trip Time 2127
Arrival Rate 2958 | Interarrival Rate 2958
```

In [36]:
```
start_flow_count = start_flow_df.pivot_table(index='Start ID', columns='Start Time', aggfunc='size', fill_value
start_flow_count = start_flow_count.loc[:, 16:23]
start_flow_count.head()
```

Out[36]:

Start Time	16	17	18	19	20	21	22	23
Start ID								
7000	7	20	9	18	13	14	12	24
7006	4	3	9	9	8	7	19	7
7016	2	6	7	10	17	37	54	41
7030	7	8	4	5	7	15	4	22
7033	7	5	6	5	11	31	14	15

In [37]:
```
end_flow_count = end_flow_df.pivot_table(index='End ID', columns='End Time', aggfunc='size', fill_value=0)
end_flow_count = end_flow_count.loc[:, 16:23]
end_flow_count.head()
```

Out[37]:

End Time	16	17	18	19	20	21	22	23
End ID								
7000	4	4	3	6	9	4	8	4
7006	5	3	10	8	1	0	3	5
7016	1	6	9	10	17	19	20	34
7030	0	1	6	3	0	1	0	3
7033	10	8	13	5	10	24	8	16

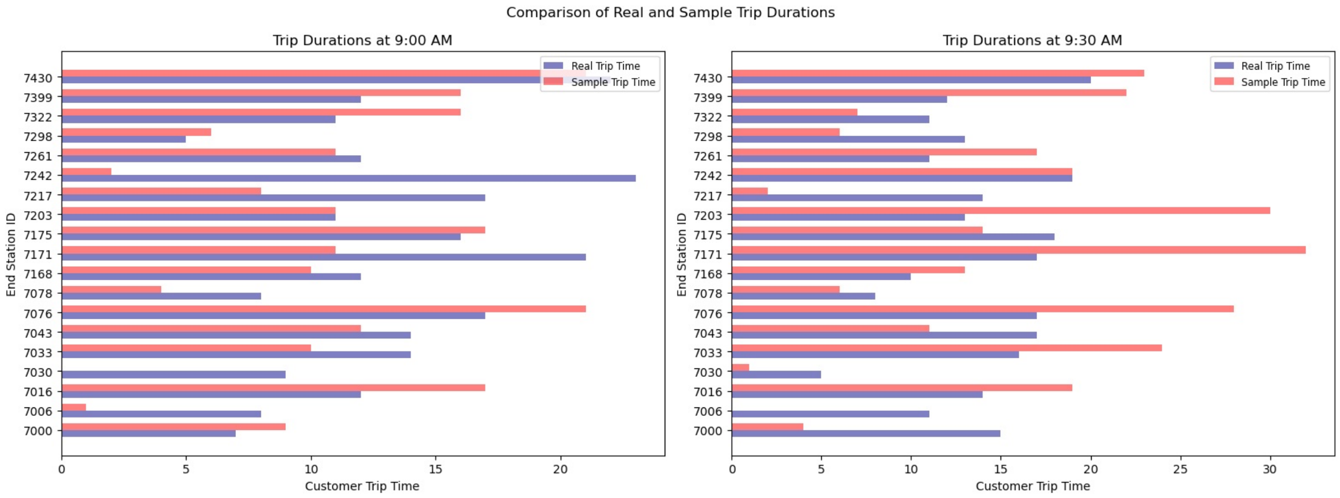
In [38]:
```
flow_rate = start_flow_count - end_flow_count
flow_rate.head()
```

Out[38]:

Start Time	16	17	18	19	20	21	22	23
Start ID								
7000	3	16	6	12	4	10	4	20
7006	-1	0	-1	1	7	7	16	2
7016	1	0	-2	0	0	18	34	7
7030	7	7	-2	2	7	14	4	19
7033	-3	-3	-7	0	1	7	6	-1

Trip Time Verfication

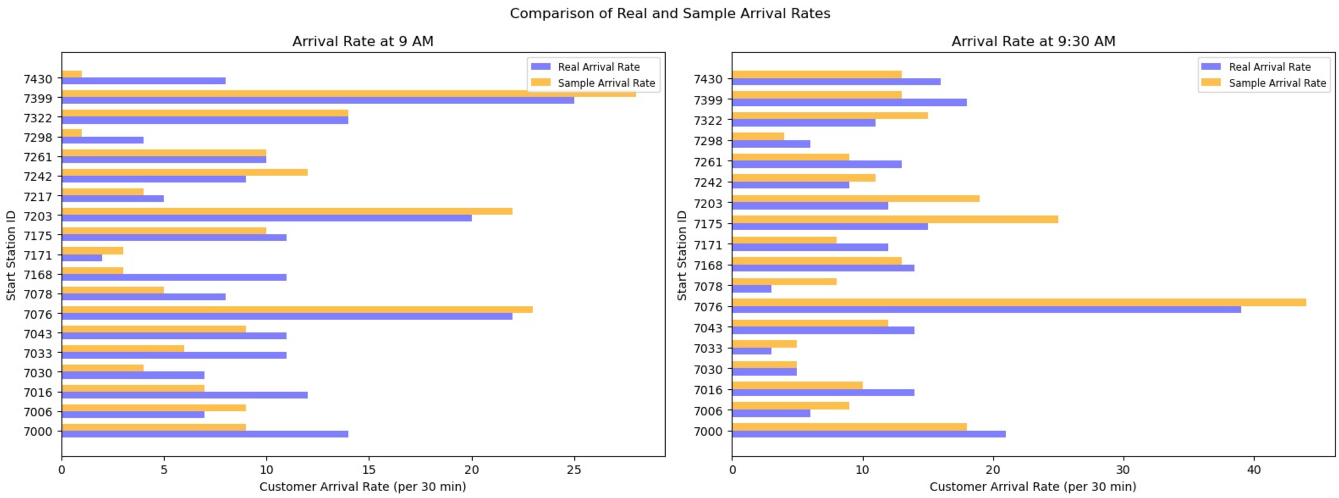
In [39]:
```
fig, ax = plt.subplots(1, 2, figsize=(16, 6))

# Plot for T=16
T = 20
real_Trip_Duration = subset_df[subset_df["End Time (per 30min)"] == T][["End Time (per 30min)", "End Station Id
avg_trip_time = np.round(real_Trip_Duration.groupby("End Station Id")["Trip_Duration"].mean())

sample_trip = end_flow_df[end_flow_df["End Time"]== T]["End ID"].value_counts()
sample_trips = pd.DataFrame({"End Station Id": sample_trip.index, "Sample Trip Times": sample_trip.values})

comparison_df = pd.merge(avg_trip_time, sample_trips,
                         on="End Station Id", how="outer")
comparison_df.columns = ["End Station Id","Real Trip Times", "Sample Trip Times"]
comparison_df = comparison_df.sort_values(by="End Station Id")

bar_width = 0.35
ax[0].barh(np.arange(len(comparison_df)), comparison_df["Real Trip Times"],
           color="darkblue", alpha=0.5, height=bar_width, label="Real Trip Time")
ax[0].barh(np.arange(len(comparison_df))+bar_width, comparison_df["Sample Trip Times"],
           color="red", alpha=0.5, height=bar_width, label="Sample Trip Time")
ax[0].set_yticks(np.arange(len(comparison_df))+bar_width / 2)
ax[0].set_yticklabels(comparison_df["End Station Id"])
ax[0].set_ylabel("End Station ID")
ax[0].set_xlabel("Customer Trip Time")
ax[0].set_title("Trip Durations at 9:00 AM")
ax[0].legend(loc='upper right', fontsize='small')

# Plot for T=17
T = 21
real_Trip_Duration = subset_df[subset_df["End Time (per 30min)"] == T][["End Time (per 30min)", "End Station Id
avg_trip_time = np.round(real_Trip_Duration.groupby("End Station Id")["Trip_Duration"].mean())


sample_trip = end_flow_df[end_flow_df["End Time"]
                          == T]["End ID"].value_counts()
sample_trips = pd.DataFrame(
    {"End Station Id": sample_trip.index, "Sample Trip Times": sample_trip.values})

comparison_df = pd.merge(avg_trip_time, sample_trips,
                         on="End Station Id", how="outer")
comparison_df.columns = ["End Station Id",
                         "Real Trip Times", "Sample Trip Times"]
comparison_df = comparison_df.sort_values(by="End Station Id")

ax[1].barh(np.arange(len(comparison_df)), comparison_df["Real Trip Times"],color="darkblue", alpha=0.5, height=
ax[1].barh(np.arange(len(comparison_df))+bar_width, comparison_df["Sample Trip Times"], color="red", alpha=0.5,
ax[1].set_yticks(np.arange(len(comparison_df))+bar_width / 2)
ax[1].set_yticklabels(comparison_df["End Station Id"])
ax[1].set_ylabel("End Station ID")
ax[1].set_xlabel("Customer Trip Time")
ax[1].set_title("Trip Durations at 9:30 AM")
ax[1].legend(loc='upper right', fontsize='small')
fig.suptitle("Comparison of Real and Sample Trip Durations")
fig.tight_layout()
plt.show()
```



Arrival Rate Verfication

In [40]:
```
fig, axs = plt.subplots(1, 2, figsize=(16, 6))

# First subplot for T=16
T = 18
```

```
real_8am_S_ID = arrival_df[arrival_df["Start Time (per 30min)"] == T][["Start Station Id", "ArrivalRate (per 30
sample_8am_s_ID = start_flow_df[start_flow_df["Start Time"]== T]["Start ID"].value_counts()
sample_8am_s_ID_df = pd.DataFrame({"Start Station Id": sample_8am_s_ID.index, "ArrivalRate (per 30min)": sample_

comparison_df = pd.merge(real_8am_S_ID, sample_8am_s_ID_df, on="Start Station Id", how="outer")
comparison_df.columns = ["Start Station Id", "Real ArrivalRate (per 30min)", "Sample ArrivalRate (per 30min)"]
comparison_df = comparison_df.sort_values(by="Start Station Id")

bar_width = 0.35
axs[0].barh(np.arange(len(comparison_df)), comparison_df["Real ArrivalRate (per 30min)"],color="blue", alpha=0.
axs[0].barh(np.arange(len(comparison_df))+bar_width, comparison_df["Sample ArrivalRate (per 30min)"],color="ora
axs[0].set_yticks(np.arange(len(comparison_df))+bar_width / 2)
axs[0].set_yticklabels(comparison_df["Start Station Id"])
axs[0].set_ylabel("Start Station ID")
axs[0].set_xlabel("Customer Arrival Rate (per 30 min)")
axs[0].set_title("Arrival Rate at 9 AM")
axs[0].legend(loc='upper right', fontsize='small')

# Second subplot for T=17
T = 19
real_8am_S_ID = arrival_df[arrival_df["Start Time (per 30min)"] == T][["Start Station Id", "ArrivalRate (per 30
sample_8am_s_ID = start_flow_df[start_flow_df["Start Time"]== T]["Start ID"].value_counts()
sample_8am_s_ID_df = pd.DataFrame({"Start Station Id": sample_8am_s_ID.index, "ArrivalRate (per 30min)": sample_

comparison_df = pd.merge(real_8am_S_ID, sample_8am_s_ID_df,on="Start Station Id", how="outer")
comparison_df.columns = ["Start Station Id", "Real ArrivalRate (per 30min)", "Sample ArrivalRate (per 30min)"]
comparison_df = comparison_df.sort_values(by="Start Station Id")

bar_width = 0.35
axs[1].barh(np.arange(len(comparison_df)), comparison_df["Real ArrivalRate (per 30min)"],color="blue", alpha=0.
axs[1].barh(np.arange(len(comparison_df))+bar_width, comparison_df["Sample ArrivalRate (per 30min)"],color="ora
axs[1].set_yticks(np.arange(len(comparison_df))+bar_width / 2)
axs[1].set_yticklabels(comparison_df["Start Station Id"])
axs[1].set_ylabel("Start Station ID")
axs[1].set_xlabel("Customer Arrival Rate (per 30 min)")
axs[1].set_title("Arrival Rate at 9:30 AM")
axs[1].legend(loc='upper right', fontsize='small')

fig.suptitle("Comparison of Real and Sample Arrival Rates")
fig.tight_layout()
plt.show()
```



Starting Solutions

Equal Allocation

In [41]:

```
unique_stations = np.unique(subset_df["Start Station Id"].values)
num_stations = len(unique_stations)
total_capacity = 500
total_bikes = 399
Stations = {}
level_sum = 0
level_sum = 0
capacity_sum = 0


for i, station in enumerate(unique_stations):
    x_i = 399//19
    r_i = capacity_per_station
    if i == num_stations - 1:
        # Allocate the remaining capacity to the last station
        r_i = total_capacity - capacity_per_station * (num_stations - 1)
    Stations[station] = Station(station_id=station, level=x_i, capacity=r_i)
```

```python
    level_sum += x_i
    for station_id, station in Stations.items():
        capacity_sum += station.capacity


StationDict = {station.id: station for station in Stations.values()}

# Print the initial bike list for each station
val_level_sum = []
val_capacity_sum = []
count = 0
for station_id, station in Stations.items():
    count += 1
    print(f"Station {station_id} Initial Bike List: {station.Get_Bike_List()}")
    val_level_sum.append(station.level)
    val_capacity_sum.append(station.capacity)
print(count)

print("Level Sum", sum(val_level_sum))
print("Capacity Sum", sum(val_capacity_sum))
```

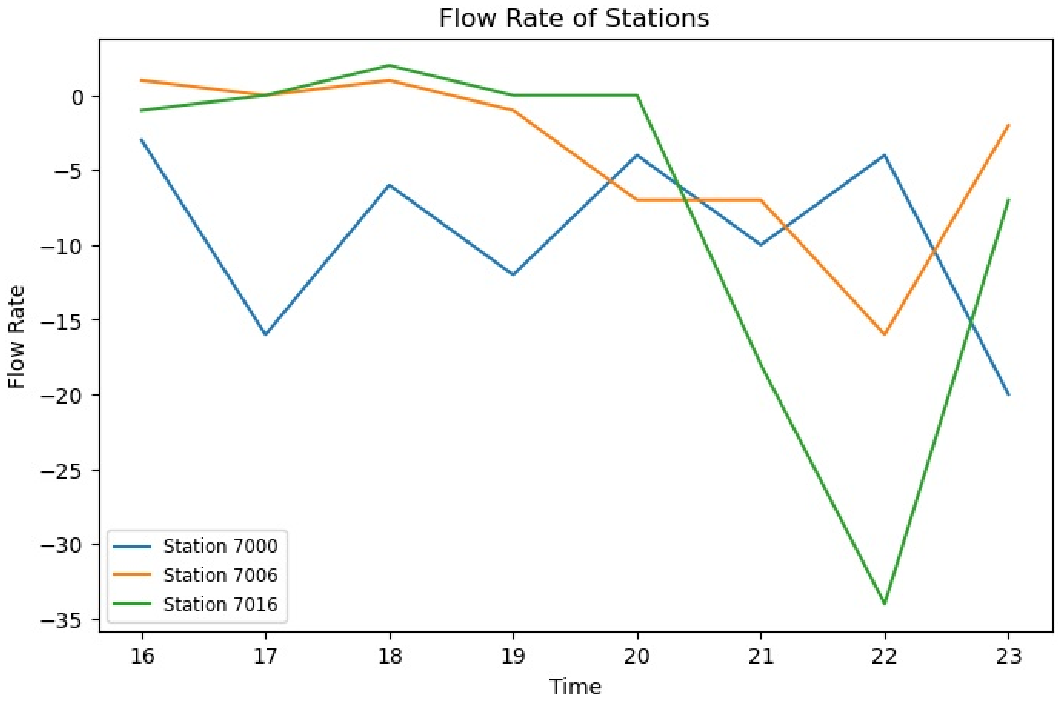
```
Station 7000 Initial Bike List: ['7000-1', '7000-2', '7000-3', '7000-4', '7000-5', '7000-6', '7000-7', '7000-8'
, '7000-9', '7000-10', '7000-11', '7000-12', '7000-13', '7000-14', '7000-15', '7000-16', '7000-17', '7000-18',
'7000-19', '7000-20', '7000-21']
Station 7006 Initial Bike List: ['7006-1', '7006-2', '7006-3', '7006-4', '7006-5', '7006-6', '7006-7', '7006-8'
, '7006-9', '7006-10', '7006-11', '7006-12', '7006-13', '7006-14', '7006-15', '7006-16', '7006-17', '7006-18',
'7006-19', '7006-20', '7006-21']
Station 7016 Initial Bike List: ['7016-1', '7016-2', '7016-3', '7016-4', '7016-5', '7016-6', '7016-7', '7016-8'
, '7016-9', '7016-10', '7016-11', '7016-12', '7016-13', '7016-14', '7016-15', '7016-16', '7016-17', '7016-18',
'7016-19', '7016-20', '7016-21']
Station 7030 Initial Bike List: ['7030-1', '7030-2', '7030-3', '7030-4', '7030-5', '7030-6', '7030-7', '7030-8'
, '7030-9', '7030-10', '7030-11', '7030-12', '7030-13', '7030-14', '7030-15', '7030-16', '7030-17', '7030-18',
'7030-19', '7030-20', '7030-21']
Station 7033 Initial Bike List: ['7033-1', '7033-2', '7033-3', '7033-4', '7033-5', '7033-6', '7033-7', '7033-8'
, '7033-9', '7033-10', '7033-11', '7033-12', '7033-13', '7033-14', '7033-15', '7033-16', '7033-17', '7033-18',
'7033-19', '7033-20', '7033-21']
Station 7043 Initial Bike List: ['7043-1', '7043-2', '7043-3', '7043-4', '7043-5', '7043-6', '7043-7', '7043-8'
, '7043-9', '7043-10', '7043-11', '7043-12', '7043-13', '7043-14', '7043-15', '7043-16', '7043-17', '7043-18',
'7043-19', '7043-20', '7043-21']
Station 7076 Initial Bike List: ['7076-1', '7076-2', '7076-3', '7076-4', '7076-5', '7076-6', '7076-7', '7076-8'
, '7076-9', '7076-10', '7076-11', '7076-12', '7076-13', '7076-14', '7076-15', '7076-16', '7076-17', '7076-18',
'7076-19', '7076-20', '7076-21']
Station 7078 Initial Bike List: ['7078-1', '7078-2', '7078-3', '7078-4', '7078-5', '7078-6', '7078-7', '7078-8'
, '7078-9', '7078-10', '7078-11', '7078-12', '7078-13', '7078-14', '7078-15', '7078-16', '7078-17', '7078-18',
'7078-19', '7078-20', '7078-21']
Station 7168 Initial Bike List: ['7168-1', '7168-2', '7168-3', '7168-4', '7168-5', '7168-6', '7168-7', '7168-8'
, '7168-9', '7168-10', '7168-11', '7168-12', '7168-13', '7168-14', '7168-15', '7168-16', '7168-17', '7168-18',
'7168-19', '7168-20', '7168-21']
Station 7171 Initial Bike List: ['7171-1', '7171-2', '7171-3', '7171-4', '7171-5', '7171-6', '7171-7', '7171-8'
, '7171-9', '7171-10', '7171-11', '7171-12', '7171-13', '7171-14', '7171-15', '7171-16', '7171-17', '7171-18',
'7171-19', '7171-20', '7171-21']
Station 7175 Initial Bike List: ['7175-1', '7175-2', '7175-3', '7175-4', '7175-5', '7175-6', '7175-7', '7175-8'
, '7175-9', '7175-10', '7175-11', '7175-12', '7175-13', '7175-14', '7175-15', '7175-16', '7175-17', '7175-18',
'7175-19', '7175-20', '7175-21']
Station 7203 Initial Bike List: ['7203-1', '7203-2', '7203-3', '7203-4', '7203-5', '7203-6', '7203-7', '7203-8'
, '7203-9', '7203-10', '7203-11', '7203-12', '7203-13', '7203-14', '7203-15', '7203-16', '7203-17', '7203-18',
'7203-19', '7203-20', '7203-21']
Station 7217 Initial Bike List: ['7217-1', '7217-2', '7217-3', '7217-4', '7217-5', '7217-6', '7217-7', '7217-8'
, '7217-9', '7217-10', '7217-11', '7217-12', '7217-13', '7217-14', '7217-15', '7217-16', '7217-17', '7217-18',
'7217-19', '7217-20', '7217-21']
Station 7242 Initial Bike List: ['7242-1', '7242-2', '7242-3', '7242-4', '7242-5', '7242-6', '7242-7', '7242-8'
, '7242-9', '7242-10', '7242-11', '7242-12', '7242-13', '7242-14', '7242-15', '7242-16', '7242-17', '7242-18',
'7242-19', '7242-20', '7242-21']
Station 7261 Initial Bike List: ['7261-1', '7261-2', '7261-3', '7261-4', '7261-5', '7261-6', '7261-7', '7261-8'
, '7261-9', '7261-10', '7261-11', '7261-12', '7261-13', '7261-14', '7261-15', '7261-16', '7261-17', '7261-18',
'7261-19', '7261-20', '7261-21']
Station 7298 Initial Bike List: ['7298-1', '7298-2', '7298-3', '7298-4', '7298-5', '7298-6', '7298-7', '7298-8'
, '7298-9', '7298-10', '7298-11', '7298-12', '7298-13', '7298-14', '7298-15', '7298-16', '7298-17', '7298-18',
'7298-19', '7298-20', '7298-21']
Station 7322 Initial Bike List: ['7322-1', '7322-2', '7322-3', '7322-4', '7322-5', '7322-6', '7322-7', '7322-8'
, '7322-9', '7322-10', '7322-11', '7322-12', '7322-13', '7322-14', '7322-15', '7322-16', '7322-17', '7322-18',
'7322-19', '7322-20', '7322-21']
Station 7399 Initial Bike List: ['7399-1', '7399-2', '7399-3', '7399-4', '7399-5', '7399-6', '7399-7', '7399-8'
, '7399-9', '7399-10', '7399-11', '7399-12', '7399-13', '7399-14', '7399-15', '7399-16', '7399-17', '7399-18',
'7399-19', '7399-20', '7399-21']
Station 7430 Initial Bike List: ['7430-1', '7430-2', '7430-3', '7430-4', '7430-5', '7430-6', '7430-7', '7430-8'
, '7430-9', '7430-10', '7430-11', '7430-12', '7430-13', '7430-14', '7430-15', '7430-16', '7430-17', '7430-18',
'7430-19', '7430-20', '7430-21']
19
Level Sum 399
Capacity Sum 500
```

Flow Rate Model Solution

In [42]:

```python
plt.figure(figsize=(8, 5))
unique_stations = np.unique(subset_df["Start Station Id"].values)
```

```python
for station_ids in unique_stations[:3]:
    start_flow = start_flow_count.loc[station_ids]
    end_flow = end_flow_count.loc[station_ids]
    flow_rate = end_flow - start_flow
    plt.plot(flow_rate.index, flow_rate.values, label=f'Station {station_ids}')

plt.xlabel('Time')
plt.ylabel('Flow Rate')
plt.title('Flow Rate of Stations')
plt.legend(loc='lower left', fontsize='small')
plt.show()
```



Optimizing Bike Allocations

- Time Frame: 8am - 12pm
- number of stations = 20
- station capcities: 12 <= r_i <= 32
- Bike per station: 0 <= x_i <= r_i
- total capacity of stations = 500
- total number of bikes = 399

RUN SIMULATION FUNCTION

In [74]:

```python
def run_simulation(intial_StationDict, T, Calendar):
    global CI_total_error_list, CI_Full_Error_list, CI_Empty_Error_list
    global CI_inter_arrival_time_list, CI_arrival_time_list, CI_start_station_id_list, CI_end_station_id_list, 
    objective_fun_list = []
    new_StationDict = deepcopy(intial_StationDict)

    for i in range(3):
        print("loop", i+1)
        Stations = {}
        val_level_sum = []
        val_capacity_sum = []
        x_i = 0
        r_i = 0
        total_bikes_assigned = 0

        for i, (station_id, station) in enumerate(new_StationDict.items()):
            x_i = station.level
            r_i = station.capacity
            Stations[station_id] = Station(station_id=station_id, level=x_i, capacity=r_i)
            total_bikes_assigned += x_i

        for station_id, station in Stations.items():
            #print(f"Station {station_id} Initial Bike List: {station.Get_Bike_List()}")
            val_level_sum.append(station.level)
            val_capacity_sum.append(station.capacity)
        print("Level Sum", sum(val_level_sum))
        print("Capacity Sum", sum(val_capacity_sum))
```

```
    NextCustomerID.counter = 0

    ZSimRNG = SimRNG_Modified.InitializeRNSeed()

    TheCTStats = []
    TheDTStats = []
    TheQueues = []
    TheResources = []


    total_capacity = 500
    total_bikes = 399
    Full_Error = 0
    Empty_Error = 0
    inital_count = 0


    total_capacity = 500
    total_bikes = 399
    Stations = {}
    level_sum = 0
    level_sum = 0
    capacity_sum = 0
    Full_Error = 0
    Empty_Error = 0
    inital_count = 0


    Full_Error = 0
    Empty_Error = 0
    inital_count = 0

    ##############################################################################################################
    unique_stations = np.unique(subset_df["Start Station Id"].values)
    num_stations = len(unique_stations)
    total_capacity = 500
    total_bikes = 399
    Stations = {}
    level_sum = 0
    level_sum = 0
    capacity_sum = 0
    Full_Error = 0
    Empty_Error = 0
    inital_count = 0


    ##################################

##################################################################################################################

##################################################################################################################

    SimFunctions.SimFunctionsInit(Calendar, TheQueues, TheCTStats, TheDTStats, TheResources)
    SimFunctions.Schedule(Calendar, "Start", 0)

    NextEvent = Calendar.Remove()
    SimClasses.Clock = NextEvent.EventTime
    if NextEvent.EventType == "Start":
        Start()

##################################################################################################################
# SIMULATION RUN
    for T in range(15, 25):  # T = hours intervals
        inital_count += 1
        hour = T // 2
        minute = 00 if T % 2 == 0 else 30
        unit = 'PM' if hour >= 12 else 'AM'
        #print()
        if T == 16 or T == 20 or T == 24:
            print("Interval:", T)
        mini = 0
        SimFunctions.Schedule(Calendar, "inital_Customer_Arrival", 0)

        if inital_count == 2:
            Full_Error = 0
            Empty_Error = 0

        while True:
            #print("Clock: {:02d}:{:02d} {:s}".format(hour, minute, unit))
            NextEvent = Calendar.Remove()
            SimClasses.Clock = NextEvent.EventTime
            minute = int((SimClasses.Clock) % 60)
            if SimClasses.Clock >= (T+1) * 30:
                break
            if NextEvent.EventType == "inital_Customer_Arrival":
                Empty_Error = inital_Customer_Arrival(Empty_Error, CustomerList, T, minute)
            elif NextEvent.EventType == "Customer_Arrival":
                Empty_Error = Customer_Arrival(Empty_Error, CustomerList, T, minute)
            elif NextEvent.EventType == "Bike_Arrival":
```

```
                Full_Error = Bike_Arrival(Full_Error, CustomerList, T, minute)

        # if T == 24:
        #     while Calendar.N() > 0:
        #         NextEvent = Calendar.Remove()
        #         SimClasses.Clock = NextEvent.EventTime
        #         minute = int((SimClasses.Clock) % 60)
        #         if NextEvent.EventType == "Bike_Arrival":
        #             Full_Error = Bike_Arrival(Full_Error, CustomerList, T, minute)
    ##########################################################################################

        objective_fun = Full_Error + Empty_Error
        objective_fun_list.append(objective_fun)
    CI_total_error_list.append(objective_fun)
    CI_Full_Error_list.append(Full_Error)
    CI_Empty_Error_list.append(Empty_Error)

    CI_inter_arrival_time_list.append(inter_arrival_time_list)
    CI_arrival_time_list.append(arrival_time_list)
    CI_start_station_id_list.append(start_station_id_list)
    CI_end_station_id_list.append(end_station_id_list)
    CI_start_time_list.append(start_time_list)
    CI_end_time_list.append(end_time_list)
    CI_trip_time_list.append(trip_time_list)

    print("CI_95 of ERRORS", CI_95(objective_fun_list))
    print(f"End of BikeSim")
    print("----------------------------------------------------------------------------------------------
    return CI_95(objective_fun_list)[0]
```

OPT Sim Function

In [326…

```
def OPT_RUN_SIMULATION(intial_StationDict, T, Calendar):
    global CI_total_error_list, CI_Full_Error_list, CI_Empty_Error_list
    global CI_inter_arrival_time_list, CI_arrival_time_list, CI_start_station_id_list, CI_end_station_id_list,
    objective_fun_list = []
    new_StationDict = deepcopy(intial_StationDict)

    NextCustomerID.counter = 0
    ZSimRNG = SimRNG_Modified.InitializeRNSeed()
    np.random.seed(1)

    Stations = {}
    val_level_sum = []
    val_capacity_sum = []
    x_i = 0
    r_i = 0

    TheCTStats = []
    TheDTStats = []
    TheQueues = []
    TheResources = []

    total_capacity = 500
    total_bikes = 399
    Full_Error = 0
    Empty_Error = 0
    inital_count = 0
    Stations = {}
    level_sum = 0
    level_sum = 0
    capacity_sum = 0
    Full_Error = 0
    Empty_Error = 0
    inital_count = 0

    for i, (station_id, station) in enumerate(new_StationDict.items()):
        x_i = station.level
        r_i = station.capacity
        Stations[station_id] = Station(station_id=station_id, level=x_i, capacity=r_i)

    for station_id, station in Stations.items():
        print(f"    Station {station_id} Initial Bike List: {station.Get_Bike_List()}")
        #print("        Capacity",station.capacity)
        val_level_sum.append(station.level)
        val_capacity_sum.append(station.capacity)
    print("     Level Sum", sum(val_level_sum))
    print("     Capacity Sum", sum(val_capacity_sum))

    ##########################################################################################
    unique_stations = np.unique(subset_df["Start Station Id"].values)
    num_stations = len(unique_stations)
    Stations = {}

################################################################################################
```

```
    SimFunctions.SimFunctionsInit(
        Calendar, TheQueues, TheCTStats, TheDTStats, TheResources)
    SimFunctions.Schedule(Calendar, "Start", 0)

    NextEvent = Calendar.Remove()
    SimClasses.Clock = NextEvent.EventTime
    if NextEvent.EventType == "Start":
        Start()

##############################################################################################################
# SIMULATION RUN
    for T in range(15, 25):  # T = hours intervals
        inital_count += 1
        hour = T // 2
        minute = 00 if T % 2 == 0 else 30
        unit = 'PM' if hour >= 12 else 'AM'
        # print()
        if T == 16 or T == 20 or T == 24:
            print("             Interval:", T//2)
        mini = 0
        SimFunctions.Schedule(Calendar, "inital_Customer_Arrival", 0)

        if inital_count == 2:
            Full_Error = 0
            Empty_Error = 0

        while True:
            #print("Clock: {:02d}:{:02d} {:s}".format(hour, minute, unit))
            NextEvent = Calendar.Remove()
            SimClasses.Clock = NextEvent.EventTime
            minute = int((SimClasses.Clock) % 60)
            if SimClasses.Clock >= (T+1) * 30:
                break
            if NextEvent.EventType == "inital_Customer_Arrival":
                Empty_Error = inital_Customer_Arrival(
                    Empty_Error, CustomerList, T, minute)
            elif NextEvent.EventType == "Customer_Arrival":
                Empty_Error = Customer_Arrival(
                    Empty_Error, CustomerList, T, minute)
            elif NextEvent.EventType == "Bike_Arrival":
                Full_Error = Bike_Arrival(
                    Full_Error, CustomerList, T, minute)

        if T == 24:
            while Calendar.N() > 1:
                NextEvent = Calendar.Remove()
                SimClasses.Clock = NextEvent.EventTime
                minute = int((SimClasses.Clock) % 60)
                if NextEvent.EventType == "Bike_Arrival":
                    Full_Error = Bike_Arrival(Full_Error, CustomerList, T, minute)
##############################################################################################################
    objective_fun = 0
    objective_fun = Full_Error + Empty_Error
    #print(f"Error = {objective_fun}")
    objective_fun_list.append(objective_fun)

    CI_total_error_list.append(objective_fun)
    CI_Full_Error_list.append(Full_Error)
    CI_Empty_Error_list.append(Empty_Error)

    CI_inter_arrival_time_list.append(inter_arrival_time_list)
    CI_arrival_time_list.append(arrival_time_list)
    CI_start_station_id_list.append(start_station_id_list)
    CI_end_station_id_list.append(end_station_id_list)
    CI_start_time_list.append(start_time_list)
    CI_end_time_list.append(end_time_list)
    CI_trip_time_list.append(trip_time_list)

    print("ERROR", objective_fun)
    print(f"End of BikeSim")
    #print("-----------------------------------------------------------------------------------------------
    return objective_fun
```

Heuristic 1

In [327…

```
def generate_trial_solution(inital_stations_dict, Stations, w):
    new_stations = deepcopy(inital_stations_dict)

    statE = [station_id for station_id, station in Stations.items() if station.level + w <= station.capacity]
    statF = [station_id for station_id, station in Stations.items() if station.level - w >= 0]

    sE = np.random.choice(statE)
    sF = np.random.choice(statF)

    new_stations[sE].level += w
```

```
    new_stations[sF].level -= w

    total_bikes_assigned = sum(station.level for station in new_stations.values())
    remaining_bikes = 399 - total_bikes_assigned
    sorted_stations = sorted(new_stations.items(), key=lambda x: x[1].level)

    while remaining_bikes > 0:
        for station_id, station in sorted_stations:
            if station.level < station.capacity and remaining_bikes > 0:
                station.level += 1
                remaining_bikes -= 1
                if remaining_bikes == 0:
                    break

    Mod_Stations = {}
    x_i = 0
    r_i = 0

    for station_id, station in new_stations.items():
        #print(f"Inital Station {station_id} Bike List: {station.Get_Bike_List()}")
        x_i = station.level
        r_i = station.capacity
        Mod_Stations[station_id] = Station(station_id=station_id, level=x_i, capacity=r_i)

    for station_id, station in Mod_Stations.items():
        print(f"Station {station_id} NEW Bike List: {station.Get_Bike_List()}")

    return Mod_Stations
```

Run 1

In [2]:

```
inter_arrival_time_list = []
arrival_time_list = []
start_station_id_list = []
end_station_id_list = []
start_time_list = []
end_time_list = []
trip_time_list = []

Stations = []
CustomerList = []
CI_Full_Error_list = []
CI_Empty_Error_list = []

CI_inter_arrival_time_list = []
CI_arrival_time_list = []
CI_start_station_id_list = []
CI_end_station_id_list = []
CI_start_time_list = []
CI_end_time_list = []
CI_trip_time_list = []
CI_total_error_list = []

Full_Error = 0
Empty_Error = 0
inital_count = 0

Stations = []
CustomerList = []

Calendar = SimClasses.EventCalendar()
ZSimRNG = SimRNG_Modified.InitializeRNSeed()

unique_stations = np.unique(subset_df["Start Station Id"].values)
num_stations = len(unique_stations)
total_capacity = 500
total_bikes = 399
min_cap = 12
max_cap = 32

level_sum = 0
level_sum = 0
capacity_sum = 0
Full_Error = 0
Empty_Error = 0
inital_count = 0

Stations = {}
best_total_error = float('inf')  # Set best_total_error to a high initial value
best_bike_list = {}
num_replications = 1
opt_error_list = []

num_iterations = 20
num_reps = 2
w = 2
```

```
####################################################################################################
# Initialize the stations and create a dictionary mapping station IDs to Station instances

capacity_per_station = total_capacity // num_stations
for i, station in enumerate(unique_stations):
    x_i = total_bikes // 19
    r_i = capacity_per_station
    if i == num_stations - 1:
        # Allocate the remaining capacity to the last station
        r_i = total_capacity - capacity_per_station * (num_stations - 1)
    Stations[station] = Station(station_id=station, level=x_i, capacity=r_i)
    level_sum += x_i
    for station_id, station in Stations.items():
        capacity_sum += station.capacity

StationDict = {station.id: station for station in Stations.values()}
inital_stations_dict = deepcopy(StationDict)
####################################################################################################
#Initial Run
new_total_error = OPT_RUN_SIMULATION(StationDict, T, Calendar)

if new_total_error < best_total_error:
    best_total_error = new_total_error
    inital_stations_dict = (inital_stations_dict)

station_id = CI_start_station_id_list[-1]
end_station_id = CI_end_station_id_list[-1]
start_times = CI_start_time_list[-1]
end_times = CI_end_time_list[-1]

start_flow_df = pd.DataFrame({"Start ID": station_id, "Start Time": start_times})
end_flow_df = pd.DataFrame({"End ID": end_station_id, "End Time": end_times})

start_flow_count = start_flow_df.pivot_table(index='Start ID', columns='Start Time', aggfunc='size', fill_value
start_flow_count = start_flow_count.loc[:, 16:]
end_flow_count = end_flow_df.pivot_table(index='End ID', columns='End Time', aggfunc='size', fill_value=0)
end_flow_count = end_flow_count.loc[:, 16:]
flow_rate = end_flow_count - start_flow_count
CI_95_LIST = []
avg_error_1 = []
####################################################################################################
for iterations in range(num_iterations):
    # Generate trial solution
    trial_solution = generate_trial_solution(inital_stations_dict, StationDict, w)

    # Simulate and evaluate
    rep_error_list = []
    for reps in range(num_reps):
        Calendar = SimClasses.EventCalendar()
        ZSimRNG = SimRNG_Modified.InitializeRNSeed()
        rep_error = OPT_RUN_SIMULATION(trial_solution, T, Calendar)
        rep_error_list.append(rep_error)

    total_error_rep = CI_95(rep_error_list)
    CI_95_LIST.append(total_error_rep)
    new_total_error = total_error_rep[0]
    avg_error_1.append(new_total_error)

    print(f"Iteration: {iterations +1} -> Total Error: {total_error_rep}")
    print("------------------------------------------------------------------------------------------------------

    # If the new total error is better than the current best total error, update the inital_stations_dict
    if new_total_error < best_total_error:
        best_total_error = new_total_error
        inital_stations_dict = deepcopy(trial_solution)

print("Best Bike List")
for station_id, station in inital_stations_dict.items():
    print(f"Station {station_id} Optimal Bike List: {station.Get_Bike_List()}")

print(f"List of Errors: {avg_error_1}")
```

In [ ]:

```
def generate_trial_solution(inital_stations_dict, Stations, w):
    new_stations = deepcopy(inital_stations_dict)

    statE = [station_id for station_id, station in Stations.items(
    ) if station.level + w <= station.capacity]
    statF = [station_id for station_id,
             station in Stations.items() if station.level - w >= 0]

    sE = np.random.choice(statE)
    sF = np.random.choice(statF)

    new_stations[sE].level += w
    new_stations[sF].level -= w

    total_bikes_assigned = sum(
        station.level for station in new_stations.values())
```

```
    remaining_bikes = 399 - total_bikes_assigned
    sorted_stations = sorted(new_stations.items(), key=lambda x: x[1].level)

    while remaining_bikes > 0:
        for station_id, station in sorted_stations:
            if station.level < station.capacity and remaining_bikes > 0:
                station.level += 1
                remaining_bikes -= 1
                if remaining_bikes == 0:
                    break

    Mod_Stations = {}
    x_i = 0
    r_i = 0

    for station_id, station in new_stations.items():
        #print(f"Inital Station {station_id} Bike List: {station.Get_Bike_List()}")
        x_i = station.level
        r_i = station.capacity
        Mod_Stations[station_id] = Station(
            station_id=station_id, level=x_i, capacity=r_i)

    for station_id, station in Mod_Stations.items():
        print(f"Station {station_id} NEW Bike List: {station.Get_Bike_List()}")

    return Mod_Stations


for iterations in range(num_iterations):
    # Generate trial solution
    trial_solution = generate_trial_solution(
        inital_stations_dict, StationDict, w)

    # Simulate and evaluate
    rep_error_list = []
    for reps in range(num_reps):
        Calendar = SimClasses.EventCalendar()
        ZSimRNG = SimRNG_Modified.InitializeRNSeed()
        rep_error = OPT_RUN_SIMULATION(trial_solution, T, Calendar)
        rep_error_list.append(rep_error)

    total_error_rep = CI_95(rep_error_list)
    CI_95_LIST.append(total_error_rep)
    new_total_error = total_error_rep[0]
    avg_error_1.append(new_total_error)

    print(f"Iteration: {iterations +1} -> Total Error: {total_error_rep}")
    print("----------------------------------------------------------------------------------------------------

    # If the new total error is better than the current best total error, update the inital_stations_dict
    if new_total_error < best_total_error:
        best_total_error = new_total_error
        inital_stations_dict = deepcopy(trial_solution)

print("Best Bike List")
for station_id, station in inital_stations_dict.items():
    print(f"Station {station_id} Optimal Bike List: {station.Get_Bike_List()}")

print(f"List of Errors: {avg_error_1}")
```

Heuristic 2

In [351…

```
top_stations = flow_rate.sum(axis=1).nlargest(1).index.tolist()
bottom_stations = flow_rate.sum(axis=1).nsmallest(5).index.tolist()
```

In [333…

```
def generate_trial_solution(inital_stations_dict, flow_rate, w):
    new_stations = deepcopy(inital_stations_dict)

    # # Find the stations with the highest and lowest flow rates
    top_stations = flow_rate.sum(axis=1).nlargest(5).index.tolist()
    bottom_stations = flow_rate.sum(axis=1).nsmallest(5).index.tolist()

    station_id1 = np.random.choice(top_stations)
    station_id2 = np.random.choice(bottom_stations)

    new_stations[station_id1].level -= w
    new_stations[station_id2].level += w

    total_bikes_assigned = sum(station.level for station in new_stations.values())
    remaining_bikes = 399 - total_bikes_assigned
    sorted_stations = sorted(new_stations.items(), key=lambda x: x[1].level)

    while remaining_bikes > 0:
        for station_id, station in sorted_stations:
            if station.level < station.capacity and remaining_bikes > 0:
                station.level += 1
```

```
                remaining_bikes -= 1
                if remaining_bikes == 0:
                    break

    Mod_Stations = {}
    x_i = 0
    r_i = 0

    for station_id, station in new_stations.items():
        print(f"Inital Station {station_id} Bike List: {station.Get_Bike_List()}")
        x_i = station.level
        r_i = station.capacity
        Mod_Stations[station_id] = Station(
            station_id=station_id, level=x_i, capacity=r_i)

    #for station_id, station in Mod_Stations.items():
     #   print(f"Station {station_id} NEW Bike List: {station.Get_Bike_List()}")

    return Mod_Stations
```

Run 2

```
In [ ]: NextCustomerID.counter = 0

ZSimRNG = SimRNG_Modified.InitializeRNSeed()
np.random.seed(1)

Calendar = SimClasses.EventCalendar()
TheCTStats = []
TheDTStats = []
TheQueues = []
TheResources = []
Stations = []
CustomerList = []
CI_Full_Error_list = []
CI_Empty_Error_list = []

CI_inter_arrival_time_list = []
CI_arrival_time_list = []
CI_start_station_id_list = []
CI_end_station_id_list = []
CI_start_time_list = []
CI_end_time_list = []
CI_trip_time_list = []


CI_total_error_list = []

for days in range(0, 3, 1):
    Full_Error = 0
    Empty_Error = 0
    inital_count = 0

    inter_arrival_time_list = []
    arrival_time_list = []
    start_station_id_list = []
    end_station_id_list = []
    start_time_list = []
    end_time_list = []
    trip_time_list = []
##############################################################################################################
    # Initialize the stations and create a dictionary mapping station IDs to Station instances
    unique_stations = np.unique(subset_df["Start Station Id"].values)
    num_stations = len(unique_stations)
    total_capacity = 500
    total_bikes = 399
    Stations = {}
    level_sum = 0
    level_sum = 0
    capacity_sum = 0
    capacity_per_station = total_capacity // num_stations

    for i, station in enumerate(unique_stations):
        x_i = total_bikes // 19
        r_i = capacity_per_station
        if i == num_stations - 1:
            # Allocate the remaining capacity to the last station
            r_i = total_capacity - capacity_per_station * (num_stations - 1)
        Stations[station] = Station(
            station_id=station, level=x_i, capacity=r_i)
        level_sum += x_i
        for station_id, station in Stations.items():
            capacity_sum += station.capacity

    StationDict = {station.id: station for station in Stations.values()}

    # Print the initial bike list for each station
```

```
    val_level_sum = []
    val_capacity_sum = []
    count = 0
    for station_id, station in Stations.items():
        count += 1
        print(
            f"Station {station_id} Initial Bike List: {station.Get_Bike_List()}")
        val_level_sum.append(station.level)
        val_capacity_sum.append(station.capacity)
    print("Number of Stations", count)
    print("Level Sum", sum(val_level_sum))
    print("Capacity Sum", sum(val_capacity_sum))

###############################################################################################################

    SimFunctions.SimFunctionsInit(
        Calendar, TheQueues, TheCTStats, TheDTStats, TheResources)
    SimFunctions.Schedule(Calendar, "Start", 0)

    NextEvent = Calendar.Remove()
    SimClasses.Clock = NextEvent.EventTime
    if NextEvent.EventType == "Start":
        Start()

###############################################################################################################
# SIMULATION RUN
    for T in range(15, 24):  # T = hours intervals
        inital_count += 1
        hour = T // 2
        minute = 00 if T % 2 == 0 else 30
        unit = 'PM' if hour >= 12 else 'AM'
        print()
        print("Interval:", T)
        mini = 0
        SimFunctions.Schedule(Calendar, "inital_Customer_Arrival", 0)

        if inital_count == 2:
            Full_Error = 0
            Empty_Error = 0

        while True:
            #print("Clock: {:02d}:{:02d} {:s}".format(hour, minute, unit))
            NextEvent = Calendar.Remove()
            SimClasses.Clock = NextEvent.EventTime
            minute = int((SimClasses.Clock) % 60)
            if SimClasses.Clock >= (T+1) * 30:
                break
            if NextEvent.EventType == "inital_Customer_Arrival":
                Empty_Error = inital_Customer_Arrival(
                    Empty_Error, CustomerList, T, minute)
            elif NextEvent.EventType == "Customer_Arrival":
                Empty_Error = Customer_Arrival(
                    Empty_Error, CustomerList, T, minute)
            elif NextEvent.EventType == "Bike_Arrival":
                Full_Error = Bike_Arrival(Full_Error, CustomerList, T, minute)

###############################################################################################################
# OPTIMIZE
    objective_fun = Full_Error + Empty_Error
    total_error_list.append(objective_fun)

# OPTIMZE

    CI_Full_Error_list.append(Full_Error)
    CI_Empty_Error_list.append(Empty_Error)
    CI_total_error_list.append(total_error_list)

    CI_inter_arrival_time_list.append(inter_arrival_time_list)
    CI_arrival_time_list.append(arrival_time_list)
    CI_start_station_id_list.append(start_station_id_list)
    CI_end_station_id_list.append(end_station_id_list)
    CI_start_time_list.append(start_time_list)
    CI_end_time_list.append(end_time_list)
    CI_trip_time_list.append(trip_time_list)

    print(f"End of Day {days}")
    print("----------------------------------------------------------------------------------------------------
    print()


Errors_DF = pd.DataFrame({"Full Error": CI_Full_Error_list,
                          "Empty Error": CI_Empty_Error_list,
                          "Total Error": CI_total_error_list})

BikeSim_DF = pd.DataFrame({"Start Station ID": CI_start_station_id_list,
                           "End Station ID": CI_end_station_id_list,
                           "Arrival Rate": CI_arrival_time_list,
                           "Interarrival Rate": CI_inter_arrival_time_list,
                           "Start Time": CI_start_time_list,
```

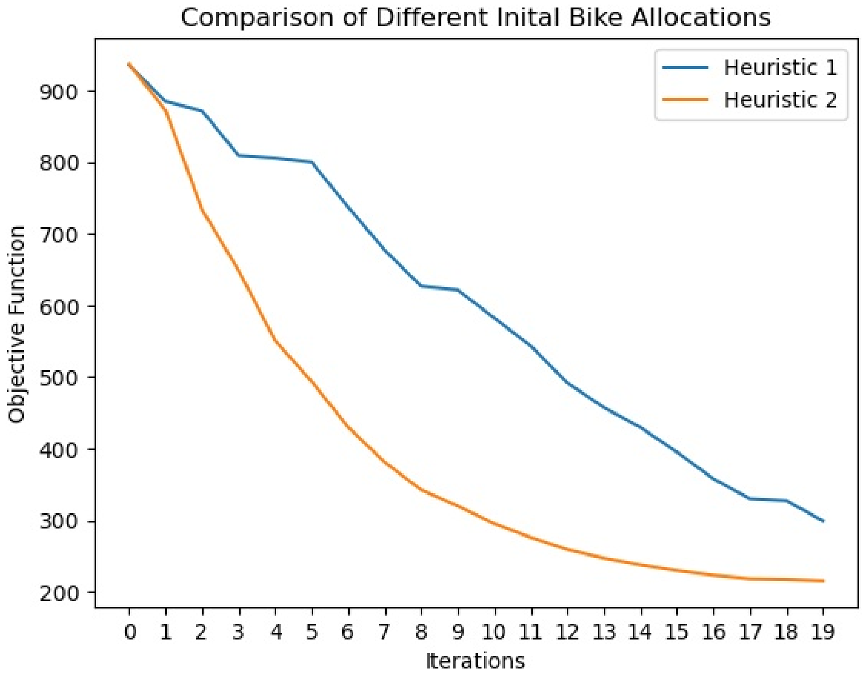
```
                          "End Time": CI_end_time_list,
                          "Trip Time": CI_trip_time_list})
print(f"Num of Full Errors: {CI_Full_Error_list}")
print(f"Num of Empty Errors: {CI_Empty_Error_list}")
print(f"Total Errors: {CI_95(total_error_list)}")
```

In [3]:
```
inter_arrival_time_list = []
arrival_time_list = []
start_station_id_list = []
end_station_id_list = []
start_time_list = []
end_time_list = []
trip_time_list = []

Stations = []
CustomerList = []
CI_Full_Error_list = []
CI_Empty_Error_list = []

CI_inter_arrival_time_list = []
CI_arrival_time_list = []
CI_start_station_id_list = []
CI_end_station_id_list = []
CI_start_time_list = []
CI_end_time_list = []
CI_trip_time_list = []
CI_total_error_list = []

Full_Error = 0
Empty_Error = 0
inital_count = 0

Stations = []
CustomerList = []

Calendar = SimClasses.EventCalendar()
ZSimRNG = SimRNG_Modified.InitializeRNSeed()

unique_stations = np.unique(subset_df["Start Station Id"].values)
num_stations = len(unique_stations)
total_capacity = 500
total_bikes = 399
min_cap = 12
max_cap = 32

level_sum = 0
level_sum = 0
capacity_sum = 0
Full_Error = 0
Empty_Error = 0
inital_count = 0

Stations = {}
best_total_error = float('inf')  # Set best_total_error to a high initial value
best_bike_list = {}
opt_error_list = []

w = 2

##################################################################################################################
# Initialize the stations and create a dictionary mapping station IDs to Station instances
capacity_per_station = total_capacity // num_stations
for i, station in enumerate(unique_stations):
    x_i = total_bikes // 19
    r_i = capacity_per_station
    if i == num_stations - 1:
        # Allocate the remaining capacity to the last station
        r_i = total_capacity - capacity_per_station * (num_stations - 1)
    Stations[station] = Station(station_id=station, level=x_i, capacity=r_i)
    level_sum += x_i
    for station_id, station in Stations.items():
        capacity_sum += station.capacity

StationDict = {station.id: station for station in Stations.values()}
inital_stations_dict = deepcopy(StationDict)
##################################################################################################################
# Initial Run
new_total_error = OPT_RUN_SIMULATION(StationDict, T, Calendar)

if new_total_error < best_total_error:
    best_total_error = new_total_error
    inital_stations_dict = (inital_stations_dict)

station_id = CI_start_station_id_list[-1]
end_station_id = CI_end_station_id_list[-1]
start_times = CI_start_time_list[-1]
end_times = CI_end_time_list[-1]
```

```python
start_flow_df = pd.DataFrame({"Start ID": station_id, "Start Time": start_times})
end_flow_df = pd.DataFrame({"End ID": end_station_id, "End Time": end_times})

start_flow_count = start_flow_df.pivot_table(index='Start ID', columns='Start Time', aggfunc='size', fill_value
start_flow_count = start_flow_count.loc[:, 16:]
end_flow_count = end_flow_df.pivot_table(index='End ID', columns='End Time', aggfunc='size', fill_value=0)
end_flow_count = end_flow_count.loc[:, 16:]
flow_rate = end_flow_count - start_flow_count

num_iterations = 20
num_replications = 2
CI_95_LIST = []
avg_error_2 = []
####################################################################################################################
for iterations in range(num_iterations):
    #print("Simulated Days", iterations)
    # Generate trial solution
    trial_solution = generate_trial_solution(inital_stations_dict, flow_rate, w)

    # Simulate and evaluate
    new_total_error = 0
    rep_error_list = []
    for _ in range(num_replications):
        Calendar = SimClasses.EventCalendar()
        ZSimRNG = SimRNG_Modified.InitializeRNSeed()
        rep_error = OPT_RUN_SIMULATION(trial_solution, T, Calendar)
        rep_error_list.append(rep_error)
    ######################################
        station_id = CI_start_station_id_list[-1]
        end_station_id = CI_end_station_id_list[-1]
        start_times = CI_start_time_list[-1]
        end_times = CI_end_time_list[-1]
        start_flow_df = pd.DataFrame({"Start ID": station_id, "Start Time": start_times})
        end_flow_df = pd.DataFrame({"End ID": end_station_id, "End Time": end_times})
        start_flow_count = start_flow_df.pivot_table(index='Start ID', columns='Start Time', aggfunc='size', fi
        start_flow_count = start_flow_count.loc[:, 16:]
        end_flow_count = end_flow_df.pivot_table(index='End ID', columns='End Time', aggfunc='size', fill_value
        end_flow_count = end_flow_count.loc[:, 16:]
        flow_rate = end_flow_count - start_flow_count
    ######################################

    total_error_rep = CI_95(rep_error_list)
    CI_95_LIST.append(total_error_rep)
    new_total_error = total_error_rep[0]
    avg_error_2.append(new_total_error)
    print(f"Iteration: {iterations +1} -> Total Error: {total_error_rep}")

    # If the new total error is better than the current best total error, update the inital_stations_dict
    if new_total_error < best_total_error:
        best_total_error = new_total_error
        inital_stations_dict = deepcopy(trial_solution)

print("Best Bike List")
for station_id, station in inital_stations_dict.items():
    print(f"Station {station_id} Initial Bike List: {station.Get_Bike_List()}")

print(avg_error_2)
```

In [499…

```python
plt.plot(avg_error_1, label="Heuristic 1")
plt.xlabel("Interations")
plt.ylabel("Objective Function")

plt.plot(avg_error_2, label="Heuristic 2")
plt.xlabel("Interations")
plt.legend()
plt.ylabel("Objective Function")
plt.title("Comparison of Different Inital Bike Allocations")
plt.xlabel("Iterations")
plt.ylabel("Objective Function")
plt.legend()
plt.xticks([i for i in range(0, 20)])
plt.show()
```


Comparison of Different Inital Bike Allocations
Heuristic 1
Heuristic 2
Objective Function
200
300
400
500
600
700
800
900
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
Iterations