# MIE 1613 Project Final Report

# Bike Rebalancing Optimization Simulation in Bike Sharing Problem

Yirui Chen (1005055333, yirui.chen@mail.utoronto.ca)

Yingzheng Ma (1004822258, yingzheng.ma@mail.utoronto.ca)

## 1. Introduction

This section describes the background of the simulation problem and the outline of the whole report.

### 1.1 Background Information

Bike Share Toronto is a large-scale system with approximately 360 stations, 3750 bikes and 6227 docks where customers can pick up a bike at one station and then return the bike at any other stations. The time-varying bike demand and the randomly return rule create some management problems, chief among these is the issue of system imbalance. Bicycles are clustered in certain geographical areas which leaves other areas devoid of bikes. To prevent this from happening frequently, operators move bicycles through the network to achieve reallocation. The purposes of the bike rebalancing optimization are to reduce the whole cost of the operation, get the maximum profit and minimize the waiting time of customers who cannot find a bike, or cannot find a dock to return a bike.

### 1.2 Simulation background

Intuitively, we know in hush hours, there will be more bike demand than other time. Both of which that a customer can't find a bike to use or can't find an empty dock to return bike immediately will decrease the satisfaction of the customers or let the company lose opportunities to make more money.

Nowadays, most bike share programs rebalance their bikes during the early morning, and then because of the random movement, bikes will be relocated and rebalanced naturally. But there is also manual rebalancing process during daytime. In the simulation optimization, the main idea is to explore how many bikes should be rebalanced in order to get the maximum profit in the whole system. We consider the five stations as a system and the revenue of the whole system should be the maximum.

Subsets of stations (Bay St / St. Joseph St, Union Station, College St / Major St, Queens Quay / Yonge St, Madison Ave / Bloor St W, and customers arrive at these stations to rent bikes and return bikes) are considered and we assume that the arrival and the return processes are non-stationary Poisson process.

### 1.3 Report Outline

In this report, a discrete-event simulation model was built to solve a bike rebalancing problem, which was how many bikes should be allocated to each station at the beginning of the day in order to meet the bike demand in the morning rush hours (from 7 AM to 10 AM) and get the maximum profit with the minimum average waiting time of each customer. In order to simplify this problem, 5 substations were chosen.

In the *Problem Description* section, the problem will be proposed and the definition of objective function and variable will be shown.

In the *Model Description* section, and the basic thought of the discrete-event simulation model and each function we defined using PythonSim packages are introduced.

Finally, the *Results* of the best number of initial bikes in each station based on the constrains will be given, and the limitation of the model and future work on how to improve the model to use for a realistic problem will be provided.

## 2. Problem Description

Inspiration of the optimization problem was obtained from Jian, Nanjing, et al (2016)'s research and did some simplification, but also some creation.

This section introduces the problem and its input data.

### 2.1 Problem Statement

In order to simply this operation simulation problem, we ignored the annual membership and causal customers' difference, roughly regarded the service fee is a constant value. What's more, there are a lot of cost, such as the sum of the expected stock-out cost for the shortage, the potential loss of business opportunity because the bike is in use and the rebalancing operation fee bikes, which are all treated as constant values roughly and through the whole simulation process, we calculated the gross revenue by these estimated fees.

The total number of bikes of the five stations and the docks of each station are also fixed, we try to know how many bikes should be assigned to each station in the early mornings to satisfy the demand of bikes in the rush hours and get the maximum revenue.

In our simulation optimization model, we assumed that 70 bikes are totally assigned to these 5 stations constrained by the capacity of each station. We assume that there are $x_{ij}$ $(i = 1, \dots, 6, j = 1, \dots 9)$ bikes at station $i$ in the interval $j$ before the start of the morning rush hour, which has $r_i(i = 1,2,3,4,5)$ capacity each. In the meanwhile, there is manual rebalancing process during the daytime whenever the number of the bikes in a station down hit the rebalancing threshold. In the simulation, we assume that if the number of bikes less than 2, the manual rebalancing process will be scheduled until the number reaches

5. If the number of bikes lies below 0, the new bike isn't moved to the station but a customer comes, he will be wait at most $T_{ij}$ then he leaves and the company loss the potential business opportunity. $T_{ij}$ is used to donate the waiting time of the customers at station $i$, interval $j$. $T_{ij}$ also includes the customer waits to return the bike if there is no empty docks in the station. If customer has waited too long to return the bike, refund is given. The simulation optimization problem is then to

$$\text{Maximum } Ef(x)$$

$$\text{Subject to } \sum_{i=1}^{5} x_i \leq 70$$

$$T_{ij} \leq 5$$

$$0 \leq x_i \leq r_i, \forall i$$

$$r_1 = 15, r_2 = 27, r_3 = 15, r_4 = 11, r_5 = 15$$

Where, $f(x)$ is the whole revenue of the 5-station system. We run multiply replications to get the expected maximum revenue and return the optimal initial numbers of each station.

Thinning method is used to solve the Non-stationary Poisson Process arrival and return rates of customers.

## 2.2 Data Input

Real trip data from Bike Share Toronto Ridership (2017 Q3) was used to generate the arrival and return rates. There are 92 days' ride data. The reason why we choose Q3 because Q4 and Q1 in Toronto are so cold that many people don't want to ride a bike; Q3 is better than Q2 in climate and there are many tourists. More richer dataset will lead to more clear results, but in the other side, maybe the results derived by Q3 data cannot be applied to other seasons because the climate is so special in Toronto.

In order to simplify the problem, we only choose 5 stations from 360 stations to explore and we know the number of docks of each station, which is the ceil of each station's capacity.

We split our ride data into 9 intervals and one interval only lasts 20 mins because after carefully analyzing the data, we found most people use the sharing bike for 10-30 mins, so it is reasonable to choose 20 mins.

## 3. Model Description

In this section, we will introduce our discrete-event simulation model and the logic of all the function.

The simulation of the bicycle-sharing system is programmed using PythonSim, a collection of VBA subs, functions and class modules that aid in developing discrete-event simulations.

Our model consists of the constants we have defined (Fig.1), the global variables, an initialization sub, some event routines and three functions: Rental, RideEnd and rebalancing. The global variables revenue, cost, and Loss are exactly key variables that we need to track to calculate profits. We use ten FIFOQueue objects, five of which are used to denote queues waiting for pick up a bike at five stations, and the remaining are used to denote queues waiting for return a bike at five stations. The variable names of these two types of FIFOQueue objects are “pickup_queue” and “return_queue” respectively. We use these FIFOQueue objects to hold the entities represents customers who cannot find a bike then wait, or cannot find a dock to return a bike then wait until someone (the next customer or rebalancing vehicles) rents a bike. If customers have waited too long to return the bike or pick up a bike, refund is given.

```
Single_Trip_fee = 3.75 # CAD per bike per ride
loss = 3.75 # Canadian Dollar for cost of loss of business oportunity
fuel_costs = 3 # Canadian Dollar for rebalancing vehicles carrying bikes
threshold = 2.0 #schedule the "rebalancing" once the number of bikes is less than the threshold
up_to = 5.0 # the number of bikes in the station would reach 5 after rebalancing
delay = 20 # it takes 20 minutes for bikes to be transported to the station for bike rebalancing
replications = 100 #number of simulation runs
operation_cost = 2 # Canadian Dollar per bike for operation
RunLength = 180 # 3 hours
```

**Figure 1.** Simulation Constants

The system evolves as follows. For each station, schedule the first bike rental event and the first bike

returning event to get the simulation started. After data cleaning, the renting rates can be estimated by averaging the renting counts for each twenty minutes. (Table1). The piecewise-constant renting rate function then can be used together with the thinning method to generate the NSPP. In this way, the inter-arrival time is generated to schedule the bike rental events. So does the bike return events (Table 2).

Notice that we modified Class EventNotice and function Schedule of SimFunctions.py to apply to our model. Specifically, we added **Kwargs to let the functions take an arbitrary number of keyword arguments. For instance, if the function takes the argument “station_id” and “station_id is set to 1, it will be applied to Station 1(Fig 2). Similar keyword arguments are “num_oredered”,"pickup_queue", "return_queue" and “signal”.

As a result, apart from EventType and EventTime, every event possesses some other unique characteristics. For example rebalancing events may have attributes specifying the station, the queue, feedback signal polarity, and ordered number of bikes for rebalancing. When the number of station is greater than one, it might be harder to keep track of all stations without indicating which station is being simulated.(Fig.3)

```
def Schedule(calendar,EventType, EventTime, **kwargs):
    #Schedule future events of EventType to occur at time SimClasses.Clock + EventTime

    addedEvent = SimClasses.EventNotice()
    addedEvent.EventType = EventType
    addedEvent.EventTime = SimClasses.Clock + EventTime
    addedEvent.kwargs = kwargs
    # print("SimClasses.Clock is %f" % SimClasses.Clock)
    # print(EventTime)
    calendar.Schedule(addedEvent)
```

**Figure 2.** Modified Schedule Function of SimFunctions.py

```
SimFunctions.Schedule(Calendar,"rebalancing",delivery_delay,stationID=station_id,
                      pickup_queue = TheQueues[station_id],dropoff_queue = TheQueues[station_id+5],
                      num_ordered=quantity[station_id], signal= -1)
```

**Figure 3.** Characteristics of Rebalancing Event

If the type of the next event is “Rent a bike”, Rental function will be called to record the changes of the

value of the variable of interest. In function Rental, first we use function NSPP to generate the inter-arrival time to schedule the next bike rental event for the station. Then we need to check if there are people waiting to put the bikes back (e.g., the Return_queue.NumQueue()). If the customer has waited too long to return the bike, they will receive a full refund and the global variable Loss increases by $3.75. We use the object from DTStat to record the waiting time. If no one is waiting in the queue, check if there are bikes available. If there are idle bikes, the customer pays to rent a bike and the revenue increases by $3.75. If not, the customer begins waiting for bikes to become idle so we need to derive a new object from the Entity to denote the customer and add it to "Pickup_queue". Once the number of bikes is less than 2, we need to schedule rebalancing operation event to carry bikes from the center to the station. We assume there is 20 mins rebalancing delay due to the distance traffic condition, so we defined the constant variable "delay" which equals to 20 to represent the EventTime of the rebalancing event.

The RideEnd function illustrates a customer returns the bike to the dock. First, the function NSPP is used to generate inter-arrival times to schedule the next returning. When the customer ends the ride and arrives at a station, we need to check if someone has been waiting to rent a bike due to no idle bikes at the station or if there is an empty rack for the customer to return. Customers leave after waiting 5 minutes. Therefore, we need to calculate waiting time and check whether the customer has left due to waiting too long. If so, we will lose a customer and the cost will increase due to loss of business opportunity, the customer whose waiting time is less than 5 minutes will get the bike. If no one is waiting to pick up the bike, check if there are empty racks to keep the bike. Derive a new object from the Entity and add it to the "Return_queue" if no empty racks. Once the number of people waiting in the queue to return bikes is more than 5, schedule rebalancing operation event to carry bikes from the station.

The rebalancing policies are actually quite simple. The model dynamically redistributes bikes across stations in order to avoid them becoming overly full or empty. We assume there is a center that controls all the repositioning vehicles. When there is no empty dock, the station will send a positive signal to the center and then a vehicle is sent to carry bikes back to the center. We implement this by setting "signal" to +1 when scheduling a "rebalancing" event. Similarly, when there is no bike, the station will send a negative signal to the center. Every time when the rebalancing event is scheduled, the cost is the number of redistributed bikes multiplied by the operation cost and fuel costs. When the rebalancing vehicle reaches the station, we also need to determine if someone is waiting in queue and serve them first.

# 4. Results

## 4.1 Results of the NSPP

After averaging the counts the 92 days' morning rush hours riding data, the arrival and return rates of each station and each interval are generated.

**Table 1.** Customer arrival rates

| | Bay St / St. Joseph St | Union Station | College St / Major St | Queens Quay / Yonge St | Madison Ave / Bloor St W |
|---|---|---|---|---|---|
| **7:00-7:20** | 0.336957 | 1.869565 | 0.250000 | 0.173913 | 0.152174 |
| **7:20-7:40** | 0.945652 | 4.967391 | 0.608696 | 0.206522 | 0.184783 |
| **7:40-8:00** | 1.271739 | 7.923913 | 0.304348 | 0.282609 | 0.282609 |
| **8:20-8:40** | 1.652174 | 6.086957 | 0.521739 | 0.945652 | 0.695652 |
| **8:40-9:00** | 1.630435 | 4.717391 | 1.184783 | 1.141304 | 1.239130 |
| **9:00-9:20** | 1.173913 | 3.478261 | 0.923913 | 0.847826 | 1.163043 |
| **9:20-9:40** | 1.184783 | 3.217391 | 0.913043 | 0.597826 | 0.532609 |
| **9:40-10:00** | 0.728261 | 1.706522 | 0.597826 | 1.010870 | 0.500000 |
| **MaxRentingRate** | 1.652174 | 7.923913 | 1.184783 | 1.141304 | 1.239130 |

**Table 2.** Customer return bikes rates

| | Bay St / St. Joseph St | Union Station | College St / Major St | Queens Quay / Yonge St | Madison Ave / Bloor St W |
|---|---|---|---|---|---|
| **7:00-7:20** | 0.054348 | 1.467391 | 0.043478 | 0.054348 | 0.097826 |
| **7:20-7:40** | 0.293478 | 1.739130 | 0.086957 | 0.217391 | 0.097826 |
| **7:40-8:00** | 0.304348 | 2.413043 | 0.141304 | 0.413043 | 0.250000 |
| **8:20-8:40** | 0.826087 | 4.206522 | 0.413043 | 1.032609 | 0.195652 |
| **8:40-9:00** | 0.880435 | 3.869565 | 1.065217 | 1.923913 | 0.641304 |
| **9:00-9:20** | 0.902174 | 4.141304 | 0.760870 | 1.000000 | 0.195652 |
| **9:20-9:40** | 0.489130 | 3.021739 | 0.619565 | 0.771739 | 0.184783 |
| **9:40-10:00** | 0.380435 | 1.652174 | 0.815217 | 0.858696 | 0.206522 |
| **MaxReturnRate** | 0.902174 | 4.206522 | 1.065217 | 1.923913 | 0.641304 |

From the arrival and return rates, we can see the rates increase from 7 AM to 9:20 AM, and then decrease in most stations, which consistent with intuition. And the Union Station is the busiest station in our system. The piecewise-constant arrival rate function then can be used together with the thinning method to generate the NSPP of each station.

## 4.2 Results of the best initial number of bikes

We assume that the initial number should be from 5 to capacity number of bikes in each station, which is 480 trials total in our system. We use for-in statement to generate all trial solutions and apply our model to each trial solution. We make 100 simulation runs of 180 time units and use two-dimensional arrays to record the result of each trial solution of each simulation run. Balance[k][j] and Waittime[k][j] refer to the result of the $j^{th}$ trial solution in the $k^{th}$ simulation run. Then the average of the results of 100 simulation runs is calculated for each trial solution to obtain the expected balance of each trial solution.

**Table 3.** The best initial number of the system

| | Bay St / St. Joseph St | Union Station | College St / Major St | Queens Quay / Yonge St | Madison Ave / Bloor St W |
|---|---|---|---|---|---|
| **MaxBalance** | 13 | 25 | 9 | 10 | 13 |

The result means that under all constrains, if the total number of bikes is 70, the way the total number of bikes assigned to each station in the early morning can get the maximum revenue. From the results, we can see the initial numbers of all stations are near the capacity, which means it is better to place enough bikes to satisfy the demand of morning rush hours. But considering the budge of bike numbers and cost, the initial numbers of bikes don't reach the capacity. What's more, the busier station should be assigned more bikes, such as Union Station in our simulation example.

### 4.3 Future Work

The problem is a simplified system, there is only 5 stations and the system isn't a closed system, the different service fee between the annual membership and casual customer isn't taken into consideration, the rebalancing costs are estimated and only morning rush hours have been considered…All of these shortcomings of the simulation makes it far away from a complete research. But the logic of our simulation is a foundation of the further optimization.

In the future, we will consider all the stations and bikes of Bike Share Toronto. In such case, it is a complete and closed system. The flow of bikes will be within the system and we will clearly know every bike's movement. Based on the whole system, we can get the arrival rates of all stations in a whole day. And we can generate the possibility of the bike movement in each station, which can be used in the new simulation optimization. What's more, we can also know where the rebalanced bikes come from and the rebalancing distance, which is the main cost of the rebalancing process. We should know the number of annual memberships and casual and calculate the proportion of them in each ride, which can be used to calculate the precise service fee of each ride in simulation.

## REFERENCES


Jian, N., Freund, D., Wiberg, H. M., & Henderson, S. G. (2016, December). Simulation optimization for a large-scale bike-sharing system. In Proceedings of the 2016 Winter Simulation Conference (pp. 602-613). IEEE Press.

Jian, N., & Henderson, S. G. (2015, December). An introduction to simulation optimization. In 2015 Winter Simulation Conference (WSC) (pp. 1780-1794). IEEE.

Maggioni, F., Cagnolari, M., Bertazzi, L., & Wallace, S. W. (2019). Stochastic optimization models for a bike-sharing problem with transshipment. European Journal of Operational Research, 276(1), 272-283.

Liu, J., Sun, L., Chen, W., & Xiong, H. (2016, August). Rebalancing bike sharing systems: A multi-source data smart optimization. In Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (pp. 1005-1014). ACM.

Ciancio, C., Ambrogio, G., & Laganá, D. (2017, September). A Stochastic Maximal Covering Formulation for a Bike Sharing System. In International Conference on Optimization and Decision Science (pp. 257-265). Springer, Cham.

O'Mahony, E., & Shmoys, D. B. (2015, February). Data analysis and optimization for (citi) bike sharing. In Twenty-ninth AAAI conference on artificial intelligence.

Zhou, Y., Wang, L., Zhong, R., & Tan, Y. (2018). A Markov Chain Based Demand Prediction Model for Stations in Bike Sharing Systems. Mathematical Problems in Engineering, 2018.

# APPENDIX

# Data Cleaning

In [18]:

```
import csv
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from datetime import datetime
```

## Load the Dataset

In [13]:

```
bike_df = pd.read_csv("Bikeshare_Ridership_2017_Q3.csv")
```

In [14]:

```
bike_df.head()
```

Out[14]:

| | trip_id | trip_start_time | trip_stop_time | trip_duration_seconds | from_station_name | to_statio |
|---|---|---|---|---|---|---|
| **0** | 1253914 | 7/1/2017 0:00 | 7/1/2017 0:15 | 910 | Princess St / Adelaide St E | 424 Wellii |
| **1** | 1253915 | 7/1/2017 0:01 | 7/1/2017 0:15 | 837 | Fort York Blvd / Capreol Crt | H (Queens ( |
| **2** | 1253916 | 7/1/2017 0:01 | 7/1/2017 0:14 | 786 | Fort York Blvd / Capreol Crt | H (Queens ( |
| **3** | 1253917 | 7/1/2017 0:01 | 7/1/2017 0:25 | 1420 | Elizabeth St / Edward St (Bus Terminal) | Bost Qu |
| **4** | 1253918 | 7/1/2017 0:01 | 7/1/2017 0:25 | 1437 | Elizabeth St / Edward St (Bus Terminal) | Bost Qu |

In [19]:

```
# Transfer format.
bike_df['trip_start_time']=pd.to_datetime(bike_df['trip_start_time'])
bike_df['trip_stop_time']=pd.to_datetime(bike_df['trip_stop_time'])
```

In [31]:

```
# Split the month, day, hour and minute to separate columns
bike_df['start_hour']=bike_df['trip_start_time'].apply(lambda x: x.hour)
bike_df['start_minute']=bike_df['trip_start_time'].apply(lambda x: x.minute)
bike_df['start_month']=bike_df['trip_start_time'].apply(lambda x: x.month)
bike_df['start_day']=bike_df['trip_start_time'].apply(lambda x: x.day)

bike_df['stop_hour']=bike_df['trip_stop_time'].apply(lambda x: x.hour)
bike_df['stop_minute']=bike_df['trip_stop_time'].apply(lambda x: x.minute)
bike_df['stop_month']=bike_df['trip_stop_time'].apply(lambda x: x.month)
bike_df['stop_day']=bike_df['trip_stop_time'].apply(lambda x: x.day)
```

# Calculate Arrival Rate

In [194]:

```
# Take out morning rush hours data
morning_df = bike_df[(bike_df['start_hour'] >= 7)&(bike_df['start_hour'] < 10)]
return_df = bike_df[(bike_df['stop_hour'] >= 7)&(bike_df['stop_hour'] < 10)]
```

In [139]:

```
morning_df['Value'] = np.nan
```

```
/anaconda2/envs/python3/lib/python3.7/site-packages/ipykernel_launch
er.py:1: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: http://pandas.pydata.org/panda
s-docs/stable/indexing.html#indexing-view-versus-copy
  """Entry point for launching an IPython kernel.
```

In [147]:

```
for i,row in morning_df.iterrows():
    if (row['start_minute'] >= 0) and (row['start_minute'] < 20):
        morning_df.at[i, 'Value'] = 1
    elif (row['start_minute'] >= 20) and (row['start_minute'] < 40):
        morning_df.at[i, 'Value'] = 2
    else:
        morning_df.at[i, 'Value'] = 3
```

In [189]:

```
# Define a function to count the arrive rate of a station of all intervals.
def start_station(station_name):
    df = morning_df[morning_df['from_station_name'] == station_name]
    c = ['7:00-7:20', '7:20-7:40', '7:40-8:00', '8:00-8:20', '8:20-8:40', '8:40-
9:00',
                                    '9:00-9:20', '9:20-9:40', '9:40-10:00']
    count = pd.DataFrame(columns = c)
    gross = []
    for h in range(7,10,1):
        for i in range(1,4,1):
            counts = []
            for m in range(7,10,1):
                for d in range(1,32,1):
                    counts.append(len(df[(df['start_month'] == m) & (df['start_d
ay'] == d) &
                                         (df['start_hour']== h) & (df['Value'] ==
i)]))
            gross.append(counts)

    for j in range(0,9,1):
        count[c[j]] = gross[j]

    count = count.drop(count.index[len(count)-1])
    count.loc['Mean'] = count.mean()

    return count
```

In [191]:

```
Bay_St = start_station('Bay St / St. Joseph St');
Union = start_station('Union Station');
College_St = start_station('College St / Major St');
Queens = start_station('Queens Quay / Yonge St');
Madison = start_station('Madison Ave / Bloor St W');
```

In [193]:

```
# Write all the data into an Excel file.
with pd.ExcelWriter('BikeRentCount.xlsx') as writer:
    Bay_St.to_excel(writer, sheet_name = 'Bay_St')
    Union.to_excel(writer, sheet_name = 'Union')
    College_St.to_excel(writer, sheet_name = 'College_St')
    Queens.to_excel(writer, sheet_name = 'Queens')
    Madison.to_excel(writer, sheet_name = 'Madison')
```

# Calculate Return Rate

In [195]:

```
return_df['Value'] = np.nan
for i,row in return_df.iterrows():
    if (row['stop_minute'] >= 0) and (row['stop_minute'] < 20):
        return_df.at[i, 'Value'] = 1
    elif (row['stop_minute'] >= 20) and (row['stop_minute'] < 40):
        return_df.at[i, 'Value'] = 2
    else:
        return_df.at[i, 'Value'] = 3
```

```
/anaconda2/envs/python3/lib/python3.7/site-packages/ipykernel_launch
er.py:1: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: http://pandas.pydata.org/panda
s-docs/stable/indexing.html#indexing-view-versus-copy
  """Entry point for launching an IPython kernel.
```

In [196]:

```
# Define a function to count the return rate of chosen stations.
def return_station(station_name):
    df = return_df[return_df['to_station_name'] == station_name]
    c = ['7:00-7:20', '7:20-7:40', '7:40-8:00', '8:00-8:20', '8:20-8:40', '8:40-
9:00',
                                     '9:00-9:20', '9:20-9:40', '9:40-10:00']
    count = pd.DataFrame(columns = c)
    gross = []
    for h in range(7,10,1):
        for i in range(1,4,1):
            counts = []
            for m in range(7,10,1):
                for d in range(1,32,1):
                    counts.append(len(df[(df['stop_month'] == m) & (df['stop_da
y'] == d) &
                                         (df['stop_hour']== h) & (df['Value'] ==
i)]))
            gross.append(counts)

    for j in range(0,9,1):
        count[c[j]] = gross[j]

    count = count.drop(count.index[len(count)-1])
    count.loc['Mean'] = count.mean()

    return count
```

In [197]:

```
Bay_St_r = return_station('Bay St / St. Joseph St');
Union_r = return_station('Union Station');
College_St_r = return_station('College St / Major St');
Queens_r = return_station('Queens Quay / Yonge St');
Madison_r = return_station('Madison Ave / Bloor St W');
```

In [198]:

```
with pd.ExcelWriter('BikeReturnCount.xlsx') as writer:
    Bay_St_r.to_excel(writer, sheet_name = 'Bay_St')
    Union_r.to_excel(writer, sheet_name = 'Union')
    College_St_r.to_excel(writer, sheet_name = 'College_St')
    Queens_r.to_excel(writer, sheet_name = 'Queens')
    Madison_r.to_excel(writer, sheet_name = 'Madison')
```

# Bike Rebalancing Optimization Simulation

In [3]:

```
import SimFunctions
import SimRNG
import SimClasses
import numpy as np
import pandas as pd
#import matplotlib.pyplot as plt
import scipy.stats as stats
import xlrd
```

In [4]:

```
# Read the data file we got through data cleaning.
wb = xlrd.open_workbook('BikeRentCount.xlsx')
# The file consists of five sheets and each sheet contains data for one station.
sheets = wb.sheet_names()
MaxRentingRate=[]
RentingRate=[]
for i in range(len(sheets)):
    data = pd.read_excel('BikeRentCount.xlsx', sheet_name=sheets[i])
    #Calculate the average number of rented bikes for each twenty minutes of eac
h station.
    rentingRates = data.mean()
    RentingRate.append(rentingRates)
    # Find the maximum of these rates that would be used for function NSPP
    maxRentingRate = max(rentingRates)
    MaxRentingRate.append(maxRentingRate)
```

In [5]:

```
c = ["Bay St / St. Joseph St", "Union Station", "College St / Major St", "Queens
Quay / Yonge St", "Madison Ave / Bloor St W"]
Rent = pd.DataFrame(columns = c, index = ["7:00-7:20", "7:20-7:40", "7:40-8:00",
"8:20-8:40", "8:40-9:00",
                                          "9:00-9:20", "9:20-9:40", "9:40-10:00"
, "MaxRentingRate"])
```

In [6]:

```
for i in range(0,5,1):
    Rent[c[i]] = RentingRate[i]
Rent.loc['MaxRentingRate'] = MaxRentingRate
Rent
```

Out[6]:

| | Bay St / St. Joseph St | Union Station | College St / Major St | Queens Quay / Yonge St | Madison Ave / Bloor St W |
|---|---|---|---|---|---|
| **7:00-7:20** | 0.336957 | 1.869565 | 0.250000 | 0.173913 | 0.152174 |
| **7:20-7:40** | 0.945652 | 4.967391 | 0.608696 | 0.206522 | 0.184783 |
| **7:40-8:00** | 1.271739 | 7.923913 | 0.304348 | 0.282609 | 0.282609 |
| **8:20-8:40** | 1.652174 | 6.086957 | 0.521739 | 0.945652 | 0.695652 |
| **8:40-9:00** | 1.630435 | 4.717391 | 1.184783 | 1.141304 | 1.239130 |
| **9:00-9:20** | 1.173913 | 3.478261 | 0.923913 | 0.847826 | 1.163043 |
| **9:20-9:40** | 1.184783 | 3.217391 | 0.913043 | 0.597826 | 0.532609 |
| **9:40-10:00** | 0.728261 | 1.706522 | 0.597826 | 1.010870 | 0.500000 |
| **MaxRentingRate** | 1.652174 | 7.923913 | 1.184783 | 1.141304 | 1.239130 |

In [7]:

```
# Similar data processing for 'BikeReturnCount.xlsx'
wb = xlrd.open_workbook('BikeReturnCount.xlsx')
sheets = wb.sheet_names()
MaxReturnRate=[]
ReturnRate=[]
for i in range(len(sheets)):
    data = pd.read_excel('BikeReturnCount.xlsx', sheet_name=sheets[i])
    returnRates = data.mean()
    ReturnRate.append(returnRates)
    maxReturnRate = max(returnRates)
    MaxReturnRate.append(maxReturnRate)
```

In [8]:

```
Return = pd.DataFrame(columns = c, index = ["7:00-7:20", "7:20-7:40", "7:40-8:0
0", "8:20-8:40", "8:40-9:00",
                                            "9:00-9:20", "9:20-9:40", "9:40-10:0
0", "MaxReturnRate"])
for i in range(0,5,1):
    Return[c[i]] = ReturnRate[i]
Return.loc['MaxReturnRate'] = MaxReturnRate
Return
```

Out[8]:

| | Bay St / St. Joseph St | Union Station | College St / Major St | Queens Quay / Yonge St | Madison Ave / Bloor St W |
|---|---|---|---|---|---|
| **7:00-7:20** | 0.054348 | 1.467391 | 0.043478 | 0.054348 | 0.097826 |
| **7:20-7:40** | 0.293478 | 1.739130 | 0.086957 | 0.217391 | 0.097826 |
| **7:40-8:00** | 0.304348 | 2.413043 | 0.141304 | 0.413043 | 0.250000 |
| **8:20-8:40** | 0.826087 | 4.206522 | 0.413043 | 1.032609 | 0.195652 |
| **8:40-9:00** | 0.880435 | 3.869565 | 1.065217 | 1.923913 | 0.641304 |
| **9:00-9:20** | 0.902174 | 4.141304 | 0.760870 | 1.000000 | 0.195652 |
| **9:20-9:40** | 0.489130 | 3.021739 | 0.619565 | 0.771739 | 0.184783 |
| **9:40-10:00** | 0.380435 | 1.652174 | 0.815217 | 0.858696 | 0.206522 |
| **MaxReturnRate** | 0.902174 | 4.206522 | 1.065217 | 1.923913 | 0.641304 |

# Simulation constants

In [25]:

```
Single_Trip_fee = 3.75 # CAD per bike per ride
loss = 3.75 # Canadian Dollar for cost of loss of business oportunity
fuel_costs = 3 # Canadian Dollar for rebalancing vehicles carrying bikes
threshold = 2.0 #schedule the "rebalancing" once the number of bikes is less tha
n the threshold
up_to = 5.0 # the number of bikes in the station would reach 5 after rebalancing
delay = 20 # it takes 20 minutes for bikes to be transported to the station for
 bike rebalancing
replications = 100 #number of simulation runs
operation_cost = 2 # Canadian Dollar per bike for operation
RunLength = 180 # 3 hours
```

# Main Simulation Process

In [10]:

```
def Rental(**kwargs):
    global revenue, Loss
    #Check which station the rental event occurs in
    station_id = kwargs['stationID']
    Pickup_queue = kwargs["pickup_queue"]
    Return_queue = kwargs["return_queue"]
    #Use NSPP to schedule the next bike rental event for current station
    interarrival = NSPP(station_id,MaxRentingRate,RentingRate)
    SimFunctions.Schedule(Calendar,"Rent a bike",interarrival,stationID = statio
n_id,
                          pickup_queue = TheQueues[station_id],return_queue = Th
eQueues[station_id+5])

    # Checks if there are people waiting to put the bikes back.
    if Return_queue.NumQueue() > 0:
        DepartingCustomer = Return_queue.Remove()
        Waiting_time = SimClasses.Clock - DepartingCustomer.CreateTime
        Wait.Record(SimClasses.Clock - DepartingCustomer.CreateTime)
        # If customer has waited too long to return the bike, refund is given.
        # The customer gets the bike from the people waiting to put the bikes ba
ck.
        if Waiting_time >5:
            Loss += Single_Trip_fee

    #If no one is waiting to put the bikes back,
    #check if there are bikes available and update number of bikes for the stati
on.
    else:
        if Num_bikes[station_id]>0:
            Num_bikes[station_id]-=1
            # Customer pays to rent bike.
            revenue += Single_Trip_fee
        #No bikes available so the customer begins waiting for bikes to become a
vailable.
        else:
            Customer = SimClasses.Entity()
            Pickup_queue.Add(Customer)

    #Schedule the bike rebalancing event once the number of bikes is less than t
he threshold
    """
    There are two conditions to be met for the rebalancing operation. Schedule t
he rebalancing operation
    once the number of bikes is less than the threshold and the previous rebalan
ce operation has ended.
    """
    if Num_bikes[station_id] <= threshold and quantity[station_id] == 0:
        quantity[station_id] = up_to - Num_bikes[station_id]
        SimFunctions.Schedule(Calendar,"rebalancing",delay,stationID=station_id,
                              pickup_queue = TheQueues[station_id],return_queue
= TheQueues[station_id+5],
                              num_ordered=quantity[station_id], signal= -1)
```

In [11]:

```
def RideEnd(**kwargs):
    global  revenue, Loss
    station_id = kwargs['stationID']
    Return_queue = kwargs['return_queue']
    Pickup_queue = kwargs['pickup_queue']
     ##Use NSPP to schedule the next end of ride for current station
    interarrival = NSPP(station_id,MaxReturnRate,ReturnRate)
    SimFunctions.Schedule(Calendar,"Return a bike",interarrival,stationID = stat
ion_id,
                          pickup_queue = TheQueues[station_id],return_queue = Th
eQueues[station_id+5])

    # We assume not every customer will wait for a bike and eventually take a bi
ke.Customers leave after 5 mins.
    while(True):
        # Check if customers are waiting.
        if Pickup_queue.NumQueue() == 0:
            #If no one is waiting to pick up the bike,check if there are empty r
acks to keep the bike.
            if Num_bikes[station_id] < Num_docks[station_id]:
                # Customer returns the bike to the rack.
                Num_bikes[station_id] += 1
            # No empty racks. The customer begins waiting in queue.
            else:
                Customer = SimClasses.Entity()
                Return_queue.Add(Customer)
            break
        #If there are customers waiting to pick up the bike,
        #check if the customer has left due to waiting too long to pick up the b
ike.
        DepartingCustomer = Pickup_queue.Remove()
        Waiting_time = SimClasses.Clock - DepartingCustomer.CreateTime
        Wait.Record(SimClasses.Clock - DepartingCustomer.CreateTime)
        if Waiting_time < 5:
            # Next waiting customer gets a bike from the customer who has finish
ed the ride.
            # Break the while loop.
            break
        else:
            # We lose a customer because the customer has waited too long
            Loss += loss

    if Return_queue.NumQueue() >= threshold and quantity[station_id] == 0:
        quantity[station_id] = Return_queue.NumQueue()
        SimFunctions.Schedule(Calendar,"rebalancing",delay,stationID=station_id,
                              pickup_queue = TheQueues[station_id],return_queue
= TheQueues[station_id+5],
                              num_ordered=quantity[station_id], signal= 1)
```

In [12]:

```
def rebalancing(**kwargs):
    global revenue ,cost, Loss
    station_id = kwargs['stationID']
    Return_queue = kwargs['return_queue']
    Pickup_queue = kwargs['pickup_queue']
    #The number of bikes needed for bike rebalancing
    Num_ordered = kwargs['num_ordered']
    Signal = kwargs['signal']
    #Cost due to bike rebalanccing
    cost += (Num_ordered * operation_cost) + fuel_costs
    # Signal is -1 means the station is almost out of bikes for customers rentin
g bikes,
    # then the center will carry bikes to the station
    if Signal == -1:
        #The number of bikes changes after the rebalancing operation
        Num_bikes[station_id] += Num_ordered
        # We assume not every customer will wait for a bike and eventually take
 a bike..Customers leave after 5 mins.
        while(True):
            # Check whether customers are waiting in the pick_up queue and wheth
er the ordered number of bikes is in short supply.
            if (Pickup_queue.NumQueue() == 0.0) | (Num_bikes[station_id] == 0.0
):
                break

            DepartingCustomer = Pickup_queue.Remove()
            Waiting_time = SimClasses.Clock - DepartingCustomer.CreateTime
            Wait.Record(SimClasses.Clock - DepartingCustomer.CreateTime)
            if Waiting_time < 5:
                # Next waiting customer gets a bike
                 Num_bikes[station_id] -= 1
            else:
                # We lose a customer
                Loss += loss
    # Signal is +1 means the station is out of racks for customers returning bik
es,
    # then the center will carry bikes back to the center
    else:
        while(True):
            # Check if customers are waiting in the Return queue
            if (Return_queue.NumQueue() == 0.0)| (Num_ordered==0.0):
                break

            DepartingCustomer = Return_queue.Remove()
            Num_ordered-=1
            Waiting_time = SimClasses.Clock - DepartingCustomer.CreateTime
            Wait.Record(SimClasses.Clock - DepartingCustomer.CreateTime)
            if Waiting_time > 5:
                # If customer has waited too long to return the bike, refund is
 given.
                 Loss += Single_Trip_fee
        Num_bikes[station_id] -= Num_ordered
    #Set the num_oredered to zero so the current rebalancing operation is over.
    quantity[station_id] = 0
```

In [13]:

```
# Prevent index out of range
def PW_ArrRate(p,t,Rates):
    hour = int(t/20)
    if hour <= 8:
        return Rates[p][hour]
    else:
        return Rates[p][-1]
```

In [14]:

```
def NSPP(q,MaxRate,Rate):
    PossibleArrival = SimClasses.Clock + SimRNG.Expon(1/(MaxRate[q]/20), 1)
    while SimRNG.Uniform(0, 1, 1) >= PW_ArrRate(q,PossibleArrival,Rate)/(MaxRate
[q]):
        PossibleArrival = PossibleArrival + SimRNG.Expon(1/(MaxRate[q]/20), 1)
    nspp = PossibleArrival - SimClasses.Clock
    return nspp
```

In [15]:

```
# Get all trial solution
trial_solution = []
for a in range(5,15):
    for b in range(5,27):
        for c in range(5,11):
            for d in range(5,15):
                for e in range(5,15):
                    # Initial number of bikes available in the system.
                    if (a+b+c+d+e)==70:
                        trial_solution.append([a,b,c,d,e])
len(trial_solution)
```

Out[15]:

```
480
```

In [26]:

```
Calendar = SimClasses.EventCalendar()
Wait = SimClasses.DTStat()

Pickup_Queue1 = SimClasses.FIFOQueue()
Pickup_Queue2 = SimClasses.FIFOQueue()
Pickup_Queue3 = SimClasses.FIFOQueue()
Pickup_Queue4 = SimClasses.FIFOQueue()
Pickup_Queue5 = SimClasses.FIFOQueue()

Return_Queue1 = SimClasses.FIFOQueue()
Return_Queue2 = SimClasses.FIFOQueue()
Return_Queue3 = SimClasses.FIFOQueue()
Return_Queue4 = SimClasses.FIFOQueue()
Return_Queue5 = SimClasses.FIFOQueue()

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

TheQueues.append(Pickup_Queue1)
TheQueues.append(Pickup_Queue2)
TheQueues.append(Pickup_Queue3)
TheQueues.append(Pickup_Queue4)
TheQueues.append(Pickup_Queue5)
TheQueues.append(Return_Queue1)
TheQueues.append(Return_Queue2)
TheQueues.append(Return_Queue3)
TheQueues.append(Return_Queue4)
TheQueues.append(Return_Queue5)

TheDTStats.append(Wait)

AllWaitMean = np.zeros((replications,len(trial_solution)))

ZSimRNG = SimRNG.InitializeRNSeed()

#Create an 2d array to record the balance of each trial solution in each simulat
ion run
output = np.zeros((replications,len(trial_solution)))
for k in range(replications):
    for j in range(len(trial_solution)):
        Num_bikes=[]
        initial_Num_bikes=[]
        #Apply the our model to each trial solution
        for i in range(0,5,1):
            Num_bikes.append(trial_solution[j][i])
            initial_Num_bikes.append(trial_solution[j][i])
        # Number of bike racks (total) at each station. This is the maximum park
ing
        # capacity for every station.
        Num_docks=[15,27,11,15,15]
        #initial number of ordered bikes for rebalancing
        quantity=[0,0,0,0,0]
        cost = 0
        Loss = 0
        revenue = 0

        SimClasses.Clock = 0.0
```

```
        SimFunctions.SimFunctionsInit(Calendar,TheQueues,TheCTStats,TheDTStats,T
heResources)
        for i in range(0,5,1):
            #Use NSPP to schedule the first arrival for each station.
            #Initialize queues at each station.
            SimFunctions.Schedule(Calendar,"Rent a bike",NSPP(i,MaxRentingRate,R
entingRate),stationID = i,
                                  pickup_queue = TheQueues[i],return_queue = The
Queues[i+5])
            SimFunctions.Schedule(Calendar,"Return a bike",NSPP(i,MaxReturnRate,
ReturnRate),stationID = i,
                                  pickup_queue = TheQueues[i],return_queue = The
Queues[i+5])
        SimFunctions.Schedule(Calendar,"EndSimulation",RunLength)
        NextEvent = Calendar.Remove()
        SimClasses.Clock = NextEvent.EventTime
        if NextEvent.EventType == "Rent a bike":
            Rental(**NextEvent.kwargs)
        elif NextEvent.EventType == "Return a bike":
            RideEnd(**NextEvent.kwargs)
        elif NextEvent.EventType == "rebalancing":
            rebalancing(**NextEvent.kwargs)

        while NextEvent.EventType != "EndSimulation":
            NextEvent = Calendar.Remove()
            SimClasses.Clock = NextEvent.EventTime
            if NextEvent.EventType == "Rent a bike":
                Rental(**NextEvent.kwargs)
            elif NextEvent.EventType == "Return a bike":
                RideEnd(**NextEvent.kwargs)
            elif NextEvent.EventType == "rebalancing":
                rebalancing(**NextEvent.kwargs)
        #cost for repositioning bike overnight
        num_repositioning_bikes= (abs(np.array(Num_bikes)-np.array(initial_Num_b
ikes)).sum())
        cost += (num_repositioning_bikes * operation_cost) + fuel_costs
        balance = revenue - cost - Loss
        output[k][j]= float(balance)
        AllWaitMean[k][j] = Wait.Mean()
```

In [27]:

```
output
```

Out[27]:

```
array([[ 52.5 , 181.5 , 119.25, ..., 189.5 , 177.  , 195.25],
       [ 71.75, 222.25, 169.75, ..., 167.  , 157.75, 192.25],
       [201.75, 190.  , 122.  , ..., 178.25, 176.75, 119.25],
       ...,
       [ 76.25, 154.75, 172.25, ...,  99.5 , 184.5 , 171.  ],
       [163.75, 121.25, 190.5 , ..., 148.5 , 165.25, 115.75],
       [171.  , 109.25,  92.25, ..., 183.5 , 116.75, 161.25]])
```

In [28]:

```
AllWaitMean
```

Out[28]:

```
array([[ 8.95683223,  0.        , 10.91489604, ...,  0.        ,
        12.16744216,  0.70671506],
       [21.523585  ,  0.        ,  2.52155378, ..., 14.21458231,
         4.97163335,  0.77443413],
       [ 3.15974946,  0.        , 11.96479682, ..., 10.2703596 ,
         2.81437474, 10.15607346],
       ...,
       [21.4242369 ,  3.08941932, 22.5869654 , ..., 15.02635598,
         0.        , 11.56002051],
       [ 7.4305721 , 15.62797159, 18.16469207, ..., 16.14978257,
        18.31180626, 10.99547289],
       [23.65810476, 11.2495989 , 17.34076682, ..., 28.93936329,
         9.42145589,  5.39122265]])
```

In [36]:

```
#Get the the expectation by calculating the mean over all simulation runs for ea
ch trial solution
Estimated_Expected_waittime=np.mean(AllWaitMean,axis=0)
#Get the the expectation by calculating the mean over all simulation runs for ea
ch trial solution
Estimated_Expected_balance=np.mean(output,axis=0)
```

In [37]:

```
# get the maximum profit subject to a waiting time constraint
i=Estimated_Expected_balance[0]
for x,y in zip(Estimated_Expected_balance, Estimated_Expected_waittime):
    # control waiting time within 5mins
    if (x>=i)&(y<=5):
        i=x
np.argwhere(Estimated_Expected_balance==i)
```

Out[37]:

```
array([[280]], dtype=int64)
```

In [38]:

```
MaxBalance = trial_solution[280]
```

In [39]:

```
result = pd.DataFrame(index = ["MaxBalance"], columns = ["Bay St / St. Joseph S
t", "Union Station",
                                 "College St / Major St", "Queens Quay / Yonge S
t", "Madison Ave / Bloor St W"])
#result['MinWaitingTime'] = MinWaitingTime
result.loc['MaxBalance'] = MaxBalance
result
```

Out[39]:

| | Bay St / St. Joseph St | Union Station | College St / Major St | Queens Quay / Yonge St | Madison Ave / Bloor St W |
|---|---|---|---|---|---|
| **MaxBalance** | 13 | 25 | 9 | 10 | 13 |