Columbia University

IEOR E4418 Transportation Analytics & Logistics

Professor Adam Elmachtoub

Final Project

An in-depth analysis of a repair method for Citi Bike

Emily Labattaglia (eel2141)
Lars Patrik Roeller (lr2801)
Konstantinos Valaris (kv2275)

Fall 2016

Contents

1 Introduction

1.1 The Topic

In 2016 alone, the Citi Bike program had an average fleet of 7,390 bikes and averaged 38,991 rides per day. Like cars, trucks, and trains, there are malfunctions in the mechanics. Bikes break down. When a bike breaks down, however, there are a number of questions that arise in the process getting the bike off and back on the road. If the bike needs to be taken out of its dock, where are they taken to? How are these bikes retrieved? These questions are important in the bike repair process.

1.2 Current Methods of Repair

We've gathered information by reaching out to various Citi Bike representatives. There are two ways to find out if a bike is broken. There are Field Bike Mechanics who check every single bike in the field at least once a month. If they encounter an issue, then they try to fix it out in the field. These types of repairs include pumping air into the tires, tightening the basket, and brake related issues. If they assess the bike and decide it needs a more thorough tune up that is unable to be done in the field, they "red light" the bike in the dock.

To "red light" a bike means to press the wrench button they have installed on each dock for each bike. This locks the bike into the dock, so that other riders cannot take it out, and it also notifies Citi Bike that the bike needs attention. Users can also press this button themselves, if they experience a malfunction while riding.

When Citi Bike sees a red lit bike, they will first send the Field Mechanic to check on it. Citi Bike will then pick up any bikes they weren't able to fix in the field, and take them to one of their two bike shops. The two bike shops are located at 1) the Farley Building on 31st Street by Penn Station and 2) Sunset Park, where Citi Bike's main offices are located.

2 The Proposed Problem

2.1 Problem Statement

After evaluating Citi Bike's current methods, we confirmed this process as a problem that can be optimized. Costs lie in the repair truck itself, gas and labor to drive to, check and drive the bikes back.

Our goal is to find the optimal number of trucks that minimize the total costs while maintaining the number of functioning bikes at each station above 98%. Our model and simulation are dependent upon Citi Bike's real-time and historical data, so we can see how the number of trucks needed vary upon time of day.

2.2 Business Opportunity

This repair problem poses a very important business opportunity. The breakdown of a bike is a probabilistic event, and its operations must be anticipated. For this problem, the business opportunity lies predominantly in the operational cost of the trucks and customer satisfaction.

Cost of truck & truck operations: By knowing the optimal number of trucks to use, we can essentially cut down on operational costs while taking into account their capacity, utilization rate, and distance traveled.

Customer satisfaction: If a disabled bike is not repaired or picked up in a timely manner, it can inconvenience the riders. The disabled bike could be occupying a dock and preventing the rider from returning their bike, or the disabled bike can disappoint an eager rider who now has to walk to work. It is natural to assume that Citi Bike would want to have a guaranteed minimum percentage of functioning bikes available.

3 Methodology

3.1 The Modeling Framework

As mentioned in section 2.1, our objective is cost minimization with respect to repair trucks while maintaining at least 98% of the bikes functional at all times. We will model the cost function as a simple linear function with a fixed cost per truck and a variable cost proportional to the driving distance of each truck. The mathematical model is therefore as follows:

$$\min C = n \times FC + D \times VC$$

s.t.

$$\frac{\#\text{ of functioning bikes}}{\#\text{ total bikes}} \geqslant 0.98$$

where

C: total cost ($/hour)

FC: fixed cost ($/truck/hour)

VC: variable cost ($/km)

n: number of trucks

D: total distance driven by all trucks per hour (km/hour)

The only decision variable in our model will be the number of trucks to use (n).

3.2 The Methodology

In order to solve this optimization problem, we will employ a discrete event simulation model (see section 3.3 for explanation). This simulation allows us to model a progression of bike breakdowns and truck pick-ups, simulating forward in time for a specified amount of steps. Trucks are assigned a specific area within New York City and pick up broken bikes from the Citi Bike stations. Upon reaching their full capacity, the trucks then drive to the repair station locations and drop off the broken bikes. With this simulation, we can track parameters such as the number of broken bikes in the system, or the % of time the trucks are not in use.

The starting state of the simulation is such that no bikes are broken. Therefore, the simulation needs to run for some unknown time until it has converged to a (nearly) steady state in which the number of broken bikes is roughly constant over time. The parameters that we gather using the simulation will be recorded in this steady state situation.

In order to solve the above optimization problem, we repeat the discrete event simulation for the number of trucks n ranging from 1 truck to 5 trucks. Parameters for each of these simulation runs are recorded in steady state. The optimal solution can then be found by locating the solution that achieves 98% functioning bikes at the smallest total cost. This can be done simply by inspection since there are only 5 possible solutions in the solution space.

3.3 Discrete Event Simulation

Discrete event simulation is used to model complex systems as an organized and ordered sequence of events. The simulation captures the changes of state over a period of time, where events are arrivals and departures. In our specific problem, the arrivals are bike breakdowns, and the departures are trucks arriving to pick the bikes up and drop them off at a bike warehouse.

The time between successive arrivals $t_{\alpha,i}$ (bike breakdowns) at one Citi Bike station is modeled as an expo-

nential distribution. The rate of this distribution for station i is:

$$\lambda_i = (\text{individual bike breakdown}) \times (\#\text{ bikes arriving to station i})$$

The time between successive departures t_{dj} (truck arriving at station) is modeled as exponential as well. The rate of this distribution for each truck j is:

$$\mu_j = \frac{\text{normalization factor}}{\text{distance from current station to next station}}$$

The normalization factor is used to calibrate the rate such that an average speed that matches the driving speeds in Manhattan is achieved.

In the initial state of the simulation, no bikes are broken. The $t_{\alpha i}$ for each bike is simulated using the inverse transform method and each t_{dj} is set to infinity since no truck will move unless there is a broken bike to be serviced. Next, the smallest of all arrival and departure times is determined. This is the first discrete event in the simulation and therefore must be an arrival (broken bike). The truck responsible for the respective station is sent to this station by simulating a truck arrival time, again using the inverse transform method. The simulation then loops through the code, looking for the smallest arrival or departure time. The pseudocode found in the appendix illustrates the general idea.

3.4 Data

The data that is used in the model is summarized in Figure 1. The following sections will talk about how this data is used in the simulation.

The two types of location data, namely *the Citi Bike station locations* and *the repair station locations* are used to set the next destination of each truck in the simulation. In addition, the distance between the stations is used to update the travel time rate of each truck, since distance is the only factor we use to calculate travel time between two stations.
The number of bikes at each station are used to find the percentage of broken bikes in the system to the total number of bikes. This data is not used in the discrete event simulation, but rather in the output of the simulation, where we display the percentage of bikes that are still functioning at any point in time.
The *number of bike repairs per month, number of rides per month* and *average ride duration for each month* are all used to find the bike breakdown rate of a single bike per hour. The equation is as follows:

$$\lambda_{bike} = \frac{\text{bike repairs per month}}{\#\text{ rides per month} \times \text{avg ride duration}}$$

Table 1 shows the single bike breakdown rates for each month in 2016.
In order to find the total rate of bike breakdowns for a station, the distribution of bike end station locations is used as follows:

$$\lambda_i = \lambda_{bike} \times \#\text{ bikes arriving at station i}$$

where the number of bikes en route to station i is taken from the distribution of bike end station locations.

3.5 Code Sequence

This section will provide additional details regarding the implementation of the model. A detailed explanation of the code is provided in the appendix in the form of code comments.

The first step in the code is to load all the needed data, as explained in section 3.4 of this report. The bike station locations are then grouped into clusters using the K-Means algorithm, where the number of clusters

is set to the number of trucks to be used. Each truck is then assigned to a cluster and each cluster is assigned one of the two repair station locations, based on distance to the cluster center.

Next, the main loop of the code is entered, where the discrete simulation is performed as explained in section 3.3. In each loop, the next event is found and the simulation is updated. If a truck has reached its full capacity it is routed to the repair station location to drop off the bikes. If, on the other hand, a truck has just reached a Citi Bike location and picked up all the bikes, it is sent to the next station in its cluster with the most number of broken bikes.

The loop described above runs until a set number of iterations has been reached at which point the parameters of the simulation run are returned.

3.6 Modeling Assumptions

The proposed model is naturally an oversimplification of the real world. The main modeling assumptions will be listed in the next paragraphs.

One of the main assumptions in the model is that only broken bike pick-ups are modeled. Specifically, the trucks do not return fixed bikes to the stations. Moreover, trucks will always choose the next station they visit to be the station in their cluster with the most number of broken bikes, regardless of how far away that station is.

In terms of service time and arrival distributions, the model assumes exponential distributions for both of these. However, the model would still be valid for other types of distributions. In addition, the model assumes that the rate of the exponential travel time distribution only depends on the direct distance between origin and destination. Specifically, the travel time is not influenced by the true driving time through the streets of New York City. This can be a good assumption as long as a truck does not have to cross one of the rivers traversing the city, which is sometimes the case in our model.

Finally, the last assumption is that the fixed cost per truck is \$60/truck/hour and the variable cost is \$5/km. These numbers were picked without any data to support their validity. Instead, they are chosen in order to illustrate how the model could be used to find the optimal number of trucks. If this model were to be used for real decision making, then these two values would have to be carefully estimated using other models.

4 Conclusion

4.1 Analysis

The analysis is based on the long-run simulation output of our model. The simulation is run for a set number of trucks until steady state is reached. The bike breakdown rate can be adjusted for 'afternoon 'conditions or for 'night 'conditions. The following paragraphs will discuss the optimal solution for either of these cases.

Figure 2 shows the output for one truck under afternoon bike breakdown conditions (click here for simulation video). An instant of the simulation can be seen in Figure 8. The top left graph shows the distance the truck is traveling. The steady increase in distance shows that there are no idle times for this truck, i.e. it is working at full capacity. The top right graph shows the average distance per time step in the simulation. This graph is useful to see how the simulation reaches steady state. The bottom left graph shows that the number of broken bikes increases until it reaches approximately 1000 broken bikes. As you can see on the bottom right graph, this corresponds to more than 10% broken bikes, which in turn means that one truck will not be enough to cope with the afternoon conditions.

In Figure 3, which displays the output for two trucks under afternoon conditions, the situation is a little better (click here for simulation video). An instant of the simulation can be seen in Figure 9. The trucks are still working at full capacity, as seen in the top left graph, but they manage to keep the percentage of broken bikes at around 5%. This is, however, still too large based on our constraint.

Finally, for three trucks under afternoon conditions (Figure 4), the percentage of functioning bikes is at about 99% which meets our constraints (click here for simulation video). An instant of the simulation can be seen in Figure 10. In addition, the distance traveled is still increasing steadily, showing that truck idle times are not significant.

For the 'night' conditions, one would expect a smaller amount of trucks to be needed, since there are fewer bikes being used. This intuition is confirmed in Figure 5, which shows the output for one truck under 'night' conditions (click here for simulation video). From the bottom right graph one can see that just a single truck is able to keep the percentage of functioning bikes at almost 100%. Therefore, no simulation runs for two or more trucks are needed.

4.2 Sensitivity Analysis

After concluding the optimal number of trucks is 3 from the long-run simulation, a sensitivity analysis was performed to observe how this number may change for varying parameters. The two parameters changed in this analysis are *truck capacity* and *bike breakdown rate*. For the truck capacity parameter, the capacities are set to 5, 10, and 15 bikes per truck. For the bike breakdown rate sensitivity analysis, three rates were chosen - a high, low, and median.

4.2.1 Varying Truck Capacities

Figure 6 shows the results when running the simulation for an increasing amount of trucks from 1 to 5. The four metrics recorded during this simulation are total distance traveled per truck, idle time (in %), percentage of functioning bikes to total number of bikes, and total cost in $/hour for varying truck quantities.

In the upper left hand graph, in Figure 6, the total distance traveled per truck is decreasing as the number of trucks increase. This is to be expected, because as the number of trucks increase, the workload is divided among the trucks and each is traveling less distance. Here, truck capacity does not seem to be an important factor. In the upper right hand graph, the idle time starts at 0% for 1 truck, it is traveling at all times. The idle time generally increases as the number of trucks increase, because the more trucks in use, the less distance each truck is traveling, and therefore has a higher idle time percentage. Idle time and distance traveled are inversely related, which is observed in these two graphs.

The lower right hand graph of Figure 6 represents the total cost associated with having 1 to 5 trucks for capacities 5, 10, and 15. The total cost includes the fixed costs per additional truck and its variable costs depending upon distance traveled. As stated earlier, it is assumed in the model that the fixed cost per truck is 60 $/$(truck \times hour)$ and the variable cost is 5 $/kilometer. The variable costs are decreasing as the number of trucks increase, since the distance traveled by each truck is less. Hence, the slight convergence as number of trucks increases. For this graph also, truck capacity seems to have no large effect.

Finally, the lower left hand graph of Figure 6 represents the percentage of functioning bikes to total number of bikes. For varying truck capacity, the percentage of functioning bikes is above 98% for all capacities at *3 trucks*.

It is also critical to notice that more than 3 trucks causes the idle time percentage to drastically increase. This can be interpreted that at 4 trucks, each truck is not being used to its full capability. Hence, this reaffirms that 3 trucks is the optimal number while maintaining percentage of functioning bikes above 98%. From this sensitivity analysis, it can be concluded that changing truck capacity does not have a significant impact for a smaller number of trucks, and the optimal number remains at 3.

4.2.2 Varying Bike Breakdown Rates

Figure 7 shows the results when running the simulation for an increasing amount of trucks from 1 to 5, and varying bike breakdown rates. The metrics recorded during the simulation are the same four metrics for when we varied the truck capacity: total distance traveled per truck, idle time (in %), percentage of functioning

bikes to total number of bikes, and total cost in $/hour for varying truck quantities.

We chose the 3 bike breakdown rates used in this sensitivity analysis from Table 1. The figure displays how we estimated the average breakdown rate per hour, which lists the breakdown rates by month. We used the highest and lowest rates that we recorded for the months of 2016.

For the upper two graphs in this figure, the total distance traveled is initially the same for all three rates at 1 truck, and thus idle time is 0%. As the number of trucks increases, the idle time reaches above 0% and the travel distance per truck decreases and has larger idle times.

Looking at the lower right hand graph in Figure 7, the total cost increases at a smaller rate for smaller breakdown rates once the idle time increases and truck travel distance starts to decrease.

Again, the lower left hand graph in Figure 7 represents the percentage of functioning bikes to total number of bikes. For varying bike breakdown rates, the percentage of functioning bikes is above 98% for all breakdown rates at *3 trucks*. This also reaffirms that the optimal solution for the afternoon is 3 trucks. Similar to the truck capacity analysis, the idle time is also critical to notice.

For more than 3 trucks, the idle time percentage drastically increases for the two higher breakdown rates. Likewise, this can be interpreted that at 4 trucks, each truck is not being used to its full capability with idle time percentages of 51, 41, and 34 respectively with increasing breakdown rates. This also reaffirms that 3 trucks is the optimal number while maintaining percentage of functioning bikes above 98%. From this sensitivity analysis, as well, it can be concluded that varying the bike breakdown rates does not change the optimal number of bikes to use in the afternoon.

4.3 Solution

From our data and analysis, we have concluded that the optimal number of trucks is **3 in the afternoon** and **1 during the night**. In the afternoon, we analyzed the long-run simulation output, to see that 3 trucks gives us a percentage of functioning bikes to be about 99%, which meets our problem constraint. For the 'night' conditions, Figure 5 shows that a single truck is able to keep the percentage of functioning bikes at almost 100%. Therefore, we concluded that the optimal number of bikes is 1 for this time of day.

Our solution for Citi Bike is to implement these optimal numbers of truck to be used to pick up disabled bikes, based on time of day. Three trucks for the afternoon and one truck during the night satisfies the problem constraints of minimizing costs while maintaining the percentage of functioning bikes above 98%. For Citi Bike as a company, this is overall very important to keep their riders happy, while also being conscious of their operating costs.

5 Appendices

5.1 Tables, Plots, Graphs, Visuals

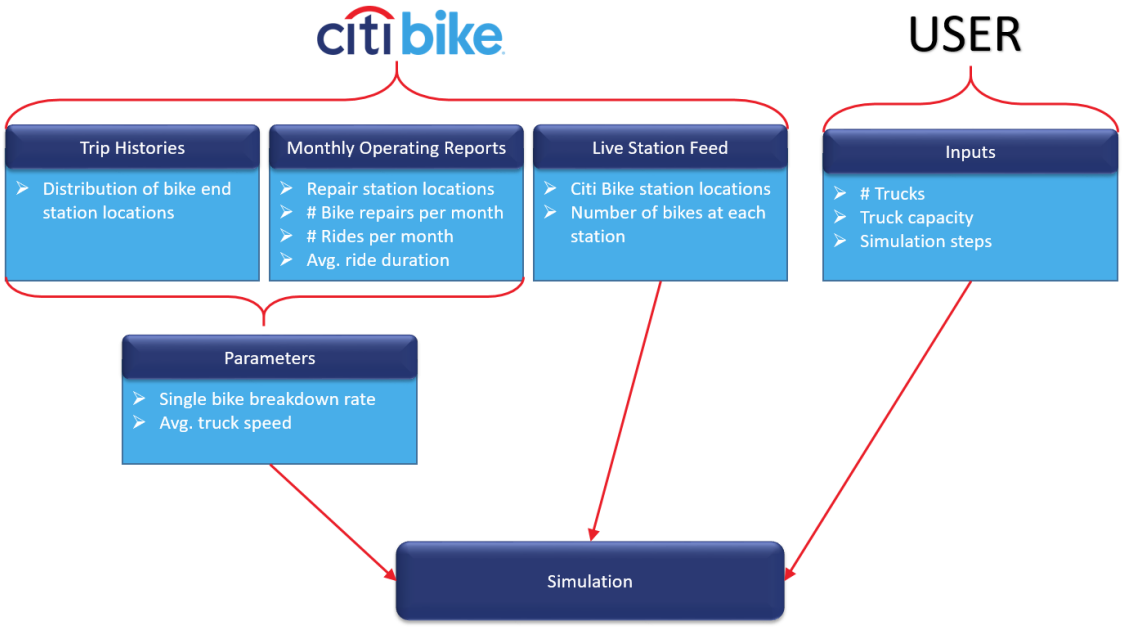

Figure 1: Data and parameter inputs

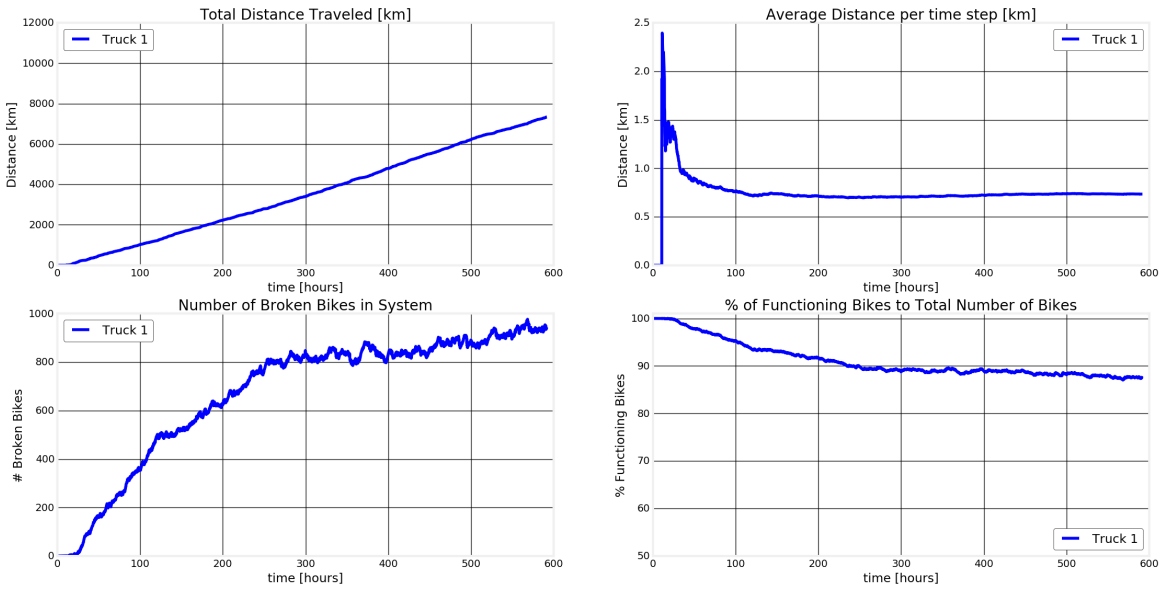

Figure 2: One truck long run simulation (afternoon)

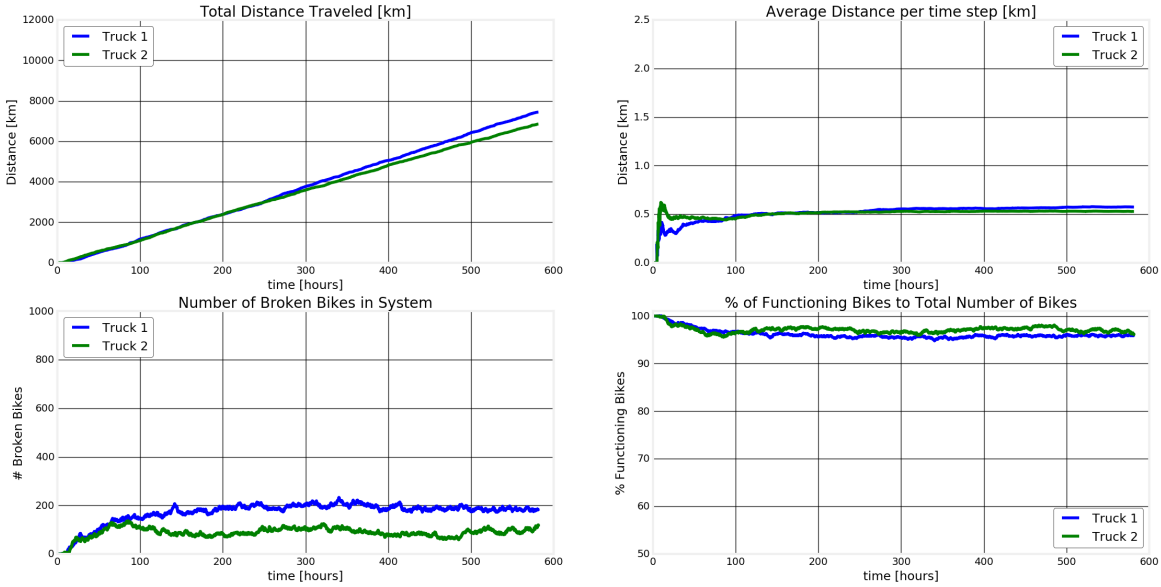

Figure 3: Two trucks long run simulation (afternoon)

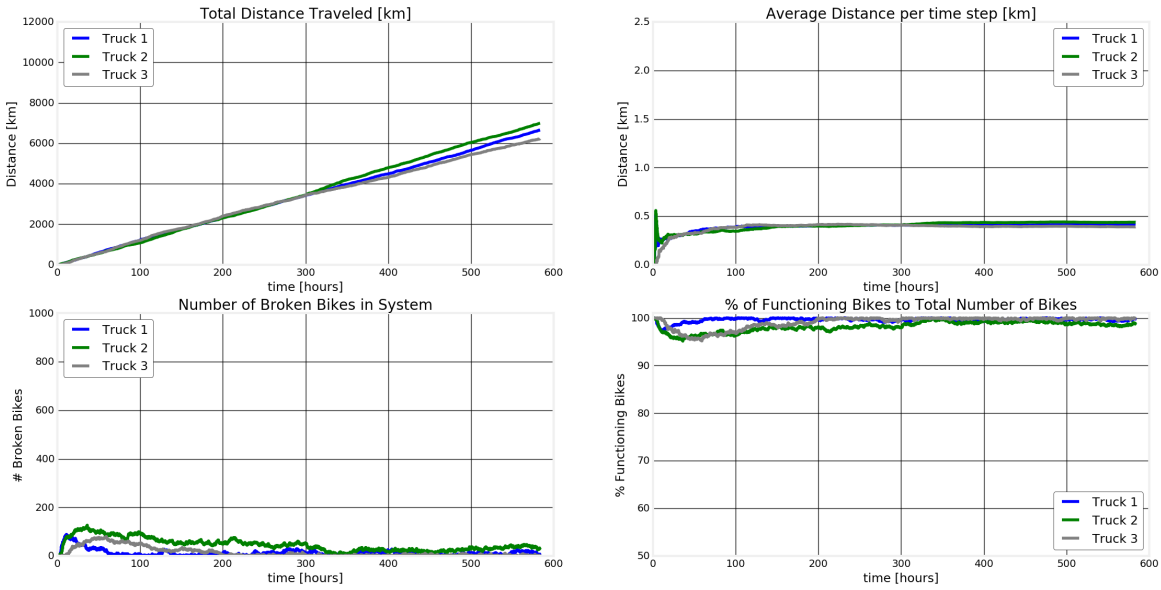

Figure 4: Three trucks long run simulation (afternoon)

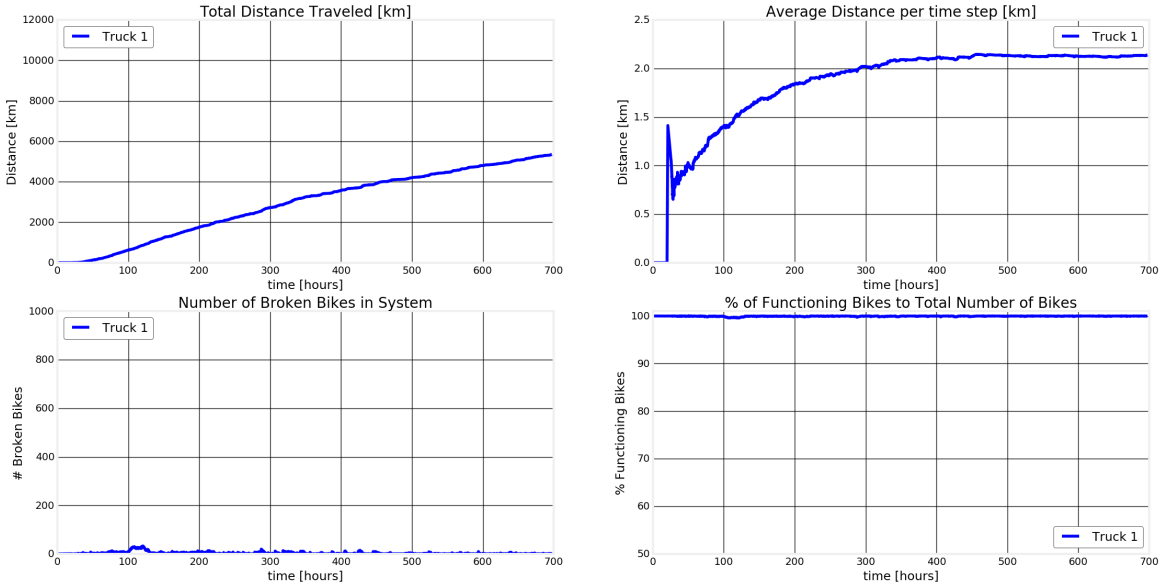

Figure 5: One truck long run simulation (night)

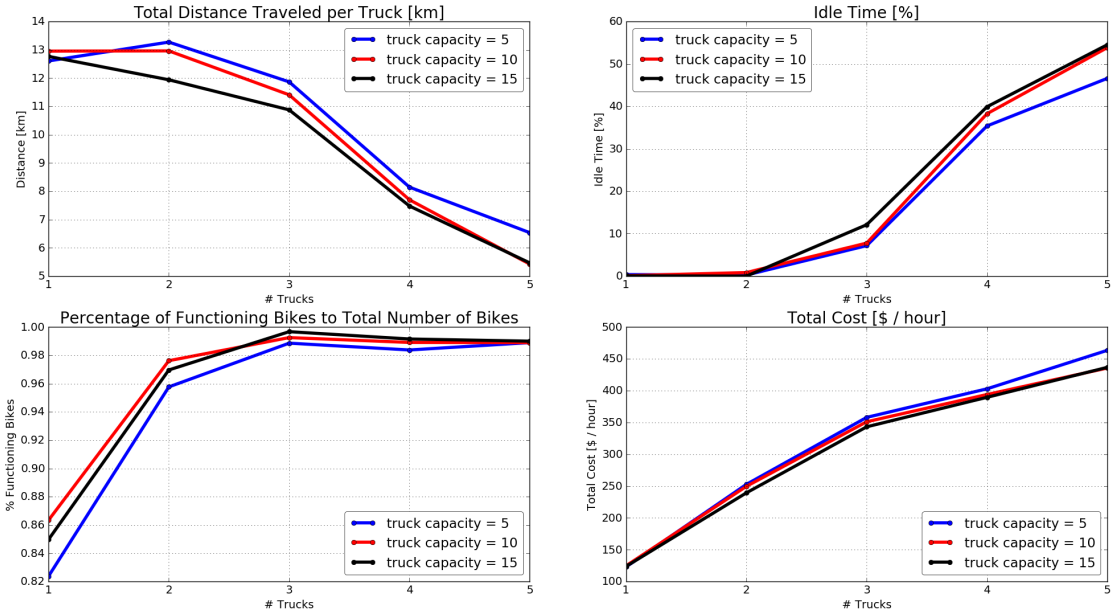

Figure 6: Sensitivity analysis of truck capacity

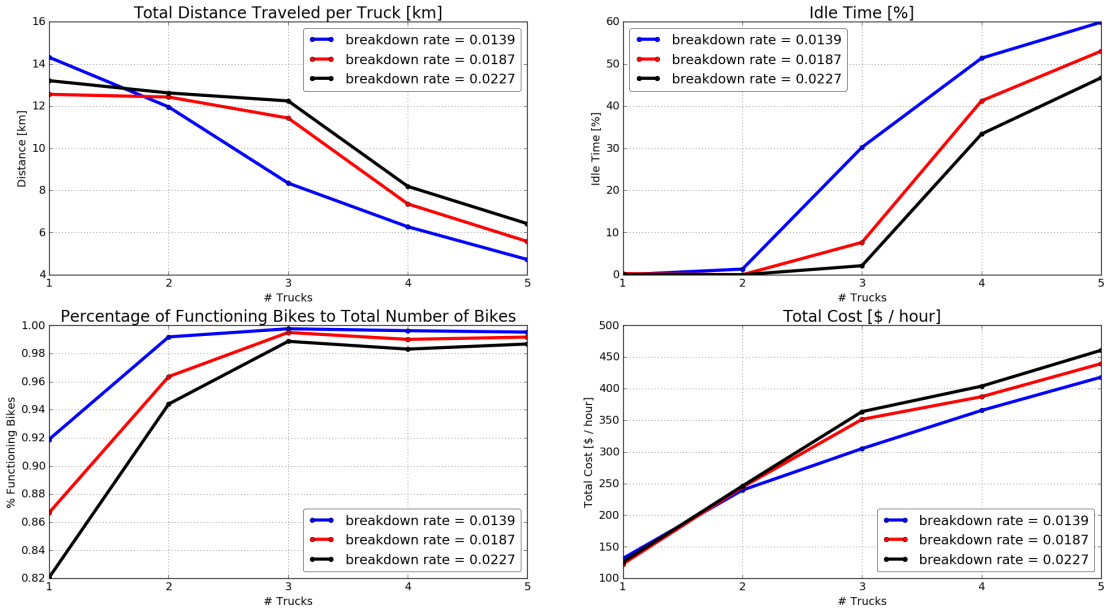

Figure 7: Sensitivity analysis of bike breakdown rate

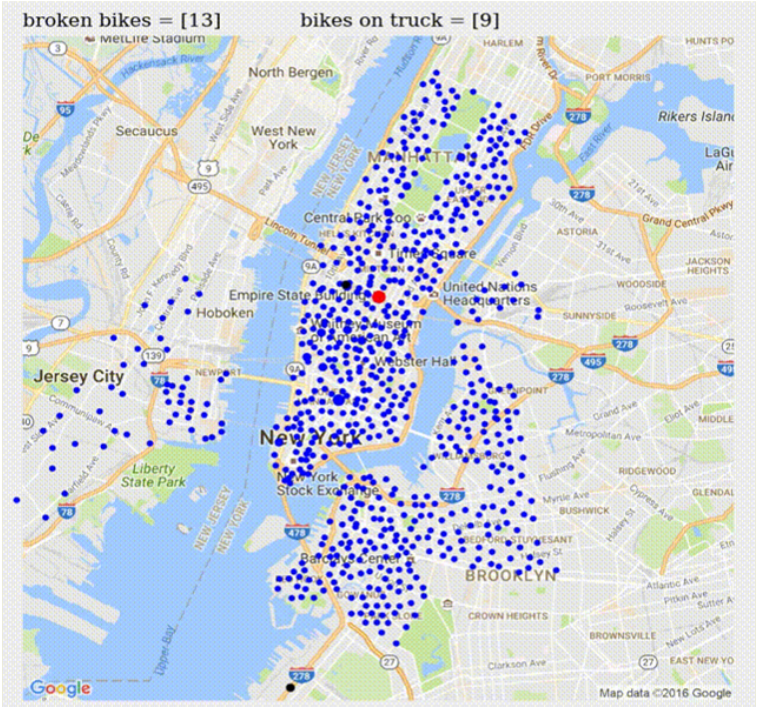

Figure 8: Instant of simulation for 1 truck in the afternoon

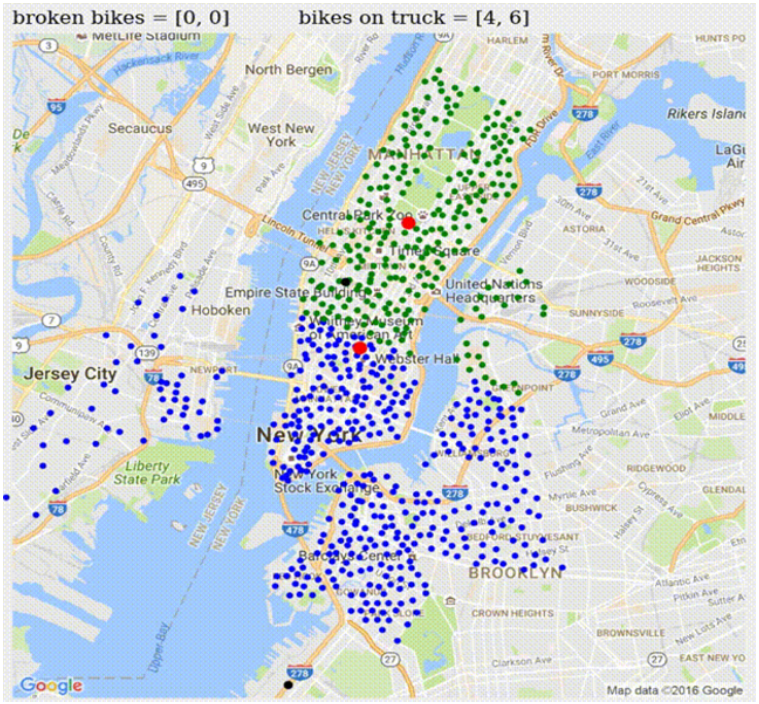

Figure 9: Instant of simulation for 2 trucks in the afternoon

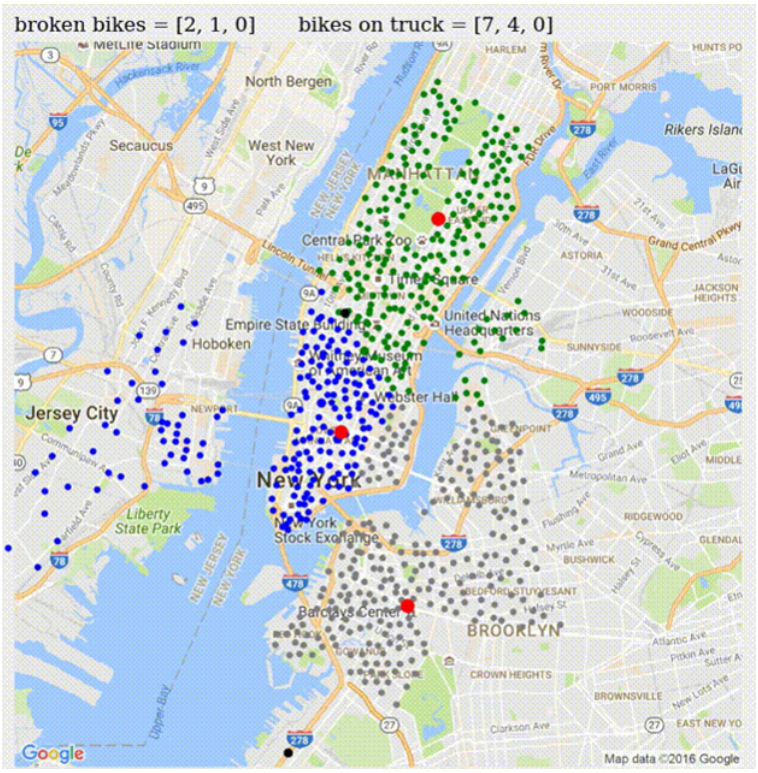

Figure 10: Instant of simulation for 3 trucks in the afternoon

Month	# Repairs	# Rides	Avg. Ride Length [min.]	Bike-Breakdown-Rate (per hour)
January	2,319	509478	12	0.02276
February	2,655	560865	12	0.02367
March	3791	919911	14	0.01766
April	5253	1013126	14	0.02222
May	5417	1212342	14.81	0.01810
June	5584	1460303	14.71	0.01560
July	4687	1379871	14.7	0.01386
August	6394	1557404	14.72	0.01673
September	7569	1648525	14.73	0.01870
October	6239	1573653	13.83	0.01720
Average bike-breakdown-rate (per hour): 0.01865				

Table 1

5.2 Discrete Event Simulation Pseudocode

initialization: set $\lambda_{\alpha,i}$
simulate the $t_{\alpha,i}$ using inverse transform method and set all $t_{d,j} = \infty$
set $t = 0$
set initial truck positions $(truck_x, truck_y)$
loop:
    plot updated animation
    if the next event is a truck arrival:
        set t to the time the event happened
        if the truck arrived at a repair station:
            set variable that keeps track of number of bikes on truck to 0
        else if we arrived at a bike station
            add number of broken bikes to variable that keeps track of number of bikes on truck
            set number of broken bikes at that station to 0
        if the truck capacity has not been reached:
            set next destination of truck to the bike station in its cluster with the most broken bikes
        else if the truck capacity has been reached:
            set next destination of truck to the repair station
        set $\mu_(d,j) = \frac{\text{normalization factor}}{\text{distance to next destination}}$
        simulate next truck arrival $t_{d,j} = t - \log(Uniform(0,1)/\mu_{d,j}$(inverse transform method)
    else if the next event is a bike breakdown:
        if the bike breakdown is the first broken bike in the cluster
            set next destination of truck to that bike station
        else
            add 1 to the number of broken bikes at that station
        simulate next bike breakdown $t_{a,i} = t - \log Uniform(0,1)/\lambda_{a,i}$ (inverse transform method)
    update all truck positions $(truck_x, truck_y)$

5.3 Code

Listing 1: Main code including discrete event simulation

```
# Transportation Project Simulation
def simulation(num_stat,num_trucks,lambda_bike,norm_factor,load_rate,truck_cap,time_of_day,
        num_steps,plot_animation,save_animation):
        # import libraries
        import matplotlib.pyplot as plt
        from matplotlib import style
        import random as rd
        import math
        from live_feed import station_status, station_info
        import google_map_api_cornerpoints
        from io import BytesIO
        from PIL import Image
        from matplotlib import rcParams
        from sklearn.cluster import KMeans
        import numpy as np
        np.seterr(all='ignore') # ignore division by 0 error (we set equal to Inf in that case)
        import csv
        import pandas as pd
```

```
# events are:   t_a[i]: bike breaks down on station i
#               t_d[i]: truck i arrives at a station

# creating styled window for graphs
rcParams['figure.figsize'] = (10, 10) #Size of figure
rcParams['grid.alpha'] = 0
style.use('fivethirtyeight')

# repair station locations
rep_station_x_pos = [-73.9952, -74.0124]
rep_station_y_pos = [40.7512 , 40.65]
num_rep_stat = len(rep_station_x_pos)
rep_locations = [[rep_station_x_pos[i],rep_station_y_pos[i]] for i in range(num_rep_stat)]

# bike stations
station_lat = station_info('lat')
station_lon = station_info('lon')
station_x_pos = station_lon[0:num_stat]
station_y_pos = station_lat[0:num_stat]
markersize = np.array([20 for i in range(num_stat)])

# finding clusters
data_cluster = list(zip(station_x_pos,station_y_pos))
kmeans = KMeans(n_clusters=num_trucks, random_state=0).fit(data_cluster)
cluster_labels = kmeans.labels_.tolist()
cluster_centers = kmeans.cluster_centers_

# finding closest repair station for each cluster
def closest_node(node, nodes):
        nodes = np.asarray(nodes)
        dist_2 = np.sum((nodes - node)**2, axis=1)
        return np.argmin(dist_2)

closest_rep_station = [closest_node(cluster_centers[i], rep_locations) for i in range(
    num_trucks)]

# adding repair station location to end of bike station positions
# (needed so that we can send the trucks to the repair stations simply by sending them to
# the last station_x_pos and station_y_pos)
station_x_pos = station_x_pos + rep_station_x_pos
station_y_pos = station_y_pos + rep_station_y_pos

# initial truck positions (set to first bike station location of each cluster)
truck_x_pos = [station_x_pos[cluster_labels.index(i)] for i in range(num_trucks)]
truck_y_pos = [station_y_pos[cluster_labels.index(i)] for i in range(num_trucks)]

# mapping between latitude/longitude and pixels for plotting
# needed because the google map in the background does not use latitude/ longitude information
# but the station and truck locations do
centerLat = 40.73
centerLon = -73.9851
zoom = 12
mapWidth = 640
```

```
mapHeight = 640
centerPoint = google_map_api_cornerpoints.G_LatLng(centerLat, centerLon)
corners = google_map_api_cornerpoints.getCorners(centerPoint, zoom, mapWidth, mapHeight)
# the mapping is (x_plot, y_plot) = (longitude * m_x + n_x,latitude * m_y + n_y)
m_x = 1279.5 / (corners['E']-corners['W'])
n_x = -corners['W']*m_x
m_y = 1279.5 / (corners['S']-corners['N'])
n_y = -corners['N']*m_y

# map the bike station positions to the plotting reference frame
station_plot_x = np.array([i*m_x + n_x for i in station_x_pos])
station_plot_y = np.array([i*m_y + n_y for i in station_y_pos])

# map the repair station positions to the plotting reference frame
rep_station_plot_x = [i*m_x + n_x for i in rep_station_x_pos]
rep_station_plot_y = [i*m_y + n_y for i in rep_station_y_pos]

# image of NYC (background)
image = Image.open("file.png")

# colors to use for plotting clusters (up to 5 clusters supported)
cluster_col = ['blue','green','gray','orange','cyan']

# stores list of station-ids. This is the order in which all stations will be referenced
station_order = list(map(int, station_info('station_id')))

# create a pandas data frame of zeros with colum names "columns" and index station_id
columns = ['night','morning','noon','afternoon','evening']
data = np.empty((len(station_order),5,))
data[:] = np.zeros(1)
station_df = pd.DataFrame(data,index = station_order,columns = columns)
station_df.index.name = 'station_id'

# read in the bike ride distribution data gathered in R
station_df2 = pd.read_csv('frequency.csv',index_col='station_id')

# superimpose the data onto the pandas dataframe. This is necessary because stations are
# omitted from the bike ride distribution if there were no bikes going to them. This
# step sets the values for those stations to zero.
station_df.update(station_df2)

# the bike ride distribution is "bikes_avail". Multiply by the single bike breakdown rate
# to get the overall bike breakdown rate at each station (lambda stations)
bikes_avail = list(np.array(station_df[time_of_day])*14/(60*30))[0:num_stat]
lambda_stations = np.array([i*lambda_bike for i in bikes_avail])

# number of bikes at each station. Used to find total number of bikes in the streets
bikes_in_stations = station_status('num_bikes_available')[0:num_stat]

# initial truck rates to reach destinations
mu = [norm_factor / (10 + load_rate) for i in range(num_trucks)]

# lists that will be returned by the function for analysis
time_list = [0] # list of times at which the events occur
```

```
# list of lists where each list records the number of broken bikes in the system at each event
    time
num_broken_bikes = [[0] for i in range(num_trucks)]
# distance traveled by each truck between the last event time and current event time
delta_dist = [[0] for i in range(num_trucks)]

# font used for text on the plot
font_dict = {'family': 'serif', 'color': 'black', 'size':15}

# definitions for loop
R = 6371 # earth radius(used to find distance traveled by the trucks)
t = 0 # initializing current time
# initializing current number of broken bikes in each cluster (list of length(num_trucks))
n = [0 for i in range(num_trucks)]
# order is a list of list where each list stores station id's of the broken bikes.
order = [[] for i in range(num_trucks)] # Each time there is a new broken bike it's
                                        # station-id is appended to order
last = [cluster_labels.index(i) for i in range(num_trucks)] # stores last station-id each truck
                                                            # was at
current = [cluster_labels.index(i) for i in range(num_trucks)] # stores station-id each truck
                                                               # is going to
next_dest = [float('NaN') for i in range(num_trucks)] # stores next station-id that each truck
                                                      # will go to after "current"
current_num_bikes = [0 for i in range(num_trucks)] # stores the current number of bikes
                                                   # that need to be picked up at "current"
bikes_on_truck = [0 for i in range(num_trucks)] # stores the number of bikes on each truck
# stores the last time each truck arrived at a bike station location
t_last_d = [0 for i in range(num_trucks)]
t_perc = [0 for i in range(num_trucks)] # stores how much of the current trip each
                                        # truck has completed (between 0 and 1)

# initializing next arrivals
t_d = [float('Inf') for i in range(num_trucks)] # Next arrival time for each truck. Infinity
                                                # because the trucks won't move until there is
                                                # a broken bike
t_a = list(-math.log(rd.uniform(0,1)) / lambda_stations) # time of next broken bike
                                                         # at each station

# main loop
for i in range(num_steps):
    if plot_animation: # only plot if set to True since plotting takes a LOT of time
        plt.clf() # erase current plot
        plt.imshow(image) # insert background of Manhattan
        ax = plt.gca()
        ax.set_axis_bgcolor('none')
        # map the updated truck positions to the plotting reference frame
        truck_plot_x = [i*m_x + n_x for i in truck_x_pos]
        truck_plot_y = [i*m_y + n_y for i in truck_y_pos]
        for clusters in range(num_trucks): # for each of the clusters
            # only plot the current cluster stations (we are looping through each)
            which_points = [i == clusters for i in cluster_labels]
            plt.scatter(station_plot_x[0:-num_rep_stat][which_points],
                        station_plot_y[0:-num_rep_stat][which_points],
                        color=cluster_col[clusters],
```

```
                    s = markersize[which_points])
        plt.scatter(truck_plot_x, truck_plot_y, color='red',s=100) # plot updated truck
                                                                   # positions
        # plot repair station positions
        plt.scatter(rep_station_plot_x,rep_station_plot_y,color = 'black', s = 40)
        # plot number of broken bikes for each cluster
        plt.text(0, -20,"broken bikes = " + str(n), fontdict=font_dict)
        plt.text(500, -20,"bikes on truck = " + str(bikes_on_truck),
                fontdict=font_dict) # plot number of bikes on each truck

        plt.pause(0.00001) # pause a short time so the plot is shown on screen
        # If set to True, each animation step will be saved as a png file.
        # Used to create gifs of the animation. It slows down the simulation by a lot!
        if save_animation:
                plt.savefig(str(i) + '.png', dpi=100) # save as individual png files


# find which t_d or t_a is the minimum
val, counter = min((val, idx) for (idx, val) in enumerate(t_d+t_a))

# if the next event is a truck arrival
if counter < len(t_d):
        t_perc[counter] = 0 # truck has completed 0% of its trip to next destination
                            # (after the one at which it has just arrived)
        t_last_d[counter] = t_d[counter] # the last arrival time is the current arrival
                                         # time (since it has just arrived)
        t = t_d[counter] # the current time is now the arrival time of the truck
        time_list.append(t) # append current event time to list of event times
                            # (used for analysis of run)
        n[counter] -= current_num_bikes[counter] # the truck picks up all bikes from
                                                 # the current location so there are
                                                 # less broken bikes on the streets now
        # record current number of broken bikes on streets (used for analysis of run)
        [num_broken_bikes[i].append(n[i]) for i in range(num_trucks)]
        if current_num_bikes[counter] == 0: # if we have arrived at a
                                            # repair station location
                bikes_on_truck[counter] = 0 # unload all the bikes currently on truck
        else: # if we are at a bike station location
                # load all currently broken bikes from the station to the truck
                bikes_on_truck[counter] += current_num_bikes[counter]
                # update markersize for station on plot
                markersize[current[counter]] = markersize[current[counter]] \
                / (2**current_num_bikes[counter])

        # set updated position of truck
        x_dist = math.radians(station_x_pos[current[counter]]-truck_x_pos[counter]) \
        *math.cos(math.radians(station_y_pos[current[counter]]+truck_y_pos[counter])\
        / 2)
        y_dist = math.radians(station_y_pos[current[counter]] - truck_y_pos[counter])
        delta_dist[counter].append(math.sqrt(x_dist**2 + y_dist**2) * R)
        truck_x_pos[counter] = station_x_pos[current[counter]]
        truck_y_pos[counter] = station_y_pos[current[counter]]

        # set position of other trucks
```

```
for z in range(num_trucks):
        if z != counter:
                t_perc[z] = (t - t_last_d[z]) / (t_d[z] - t_last_d[z])
                x_dist = math.radians(station_x_pos[last[z]] + t_perc[z] \
                    *(station_x_pos[current[z]] - station_x_pos[last[z]]) \
                    - truck_x_pos[z]) \
                    * math.cos(math.radians(station_y_pos[last[z]] \
                        + t_perc[z]*(station_y_pos[current[z] \
                        -station_y_pos[last[z]])+truck_y_pos[z]) / 2)
                y_dist = math.radians(station_y_pos[last[z]]+t_perc[z] \
                *(station_y_pos[current[z]] - station_y_pos[last[z]]) \
                - truck_y_pos[z])
                delta_dist[z].append(math.sqrt(x_dist**2 + y_dist**2) * R)
                truck_x_pos[z] = station_x_pos[last[z]] + t_perc[z] \
                *(station_x_pos[current[z]] - station_x_pos[last[z]])
                truck_y_pos[z] = station_y_pos[last[z]] + t_perc[z] \
                *(station_y_pos[current[z]] - station_y_pos[last[z]])

# if there are no trucks left to pick up and if the truck has not reached its
# full capacity yet
if n[counter] == 0 and bikes_on_truck[counter] < truck_cap:
        last[counter] = current[counter] # the last arrival station-id is set to
                                         # the current one
        t_d[counter] = float('Inf') # the next truck arrival time is infinity
                                    # since it has nowhere to go for now
        current_num_bikes[counter] = 0 # there are no bikes at the next location
                                       # (since there is no next location)
        order[counter] = [value for value in order[counter] if value \
                          != current[counter]] # delete the bikes that have
                                               # been picked up from "order"
else: # if there are other bikes to pick up or if the truck capacity is reached
        if bikes_on_truck[counter] >= truck_cap: # if truck capacity is reached
            # set the next_destination to repair station
            next_dest[counter] = num_stat + closest_rep_station[counter]
            current_num_bikes[counter] = 0 # there are no bikes to pick up
                                           # at the repair station location
        else: # if there are other bikes to pick up (at other stations)
            next_dest[counter] = max(set(order[counter]),
            key=order[counter].count) # set the next destination to the one
                                      # with the most number of broken bikes
            # set to number of broken bikes at next location
            current_num_bikes[counter] = len([i for i, j in \
                    enumerate(order[counter]) if j == next_dest[counter]])
            # delete the bikes that have been picked up from "order"
            order[counter] = [value for value in order[counter] if value \
                              != next_dest[counter]]

        # update travel time rate to account for distance between current
        # location and next location
        mu[counter] = norm_factor / (load_rate \
                    + math.sqrt((station_x_pos[next_dest[counter]] \
                    - truck_x_pos[counter])**2 \
                    + (station_y_pos[next_dest[counter]] \
                    - truck_y_pos[counter])**2))
```

```
                # simulate next arrival time of this truck
                t_d[counter] = t - math.log(rd.uniform(0,1)) / mu[counter]

                # update "last" and "current" since the truck has arrived at a station
                # and is now going to the next station
                last[counter] = current[counter]
                current[counter] = next_dest[counter]
                next_dest[counter] = float('NaN') # we don't know yet what the next
                                                  #destination will be after current one

        else: # if the next event is a bike breakdown
            counter = counter - len(t_d) # counter is the index of which station it is
            t = t_a[counter] # update the time to reflect that an arrival has happened
            time_list.append(t) # append current time to list of event times
            cluster = cluster_labels[counter] # figure out in which cluster the arrival is
            # update how far trucks have traveled (since time has progressed)
            for z in range(num_trucks):
                t_perc[z] = (t - t_last_d[z]) / (t_d[z] - t_last_d[z])

            n[cluster] += 1 # add 1 to the number of broken bikes on the streets
            # append the current number of broken bikes on the streets (for later analysis)
            [num_broken_bikes[i].append(n[i]) for i in range(num_trucks)]
            # simulate next broken bike time at that station
            t_a[counter] = t - math.log(rd.uniform(0,1)) / lambda_stations[counter]

            # if the broken bike is the first one in the cluster and if the truck capacity
            # has not been reached
            if n[cluster] == 1 and bikes_on_truck[cluster] < truck_cap:
                last[cluster] = current[cluster] # update last station-id to current one
                current[cluster] = counter # update the current station to the one in
                                           # which the broken bike happened

                # updated travel time rate
                mu[cluster] = norm_factor / (load_rate \
                        + math.sqrt((station_x_pos[counter] \
                        - truck_x_pos[cluster])**2 \
                            + (station_y_pos[counter] \
                            - truck_y_pos[cluster])**2))
                # simulate travel time of truck
                t_d[cluster] = t - math.log(rd.uniform(0,1)) / mu[cluster]
                current_num_bikes[cluster] = 1 # set number of broken bikes at the
                                               # station that we are going to to 1

                # update position of all other trucks
                for y in range(num_trucks):
                    if y != cluster:
                        x_dist = math.radians(station_x_pos[last[y]] \
                            + t_perc[y]*(station_x_pos[current[y]] \
                                - station_x_pos[last[y]]) - truck_x_pos[y]) \
                                * math.cos(math.radians(station_y_pos[last[y]] \
                                + t_perc[y]*(station_y_pos[current[y]] \
                                - station_y_pos[last[y]])+truck_y_pos[y]) / 2)
                        y_dist = math.radians(station_y_pos[last[y]] \
                                + t_perc[y]*(station_y_pos[current[y]] \
```

```
                                        - station_y_pos[last[y]]) - truck_y_pos[y])
                                    delta_dist[y].append(math.sqrt(x_dist**2+y_dist**2)\
                                        *R)
                                    truck_x_pos[y] = station_x_pos[last[y]] + t_perc[y] \
                                    *(station_x_pos[current[y]] - station_x_pos[last[y]])
                                    truck_y_pos[y] = station_y_pos[last[y]] + t_perc[y] \
                                    *(station_y_pos[current[y]] - station_y_pos[last[y]])
                            delta_dist[cluster].append(0)

                        else: # if the broken bike is not the first one in the cluster or if the truck
                            # capacity has been reached
                            # update all truck positions
                            for y in range(num_trucks):
                                x_dist = math.radians(station_x_pos[last[y]] + t_perc[y] \
                                    *(station_x_pos[current[y]] - station_x_pos[last[y]]) \
                                    - truck_x_pos[y]) \
                                    * math.cos(math.radians(station_y_pos[last[y]] \
                                    + t_perc[y]*(station_y_pos[current[y]] \
                                    - station_y_pos[last[y]])+truck_y_pos[y]) / 2)
                                y_dist = math.radians(station_y_pos[last[y]] + t_perc[y] \
                                    *(station_y_pos[current[y]] \
                                    - station_y_pos[last[y]]) - truck_y_pos[y])
                                delta_dist[y].append(math.sqrt(x_dist**2 + y_dist**2) * R)
                                truck_x_pos[y] = station_x_pos[last[y]] + t_perc[y] \
                                    *(station_x_pos[current[y]] - station_x_pos[last[y]])
                                truck_y_pos[y] = station_y_pos[last[y]] + t_perc[y] \
                                    *(station_y_pos[current[y]] - station_y_pos[last[y]])
                            if current[cluster] == counter: # if the bike breakdown happened at the
                                                            # station that we are currently driving to
                                current_num_bikes[cluster] += 1 # add 1 to the number of
                                                               # broken bikes at the station we
                                                               # are going to
                            else: # if the bike breakdown did not happen at the station that we are
                                # currently driving to
                                order[cluster].append(counter) # add the station-id of the
                                                               # broken bike to "order"

                        markersize[counter] = markersize[counter] * 2 # update markersize of the station
                                                                      # at which the breakdown happened

        return time_list, bikes_avail, bikes_in_stations, num_broken_bikes, delta_dist, cluster_labels
```

Listing 2: Library to import live station feed data

```
# station status feed
def station_status(key):
    # keys:'eightd_has_available_keys', 'is_installed', 'is_renting',
    # 'is_returning', 'last_reported', 'num_bikes_available',
    # 'num_bikes_disabled', 'num_docks_available', 'num_docks_disabled',
    # 'station_id'
        from urllib.request import urlopen
        import json
        import pandas as pd
```

```
        data_stations = urlopen("https://gbfs.citibikenyc.com/gbfs/en/station_status.json").read()
                                .decode("utf-8")
        data_stations = json.loads(data_stations)
        data_stations = data_stations["data"]
        data_stations = data_stations["stations"]
        data_stations = pd.DataFrame(data_stations)

        return list(data_stations[key])

# station information
def station_info(key):
    # 'capacity', 'eightd_has_key_dispenser', 'lat', 'lon', 'name',
    # 'region_id', 'rental_methods', 'short_name', 'station_id'
        from urllib.request import urlopen
        import json
        import pandas as pd

        data_station_info = urlopen("https://gbfs.citibikenyc.com/gbfs/en/station_information.json")
                                .read().decode("utf-8")
        data_station_info = json.loads(data_station_info)
        data_station_info = data_station_info["data"]
        data_station_info = data_station_info["stations"]
        data_station_info = pd.DataFrame(data_station_info)
        return list(data_station_info[key])
```

Listing 3: Code for one long simulation run

```
from Project_simulation_12 import simulation
import matplotlib.pyplot as plt
from matplotlib import rcParams
import numpy as np

num_steps = 10
num_stat = 664 # max 664
num_trucks = 1
lambda_bike = 0.0187 # see Excel sheet for numbers
norm_factor = 0.148
load_rate = 0.005
truck_cap = 10
time_of_day = 'night' # pick between 'night','morning','noon','afternoon','evening'
plot_animation = True
save_animation = False

# calling the main code for the discrete event simulation
times, bikes_avail, bikes_in_stations, broken_bikes, delta_distance, clusters = simulation(num_stat,
    num_trucks,lambda_bike,norm_factor,load_rate,truck_cap,time_of_day,num_steps,
    plot_animation,save_animation)

# running statistics
distance = [np.cumsum(i) for i in delta_distance]
avg_distance = [[distance[i][j] / (j+1) for j in range(num_steps+1)] for i in range(num_trucks)]
bikes_in_clusters = [[bikes_in_stations[i] for i in range(len(bikes_avail)) if clusters[i] == j] \
```

```
                    for j in range(num_trucks)] # used for perc_good_bikes
bikes_per_cluster = [sum(i) for i in bikes_in_clusters] # used for perc_good_bikes
perc_good_bikes = [[100*(1 - (broken_bikes[i][j] / bikes_per_cluster[i])) \
                    for j in range(num_steps+1)] for i in range(num_trucks)]

#final statistics
total_distance = [sum(i) for i in delta_distance]
average_speed = [total_distance[i] / times[-1] for i in range(num_trucks)]
perc_idle_time = [sum([times[i+1] - times[i] for i in range(num_steps) if delta_distance[j][i] == 0]) \
                  / times[-1] for j in range(num_trucks)]
print(average_speed) # set norm_factor such that average speed is 16km/h at perc_idle_time = 0

# plotting parameter settings
rcParams['grid.alpha'] = 15
rcParams['grid.color'] = 'k'
rcParams['legend.edgecolor'] = 'k'
rcParams['legend.facecolor'] = 'w'

# using same colors for lines as cluster colors in animation
cluster_col = ['blue','green','gray','orange','cyan']

f, axarr = plt.subplots(2, 2) # initializing graph
#plotting results
for i in range(num_trucks):
        axarr[0, 0].plot(times,distance[i], color=cluster_col[i], label = 'Truck ' + str(i+1))
        axarr[0, 1].plot(times,avg_distance[i], color=cluster_col[i], label = 'Truck ' + str(i+1))
        axarr[1, 0].plot(times,broken_bikes[i], color=cluster_col[i], label = 'Truck ' + str(i+1))
        axarr[1, 1].plot(times,perc_good_bikes[i],color=cluster_col[i], label = 'Truck ' + str(i+1))
# setting titles for plots
axarr[0, 0].set_title('Total Distance Traveled [km]')
axarr[0, 1].set_title('Average Distance per time step [km]')
axarr[1, 0].set_title('Number of Broken Bikes in System')
axarr[1, 1].set_title('% of Functioning Bikes to Total Number of Bikes')
# setting y axis limits
axarr[0, 0].set_ylim([0, 12000])
axarr[0, 1].set_ylim([0, 2.5])
axarr[1, 0].set_ylim([0, 1000])
axarr[1, 1].set_ylim([50, 101])
# setting axis labels
axarr[0, 0].set_xlabel('time [hours]')
axarr[0, 1].set_xlabel('time [hours]')
axarr[1, 0].set_xlabel('time [hours]')
axarr[1, 1].set_xlabel('time [hours]')
axarr[0, 0].set_ylabel('Distance [km]')
axarr[0, 1].set_ylabel('Distance [km]')
axarr[1, 0].set_ylabel('# Broken Bikes')
axarr[1, 1].set_ylabel('% Functioning Bikes')
# setting background color
axarr [0,0]. set_axis_bgcolor('none')
axarr [0,1]. set_axis_bgcolor('none')
axarr [1,0]. set_axis_bgcolor('none')
axarr [1,1]. set_axis_bgcolor('none')
# setting legends
axarr [0,0]. legend(loc='upper left')
```

```
axarr[0,1].legend(loc='upper right')
axarr[1,0].legend(loc='upper left')
axarr[1,1].legend(loc='lower right')

plt.show()
```

Listing 4: Code for sensitivity analysis

```
from Project_simulation_12 import simulation
import matplotlib.pyplot as plt
import matplotlib.ticker as plticker
from matplotlib import rcParams
import numpy as np

# constants
num_steps = 10000
num_stat = 664
num_trucks = 5
norm_factor = 0.148
load_rate = 0.005
plot_animation = False
save_animation = False
last_num_hours = 200 # only look at this many of the last hours

# pick one of these to iterate over
lambda_bike = [0.0139, 0.0187, 0.0227] # lowest month, september, highest month
truck_cap = 10
fixed_cost_truck = 60 # dollars per hour per truck
distance_cost_truck = 5 # dollars per kilometer
time_of_day = 'afternoon' # pick between 'night','morning','noon','afternoon','evening'

# plotting style parameters
rcParams['grid.alpha'] = 15
rcParams['grid.color'] = 'k'
rcParams['legend.edgecolor'] = 'k'
rcParams['legend.facecolor'] = 'w'

# initializing plot
f, axarr = plt.subplots(2, 2)
cluster_cols = ['blue','red','black','green','gray'] # colors to distinguish different runs
counter = 0
for n in lambda_bike: # change to whichever one you are iterating over
        # list to store statistics
        total_distance_list = []
        perc_idle_time_list = []
        perc_good_bikes_list = []
        cost_list = []
        for m in range(num_trucks):
                # change name in simulation call to n for the variable that you want to iterate over
                times,bikes_avail,bikes_in_stations,broken_bikes,delta_distance,clusters =
                simulation(num_stat,m+1,n,norm_factor,load_rate,truck_cap,time_of_day,
                        num_steps,plot_animation,save_animation)
```

```
                # finding step number that starts last hour of simulation run
                index = next((i for i, v in enumerate(times) if v>=(times[-1]-last_num_hours)),-1)
                # statistics (last hours of run)
                delta_distance_concat = [delta_distance[i][index:] for i in range(m+1)]
                total_distance = [sum(i)/(last_num_hours*(m+1)) for i in delta_distance_concat]
                perc_idle_time = [sum([times[i+1] - times[i] for i in range(index,num_steps) if
                        delta_distance[j][i] == 0])*100 / last_num_hours for j in range(m+1)]
                total_broken_bikes = sum([broken_bikes[i][-1] for i in range(m+1)])
                perc_good_bikes = 1 - (total_broken_bikes / sum(bikes_in_stations))

                # cost per hour single run
                cost = (fixed_cost_truck + sum(total_distance)*distance_cost_truck)*(m+1)

                # storing statistics in list
                total_distance_list.append(sum(total_distance))
                perc_idle_time_list.append(np.mean(perc_idle_time))
                perc_good_bikes_list.append(perc_good_bikes)
                cost_list.append(cost)

        # plotting results
        axarr[0, 0].plot([i+1 for i in range(num_trucks)],total_distance_list,'-o',color = cluster_cols[
            counter],label = 'breakdown rate = ' + str(n))
        axarr[0, 1].plot([i+1 for i in range(num_trucks)],perc_idle_time_list,'-o',color =
            cluster_cols[counter],label = 'breakdown rate = ' + str(n))
        axarr[1, 0].plot([i+1 for i in range(num_trucks)],perc_good_bikes_list,'-o',color =
            cluster_cols[counter],label = 'breakdown rate = ' + str(n))
        axarr[1, 1].plot([i+1 for i in range(num_trucks)],cost_list,'-o',color = cluster_cols[counter],
            label = 'breakdown rate = ' + str(n))
        counter += 1

# setting x-axis ticker intervals
loc = plticker.MultipleLocator(base=1.0) # this locator puts ticks at regular intervals
axarr[0,0].xaxis.set_major_locator(loc)
axarr[0,1].xaxis.set_major_locator(loc)
axarr[1,0].xaxis.set_major_locator(loc)
axarr[1,1].xaxis.set_major_locator(loc)

# setting axis labels
axarr[0, 0].set_xlabel('# Trucks')
axarr[0, 1].set_xlabel('# Trucks')
axarr[1, 0].set_xlabel('# Trucks')
axarr[1, 1].set_xlabel('# Trucks')
axarr[0, 0].set_ylabel('Distance [km]')
axarr[0, 1].set_ylabel('Idle Time [%]')
axarr[1, 0].set_ylabel('% Functioning Bikes')
axarr[1, 1].set_ylabel('Total Cost [$ / hour]')

# setting titles for plots
axarr[0, 0].set_title('Total Distance Traveled per Truck [km]')
axarr[0, 1].set_title('Idle Time [%]')
axarr[1, 0].set_title('Percentage of Functioning Bikes to Total Number of Bikes')
axarr[1, 1].set_title('Total Cost [$ / hour]')

# setting legends
```

```
axarr [0,0]. legend(loc='upper right')
axarr [0,1]. legend(loc='upper left')
axarr [1,0]. legend(loc='lower right')
axarr [1,1]. legend(loc='lower right')

# turning on grid
axarr [0,0]. grid()
axarr [0,1]. grid()
axarr [1,0]. grid()
axarr [1,1]. grid()

plt.show()
```