Kapil Sinha
10/24/16

Ph 11 Hurdle 1

Problem statement

In large cities with Bike-Share, bikes tend to concentrate at certain bike stations (sinks) and have low numbers in other stations (sources).

On average, 93.992% of a stocked-out station's demand is lost (only 6.008% of its unserved users substitute to other stations) (Kabra, Belavina, Girotra). In Paris, the Bike-Share system encourages users to return bikes to these "sources" by offering 15 minutes of riding credit, but its bike redistribution costs keep increasing.

We must find a way to minimize this source-sink problem.

Literature Review/Past Research

Bike-share in its current form is quite new, possible only with the arrival of technology that allows for extensive data gathering from each station and user-friendliness on the consumer's part.

Hence, research on bike-share is relatively new. As bike-share has become more popular, much of the research are feasibility studies and other such studies examining demand, some of which compare it to pricing (Frade and Ribeiro, "Bike-Share Planning Guide"). These papers focus largely on the business strategy, such as system sizing, public relations, and cost-benefit analysis. Many of these are very specific to certain locations and concentrate a lot on infrastructure. Since in our model, we make a lot of simplifications, much of this is not very relevant. More importantly, none of these studies propose solutions to the source-sink problem. In fact, the Washington Bike-Share system accepts losing money through its system, much of which is caused by the station "rebalancing operations" (Kurtzleben).

In Kabra et al., the authors aimed to estimate the impact of accessibility and availability of bike-share stations, developing sophisticated models and simulations. Nevertheless, the authors do not at all focus on the source-sink problem and in fact only mention once that "system managers regularly transfer bikes from full stations to empty ones." Thus I drew heavily from their physical measurements to make reasonable assumptions about the city and I incorporated their concepts of accessibility (which I use as Manhattan distance) and availability (which I use as average bike availability) in my model. However, their model accounted for many complicated variables and used certain algorithms that I did not understand. While the paper was very useful for a background on the bike-share system, it did not provide any insight on the source-sink problem itself. (Kabra et al.)

Hence, I did not get much insight from past bike-share studies. Instead, I drew from my knowledge of a similar business but different logistical problem – rideshare. Rideshare businesses like Uber often employ simple supply and demand principles through surge pricing, changing price based on supply by drivers and demand by riders ("What is surge?"). This principle can be translated to the bikeshare paradigm with no change to the bikeshare station infrastructure because it uses the same pay stations; this makes surge/dynamic pricing an attractive and assimilable solution.

Methods/Understanding how to proceed

We can rudimentarily think of this problem as an optimization problem. First we must define our *objective*. Are we attempting to...

1. Maximize profit?
2. Maximize bike usage?
3. Minimize amount (and thus cost) of relocating bikes from sinks to sources?

We see that #1 is the goal of a purely profit-maximizing firm and #2 is the goal of a purely philanthropic institution. As city planners, we likely are not consumed by greed (Kurtzleben) and

also do not have an unlimited budget. Because of the specific nature of our problem, we focus on #3 but also try not to greatly decrease bike usage.

Now we must define our *constraints* i.e. what we can change to alter bikers' behavior. Thus we list potentially relevant variables and make reasonable judgments as to what we can control. Relevant variables:

- Map of the city – size of city, location and popularity of certain points of interest (we define points of interests as any place that attracts a large number of people at a given time e.g. tourist spots, companies, restaurants, etc.)
- Location of all people and population size
- Location and number of Bike-Share users
- Location and number of stations
- Price of renting a Bike-Share bike

Obviously we cannot change the city layout or the number and location of individuals. For this study, we will not change the location and number of stations either; in general, stations are permanent and expensive to install and extensive research is done in studies planning bike-share in cities to place them in accessible places. We leave the intelligent placement of stations for further research. We can, however, control the price of renting *and* dropping off of a Bike-Share bike.

Simplifications/General Assumptions:

1. "Effective population" consists of only Bike-Share users.
    1. Explanation: We are mainly only considered with the population of Bike-Share users in our model (e.g. if the actual population is 2 million and 25% of the population are Bike-Share users, then the effective population is 0.5 million)
    2. Purpose: This will allow us to ignore the rest of the population (who have minimal impact on Bike-Share) for our model
2. There is no influx or outflux of Bike-Share users in the city
    1. Explanation: The number of Bike-Share users is likely stable so this mainly decreases randomness in the location of users
    2. Purpose: A constant number of users will allow us to ignore new users arriving at certain locations (such as through transits) and users leaving the city, reducing complexity
3. Individual Bike-Share stations can be treated as though they are in monopolistic competition
    1. Explanation: Bike-Share stations can be represented as spatially differentiated products (Kabra et al.)
    2. Purpose: We can use the straight line demand curve for monopolistic competition, allowing us to predict changes in demand and user behavior based on changes in price

To start with a simple model, we will just create our own city (excluding the suburbs), which we name Paristopia, due to its idealistically simple layout and its similarities to Paris (see *Appendix A* for some select statistics of Paris and its Bike-Share). We will define some of its characteristics:

- Paristopia is a 10 km x 10 km area ("Key Figures") with 10 m x 10 m squares as its smallest component i.e. it is not a continuous distribution
    - This allows us to simplify the model without significantly affecting the model
- The effective population (of Bike-Share users) is 245,000 (Barz)
- Paristopia has 1261 stations, 16393 bikes (each station starts with 13 bikes), 40352 terminals (each station can hold 32 bikes) ("Statistiques")
- There is a grid of 34 x 34 bike stations (represented as a point), each one 300 m away from the next (Barz), and the remaining 105 stations are positioned randomly
- The <u>average</u> number of trips per hour (see *Appendix A* for average number of trips per station per minute and per hour), using 1261 stations, is (Kabra):

AM	PM
12 – 1: 3783	12 – 1: 6053
1 – 2: 2648	1 – 2: 6053
2 – 3: 1892	2 – 3: 6053
3 – 4: 1135	3 – 4: 6053
4 – 5: 946	4 – 5: 6809
5 – 6: 757	5 – 6: 8701
6 – 7: 946	6 – 7: 10592
7 – 8: 3026	7 – 8: 9836
8 – 9: 7566	8 – 9: 7188
9 – 10: 6053	9 – 10: 4918
10 – 11: 4540	10 – 11: 4540
11 – 12: 5296	11 – 12: 4540

Note that the total number of trips in a day is approximately 120,000, which is very close to the number for Paris ("Key Figures")

- Bike users do not move. In reality, everyone is obviously constantly moving around but this simplification does not affect our results much because bikers probably are spread relatively uniformly.
- There are several points of interest in the city, each of which we attribute a "popularity" number that is a factor of how many people want to come to it (the stations near these points are "sinks"); the following are arbitrary points of interest chosen for Paristopia
    - Note that the below popularity scores are arbitrary and unitless and so only the relative amounts matter; they are ultimately used to calculate probability, a proportion.
  - 80,000 10m x 10 m squares (each with popularity score 1) representing small stores, etc.
  - 4,000 30m x 30m squares (each with popularity score 10)
  - 200 50m x 50m squares (each with popularity score 100
  - 10 70m x 70m squares (each with popularity score 1,000)
  - 1 90m x 90m square (popularity score 5,000) – this is our Eiffel Tower
- A bike-share user's decision on which station to rent a bike from depends on:
  - (Manhattan) distance to a station (a measure of accessibility), price of renting from the station, average bike availability (a measure of reliability) in the station's area
    - (Walking) Manhattan distance: For the first 300 meters, every additional meter of walking decreases a user's likelihood of renting from that station by 0.252%; after that, every additional meter decreases the likelihood by 1.367% (Kabra)
    - Price: The higher the price, the less likely a user will rent from that station according to supply and demand (we model this as monopolistic competition – see below)
    - Average bike availability: A user is less likely to rent from a station that historically has had few bikes. The immediate loss of sales is the short-term effect of low availability, which is the concept we account for in our model. In addition, consistently low numbers of bikes at a station decreases the popularity and reliability of stations, which is a long-term effect of low availability (Kabra). We ignore this long-term effect as it has less impact than the short-term effect and is more difficult to model. We say that the likelihood that a bike-share user rents from a station decreases linearly starting from when it has fewer than 5 bikes.
- A bike-share user's decision on which station to drop a bike to depends on:
  - (Manhattan) distance to the dropoff station, price of the dropoff station, average bike availability at the dropout station, and popularity of the area surrounding the station

- (Biking) Manhattan distance: The average distance ridden on a single trip, using 120,000 trips per day, is 2 km (Melvin). We essentially use the same model as the Manhattan distance to a rental station (see above), multiplying the 300 meters by 10 and compensating by dividing the likelihood decreases per meter by 10 (i.e. for the first 3000 meters, every additional meter of walking decreases a user's likelihood of renting from that station by 0.0252%; after that, every additional meter decreases the likelihood by 0.1367%
- Price: Same concept as above but for dropoff stations
- Average bike availability: We say that the likelihood that a bike-share user rents from a station decreases linearly starting from when it has more than 27 bikes (out of 32).
- Popularity: A user is more likely to go to a dropoff station if the surrounding area is more "popular" i.e. there are more popular points of interest in the area. We combine the walking Manhattan distance and the points of interest popularity scores (see above)

Price:

The independent variable in this study is price of renting and dropping off bikes at stations.

To create a basic demand model for prices and quantity demanded, we research prices of existing Bike-Shares. For simplicity, we ignore the monthly and annual passes that are common throughout Bike-Shares, we assume that the average user takes four trips on an unlimited day pass, and that a trip has a 30 minute maximum, which is a well-established standard. The Metro Bike-Share in downtown Los Angeles charges $3.50 for a 30 minute trip, Citi Bike-Share charges $3 per trip, Bay Area Bike-Share charges $2.25 per trip, Santa Monica Bike-Share charges $3.50 per trip, and Paris Vélib' (up until 30 minutes) is free ("Bay Area Bike Share", "Citi Bike", "Paris Vélib'", "Metro Bike Share", "Santa Monica Bike Share").

We assume [general assumption 3] that since bike-share stations are spatially differentiated products (Kabra et al.), we can treat it as monopolistic competition. We adjust the linear supply/demand curve, replacing quantity (of a station) demanded with the probability that a user uses a station (effectively replacing quantity demanded of a population with probability that a station is demanded by an individual).

Moreover, to address the source-sink problem, we decide that instead of charging at a station where the user picks up the bike, we will charge at both the station where the user picks up the bike and the station where the user drops off the bike (e.g. instead of charging 3 dollars at the rental station, we could charge approximately $1.50 at the rental station and $1.50 at the dropoff station. This allows us to adjust prices to not only solve the "source" problem but also the "sink problem."

To adjust the price to affect the rental and dropoff of bikes of stations, we have two relatively simple main options. 1) We can measure the rental and dropoff rates of change per station after a certain number of users have moved or after a unit of time; then we can calculate the appropriate changes in dropoff and rental prices per station. 2) After every user transaction in a particular station, we adjust dropoff and rental prices of that station based on a simple function.

Because there are so many stations (many of which are not near points of interest), determining the rate of change in number of bikes per station may be difficult. Moreover, calculating the appropriate change in price given the station's other qualities may be difficult and not easily extensible to real-life scenarios where we do not have numbers for popularity in the surrounding area of a station. We thus choose the second option, as it is easier to model and is easily applicable to real-life situations because it can be used by existing stations without changing the bike-share infrastructure (i.e. a network of digital pay-stations that can keep track of the number of bikes in each station).

Then we consider cases: For renting a bike from a station, the price is highest at 1 bike (close to $3.00 since this is the maximum price we can charge a user such that the user still has any probability of renting the bike, based on our supply/demand curve), $1.50 for 13 bikes (the starting

price and starting number of bikes in a station), and close to free at 32 bikes (full station). Similarly, for dropping off a bike at a station, the price is highest at 31 bikes (close to $3.00), $1.50 for 13 bikes, and close to free at 0 bikes (empty station). We connect these points using a kinked curve.

After running the program simulations a few times, we realized that we needed to increase the price of dropping off a bike at a station after the program reaches a certain number of bikes in a given station because popular stations were being filled very quickly since users were willing to pay the increased amount so that they could go to those stations.

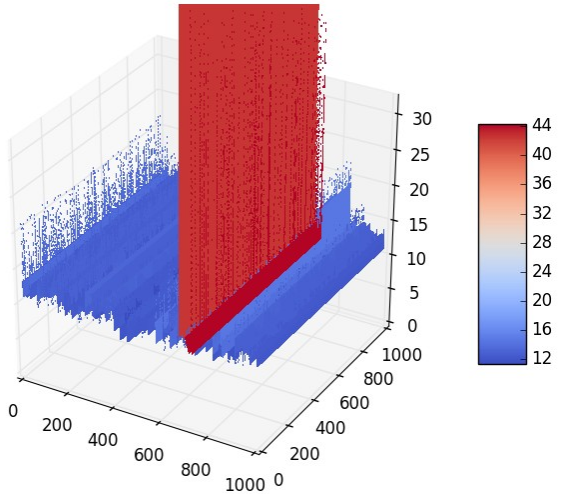
See *Appendix B* for the code of my simulation program.

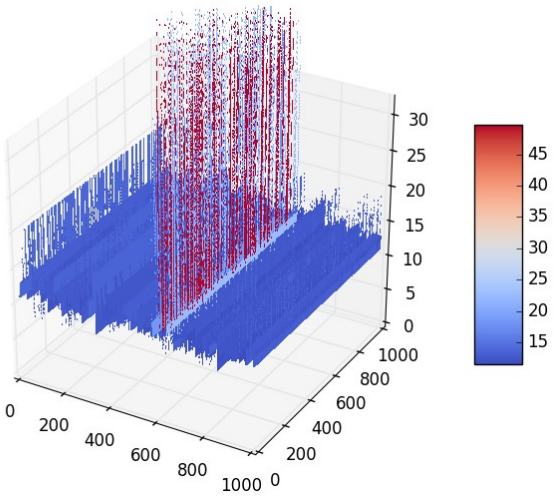
Results & Discussion

Please vist https://github.com/kapilsinha/Ph11-Hurdle-1 to see all the results from my simulations as well as the source code.

First, we note that the simulations run were done on 100 users instead of the thousands that were intitially modeled since the simulation is very time and computationally intensive. This still lets us see the general trends, but it prevents us from being fully confident that our solution works. Luckily, we simply need a more powerful computer to run the simulation in order to get more data and draw our conclusions.

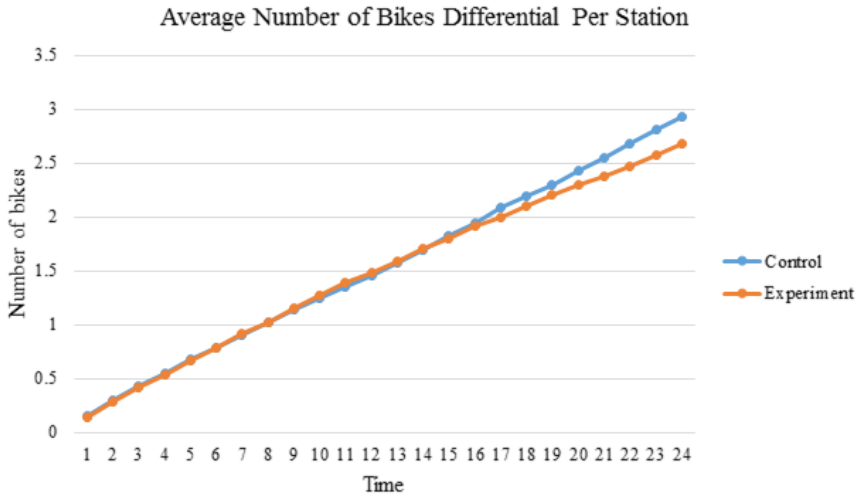
We see the below graphs (x and y represent the position of the station, and z represents the number of bikes in that station), with the control (without altering the price of stations) on the left side and the experiment (altering the price of stations) on the right side:





Though the graphs are complex, we make a few key observations: 1. The “sink” problem is far greater than the “source” problem. Since some areas with points of interest in the vicinity are so much more popular than others, bikers are drawn to those places fairly uniformly from nearby places. This can be seen with the red spike in both graphs. 2. The sink problem in the very popular locations is slightly alleviated in the experiment section since the peak is not nearly as prominent (note that there are several stations surrounding the very popular points of interest). 3. In the given time period and the number of users, the control and experiment results are quite similar. Only the stations surrounding the most popular location (Paristopia’s Eiffel Tower) are filled, while the rest are close to the initital number of bikes. Another simulation with the full number of users and perhaps a longer period of time can shed more light on the differences between the experiment and the control.

To see the more minute differences between the experiment and control simulations and to make predictions of their behavior with more users, we look at the graphs below:



These graphs represent the average differential (difference) in initial number of bikes of a station (13) and its current number of bikes. It is a measure of the source-sink problem; the higher the number, the greater the disparity in number of bikes between the stations and so the greater the source-sink problem. We see that the experiment and control differentials increase at approximately the same rate, but near the end, the experiment simulation's differential is slightly lower and appears to be growing at a lower rate than the control simulation's differential. Extrapolating this trend, we can see that over time, the differential in the experiment group would likely be significantly lower than that in the control group; a greater number of simulations is required to draw this conclusion.

Regarding the very popular points of interest (e.g. the Eiffel Tower), it is unlikely to be able to solve the sink problem in that location with solely prices because users' desire to travel to these places is so great. Moreover, increasing price too much to reduce users' travel is likely counterproductive as it excessively restricts use of the station. Another solution specifically for such popular locations is to deliberately add more stations to the area. In our model, we placed most of our stations in a grid and randomly placed the remainder; in reality, however, extensive research is done in the effort to maximize usage of stations. Adding more stations is likely a feasible solution that can be tested in the future.

Now we note that in our simulation, we did not move the users after they rented and dropped off a bike as a simplification. In reality, however, there would be a greater number of bikers at points of interest, and so price reductions of rental bikes in those locations would be more likely to draw these bikers to rent a bike to return home. As this was not accounted for in our model, we can reasonably claim that some of the benefits of altering price in popular areas would be greater in reality than is apparent in the model, for this reason.

Also, it is worth noting that our model of altering prices is likely to make some users unhappy in the short run as it opposes their preferred paths with increased costs. In the long run, however, the users will be satisfied if they realize the benefits gained of having evenly distributed bikes. Looking at consumer reactions to rideshare surge pricing though, perhaps these long-term benefits will not be so apparent. Nevertheless, lower costs of relocating bikes outweighs mild consumer dissatisfaction.

We conclude that adjusting prices of stations does help alleviate the source-sink problem and it is a valid solution to the problem.

Further Research

In regards to our model itself, there certainly are more optimal ways to change price. In particular, machine learning algorithms that change price of stations seem like a good avenue to explore that was not approached in this study due to its complexity and time constraints. Moreover,

it would be interesting to note the overall profit changes that are associated with changing prices of stations; perhaps the model can then be optimized so that in addition to addressing the source-sink problem the city makes a profit.

As stated above, adjusting locations and number of stations in addition to controlling price of stations would likely have an even bigger impact on the source-sink problem. Of course, since this requires potentially expensive changes in the infrastructure, there must be extensive research conducted on the cost-effectiveness of these changes.

Appendix A: An Annotated Compilation of Miscellaneous Facts of Paris and its Bike-Share

http://www.parisavelo.net/stats.php ("Statistiques")

- Number of stations: 1261
- Number of bikes: 16612
- Number of free terminals: 23511
- Number of terminals: 41295
- Number of full stations: 88
- Number of empty stations: 168

https://www.thelocal.fr/20140715/velib-seven-stats-for-bikes-7th-birthday (Melvin)

- In the past seven years Velib' users have ridden 614 million kilometres on the ubiquitous grey machines. --> 240,000 km per day
- Velib's busiest time of the day is evening rush-hour, 6pm to 7pm, accounting for 9.2 percent of all rentals.
- Velib's slowest time is from 5-6am, accounting for 0.5 percent of rentals.
- Of Paris's 30 million tourists per year, about 2.5 million of them rent out a Velib' to wander the City of Light. --> 8.33%

https://urbanistinparis.wordpress.com/2013/04/15/show-me-the-data-or-stuff-ive-read-about-velib-part-4-of-4/ (Barz)

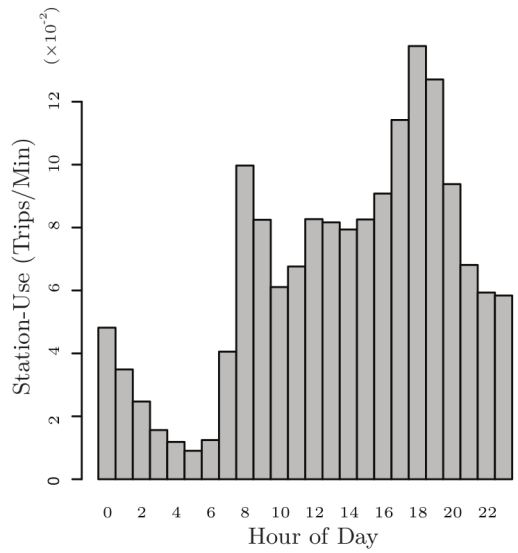
- Stations are an average of 300-meters apart
- In 2008, the first full year of operation, annual Vélib' rentals totaled 27.9 million (!) and then proceeded to decline through 2010. In 2011 though, there was a major uptick! Over 31.3 million bicycle rentals took place in 2011
- After these new offers targeting youth and longer rides went into effect, 22% of subscribers were identified as youth aged 16-25, an increase from 14% of subscribers in 2009, and 43% of subscribers were found to live in the near suburbs served by Vélib' as opposed to 36% of subscribers in 2009.
- Before 2011, users could become annual subscribers at a price of 29€ per year. In April 2011, Vélib' diversified the subscription choices to include the Classique for 29€ per year (unlimited 30-minute trips — the same as the original long-term subscription) and the Passion for 39€ per year (unlimited 45-minute trips) as well as price reductions of 10-20€ for youth aged 14-26, students, and teachers.

https://urbanistinparis.wordpress.com/2013/01/29/stuff-ive-read-about-velib-part-1-of-3/ (2013) (Barz)

- 245,000 annual subscribers
- Since 2007, Vélib' riders have logged 130 million trips (2007 – Jan 2013) --> 60,000 trips/day on average
- Because the bike is used by around 10-12 different people a day, it lives in the street, and it travels around 10,000 km a year
  http://www.pbs.org/e2/episodes/308_paris_velo_liberte_trailer.html

http://next.paris.fr/english/presentation-of-the-city/key-figures-for-paris/rub_8125_stand_29918_port_18748 ("Key Figures")

- Paris is 10,539 hectares --> approximately 100 square kilometers
- 110,000 journeys on Vélib bicycles a day



(Kabra)

Estimate of the number of trips/minute of a station by time:

AM	PM
12 – 1: 0.05	12 – 1: 0.08
1 – 2: 0.035	1 – 2: 0.08
2 – 3: 0.025	2 – 3: 0.08
3 – 4: 0.015	3 – 4: 0.08
4 – 5: 0.0125	4 – 5: 0.09
5 – 6: 0.01	5 – 6: 0.115
6 – 7: 0.0125	6 – 7: 0.14
7 – 8: 0.04	7 – 8: 0.13
8 – 9: 0.10	8 – 9: 0.095
9 – 10: 0.08	9 – 10: 0.065
10 – 11: 0.06	10 – 11: 0.06
11 – 12: 0.07	11 – 12: 0.06

Total: (1.585 trips/minute)*60 minutes = 95 trips/(day x station)

Thus the number of trips/hour of a station by time:

AM	PM
12 – 1: 3	12 – 1: 4.8
1 – 2: 2.1	1 – 2: 4.8
2 – 3: 1.5	2 – 3: 4.8
3 – 4: 0.9	3 – 4: 4.8
4 – 5: 0.75	4 – 5: 5.4
5 – 6: 0.6	5 – 6: 6.9
6 – 7: 0.75	6 – 7: 8.4
7 – 8: 2.4	7 – 8: 7.8
8 – 9: 6	8 – 9: 5.7
9 – 10: 4.8	9 – 10: 3.9
10 – 11: 3.6	10 – 11: 3.6
11 – 12: 4.2	11 – 12: 3.6

Appendix B: Code for Simulation Program

To view the code online, visit https://github.com/kapilsinha/Ph11-Hurdle-1 (and more specifically, https://github.com/kapilsinha/Ph11-Hurdle-1/blob/master/Ph11_hurdle1_general_version.py)

```
#Kapil Sinha
#10/24/16
#Ph 11 Hurdle 1

import random
import numpy
import math
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
import matplotlib.pyplot as plt

INITIAL_NUM_BIKES = 13 #number of bikes per station initially
INITIAL_PRICE = 1.50
NUM_TERMINALS = 32 #constant number of terminals in each station

def generateCity():
    '''Creates the city: 1000x1000 Point array with random Points assigned popularity values, 34x34 Stations with 105 extra randomly placed Stations, and 245,000 randomly placed people'''
    city_map = [[Point(x,y) for x in range(1000)] for y in range(1000)]
    #2-D array of the city map where each Point is a 10m x 10m square
    assignPopularity(city_map) #set popularity values for random Points
    for i in range(5,996,30): #place 34x34 stations spaced by 30 (x 10 m)
        for j in range(5,996,30):
            city_map[i][j].station = Station(i,j)
            city_map[i][j].incrementNumStations()
    for k in range(105): #randomly place remaining 105 stations
        x = random.randint(0,999)
        y = random.randint(0,999)
        city_map[x][y].station = Station(x,y)
        city_map[x][y].incrementNumStations()
    for i in range(245000): #randomly place 245,000 people (Bike-Share users)
        x = random.randint(0,999)
        y = random.randint(0,999)
        city_map[x][y].userlist.append(User(x,y))
        city_map[x][y].incrementNumPeople()
    return city_map

def assignPopularity(city_map):
    '''Assigns popularity values to randomly chosen locations such that there are 80,000 Points with popularity 1, 4,000 3 x 3 Point squares with popularity 10, 200 5 x 5 Point squares with popularity 100, and 10  7 x 7 Point squares with popularity 1,000, and 1 9 x 9 Point square with popularity 5,000'''
    #Note: we are adding these popularity values to Points with replacement; that is, a Point that is assigned a certain popularity may be overrided by the next statement; however, since we are assigning increasing popularity values and some randomness is acceptable, this should not cause any errors
    for a in range(80000): #(Approximately) 80,000 10m x 10m squares with popularity 1
        x = random.randint(0,999)
        y = random.randint(0,999)
        city_map[x][y].popularity = 1
    for b in range(4000): #4,000 30m x 30m squares with popularity 10
        x = random.randint(1,998)
        y = random.randint(1,998)
        for k in range(x-1,x+2):
            for l in range(y-1,y+2):
                city_map[k][l].popularity = 10
    for c in range(200): #200 50m x 50m squares with popularity 100
        x = random.randint(2,997)
        y = random.randint(2,997)
```

```
        for m in range(x-2,x+3):
            for n in range(y-2,y+3):
                city_map[m][n].popularity = 100
    for d in range(10): #10 70m x 70m squares with popularity 1000
        x = random.randint(3,996)
        y = random.randint(3,996)
        for o in range(x-3,x+4):
            for p in range(y-3,y+4):
                city_map[o][p].popularity = 1000
    for e in range(1): #1 90m x 90m square with popularity 5,000 i.e. Eiffel Tower
        x = random.randint(4,997)
        y = random.randint(1,998)
        for q in range(x-4,x+5):
            for r in range(y-4,y+5):
                city_map[q][r].popularity = 5000

def numTripsinHour(t):
    """Description: Returns the total number of trips in an hour given the time
    Input Argument: time t of day (0-24)
    Return Value: int"""
    if t >= 0 and t < 1:
        return 3783
    if t >= 1 and t < 2:
        return 2648
    if t >= 2 and t < 3:
        return 1892
    if t >= 3 and t < 4:
        return 1135
    if t >= 4 and t < 5:
        return 946
    if t >= 5 and t < 6:
        return 757
    if t >= 6 and t < 7:
        return 946
    if t >= 7 and t < 8:
        return 3026
    if t >= 8 and t < 9:
        return 7566
    if t >= 9 and t < 10:
        return 6053
    if t >= 10 and t < 11:
        return 4540
    if t >= 11 and t < 12:
        return 5296
    if t >= 12 and t < 13:
        return 6053
    if t >= 13 and t < 14:
        return 6053
    if t >= 14 and t < 15:
        return 6053
    if t >= 15 and t < 16:
        return 6053
    if t >= 16 and t < 17:
        return 6809
    if t >= 17 and t < 18:
        return 8701
    if t >= 18 and t < 19:
        return 10592
    if t >= 19 and t < 20:
        return 9836
```

```
    if t >= 20 and t < 21:
        return 7188
    if t >= 21 and t < 22:
        return 4918
    if t >= 22 and t < 23:
        return 4540
    if t >= 23 and t < 24:
        return 4540
    if t>=24 and t < 25:
        return 50 # This is done purely for testing purposes (running this program over t=0 to t=24 is computationally
heavy)
    else:
        raise ValueError('Time must be between 0 and 24')

def incrementTime(t):
    '''Increases t by 1 hour unless it is between 23 and 24, in which case it returns a number between 0 and 1 (goes to the
next day)'''
    if t >= 0 and t <23:
        return t+1
    elif t >= 23:
        return (t+1)-24
    else:
        raise ValueError('Time cannot be negative')

def findBikingUsers(city_map, t):
    '''Returns a random subset of Users of a size depending on t. We can randomly choose n people because our time step
is 1 hour and so we don't need to consider distance from a station as a factor of WHETHER a user will rent a bike (as
was done in literature with a kinked curve); moreover, we evenly spaced out most of the stations, so distance from
stations is relatively constant. Instead, the distance from a station will be a factor of WHICH station the biker will rent
from'''
    user_list = [] #list of all users
    for x in range(1000):
        for y in range(1000):
            user_list += city_map[x][y].userlist
    return random.sample(user_list, numTripsinHour(t))
    # return a random sample of size numTripsinHour(t) of all the Users

def calcManhattanDistance(x_origin,y_origin,x_destination,y_destination):
    '''Manhattan distanc is a factor of the likelihood that a user picks up a bike'''
    return abs(x_origin - x_destination) + abs(y_origin - y_destination)

def calcProbabilityBikeAvailability(bike_availability, isRentStation):
    '''Calculates the probability that a certain user will use a station depending on the average bike availability, using a
basic kinked function. Will be used in calcRentStation() and calcDropOffStation()
bike_availabliity - integer returned from averageBikeAvailability()
isRentStation - True if used in calcRentStation, False if used in calcDropOffStation'''
    if isRentStation == True:
        if bike_availability >= 5:
            return 1.0 #100% probability i.e. bike availability is not a factor in the user's decision-making if there are more
than 5 bikes in a station
        else:
            return 0.2*bike_availability # if there are fewer than 5 bikes in that area, the probability that a user will go to
that station decreases linearly (such that if there are 0 bikes on average, there is no chance that the user will go there to
rent a bike
    else: #if DropOffStation
        if bike_availability <= 27: #5 less than the number of terminals
            return 1.0
        else: #if the average station in the area is almost full, the chance a user will drop off a bike there decreases
            return 0.2* bike_availability
```

```
def calcProbabilityManhattanDistance(manhattan_distance, isWalkingDistance):
    '''Calculates the probability that a certain user will use a station depending on the walking Manhattan distance to the station (for renting a bike) and biking Manhattan distance to a station (for dropping off a a bike)'''
    if isWalkingDistance == True:
        if manhattan_distance < 300:
            return 1 - (.00252 * manhattan_distance)
        else:
            return max(0,1 - (.00252 * manhattan_distance + .01367 * (manhattan_distance - 300))) #zero probability of walking to a station over 315 m away
    else: #isBikingDistance #WRITE LOGIC FOR THESE NUMBERS IN THE PAPER
        if manhattan_distance < 3000:
            return 1 - (.000252 * manhattan_distance)
        else:
            return max(0,1 - (.000252 * manhattan_distance + .001367 * (manhattan_distance - 3000))) #zero probability of biking to a station over 3150 m away

def calcProbabilityPopularity(city_map,station):
    '''Calculates the probability that a certain user will use a station depending on the surrounding points of interest. Uses the Manhattan distance calculation from calcProbailityManhattanDistance() function to find the popularity in a certain distance from the station (because the biker then has to walk to the point of interest)'''
    popularity_in_area = 0
    #we could cycle through the entire city_map but we know from our previous Manhattan distance assumptions that no user will walk more than 315 m (rounding up to 320 m) to arrive at his/her destination so this makes our computation faster
    for i in range(max(0,station.x-32),min(station.x+33,999)):
        for j in range(max(0,station.y-32),min(station.y+33,999)):
            popularity_in_area += (calcProbabilityManhattanDistance(calcManhattanDistance(station.x,station.y,i,j), True)*city_map[i][j].popularity)
    return popularity_in_area

def calcProbabilityPrice(price): #price of picking up/dropping off at a station is in dollars
    '''Returns the probability that a user will use a certain station based on price. We extend the supply/demand curve for monopolistic competition (a straight line) such that probability (replacing quantity demanded) is 1 at 0 dollars and 0 at 3 dollars per station (note that we are charging at both the rental station and the dropoff station).'''
    if (1 - price/3) < 0:
        return 0
    return 1 - (price/3)

def calcRentStation(city_map, user, station_list):
    '''Determines which station a biking user will rent from. Uses Manhattan distance, average bike availability, and price'''
    station_probability = [] #We are creating a list of probabilities whose index matches those of the stations; a dictionary may be more appropriate here but the following is simple for a list
    for i in range(len(station_list)):
        bike_availability_probability = calcProbabilityBikeAvailability(station_list[i].averageBikeAvailability(city_map),True)
        manhattan_distance_probability = calcProbabilityManhattanDistance(calcManhattanDistance(user.x,user.y,station_list[i].x,station_list[i].y), True)
        price_probability = calcProbabilityPrice(station_list[i].rental_price)
        station_probability.append(bike_availability_probability * manhattan_distance_probability * price_probability)
    station_probability_normalized = normalizeList(station_probability)
    rent_station = numpy.random.choice(station_list, 1, p=station_probability_normalized)[0] #randomly choose a station based on the calculated probabilities (allows for randomness that would exist in the real world with the necessary constraints)
    return rent_station

def calcDropOffStation(city_map, user, station_list):
    '''Determines at which station a biking user will drop off a bike. Uses popularity, Manhattan distance, average bike availability, and price'''
```

```
    station_probability = [] #We are creating a list of probabilities whose index matches those of the stations (like in calcRentStation)
    for i in range(len(station_list)):
        bike_availability_probability = calcProbabilityBikeAvailability(station_list[i].averageBikeAvailability(city_map),False)
        popularity_probability = calcProbabilityPopularity(city_map,station_list[i])
        manhattan_distance_probability = calcProbabilityManhattanDistance(calcManhattanDistance(user.x,user.y,station_list[i].x,station_list[i].y), False)
        price_probability = calcProbabilityPrice(station_list[i].dropoff_price)
        station_probability.append(bike_availability_probability * manhattan_distance_probability * price_probability * popularity_probability)
    station_probability_normalized = normalizeList(station_probability)
    dropoff_station = numpy.random.choice(station_list, 1, p=station_probability_normalized)[0] #randomly choose a station based on the calculated probabilities
    return dropoff_station


def normalizeList(lst):
    '''Helper function used in calcRentStation() and calcDropOffStation(). Returns a list whose elements add up to 1. Divides every element in a list by the sum of the list'''
    lst_sum = 0
    for i in range(len(lst)):
        lst_sum += lst[i]
    lst_normalized = []
    for j in range(len(lst)):
        lst_normalized.append(float(lst[j])/float(lst_sum))
    lst_normalized_sum = 0
    for k in range(len(lst_normalized)):
        lst_normalized_sum += lst_normalized[k]
    return lst_normalized

def getStationList(city_map):
    '''Returns a list of all the stations. Used in calcRentStation() and calcDropOffStation().'''
    station_list = [] #list of all stations
    for x in range(1000):
        for y in range(1000):
            if city_map[x][y].num_stations > 0:
                station_list.append(city_map[x][y].station)
    return station_list

def adjustStationPrice(station, isRentBike):
    '''Adjusts price of dropping off and renting a bike from a station based on the number of bikes. The actual numbers that have been created as constants i.e. INITIAL_NUM_BIKES, INITIAL_PRICE, and NUM_TERMINALS have been used here so that the functions are easier to calculate and understand'''
    if isRentBike == True:
        if station.num_bikes < 13:
            station.price = (-1.5/13)*(station.num_bikes)+3
        elif station.num_bikes >= 13 and station.num_bikes < 18:
            station.price = (-1.5/19)*(station.num_bikes)+((32*1.5)/19)
        else: #num_bikes >= 18
            station.price = -(1.0/28)*(station.num_bikes-18)+0.5
    else: #isRentBike == False
        if station.num_bikes < 13:
            station.price = (1.5/13)*(station.num_bikes)
        elif station.num_bikes >= 13 and station.num_bikes < 20:
            station.price = (1.5/19)*(station.num_bikes)+(3-((1.5*32)/19))
        else: #num_bikes >= 20
            station.price = (1.0/280)*(station.num_bikes-18)+2.95

def visualizeCity(station_list,t):
```

```
    '''Plots a 3-D graph of the x and y values of the station along with the number of bikes on the z axis'''
    X = [] #x_list
    Y = [] #y_list
    Z = [] #num_bikes_list
    for station in station_list:
        X.append(station.x)
        Y.append(station.y)
        Z.append(station.num_bikes)
    fig = plt.figure()
    ax = fig.gca(projection='3d')
    X, Y = numpy.meshgrid(X, Y)
    surf = ax.plot_surface(X, Y, Z, rstride=1, cmap=cm.coolwarm, linewidth=0, antialiased=False) #, rstride=1,
cstride=1, cmap=cm.coolwarm, linewidth=0) #, antialiased=False)
    ax.set_zlim(0, 32)
    fig.colorbar(surf, shrink=0.5, aspect=5)
    image_name = 'Paristopia_t='+str(t)+'.png'
    plt.savefig(image_name, bbox_inches='tight')
    #plt.show()

def visualizeCity2(station_list):
    '''Returns a condensed 2-D array of city_map with just the stations for easy visualization of changes in number of
bikes in each station''' #OUTDATED - REPLACED BY visualizeCity()
    condensed_city = []
    for station in station_list:
        condensed_city.append((station.x,station.y,station.num_bikes,station.rental_price,station.dropoff_price))
    print condensed_city

def numBikesDifference(station_list):
    '''Returns the average difference between a station's initial bike count (13) and its current number of bikes for all the
stations). The higher this difference, the more uneven the number of bikes in station (and the worse the source/sink
problem)'''
    num_bikes_difference = 0
    for i in range(len(station_list)):
        num_bikes_difference += abs(station_list[i].num_bikes - INITIAL_NUM_BIKES)
    return float(num_bikes_difference)/len(station_list)

class Point:
    '''Each Point object (there are 1,000,000) represents a 10 m x 10 m area and contains:
    x - x coordinate of location
    y - y coordinate of location
    popularity - popularity of location
    num_stations - number of stations (greater than 1 only if a randomly assigned station is in an area where there already
is another station)
    num_people - number of people (Bike-Share users) in area'''
    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.popularity = 0 #every Point is initialized with 0 "popularity" but some will be given certain values if they are
points of interest
        self.station = None #variable station may hold Station object but is not initialized with one
        self.num_stations = 0 #a point is initialized with 0 stations
        self.userlist = [] #list userlist may contain several User objects but is initialized empty
        self.num_people = 0 #a point is initialized with 0 people (Bike-Share users)
    def incrementNumStations(self):
        self.num_stations += 1
    def incrementNumPeople(self):
        self.num_people += 1


class Station:
```

```
    def __init__(self,x,y):
        self.x = x
        self.y = y
        self.num_bikes = INITIAL_NUM_BIKES
        self.num_terminals = NUM_TERMINALS
        self.rental_price = INITIAL_PRICE
        self.dropoff_price = INITIAL_PRICE
    def incrementNumBikes(self):
        self.num_bikes+=1
    def decrementNumBikes(self):
        self.num_bikes-=1
    def averageBikeAvailability(self,city_map):
        num_bikes_in_area = 0
        num_stations_in_area = 0
        for x in range(self.x-30,self.x+30):
            if x < 0: #make sure that x is between 0 and 999
                x = 0
            if x >= 1000:
                x = 999
            for y in range(self.y-30,self.y+30):
                if y < 0: #make sure that y is between 0 and 999
                    y = 0
                if y >= 1000:
                    y = 999
                if city_map[x][y].num_stations != 0:
                    num_stations_in_area += 1
                    num_bikes_in_area += city_map[x][y].station.num_bikes
        return float(num_bikes_in_area)/float(num_stations_in_area)
    #returns the average number of bikes in a station's area
    #(600 m by 600 m square with that station as the center)
    #at a given time


class User:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def rentBike(self, rent_station): #action taken when user decides to rent a bike
        #note that the user is not moved when he/she rents a bike; this simplification allows us to ignore random
movements of bike users so we assume that bike renters are uniformly distributed
        rent_station.decrementNumBikes()
    def dropOffBike(self, dropoff_station): #action taken when user decides to drop off a bike
        #note that the user is not moved when he/she drops off a bike; we assume that bikers are uniformly distributed
        dropoff_station.incrementNumBikes()

city_map = generateCity()

#Experiment - changing prices
station_list = getStationList(city_map)
visualizeCity(station_list,-1)
for t in range(24):
    print t
    biking_users = findBikingUsers(city_map, t) #find a way to increment the time and measure results
    for user in biking_users:
        station = calcRentStation(city_map,user,station_list)
        user.rentBike(station)
        adjustStationPrice(station,True)
    for user in biking_users:
        station = calcDropOffStation(city_map,user,station_list)
        user.dropOffBike(station)
```

```
        adjustStationPrice(station,False)
    visualizeCity(station_list,t)
    print numBikesDifference(station_list)


#Control:
station_list = getStationList(city_map)
visualizeCity(station_list,-1)
for t in range(24):
    print t
    biking_users = findBikingUsers(city_map, t)
    for user in biking_users:
        user.rentBike(calcRentStation(city_map,user,station_list))
    for user in biking_users:
user.dropOffBike(calcDropOffStation(city_map,user,station_li
st))
    visualizeCity(station_list,t)
    print numBikesDifference(station_list)
```

References

Barz, Sara. “Show Me the Data! or Stuff I’ve Read About Vélib’, Part 4 of 4.” *An Urbanist in Paris*. N.p., 15 Apr. 2013. Web. 15 Oct. 2016.

Barz, Sara. “Stuff I’ve Read About Vélib’, Part 1 of 4.” *An Urbanist in Paris*. N.p., 15 Apr. 2013. Web. 15 Oct. 2016.

“Bay Area Bike Share Membership.” *Bay Area Bike Share*. N.p., n.d. Web. 22 Oct. 2016.

“Citi Bike Membership.” *Citi Bike NYC*. N.p., n.d. Web. 22 Oct. 2016.

Frade, Inês, and Anabela Ribeiro. “Bicycle Sharing Systems Demand.” *Procedia - Social and Behavioral Sciences* 111 (2014): 518-27. Web.

Kabra, Ashish, Elena Belavina, and Karan Girotra. “Bike-Share Systems: Accessibility and Availability.” *Environmentally Responsible Supply Chains Springer Series in Supply Chain Management* (2016): 127-42. Web.

“Key Figures for Paris.” *Mairie De Paris*. N.p., n.d. Web. 15 Oct. 2016.

Kurtzleben, Danielle. “Bike Sharing Systems Aren't Trying to Peddle for Profit.” U.S.News & World Report, 12 Apr. 2012. Web. 21 Oct. 2016.

Melvin, Joshua. “Paris: Seven Years of Velib' Bikes in 7 Stats.” - *The Local*. N.p., 15 July 2014. Web. 15 Oct. 2016.

“Metro Bike Share Pricing.” *Metro Bike Share*. N.p., 30 Sept. 2016. Web. 22 Oct. 2016.

“Paris Vélib’ Subscriptions and Fees” *Vélib’: Mairie de Paris*. N.p., n.d. Web. 22 Oct. 2016.

“Paris: Vélo Liberté.” *PBS*. N.p., n.d. Web. 15 Oct. 2016.

“Santa Monica Bike Share Plans and Pricing.” *Santa Monica Bike Share*. N.p., n.d. Web. 22 Oct. 2016.

“Statistiques Sur Le Fonctionnement Du Service Vlib - Paris Vlo.” *Statistiques Sur Le Fonctionnement Du Service Vlib - Paris Vlo*. N.p., n.d. Web. 15 Oct. 2016.

“The Bike-Share Planning Guide.” *The Institute of Transportation and Development Policy*. ITDP, n.d. Web.

“What Is Surge?” *Uber*. Uber Technologies Inc., n.d. Web. 22 Oct. 2016.