Keep Austin on time: A study of real-time bus tracking in MetroRapid

Ryan Young

October 26, 2014

1 Introduction

1.1 Life in the slow lane

Of all the reasons given for not using public transportation, "it takes too long" is probably the most widespread. After all, compared to the convenience of the private automobile, even the simplest grocery store errand demands a complex, multi-step planning process; transfers must be timed, stops must be located, and page after page of schedules must be carefully inspected. On top of that, the transit vehicles themselves run slow—and quite often, they run late, especially during periods of heavy traffic and road construction. Because of these factors, the choice between stewing in a city bus and blazing home in a muscle car in half the time is all too clear for the vast majority of Americans. Public transportation is inconvenient, unreliable, *and especially slow.*

The bus is the most widely employed mode of transit, but it is also the most disadvantaged in terms of speed. Condemned to use the same lanes used by general traffic, the bus faces the dual challenge of making stops and negotiating rush hour, while subways, metros, and even light rail vehicles (historically known as "streetcars") glide from station to station on dedicated rights of way. But trains are outrageously expensive. American transit agencies have turned to a lower cost option: *make the buses go faster.* A new concept, known as **Bus Rapid Transit** (BRT), promises the benefits of fast and reliable rail systems combined with the cost effectiveness and flexibility of ordinary buses. Dedicated transit lanes segregate buses from other traffic. Off-vehicle fare payments, upgraded stops, and bigger doors allow passengers to board faster. Traffic signal tweaks give buses green lights. All of these perks shave minutes off the schedule and improve the rider experience.

However, every BRT system still faces reliability issues. Even the fanciest bus with the best BRT infrastructure in place can run late. *For the riders, a faster bus matters not if they find themselves waiting forever for it.* Twenty first century technology offers a potential solution: real-time bus tracking. GPS trackers fitted onto buses allow the transit agency to predict precisely when they will arrive at the stop. This information is made available to customers via hotlines, text messages, and smartphone apps. For many riders, real-time "next bus" estimates are game changers. One can imagine, for example, sitting in an air-conditioned coffee shop and deciding whether to stay inside or brave the hot summer weather on the chance that the bus is actually keeping its schedule. By decreasing waiting times—and passenger frustration—real-time tracking makes public transit feel faster and more convenient.

Austin's own Capital Metro transit authority recognized BRT's enormous potential and implemented two new BRT routes this year. Route 801, established in January, and route 803, established in August, now carry passengers along Austin's busiest north-south corridors. Branded as "MetroRapid," the new routes feature:

- Bus lanes in downtown Austin.
- Fewer stops with wider spacing.
- Traffic signal prioritization outside of downtown Austin, which lengthens green lights so that buses can catch them.
- Premium, high-capacity buses with larger doors and the ability to board through any door, not just the front one, thereby streamlining the boarding process.
- Pre-boarding payments via smart cards and the Capital Metro smartphone app, also reducing boarding times.
- Upgraded bus stops, with raised boarding platforms at select ones.

MetroRapid (along with Capital Metro's commuter rail service, MetroRail) also includes a real-time bus tracking system provided by the Trapeze Group. GPS positions for each bus are updated within 90 seconds; predicted arrival times are then displayed at digital signs at bus stops, within the Capital Metro smartphone app, and on the agency's online trip planner.

1.2 A real-time problem

Despite studying computer science, I have always held an interest in public transportation. My friends and family roll their eyes (and call me outright crazy) whenever I insist on riding the bus around town. I have often played transportation computer games such as *SimCity*, *Transport Tycoon*, and *Simutrans*, and I enjoy learning about the latest transit technologies such as bus rapid transit.

One day, while exploring Austin, I was waiting for the 803 MetroRapid bus at the UT Pickle Research Campus. I noticed that the displayed arrival time was fluctuating between two to five minutes. I wondered how bus arrival times are estimated, and if the algorithms could be improved.

Sure enough, I unconvered a local news story from Columbus, Ohio complaining about deficiencies in Trapeze's software. The transit agency had invested millions of dollars and years of development time into a system that simply didn't work. The agency claimed that the the accuracy rates were "as low as 25 percent" and that the GPS tracking technology was unreliable (Rouan).

A search of of the UT library system revealed a few of papers on the subject of bus prediction. The first, by Wei-Hua Lin and Jian Zeng, was an experimental study of real-time bus tracking in Blacksburg, Virginia, a small rural town. The resesarchers used GPS-based bus locations and on-time performance statistics from the Blacksburg transit agency to develop and compare four bus arrival prediction algorithms. The algorithms were measured on the metrics of deviation (how closely the algorithms predicted the actual time) and stability (how much the algorithm predictions changed over time). The study concluded that the best-performing algorithm took into account the location of the bus, its adherance to the schedule, and whether or not the stop is a time point (at time point stops, the bus must wait until the listed departure time to proceed if it is running early) (Lin and Zeng). However, limitations in the quality of the data were noted. GPS tracking data have a limited resolution, and tracking units can go for minutes at a time without updates. For cases like these, location data have to be interpolated, sacrificing accuracy (104).

Although interesting, the study has little direct application to MetroRapid due to the small population of Blacksburg. The study authors did not take traffic into account because Blacksburg has little traffic congestion (Lin and Zeng 106). Obviously, this would not be the case for Austin, as anybody driving down Guadalupe Street during rush hour can attest to.

The second paper, by Ali Abdelfattah and Ata Khan, was about models for predicting bus delays. Like the Blacksburg study, the researchers created a number of bus arrival time algorithms and measured their performance. However, unlike the Blacksburg study, they used a computerized traffic simulation based on an Ottawa, Canada bus route to develop the models instead of real-world data. After tuning the models within the simulation program, the authors then field-tested them by predicting bus arrival times at a number of intersections. The study concluded that the most effective algorithm used a combination of distance, number of bus stops, and travel time as inputs (Abdelfattah and Khan).

For my inquiry, I decided to apply Abdelfattah and Khan's study to MetroRapid. Using their models, could I predict bus arrival times better than Capital Metro could? While the question of making better predictions for bus arrival times has been studied by others, to my knowledge nobody has examined the problem with today's widely deployed bus arrival prediction systems–and certainly not with Capital Metro's.

2 The experiment

The general procedure was:

1. Develop a model to predict the estimated arrival time for a bus.
2. Collect location and estimated arrival time data for MetroRapid buses over a week.
3. Tune the model to fit this data and attempt to predict bus arrival times for the next few days.

To simplify the experiment, I decided to focus on four bus stops instead of the entire MetroRapid system. Half were located in suburban areas, and the other half were located in downtown:

- Crestview Station (suburban)
- UT West Mall Station (downtown)
- Republic Square Station (downtown)
- Pleasant Hills Station (suburban)

All four stations were on the southbound side for route 801. Route 801 was selected because it is the more mature of the two MetroRapid routes. The southbound direction was chosen by coin toss.

2.1 The scraper

First, I wrote a computer program (see A.1) to record bus location data from Capital Metro. This data is publicly available through a hidden service. For help, I turned to the authors of the MetroRappid app, a smartphone app created by local Austinites that provides an alternative to the official Capital Metro app. They had written documentation for requesting locations and arrival times from Capital Metro's servers (Dawoodjee).

My program, written in the Python programming language, collects and saves bus locations every 30 seconds. Note that, as in the Blacksburg study, the intervals between GPS updates can vary. Capital Metro quotes within 90 seconds, while MetroRappid authors claim between 30 to 90 seconds. I chose 30 seconds to achieve high precision while minimizing the number of requests made to Capital Metro servers.

Due to the way Trapeze's software is designed, estimated arrival times for each station must be retrieved separately. Data for each station was retrieved every 60 seconds. With four stations in the experiment, this worked out to one request every 15 seconds.

Once I got the program written and working, I let it run for a week while I turned my attention to developing a model.

2.2 My algorithm

For ease of implementation, I opted to implement Abdelfattah and Khan's linear bus delay model:

$$\begin{aligned} DELAY = 0.4855 + 0.0287 \times DENS.LT + 0.0168 \times DENS.TH + \\ 0.9654 \times LENGTH - 1.1969 \times M/T + 0.1130 \times STATION \end{aligned}$$

- $DENS.LT$ is the left-turn vehicle density in vehicles per lane per kilometer. For this value, I estimated that about $\frac{1}{8}$ of all traffic would be making left turns. Thus, I used $\frac{1}{8} \times DENS.TH$.
- $DENS.TH$ is the straight-through vehicle density in vehicles per lane per kilometer. For this value, I estimated about 15 vehicles/lane/km for suburban stations and double that for downtown stations. This was based on analysis of Google Earth imagery.

- $LENGTH$ is distance to the station in kilometers. For this value, I used the straight-line difference between the station's coordinates and the bus's coordinates. While this is not as ideal as the roadway distance, the route of the 801 is mostly straight, so this approximation seemed "good enough."

- M/T is the ratio of moving time to traveling time. (Traveling time is time spent between stations, while moving time is time spent actually moving, i.e. not at a stoplight or stuck in traffic.) Based on my own experience riding MetroRapid, I estimated this at 0.5.

- $STATION$ is the number of bus stops that the bus must pass before arriving at the target stop. This was computed based on the stations per kilometer ratio; that is, $STATION = LENGTH \times (stations\ per\ km)$. For suburban areas, this is 2. For downtown areas, this is 3.

- $DELAY$ is the delay, in minutes, that the bus is expected to face.

According to Adelfattah and Khan, from $DELAY$ we can estimate the arrival time with the equation:

$$travel\ time = \frac{distance}{bus\ speed\ without\ delay} + DELAY$$

Where ($bus\ speed\ without\ delay$) is assumed to be 40 km/h. This is the same value that I used in my program.

Finally, by adding ($travel\ time$) to the current time, we arrive at the predicted arrival time.

2.3 The first analysis

To measure the performance of arrival time estimates, the following metrics were used:

- Mean error, the mean of the differences between the predicted and actual arrival times. This gives a good measure of how close the algorithm was.

- Standard deviation error, the standard deviation of the differences between the predicted and actual arrival times. This measures the spread of the predicted times. Lower spread is more desirable.

- Stability. Taking a cue from the Blacksburg study, I also included this metric, which Lin and Zeng defined mathematically as the average of $|e_n - e_{n-1}|$; that is, the average of the differences between successive predicted times. This metric is designed to detect algorithms that produce vastly different times in a short period of time. For example, predicting "5:45 PM" one minute and then "5:57 PM" the next is confusing and undesirable from a passenger's perspective.

After a week had elapsed, I extended my program to determine the actual bus arrival times for each station and compare them with Capital Metro's predicted times. Taking another hint from the Blacksburg study, I also compared the arrival times with the times listed on the schedule.

Capital Metro's system stops predicting arrival times once the bus has actually arrived at the stop. Therefore, we can assume that the actual arrival time is the time at which Capital Metro no longer gives an arrival estimate for that bus.

2.3.1 Tables and figures

Current time	Actual arrival time	Scheduled arrival time	Capital Metro est. arrival time	Capital Metro est. error (min)
11:13 AM	11:45 AM	11:44 AM	11:44 AM	-1
11:14 AM	11:45 AM	11:44 AM	11:44 AM	-1
11:15 AM	11:45 AM	11:44 AM	11:45 AM	0
11:16 AM	11:45 AM	11:44 AM	11:45 AM	0
11:17 AM	11:45 AM	11:44 AM	11:45 AM	0
11:18 AM	11:45 AM	11:44 AM	11:45 AM	0
11:20 AM	11:45 AM	11:44 AM	11:44 AM	-1
11:21 AM	11:45 AM	11:44 AM	11:44 AM	-1
11:22 AM	11:45 AM	11:44 AM	11:43 AM	-2
11:23 AM	11:45 AM	11:44 AM	11:43 AM	-2
11:24 AM	11:45 AM	11:44 AM	11:43 AM	-2
11:25 AM	11:45 AM	11:44 AM	11:44 AM	-1
11:26 AM	11:45 AM	11:44 AM	11:44 AM	-1
11:28 AM	11:45 AM	11:44 AM	11:44 AM	-1
11:29 AM	11:45 AM	11:44 AM	11:44 AM	-1
11:30 AM	11:45 AM	11:44 AM	11:44 AM	-1
11:31 AM	11:45 AM	11:44 AM	11:44 AM	-1
11:32 AM	11:45 AM	11:44 AM	11:45 AM	0
11:33 AM	11:45 AM	11:44 AM	11:45 AM	0
11:34 AM	11:45 AM	11:44 AM	11:44 AM	-1
11:36 AM	11:45 AM	11:44 AM	11:44 AM	-1
11:37 AM	11:45 AM	11:44 AM	11:45 AM	0
11:38 AM	11:45 AM	11:44 AM	11:45 AM	0
11:39 AM	11:45 AM	11:44 AM	11:45 AM	0
11:40 AM	11:45 AM	11:44 AM	11:46 AM	1
11:41 AM	11:45 AM	11:44 AM	11:46 AM	1
11:42 AM	11:45 AM	11:44 AM	11:46 AM	1
11:44 AM	11:45 AM	11:44 AM	11:45 AM	0
11:45 AM	11:45 AM	11:44 AM	11:46 AM	1

Table 1: Sample data for the Wednesday 11:44 AM bus at UT West Mall Station.

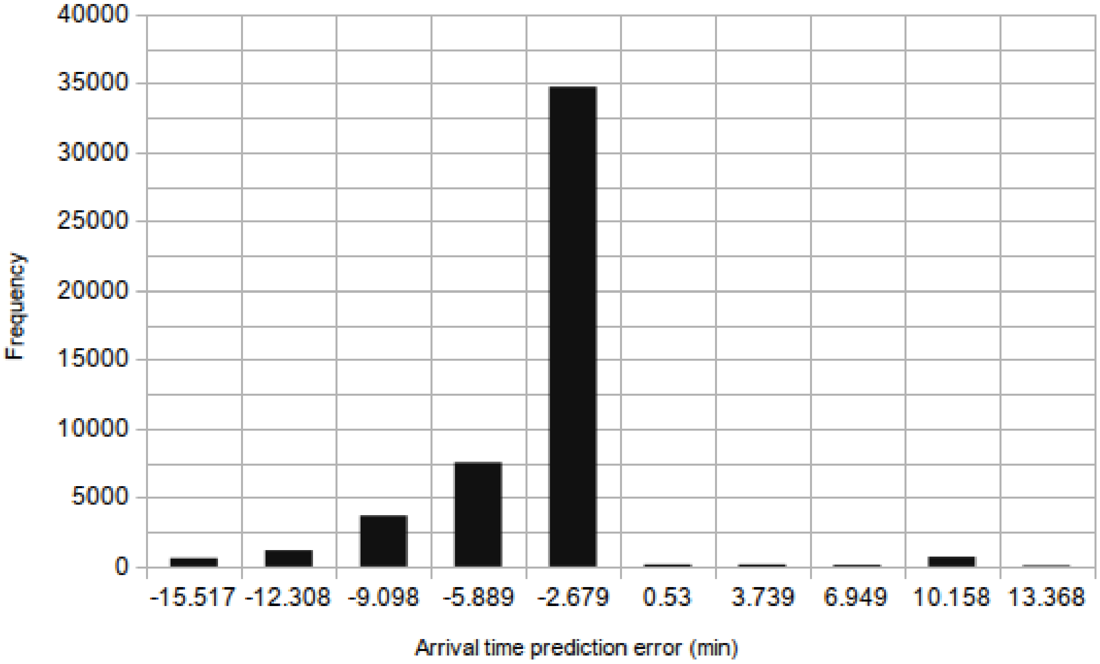

Figure 1: Histogram for schedule-based arrival time predictions.

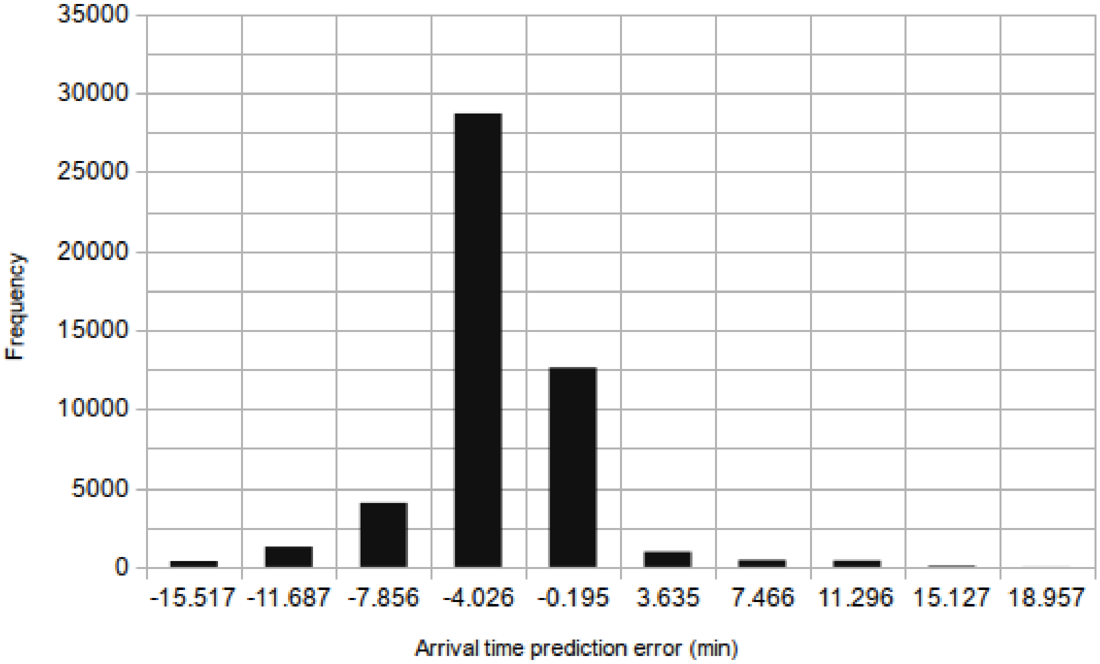

Figure 2: Histogram for Capital Metro arrival time predictions.

	Schedule	Capital Metro predicted
Error mean	-1.785	-1.137
Error std. deviation	2.342	2.614
Stability	0.0	0.390

Table 2: Summary of bus arrival prediction data over a six-day period. All units in minutes.

2.3.2 Comments

This analysis was very surprising.

First, note that the "stability" value for the schedule is zero. This is to be expected, because the arrival time listed on the schedule–and its difference from the actual arrival time–is constant.

More intriguing, however, are the mean and standard deviation values. On average, buses arrived within a minute of the time predicted by Capital Metro and within two minutes of the scheduled time. From the standard deviation values and the histogram plots, we can conclude that in the majority of cases the error was within five minutes. First, this indicates that Capital Metro ought to be congratulated, because most of the MetroRapid buses were running on time. Second, this calls into question the usefulness of real-time bus tracking. The expense of onboard GPS trackers and the computer systems required to utilize them amount to what is effectively *a 0.65-minute difference* in the average prediction error, compared to simply handing each passenger a copy of the bus schedule.

2.4 Tuning my algorithm

From the initial data I had collected, I was now ready to adjust and fine-tune my own arrival time model. However, I soon realized that I had made a major mistake.

When requesting bus locations, Capital Metro's server returns not one but a list of locations for each bus. The locations go back in time, creating a "track" for the bus as it moves along its route. Based on the design of the MetroRappid app, I had assumed that the *last* location in the list is the latest one.

But as I examined the data by hand, things weren't adding up. A bus would be considered "arrived" when it seemed from the location data that it was actually still several blocks away. Thinking it was an anamoly, I discarded the set. But then bus after bus showed the same peculiarity. Finally, it dawned on me; the *first* location is the latest, not the last one.

I could still analyze the performance of Capital Metro's arrival predictions, because I did not need good bus location data to do that. However, my week's worth of data were now useless for developing my own algorithm. My only choice was to quickly correct the program's flaw and collect a weekend's worth of good location data to use.

After collecting the weekend data set, I extended my program (see A.2) to analyze it in retrospect, applying my new algorithm to determine its performance. After some observations, I made the following tweaks to my model:

- Do not predict arrival times if the scheduled arrival time is at least 30 minutes away. This is because predicted times varied substantially when the bus was still 30 minutes out, and they would not be of much use to the passenger anyway. From the raw data, it appears that the Capital Metro system does this as well.

Current time	Actual arrival time	Scheduled arrival time	Capital Metro est. arrvl. time
09:52 AM	*10:51 AM*	*10:52 AM*	*10:52 AM*
09:53 AM	*10:51 AM*	*10:52 AM*	*10:52 AM*
09:54 AM	*10:51 AM*	*10:52 AM*	*10:52 AM*
09:56 AM	*10:51 AM*	*10:52 AM*	*10:52 AM*
...	...	...	...
10:21 AM	*10:51 AM*	*10:52 AM*	*10:52 AM*
10:22 AM	*10:51 AM*	*10:52 AM*	*10:52 AM*
10:23 AM	10:51 AM	10:52 AM	10:53 AM
10:24 AM	10:51 AM	10:52 AM	10:54 AM
10:25 AM	10:51 AM	10:52 AM	10:53 AM
...	...	...	...

Table 3: Estimated arrival times remain constant until 30 minutes before the scheduled arrival time.

- Reduce the prediction frequency from every one minute to every two minutes. This smooths out abrupt changes in the predicted arrival time.

After these modifications, the results seemed promising:

	Schedule	Capital Metro predicted	My algorithm predicted
Error mean	-1.954	-1.425	-0.794
Error std. deviation	3.956	3.729	4.384
Stability	0.0	0.196	0.202

Table 4: Summary of bus arrival prediction data over a two-day period. My algorithm tested retrospectively. All units in minutes.

Although standard deviation and stability were slightly higher than the Capital Metro data, indicating that my algorithm's predicted times varied slightly more, the mean error was lower. With my model tested, it was time to see how it performed in the real world.

2.5 The second analysis

I modified the scraper program (see A.3) to predict bus arrival times using my algorithm in addition to collecting Capital Metro's predicted arrival times. Thus, the algorithm was tested by interpreting data in real time instead of reexamining data that had been previously collected.

After running for two days (on Sunday and Monday), the results were in.

2.5.1 Tables and figures

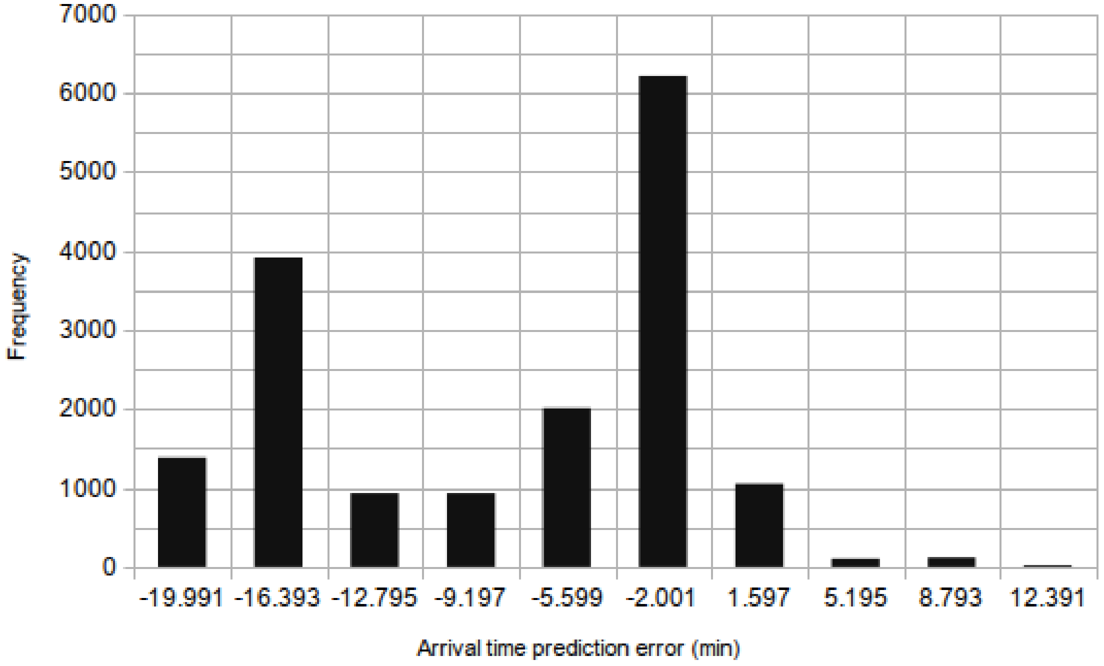

Figure 3: Histogram for my algorithm's arrival time predictions.

	Schedule	Capital Metro predicted	My algorithm predicted
Error mean	-1.632	-0.741	-6.372
Error std. deviation	3.230	2.795	7.048
Stability	0.0	0.296	0.491

Table 5: Summary of bus arrival prediction data over a two-day period.

2.5.2 Comments

Needless to say, the algorithm's performance was poor. On average, it produced arrival times that were approximately six minutes too early. Considering the high standard deviation and a histogram that indicates widely dispersed values, the predicted times were not very precise either. By every measure, the algorithm was a worse predictor of bus arrival times than both Capital Metro and the bus schedule.

3 Conclusions

The biggest problem with the experiment was a simple lack of resources. Attempting to come up with a good model for predicting bus arrival times proved far too difficult and complex for me to execute well. Consider the equation that I was using; I had to guess almost all of the variables based on my own intuition, and despite my enthusiasm for the subject, I am no transportation engineer. Also, without time and money, I could not determine these values precisely using surveys or by consulting with the city of Austin.

Another major issue was the quality of the GPS-based bus location data. As in the Blacksburg study, oftentimes buses would not send updated location information for minutes at a time, as demonstrated by a typical recording of a bus's track:

Time	Location (latitude, longitude)
05:50 PM	30.222734, -97.766434
05:51 PM	30.222488, -97.766525
05:51 PM	30.218885, -97.766525
05:52 PM	30.218885, -97.766525
05:53 PM	30.20541, -97.774696
05:53 PM	30.20541, -97.774696
05:54 PM	30.20541, -97.774696
05:54 PM	30.20541, -97.774696
05:55 PM	30.20541, -97.774696
05:55 PM	30.20541, -97.774696
05:56 PM	30.200048, -97.776283
05:56 PM	30.200048, -97.776283
05:57 PM	30.192375, -97.779228
05:57 PM	30.192375, -97.779228
05:58 PM	30.192375, -97.779228
05:58 PM	30.171007, -97.786095
05:59 PM	30.171007, -97.786095
05:59 PM	30.171007, -97.786095
06:00 PM	30.171007, -97.786095

Hence, there was always a high degree of uncertainty as to the exact location of the bus.

Other flaws in my methodology were more subtle. For instance, I had assumed that a bus had arrived at the stop when Capital Metro no longer offered an arrival estimate for it. Close inspection of the location data, however, revealed that this would sometimes result in a computed arrival time that was a minute or two later than the actual arrival time. As the GPS data are not precise enough to determine exactly when a bus has arrived, the ideal solution would be to have observers recording arrival times at the stop. Unfortunately, this was not practical concept for my inquiry.

A secondary issue was that I measured the distances between buses and stops in terms of straight-line distances, not roadway distances (as was done in the Blacksburg and Ottawa studies), which would be more accurate. The latter requires a detailed computer representation of the bus route, something that was out of the question given my limited skill set.

While my bus arrival time prediction model was a complete failure, there was one neat thing that I learned from my inquiry: real-time arrival estimates, at least for route 801, do not actually make much of a difference. The difference compared to the published schedule was a minute or two at best. Since the buses are almost always running early or on time, Capital Metro might consider tightening the schedule to reduce the travel time along the route. Also, I suggest a follow up study focusing on the effectiveness of current real-time tracking systems. Can their costs be justified? Would it be more effective, for example, to reserve their use for detours and extreme traffic jams?

As for me, the next time I'm waiting to catch a ride on the MetroRapid, I won't be staring nervously at the estimated arrival time.

Works Cited

Abdelfattah, Ali M and Ata M Khan. "Models for predicting bus delays". *Transportation Research Record: Journal of the Transportation Research Board* 1623.1 (1998): 8–15. Print.

Dawoodjee, Luqmaan. "The CapMetro API". 2014. Web. 17 Oct. 2014.

Lin, Wei-Hua and Jian Zeng. "Experimental study of real-time bus arrival time prediction with GPS data". *Transportation Research Record: Journal of the Transportation Research Board* 1666.1 (1999): 101–109. Print.

Rouan, Rick. "COTA says its real-time bus-tracking system doesn't work". The Columbus Dispatch, 24 July 2014. Web. 17 Oct. 2014.

A Source code listings

A.1 phaseI.py

```python
import sys
import time
import datetime
import pickle
import sched
import urllib.request
import urllib.parse
import xml.dom.minidom
import math
import statistics

DATA_FILE = "phaseI.pickle"

CAPMETRO_NEXTBUS = "http://www.capmetro.org/planner/s_nextbus2.asp"
CAPMETRO_BUSLOCS = "http://www.capmetro.org/planner/s_buslocation.asp"
POLL_INTERVAL = 30
SAVE_INTERVAL = 15 * 60

# Split trips at points that are this amount of time apart.
TRIP_TIMEDELTA_SPLIT = datetime.timedelta(hours = 12)
ACCEPTABLE_ERROR_MIN = 20
DEFAULT_SPEED_KMH = 40
ARRIVAL_ESTIMATE_INTERVAL = datetime.timedelta(minutes = 2)

class Station:
    def __init__(self, stop_id, name, latitude, longitude):
        self.stop_id = stop_id
        self.name = name
```

```
        self.trips = {}
        self.latitude = latitude
        self.longitude = longitude
    def record_trip(self, timestamp, trip_id, vehicle_id, sched_arrival, est_arrival):
        def fix_time(time):
            time = time.strip()
            # Error parsing estimated arrival time '12:02 XM': time data '12:02 XM'
            # does not match format '%I:%M %p'
            time = time.replace("X", "A")
            # Error parsing estimated arrival time '00:29 AM': time data '00:29 AM'
            # does not match format '%I:%M %p'
            time = time.replace("00", "12")
            return time
        if not trip_id in self.trips:
            self.trips[trip_id] = []
        sched_arrival_time = None
        sched_arrival = fix_time(sched_arrival)
        try:
            sched_arrival_time = datetime.datetime.strptime(sched_arrival,
                    "%I:%M␣%p").time()
        except Exception as e:
            print("Error␣parsing␣scheduled␣arrival␣time␣'" + sched_arrival + "':" +
                    str(e))
        est_arrival_time = None
        est_arrival = fix_time(est_arrival)
        try:
            est_arrival_time = datetime.datetime.strptime(est_arrival,
                    "%I:%M␣%p").time()
        except Exception as e:
            print("Error␣parsing␣estimated␣arrival␣time␣'" + est_arrival + "':␣"
                    + str(e))
        self.trips[trip_id].append({
            "timestamp": timestamp,
            "sched_arrival_time": sched_arrival_time,
            "est_arrival_time": est_arrival_time,
            "vehicle_id": vehicle_id
        })

stations = [
    Station(497, "UT␣West␣Mall", 30.286064, -97.741815),
    Station(5867, "Republic␣Square", 30.267751, -97.746857),
    Station(5606, "Crestview", 30.337852, -97.719035),
    Station(5872, "Pleasant␣Hill", 30.192391, -97.779203)
]
vehicles = {}

# Credit John D. Cook, http://www.johndcook.com/python_longitude_latitude.html
# (public domain)
def distance_on_unit_sphere(lat1, long1, lat2, long2):

    # Convert latitude and longitude to
    # spherical coordinates in radians.
    degrees_to_radians = math.pi/180.0

    # phi = 90 - latitude
    phi1 = (90.0 - lat1)*degrees_to_radians
    phi2 = (90.0 - lat2)*degrees_to_radians

    # theta = longitude
    theta1 = long1*degrees_to_radians
    theta2 = long2*degrees_to_radians

    # Compute spherical distance from spherical coordinates.

    # For two locations in spherical coordinates
    # (1, theta, phi) and (1, theta, phi)
    # cosine( arc length ) =
    #    sin phi sin phi' cos(theta-theta') + cos phi cos phi'
```

```python
    # distance = rho * arc length

    cos = (math.sin(phi1)*math.sin(phi2)*math.cos(theta1 - theta2) +
           math.cos(phi1)*math.cos(phi2))
    arc = math.acos( cos )

    # Remember to multiply arc by the radius of the earth
    # in your favorite set of units to get length.
    return arc

def dom_value(node):
    if node.firstChild == None:
        return ""
    else:
        return node.firstChild.data

def combine_time_date(time, dt):
    # Combines a time with date information from a datetime. We need some extra
    # logic to handle cases straddling midnight.
    midnight = datetime.time(0, 0)
    if time > midnight and dt.time() < midnight:
        dt_correct = datetime.timedelta(days = 1)
    elif time < midnight and dt.time() > midnight:
        dt_correct = datetime.timedelta(days = -1)
    else:
        dt_correct = datetime.timedelta()
    return datetime.datetime.combine(dt.date(), time) + dt_correct

def load_data():
    global stations
    global vehicles
    try:
        prev_data = pickle.load(open(DATA_FILE, "rb"))
        stations = prev_data["stations"]
        vehicles = prev_data["vehicles"]
    except Exception as e:
        print("Error loading data file: " + str(e))

def save_data():
    global stations
    global vehicles
    print("Saving data...")
    pickle.dump({ "stations": stations, "vehicles": vehicles },
           open(DATA_FILE, "wb"))

def retrieve_station_data(station):
    url = CAPMETRO_NEXTBUS + "?" + urllib.parse.urlencode({ "route": "801",
        "stopid": str(station.stop_id) })
    http_handle = None
    try:
        print("Downloading " + url)
        http_handle = urllib.request.urlopen(url)
    except Exception as e:
        print("Error retrieving data for station " + str(station.stop_id) + ": " +
              str(e))
        return False
    xml_tree = None
    now = datetime.datetime.now()
    try:
        xml_tree = xml.dom.minidom.parse(http_handle)
        runs = xml_tree.getElementsByTagName("Run")
        for trip in runs:
            vehicle_id = dom_value(trip.getElementsByTagName("Vehicleid")[0]).strip()
            if dom_value(trip.getElementsByTagName("Valid")[0]) == "Y":
                station.record_trip(
                    timestamp = now,
                    trip_id = dom_value(trip.getElementsByTagName("Tripid")[0]).strip(),
                    vehicle_id = vehicle_id,
```

```
                    sched_arrival = dom_value(trip.getElementsByTagName(”Triptime”)[0]),
                    est_arrival = dom_value(trip.getElementsByTagName(”Estimatedtime”)[0])
                )
    except Exception as e:
        print(”Error␣parsing␣tracking␣data␣for␣station␣” + str(station.stop_id) +
                ”:␣” + str(e))
        return False
    return True

def retrieve_vehicle_data():
    global vehicles
    url = CAPMETRO_BUSLOCS + ”?” + urllib.parse.urlencode({ ”route”: ”801” })
    try:
        print(”Downloading␣” + url)
        http_handle = urllib.request.urlopen(url)
    except Exception as e:
        print(”Error␣retrieving␣bus␣tracking␣data:␣” + str(e))
        return False
    xml_tree = None
    now = datetime.datetime.now()
    try:
        xml_tree = xml.dom.minidom.parse(http_handle)
        xml_vehicles = xml_tree.getElementsByTagName(”Vehicle”)
        for vehicle in xml_vehicles:
            vehicle_id = dom_value(vehicle.getElementsByTagName(”Vehicleid”)[0]).strip()
            latest_pos = [float(x) for x in dom_value(
                    vehicle.getElementsByTagName(”Position”)[0]).split(”,”)]
            if not vehicle_id in vehicles:
                vehicles[vehicle_id] = []
            vehicle_data = vehicles[vehicle_id]
            # Do not record duplicate data (data that hasn’t been updated within our
            # update interval).
            if (len(vehicle_data) == 0 or
                    vehicle_data[-1][”latitude”] != latest_pos[0] or
                    vehicle_data[-1][”longitude”] != latest_pos[1]):
                vehicle_data.append({
                    ”timestamp”: now,
                    ”block”: dom_value(
                            vehicle.getElementsByTagName(”Block”)[0]).strip(),
                    ”reliable”: dom_value(
                            vehicle.getElementsByTagName(”Reliable”)[0]).strip() == ”Y”,
                    ”off_route”: dom_value(
                            vehicle.getElementsByTagName(”Offroute”)[0]).strip() == ”Y”,
                    ”stopped”: dom_value(
                            vehicle.getElementsByTagName(”Stopped”)[0]).strip() == ”Y”,
                    ”in_service”: dom_value(
                            vehicle.getElementsByTagName(”Inservice”)[0]).strip() == ”Y”,
                    ”speed”: float(dom_value(vehicle.getElementsByTagName(”Speed”)[0])),
                    ”heading”: int(dom_value(vehicle.getElementsByTagName(”Heading”)[0])),
                    ”latitude”: latest_pos[0],
                    ”longitude”: latest_pos[1]
                })
    except Exception as e:
        print(”Error␣parsing␣bus␣tracking␣data:␣” + str(e))
        return False
    return True

def collect():
    load_data()
    scheduler = sched.scheduler(time.time, time.sleep)
    def collect_stations(i):
        retrieve_station_data(stations[i])
        i = (i + 1) % len(stations)
        now = datetime.datetime.now()
        now_time = now.time()
        # Cease collecting between 12:45 AM and 4:45 AM.
        if now_time >= datetime.time(0, 45) and now_time <= datetime.time(4, 45):
            scheduler.enterabs(time.mktime(datetime.datetime.combine(now,
```

```
                    datetime.time(4, 45)).timetuple()), 2, collect_stations, (0,))
        else:
            scheduler.enter(POLL_INTERVAL / len(stations), 2, collect_stations,
                    (i,))
    def collect_vehicles():
        retrieve_vehicle_data()
        now = datetime.datetime.now()
        now_time = now.time()
        # Cease collecting between 12:45 AM and 4:45 AM.
        if now_time >= datetime.time(0, 45) and now_time <= datetime.time(4, 45):
            scheduler.enterabs(time.mktime(datetime.datetime.combine(now,
                    datetime.time(4, 45)).timetuple()), 1, collect_vehicles, ())
        else:
            scheduler.enter(POLL_INTERVAL, 1, collect_vehicles, ())
    def collect_save():
        save_data()
        scheduler.enter(SAVE_INTERVAL, 3, collect_save, ())

    collect_vehicles()
    collect_stations(0)
    collect_save()
    try:
        scheduler.run()
    except KeyboardInterrupt:
        print(”Stopped.”)
        save_data()
        sys.exit(0)

def split_unique_trips(trip):
    # Trips are distinguished by trip ID, but this is only unique for one day.
    # This function splits one ”trip” into separate trips for each date.
    trips = []
    i = 0
    this_ts = None
    last_ts = None
    last_split = 0
    while i < len(trip):
        this_ts = trip[i][”timestamp”]
        if last_ts != None and this_ts - last_ts > TRIP_TIMEDELTA_SPLIT:
            trips.append(trip[last_split:i])
            last_split = i
        last_ts = this_ts
        i += 1
    if trip[last_split:] != []:
        trips.append(trip[last_split:])
    return trips

def get_vehicle_position(vehicle_id, dt):
    global vehicles
    if vehicle_id in vehicles:
        for point in vehicles[vehicle_id]:
            if (abs(point[”timestamp”] - dt) < datetime.timedelta(seconds = 30) or
                    point[”timestamp”] > dt):
                return point
        return None
    else:
        return None

def trip_error_data(trip, station):
    error_data = {
        ”schedule”: [],
        ”capmetro”: [],
        ”me”: []
    }
    arrival_time = trip[-1][”timestamp”]
    this_ts = None
    this_dt_correct = None
    this_sched = None
```

```python
    this_est = None
    for point in trip:
        this_ts = point["timestamp"]
        vehicle_id = point["vehicle_id"]
        this_est = combine_time_date(point["est_arrival_time"], this_ts)
        this_sched = combine_time_date(point["sched_arrival_time"], this_ts)
        # Calculate error and append to the lists.
        schedule_error_min = (this_sched - arrival_time).total_seconds() / 60
        capmetro_error_min = (this_est - arrival_time).total_seconds() / 60
        if abs(schedule_error_min) < ACCEPTABLE_ERROR_MIN:
            error_data["schedule"].append(schedule_error_min)
            error_data["capmetro"].append(capmetro_error_min)
    return error_data

def compute_stability(data):
    diffs = []
    i = 1
    while i < len(data):
        diffs.append(abs(data[i] - data[i - 1]))
        i += 1
    return statistics.mean(diffs)

def analyze():
    def append_keys(dict1, dict2):
        for key, value in dict1.items():
            dict2[key].append(value)
    global stations
    load_data()
    metrics = {
        "schedule": {
            "_data": [],
            "stability": []
        },
        "capmetro": {
            "_data": [],
            "stability": []
        }
    }
    for station in stations:
        for trip_id, trip_data in station.trips.items():
            trips = split_unique_trips(trip_data)
            for trip in trips:
                metrics["schedule"]["_data"] += this_error_data["schedule"]
                metrics["capmetro"]["_data"] += this_error_data["capmetro"]
                if len(this_error_data["schedule"]) > 1:
                    metrics["schedule"]["stability"].append(
                            compute_stability(this_error_data["schedule"]))
                    metrics["capmetro"]["stability"].append(
                            compute_stability(this_error_data["capmetro"]))

    print("\nBUS␣SCHEDULE␣ONLY")
    print("Error␣mean:␣" + str(statistics.mean(metrics["schedule"]["_data"])))
    print("Error␣std␣dev:␣" + str(statistics.stdev(metrics["schedule"]["_data"])))
    print("Stability:␣" + str(statistics.mean(metrics["schedule"]["stability"])))
    print("\nCAPITAL␣METRO␣ESTIMATES")
    print("Error␣mean:␣" + str(statistics.mean(metrics["capmetro"]["_data"])))
    print("Error␣std␣dev:␣" + str(statistics.stdev(metrics["capmetro"]["_data"])))
    print("Stability:␣" + str(statistics.mean(metrics["capmetro"]["stability"])))

if __name__ == "__main__":
    if sys.argv[1] == "collect":
        collect()
    elif sys.argv[1] == "analyze":
        analyze()
```

A.2 phaseII.py

```
import sys
import time
import datetime
import pickle
import sched
import urllib.request
import urllib.parse
import xml.dom.minidom
import math
import statistics

DATA_FILE = "phaseII.pickle"

CAPMETRO_NEXTBUS = "http://www.capmetro.org/planner/s_nextbus2.asp"
CAPMETRO_BUSLOCS = "http://www.capmetro.org/planner/s_buslocation.asp"
POLL_INTERVAL = 30
SAVE_INTERVAL = 15 * 60

# Split trips at points that are this amount of time apart.
TRIP_TIMEDELTA_SPLIT = datetime.timedelta(hours = 12)
ACCEPTABLE_ERROR_MIN = 20
DEFAULT_SPEED_KMH = 40
ARRIVAL_ESTIMATE_INTERVAL = datetime.timedelta(minutes = 2)

class Station:
    def __init__(self, stop_id, name, latitude, longitude):
        self.stop_id = stop_id
        self.name = name
        self.trips = {}
        self.latitude = latitude
        self.longitude = longitude
    def record_trip(self, timestamp, trip_id, vehicle_id, sched_arrival, est_arrival):
        def fix_time(time):
            time = time.strip()
            # Error parsing estimated arrival time '12:02 XM': time data '12:02 XM'
            # does not match format '%I:%M %p'
            time = time.replace("X", "A")
            # Error parsing estimated arrival time '00:29 AM': time data '00:29 AM'
            # does not match format '%I:%M %p'
            time = time.replace("00", "12")
            return time
        if not trip_id in self.trips:
            self.trips[trip_id] = []
        sched_arrival_time = None
        sched_arrival = fix_time(sched_arrival)
        try:
            sched_arrival_time = datetime.datetime.strptime(sched_arrival,
                    "%I:%M␣%p").time()
        except Exception as e:
            print("Error␣parsing␣scheduled␣arrival␣time␣'" + sched_arrival + "':" +
                    str(e))
        est_arrival_time = None
        est_arrival = fix_time(est_arrival)
        try:
            est_arrival_time = datetime.datetime.strptime(est_arrival,
                    "%I:%M␣%p").time()
        except Exception as e:
            print("Error␣parsing␣estimated␣arrival␣time␣'" + est_arrival + "':␣"
                    + str(e))
        self.trips[trip_id].append({
            "timestamp": timestamp,
            "sched_arrival_time": sched_arrival_time,
            "est_arrival_time": est_arrival_time,
            "vehicle_id": vehicle_id
        })

stations = [
    Station(497, "UT␣West␣Mall", 30.286064, -97.741815),
```

```
    Station(5867, "Republic␣Square", 30.267751, -97.746857),
    Station(5606, "Crestview", 30.337852, -97.719035),
    Station(5872, "Pleasant␣Hill", 30.192391, -97.779203)
]
vehicles = {}

# Credit John D. Cook, http://www.johndcook.com/python_longitude_latitude.html
# (public domain)
def distance_on_unit_sphere(lat1, long1, lat2, long2):

    # Convert latitude and longitude to
    # spherical coordinates in radians.
    degrees_to_radians = math.pi/180.0

    # phi = 90 - latitude
    phi1 = (90.0 - lat1)*degrees_to_radians
    phi2 = (90.0 - lat2)*degrees_to_radians

    # theta = longitude
    theta1 = long1*degrees_to_radians
    theta2 = long2*degrees_to_radians

    # Compute spherical distance from spherical coordinates.

    # For two locations in spherical coordinates
    # (1, theta, phi) and (1, theta, phi)
    # cosine( arc length ) =
    #    sin phi sin phi' cos(theta-theta') + cos phi cos phi'
    # distance = rho * arc length

    cos = (math.sin(phi1)*math.sin(phi2)*math.cos(theta1 - theta2) +
           math.cos(phi1)*math.cos(phi2))
    arc = math.acos( cos )

    # Remember to multiply arc by the radius of the earth
    # in your favorite set of units to get length.
    return arc

def dom_value(node):
    if node.firstChild == None:
        return ""
    else:
        return node.firstChild.data

def combine_time_date(time, dt):
    # Combines a time with date information from a datetime. We need some extra
    # logic to handle cases straddling midnight.
    midnight = datetime.time(0, 0)
    if time > midnight and dt.time() < midnight:
        dt_correct = datetime.timedelta(days = 1)
    elif time < midnight and dt.time() > midnight:
        dt_correct = datetime.timedelta(days = -1)
    else:
        dt_correct = datetime.timedelta()
    return datetime.datetime.combine(dt.date(), time) + dt_correct

def load_data():
    global stations
    global vehicles
    try:
        prev_data = pickle.load(open(DATA_FILE, "rb"))
        stations = prev_data["stations"]
        vehicles = prev_data["vehicles"]
    except Exception as e:
        print("Error␣loading␣data␣file:␣" + str(e))

def save_data():
    global stations
```

```
    global vehicles
    print("Saving␣data...")
    pickle.dump({ "stations": stations, "vehicles": vehicles },
            open(DATA_FILE, "wb"))

def retrieve_station_data(station):
    url = CAPMETRO_NEXTBUS + "?" + urllib.parse.urlencode({ "route": "801",
        "stopid": str(station.stop_id) })
    http_handle = None
    try:
        print("Downloading␣" + url)
        http_handle = urllib.request.urlopen(url)
    except Exception as e:
        print("Error␣retrieving␣data␣for␣station␣" + str(station.stop_id) + ":␣" +
                str(e))
        return False
    xml_tree = None
    now = datetime.datetime.now()
    try:
        xml_tree = xml.dom.minidom.parse(http_handle)
        runs = xml_tree.getElementsByTagName("Run")
        for trip in runs:
            vehicle_id = dom_value(trip.getElementsByTagName("Vehicleid")[0]).strip()
            if dom_value(trip.getElementsByTagName("Valid")[0]) == "Y":
                station.record_trip(
                    timestamp = now,
                    trip_id = dom_value(trip.getElementsByTagName("Tripid")[0]).strip(),
                    vehicle_id = vehicle_id,
                    sched_arrival = dom_value(trip.getElementsByTagName("Triptime")[0]),
                    est_arrival = dom_value(trip.getElementsByTagName("Estimatedtime")[0])
                )
    except Exception as e:
        print("Error␣parsing␣tracking␣data␣for␣station␣" + str(station.stop_id) +
                ":␣" + str(e))
        return False
    return True

def retrieve_vehicle_data():
    global vehicles
    url = CAPMETRO_BUSLOCS + "?" + urllib.parse.urlencode({ "route": "801" })
    try:
        print("Downloading␣" + url)
        http_handle = urllib.request.urlopen(url)
    except Exception as e:
        print("Error␣retrieving␣bus␣tracking␣data:␣" + str(e))
        return False
    xml_tree = None
    now = datetime.datetime.now()
    try:
        xml_tree = xml.dom.minidom.parse(http_handle)
        xml_vehicles = xml_tree.getElementsByTagName("Vehicle")
        for vehicle in xml_vehicles:
            vehicle_id = dom_value(vehicle.getElementsByTagName("Vehicleid")[0]).strip()
            latest_pos = [float(x) for x in dom_value(
                    vehicle.getElementsByTagName("Position")[0]).split(",")]
            if not vehicle_id in vehicles:
                vehicles[vehicle_id] = []
            vehicle_data = vehicles[vehicle_id]
            # Do not record duplicate data (data that hasn't been updated within our
            # update interval).
            if (len(vehicle_data) == 0 or
                    vehicle_data[-1]["latitude"] != latest_pos[0] or
                    vehicle_data[-1]["longitude"] != latest_pos[1]):
                vehicle_data.append({
                    "timestamp": now,
                    "block": dom_value(
                            vehicle.getElementsByTagName("Block")[0]).strip(),
                    "reliable": dom_value(
```

```
                            vehicle.getElementsByTagName("Reliable")[0]).strip() == "Y",
                    "off_route": dom_value(
                            vehicle.getElementsByTagName("Offroute")[0]).strip() == "Y",
                    "stopped": dom_value(
                            vehicle.getElementsByTagName("Stopped")[0]).strip() == "Y",
                    "in_service": dom_value(
                            vehicle.getElementsByTagName("Inservice")[0]).strip() == "Y",
                    "speed": float(dom_value(vehicle.getElementsByTagName("Speed")[0])),
                    "heading": int(dom_value(vehicle.getElementsByTagName("Heading")[0])),
                    "latitude": latest_pos[0],
                    "longitude": latest_pos[1]
                })
    except Exception as e:
        print("Error␣parsing␣bus␣tracking␣data:␣" + str(e))
        return False
    return True

def collect():
    load_data()
    scheduler = sched.scheduler(time.time, time.sleep)
    def collect_stations(i):
        retrieve_station_data(stations[i])
        i = (i + 1) % len(stations)
        now = datetime.datetime.now()
        now_time = now.time()
        # Cease collecting between 12:45 AM and 4:45 AM.
        if now_time >= datetime.time(0, 45) and now_time <= datetime.time(4, 45):
            scheduler.enterabs(time.mktime(datetime.datetime.combine(now,
                    datetime.time(4, 45)).timetuple()), 2, collect_stations, (0,))
        else:
            scheduler.enter(POLL_INTERVAL / len(stations), 2, collect_stations,
                    (i,))
    def collect_vehicles():
        retrieve_vehicle_data()
        now = datetime.datetime.now()
        now_time = now.time()
        # Cease collecting between 12:45 AM and 4:45 AM.
        if now_time >= datetime.time(0, 45) and now_time <= datetime.time(4, 45):
            scheduler.enterabs(time.mktime(datetime.datetime.combine(now,
                    datetime.time(4, 45)).timetuple()), 1, collect_vehicles, ())
        else:
            scheduler.enter(POLL_INTERVAL, 1, collect_vehicles, ())
    def collect_save():
        save_data()
        scheduler.enter(SAVE_INTERVAL, 3, collect_save, ())

    collect_vehicles()
    collect_stations(0)
    collect_save()
    try:
        scheduler.run()
    except KeyboardInterrupt:
        print("Stopped.")
        save_data()
        sys.exit(0)

def split_unique_trips(trip):
    # Trips are distinguished by trip ID, but this is only unique for one day.
    # This function splits one "trip" into separate trips for each date.
    trips = []
    i = 0
    this_ts = None
    last_ts = None
    last_split = 0
    while i < len(trip):
        this_ts = trip[i]["timestamp"]
        if last_ts != None and this_ts - last_ts > TRIP_TIMEDELTA_SPLIT:
            trips.append(trip[last_split:i])
```

```
            last_split = i
        last_ts = this_ts
        i += 1
    if trip[last_split:] != []:
        trips.append(trip[last_split:])
    return trips

def get_vehicle_position(vehicle_id, dt):
    global vehicles
    if vehicle_id in vehicles:
        for point in vehicles[vehicle_id]:
            if (abs(point["timestamp"] - dt) < datetime.timedelta(seconds = 30) or
                    point["timestamp"] > dt):
                return point
        return None
    else:
        return None

def estimate_arrival_time(vehicle_id, dt, scheduled_dt, station):
    if scheduled_dt - dt <= datetime.timedelta(minutes = 30):
        pos = get_vehicle_position(vehicle_id, dt)
        if (pos != None and not pos["off_route"] and
                pos["latitude"] != 0 and pos["longitude"] != 0):
            if station.name == "Pleasant Hill" or station.name == "Crestview":
                traffic_density = 15
                station_density = 2
            elif station.name == "UT West Mall" or station.name == "Republic Square":
                traffic_density = 30
                station_density = 3
            dist_km = distance_on_unit_sphere(pos["latitude"], pos["longitude"],
                    station.latitude, station.longitude) * 6373
            delay_min = (0.4855 + 0.0287 * traffic_density / 8 + 0.0168 *
                    traffic_density + 0.9654 * dist_km - 1.1969 * 0.5 + 0.1130 *
                    dist_km * station_density)
            return dt + datetime.timedelta(hours = dist_km / DEFAULT_SPEED_KMH +
                    delay_min / 60)
        else:
            return scheduled_dt
    else:
        return scheduled_dt

def trip_error_data(trip, station):
    error_data = {
        "schedule": [],
        "capmetro": [],
        "me": []
    }
    arrival_time = trip[-1]["timestamp"]
    this_ts = None
    this_dt_correct = None
    this_sched = None
    this_est = None
    this_my_est = None
    this_my_est_last_dt = None
    for point in trip:
        this_ts = point["timestamp"]
        vehicle_id = point["vehicle_id"]
        this_est = combine_time_date(point["est_arrival_time"], this_ts)
        this_sched = combine_time_date(point["sched_arrival_time"], this_ts)
        if (this_my_est_last_dt == None or
                this_ts - this_my_est_last_dt > ARRIVAL_ESTIMATE_INTERVAL):
            this_my_est = estimate_arrival_time(vehicle_id, this_ts, this_sched,
                    station)
            this_my_est_last_dt = this_ts
        # Calculate error and append to the lists.
        schedule_error_min = (this_sched - arrival_time).total_seconds() / 60
        capmetro_error_min = (this_est - arrival_time).total_seconds() / 60
        me_error_min = (this_my_est - arrival_time).total_seconds() / 60
```

```
        if abs(schedule_error_min) < ACCEPTABLE_ERROR_MIN:
            error_data["schedule"].append(schedule_error_min)
            error_data["capmetro"].append(capmetro_error_min)
            error_data["me"].append(me_error_min)
    return error_data

def compute_stability(data):
    diffs = []
    i = 1
    while i < len(data):
        diffs.append(abs(data[i] - data[i - 1]))
        i += 1
    return statistics.mean(diffs)

def analyze():
    def append_keys(dict1, dict2):
        for key, value in dict1.items():
            dict2[key].append(value)
    global stations
    load_data()
    metrics = {
        "schedule": {
            "_data": [],
            "stability": []
        },
        "capmetro": {
            "_data": [],
            "stability": []
        },
        "me": {
            "_data": [],
            "stability": []
        }
    }
    for station in stations:
        for trip_id, trip_data in station.trips.items():
            trips = split_unique_trips(trip_data)
            for trip in trips:
                metrics["schedule"]["_data"] += this_error_data["schedule"]
                metrics["capmetro"]["_data"] += this_error_data["capmetro"]
                metrics["me"]["_data"] += this_error_data["me"]
                if len(this_error_data["schedule"]) > 1:
                    metrics["schedule"]["stability"].append(
                            compute_stability(this_error_data["schedule"]))
                    metrics["capmetro"]["stability"].append(
                            compute_stability(this_error_data["capmetro"]))
                    metrics["me"]["stability"].append(
                            compute_stability(this_error_data["me"]))

    print("\nBUS␣SCHEDULE␣ONLY")
    print("Error␣mean:␣" + str(statistics.mean(metrics["schedule"]["_data"])))
    print("Error␣std␣dev:␣" + str(statistics.stdev(metrics["schedule"]["_data"])))
    print("Stability:␣" + str(statistics.mean(metrics["schedule"]["stability"])))
    print("\nCAPITAL␣METRO␣ESTIMATES")
    print("Error␣mean:␣" + str(statistics.mean(metrics["capmetro"]["_data"])))
    print("Error␣std␣dev:␣" + str(statistics.stdev(metrics["capmetro"]["_data"])))
    print("Stability:␣" + str(statistics.mean(metrics["capmetro"]["stability"])))
    print("\nMY␣ESTIMATES")
    print("Error␣mean:␣" + str(statistics.mean(metrics["me"]["_data"])))
    print("Error␣std␣dev:␣" + str(statistics.stdev(metrics["me"]["_data"])))
    print("Stability:␣" + str(statistics.mean(metrics["me"]["stability"])))

if __name__ == "__main__":
    if sys.argv[1] == "collect":
        collect()
    elif sys.argv[1] == "analyze":
        analyze()
```

A.3 phaseIII.py

```python
import sys
import time
import datetime
import pickle
import sched
import urllib.request
import urllib.parse
import xml.dom.minidom
import math
import statistics

DATA_FILE = "phaseIII.pickle"

CAPMETRO_NEXTBUS = "http://www.capmetro.org/planner/s_nextbus2.asp"
CAPMETRO_BUSLOCS = "http://www.capmetro.org/planner/s_buslocation.asp"
POLL_INTERVAL = 30
SAVE_INTERVAL = 15 * 60

# Split trips at points that are this amount of time apart.
TRIP_TIMEDELTA_SPLIT = datetime.timedelta(hours = 12)
ACCEPTABLE_ERROR_MIN = 20
DEFAULT_SPEED_KMH = 40
ARRIVAL_ESTIMATE_INTERVAL = datetime.timedelta(minutes = 2)

class Station:
    def __init__(self, stop_id, name, latitude, longitude):
        self.stop_id = stop_id
        self.name = name
        self.trips = {}
        self.latitude = latitude
        self.longitude = longitude
        self._my_est_last = None
        self._my_est_last_dt = None
    def record_trip(self, timestamp, trip_id, vehicle_id, sched_arrival,
            est_arrival, my_est_arrival):
        if not trip_id in self.trips:
            self.trips[trip_id] = []
        data = {
            "timestamp": timestamp,
            "sched_arrival": sched_arrival,
            "est_arrival": est_arrival,
            "vehicle_id": vehicle_id
        }
        if (self._my_est_last == None or
                timestamp - self._my_est_last_dt > ARRIVAL_ESTIMATE_INTERVAL):
            data["my_est_arrival"] = my_est_arrival
            self._my_est_last = my_est_arrival
            self._my_est_last_dt = timestamp
        else:
            data["my_est_arrival"] = self._my_est_last
        self.trips[trip_id].append(data)

stations = [
    Station(497, "UT␣West␣Mall", 30.286064, -97.741815),
    Station(5867, "Republic␣Square", 30.267751, -97.746857),
    Station(5606, "Crestview", 30.337852, -97.719035),
    Station(5872, "Pleasant␣Hill", 30.192391, -97.779203)
]
vehicles = {}

# Credit John D. Cook, http://www.johndcook.com/python_longitude_latitude.html
# (public domain)
def distance_on_unit_sphere(lat1, long1, lat2, long2):

    # Convert latitude and longitude to
```

```python
    # spherical coordinates in radians.
    degrees_to_radians = math.pi/180.0

    # phi = 90 - latitude
    phi1 = (90.0 - lat1)*degrees_to_radians
    phi2 = (90.0 - lat2)*degrees_to_radians

    # theta = longitude
    theta1 = long1*degrees_to_radians
    theta2 = long2*degrees_to_radians

    # Compute spherical distance from spherical coordinates.

    # For two locations in spherical coordinates
    # (1, theta, phi) and (1, theta, phi)
    # cosine( arc length ) =
    #    sin phi sin phi' cos(theta-theta') + cos phi cos phi'
    # distance = rho * arc length

    cos = (math.sin(phi1)*math.sin(phi2)*math.cos(theta1 - theta2) +
           math.cos(phi1)*math.cos(phi2))
    arc = math.acos( cos )

    # Remember to multiply arc by the radius of the earth
    # in your favorite set of units to get length.
    return arc

def dom_value(node):
    if node.firstChild == None:
        return ""
    else:
        return node.firstChild.data

def load_data():
    global stations
    global vehicles
    try:
        prev_data = pickle.load(open(DATA_FILE, "rb"))
        stations = prev_data["stations"]
        vehicles = prev_data["vehicles"]
    except Exception as e:
        print("Error␣loading␣data␣file:␣" + str(e))

def save_data():
    global stations
    global vehicles
    print("Saving␣data...")
    pickle.dump({ "stations": stations, "vehicles": vehicles },
                open(DATA_FILE, "wb"))

def parse_time(time_s, now_dt):
    time_s = time_s.strip()
    # Error parsing estimated arrival time '12:02 XM': time data '12:02 XM'
    # does not match format '%I:%M %p'
    time_s = time_s.replace("X", "A")
    # Error parsing estimated arrival time '00:29 AM': time data '00:29 AM'
    # does not match format '%I:%M %p'
    time_s = time_s.replace("00", "12")
    try:
        time = datetime.datetime.strptime(time_s, "%I:%M␣%p").time()
    except Exception as e:
        print("Error␣parsing␣time␣'" + time_s + "':" + str(e))
        return None
    # We need some extra logic to handle cases straddling midnight.
    if (time > datetime.time(0, 0) and
            now_dt.time() < datetime.time(0, 0)):
        dt_correct = datetime.timedelta(days = 1)
    elif (time < datetime.time(0, 0) and
```

```
            now_dt.time() > datetime.time(0, 0)):
        dt_correct = datetime.timedelta(days = -1)
    else:
        dt_correct = datetime.timedelta()
    return datetime.datetime.combine(now_dt.date(), time) + dt_correct

def get_vehicle_position(vehicle_id, dt):
    global vehicles
    if vehicle_id in vehicles:
        for point in vehicles[vehicle_id]:
            if point[”timestamp”] > dt - datetime.timedelta(seconds = 30):
                return point
        return None
    else:
        return None

def estimate_arrival_time(vehicle_id, dt, scheduled_dt, station):
    if scheduled_dt - dt <= datetime.timedelta(minutes = 30):
        pos = get_vehicle_position(vehicle_id, dt)
        if (pos != None and not pos[”off_route”] and
                pos[”latitude”] != 0 and pos[”longitude”] != 0):
            if station.name == ”Pleasant␣Hill” or station.name == ”Crestview”:
                traffic_density = 15
                station_density = 2
            elif station.name == ”UT␣West␣Mall” or station.name == ”Republic␣Square”:
                traffic_density = 30
                station_density = 3
            dist_km = distance_on_unit_sphere(pos[”latitude”], pos[”longitude”],
                    station.latitude, station.longitude) * 6373
            delay_min = (0.4855 + 0.0287 * traffic_density / 8 + 0.0168 *
                    traffic_density + 0.9654 * dist_km - 1.1969 * 0.5 + 0.1130 *
                    dist_km * station_density)
            return dt + datetime.timedelta(hours = dist_km / DEFAULT_SPEED_KMH +
                    delay_min / 60)
        else:
            return scheduled_dt
    else:
        return scheduled_dt

def retrieve_station_data(station):
    url = CAPMETRO_NEXTBUS + ”?” + urllib.parse.urlencode({ ”route”: ”801”,
        ”stopid”: str(station.stop_id) })
    http_handle = None
    try:
        print(”Downloading␣” + url)
        http_handle = urllib.request.urlopen(url)
    except Exception as e:
        print(”Error␣retrieving␣data␣for␣station␣” + str(station.stop_id) + ”:␣” +
               str(e))
        return False
    xml_tree = None
    now = datetime.datetime.now()
    try:
        xml_tree = xml.dom.minidom.parse(http_handle)
        runs = xml_tree.getElementsByTagName(”Run”)
        for trip in runs:
            vehicle_id = dom_value(trip.getElementsByTagName(”Vehicleid”)[0]).strip()
            if dom_value(trip.getElementsByTagName(”Valid”)[0]) == ”Y”:
                station.record_trip(
                    timestamp = now,
                    trip_id = dom_value(trip.getElementsByTagName(”Tripid”)[0]).strip(),
                    vehicle_id = vehicle_id,
                    sched_arrival = parse_time(dom_value(
                            trip.getElementsByTagName(”Triptime”)[0]), now),
                    est_arrival = parse_time(dom_value(
                            trip.getElementsByTagName(”Estimatedtime”)[0]), now),
                    my_est_arrival = estimate_arrival_time(vehicle_id, now, parse_time(
                            dom_value(trip.getElementsByTagName(”Triptime”)[0]), now), station)
```

```
                )
    except Exception as e:
        print("Error␣parsing␣tracking␣data␣for␣station␣" + str(station.stop_id) +
                ":␣" + str(e))
        return False
    return True

def retrieve_vehicle_data():
    global vehicles
    url = CAPMETRO_BUSLOCS + "?" + urllib.parse.urlencode({ "route": "801" })
    try:
        print("Downloading␣" + url)
        http_handle = urllib.request.urlopen(url)
    except Exception as e:
        print("Error␣retrieving␣bus␣tracking␣data:␣" + str(e))
        return False
    xml_tree = None
    now = datetime.datetime.now()
    try:
        xml_tree = xml.dom.minidom.parse(http_handle)
        xml_vehicles = xml_tree.getElementsByTagName("Vehicle")
        for vehicle in xml_vehicles:
            vehicle_id = dom_value(vehicle.getElementsByTagName("Vehicleid")[0]).strip()
            latest_pos = [float(x) for x in dom_value(
                    vehicle.getElementsByTagName("Position")[0]).split(",")]
            if not vehicle_id in vehicles:
                vehicles[vehicle_id] = []
            vehicle_data = vehicles[vehicle_id]
            # Do not record duplicate data (data that hasn't been updated within our
            # update interval).
            if (len(vehicle_data) == 0 or
                    vehicle_data[-1]["latitude"] != latest_pos[0] or
                    vehicle_data[-1]["longitude"] != latest_pos[1]):
                vehicle_data.append({
                    "timestamp": now,
                    "block": dom_value(
                            vehicle.getElementsByTagName("Block")[0]).strip(),
                    "reliable": dom_value(
                            vehicle.getElementsByTagName("Reliable")[0]).strip() == "Y",
                    "off_route": dom_value(
                            vehicle.getElementsByTagName("Offroute")[0]).strip() == "Y",
                    "stopped": dom_value(
                            vehicle.getElementsByTagName("Stopped")[0]).strip() == "Y",
                    "in_service": dom_value(
                            vehicle.getElementsByTagName("Inservice")[0]).strip() == "Y",
                    "speed": float(dom_value(vehicle.getElementsByTagName("Speed")[0])),
                    "heading": int(dom_value(vehicle.getElementsByTagName("Heading")[0])),
                    "latitude": latest_pos[0],
                    "longitude": latest_pos[1]
                })
    except Exception as e:
        print("Error␣parsing␣bus␣tracking␣data:␣" + str(e))
        return False
    return True

def collect():
    load_data()
    scheduler = sched.scheduler(time.time, time.sleep)
    def collect_stations(i):
        retrieve_station_data(stations[i])
        i = (i + 1) % len(stations)
        now = datetime.datetime.now()
        now_time = now.time()
        # Cease collecting between 12:45 AM and 4:45 AM.
        if now_time >= datetime.time(0, 45) and now_time <= datetime.time(4, 45):
            scheduler.enterabs(time.mktime(datetime.datetime.combine(now,
                    datetime.time(4, 45)).timetuple()), 2, collect_stations, (0,))
        else:
```

```
            scheduler.enter(POLL_INTERVAL / len(stations), 2, collect_stations,
                    (i,))
    def collect_vehicles():
        retrieve_vehicle_data()
        now = datetime.datetime.now()
        now_time = now.time()
        # Cease collecting between 12:45 AM and 4:45 AM.
        if now_time >= datetime.time(0, 45) and now_time <= datetime.time(4, 45):
            scheduler.enterabs(time.mktime(datetime.datetime.combine(now,
                    datetime.time(4, 45)).timetuple()), 1, collect_vehicles, ())
        else:
            scheduler.enter(POLL_INTERVAL, 1, collect_vehicles, ())
    def collect_save():
        save_data()
        scheduler.enter(SAVE_INTERVAL, 3, collect_save, ())

    collect_vehicles()
    collect_stations(0)
    collect_save()
    try:
        scheduler.run()
    except KeyboardInterrupt:
        print("Stopped.")
        save_data()
        sys.exit(0)

def split_unique_trips(trip):
    # Trips are distinguished by trip ID, but this is only unique for one day.
    # This function splits one "trip" into separate trips for each date.
    trips = []
    i = 0
    this_ts = None
    last_ts = None
    last_split = 0
    while i < len(trip):
        this_ts = trip[i]["timestamp"]
        if last_ts != None and this_ts - last_ts > TRIP_TIMEDELTA_SPLIT:
            trips.append(trip[last_split:i])
            last_split = i
        last_ts = this_ts
        i += 1
    if trip[last_split:] != []:
        trips.append(trip[last_split:])
    return trips

def get_station_arrival_time(search_start_dt, vehicle_id, station):
    global stations
    for this_station in stations:
        for trip_id, trip in this_station.trips.items():
            if len(trip) > 0 and trip[0]["vehicle_id"] == vehicle_id:
                for point in trip:
                    ts = point["timestamp"]
                    vehicle_pos = get_vehicle_position(vehicle_id, ts)
                    if (ts > search_start_dt and
                            ts - search_start_dt < TRIP_TIMEDELTA_SPLIT and
                            vehicle_pos != None):
                        dist_m = distance_on_unit_sphere(station.latitude, station.longitude,
                                vehicle_pos["latitude"], vehicle_pos["longitude"]) * 6373000
                        if dist_m < ARRIVAL_DISTANCE_M:
                            return ts
    return None

def trip_error_data(trip, station):
    error_data = {
        "schedule": [],
        "capmetro": [],
        "me": []
    }
```

```
    arrival_time = trip[-1][”timestamp”]
    for point in trip:
        # Calculate error and append to the lists.
        schedule_error_min = (point[”sched_arrival”] - arrival_time).total_seconds() / 60
        capmetro_error_min = (point[”est_arrival”] - arrival_time).total_seconds() / 60
        my_error_min = (point[”my_est_arrival”] - arrival_time).total_seconds() / 60
        if (abs(schedule_error_min) < ACCEPTABLE_ERROR_MIN):
            error_data[”schedule”].append(schedule_error_min)
            error_data[”capmetro”].append(capmetro_error_min)
            error_data[”me”].append(my_error_min)
    return error_data

def compute_stability(data):
    diffs = []
    i = 1
    while i < len(data):
        diffs.append(abs(data[i] - data[i - 1]))
        i += 1
    return statistics.mean(diffs)

def analyze():
    def append_keys(dict1, dict2):
        for key, value in dict1.items():
            dict2[key].append(value)
    global stations
    load_data()
    metrics = {
        ”schedule”: {
            ”_data”: [],
            ”stability”: []
        },
        ”capmetro”: {
            ”_data”: [],
            ”stability”: []
        },
        ”me”: {
            ”_data”: [],
            ”stability”: []
        }
    }
    for station in stations:
        for trip_id, trip_data in station.trips.items():
            trips = split_unique_trips(trip_data)
            for trip in trips:
                this_error_data = trip_error_data(trip, station)
                metrics[”schedule”][”_data”] += this_error_data[”schedule”]
                metrics[”capmetro”][”_data”] += this_error_data[”capmetro”]
                metrics[”me”][”_data”] += this_error_data[”me”]
                if len(this_error_data[”schedule”]) > 1:
                    metrics[”schedule”][”stability”].append(
                            compute_stability(this_error_data[”schedule”]))
                    metrics[”capmetro”][”stability”].append(
                            compute_stability(this_error_data[”capmetro”]))
                    metrics[”me”][”stability”].append(
                            compute_stability(this_error_data[”me”]))

    print(”\nBUS_SCHEDULE_ONLY”)
    print(”Error␣mean:␣” + str(statistics.mean(metrics[”schedule”][”_data”])))
    print(”Error␣std␣dev:␣” + str(statistics.stdev(metrics[”schedule”][”_data”])))
    print(”Stability:␣” + str(statistics.mean(metrics[”schedule”][”stability”])))
    print(”\nCAPITAL_METRO_ESTIMATES”)
    print(”Error␣mean:␣” + str(statistics.mean(metrics[”capmetro”][”_data”])))
    print(”Error␣std␣dev:␣” + str(statistics.stdev(metrics[”capmetro”][”_data”])))
    print(”Stability:␣” + str(statistics.mean(metrics[”capmetro”][”stability”])))
    print(”\nMY_ESTIMATES”)
    print(”Error␣mean:␣” + str(statistics.mean(metrics[”me”][”_data”])))
    print(”Error␣std␣dev:␣” + str(statistics.stdev(metrics[”me”][”_data”])))
    print(”Stability:␣” + str(statistics.mean(metrics[”me”][”stability”])))
```

```

if __name__ == "__main__":
    if sys.argv[1] == "collect":
        collect()
    elif sys.argv[1] == "analyze":
        analyze()
```