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 researchers 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:

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

- 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 straightline 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	Scheduled	Capital Metro	Capital Metro
	time	arrival time	est. arrival time	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.

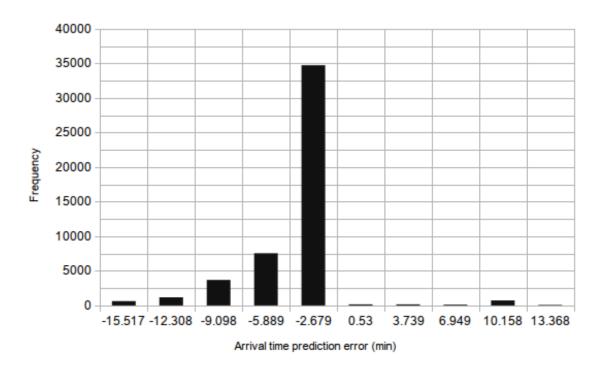


Figure 1: Histogram for schedule-based arrival time predictions.

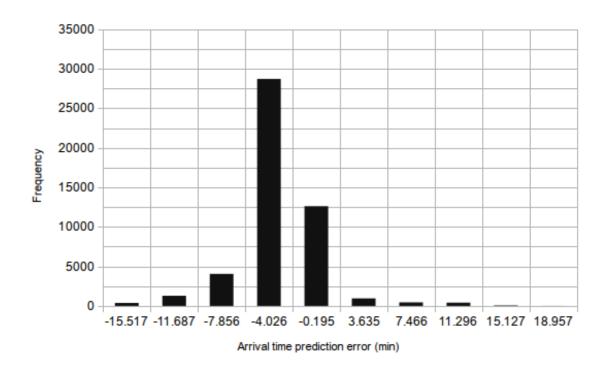


Figure 2: Histogram for Capital Metro arrival time predictions.

	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	Scheduled	Capital Metro
	time	arrival time	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

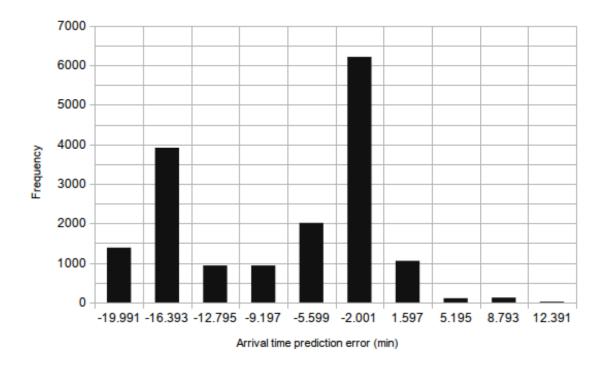


Figure 3: Histogram for my algorithm's arrival time predictions.

	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

```
import sys
  import time
3 import datetime
   import pickle
   import sched
   import urllib.request
   import urllib.parse
   import xml.dom.minidom
   import math
10
   import statistics
11
   DATA_FILE = "phaseI.pickle"
12
13
   CAPMETRO_NEXTBUS = "http://www.capmetro.org/planner/s_nextbus2.asp"
14
   CAPMETRO\_BUSLOCS = "http://www.capmetro.org/planner/s\_buslocation.asp"
15
   POLL_INTERVAL = 30
16
   SAVE_INTERVAL = 15 * 60
17
18
   # Split trips at points that are this amount of time apart.
19
   TRIP_TIMEDELTA_SPLIT = datetime.timedelta(hours = 12)
20
   ACCEPTABLE\_ERROR\_MIN = 20
21
   DEFAULT\_SPEED\_KMH = 40
22
   ARRIVALESTIMATE_INTERVAL = datetime.timedelta(minutes = 2)
23
24
   class Station:
25
        def __init__(self , stop_id , name, latitude , longitude):
26
            self.stop_id = stop_id
27
            self.name = name
```

```
29
             self.trips = \{\}
30
             self.latitude = latitude
31
             self.longitude = longitude
        def record_trip(self, timestamp, trip_id, vehicle_id, sched_arrival, est_arrival):
32
            def fix_time(time):
33
                 time = time.strip()
34
                 # Error parsing estimated arrival time '12:02 XM': time data '12:02 XM'
35
                 # does not match format '%I:%M %p'
36
                 \texttt{time} \, = \, \texttt{time.replace} \, (\text{"X"} \, , \, \text{"A"} \, )
37
                 # Error parsing estimated arrival time '00:29 AM': time data '00:29 AM'
38
                 # does not match format '%I:%M %p'
39
                 time = time.replace("00", "12")
40
41
                 return time
             if not trip_id in self.trips:
42
                 self.trips[trip_id] = []
43
44
             sched_arrival_time = None
             sched_arrival = fix_time(sched_arrival)
45
46
             \mathbf{try}:
                 sched_arrival_time = datetime.datetime.strptime(sched_arrival,
47
48
                          "%I:%M_%p").time()
49
            except Exception as e:
                 print ("Error_parsing_scheduled_arrival_time_'" + sched_arrival + "':" +
50
                          str(e))
51
             est_arrival_time = None
52
             est_arrival = fix_time(est_arrival)
53
54
             \mathbf{try}:
                 est_arrival_time = datetime.datetime.strptime(est_arrival,
55
                          "%I:%M_%p").time()
56
            except Exception as e:
57
                 print("Error_parsing_estimated_arrival_time_'" + est_arrival + "':_"
58
                          + str(e))
59
60
             self.trips[trip_id].append({
61
                 "timestamp": timestamp,
                 "sched_arrival_time": sched_arrival_time,
62
                 "est_arrival_time": est_arrival_time,
63
                 "vehicle_id": vehicle_id
64
            })
65
66
    stations = [
67
        Station (497, "UT_West_Mall", 30.286064, -97.741815),
68
        Station (5867, "Republic_Square", 30.267751, -97.746857), Station (5606, "Crestview", 30.337852, -97.719035),
69
70
        Station (5872, "Pleasant_Hill", 30.192391, -97.779203)
71
72
73
    vehicles = \{\}
74
   \#\ Credit\ John\ D.\ Cook,\ http://www.johndcook.com/python\_longitude\_latitude.html
75
   # (public domain)
76
   def distance_on_unit_sphere(lat1, long1, lat2, long2):
77
78
        # Convert latitude and longitude to
79
80
        # spherical coordinates in radians.
        degrees_to_radians = math.pi/180.0
81
82
        \# phi = 90 - latitude
83
        phi1 = (90.0 - lat1)*degrees_to_radians
84
        phi2 = (90.0 - lat2)*degrees_to_radians
85
86
87
        \# theta = longitude
        theta1 = long1*degrees_to_radians
88
89
        theta2 = long2*degrees_to_radians
90
        # Compute spherical distance from spherical coordinates.
91
92
        # For two locations in spherical coordinates
93
94
        # (1, theta, phi) and (1, theta, phi)
        \# cosine(arc length) =
95
              sin phi sin phi' cos(theta-theta') + cos phi cos phi'
96
```

```
97
                \# distance = rho * arc length
 98
 99
                \cos = (\text{math.sin}(\text{phi1})*\text{math.sin}(\text{phi2})*\text{math.cos}(\text{theta1} - \text{theta2}) +
                               math.cos(phi1)*math.cos(phi2))
100
                arc = math.acos(cos)
101
102
                # Remember to multiply arc by the radius of the earth
103
104
                # in your favorite set of units to get length.
                return arc
105
106
        def dom_value(node):
107
                 if node.firstChild == None:
108
                        return ""
109
110
                 else:
                        return node.firstChild.data
111
112
        def combine_time_date(time, dt):
113
114
                # Combines a time with date information from a datetime. We need some extra
                # logic to handle cases straddling midnight.
115
116
                midnight = datetime.time(0, 0)
117
                 if time > midnight and dt.time() < midnight:
                         dt_correct = datetime.timedelta(days = 1)
118
                 elif time < midnight and dt.time() > midnight:
119
                         dt_{correct} = datetime.timedelta(days = -1)
120
121
                         dt_correct = datetime.timedelta()
122
                return datetime.datetime.combine(dt.date(), time) + dt_correct
123
124
        def load_data():
125
                global stations
126
                global vehicles
127
128
                try:
                         prev_data = pickle.load(open(DATA_FILE, "rb"))
129
                         stations = prev_data["stations"]
130
                         vehicles = prev_data["vehicles"]
131
                except Exception as e:
132
                         print("Error_loading_data_file:_" + str(e))
133
134
        def save_data():
135
136
                global stations
                 global vehicles
137
                print("Saving_data...")
138
                 pickle.dump({ "stations": stations, "vehicles": vehicles },
139
                                open(DATA_FILE, "wb"))
140
141
        def retrieve_station_data(station):
142
                 {\tt url} = {\tt CAPMETRO\_NEXTBUS} + "?" + {\tt urllib.parse.urlencode} (\{ \ "route": "801", \ "route": "8
143
                        "stopid": str(station.stop_id) })
144
                http_handle = None
145
146
                         print("Downloading_" + url)
147
148
                         http_handle = urllib.request.urlopen(url)
                except Exception as e:
149
                        print("Error_retrieving_data_for_station_" + str(station.stop_id) + ":_" +
150
151
                                         str(e))
                        return False
152
                 xml_tree = None
153
                now = datetime.datetime.now()
154
155
                         xml\_tree = xml.dom.minidom.parse(http\_handle)
156
                         runs = xml_tree.getElementsByTagName("Run")
157
                         for trip in runs:
158
                                 vehicle_id = dom_value(trip.getElementsByTagName("Vehicleid")[0]).strip()
159
                                 if dom_value(trip.getElementsByTagName("Valid")[0]) == "Y":
160
                                         station.record_trip(
161
162
                                                 timestamp = now,
                                                 trip_id = dom_value(trip.getElementsByTagName("Tripid")[0]).strip(),
163
                                                 vehicle_id = vehicle_id ,
164
```

```
165
                         sched_arrival = dom_value(trip.getElementsByTagName("Triptime")[0]).
                         est_arrival = dom_value(trip.getElementsByTagName("Estimatedtime")[0])
166
167
        except Exception as e:
168
            print("Error_parsing_tracking_data_for_station_" + str(station.stop_id) +
169
                     ": _" + str(e))
170
            return False
171
172
        return True
173
    def retrieve_vehicle_data():
174
        global vehicles
175
        url = CAPMETRO.BUSLOCS + "?" + urllib.parse.urlencode({ "route": "801" })
176
177
        try:
             print("Downloading_" + url)
178
             http_handle = urllib.request.urlopen(url)
179
180
        except Exception as e:
            print("Error_retrieving_bus_tracking_data:_" + str(e))
181
182
             return False
        xml_tree = None
183
184
        now = datetime.datetime.now()
185
             xml_tree = xml.dom.minidom.parse(http_handle)
186
             xml_vehicles = xml_tree.getElementsByTagName("Vehicle")
187
             for vehicle in xml_vehicles:
188
                 vehicle_id = dom_value(vehicle.getElementsByTagName("Vehicleid")[0]).strip()
189
190
                 latest_pos = [float(x) for x in dom_value(
                         vehicle.getElementsByTagName("Position")[0]).split(",")]
191
                 if not vehicle_id in vehicles:
192
                     vehicles [vehicle_id] = []
193
                 vehicle_data = vehicles[vehicle_id]
194
                 \# Do not record duplicate data (data that hasn't been updated within our
195
                 \# update interval).
196
197
                 if (len(vehicle_data) == 0 or
                         vehicle_data[-1]["latitude"] != latest_pos[0] or
198
                         vehicle_data[-1]["longitude"] != latest_pos[1]):
199
                     vehicle_data.append({
200
                         "timestamp": now,
201
                         "block": dom_value(
202
                                  vehicle.getElementsByTagName("Block")[0]).strip(),
203
                         "reliable": dom_value(
204
                                  vehicle.getElementsByTagName("Reliable")[0]).strip() == "Y",
205
                         "off_route": dom_value(
206
                                  vehicle.getElementsByTagName("Offroute")[0]).strip() == "Y",
207
                         "stopped": dom_value(
208
                                  vehicle.getElementsByTagName("Stopped")[0]).strip() == "Y",
209
                         "in_service": dom_value(
210
                                  vehicle.getElementsByTagName("Inservice")[0]).strip() = "Y",
211
                         "speed": float(dom_value(vehicle.getElementsByTagName("Speed")[0]))
212
                         "heading": int(dom_value(vehicle.getElementsByTagName("Heading")[0])),
213
                         "latitude": latest_pos[0]
214
                         "longitude": latest_pos[1]
215
216
        except Exception as e:
217
            print("Error_parsing_bus_tracking_data:_" + str(e))
218
            return False
219
220
        return True
221
    def collect():
222
223
        load_data()
        scheduler = sched.scheduler(time.time, time.sleep)
224
225
        def collect_stations(i):
226
             retrieve_station_data(stations[i])
            i = (i + 1) \% len(stations)
227
            now = datetime.datetime.now()
228
            now_time = now.time()
229
230
            # Cease collecting between 12:45 AM and 4:45 AM.
            if now_time >= datetime.time(0, 45) and now_time <= datetime.time(4, 45):
231
                 scheduler.enterabs(time.mktime(datetime.datetime.combine(now,
232
```

```
233
                          datetime.time(4, 45)).timetuple()), 2, collect_stations, (0,))
234
             else:
235
                 scheduler.enter(POLL_INTERVAL / len(stations), 2, collect_stations,
236
         def collect_vehicles():
237
             retrieve_vehicle_data()
238
             now = datetime.datetime.now()
239
             now_time = now.time()
240
             # Cease collecting between 12:45 AM and 4:45 AM.
241
             if now_time >= datetime.time(0, 45) and now_time <= datetime.time(4, 45):
242
243
                 scheduler.enterabs(time.mktime(datetime.datetime.combine(now,
                          datetime.time(4, 45)).timetuple()), 1, collect_vehicles, ())
244
245
                 scheduler.enter(POLL_INTERVAL, 1, collect_vehicles, ())
246
247
         def collect_save():
248
             save_data()
             scheduler.enter(SAVE_INTERVAL, 3, collect_save, ())
249
250
         collect_vehicles()
251
252
         collect_stations(0)
253
         collect_save()
254
        trv:
255
             scheduler.run()
        except KeyboardInterrupt:
256
             print("Stopped.")
257
258
             save_data()
259
             sys.exit(0)
260
    def split_unique_trips(trip):
261
        # Trips are distinguished by trip ID, but this is only unique for one day.
262
        # This function splits one "trip" into separate trips for each date.
263
264
         trips = []
265
        i = 0
         this_ts = None
266
267
         last_ts = None
         last\_split = 0
268
         while i < len(trip):
269
             this_ts = trip[i]["timestamp"]
270
             if last_ts != None and this_ts - last_ts > TRIP_TIMEDELTA_SPLIT:
271
272
                  trips.append(trip[last_split:i])
                 last\_split = i
273
             last_ts = this_ts
274
             i += 1
275
         if trip[last_split:] != []:
276
             trips.append(trip[last_split:])
277
        return trips
278
279
    def get_vehicle_position(vehicle_id, dt):
280
        global vehicles
^{281}
282
         if vehicle_id in vehicles:
             for point in vehicles[vehicle_id]:
283
                 if (abs(point["timestamp"] - dt) < datetime.timedelta(seconds = 30) or
    point["timestamp"] > dt):
284
285
286
                      return point
             return None
287
288
         else:
289
             return None
290
291
    def trip_error_data(trip, station):
         error_data = {
292
             "schedule":
293
             "capmetro": [],
294
             "me": []
295
296
         arrival\_time = trip[-1]["timestamp"]
297
298
         this_ts = None
         this_dt_correct = None
299
         this\_sched = None
300
```

```
301
         this\_est = None
         for point in trip:
302
              this_ts = point ["timestamp"]
303
              vehicle_id = point["vehicle_id"]
304
              this_est = combine_time_date(point["est_arrival_time"], this_ts)
305
              this_sched = combine_time_date(point["sched_arrival_time"], this_ts)
306
             \# Calculate error and append to the lists.
307
              schedule_error_min = (this_sched - arrival_time).total_seconds() / 60
308
309
              capmetro_error_min = (this_est - arrival_time).total_seconds() / 60
              if abs(schedule_error_min) < ACCEPTABLE_ERROR_MIN:</pre>
310
                  error_data ["schedule"].append(schedule_error_min)
311
                  error_data["capmetro"].append(capmetro_error_min)
312
         return error_data
313
314
     def compute_stability(data):
315
         diffs = []
316
         i = 1
317
318
         while i < len(data):
              diffs.append(\mathbf{abs}(\,\mathrm{data}\,[\,i\,]\,-\,\mathrm{data}\,[\,i\,-\,1]))
319
320
         return statistics.mean(diffs)
321
322
     def analyze():
323
         def append_keys(dict1, dict2):
324
             for key, value in dict1.items():
325
                  dict2 [key].append(value)
326
327
         global stations
328
         load_data()
         metrics = {
329
              "schedule": {
330
                  "_data": [],
331
                  "stability": []
332
333
              "capmetro": {
334
                  "_data": [],
335
                  "stability": []
336
337
338
         for station in stations:
339
             for trip_id , trip_data in station.trips.items():
340
                  trips = split_unique_trips(trip_data)
341
                  for trip in trips:
342
                       metrics ["schedule"] ["_data"] += this_error_data ["schedule"] metrics ["capmetro"] ["_data"] += this_error_data ["capmetro"]
343
344
                       if len(this_error_data["schedule"]) > 1:
345
                           metrics["schedule"]["stability"].append(
346
                                    compute_stability(this_error_data["schedule"]))
347
                           metrics ["capmetro"] ["stability"]. append (
348
                                    compute_stability(this_error_data["capmetro"]))
349
350
         print("\nBUS_SCHEDULE_ONLY")
351
         print("Error_mean: _" + str(statistics.mean(metrics["schedule"]["_data"])))
352
         print("Error_std_dev:_" + str(statistics.stdev(metrics["schedule"]["_data"])))
353
         print("Stability: _" + str(statistics.mean(metrics["schedule"]["stability"])))
354
         print("\nCAPITAL_METRO_ESTIMATES")
355
         print("Error_mean: _" + str(statistics.mean(metrics["capmetro"]["_data"])))
356
         print("Error_std_dev:_" + str(statistics.stdev(metrics["capmetro"]["_data"])))
357
         print("Stability: _" + str(statistics.mean(metrics["capmetro"]["stability"])))
358
359
     if __name__ == "__main__":
360
         if sys.argv[1] == "collect":
361
362
              collect()
         elif sys.argv[1] == "analyze":
363
             analyze()
364
```

A.2 phaseII.py

```
import sys
   import time
   import datetime
   import pickle
   import sched
   import urllib.request
   import urllib.parse
   import xml.dom.minidom
8
   import \ \mathrm{math}
9
   import statistics
10
11
   DATA_FILE = "phaseII.pickle"
12
13
   CAPMETRO_NEXTBUS = "http://www.capmetro.org/planner/s_nextbus2.asp"
14
   CAPMETRO.BUSLOCS = "http://www.capmetro.org/planner/s_buslocation.asp"
15
16
   POLLINTERVAL = 30
   SAVE_INTERVAL = 15 * 60
17
18
   # Split trips at points that are this amount of time apart.
19
20
   TRIP_TIMEDELTA_SPLIT = datetime.timedelta(hours = 12)
   ACCEPTABLE\_ERROR\_MIN = 20
21
   DEFAULT\_SPEED\_KMH = 40
22
   ARRIVAL ESTIMATE INTERVAL = datetime.timedelta(minutes = 2)
23
24
25
   class Station:
        def __init__(self , stop_id , name, latitude , longitude):
26
27
            self.stop_id = stop_id
28
            self.name = name
            self.trips = \{\}
29
            self.latitude = latitude
30
31
            self.longitude = longitude
        def record_trip(self, timestamp, trip_id, vehicle_id, sched_arrival, est_arrival):
32
33
            def fix_time(time):
                time = time.strip()
34
                # Error parsing estimated arrival time '12:02 XM': time data '12:02 XM'
35
                \# does not match format '% I:%M %p'
36
                time = time.replace("X", "A")
37
                # Error parsing estimated arrival time '00:29 AM': time data '00:29 AM'
38
                # does not match format '%I:%M %p'
39
                time = time.replace("00", "12")
40
41
                return time
            if not trip_id in self.trips:
42
                 self.trips[trip\_id] = []
43
            sched_arrival_time = None
44
            sched_arrival = fix_time(sched_arrival)
45
46
            trv:
                 sched_arrival_time = datetime.datetime.strptime(sched_arrival,
47
                         "%I:%M_%p").time()
48
            except Exception as e:
49
                print("Error_parsing_scheduled_arrival_time_'" + sched_arrival + "':" +
50
                         \mathbf{str}(e)
51
52
            est_arrival_time = None
            est_arrival = fix_time(est_arrival)
53
54
            \mathbf{try}:
                 est_arrival_time = datetime.datetime.strptime(est_arrival,
55
                         "%I:%M_%p").time()
56
57
            except Exception as e:
                print("Error_parsing_estimated_arrival_time_'" + est_arrival + "':_"
58
                         + str(e))
59
            self.trips[trip\_id].append({
60
61
                 "timestamp": timestamp,
                "sched_arrival_time": sched_arrival_time,
62
                "est_arrival_time": est_arrival_time,
63
                "vehicle_id": vehicle_id
64
            })
65
66
   stations =
67
        Station (497, "UT_West_Mall", 30.286064, -97.741815),
68
```

```
70
71
72
    vehicles = \{\}
73
74
    \#\ Credit\ John\ D.\ Cook,\ http://www.johndcook.com/python\_longitude\_latitude.html
75
    # (public domain)
76
    def distance_on_unit_sphere(lat1, long1, lat2, long2):
77
78
        # Convert latitude and longitude to
79
         # spherical coordinates in radians.
80
         degrees_to_radians = math.pi/180.0
81
82
        \# phi = 90 - latitude
83
84
         phi1 = (90.0 - lat1)*degrees_to_radians
         phi2 = (90.0 - lat2)*degrees_to_radians
85
86
        \# theta = longitude
87
88
         theta1 = long1*degrees_to_radians
89
        theta2 = long2*degrees_to_radians
90
91
        # Compute spherical distance from spherical coordinates.
92
        # For two locations in spherical coordinates
93
94
        \# (1, theta, phi) and (1, theta, phi)
95
        \# cosine(arc length) =
              sin phi sin phi' cos(theta-theta') + cos phi cos phi'
96
        \# \ distance = rho * arc \ length
97
98
         \cos = (\mathrm{math.sin}\,(\,\mathrm{phi1}\,)*\,\mathrm{math.sin}\,(\,\mathrm{phi2}\,)*\,\mathrm{math.cos}\,(\,\mathrm{theta1}\,-\,\mathrm{theta2}\,)\,+
99
100
                math.cos(phi1)*math.cos(phi2))
101
         arc = math.acos(cos)
102
103
        # Remember to multiply arc by the radius of the earth
        \# in your favorite set of units to get length.
104
        return arc
105
106
107
    def dom_value(node):
         if node.firstChild == None:
108
             return ""
109
110
             return node.firstChild.data
111
112
    def combine_time_date(time, dt):
113
        \# Combines a time with date information from a datetime. We need some extra
114
        # logic to handle cases straddling midnight.
115
         midnight = datetime.time(0, 0)
116
         if time > midnight and dt.time() < midnight:</pre>
117
118
             dt-correct = datetime.timedelta(days = 1)
         elif time < midnight and dt.time() > midnight:
119
120
             dt\_correct = datetime.timedelta(days = -1)
         else:
121
             dt_correct = datetime.timedelta()
122
        return datetime.datetime.combine(dt.date(), time) + dt_correct
123
124
125
    def load_data():
        global stations
126
127
         global vehicles
128
         try:
             prev_data = pickle.load(open(DATA_FILE, "rb"))
129
130
             stations = prev_data["stations"]
             vehicles = prev_data["vehicles"]
131
         except Exception as e:
132
             print("Error_loading_data_file:_" + str(e))
133
134
135
    def save_data():
        global stations
136
```

```
137
        global vehicles
        print("Saving_data...")
138
         pickle.dump({ "stations": stations, "vehicles": vehicles },
139
                 open(DATA_FILE, "wb"))
140
141
    def retrieve_station_data(station):
142
         url = CAPMETRO.NEXTBUS + "?" + urllib.parse.urlencode({ "route": "801",
143
            "stopid": str(station.stop_id) })
144
        http_handle = None
145
146
        try:
             print("Downloading_" + url)
147
             http-handle = urllib.request.urlopen(url)
148
        except Exception as e:
149
            print ("Error_retrieving_data_for_station_" + str(station.stop_id) + ":_" +
150
                     str(e))
151
            return False
152
        xml_tree = None
153
154
        now = datetime.datetime.now()
        trv:
155
156
             xml_tree = xml.dom.minidom.parse(http_handle)
             runs = xml_tree.getElementsByTagName("Run")
157
             for trip in runs:
158
                 vehicle_id = dom_value(trip.getElementsByTagName("Vehicleid")[0]).strip()
159
                 if dom_value(trip.getElementsByTagName("Valid")[0]) == "Y":
160
                     station.record_trip(
161
162
                         timestamp = now,
                          trip_id = dom_value(trip.getElementsByTagName("Tripid")[0]).strip(),
163
164
                          vehicle_id = vehicle_id,
                          sched\_arrival = dom\_value(trip.getElementsByTagName("Triptime")[0])
165
                          est_arrival = dom_value(trip.getElementsByTagName("Estimatedtime")[0])
166
167
        except Exception as e:
168
             print("Error_parsing_tracking_data_for_station_" + str(station.stop_id) +
169
                     ":_" + str(e))
170
171
            return False
        return True
172
173
    def retrieve_vehicle_data():
174
175
        global vehicles
         url = CAPMETRO.BUSLOCS + "?" + urllib.parse.urlencode({ "route": "801" })
176
177
        trv:
             print("Downloading_" + url)
178
             http_handle = urllib.request.urlopen(url)
179
        except Exception as e:
180
            print("Error_retrieving_bus_tracking_data:_" + str(e))
181
            return False
182
        xml_tree = None
183
        now = datetime.datetime.now()
184
185
        try:
             xml_tree = xml.dom.minidom.parse(http_handle)
186
             xml_vehicles = xml_tree.getElementsByTagName("Vehicle")
187
188
             for vehicle in xml_vehicles:
                 vehicle_id = dom_value(vehicle.getElementsByTagName("Vehicleid")[0]).strip()
189
                 latest_pos = [float(x) for x in dom_value(
190
                          vehicle.getElementsByTagName("Position")[0]).split(",")]
191
                 if not vehicle_id in vehicles:
192
193
                     vehicles [vehicle_id] = []
                 vehicle_data = vehicles[vehicle_id]
194
195
                 # Do not record duplicate data (data that hasn't been updated within our
                 \# update interval).
196
                 if (len(vehicle_data) == 0 or
197
                          vehicle_data[-1]["latitude"] != latest_pos[0] or
198
                         vehicle_data[-1]["longitude"] != latest_pos[1]):
199
                     vehicle_data.append({
200
                          "timestamp": now,
201
202
                         "block": dom_value(
                                  vehicle.getElementsByTagName("Block")[0]).strip(),
203
                         "reliable": dom_value(
204
```

```
205
                                  vehicle.getElementsByTagName("Reliable")[0]).strip() == "Y",
                         "off_route": dom_value(
206
207
                                  vehicle.getElementsByTagName("Offroute")[0]).strip() == "Y",
                          "stopped": dom_value(
208
                                  vehicle.getElementsByTagName("Stopped")[0]).strip() == "Y",
209
                         "in_service": dom_value(
210
                                  vehicle.getElementsByTagName("Inservice")[0]).strip() = "Y",
211
                         "speed": float(dom_value(vehicle.getElementsByTagName("Speed")[0]))
212
                         "heading": int(dom_value(vehicle.getElementsByTagName("Heading")[0])),
213
                         "latitude": latest_pos[0]
214
215
                         "longitude": latest_pos[1]
216
                     })
        except Exception as e:
217
            print("Error_parsing_bus_tracking_data:_" + str(e))
218
            return False
219
220
        return True
221
222
    def collect():
        load_data()
223
224
         scheduler = sched.scheduler(time.time, time.sleep)
225
        def collect_stations(i):
             retrieve_station_data(stations[i])
226
227
             i = (i + 1) \% len(stations)
            now = datetime.datetime.now()
228
229
            now_time = now.time()
            # Cease collecting between 12:45 AM and 4:45 AM.
230
231
             if now_time \geq datetime.time(0, 45) and now_time \leq datetime.time(4, 45):
232
                 scheduler.enterabs(time.mktime(datetime.datetime.combine(now,
                         datetime.time(4, 45)).timetuple()), 2, collect_stations, (0,))
233
234
                 scheduler.enter(POLL_INTERVAL / len(stations), 2, collect_stations,
235
236
                         (i,))
        def collect_vehicles():
237
            retrieve_vehicle_data()
238
239
            now = datetime.datetime.now()
            now_time = now.time()
240
            # Cease collecting between 12:45 AM and 4:45 AM.
241
            if now_time >= datetime.time(0, 45) and now_time <= datetime.time(4, 45):
242
                 scheduler.enterabs(time.mktime(datetime.datetime.combine(now,
243
244
                         datetime.time(4, 45)).timetuple()), 1, collect_vehicles, ())
245
             else:
                 scheduler.enter(POLL_INTERVAL, 1, collect_vehicles, ())
246
        def collect_save():
247
             save_data()
248
             scheduler.enter(SAVE_INTERVAL, 3, collect_save, ())
249
250
         collect_vehicles()
251
         collect_stations(0)
252
         collect_save()
253
254
             scheduler.run()
255
256
        except KeyboardInterrupt:
            print("Stopped.")
257
             save_data()
258
259
            sys.exit(0)
260
261
    def split_unique_trips(trip):
        # Trips are distinguished by trip ID, but this is only unique for one day.
262
263
        # This function splits one "trip" into separate trips for each date.
        trips = []
264
265
         i = 0
266
         this\_ts = None
        last_ts = None
267
         last\_split = 0
268
        while i < len(trip):
269
270
             this_ts = trip[i]["timestamp"]
             if last_ts != None and this_ts - last_ts > TRIP_TIMEDELTA_SPLIT:
271
                 trips.append(trip[last_split:i])
272
```

```
273
                  last\_split = i
274
             last_ts = this_ts
275
             i += 1
         if trip[last_split:] != []:
276
             trips.append(trip[last_split:])
277
         return trips
278
279
    def get_vehicle_position(vehicle_id, dt):
280
281
         global vehicles
         if vehicle_id in vehicles:
282
             for point in vehicles [vehicle_id]:
283
                  if (abs(point["timestamp"] - dt) < datetime.timedelta(seconds = 30) or
    point["timestamp"] > dt):
284
285
286
                      return point
287
             return None
288
         else:
             return None
289
290
    \mathbf{def}\ estimate\_arrival\_time\,(\,vehicle\_id\,\,,\,\,dt\,,\,\,scheduled\_dt\,\,,\,\,station\,)\colon
291
292
         if scheduled_dt - dt <= datetime.timedelta(minutes = 30):
             pos = get_vehicle_position(vehicle_id, dt)
293
             if (pos != None and not pos["off_route"] and
294
                  pos ["latitude"] != 0 and pos ["longitude"] != 0):

if station.name == "Pleasant_Hill" or station.name == "Crestview":
295
296
                      traffic_density = 15
297
298
                      station_density = 2
                  elif station.name = "UT_West_Mall" or station.name = "Republic_Square":
299
300
                      traffic_density = 30
                      station_density = 3
301
                  dist_km = distance_on_unit_sphere(pos["latitude"], pos["longitude"],
302
303
                           station.latitude, station.longitude) * 6373
                  delay_min = (0.4855 + 0.0287 * traffic_density / 8 + 0.0168 *
304
305
                           traffic_density + 0.9654 * dist_km - 1.1969 * 0.5 + 0.1130 *
                           dist_km * station_density)
306
307
                  return dt + datetime.timedelta(hours = dist_km / DEFAULT_SPEED_KMH +
                           delay_min / 60)
308
309
             else:
310
                  return scheduled_dt
         else:
311
312
             return scheduled_dt
313
314
    def trip_error_data(trip, station):
315
         error_data = \{
             "schedule": [],
316
             "capmetro": [],
317
             "me": []
318
319
         arrival_time = trip[-1]["timestamp"]
320
         this\_ts = None
321
322
         this_dt_correct = None
         this_sched = None
323
324
         this_est = None
         this_my_est = None
325
326
         this_my_est_last_dt = None
327
         for point in trip:
328
             this_ts = point["timestamp"]
             vehicle_id = point["vehicle_id"]
329
             this_est = combine_time_date(point["est_arrival_time"], this_ts)
330
331
             this_sched = combine_time_date(point["sched_arrival_time"], this_ts)
             if (this_my_est_last_dt == None or
332
                       this_ts - this_my_est_last_dt > ARRIVAL_ESTIMATE_INTERVAL):
333
334
                  this_my_est = estimate_arrival_time(vehicle_id, this_ts, this_sched,
                           station)
335
                  this_my_est_last_dt = this_ts
336
             \# Calculate error and append to the lists.
337
338
             schedule_error_min = (this_sched - arrival_time).total_seconds() / 60
             capmetro_error_min = (this_est - arrival_time).total_seconds() / 60
339
             me_error_min = (this_my_est - arrival_time).total_seconds() / 60
340
```

```
341
              if abs(schedule_error_min) < ACCEPTABLE_ERROR_MIN:</pre>
                   error_data ["schedule"].append(schedule_error_min)
error_data ["capmetro"].append(capmetro_error_min)
342
343
                   error_data["me"].append(me_error_min)
344
         return error_data
345
346
     def compute_stability(data):
347
          diffs = []
348
349
         i = 1
         while i < len(data):
350
              diffs.append(abs(data[i] - data[i - 1]))
351
              i += 1
352
         return statistics.mean(diffs)
353
354
     def analyze():
355
356
         def append_keys(dict1, dict2):
              for key, value in dict1.items():
357
358
                   dict2 [key].append(value)
          global stations
359
360
          load_data()
361
          metrics = {
              362
363
364
365
               capmetro": {
366
                   "_data": [],
367
                   "stability": []
368
369
              "me": {
370
                   "_data": [],
371
                   "stability": []
372
373
374
375
         for station in stations:
              for trip_id , trip_data in station.trips.items():
376
                   trips = split_unique_trips(trip_data)
377
                   for trip in trips:
378
                        metrics["schedule"]["_data"] += this_error_data["schedule"]
379
                        metrics ["capmetro"] ["_data"] += this_error_data ["capmetro"]
380
                        metrics["me"]["_data"] += this_error_data["me"]
381
                        if len(this_error_data["schedule"]) > 1:
   metrics["schedule"]["stability"].append(
382
383
                                      compute_stability(this_error_data["schedule"]))
384
                            metrics ["capmetro"] ["stability"].append(
385
                                      compute_stability(this_error_data["capmetro"]))
386
                             metrics ["me"] ["stability"].append(
387
                                      compute_stability(this_error_data["me"]))
388
389
         print("\nBUS_SCHEDULE_ONLY")
390
         print("Error_mean: _" + str(statistics.mean(metrics["schedule"]["_data"])))
391
         print("Error_std_dev: _" + str(statistics.stdev(metrics["schedule"]["_data"])))
392
         print("Stability: _" + str(statistics.mean(metrics["schedule"]["stability"])))
393
         print("\nCAPITAL_METRO_ESTIMATES")
394
         print("Error_mean:_" + str(statistics.mean(metrics["capmetro"]["_data"])))
print("Error_std_dev:_" + str(statistics.stdev(metrics["capmetro"]["_data"])))
395
396
         print("Stability: " + str(statistics.mean(metrics["capmetro"]["stability"])))
397
         print("\nMY_ESTIMATES")
398
         print("Error_mean:_" + str(statistics.mean(metrics["me"]["_data"])))
print("Error_std_dev:_" + str(statistics.stdev(metrics["me"]["_data"])))
399
400
         print("Stability: " + str(statistics.mean(metrics["me" | ["stability"])))
401
402
     if = name = " = main = ":
403
          if sys.argv[1] = "collect":
404
              collect()
405
406
          elif sys.argv[1] == "analyze":
              analyze()
407
```

A.3 phaseIII.py

```
import sys
   import time
   import datetime
   import pickle
4
   import sched
   import urllib.request
   import urllib.parse
   import xml.dom.minidom
   import math
9
10
   import statistics
11
   DATA_FILE = "phaseIII.pickle"
12
13
   CAPMETRO_NEXTBUS = "http://www.capmetro.org/planner/s_nextbus2.asp"
14
   CAPMETRO_BUSLOCS = "http://www.capmetro.org/planner/s_buslocation.asp"
15
   POLL_INTERVAL = 30
16
   SAVE_INTERVAL = 15 * 60
17
18
   # Split trips at points that are this amount of time apart.
19
   TRIP_TIMEDELTA_SPLIT = datetime.timedelta(hours = 12)
20
   ACCEPTABLE\_ERROR\_MIN = 20
21
   DEFAULT\_SPEED\_KMH = 40
   ARRIVAL ESTIMATE INTERVAL = datetime.timedelta(minutes = 2)
23
24
25
        def __init__(self, stop_id, name, latitude, longitude):
26
            self.stop_id = stop_id
27
            self.name = name
28
            self.trips = \{\}
29
            self.latitude = latitude
30
            self.longitude = longitude
31
            self._my_est_last = None
32
            self._my_est_last_dt = None
33
        def record_trip(self, timestamp, trip_id, vehicle_id, sched_arrival,
34
35
                 est_arrival , my_est_arrival ):
            if not trip_id in self.trips:
36
37
                self.trips[trip_id] = []
            data = {
38
                 "timestamp": timestamp,
39
                 "sched_arrival": sched_arrival,
40
                "est_arrival": est_arrival,
41
                 "vehicle_id": vehicle_id
42
43
            if (self._my_est_last == None or
44
                     timestamp - self._my_est_last_dt > ARRIVAL_ESTIMATE_INTERVAL):
45
                 data["my_est_arrival"] = my_est_arrival
46
                 self.\_my\_est\_last = my\_est\_arrival
47
                 self._my_est_last_dt = timestamp
48
49
                data["my_est_arrival"] = self._my_est_last
50
            self.trips[trip_id].append(data)
51
52
    stations = [
53
        Station (497, "UT_West_Mall", 30.286064, -97.741815),
54
        Station (5867, "Republic_Square", 30.267751, -97.746857), Station (5606, "Crestview", 30.337852, -97.719035),
55
56
        Station (5872, "Pleasant_Hill", 30.192391, -97.779203)
57
58
59
    vehicles = \{\}
60
   \# Credit John D. Cook, http://www.johndcook.com/python_longitude_latitude.html
   \# (public domain)
62
    def distance_on_unit_sphere(lat1, long1, lat2, long2):
63
64
        # Convert latitude and longitude to
65
```

```
# spherical coordinates in radians.
         degrees_to_radians = math.pi/180.0
67
68
         \# phi = 90 - latitude
69
         phi1 = (90.0 - lat1)*degrees_to_radians
70
         phi2 = (90.0 - lat2)*degrees_to_radians
71
72
         \# theta = longitude
73
74
         theta1 = long1*degrees_to_radians
         theta2 = long2*degrees_to_radians
75
76
         # Compute spherical distance from spherical coordinates.
77
78
         # For two locations in spherical coordinates
79
         # (1, theta, phi) and (1, theta, phi)
80
81
         \# cosine(arc length) =
              sin phi sin phi' cos(theta-theta') + cos phi cos phi'
82
83
         \# \ distance = rho * arc \ length
84
85
         \cos = (\text{math.}\sin(\text{phi1})*\text{math.}\sin(\text{phi2})*\text{math.}\cos(\text{theta1} - \text{theta2}) +
86
                 \operatorname{math.cos}(\operatorname{phi1})*\operatorname{math.cos}(\operatorname{phi2}))
87
         arc = math.acos(cos)
88
         # Remember to multiply arc by the radius of the earth
89
         # in your favorite set of units to get length.
90
         return arc
91
92
    def dom_value(node):
93
         if node.firstChild == None:
94
             return ""
95
96
         else:
             return node.firstChild.data
97
98
    def load_data():
99
100
         global stations
         global vehicles
101
102
             prev_data = pickle.load(open(DATA_FILE, "rb"))
103
             stations = prev_data["stations"]
104
              vehicles = prev_data["vehicles"]
105
106
         except Exception as e:
             print("Error_loading_data_file:_" + str(e))
107
108
    def save_data():
109
         global stations
110
         global vehicles
print("Saving_data...")
111
112
         pickle.dump({ "stations": stations, "vehicles": vehicles },
113
                  open(DATA_FILE, "wb"))
114
115
    def parse_time(time_s, now_dt):
116
117
         time_s = time_s.strip()
         # Error parsing estimated arrival time '12:02 XM': time data '12:02 XM'
118
         # does not match format '%I:%M %p'
119
         time_s = time_s.replace("X", "A")
120
         # Error parsing estimated arrival time '00:29 AM': time data '00:29 AM'
121
         # does not match format '%I:%M %p
122
         time_s = time_s.replace("00", "12")
123
124
             time = datetime.datetime.strptime(time_s, "%I:%ML%p").time()
125
         except Exception as e:
126
             print("Error_parsing_time_'" + time_s + "':" + str(e))
127
             return None
128
         # We need some extra logic to handle cases straddling midnight.
129
         if (time > datetime.time(0, 0)) and
130
131
                  now_dt.time() < datetime.time(0, 0)):
              dt_{-correct} = datetime.timedelta(days = 1)
132
         elif (time < datetime.time(0, 0) and
133
```

```
134
                  now_dt.time() > datetime.time(0, 0)):
135
             dt\_correct = datetime.timedelta(days = -1)
136
         else:
              dt_correct = datetime.timedelta()
137
         return datetime.datetime.combine(now_dt.date(), time) + dt_correct
138
139
    def get_vehicle_position(vehicle_id, dt):
140
         global vehicles
141
142
         if vehicle_id in vehicles:
             for point in vehicles[vehicle_id]:
143
                  if point ["timestamp"] > dt - datetime.timedelta(seconds = 30):
144
145
                      return point
146
             return None
147
         else:
             return None
148
149
    \mathbf{def}\ estimate\_arrival\_time\,(\,vehicle\_id\,\,,\,\,dt\,,\,\,scheduled\_dt\,\,,\,\,station\,\,)\colon
150
151
         if scheduled_dt - dt <= datetime.timedelta(minutes = 30):
             pos = get_vehicle_position(vehicle_id, dt)
152
153
              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":
154
155
                       traffic_density = 15
156
                      station_density = 2
157
                  elif station.name = "UT_West_Mall" or station.name == "Republic_Square":
158
                       traffic_density = 30
159
                      station\_density = 3
160
                  dist\_km \ = \ distance\_on\_unit\_sphere \, (\,pos\,[\,"\,latitude\,"\,] \,\,, \ pos\,[\,"\,longitude\,"\,] \,\,,
161
                           station.latitude, station.longitude) * 6373
162
                  delay_min = (0.4855 + 0.0287 * traffic_density / 8 + 0.0168 *
163
                           traffic\_density \ + \ 0.9654 \ * \ dist\_km \ - \ 1.1969 \ * \ 0.5 \ + \ 0.1130 \ *
164
                           dist_km * station_density)
165
166
                  return dt + datetime.timedelta(hours = dist_km / DEFAULT_SPEED_KMH +
                           delay_min / 60)
167
168
             else:
                  return scheduled_dt
169
         else:
170
171
             return scheduled_dt
172
    def retrieve_station_data(station):
173
         url = CAPMETRO_NEXTBUS + "?" + urllib.parse.urlencode({ "route": "801",
174
             "stopid": str(station.stop_id) })
175
         http_handle = None
176
177
         try:
             print("Downloading_" + url)
178
             http_handle = urllib.request.urlopen(url)
179
         except Exception as e:
180
             print("Error_retrieving_data_for_station_" + str(station.stop_id) + ":_" +
181
182
                      str(e))
183
             return False
         xml_tree = None
184
185
         now = datetime.datetime.now()
         try:
186
             xml_tree = xml.dom.minidom.parse(http_handle)
187
188
             runs = xml_tree.getElementsByTagName("Run")
189
             for trip in runs:
                  vehicle_id = dom_value(trip.getElementsByTagName("Vehicleid")[0]).strip()
190
                  if dom_value(trip.getElementsByTagName("Valid")[0]) == "Y":
191
192
                       station.record_trip(
                           timestamp = now,
193
                           trip_id = dom_value(trip.getElementsByTagName("Tripid")[0]).strip(),
194
195
                           vehicle_id = vehicle_id,
                           sched_arrival = parse_time(dom_value(
196
                                    trip.getElementsByTagName("Triptime")[0]), now),
197
                           est_arrival = parse_time(dom_value(
198
199
                                    trip.getElementsByTagName("Estimatedtime")[0]), now),
                           my_est_arrival = estimate_arrival_time(vehicle_id, now, parse_time(
200
                                    dom_value(trip.getElementsByTagName("Triptime")[0]), now), station)
201
```

```
202
        except Exception as e:
203
204
            print("Error_parsing_tracking_data_for_station_" + str(station.stop_id) +
                     ": _" + str(e))
205
            return False
206
        return True
207
208
    def retrieve_vehicle_data():
209
        global vehicles
210
        url = CAPMETRO.BUSLOCS + "?" + urllib.parse.urlencode({ "route": "801" })
211
212
             print("Downloading_" + url)
213
             http_handle = urllib.request.urlopen(url)
214
215
        except Exception as e:
            print("Error_retrieving_bus_tracking_data:_" + str(e))
216
217
            return False
        xml_tree = None
218
219
        now = datetime.datetime.now()
        trv:
220
221
             xml_tree = xml.dom.minidom.parse(http_handle)
             xml_vehicles = xml_tree.getElementsByTagName("Vehicle")
222
             for vehicle in xml_vehicles:
223
                 vehicle_id = dom_value(vehicle.getElementsByTagName("Vehicleid")[0]).strip()
224
                 latest_pos = [float(x) for x in dom_value(
225
                         vehicle.getElementsByTagName("Position")[0]).split(",")]
226
                 if not vehicle_id in vehicles:
227
228
                     vehicles [vehicle_id] = []
229
                 vehicle_data = vehicles [vehicle_id]
                 # Do not record duplicate data (data that hasn't been updated within our
230
                 \# update interval).
231
232
                 if (len(vehicle_data) = 0 or
                         vehicle_data[-1]["latitude"] != latest_pos[0] or
233
                         vehicle_data[-1]["longitude"] != latest_pos[1]):
234
                     vehicle_data.append({
235
236
                         "timestamp": now,
                         "block": dom_value(
237
                                  vehicle.getElementsByTagName("Block")[0]).strip(),
238
                         "reliable": dom_value(
239
                                  vehicle.getElementsByTagName("Reliable")[0]).strip() == "Y",
240
                         "off_route": dom_value(
241
                                  vehicle.getElementsByTagName("Offroute")[0]).strip() == "Y",
242
                         "stopped": dom_value(
243
                                  vehicle.getElementsByTagName("Stopped")[0]).strip() == "Y",
244
                         "in_service": dom_value(
245
                                  vehicle.getElementsByTagName("Inservice")[0]).strip() = "Y",
246
                         "speed": float (dom_value(vehicle.getElementsByTagName("Speed")[0]))
247
                         "heading": int(dom_value(vehicle.getElementsByTagName("Heading")[0])),
248
                         "latitude": latest_pos[0]
249
                         "longitude": latest_pos[1]
250
251
                     })
        except Exception as e:
252
253
             print("Error_parsing_bus_tracking_data:_" + str(e))
            return False
254
        return True
255
256
257
    def collect():
258
        load_data()
        scheduler = sched.scheduler(time.time, time.sleep)
259
260
        def collect_stations(i):
             retrieve_station_data(stations[i])
261
             i = (i + 1) \% len(stations)
262
263
            now = datetime.datetime.now()
            now_time = now.time()
264
            # Cease collecting between 12:45 AM and 4:45 AM.
265
             if now_time >= datetime.time(0, 45) and now_time <= datetime.time(4, 45):
266
267
                 scheduler.enterabs(time.mktime(datetime.datetime.combine(now,
                         datetime.time(4, 45)).timetuple()), 2, collect_stations, (0,))
268
             else:
269
```

```
270
                  scheduler.enter(POLL_INTERVAL / len(stations), 2, collect_stations,
271
                          (i,))
272
         def collect_vehicles():
273
             retrieve_vehicle_data()
             now = datetime.datetime.now()
274
             now_time = now.time()
275
             # Cease collecting between 12:45 AM and 4:45 AM.
276
             if now_time >= datetime.time(0, 45) and now_time <= datetime.time(4, 45):
277
278
                  scheduler.enterabs(time.mktime(datetime.datetime.combine(now.
                          datetime.time(4, 45)).timetuple()), 1, collect_vehicles, ())
279
280
             else:
                  scheduler.enter(POLL_INTERVAL, 1, collect_vehicles, ())
281
         def collect_save():
282
283
             save_data()
             scheduler.enter(SAVE_INTERVAL, 3, collect_save, ())
284
285
         collect_vehicles()
286
287
         collect_stations(0)
         collect_save()
288
289
         try:
290
             scheduler.run()
         except KeyboardInterrupt:
291
292
             print("Stopped.")
             save_data()
293
294
             sys.exit(0)
295
296
    def split_unique_trips(trip):
         # Trips are distinguished by trip ID, but this is only unique for one day.
297
         # This function splits one "trip" into separate trips for each date.
298
         trips = []
299
         i = 0
300
         this_ts = None
301
         last_ts = None
302
         last\_split = 0
303
304
         while i < len(trip):
             this_ts = trip[i]["timestamp"]
305
             if last_ts != None and this_ts - last_ts > TRIP_TIMEDELTA_SPLIT:
306
                  trips.append(trip[last_split:i])
307
                  last\_split = i
308
309
             last_ts = this_ts
             i += 1
310
         if trip[last_split:] != []:
311
             trips.append(trip[last_split:])
312
         return trips
313
314
    def get_station_arrival_time(search_start_dt, vehicle_id, station):
315
         global stations
316
         for this_station in stations:
317
             for trip_id, trip in this_station.trips.items():
318
                  if len(trip) > 0 and trip[0]["vehicle_id"] == vehicle_id:
319
                      for point in trip:
320
321
                           ts = point ["timestamp"]
                           vehicle_pos = get_vehicle_position(vehicle_id, ts)
322
                           if (ts > search_start_dt and
323
                                   ts \ - \ search\_start\_dt \ < \ TRIP\_TIMEDELTA\_SPLIT \ \ \textbf{and}
324
325
                                   vehicle_pos != None):
326
                               dist_m = distance_on_unit_sphere(station.latitude, station.longitude,
                                        vehicle\_pos\,["\,latitude"\,]\,\,,\ vehicle\_pos\,["\,longitude"\,])\ *\ 6373000
327
328
                               if dist_m < ARRIVAL_DISTANCE_M:</pre>
                                   return ts
329
         return None
330
331
    def trip_error_data(trip, station):
332
         error_data = {
333
             "schedule":
334
             "capmetro": [],
335
             "me": []
336
         }
337
```

```
338
          arrival_time = trip[-1]["timestamp"]
339
         for point in trip:
340
              # Calculate error and append to the lists.
              schedule_error_min = (point["sched_arrival"] - arrival_time).total_seconds() / 60
341
              capmetro_error_min = (point["est_arrival"] - arrival_time).total_seconds() / 60
342
              my_error_min = (point["my_est_arrival"] - arrival_time).total_seconds() / 60
343
              if (abs(schedule_error_min) < ACCEPTABLE_ERROR_MIN):</pre>
344
                   error_data ["schedule"].append(schedule_error_min)
error_data ["capmetro"].append(capmetro_error_min)
345
346
                   error_data["me"].append(my_error_min)
347
         return error_data
348
349
     def compute_stability(data):
350
          diffs = []
351
          i = 1
352
         while i < len(data):
353
              diffs.append(abs(data[i] - data[i - 1]))
354
355
         return statistics.mean(diffs)
356
357
     def analyze():
358
         \mathbf{def} append_keys(dict1, dict2):
359
              for key, value in dict1.items():
360
                   dict2 [key].append(value)
361
362
         global stations
363
         load_data()
364
          metrics = {
              "schedule": {
365
                   "_data": []
366
                   "stability": []
367
              },
"capmetro": {
368
369
                   "_data": []
370
                   "stability": []
371
             372
373
374
                   "stability": []
375
376
377
         for station in stations:
378
              for trip_id, trip_data in station.trips.items():
379
380
                   trips = split_unique_trips(trip_data)
                   for trip in trips:
381
                        this_error_data = trip_error_data(trip, station)
382
                        metrics["schedule"]["_data"] += this_error_data["schedule"]
metrics["capmetro"]["_data"] += this_error_data["capmetro"]
metrics["me"]["_data"] += this_error_data["me"]
383
384
385
                        if len(this_error_data["schedule"]) > 1:
386
                             metrics ["schedule"] ["stability"]. append (
387
                                      compute_stability(this_error_data["schedule"]))
388
                             metrics ["capmetro"] ["stability"]. append (
389
                                      compute_stability(this_error_data["capmetro"]))
390
                             metrics ["me"]["stability"].append(
391
                                      compute_stability(this_error_data["me"]))
392
393
         print("\nBUS_SCHEDULE_ONLY")
394
         print("Error_mean: _" + str(statistics.mean(metrics["schedule"]["_data"])))
395
         print("Error_std_dev:_" + str(statistics.stdev(metrics["schedule"]["_data"])))
396
         print("Stability: _" + str(statistics.mean(metrics["schedule"]["stability"])))
397
         print("\nCAPITAL_METRO_ESTIMATES")
print("Error_mean: _" + str(statistics.mean(metrics["capmetro"]["_data"])))
398
399
         print("Error_std_dev:_" + str(statistics.stdev(metrics["capmetro"]["_data"])))
400
         print("Stability:_" + str(statistics.mean(metrics["capmetro"]["stability"])))
401
         print("\nMY_ESTIMATES")
402
         print("Error_mean:_" + str(statistics.mean(metrics["me"]["_data"])))
print("Error_std_dev:_" + str(statistics.stdev(metrics["me"]["_data"])))
403
404
         print ("Stability: " + str (statistics.mean (metrics ["me"] ["stability"])))
405
```