Final report to the Oklahoma Department of Transportation on

AUTOMATING TURNING MOVEMENT STUDIES UTILIZING NEW SEGMENTED SENSOR TECHNOLOGY

Field of Scientific Research: Sensor system for traffic counts

Principal Investigator: Dr. Sridhar Radhakrishnan
School of Computer Science
200 Felgar Street, Room 114
University of Oklahoma
Norman, OK 73019
Phone: (405) 325-1867, Fax: (405) 325-4044
Email: sridhar@ou.edu

AUTOMATING TURNING MOVEMENT STUDIES
UTILIZING NEW SEGMENTED SENSOR TECHNOLOGY

Sridhar Radhakrishnan, Professor,
School of Computer Science
200 Felgar Street, Room 110
University of Oklahoma
Norman, Oklahoma 73019.
sridhar@ou.edu

Abstract

The Oklahoma Department of Transportation (ODOT) performs approximately 200 16-hour manual turning movement count (TMC) studies each year. These studies provide vital information for intersection improvement projects, such as stop sign and traffic signal warrants and other pedestrian safety projects. Within ODOT, the demand for these studies has outstripped the capacity to respond to request and provide the data in a timely manner. Typical times from request to receipt of data are currently six to eight weeks.

This research investigated the feasibility of automating intersection data collection. Towards this goal, the researchers at the University of Oklahoma completed a number of studies and created a novel algorithm for processing timestamp data for a two-lane intersection. We believe the technology developed can be successfully applied to improve the efficiency of turning movement studies.

1. Introduction and Background

Historically, manual TMCs have been the preferred count type because of the information detail possible with discernment that comes along with human observation. A TMC involves recording the "from-to" movements of each vehicle entering an intersection during the study period. Typically, any vehicle approaching an intersection

has three potential moves: right, left, or straight. Therefore, a typical intersection with four approaches will have twelve moves. Specific intersections may be more or less complex, but considering just the basic configuration, it is easy to appreciate the effort that would be required for manual data collection at a busy intersection. The practical implication is that the time and effort required is high in relation to the amount of data produced, but that data is vital and the manual TMC has been the only viable method of collection. Another practical implication of reliance on human observation is that studies are limited in duration. In other areas of vehicle data collection that are automated, a standard practice is to perform studies for at least 24 continuous hours.

In this research we investigated the feasibility of automating TMC studies, utilizing a new portable segmented axle sensor technology. Additionally, we have developed software to generate sample data, and algorithms to decode the timestamp data.

2. Research Tasks

The project objectives involved the completion of the following tasks:

- Study the complexity and feasibility of the problem for a simple intersection such as US-62 and SH-102, as pictured in Figure 1.
- Generate software capable of interfacing to provided sample data acquisition hardware
- Generate software capable of producing sample data for a two-lane intersection
- Creation of algorithms to process timing data

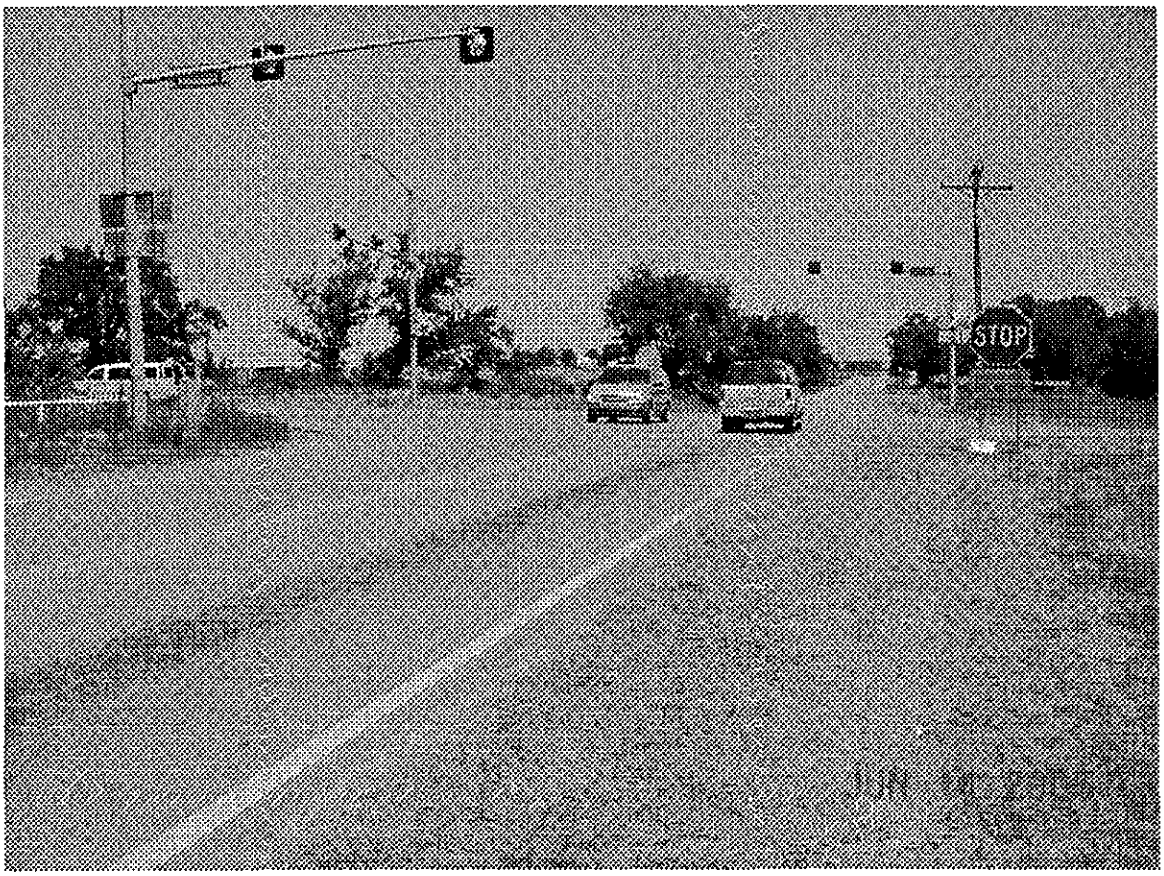

Figure 1: Example intersection to be studied is the intersection of US Highway 62 and State Highway 102.

3. Technology

The new sensor technology surpasses current technologies through its unique capability of distinguishing the position of multiple simultaneous points of contact or changes in contact pressure along its length. The basic sensor is a thin, flexible, lightweight, pressure sensitive device that, when actuated, provides discrete contact closures in a series of separate segments along its length. The individual sensor segments are adjustable in length and sensitivity. These properties allow the sensor to be deployed in such a way that a vehicle traveling thru an intersection will be detected by time and position both entering and exiting. The new event logging technology, micro processor based hardware, required to record these events will be developed thru a concurrent research project sponsored by the Florida Department of Transportation. Florida State University

coordinated with the University of Oklahoma to assure compatibility between the software and hardware development.

3. Data Acquisition Hardware and Software

We were provided with sample hardware consisting of two rubberized sensor strips, and an electronic hardware interface. The interface hardware, shown in Figure 2, provided a serial port with which to capture the data, which was provided in the format shown in Figure 3.

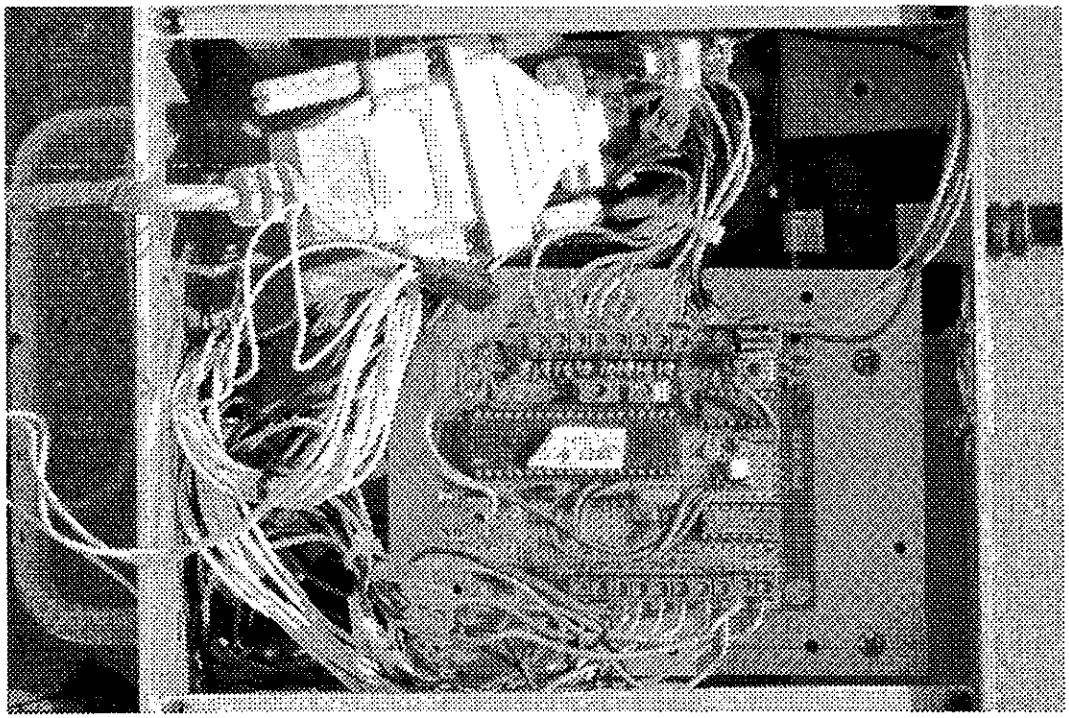

Figure 2: Sensor interface prototype hardware provided to OU

```
...
Reset AGTRSW102
begin
1 1 - 2 0 - 3 0 - 4 0 - 5 0 - 6 0 - 7 0 - 8 0 -
1 2 - 2 0 - 3 0 - 4 0 - 5 0 - 6 0 - 7 0 - 8 0 -
1 3 - 2 0 - 3 0 - 4 0 - 5 0 - 6 0 - 7 0 - 8 0 -
1 4 - 2 0 - 3 0 - 4 0 - 5 0 - 6 0 - 7 0 - 8 0 -
1 5 - 2 0 - 3 0 - 4 0 - 5 0 - 6 0 - 7 0 - 8 0 -
1 6 - 2 0 - 3 0 - 4 0 - 5 0 - 6 0 - 7 0 - 8 0 -
1 7 - 2 0 - 3 0 - 4 0 - 5 0 - 6 0 - 7 0 - 8 0 -
1 7 - 2 1 - 3 0 - 4 0 - 5 0 - 6 0 - 7 0 - 8 0 -
1 7 - 2 2 - 3 0 - 4 0 - 5 0 - 6 0 - 7 0 - 8 0 -
idle
idle
...
```

Figure 3: Sample output from sensor interface hardware.

A new line was written to the serial interface every time a segment was activated. Each line contained a count of the number of events on that segment. In the sample data, segment one is activated seven times and then segment two is activated two times. Unfortunately, the hardware is not capable of timestamping each event. Consequently, we created an interface program to precisely time the occurrence of each line, and note any differences in the number of events registered. If a counter had changed, an event was generated carrying the timestamp of the line. The graphical user interface of this software is shown in Figure 4.

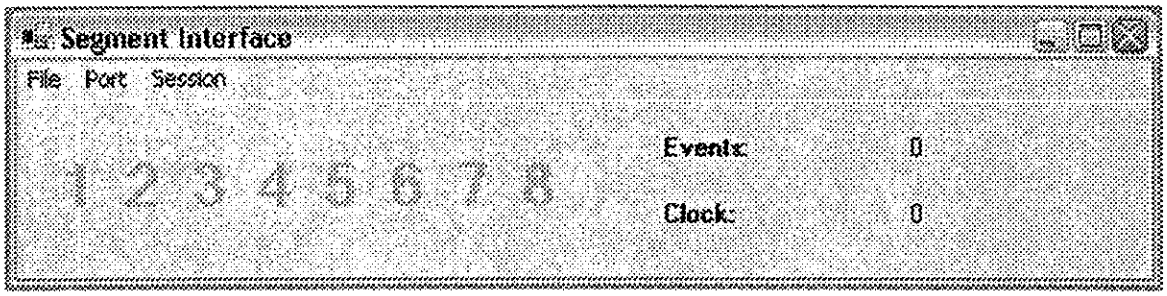

Figure 4: Hardware interface software. The large gray numbers flash when an event occurs, and two counters keep track of the current clock and number of events.

When the circuit is activated, the appropriate segment number flashes to provide feedback, the event is logged, and the event counter is incremented to provide a rough estimate of the number of vehicles recorded so far. The software was tested by bending the sensor strip to generate events.

A sample of the output from the interfacing software is shown in Figure 5. The software translates segment activation events into an event list consisting of the segment and a timestamp, separated by comma.

```
...
3, 145419.953624
3, 145420.2340272
2, 145421.7161584
2, 145421.9464896
4, 145429.8778944
4, 145430.1082256
1, 145431.4100976
1, 145431.670472
3, 145441.2442384
3, 145441.5146272
...
```

Figure 5: Output from interfacing software

Next, we attempted to test the complete sensor system's performance by deploying a strip in a residential neighborhood and recording the results with the software running on a laptop, as seen in Figure 6. A variety of vehicles from sedans, to compacts, to pickup trucks passed over the strip at approximately 25 miles per hour. Of approximately one dozen vehicles observed, only a large, heavy Chevrolet Z-71 pickup truck activated the sensor strip. The strip was flipped over to no effect.

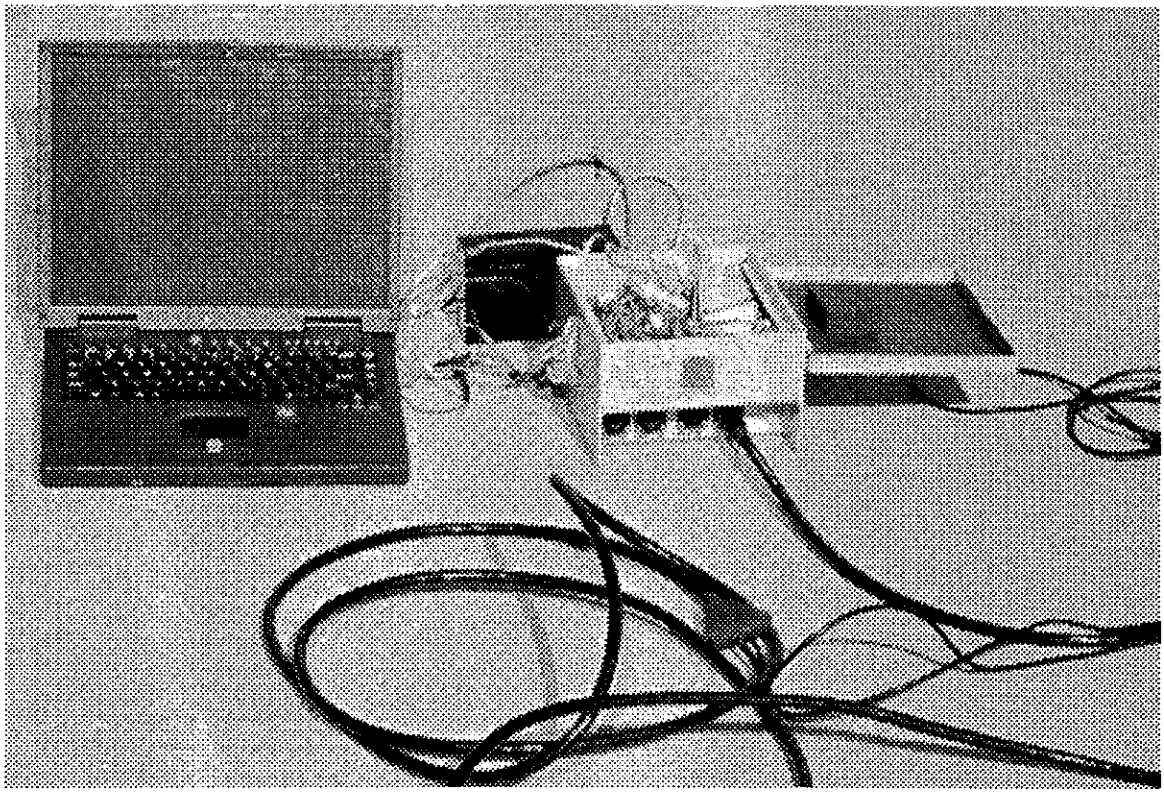

Figure 6: The sensing hardware and laptop connected to the test strip harness.

We were able to verify that time stamping and interfacing to the sensor hardware was possible with our software. However, due to a number of limitations in the system, we were unable to use the provided hardware and our software to record sample data. The most significant problem was a lack of test strips to fully cover a four-way, two-lane intersection, such as that of US-62 and SH102. Secondly, even with two additional test strips, a second data acquisition box would have been necessary at the corner opposite of the first box. Thirdly, the two boxes would need to be synchronized to generate proper data. Finally, the test strips were not sensitive enough to record most traffic. We were later informed that the strips may perform better on a day with a much higher ambient temperature. The original test was conducted at approximately 65 degrees Fahrenheit.

4. Data Generation

Because we were unable to collect real data from our model intersection due to a lack of proper hardware, we generated sample data with software. The first data generator, employed early in the project, used mathematical algorithms to generate the data. The result was very precise data that was also unrealistic and too uniform. Additionally, it was too difficult to simulate complex combinations of movements, with multiple vehicles in the intersection at the same time. Consequently, the data analysis software worked artificially-well, because the two programs were, mathematically speaking, inverse functions.

The final solution came after the completion of the hardware interfacing software. A new program was created on top of the interface software presenting the user with a user interface consisting of eight buttons arranged like sensors on a simple two-lane, four-way intersection, as seen in Figure 7. To generate data, the user clicks the buttons in real time to correspond to axle strikes. For example, to generate a right turn, the user would click the button 3 once for each axle, and then sometime later, click button 2, once per axle. This method produces data that is more realistic and "human" than the mathematically-generated data, and allows for complex turn combinations where several vehicles are in the intersection at a time. A sample of this data is provided in Figure 8.

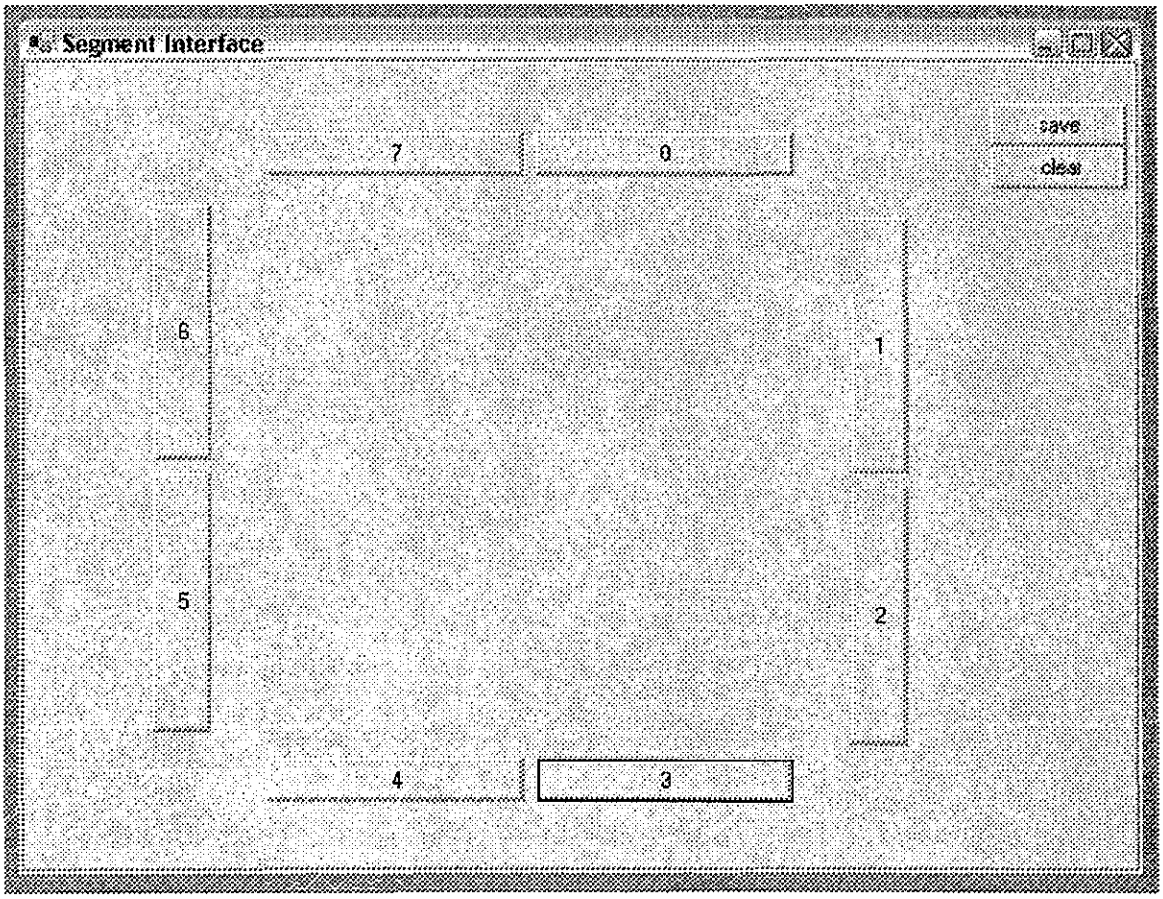

Figure 7: Data-generation program GUI

```
...
3, 145399.0435568
3, 145399.273888
0, 145401.1465808
0, 145401.3969408
3, 145411.84196
3, 145412.0823056
...
```

Figure 8: Example of sample data produced by program in Figure 7.

The data is simply the segment number followed by a time stamp. In this case, two-axle vehicles were simulated performing a variety of movements from simple to complex.

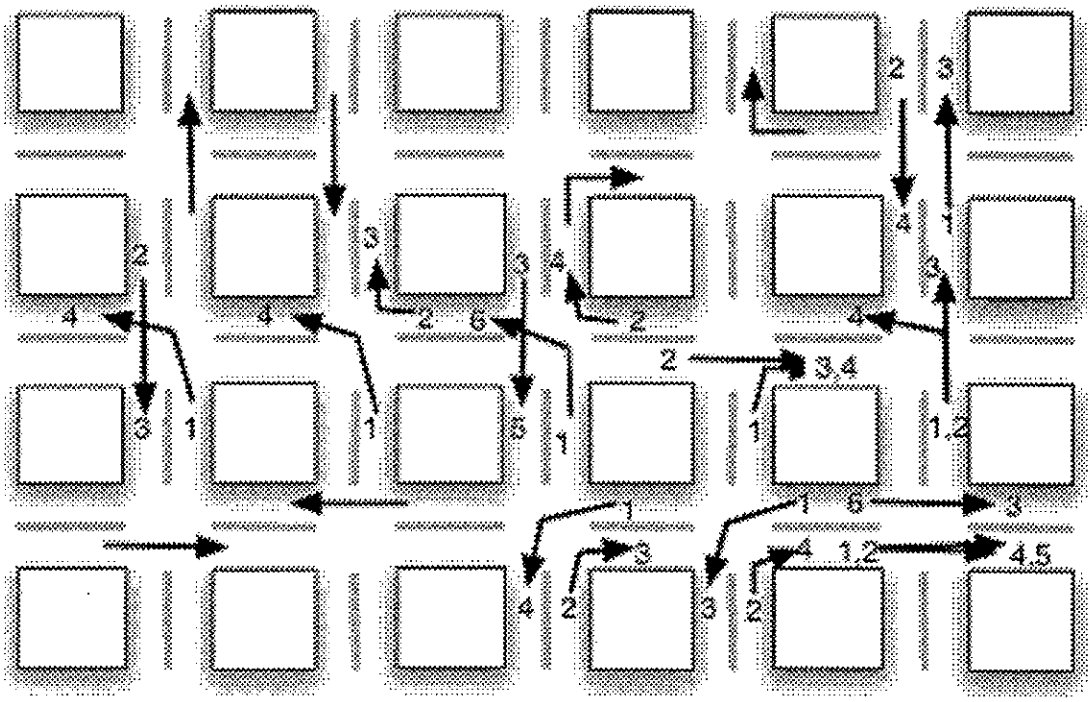

Figure 9: Graphical depiction of the vehicle movements in the sample data set. Arrows represent movements and numbers indicate the order of events for complex movement combinations.

A sample of the movement combinations used to test the software is provided in Figure 9. Each intersection diagrammed is separate from each other. The arrows indicate the path of a vehicle through the intersection. Numbers indicate the order in which events occurred when more than one vehicle used the intersection at one time. For example, the uppermost left intersection diagram indicates that a single vehicle passed straight through the intersection. Because there is only one vehicle in the intersection at a time, no numbering is provided. The uppermost right scenario indicates that one vehicle entered the intersection from the south, then another entered from the north, then the north-bound vehicle exited, and finally, the south-bound vehicle exited.

5. Algorithm

Our investigation focused on creating algorithms to interpret data from an intersection with traffic sensors arranged orthogonally, as depicted in Figure 10. Though we found

this sensor arrangement to work adequately for a small intersection of limited complexity, we discuss an alternative arrangement at the end of the paper, which may enhance the accuracy of the system in more complex situations.

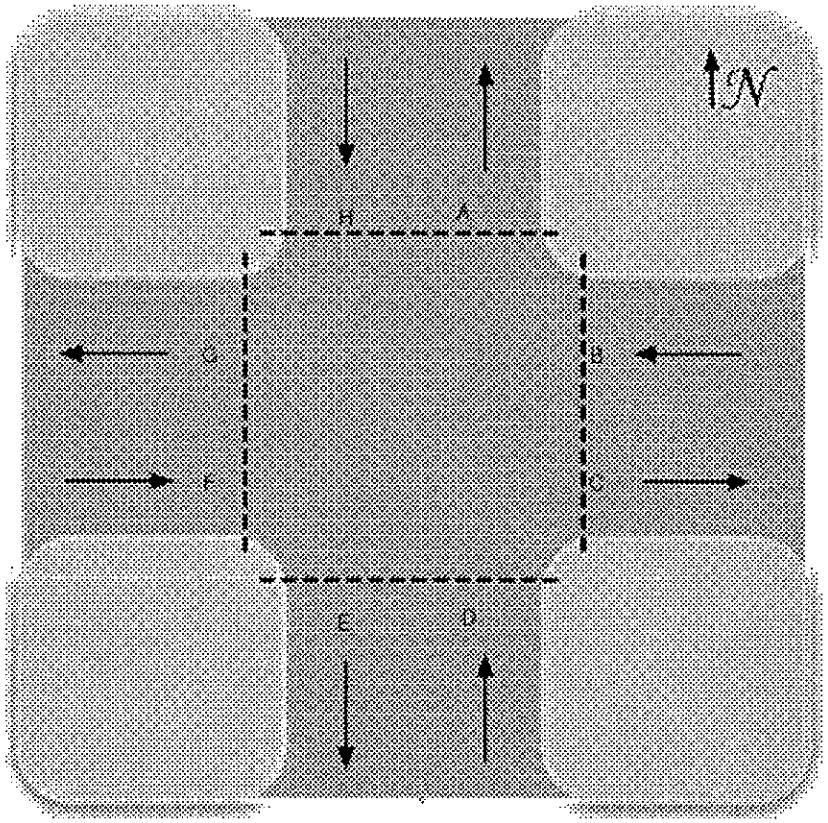

Figure 10: Orthogonal sensor arrangement assumed in testing.

Before discussing the algorithms, some terminology must first be defined. The sensors are long rubber strips, which comprise many small *segments*. Each of these segments will have a unique ID number and may be actuated independently. Each of the segments is mapped to a unique *region* in software, and each region typically contains all the segments in each lane. In Figure 10, there are eight regions, or sensors, as they will also be called. Because each lane is unidirectional, a sensor will be actuated either by vehicles entering or exiting the intersection, but not both. Sensors that are actuated by vehicles entering the intersection are called *sources*, and sensors actuated by leaving vehicles are called *sinks*. When a vehicle drives over a sensor, a number of segments are activated and given a timestamp. This occurs for each axle on the vehicle. Tire strikes

are combined to form axle strikes, and axle strikes are combined to form vehicle *events*. An event descriptor consists of the sensor the vehicle drove over and the time at which it drove over it. A *movement* is the sequence of two events, a source event followed by a sink event. For example, B-A defines a right turn in Figure 10.

Because each and every source event must have a sink event, the intersection may be thought of as a *queue* for data processing purposes. Every time a source event occurs, the event is enqueued. Every time a sink event occurs, a source event is dequeued and a pair is made forming a movement. Assuming the vehicles behave in a first-in-first-out manner, or there is at most one vehicle in the intersection at a time, or all the vehicles concurrently using the intersection are either sourced from, or sunk to a common sensor, the queue model produces results which are 100% accurate. Unfortunately, this is rarely the case. One vehicle may enter the intersection from sensor D, wait for a vehicle to pass from H to E, and then complete its left turn, passing over sensor G. In this case, the first vehicle in was the last to leave, and a queue model will indicate that a U-turn, followed by a right turn occurred, rather than a left turn and a straight.

Therefore, the crux of the problem is the proper matching of sink events to their source events. The data structure used to accomplish this matching problem is no longer a queue, but a set. Referring to Figure 10, assume the following series of events occurs: D at 0 seconds, H at 2 seconds, E at 3 seconds, and G at 4 seconds. Let us use the following notation for this sequence: [(0, D), (2, H), (3, E), (4, G)]. Each event is placed in chronological order. The following algorithm is used to process the event stream:

```
intersection = []
for each event in event_list:
     if event.type == source:
          intersection.add(event)
     else if event.type == sink:
          best_source = find_best(intersection, next_sinks)
          intersection.remove(best_source)
          log_movement(best_source, event)
```

The code loops through the list of events in chronological order and when a source event

is encountered (indicating a vehicle entering the intersection) it is placed in the intersection set. Each time a sink event is encountered, it must match to one of the source events in the intersection set. Rather than taking a greedy approach and trying to immediately determine the best source for the current sink, the algorithm looks ahead at future sink events as well, in order to create a match that cooperatively chooses the best match. If there are three source events in the intersection set when a sink event is encountered, for example, the algorithm considers the sink event, the three source events in the set, *and* the next two sink events after the present one, in order to determine the optimal match for the present sink event.

Let us assume there are two source events in the set, A and B, notated [A, B], when a sink event, X, occurs. If Y is the next sink event following X, we are left with two possible fates for A and B: (A⇒X, B⇒Y) or (A⇒Y, B⇒X). By considering both of these possibilities and assigning each of them a weight based on its probability, the best combination could theoretically be chosen.

A recursive function computes every possible unique combination of source and sink events. Each of these combinations is then passed on to a function that assigns each combination a score. The scoring function is the most important part of the algorithm.

A number of "fingerprinting" methods were considered in our studies, but were not used for various reasons. The speed of a vehicle cannot be used to match a source event to a sink event because the method for calculating speed relies both on the wheelbase of the vehicle and the assumption of constant velocity through the intersection. However, the wheelbase may vary greatly from vehicle to vehicle, and it very rare for a vehicle to maintain a constant speed through an intersection—some amount of acceleration or deceleration is expected. The next idea was based on a unique vehicle width, measured from the outside of each tire. However, it was agreed in a cooperative study with the Florida State researchers that the sensor strips cannot be built with a high-enough degree of resolution to accurately measure vehicle width.

With further analysis, we decided to use a scoring method that assigns the best score to the set of movements in which the vehicles travel at the most uniform average speed. The rationale behind the scoring method is that long turns will require more time to complete, and short turns will take less time. A left turn requires the vehicle to cover more ground than does a right turn, consequently, a left turn will normally take more time to complete. By dividing the turn distance by the amount of time needed to make the proposed turn, an average speed is computed for each movement in the proposed outcome set. Next, the standard deviation is computed for the set of speeds and this becomes the score. A lower standard of deviation indicates more uniform speeds, while a higher standard of deviation indicates less uniform speeds.

To illustrate this, consider a hypothetical intersection that has a left-turn path of 15 feet, a straight distance of 10 feet, and a right turn path of 5 feet. If three movements occurred, requiring 1, 2, and 3 seconds, we might propose these outcomes: the left turn took 3 seconds, the straight took 2, and the right turn took 1 second. In this case, the vehicles would have traveled, on average, 15/3, 10/2, and 5/1 feet per second, or 5 fps in each case. This set has a standard of deviation of zero. If another proposed outcome was that the right turn took 15 seconds, the straight 10, and the left turn 5 seconds, the outcome would mean the vehicles traveled 15/15, 10/2, and 5/3 feet per second, a less-probable set of speeds. Though there may be instances where this metric may fail, it seems to adequately select the best combinations in our tests.

Finally, each set is examined to determine if there are mutually exclusive movements, such as a N-S straight and an E-W straight, which would lead to a collision. Presently, only straight movements in opposing directions are considered, as left and right turns may be safely made concurrently, interlaced with other movements.

6. Test Results

The data analysis software was able to correctly identify each turn movement, including the complex multi-movement combinations depicted in Figure 9. We originally failed to

exclude the possibility of U-turns in the data decoder, which led to an intractable problem. For example, in the very common situation of a vehicle entering an intersection to turn left, and waiting for another vehicle to pass by, the software would interpret this as a U-turn and a right turn, rather than a left turn and a straight. By eliminating the possibility that a vehicle could complete a U-turn, a reasonable assumption for our small prototype intersection of US-62 and SH-102, the software was able to correctly identify each movement or combination of movements.

As the algorithm was written in the programming language *Python*, calling the program is done at the command line as seen in Figure 11. The next line shows the output of the program, a sum total of each type of predicted movement. The sample data is arranged as follows: {' source sensor ⇒ sink sensor': count, ...}.

```
Heron:~/research/traffic/code Luke$ python counter.py
{'7->4': 4, '5->2': 4, '3->2': 4, '3->0': 3, '3->6': 4, '1->0': 3, '1->4': 2, '1->6': 2}
```

Figure 11: Calling the data decoder program from the command line and its output, the count of each turning movement estimated.

The program was tested on the sample data depicted in Figure 9, and the test results are provided in Table 1. The first column represents the coded movement from the analysis software, the next column is the definition. The third column indicates the number of each movement predicted by the software, and the last column indicates the actual number of each movement type. As the data indicate, the algorithm was able to predict the movements with perfect accuracy.

Movement Code	Movement Type	Predicted Count	Actual Count
7 ⇒ 4	Straight, N ⇒ S	4	4
5 ⇒ 2	Straight, W ⇒ E	4	4
3 ⇒ 2	Right, S ⇒ E	4	4
3 ⇒ 0	Straight, S ⇒ N	3	3
3 ⇒ 6	Left, S ⇒ W	4	4
1 ⇒ 0	Right, E ⇒ N	3	3
1 ⇒ 4	Left, E ⇒ S	2	2

$1 \Rightarrow 6$	Straight, $E \Rightarrow W$	2	2
	Totals	26	26

Table 1: Results from test on sample data—counts of estimated and actual movements.

7. Sensor Design Considerations

Working independently of and later in conjunction with researchers at Florida State, we sought to determine the optimal segment size by creating a axle simulator. The size of each segment is an important consideration. The desire is to make them small enough so that two vehicles closely driving side by side will be sensed as two separate vehicles, and not one. If we could make the segment sizes smaller still, we would obtain more data. As the length of each segment approaches zero, the width of each tire could be precisely measured. Furthermore, the gap between the rear tires on four-tire axles could be detected, and the software could distinguish between a small vehicle and a more damaging, heavy truck. More importantly, knowing the identity of each axle would aid the assignment of axle strike events to particular vehicles. However, the more segments there are, the more difficult and expensive it becomes to build and interface the sensors, and process they data they generate.

In order to study the optimal segment size, we created a program to simulate the striking of tires over strips divided into segments of different sizes. In general, sedan tires will be narrower than truck tires, so instances when fewer segments were activated can be interpreted as events caused by sedan tires. However, we care most about the border case, where the largest practical sedan tire and the smallest possible dual tire are considered. In this case, overlap may occur, and the result will be ambiguous. The largest sedan tire considered was 9 inches, and the smallest dual truck tires were 5.5 inches with a 4 inch gap between. These dimensions were provided by Timur Mauga, from Florida State. According to John Reed, the smallest gap possible between segments ins one inch. Taking these dimensions into account, the simulator produced the data presented in Table 2. The smallest segment size that does not lead to overlap is one inch. Here, the single

tire never activates more than five segments, while the dual tires will always activate at least six segments. However, the smallest segment size considered by the Florida State researchers was four inches for practical considerations. Thus, we jointly concurred that it would not be able to distinguish between single and dual tires at the border condition with four-inch segments.

	Single Tire		Dual Tire	
Segment Size, inches	**Minimum Activated**	**Maximum Activated**	**Minimum Activated**	**Maximum Activated**
1	5	5	6	7
2	3	4	4	6
3	3	3	4	5
4	2	3	3	4
5	2	3	3	4
6	2	3	3	3
7	2	2	2	3
8	1	2	2	3
9	1	2	2	3
10	1	2	2	3
11	1	2	2	3
12	1	2	2	3
13	1	2	2	2
14	1	2	1	2

Table 2: Results of segment size study—the minimum and maximum number of segments activated by a single and dual tire are given for each segment size considered.

Unless a method for obtaining greater resolution is discovered, which allows the differentiation between single and dual tires, we believe it will be more difficult to assign multiple axle strikes to a single vehicle.

8. Sensor Layout Considerations

Although we are pleased with the results of the analysis algorithms on simple intersections in which no U-turns are permitted, we believe an alternative sensor layout has the potential of adding additional valuable data and producing better results.

Due to the increased difficulty of event matching with data produced by orthogonally-placed sensors in certain circumstances, a better method for placing the sensors was sought. Because there is no difficulty matching a sink event to a set with only one source event—in other words, only one vehicle is in the intersection at a time—methods were sought for arranging the sensors to allow only one vehicle at a time within the confines of the sensors. This was impractical for a number of reasons: each vehicle has a different size, and the shapes would require complex bending of the sensor strips and anchoring into the pavement. We then realized that by placing the sensors obliquely, that is, criss-crossing the intersection from corner to corner as in Figure 12, we would gain the ability to determine the direction a vehicle traveled as it set off the sensor. A vehicle making movement M1, driving straight from east to west as seen in Figure B, will strike the segments between points B and C closer to B first, then closer to C second. In contrast, however, a vehicle driving over the same segments, making movement M2 will first strike segments closer to point C and then segments closer to point B. Finally, a vehicle making a right turn, as seen in M3, will most likely strike the segments under the left and right wheels of each axle at approximately the same time. Assuming speeds are fairly consistent, this delay between the left and right wheels registering will be roughly the same or be spread out, but in varying orders.

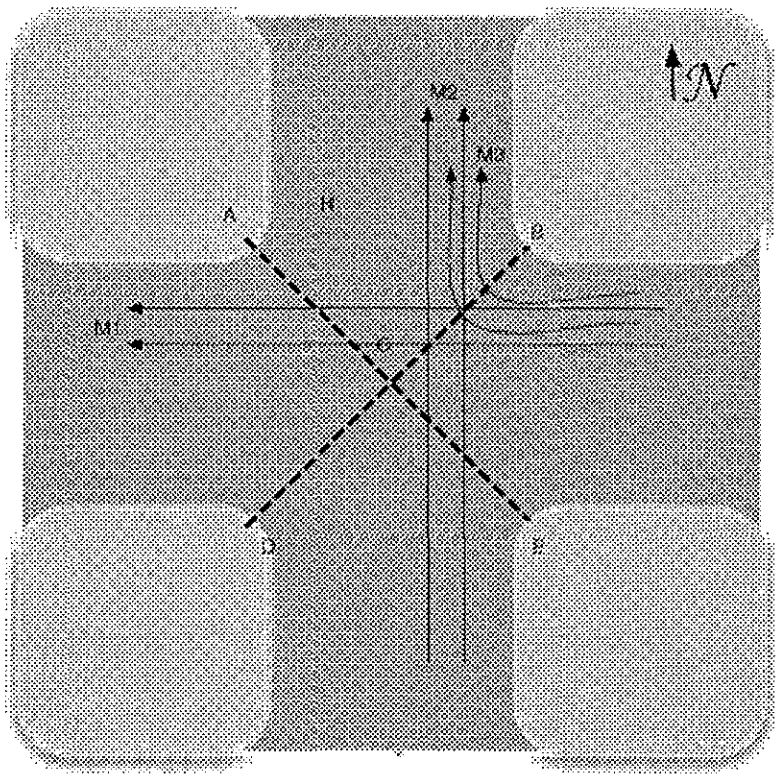

Figure 12: Alternative sensor arrangement which may enhance accuracy.

If sensor strips are to be used to record the turning movements of vehicles, an oblique arrangement will likely provide more useful data by allowing software to detect not just the presence of a vehicle, but also its direction. Though the data provided is more useful, the architecture of the software must be entirely different. Due to the higher volume and complexity of data, and the possibility for spurious wheel strikes in adjacent regions, fuzzy logic and advanced pattern recognition will likely prove necessary. It is possible that the data could be translated into a format that would enable a regular expression matching engine to be harnessed.

9. Conclusion

From our study of a simple intersection, we have developed a novel algorithm to determine the types of turning movements executed, and believe that it will be feasible to apply these same principles to intersections and situations of greater complexity. Though

our tests indicate the algorithm works quite well on artificial data, which we believe to be reasonably accurate, fine tuning and improvements cannot be made until the software is tested with real-life data, which may include significant amounts of spurious data or events not yet anticipated.

The architecture of the software enables the matching engine to scale up to intersections of any complexity by creating an intersection descriptor file that maps the sensor segments, intersection geometry, and permitted movements. It is difficult to gauge how well the software will perform on larger intersections. While larger intersections will have stricter rules, green arrows instead of green lights, and turning lanes, all of which aid in matching, the possibility of a greater number of vehicles in the intersection at one time, and increased number of movements adds to the complexity of matching. Therefore, further testing and fine tuning with real data must be completed.

Appendix: Source Code

counter.py

```
#!/usr/bin/env python
# encoding: utf-8

import env        # data describing the intersection

debug_mode = 0

def loadData(fn):
      """Given a filename, return a list of the data
      as a list of two-tuples in the format (segment, timestamp)"""
      lst = []
      f = open(fn)
      lines = f.readlines()
      f.close()
      for ln in lines:
            pieces = ln.split(",")
            lst.append((int(pieces[0]), float(pieces[1][:-1])))
      return lst


def main():
      glo_seg_evts = loadData("single-strike-data.txt")           # get
the intersection-wide segment events
      reg_seg_evts = get_reg_seg_evts(glo_seg_evts) # transform the seg
events into regional seg events
      reg_axle_evts = get_reg_axle_evts(reg_seg_evts) # transform
regional seg evts to regional axle evts
      reg_veh_evts = get_reg_veh_evts(reg_axle_evts)  # transform
regional axle evts to regional vehicle evts
      glo_veh_evts = get_glo_veh_evts(reg_veh_evts)   # get the global
vehicle events...
      mvtlog = complex_processor(glo_veh_evts)
      print mvtlog

def complex_processor(glo_veh_evts):
      intersection = []
      movementlog = {}
      for i in range(len(glo_veh_evts)):
            if glo_veh_evts[i][2] == 1:                           #
source event
                  intersection.append(glo_veh_evts[i])
            else:                                          # sink event
                  if len(intersection) == 1:    # easy/obvious match...
                        tmp = intersection.pop()
                        mvtstr = "%d->%d"%(tmp[1],glo_veh_evts[i][1])
                        if debug_mode: print mvtstr
                        if movementlog.has_key(mvtstr):
                              movementlog[mvtstr] += 1
                        else:
                              movementlog[mvtstr] = 1
                  elif len(intersection) == 0:  # error! Sink event
without prior source event
                        print "Serious mismatch error"
                  else:
                        veh_ct = len(intersection)          # number of
```

```
vehicles in the intersection currently
                        sink_evts = []                                  # a
list of sink events to match to vehicles in intersec.
                        for se in glo_veh_evts[i:]:          # pick an
equal number of sink events in the future
                              if len(sink_evts) == veh_ct:# we have
enough, so stop.
                                    break
                              else:                                     #
otherwise, keep adding...
                                    if se[2] == 0:
                                          sink_evts.append(se)
                        src = choose_best(intersection, sink_evts)  #
idx will be index of best src for curr. sink
                        mvtstr = "%d->%d"%(src[1], glo_veh_evts[i][1])
                        if debug_mode:
                              print mvtstr
                        intersection.remove(src)
                        if movementlog.has_key(mvtstr):
                              movementlog[mvtstr] += 1
                        else:
                              movementlog[mvtstr] = 1
                  if debug_mode:
                        raw_input("Press <ENTER> to continue...")
      return movementlog


def choose_best(int_vehs, sink_evts):
      """tries to find the best vehicle currently in the intersection
to match to the sink event
      which is the first item in sink_evts.  The following items in
sink events are items down the line
      just for comparison.  By looking ahead and using some logic, we
can make better choices than
      a simple greedy algorithm"""

      scores = []
      for combination in make_combinations(int_vehs, sink_evts):
            scores.append((score_combination(combination),
combination))
      scores.sort()
      bestcombo = scores[0][1]

      for evt in bestcombo:                  # look for the source event
of the sink we're looking for in this best combo
            if evt[1][1] == sink_evts[0][1]:
                  return evt[0]
      return None                                    # failure for some
reason



def score_combination(pairs):
      """Given a list of (source,sink) pairs, come up with a likelihood
score
      for this being the correct combination of pairs
      Returns None if there is an invalid movement as one of the
```

```
combinations...
      returns the standard deviations of the speeds of the movements.
The idea is that long
      turns should take longer; short turns should take shorter.  If
you divide long distances by
      long times and short distances by short times, you get similar
speeds.  If you mismatch
      long distances with short times and short distances with long
times, the speeds will vary
      wildly and the standard deviation will be high.  Thus, the pair
with the smallest standard
      deviations will have the highest likelihood of being
correct..."""
      flags = {}
      speeds = []
      for mvt in pairs:
            mdata = get_movement_data(mvt)
            if mdata == None:
                  return 1e12 # a really big number... meaning not
valid.
            spd = mdata['dist'] / mdata['time']
            speeds.append(spd)
            if mdata['flag'] != None:
                  flags[mdata['flag']] = 1
      mean = 0
      for speed in speeds:
            mean += speed * (1.0 / len(speeds))
      sig2 = 0
      for speed in speeds:
            sig2 += (mean-speed) ** 2
      if len(flags.keys()) > 1:     # if collision potential, return
"impossible" score
            return 1e12
      else:
            return ((1.0 / len(speeds)) * sig2) ** 0.5



def make_combinations(t1, t2):
      """A recursive function that takes a list of source events (t1)
and a list of sink events (t2)
      and comes up with all the possible sets of source-->sink event
pairings.  This list of sets
      of pairings can then be used to create scores for each."""
      if len(t1) <> len(t2):
            print "error"
            return []
      tmp = []
      if len(t1) == 1:
            return  [  [ ( t1[0], t2[0] ) ]   ]
      elif len(t2) == 2:
            return [   [ ( t1[0], t2[0] ), ( t1[1], t2[1] ) ],
[(t1[0],t2[1]),(t1[1],t2[0])]   ]
      else:
            for j in range(len(t2)):
                  for tail in make_combinations(t1[1:],
```

```
t2[:j]+t2[j+1:]):
                        tmp.append( [(t1[0],t2[j]),] + tail )
            return tmp


def stack_processor(glo_veh_evts):
      intersection = []
      for evt in glo_veh_evts:                  # for every event (a
car entering or leaving intersection)
            if evt[2] == 1:                           # if it is a
vehicle entering...
                  intersection.append(evt)            # push it onto
the intersection stack.
            else:                                     # veh leaving
intersec
                  tmp = intersection.pop()
                  print "%d --> %d"%(tmp[1],evt[1])
                  print "movement type: " +
str(env.get_movement_type(tmp[1],evt[1]))
                  if debug_mode:
                        raw_input("Press <ENTER> to continue...")

def queue_processor(glo_veh_evts):
      intersection = []
      for evt in glo_veh_evts:
            if evt[2] == 1:
                  intersection.insert(0,evt)
            else:
                  tmp = intersection.pop()
                  print "%d --> %d"%(tmp[1],evt[1])


# sorting function for get_glo_veh_evts
def sort_veh_evt(x,y):
      if(x[0] < y[0]): return -1
      elif(x[0] == y[0]): return 0
      else: return 1

def get_glo_veh_evts(reg_veh_evts):
      """returns a list of tuples in the format (timestamp, region,
source/sink)"""
      lst = []
      for i in range(len(reg_veh_evts)):
            for ts in reg_veh_evts[i]:
                  lst.append((ts,i,env.regions[i][2]))
      lst.sort(sort_veh_evt)
      return lst


def get_reg_axle_evts(reg_seg_evts):
      """Takes the timestamps of the segments in a region and produces
a list of
      axle strike timestamps by region.
      Currently, it just uses the segments for the axles (assumes one
seg per lane)"""
      return reg_seg_evts
```

```
def get_reg_seg_evts(seg_evts):
      """Takes all the segment events in the intersection and sorts
them into regions
      according to the region map in the env module"""
      reg_seg_evts = []
      for i in range(len(env.regions)):
            reg_seg_evts.append([])
      for evt in seg_evts:
            reg_seg_evts[env.segmap[evt[0]]].append(evt[1])
      for i in range(0, len(reg_seg_evts)):
            reg_seg_evts[i].sort()
      return reg_seg_evts


def get_reg_veh_evts(reg_axle_evts):
      """simply takes every other (not very sophisticated), starting
the timing from the time the back
      axle crosses the sensor.  Doesn't attempt to detect multi-axle
vehicles... only works on doubles."""
      #reg_veh_evts = []
      #for reg in reg_axle_evts:
      #     tmp = []
      #     for j in range(1,len(reg),2):
      #           tmp.append(reg[j])
      #     reg_veh_evts.append(tmp)
      #return reg_veh_evts
      return reg_axle_evts          # temporarily for the signle-
strike-data.txt file, which is just veh. evts.

def get_movement_data(mvt):
      """Given two regions (A,B), returns data for the movement A-->B
      such as time between the two, probability, type of movement,
etc..."""
      data = {}
      data['time'] = mvt[1][0] - mvt[0][0]      # seconds to make this
mvt...
      tmp = env.movements[mvt[0][1]][mvt[1][1]]
      if tmp != None:
            data['dist'] = tmp[1]
            data['type'] = tmp[0]
            data['flag'] = tmp[2]
      else:
            data = None

      return data

if __name__ == '__main__':
    main()
```

env.py

```
#      A simple intersection for which this is currently configured.
#
#                 |       |       |
#                 |  NW   |  NE   |
#                 |       |       |
#                 |    7  |    0  |
#_________________|                 |_________________
#      WN      6                        1     EN
#_________________                     _________________
#    WS      5                           2     ES
#_________________|                 |_________________
#                 |    4  |    3  |
#                 |       |       |
#                 |   SW  |   SE  |
#                 |       |       |
#                 |       |       |

# MOVEMENT TYPES:
St = 0       # straight
LT = 1       # left turn
RT = 2       # right turn
UT = 3       # U turn

laneWidth =        12.0
segSize =   1.5
segGap =    0.08

maxgap = 0.010          # max seconds two segments will be apart from
each other...

# format:  ('description', source/sink) where source=1, sink=0
# A region is a group of contiguous segments, usually all the segments
# in a single lane.
regions = ( (0,'NE',0),
                  (1,'EN',1),
                  (2,'ES',0),
                  (3,'SE',1),
                  (4,'SW',0),
                  (5,'WS',1),
                  (6,'WN',0),
                  (7,'NW',1) )

# Each segment in the INTERSECTION (not just per strip) has a unique ID
number
# this maps each segment ID to a unique region.
#segmap = (0,0,0,0,0,0,0,0,          # 0-7 map to seg 0
#                   1,1,1,1,1,1,1,1,      # 8-15 map to seg 1, etc...
#                   2,2,2,2,2,2,2,2,
#                   3,3,3,3,3,3,3,3,
#                   4,4,4,4,4,4,4,4,
#                   5,5,5,5,5,5,5,5,
#                   6,6,6,6,6,6,6,6,
#                   7,7,7,7,7,7,7,7)
segmap = (0,1,2,3,4,5,6,7)
```

```
# None means the path is not possible, distances in feet
# movement[source][sink] returns (turn type, turn distance) if it is a
valid
# movement, or None if it is illegal (such as a sink to a source...)
# the third item in the tuple is an optional flag for the turn... NS
means a north-south axis
# straight, EW means an east-west axis straight... two of these can't
occur simultaneously
# or there could be a collision...
movements =((None,None,None,None,None,None,None,None),

      ((RT,9.42,None),None,None,None,(LT,28.3,None),None,(St,24.0,'EW')
,None),     # 1
                  (None,None,None,None,None,None,None,None),

      ((St,24.0,'NS'),None,(RT,9.42,None),None,None,None,(LT,28.3,None)
,None),     # 3
                  (None,None,None,None,None,None,None,None),

      ((LT,28.3,None),None,(St,24.0,'EW'),None,(RT,9.42,None),None,None
,None),     # 5
                  (None,None,None,None,None,None,None,None),

      (None,(LT,28.3,None),None,None,(St,24.0,'EW'),None,(RT,9.42,None)
,None))     # 7
```

single-strike-data.txt

```
3, 0
0, 1.5922896
7, 9.9442992
4, 11.4564736
3, 16.974408
2, 18.3664096
1, 26.7785056
0, 28.29068
3, 36.05184
7, 37.0332512
0, 38.3751808
4, 39.4467216
3, 89.0881024
7, 90.830608
4, 93.0538048
6, 94.9665552
3, 141.5835872
1, 143.856856
0, 144.9384112
6, 145.7195344
3, 200.1177552
7, 202.3609808
1, 203.8030544
0, 204.5240912
4, 205.6857616
```

```
6, 206.79736
3, 324.0960272
5, 325.8212832
2, 328.021672
2, 329.9544512
3, 343.6641648
3, 344.8859216
0, 347.5898096
6, 349.5426176
5, 362.8517552
2, 364.7645056
1, 370.03208
6, 371.7745856
1, 384.3927296
3, 385.604472
2, 386.956416
4, 388.1481296
1, 417.1398176
3, 418.6920496
4, 423.659192
2, 424.8008336
5, 1415.485368
1, 1416.7972544
5, 1418.3795296
2, 1419.8616608
2, 1420.8731152
6, 1422.4253472
```

segsim.py

```
from math import floor
from random import gauss
from random import random
from math import sqrt

# set these before running the simulation
min_seg_size = 6  # smallest segment size to be simulated, in inches
max_seg_size = 20 # biggest segment size to be simulated, in inches
n_times           = 10000     # number of cars and trucks to simulate
per segment size
Wg = 0.25               # gap width in inches
max_offset  = 36.0  # max inches to offset... (probably fine as is)

# these are changed by the simulator automatically
Ws = 12.0               # segment width in inches (this will change in
the sim.)
axle_offset = 12.0      # inches axle is to the right of the left
barrier of lane


class Axle:
      def __init__(self):
            pass
      def getAxleDimensions(self):
            """return a list that describes this axle"""
```

```
class SedanAxle(Axle):
      def __init__(self):
            self.Wt_m = 6.0   # width of tire, mean
            self.Wt_s = 0.3   # width of tire, stdev
            self.Wa_m = 60.0  # width of axle (dist between tires),
mean
            self.Wa_s = 1.0   # width of axle, stdev
      def getAxleDimensions(self):
            Wt = gauss(self.Wt_m, self.Wt_s)
            Wa = gauss(self.Wa_m, self.Wa_s)
            return [[0, Wt], [Wt+Wa, Wa+2*Wt]]

class TruckAxle(Axle):
      def __init__(self):
            self.Wt_m = 8.0   # width of tire, mean
            self.Wt_s = 0.3   # width of tire, stdev
            self.Wa_m = 66.0  # width of axle, mean
            self.Wa_s = 1.0   # width of axle, stdev
            self.Wg_m = 3.0   # width of tire gap
            self.Wg_s = .1    # width of tire gap, stdev
      def getAxleDimensions(self):
            Wt = gauss(self.Wt_m, self.Wt_s)
            Wa = gauss(self.Wa_m, self.Wa_s)
            Wg = gauss(self.Wg_m, self.Wg_s)
            return [[0, Wt], [Wt+Wg, 2*Wt+Wg], [2*Wt+Wg+Wa,
3*Wt+Wg+Wa], [3*Wt+2*Wg+Wa, 4*Wt+2*Wg+Wa]]

def unique(lst):
      new = []
      for i in lst:
            if not new.__contains__(i):
                  new.append(i)
      return new

#
#  the segments go as:  |  seg 1       |gap1|  seg 2        |gap2| ...
#  the units go as:     |  unit 1          |  unit 2           | ...
#  the absolute value returned by get ActivatedSegment(x) indicates
# which unit it falls in.  The sign indicates whether it is in a gap
# region or a segment.  negative means it is in a gap region (inactive)

def getActivatedSegments(axle):
      activated_segs = []
      for tire in axle:
            xt0 = tire[0] + axle_offset
            xt1 = tire[1] + axle_offset

            s0  = getActivatedSegment(xt0)
            s1  = getActivatedSegment(xt1)

            if s0 < 0:
                  a = (s0*-1) + 1
            else:
                  a = s0
            b = int(sqrt(s1*s1))
            for i in range(a, b+1):
                  activated_segs.append(i)
```

```
        return unique(activated_segs)


def getActivatedSegment(x):
        Wu = Wg + Ws
        unit = int(x // Wu)      # unit number
        rem = x - Wu * unit
        if rem <= Ws:
                return unit+1
        else:
                return (unit+1) * -1


def getNumActivatedSegs(axle):
        return len(getActivatedSegments(axle))


def runSimulation():
        global Ws
        for ss in range(min_seg_size, max_seg_size+1):
                Ws = ss
                simulateSegSize()


def simulateSegSize():
        car_counts   = collectData(SedanAxle())
        truck_counts = collectData(TruckAxle())
        w,x = getGaussParams(car_counts)
        y,z = getGaussParams(truck_counts)
        print "%f, %f, %f, %f, %f" % (Ws, w, x, y, z)

def mean(l):
        sum = 0
        for item in l:
                sum += item
        return sum * 1.0 / len(l)

def getGaussParams(l):
        avg = mean(l)
        sum = 0
        for item in l:
                sum += (item - avg) ** 2
        return (avg, sqrt(sum/len(l)))


def collectData(axle):
        global axle_offset, max_offset
        counts = []
        for i in range(n_times):
                axle_offset = random() * max_offset
                a = axle.getAxleDimensions()
                counts.append(len(getActivatedSegments(a)))
        return counts


if __name__ == "__main__":
```

```
runSimulation()
```