Designing Decentralized Algorithms to Guide Autonomous Vehicles through Traffic Intersections

Michael Leon

Table of Contents

Acknowledgements

This project could not have been completed without the selfless support from many. My teacher Patel Parin and my mentor Cody Reeves were a huge help in developing the purpose and relevance of this project. My mentor also supplied me with more technical insight into the implementation of the project. My peers' role in this project could never be thanked enough. Maha Ali, Nicholas Ieremciuc, Ali Chaudry, and Heba Sattar were all very instrumental in this project. Their constant feedback and advice throughout the year was invaluable and helped me improve my project. My parents, Patricia and Greg Leon, encouraged me through all the bumps along the way. They provided unconditional support and encouragement. My siblings Christina and Gregory Leon also were always ready to give input and possible revisions to any portion of the project.

Need

Driving is a highly popular mode of transportation across the globe. Traffic intersections account for a disproportionately large share of car accidents. These intersections can also be very inefficient. The policies that govern these intersections, such as the stoplight and stop sign, were made for human drivers. However, soon humans will become the minority in the driving world, becoming replaced by self driving cars. The combination of inadequate control algorithms and untapped potential of self-driving cars necessitates the creation of a new intersection policy, one geared towards the future of the automotive industry.

Background

Computers have steadily pierced through every modern industry. In many instances, computers actually replace the human element of the job altogether. Soon, the act of a human driving will become a pastime. Google, Nissan, Mercedes-Benz, and Audi are already testing various self-driving car prototypes. As these cars develop and enter the consumer market, society can rethink the way its intersections work. Whether it's stoplights, stop signs, or roundabouts, these traffic intersection mechanisms all are working towards the same goal: to regulate the traffic flow of *humans*. However, in a world with roadways filled with autonomous vehicles, that core assumption of a human being behind the wheel can change. Therefore, the traffic policy in place must adapt as well, to take advantage of the automated, calculated nature of computers. This is the central focus of this study: to seek out the optimal algorithms for guiding these autonomous cars through intersections. These algorithms would become the new policy for intersections, and replace all the outdated methods like roundabouts and stop lights.

Intersections make up only a small portion of the roads, yet they unproportionally account for a large amount of accidents. "Nationally, 40 percent of all crashes involve intersections, the second largest category of accidents, led only by rear end collisions. Fifty percent of serious collisions happen in intersections and some 20 percent of fatal collisions occur there" (Statistics on Intersection Accidents, 2015). Once the self-driving car is made popular in society, practically all incidents involving judgement error will be eliminated. However, creating a safer intersection policy systematically removes and reduces the chance for accidents to occur.

Traffic stop lights, or signals, are the defacto standard for governing moderately busy to very busy traffics. There's roughly one signal for every 1,000 people. This would mean over 300,000 signals in the country (*FHWA*, 2016). Roughly 52.5% of all intersection-related accidents took place at an intersection being regulated by a traffic signal (Choi, 2013). "By alternately assigning right of way to various traffic movements, signals provide for the orderly movement of conflicting flows. They may interrupt extremely heavy flows to permit the crossing of minor movements that could not otherwise move safely through the intersection. When properly timed, traffic signals increase the traffic handling capacity of an intersection, and when installed under conditions that justify its use, it is a valuable device for improving the safety and efficiency of both pedestrian and vehicular traffic. In particular, signals may reduce certain types of accidents, most notably the angle (broadside) collision.

In addition to an increase in accident frequency, unjustified traffic signals can also cause excessive delays, disobedience of signals and diversion of traffic to inadequate alternate routes. Traffic signals are much more costly than is commonly realized, even though they represent a sound public investment when justified. A modern signal can cost taxpayers between $80,000 and $100,000 to install, depending on the complexity of the intersection and the characteristics of the traffic using it. [In addition], there is the perpetual cost of the electrical power consumed in operating a signalized intersection 24 hours a day. This cost now averages about $1,400 per year." (Pros and Cons, 2016)

Roundabouts, stop signs, and unregulated intersections are the other major kinds of intersection policies. Each of these mechanisms offers its own unique benefits and costs. Unregulated intersections are the easiest to understand. In this situation, right of way is dictated

by the standard rules of the road. There exists no external structure which assigns right of way such as in a stop sign or traffic signal. Unregulated intersections are only ideal in the narrow case in which the road is empty save for one driver agent. Stop signs are suitable for low-volume traffic. In an analysis of motor-vehicle crashes, drivers were found to have a high tendency to ignore stop signs all together, which reduces the safety of the policy greatly. Out of 700,000 police-reported motor vehicle crashes at stop signs, 70% were due to violating the sign (Retting, 2003). This same study found that one of the more effective remedies of this problem is to simply change the intersection control policy. Roundabouts can have a relatively large impact on safety. In a case study of a Swedish city, the long term effects of large scale roundabouts were tested. The results "indicated an overall decrease in accident risk by 44%.". However, even though they do tend to improve the safety of drivers, the efficiency worsens. "The speed-reducing effect is large already at a 2 [meter] deflection " (Hydén, 2000). Smaller roundabouts have less of a negative impact on efficiency.

One can find surprising accuracy in modeling the motion of a car in transit through an intersection as a 2-dimensional character. Neglection of the third dimension is quite common when creating models for cars. Even in experiments where the motion of the vehicle is most important, a 2D model is still used (Macek, 2007). This is due to the fact that essentially all of the movement happens in only 2 dimensions. Ultimately, the physics of a car travelling through a region of space is not the most important aspect for this model. Rather, this program will be focusing on the coordination of its agents. In video games and animation, 2D characters are often autonomous. As such, they need to have the "ability to navigate around their world in a life-like and improvisational manner." (Reynolds, 1999) Reynolds proceeds to outline many different

kinds of steering behaviors. The ones most relevant to this study will be seek and flee. In order to discuss these two motions, the actual driver agent needs to be formally defined. Following in Reynold's suit, a driver agent D is $D = (m,\ p,\ v,\ F_{max},\ s_{max})$ where m, F_{max}, s_{max} are scalars representing mass, maximum force and maximum speed respectively and p and v are vectors representing position and velocity, respectively. Velocity will always satisfy $|v| \leq s_{max}$. There are two vectors calculated from any driver agent which determine the actual motion of D: F_{steer} and a. A function $truncate(x, x_{max})$ yields x unless it surpasses the value of x_{max}, in which case it yields x_{max}. Using this function, $F_{steer} = truncate(direction_{steer},\ F_{max})$ and $a = \frac{F_{steer}}{m}$. A modified version of the kinematics equations applies, with the position being calculated from v and v being calculated from a; $v_n = v_{n-1} + a$ and $p_n = p_{n-1} + v$. The steering direction, $direction_{steer}$, is the crux of all the steering behaviors, like seek and flee. This is the value that they manipulate. When a character is seeking out a target, it simply sets $direction_{steer}$ to point towards the target. Conversely, the flee behavior causes $direction_{steer}$ to point away from the target if a collision is imminent. Figure 1 showcases these behaviors

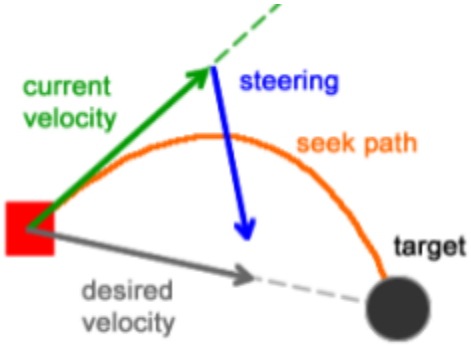


Figure 1

A valid concern with this model is that cars cannot move freely in either axis of motion. Only the y-axis (that is, when reversing or accelerating) can be freely traversed. A car cannot simply strafe sideways. While this is true, nearly all the motions produced with the outlined model do not create unrealistic movements for cars. That is, whenever horizontal movement does appear, it is always in conjunction with movement in the y-axis so as to produce a curved trajectory. This is the expected movement of real cars.

The program will be created in a decentralized architecture (b), as opposed to a centralized architecture (a.) These two terms, in the context of software engineering, refer to where the locus of power falls. Either it is in one central acting body or dispersed among the system's constituents. This also determines the mode of communication. Either the nodes of the system are all connected to one point, or they are connected to their local neighbors. Figure 2 displays a visual representation of centralized vs. decentralized architectures.

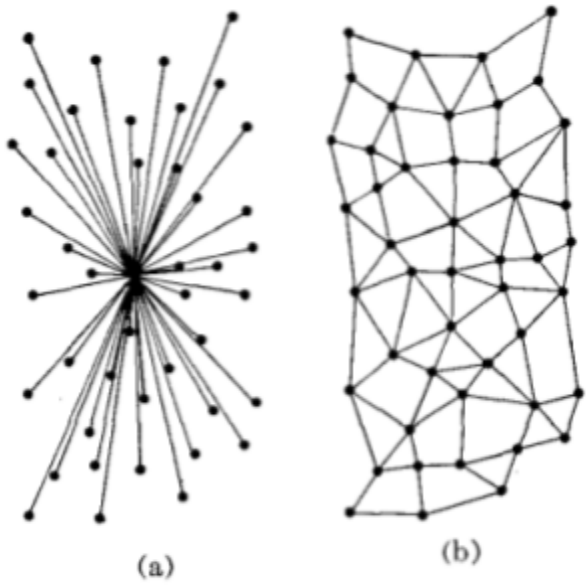


Figure 2

Locus of power here most closely means where the responsibility for making decisions lies. There is another piece of literature which details the workings of an alternate traffic policy, geared towards self driving cars. The program, titled AIM, employs a centralized system. (Kurt Dresner, 2015) All driver agents connect to a single intersection manager (IM) which works on a reservation system. Cars reserve space-time slots through the intersection which are either confirmed or denied by the IM. The problem with this approach, however, is that it inherits all the issues which come with any centralized system. The IM serves as a single point of failure. If this agent becomes unavailable, all the traffic comes to a halt and the system cannot function. The IM also creates a bottleneck effect. All communication has to flow through it, so the amount of computation is has to do rises above-linearly with respect to the amount of nodes in the network.

Decentralized systems differ in that the nodes communicate amongst their local neighbors rather than one central node, like an IM. Responsibility of decision making also is dispersed out to the nodes, rather than one central agent. When studying decentralized systems, the notion of emergent behavior becomes very important. Emergent behavior is simply an observed behavior at the global scale, which arises due to smaller interactions within the nodes, or members, of the system. As long as individual members of the population satisfy rules pertaining to local information, a desired behavior from the entire population can be achieved. Seemingly complex emergent behavior can come from simple, local rules.The canonical example is the program boids, which was also written by Reynolds. In this program, a boid, which can be thought of as a bird, must

1. steer to avoid crowding local flockmates

2. steer towards the average heading of local flockmates
3. steer to move toward the average position (center of mass) of local flockmates

Using these three simple rules, a flock of boids can create stunning formations and quickly adapt to new obstacles in their environment (Reynolds, 1987). In the same way, a set of simple rules guiding driver agents can still achieve the complex task of managing an entire intersection of these cars.

The performance of an intersection policy can be quantified through 2 variables: safety and efficiency. These are the most important characteristics to any intersection policy. The program will be built to measure these two variables constantly. Safety will be a rate of the occurrence of collisions. For example, if 100 cars pass through an intersection and there was one crash, the crash rate would be 1%. Efficiency is slightly more difficult to record. Upon creation and insertion into the simulation, each car is assigned a value t which represents the time spent inside the intersection. On each frame rendering and updating of the program, t is incremented by the amount of time which has passed since the last update, in seconds. When the vehicle finally settles into its exit lane, t ceases to be incremented and is added onto a sequence T. At any given time, the efficiency of the system can be taken as the average of T. Other statistical analysis tools can be used on this sequence of data T, such as analyzing the distribution of wait times. This dataset may be further split up into more groups, such as wait times of all left turning cars.

Bézier curves are a mathematical tool often used in computer graphics. In this study, a quadratic bezier curve will be used to create a route for a driver agent. It is defined by three

coordinates: p_1, p_2, p_3 . The points p_1 and p_3 are the endpoints of the curve, and p_2 is considered the control point. Figure 3 illustrates a sample quadratic bézier curve.

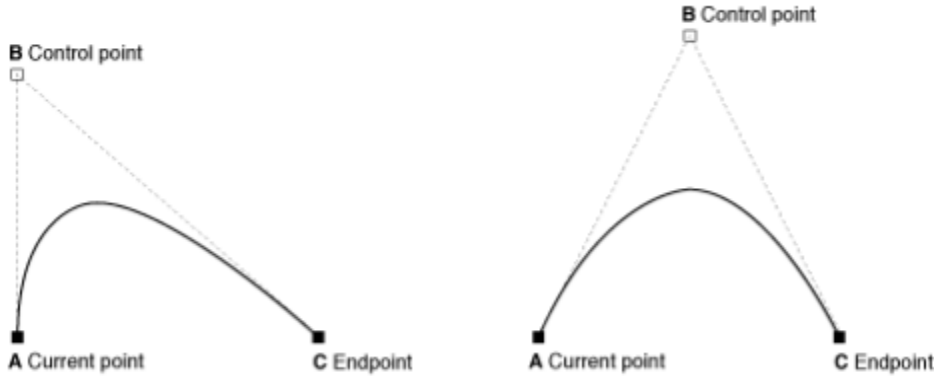


Figure 3

A point along the curve can be obtained using the function

$f(x) = p_1 * (1-x)^2 + 2xp_2 * (1-x) + p_3x^2$ where $x \in [0,1]$ (Joy, 2000).

Design Plan

The nature of this project was digital. As such, the only real material needed is a computer with a modern OS. Linux, Mac OS X, and Windows operating systems are capable of running the developed project. Ruby was chosen as the sole language. Downloads and installation instructions can be found at https://www.ruby-lang.org/en/downloads/. The RubyGems program and repository is also needed. The gem ruby-processing is used for all the graphical parts to the program. Installation is facilitated through RubyGems. All source code is publicly available for anyone to download at http://github.com/micleo2/automated-intersections.

The program can be deconstructed into three phases: setup, update, and termination. Each of these phases have many steps within them. Setup is called once on program initialization. Update is called many times each second, and loops until termination. Termination is invoked when the program ends. In the description of these steps, the actual code base will be referenced, which is located in the appendix part of this paper. Lookup of the code is not necessary, but is placed in a way that may be beneficial to the reader, as it gives a concrete implementation of what is being described. The main file, shape_runner.rb, is where all setup and draw are located (see Appendix A)

1. Setup
    a. Create a driver agent $D = (m,\ p,\ v,\ F_{max},\ s_{max})$. The position should be determined by a random selection of all open lanes. Any entry lane has exactly

three exit directions; one should also be determined randomly. There is now an exit point p_{exit} and entry point p_{entry} associated with any given car D.

b. Upon creation and insertion into the simulation, each car is assigned a value t which represents the time spent inside the intersection.

c. For every car, create a curve for the car using the entry and exit points. The curve C is a Bezier curve with three vector points p_1, p_2, p_3. A point may be be obtained from any curve using the function $f(x) = p_1 * (1-x)^2 + 2xp_2 * (1-x) + p_3x^2$ where $x \in [0,1]$. $p_1 = p_{entry}$ and $p_3 = p_{exit}$. The middle control point p_2 is equal to the center of the intersection. To a certain extent, x now represents the point at that "percentage" along the curve. For example, when $x = 1$ then $f(1) = p_3$. Similarly, when $x = 0$ then $f(0) = p_1$.

d. A sequence of points R will become the route for the driver agent. The sequence is created from 2 parameters n and C, where n is a nonnegative integer and C is a quadratic Bezier curve. The number n represents how many points will be in the sequence R. n decimal values in the range of $[0,1]$ will be uniformly distributed and then used as the input value to the function f. The points created from this process make up the sequence R. At this point, the program can dispose of C and hold reference to just R.

e. Create three polygons for each car, one which will be called hitbox, the other castbox, and the last collidebox (See Appendix B for the different polygons which

can be created.) The collidebox will serve as the actual bounds for the cars, and if two collideboxes ever touch, then a crash must be reported. The purpose of hitbox and castbox will be explained soon.

2. Update loop
    a. Appendix C contains much of the implementation for these steps.
    b. On each update, t is incremented by 1.
    c. Share the transformations and rotations applied to the car with the three polygons.
    d. Every driver agent needs a point g which represents the current target. The driver should apply the *seek* behavior towards g at all times when crossing the intersection. g starts out as R_0.
    e. There is a threshold distance d which is the same for all driver agents. When $|p - g| < d$ then $g = R_{i+1}$ where i is the the current index of g. In essence, once D approaches its current target to a sufficient distance, it moves on to the next target.
    f. If the car is not yet in the intersection, it abides by lane management behavior. The cars must maintain constant distance between the cars ahead and behind them.
    g. Perform a collision check on hitboxes to castingboxes. If there exists a collision, then one car must be selected by the ranking priority to proceed. First, *t* is considered: the car which has been at the intersection longer is given priority. In the case of a tie, the speed is used.

h. If $g = R_n$ then D has completed its path through the intersection. t ceases to be incremented and is added onto the sequence T.

i. When D finishes driving through its exit lane, it can now be removed from the simulation.

3. Termination

a. Watch for a button press on the exit button. If the user clicks the button, begin termination.

b. Stop rendering and close the window.

c. Save all data by printing relevant data to the console and to a .txt file with a randomly generated filename. Saved data includes the sequence T and the efficiency y. See Appendix D for the code which saves the data.

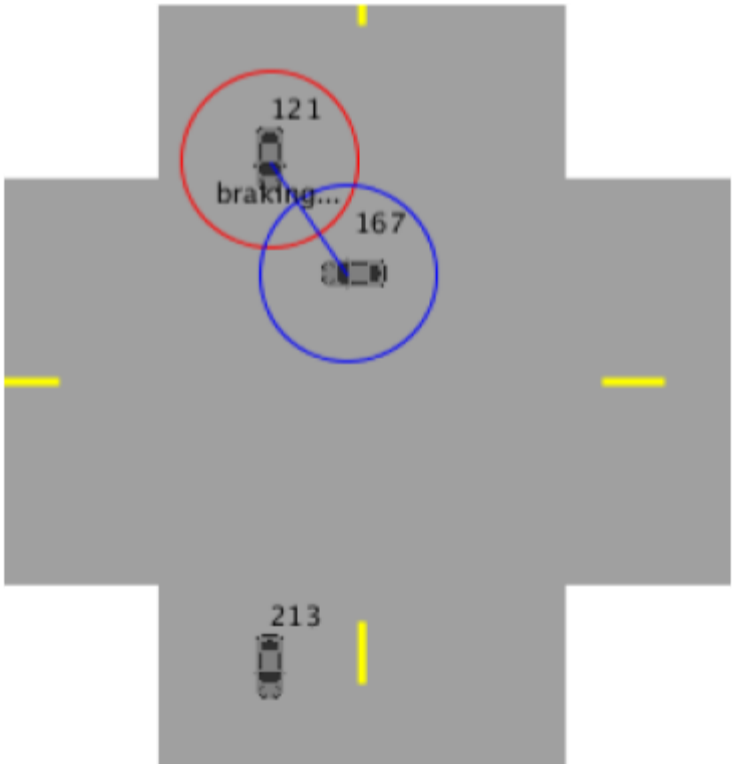
Results of Testing and Redesign

There were mainly three different models created. Each one was tested in the same way. The mean of the wait times and rate of crashes were recorded. These two represent the efficiency and safety of the model, respectively.

Design #1 circle-circle model

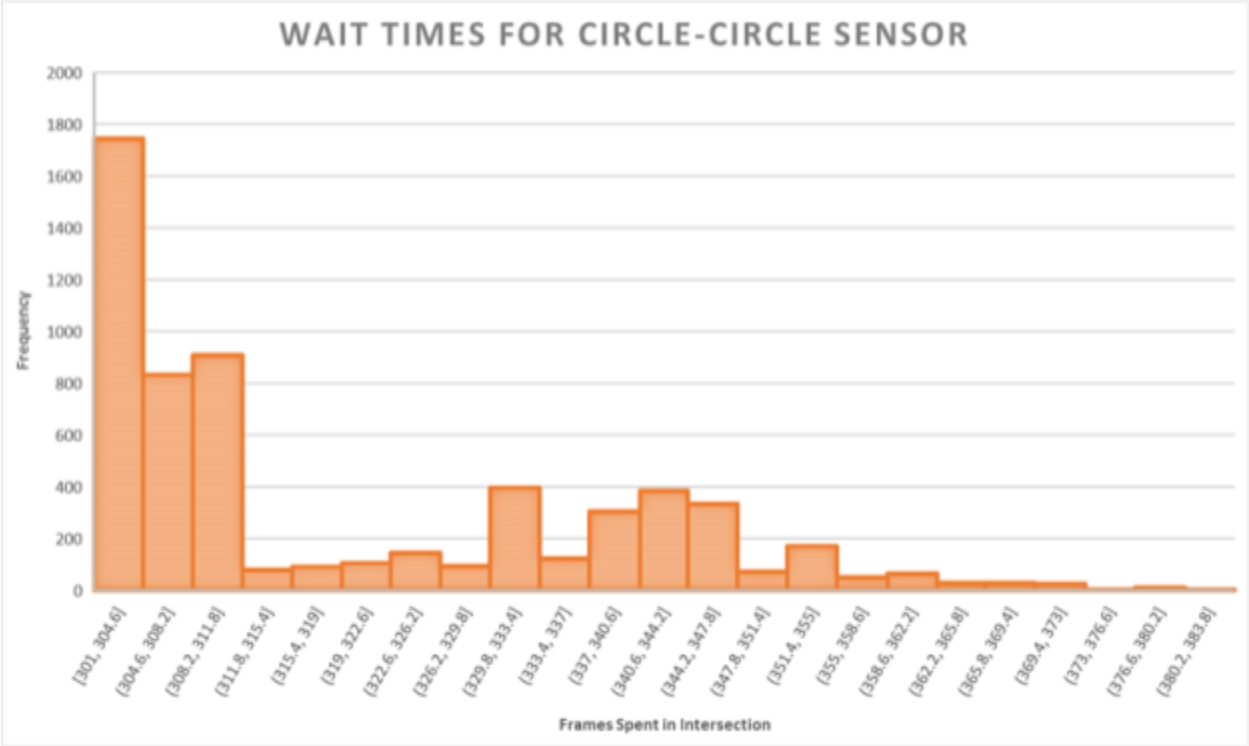
The first model used a simple distance check between two cars to determine whether or not to apply the braking rules. This model was the most simple and easy to implement.

Figure 1, circle-circle model



The program was run to service 5960 cars. *Graph 1* shows the distribution of the wait times. It is skewed right, with a $\bar{x} = 319.04$. The crash rate of the program was $\frac{393}{5960} = 6.59\%$. The skewed right shape is found with all other distributions as well. This is most likely due to the fact that the mean, located leftmost, is the optimal outcome for a car entering into the intersection. Many times, the car can cross through almost unhindered.

Graph 1, distribution of wait times for circle-circle model



Summary data for *Graph 1* can be found in *Table 1*.

Table 1, summary data for circle-circle model

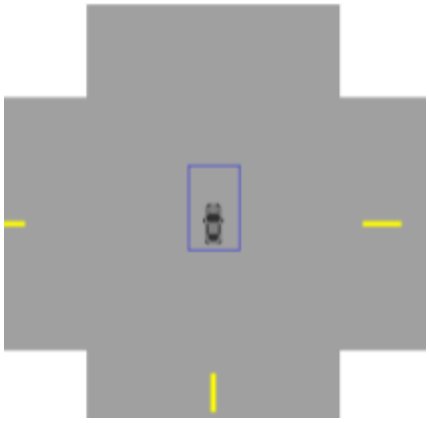
Summary Statistics	
Mean	319.0371051
Standard Error	0.244441159
Median	311
Mode	301
Standard Deviation	18.9074881
Crash Rate	6.59%

The main issue with this approach is that it includes area behind the car. When the cars are moving forward, it's nonsensical to look at whether or not it is intersecting with the road behind the car. This was the main inspiration for coming up with the casting box and hitbox system.

Design #2 triangle-rectangle model

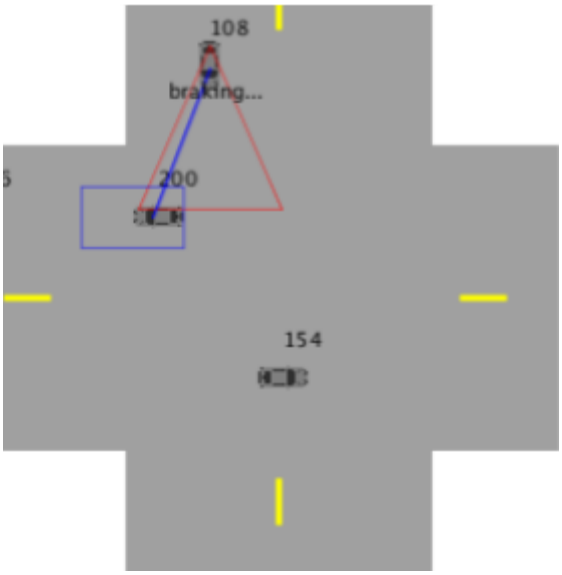
The next two models are more similar to each other than to the first one. They use the system described in the design plan of using two different shapes for evaluating whether or not to apply braking rules. The hitbox shape for both stayed constant as a rectangle. *Figure 2* shows the hitbox.

Figure 2, hitbox



The first castingbox used was a triangle. This was designed to mimic the field of view of a person. *Figure 3* shows an interaction between the castingbox and the hitbox.

Figure 3, triangle-rectangle interaction



The safety rate was significantly improved, but the efficiency did take a turn for the worse.

Graph 2 and *Table 2* show the data from a program run which serviced 5937 cars.

Graph 2, distribution of wait times for triangle-rectangle model

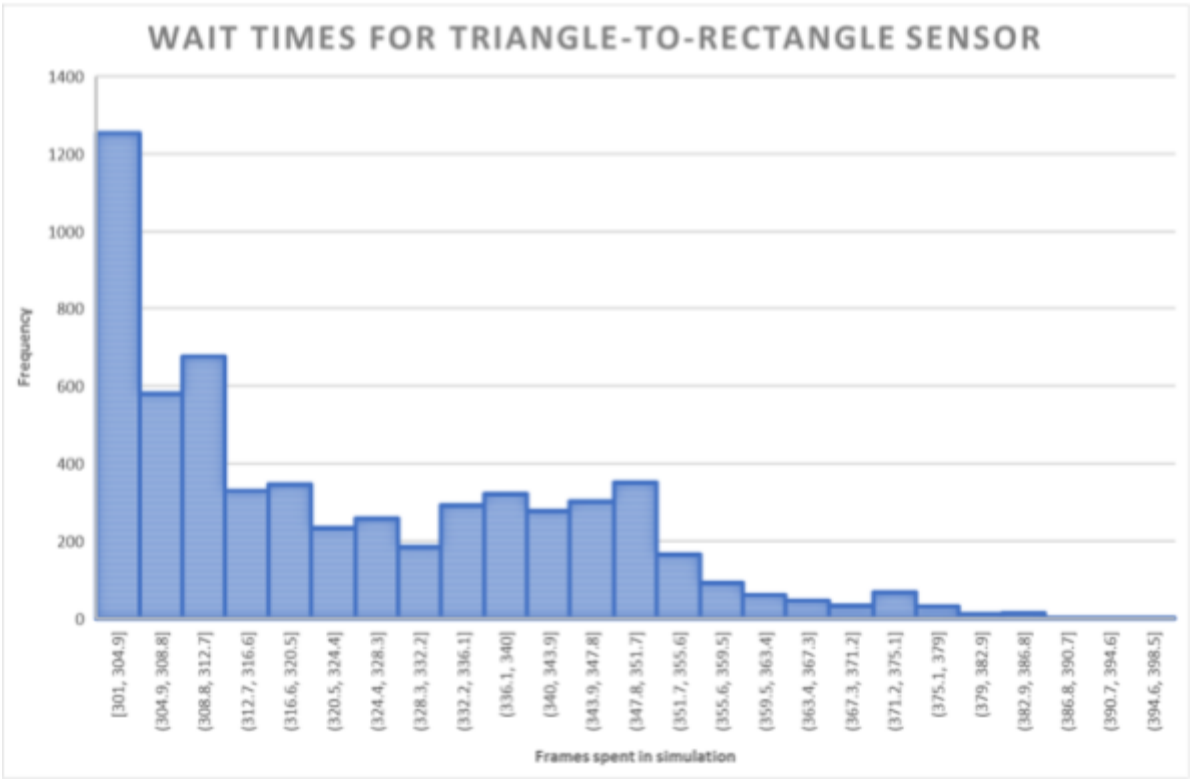


Table 2, summary data for triangle-rectangle model

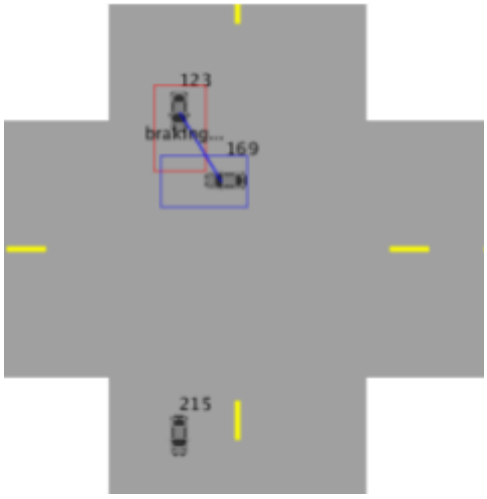
Summary Statistics	
Mean	323.4669
Standard Error	0.2597502
Median	318
Mode	301
Standard Deviation	20.014254
Crash Rate	3.24%

Another noteworthy difference between model 1 and model 2 is the standard deviation. Model 2 experienced more variation, evident with a higher standard deviation and a more "smoothed out" distribution. It's not as concentrated as model 1.

Design #3 rectangle-rectangle model

This model turned the castingbox into a rectangle, the same rectangle which the hitbox uses. *Figure 4* shows an interaction in this new model.

Figure 4, rectangle-rectangle interaction



There was a large improvement in efficiency, however the safety rate again rose to levels similar to model 1. *Graph 3* and *Table 3* show the data from a program run which serviced 6098 cars.

Graph 3, distributions of wait times for rectangle-rectangle model

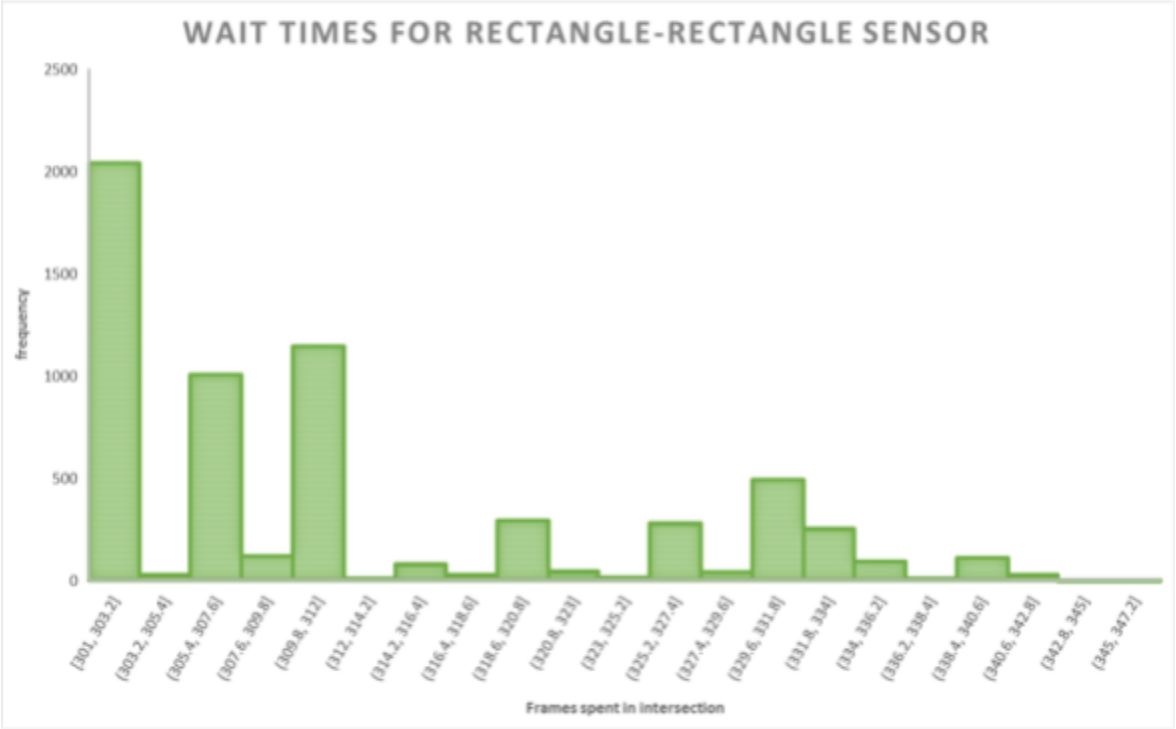


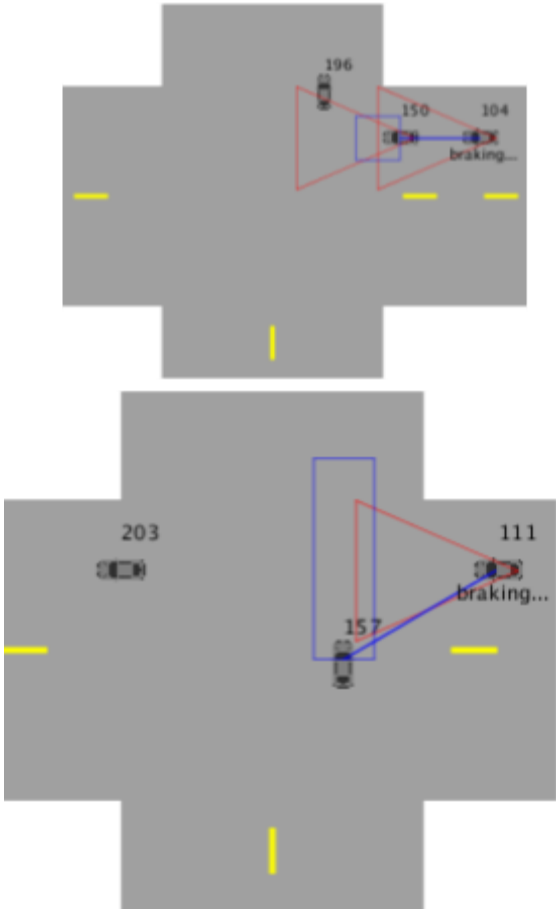
Table 3, summary data for rectangle-rectangle model

Summary Statistics	
Mean	311.74565
Standard Error	0.1496912
Median	306
Mode	301
Standard Deviation	11.689342
Crash Rate	6.46%

Design #4 changing rectangle-rectangle model

This model was the first one that changed the algorithm. The three previous models all simply manipulated the shapes. This last one took the rectangle model and changed its length on the fly. It's directly based off of the velocity of the vehicle: a faster car will draw a longer casting box, while a slower one will yield a smaller casting box. *Figure 5* illustrates one car which is braking, so its blue casting shape is shorter than the other car, which is travelling at top speed.

Figure 5, differing castbox lengths



The motive behind this is simple: if a car is moving at a higher speed, there should be a larger area that its castingbox takes up. This intuitive thought translated well into better performance for the program. *Graph 4* and *Table 4* show the data from this model.

Graph 4, distribution of wait times for dynamic triangle-rectangle model

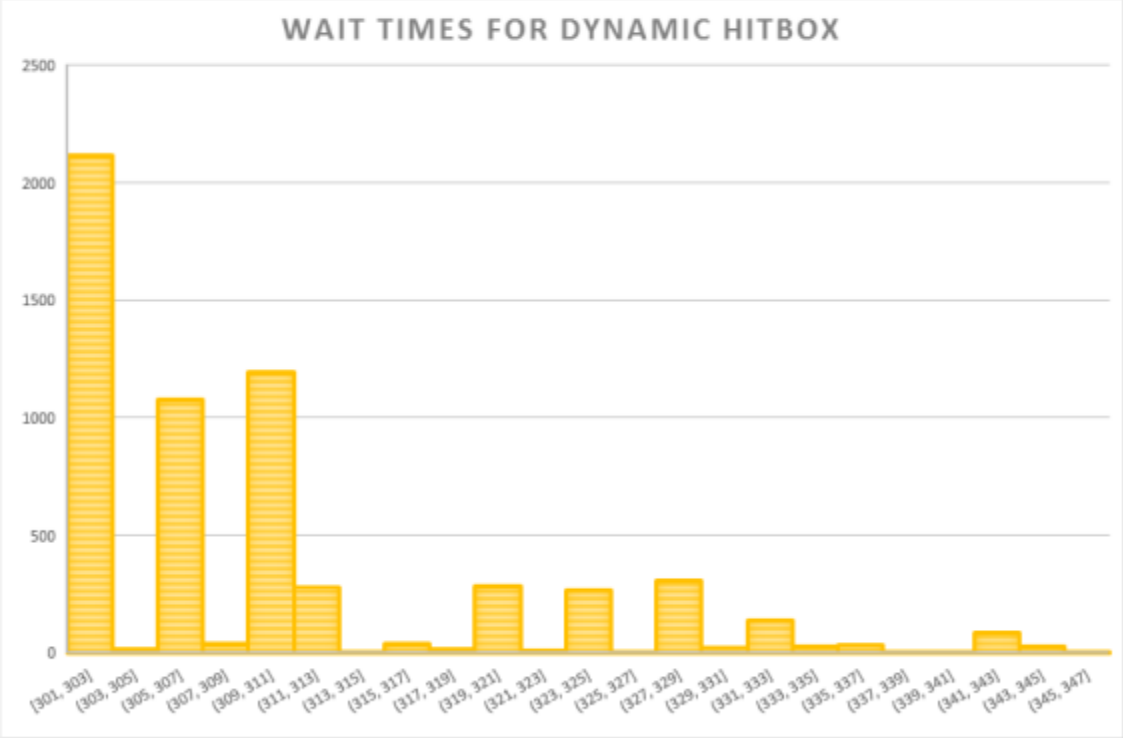


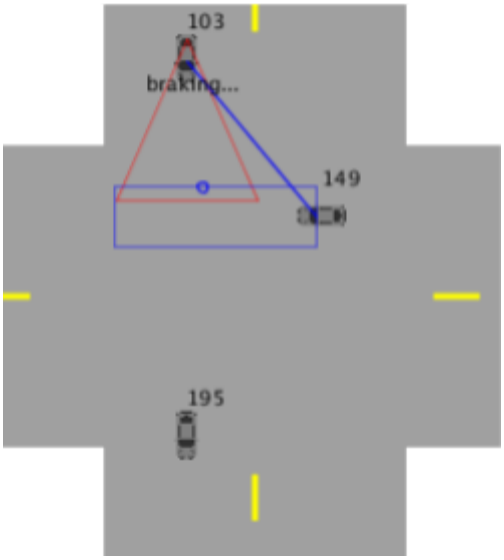
Table 4, summary data for dynamic rectangle-rectangle model

Summary Statistics	
Mean	310.0998161
Standard Error	0.133548165
Median	306
Mode	301
Standard Deviation	10.32820447
Crash Rate	2.74%

Design #5 optional acceleration

The last and best performing model took model 4 and made one more addition. In all previous cases, the car with the smaller *t* variable was forced to yield to the other vehicle. However, in many cases this was not the optimal action in terms of efficiency. Worse, in some cases it actually created traffic crashes that wouldn't have existed otherwise. This problem was exacerbated when the hitbox was extended out farther with respect to higher velocities. To solve this growing problem, an override control was given to the car with the lower *t*. The median point between the two points was calculated. If the car which was supposed to brake is significantly close to this point than the other one, then it is allowed to override the braking behavior and instead apply acceleration. Figure 6 shows the median point being drawn.

Figure 6, optional acceleration



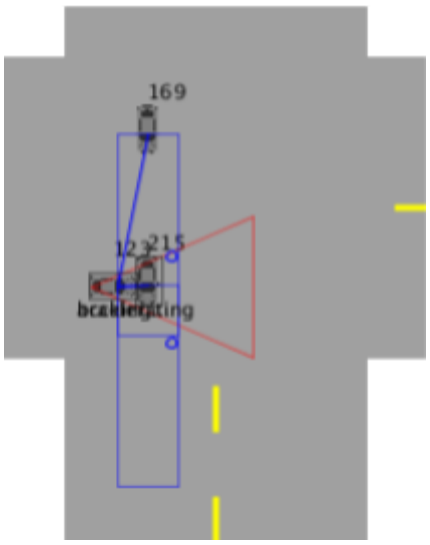
The crash rate was brought down to the ultimate lowest of any model: 0.97%. *Graph 5* and *Table 5* shows the data from this fifth model.

Conclusion

The goal of the program was to create a safe and efficient intersection for use in public roads. The testing and redesign showed strong points of improvement with respect to both factors in different models, but no one model was able to provide a union of both highest efficiency and safety. Model 2 improved over model 1 in regards to safety, a huge enhancement from 6.59% to 3.24%. Model 3 did better in the efficiency category, moving from $\bar{x}$ = 319.04 frames for model 1 to $\bar{x}$ = 311.04 frames for model 3. Models 4 and 5 offered a much better crash rate than the first three. Unfortunately, even the lowest crash rate of 0.97% is too high for installation on public roads.

Looking at sources of error is critical at this stage. Due to the digital nature of the project, any errors come from the underlying algorithm itself: it is impossible to blame crashes on wet roads or bad reaction times. Model 5 will be investigated more closely because it offered the best safety rate, which is the more important factor out of efficiency and safety. Figure 6 is a screenshot of one of the crash scenes from this model.

Figure 6



The typical crash from this model involves two or more cars travelling in the same direction, intersecting with one car travelling in a the perpendicular direction. Using *Figure 6* as a case study, the two cars (t=169, t=215) travelling South both have priority over the car moving East (car with t=123). However, when the median point between car 123 and 215 is calculated, 123 is much closer so it's allowed to override the braking behavior. It does not notice that there is another car behind 215, so it crashes with car 169. The most promising continuation is target points manipulation to the algorithm as crash avoidance techniques. Target point manipulation holds a lot of potential, but at the same time can allow for an explosion in the amount of complexity for the algorithm. This continuation would allow for the points generated for the seek behavior to be tweaked with, altering the trajectory of cars so that a crash is never possible in the first place. This would create minor swerving movements into the vehicle's path, letting the car avoid crashing situations.

Though the current project may not be suitable for installation in self driving cars, the improvement from the models shows promise. It is evidence that perhaps a decentralized algorithm can be sufficient to organize a traffic intersection, in a safe and efficient manner.

Works Cited

Choi, E.-H. (n.d.). Crash Factors in Intersection-Related Crashes: An On-Scene Perspective. PsycEXTRA Dataset. doi:10.1037/e621942011-001

Frequently Asked Questions – Part 4 – Highway Traffic Signals. (2016, August 31). Retrieved November 3, 2016, from http://mutcd.fhwa.dot.gov/knowledge/faqs/faq_part4.htm

Hydén, C., & Várhelyi, A. (2000). The effects on safety, time consumption and environment of large scale use of roundabouts in an urban area: A case study. *Accident Analysis & Prevention, 32*(1), 11-23. doi:10.1016/s0001-4575(99)00044-5

Kurt Dresner & Peter Stone, 3/08/2015, "A multiagent Approach to Autonomous Intersection Management", *Journal of Artificial Intelligence Research*, Volume 31, 591-656

M., A. &., B., T., A., M., J., D., S., & M., D. (n.d.). Statistics on Intersection Accidents. Retrieved October 03, 2016, from https://www.autoaccident.com/statistics-on-intersection-accidents.html

Macek, K., Becker, M., & Siegwart, R. (2007, January 15). Motion Planning for Car-Like Vehicles in Dynamic Urban Scenarios. *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*. doi:10.1109/iros.2006.282013

Pros and Cons of Traffic Signals. (n.d.). Retrieved October 25, 2016, from https://azdot.gov/business/engineering-and-construction/traffic/faq/pros-and-cons-of-traffic-signals

Retting, R. A., Weinstein, H. B., & Solomon, M. G. (2003). Analysis of motor-vehicle crashes at stop signs in four U.S. cities. *Journal of Safety Research, 34*(5), 485-489. doi:10.1016/j.jsr.2003.05.001

Reynolds, C. W. (1987). Flocks, herds and schools: A distributed behavioral model. *ACM SIGGRAPH Computer Graphics, 21*(4), 25-34. doi:10.1145/37402.37406

Appendix A

Code for shape_runner.rb, the main file which glues together all components of the program. Lines 15-21 correspond to the setup, and lines 40-59 correspond to the update.

```
14  def setup
15    size $width, $height
16    @all_cars = []
17    @timer = 0
18    @spawner = CarSpawner.new @all_cars, 100, $width, $height
19    @stats = SummaryStatistics.new
20    @exit_button = Button.new 50, 50, 100, 45, "EXIT"
21    @debug_button = Button.new $width-150, 50, 100, 45, "#{Config::debug?}"
22  end
23
24  def mouse_pressed
25    @exit_button.on_click do
26      @stats.save_data
27      exit()
28    end
29    @debug_button.on_click do
30      Config::toggle
31      @debug_button.message = "#{Config::debug?}"
32    end
33  end
34
35  def draw
36    frame_rate 20
37    background 255
38    Scenery::draw_road 150
39    Scenery::draw_lanes
40    @timer -= 1
41    @all_cars.each(&:draw)
42    @all_cars.each(&:update)
43    @all_cars.delete_if do |c|
44      if c.path.reached_destination? c.agent
45        @stats.wait_times << c.agent.time_in_intersection
46        true
47      else
48        false
49      end
50    end
51
52    if @timer < 0
53      car = @spawner.create_car
54      car.stats = @stats
55      @all_cars << car
56      @timer = 45
57    end
58    @all_cars.each{|c| c.react_to @all_cars}
59    [@exit_button, @debug_button].each(&:draw)
60
61    if Config::debug?
62      fill 0, 255, 0
63      text "Average wait time: #{@stats.avg_time}", 15, $height - 150
64      text "Crashes: #{@stats.crashes/2}", 15, $height - 100
65      text "Cars serviced: #{@stats.wait_times.length}", 15, $height - 50
66    end
67  end
```

Appendix B

Code for shape_factory.rb, the file which is capable of creating different polygons for use as a hitbox, castbox, or collidebox. These coordinates are relative to the position car.

```ruby
require_relative "shape"
class ShapeFactory
  class << self
    def create_triangle
      s = Shape.new
      s.verticies << Vec2D.new(-35, -70)
      s.verticies << Vec2D.new(35, -70)
      s.verticies << Vec2D.new(0, 10)
      s
    end

    def create_rectangle
      s = Shape.new
      w = 30
      s.verticies << Vec2D.new(-w/2, -35)
      s.verticies << Vec2D.new(w/2, -35)
      s.verticies << Vec2D.new(w/2, 15)
      s.verticies << Vec2D.new(-w/2, 15)
      s
    end

    def create_bounds
      s = Shape.new
      w = 14
      h = 29
      s.verticies << Vec2D.new(w/2-1, -h/2 + 5)
      s.verticies << Vec2D.new(-w/2, -h/2 + 5)
      s.verticies << Vec2D.new(-w/2, h/2)
      s.verticies << Vec2D.new(w/2-1, h/2)
      s
    end

    def create_cast
      s = Shape.new
      s.verticies << Vec2D.new(-35, -70)
      s.verticies << Vec2D.new(35, -70)
      s.verticies << Vec2D.new(30, 7)
      s.verticies << Vec2D.new(-30, 7)
      s
    end

    def create_semi_circle
      s = Shape.new
      w = 55
      s.verticies << Vec2D.new(w, 0)
      s.verticies << Vec2D.new(-w, 0)
      s.verticies << Vec2D.new(-w, -40)
      s.verticies << Vec2D.new(-w*0.6, -75)
      s.verticies << Vec2D.new(w*0.6, -75)
      s.verticies << Vec2D.new(w, -40)
      s
    end
  end
end
```

Appendix C

Code for driver_agent.rb, specifically the update method. The *time_in_intersection* variable is *t* from the design plan. Line 21 is responsible for updating *g*, the current target point.

```
20    def update
21      @path.adjust_to @agent
22      @agent.steering.seek @path.current_point
23      @agent.steering.update
24      @agent.time_in_intersection += 1
25      ang = Math::atan2 @agent.velocity.x, -@agent.velocity.y
26      [@castbox, @hitbox, @collide_box].each{|b| b.align_to ang}
27    end
```

Appendix D

Code for summary_statistics.rb, specifically the save_data method. Data is written once to the console then again to an external .txt file.

```
10    def save_data
11      # puts "Wait times: #{@wait_times}"
12      puts "Average wait time: #{avg_time}"
13      puts "Crashes: #{@crashes/2}"
14      puts "Cars serviced: #{@wait_times.length}" #amount of cars serviced
15      #log output to a text file
16      f = File.new(MathUtil::unique_name + ".txt", "w")
17      f.puts "Average wait time: #{avg_time}"
18      f.puts "Crashes: #{crashes/2}"
19      f.puts "Cars serviced: #{@wait_times.length}"
20      f.puts "Crash rate: #{(@crashes/2).fdiv @wait_times.length}"
21      @wait_times.each{|t| f.puts t}
22      f.close
23    end
```