Self-driving Infrastructure: Final Report

Integrated Engineering Design Project 2: ENGINEER 2PX3

Design Studio 01

SDI-11

April 12, 2023

Harry Zihe Shi 400301295

Kush Rana 400405522

Teghveer Singh Ateliey 400409275

Michael Yip 400401586

Executive Summary

The purpose of this report is to provide an overview of the self-driving infrastructure project. The goal of this project is to redesign the traditional intersections in Hamilton with a self-driving infrastructure that incorporates modern technology. Current automotive intersections lack communication between vehicles and often create dangerous situations for drivers. For example, at a left turn signal, a driver may have difficulty seeing incoming traffic as it is blocked by vehicles on the other side of the road, waiting to turn. Additionally, there exists a lack of synchronicity between vehicles which causes a “Phantom effect” which is when one driver fails to accelerate at a green light, causing the delay to compound and affect all the other cars that are waiting[1]. The final recommendation for this project is to incorporate various sensors throughout a traffic intersection that monitor the number of cars at each section. Additionally, the physical structure of the automotive infrastructure will be revised with the use of bridges which can allow traffic to be in continuous motion especially on left signal turns. In the new model of the automotive intersection, there will be separate sections for left turns which increases the safety of the intersection. Images and further analysis of that model will be discussed in this report. Additionally, there should be poles with mounted sensors that can act as hubs which acquire and manage data transmitted from the self-driving vehicles. This feature will allow the intersection model to reap the benefits of the latest technologies. This design is more effective than previous iterations. The proposed design is more effective than previous iterations that included an underground tunnel for vehicles to make left or right turns. The drawbacks of this design are negative impacts on the environment due to habitat displacement, budget overruns, weak network connectivity, and the need for more materials to reinforce the structure to prevent roads from collapsing. The proposed solution builds upward, eliminating these concerns. The next steps moving forward will be to meet with the government and automobile manufacturers to determine safe and respectful limitations of self-driving infrastructure. This would include the need to make decisions about the ethics and decision making of the self-driving vehicle within the infrastructure. Additionally, it would be important to meet with construction companies to manage the feasibility of the construction of the overpass and sensor mounts used in the intersection.

Introduction

The self-driving infrastructure project is focused on improving the current automotive intersection to be aligned with the rising popularity of self-driving vehicles. The stakeholders involved in this project are the government, the drivers of human and self-driven vehicles, the passengers, pedestrians, manufacturing companies, and more. These stakeholders all possess varying concerns which can be addressed by the PERSEID design method[2]. For example, the government's primary concern is the regulatory aspect of the system while its secondary concern is socio-cultural impacts. The main concern of the manufacturers is performance as they want an effective product. To manage the ethical aspects of the system, the team has decided that the conservation of human life will always be the priority of the system when making decisions. For example, if the vehicle had to decide between hitting an unexpected pedestrian or swerving and hitting a tree, then the vehicle would do so. In order to mitigate the impacts on the environment, the team would also recommend the materials be sourced ethically and sustainably. For constraints, the team must only consider the surrounding buildings as we cannot expand lanes into someone's property. Additionally, the scheduling of the project and the budget are also notable constraints. The objectives are to design a system that should improve traffic flow, create a safer environment, and implement the latest self-driving technologies in a manner that is in respect to the PERSEID design method. Lastly, one assumption that has been validated is that this intersection is mainly for self-driven and human-driven driven.

Design Process

Our initial recommendations for the client were based on the PERSEID method, taking in to account the needs and wants of our stakeholders. These considerations are to implement vehicle communication (i.e., V2X, V2P, or V2I), road expansion, dynamic traffic signalling algorithms (like SCATS (Sydney Coordinated Adaptive Traffic System) or SCOOT (Split Cycle Offset Optimisation Technique)), 3D-mapping technology, and higher speed limits. The final recommendations did not change substantially as they were deemed feasible and necessary in facilitating better traffic flow. Our team decided to include several additional considerations such as directional antennas, private sector collaboration, and piezoelectric generators. Directional antennas are considered as part of the performance layer because of the environment that our client requests that the intersection design be implemented in which is in an urban setting. Therefore, signals are susceptible to interference from high-rise buildings which are commonplace in an urban setting. By implementing directional antennas which could assist with V2X technology, signals can circumvent zones of interference, consequently minimizing the amount of latency that the end user encounters. This is critical in ensuring the safety of road users as well as improving the performance of the infrastructure. Private sector collaboration was considered as part of the socio-cultural layer as it could provide real-time data on where vehicles are located, increasing locational awareness in the V2X network. This could benefit more than 3D-mapping which is limited to the capabilities of LiDAR technology (i.e., range limitations). Some potential partners would be Uber, Lyft, and other ride share corporations which can provide invaluable information to the V2X network. Piezoelectric generators were considered as part of the environmental layer as they present a revolutionary opportunity to generate voltage via pressure from vehicles passing over road surfaces. Such generators are embedded underneath roads which means that they are generally well-protected from inclement weather thereby reducing the frequency for maintenance. Electrical energy harvested from these generators are useful for supplying power to various aspects of the V2X network such as LiDAR sensors, traffic lights, and directional antennas. Overall, implementing piezoelectric generators is an innovative approach to sustainability.

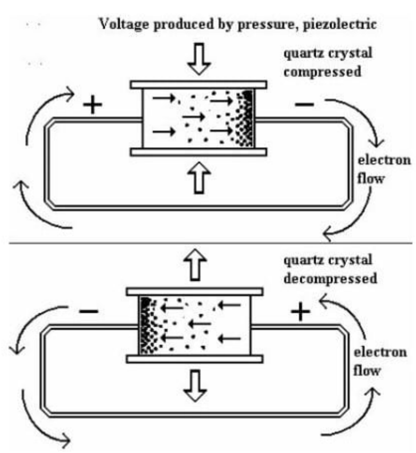


Figure 1. a diagram of the different components of piezoelectric generators by science Canada (2012).

The first client meeting shaped our perception of what needs to be considered as the client provided us with additional details regarding the project scope and what to prioritize. This included the fact that budget is not a limitation, and the safety of road users possess the highest priority. This means that while we ought not to expend extravagantly on state-of-the-art technology, we should still consider reducing the cost of our infrastructure design based on their efficiency and lifetime. For example, piezoelectric generators can be made from long-lasting materials such as polymers meaning that minimal maintenance is required as it harvests would-be wasted mechanical energy from passing vehicles by converting it to electrical energy. Also, the implementation of directional antennas is parallel with the stipulation that the lives of the road users take the highest priority. Directional antennas can minimize latencies which is critical, especially in urban settings, to avoid potential collisions due to miscommunication.

In the second client meeting, our client requested that we measure the estimated emissions based on the occurrence of different types of vehicles. This puts an emphasis on mitigating the environmental impacts. As such, this reinforces our consideration of implementing piezoelectric generators as the only aspects we can control are improving traffic flow (i.e., performance) and harvesting energy from existing processes (i.e., cars moving over roadways).

Table 1. Decision matrix

	Price	Safety	Efficiency	Installation	Total
V2X	6	9	9	5	29
3D-Mapping	7	8	8	7	30
Private Sector Collaboration	4	8	9	7	28
Wider roads	1	9	9	1	20
Sensor technology	8	8	6	4	26
Piezoelectric generators	1	8	8	3	20
Underground roads	1	6	6	1	14

Round-abouts	6	6	4	2	18

Based on the decision matrix that was constructed, roundabouts and underground roads are not feasible relative to the other considerations based on price, safety, efficiency, and installation categories. It can be observed that V2X, 3D-mapping, private sector collaboration, and sensor technology have the highest and relatively similar scores which is reasonable as those technologies are necessary to facilitate the V2X network. The wider roads consideration has a mediocre score that is at par with piezoelectric generators. Wider roads aren't desirable as our intersection is located in an urban setting; therefore, space is limited as we advise the client to restrict intruding into greenery, habitats, and potentially devaluing surrounding properties due to noise pollution. Piezoelectric generators remain in our final considerations as they are the only feasible option in limiting the environmental impact by regenerating electrical energy.

Final Proposed Design

This report's final proposed design incorporates several design choices which optimize specific considerations for the regulatory, environmental, socio-cultural, and performance aspects of the project. Overall, the final proposed design is a high-speed, adaptive intersection maximizes efficiency and the advantages of self-driven vehicles while also minimizing the horizontal space used by the system. This project was designed for use in an urban or suburban setting. As such, space complexity was one of our main design considerations. Alternatives such as roundabout or single-level intersection designs were considered, yet a double-level intersection was chosen largely due to the advantages of building upwards to save space outwards.

The double-level design was also chosen to isolate left turning vehicles in the intersection, as only left-turning vehicles will ascend to the 2nd level of the system. Left turns are the most dangerous maneuver vehicles regularly perform in an intersection [3], and this design aims to increase safety similar to left-turn signals without interfering with other traffic. Isolating left turning vehicles also presents the opportunity to have dedicated auxiliary lanes for traffic moving straight and turning right. This further prevents congestion and allows for an optimized traffic signal system.

There are four specific technical design implementations that contribute to the traffic signal system; vehicle-to-everything (V2X), object mapping, traffic signal timing, and private sector collaboration. All of these technologies involve the collection or usage of data to minimize the waiting time of vehicles at the intersection, while also using the information gathered to increase safety by communicating with self-driven vehicles and relieving vehicles from the system efficiently. V2X is among the most important of these technologies, as it facilitates wireless communication between different parts of the infrastructure and self-driven vehicles. V2X protocols boast the ability to transfer complex and critical information with minimal latency and proved more effective than other network alternatives.

Information from object mapping cameras are most commonly transported using V2X. By using object mapping, the infrastructure can determine the positions of human driven cars and predict where human drivers may be travelling to. This is useful in identifying the load on the different parts of the system and when certain vehicles arrived to the system, and can be done with just a few object mapping cameras rather than dedicated sensors for each location in the intersection. Relaying object information to self-driven cars can help autonomous vehicles react to the unexpected actions and possible intentions of human drivers. Data on vehicle location and load can be used with traffic signal timing algorithms to optimize traffic flow.

Sydney Co-ordinated Adaptive Traffic System (SCATS) is a responsive traffic algorithm developed and used in Sydney for controlling urban traffic [3]. Similar algorithms can be used to change the signal states of an intersection in accordance with load at different points of the system without compromising safety. SCATS was chosen for this project as it is recognized as one of the world's best traffic algorithms with up to 21% reduced travel times and 12% reduced fuel consumption. Other systems struggle to compare with the success that SCATS has had worldwide. The data used by this algorithm can be increased exponentially with collaboration from external sources.

By collaborating with private sector corporations such as Uber, Lyft, and Google, the intersection can receive real-time data on traffic flow and can adapt accordingly using SCATS. This information will aid the intersection in adapting to varying loads on the system, which was the subject of the final client request from week 10. Referendums with the local population will need to be held to approve of the use of data collection for infrastructure enhancement. Regardless of the use of private sector collaboration, this intersection was designed to adapt to irregular traffic flow and will not need further changes to accommodate the final client request.

The last design decision included the intersection does not directly support the system's performance. Piezoelectric generators were added to the road in the final proposed design to generate electricity from vehicles using the system, therefore increasing the energy efficiency of the intersection. The generators will be low cost and easy to maintain, adding virtually no downside to implementation [7]. Other experimental energy generation methods were explored, such as thermoelectric and photovoltaic generators. Neither proved to be a better alternative than piezoelectric generators when comparing cost, maintainability, and feasibility in this project [8, 9].

When evaluating the safety of a freshly constructed intersection or interchange, the quantity of conflict locations is a key consideration. Conflict points, which are places where separate vehicles' trajectories cross, can be divided into three types: crossing, merging, or diverging. Crossing conflict spots happen when two moving vehicles cross paths, and because head-on crashes are more likely, they are frequently linked to more severe crash types. On the other hand, conflict locations at which two or more moving vehicles merge or diverge tend to cause fewer serious collisions. This is so that the likelihood of high-impact collisions can be reduced. Merging and diverging motions typically involve cars moving at varying speeds and in the same direction. Therefore, A safer and more effective intersection or interchange design may include fewer crossing conflict sites and more merging and diverging conflict points.[4]

The analysis of a traditional two-way signal intersection, as depicted in the figure below, reveals valuable insights. The data shows that there is a total of 16 crossing conflict points, where vehicles moving in opposite directions intersect. In contrast, there are 8 merging conflict points and 8 diverging conflict points, where vehicles are either merging or diverging in the same direction. A comparison with the typical intersection layout will be done to gauge the effectiveness of the intersection or interchange design. There should be fewer overall conflicts and fewer crossing conflicts because of the novel design, which would suggest enhanced regulatory procedures. This comparison will give important information about the intersection's or interchange's safety performance, prove the efficiency of the design in preventing possible collisions, and raise overall road user safety.

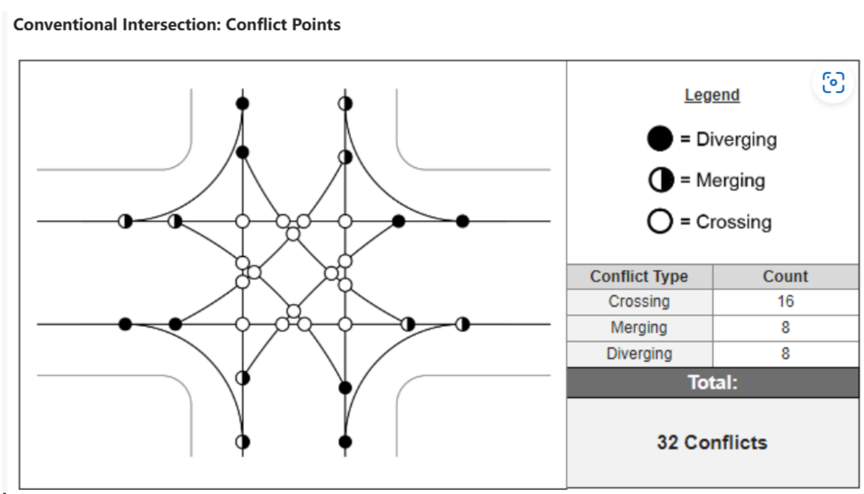


Figure 2

The analysis of the intersection layout for this design project is shown in the image below. Only 8 crossing conflicts, 4 merging conflicts, and 4 diverging conflicts, for a total of 16 conflicts, are shown. Comparatively speaking, there are just half as many disputes as there would be at a typical crossing. These results suggest that the creative intersection or interchange design has successfully decreased the number of conflicts, making it safer from the standpoint of layout. This lends support to the idea that the design has improved safety regulations and can increase overall intersection or interchange user safety. In addition, the V2X and other technologies that have been mentioned before are what the traditional intersection does not have, which will also give the driver better perspectives to improve the safety metrics of the intersections.

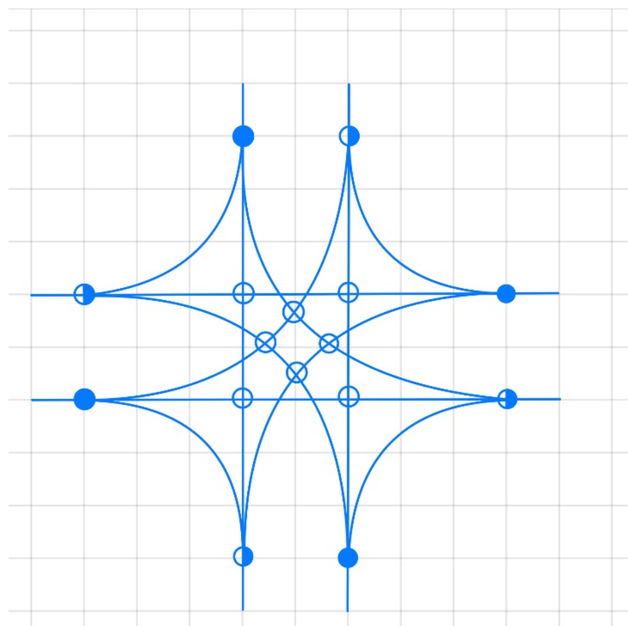


Figure 3

Additionally, a simulation of the intersection has been created to provide a visual representation of its operation. Simulations play a crucial role in data science and machine learning, as they can generate large datasets at a relatively lower cost when obtaining real-world data is not feasible or expensive. Simulations can also help address gaps in real-world data, such as capturing edge cases that may be critical for the development of the model.[6] Simulator development is a complex and difficult procedure that uses modelling methods and mathematical calculations. The appendix contains information on the relevant article and code that is not covered here. However, as long as the waiting automobiles are removed from the page, according to the simulation's findings, the constructed intersection can handle a flow rate of 70 (as stated in the Python code) without encountering traffic congestion for a period of 120 minutes. The typical intersection, in contrast, can only manage a flow rate of 40. The picture below shows a screenshot of the simulation's results.

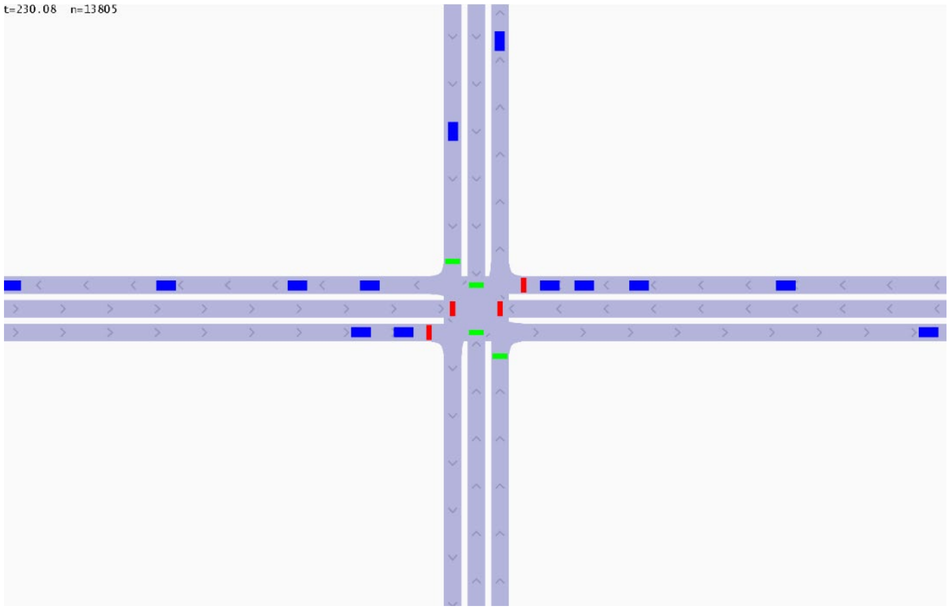


Figure 4 Intersection Simulations

References

[1] What are phantom traffic jams?" Science ABC, 22-Jan-2022. [Online]. Available: https://www.scienceabc.com/eyeopeners/what-are-phantom-traffic-jams.html. [Accessed: 12- April-2023].

[2] "2022-2023 Self Driving Infrastructure Project Summary," class notes for ENGINEER 2PX3, Faculty of Engineering, McMaster University, Winter, 2023.

[3] Feber, David J., Judith M. Feldmeier, Keith J. Crocker. "Dangerous Intersections." Best's Review, March 2000, 80. Gale Academic OneFile. https://link.gale.com/apps/doc/A60805538/AONE?...u=anon~4031e761&sid=googleScholar.. &xid=9cc4bf07. [accessed April 12, 2023].

[4] "Innovative intersections and interchanges," Innovative Intersections and Interchanges - Info ….| Virginia Department of Transportation. [Online]. Available: https://www.virginiadot.org/info/innovative_intersections_and_interchanges/dlt.asp. ….. [Accessed: 02-Feb-2023].

[5] Stevanovic, Aleksandar, Cameron Kergaye, and Peter T. Martin. "Scoot and scats: A closer look into their operations." In 88th Annual Meeting of the Transportation Research Board. Washington DC. 2009. [accessed April 12, 2023].

[6] B. Himite, "Simulating traffic flow in Python," Medium, 07-Sep-2021. [Online]. Available: https://towardsdatascience.com/simulating-traffic-flow-in-python-ee1eab4dd20f. [Accessed: 12-Apr-2023].

[7] S. Kim, "Low Power Energy Harvesting with Piezoelectric Generators." Order No. 3081259, University of Pittsburgh, United States -- Pennsylvania, 2002.

[8] Durisch, Wilhelm, Dierk Tille, A. Wörz, and Waltraud Plapp. "Characterisation of photovoltaic generators." Applied Energy 65, no. 1-4 (2000): 273-284.

[9] Champier, Daniel. "Thermoelectric generators: A review of applications." Energy Conversion and Management 140 (2017): 167-181.

Appendix

Client Request 1: Graph the average travel time as a function of load on the system

```
import random

"""
2PX3 Intersection Simulation Starting Code

Simulation for a "cautious" intersection. Modelling choices:
1) A vehicles arrives from N, E, S, or W and must wait for other cars
ahead of them to clear the intersection.
2) Only one car can be "clearing" the intersection at once.
3) Before a car can begin to clear the intersection, it must come to a stop
4) Cars will clear the intersection in a one-at-a-time counter-clockwise manner

Dr. Vincent Maccio 2022-02-01
"""

#Constants
ARRIVAL = "Arrival"
DEPARTURE = "Departure"
STOP = "Stop"
N = "North"
E = "East"
S = "South"
W = "West"
MEAN_ARRIVAL_TIME = 5
PRINT_EVENTS = False

class Driver:
    stop_time = 5
    clear_time = 10

    def __init__(self, name, arrival_time):
        self.name = name
        self.arrival_time = arrival_time

    #Returns driver instance stop time
    def get_stop_time(self):
        return self.stop_time

    #Returns driver instance clear time
    def get_clear_time(self):
        return self.clear_time


class Event:
    def __init__(self, event_type, time, direction):
        self.type = event_type
        self.time = time
        self.direction = direction


class EventQueue:

    def __init__(self):
        self.events = []

```

```
    #Add event (will get sent to the back of the queue)
    def add_event(self, event):
        #print("Adding event: " + event.type + ", clock: " + str(event.time))
        self.events.append(event)

    #Get the next event in the queue and pop it (remove it)
    #Returns removed next event
    def get_next_event(self):
        min_time = 9999999999999
        min_index = 0
        for i in range(len(self.events)):
            if self.events[i].time < min_time:
                min_time = self.events[i].time
                min_index = i
        event = self.events.pop(min_index)
        #print("Removing event: " + event.type + ", clock: " + str(event.time))

        return event


class Simulation:
    upper_arrival_time = 2 * MEAN_ARRIVAL_TIME
    def __init__(self, total_arrivals):
        self.num_of_arrivals = 0
        self.total_arrivals = total_arrivals
        self.clock = 0

        """
        Each road is represented as a list of waiting cars. You may
        want to consider making a "road" a class.
        """
        self.north, self.east, self.south, self.west = [], [], [], []
        self.north_ready = False
        self.east_ready = False
        self.south_ready = False
        self.west_ready = False
        self.intersection_free = True
        self.events = EventQueue()
        self.generate_arrival()
        self.print_events = PRINT_EVENTS
        self.data = []

    #Enable printing events as the simulation runs
    def enable_print_events(self):
        self.print_events = True

    #Method that runs the simulation
    def run(self):
        while self.num_of_arrivals <= self.total_arrivals:
            if self.print_events:
                self.print_state()
            self.execute_next_event()

    #Execute the next event in the queue
    #(Get next event, and execute appropriate method depending on event type)
    def execute_next_event(self):
        event = self.events.get_next_event()
        self.clock = event.time
        if event.type == ARRIVAL:
            self.execute_arrival(event)
        if event.type == DEPARTURE:
```

```
            self.execute_departure(event)
        if event.type == STOP:
            self.execute_stop(event)

    #Driver leaving intersection event
    def execute_departure(self, event):
        if self.print_events:
            print(str(self.clock)+ ": A driver from the " + event.direction + " has cleared the intersection.")

        #Lots of "traffic logic" below. It's just a counter-clockwise round-robin.
        if event.direction == N:

            #No drivers left to depart from the North
            if self.north == []:
                self.north_ready = False

            #Carry on to other direction waitlists
            if self.west_ready:
                self.depart_from(W)
            elif self.south_ready:
                self.depart_from(S)
            elif self.east_ready:
                self.depart_from(E)
            elif self.north_ready:
                self.depart_from(N)
            else:
                self.intersection_free = True

        if event.direction == E:

            #No drivers left to depart from the East
            if self.east == []:
                self.east_ready = False

            #Carry on to other direction waitlists
            if self.north_ready:
                self.depart_from(N)
            elif self.west_ready:
                self.depart_from(W)
            elif self.south_ready:
                self.depart_from(S)
            elif self.east_ready:
                self.depart_from(E)
            else:
                self.intersection_free = True

        if event.direction == S:

            #No drivers left to depart from the South
            if self.south == []:
                self.south_ready = False

            #Carry on to other direction waitlists
            if self.east_ready:
                self.depart_from(E)
            elif self.north_ready:
                self.depart_from(N)
            elif self.west_ready:
                self.depart_from(W)
```

```
            elif self.south_ready:
                self.depart_from(S)
            else:
                self.intersection_free = True

        if event.direction == W:

            #No drivers left to depart from the West
            if self.west == []:
                self.west_ready = False

            #Carry on to other direction waitlists
            if self.south_ready:
                self.depart_from(S)
            elif self.east_ready:
                self.depart_from(E)
            elif self.north_ready:
                self.depart_from(N)
            elif self.west_ready:
                self.depart_from(W)
            else:
                self.intersection_free = True

    #Create departure event for the first driver from the queue in the passed direction
    def depart_from(self, direction):

        #Make departure event for first car in North queue
        if direction == N:
            clear_time = self.clock + self.north[0].get_clear_time()
            new_event = Event(DEPARTURE, clear_time, N)
            driver = self.north.pop(0) #Car progessing into the intersection

        #Make departure event for first car in East queue
        if direction == E:
            clear_time = self.clock + self.east[0].get_clear_time()
            new_event = Event(DEPARTURE, clear_time, E)
            driver = self.east.pop(0) #Car progessing into the intersection

        #Make departure event for first car in South queue
        if direction == S:
            clear_time = self.clock + self.south[0].get_clear_time()
            new_event = Event(DEPARTURE, clear_time, S)
            driver = self.south.pop(0) #Car progessing into the intersection

        #Make departure event for first car in West queue
        if direction == W:
            clear_time = self.clock + self.west[0].get_clear_time()
            new_event = Event(DEPARTURE, clear_time, W)
            driver = self.west.pop(0) #Car progessing into the intersection

        self.events.add_event(new_event)
        self.intersection_free = False
        self.data.append(clear_time - driver.arrival_time)

    #Stop driver at intersection, and call depart method to add depart event to queue
    def execute_stop(self, event):
        if self.print_events:
            print(str(self.clock)+ ": A driver from the " + event.direction + " has stopped.")
```

```python

        if event.direction == N:
            self.north_ready = True
            if self.intersection_free:
                self.depart_from(N)

        if event.direction == E:
            self.east_ready = True
            if self.intersection_free:
                self.depart_from(E)

        if event.direction == S:
            self.south_ready = True
            if self.intersection_free:
                self.depart_from(S)

        if event.direction == W:
            self.west_ready = True
            if self.intersection_free:
                self.depart_from(W)

    #Start arrival event
    def execute_arrival(self, event):
        driver = Driver(self.num_of_arrivals, self.clock)
        if self.print_events:
            print(str(self.clock)+ ": A driver arrives from the " + event.direction + ".")

        if event.direction == N:
            if self.north == []: #Car needs to stop before clearing
                self.north_ready = False
            self.north.append(driver)
            new_event = Event(STOP, self.clock + driver.get_stop_time(), N)
            self.events.add_event(new_event)

        elif event.direction == E:
            if self.east == []: #Car needs to stop before clearing
                self.east_ready = False
            self.east.append(driver)
            new_event = Event(STOP, self.clock + driver.get_stop_time(), E)
            self.events.add_event(new_event)

        elif event.direction == S:
            if self.south == []: #Car needs to stop before clearing
                self.south_ready = False
            self.south.append(driver)
            new_event = Event(STOP, self.clock + driver.get_stop_time(), S)
            self.events.add_event(new_event)

        else:
            if self.west == []: #Car needs to stop before clearing
                self.west_ready = False
            self.west.append(driver)
            new_event = Event(STOP, self.clock + driver.get_stop_time(), W)
            self.events.add_event(new_event)

        self.generate_arrival() #Generate the next arrival

    #Generate a car arriving at the intersection
    def generate_arrival(self):
        #Generates a random number uniformily between 0 and upper_arrival_time
```

```
        inter_arrival_time = random.random() * self.upper_arrival_time
        time = self.clock + inter_arrival_time

        r = random.random()
        #Equally likely to arrive from each direction
        if r < 0.25:
            self.events.add_event(Event(ARRIVAL, time, N))
        elif r < 0.5:
            self.events.add_event(Event(ARRIVAL, time, E))
        elif r < 0.75:
            self.events.add_event(Event(ARRIVAL, time, S))
        else:
            self.events.add_event(Event(ARRIVAL, time, W))
        self.num_of_arrivals += 1 #Needed for the simulation to terminate

    def print_state(self):
        print("[N,E,S,W] = ["+ str(len(self.north)) + ","+ str(len(self.east)) + ","+ str(len(self.south)) +","+ str(len(self.west)) +"]")

    def generate_report(self):
        #Define a method to generate statistical results based on the time values stored in self.data
        #These could included but are not limited to: mean, variance, quartiles, etc.
        print()



def average(L):
    return sum(L)/len(L)

time_array = []
# To simulate for different load
for x in range(10,101,10):
    sim = Simulation(x)
    sim.run()
    #To get the average time
    time_array.append(average(sim.data))

print(time_array)
# init load array
load_array = list(range(10,101,10))
print(load_array)

import matplotlib.pyplot as plt
# x-axis: load; y-axis: time
plt.plot(load_array,time_array)
plt.ylabel('time(s)')
plt.xlabel('load(num)')
plt.show()
```

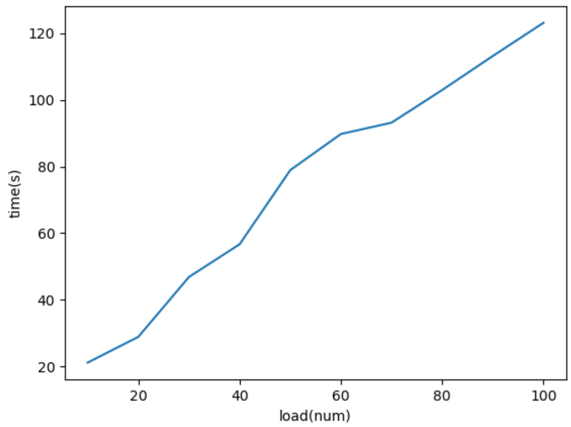


Figure # number of cars vs. the average waiting time

Client Request 2: graphical representation of the diff in performance of self-driving and human driven.

```
import random
import matplotlib.pyplot as plt


#Constants
#   Event types:
ARRIVAL = "Arrival"
STOP = "Stop"
CLEAR = "Clear"

#   Directions:
N = "North"
E = "East"
S = "South"
W = "West"


#Controls
TIME_STEP = 0.5
ARRIVAL_TIME = 10
HUMAN_CLEAR_TIME = 3.5
SDC_CLEAR_TIME = 2
HUMAN_TURNING_TIME = 5.5
SDC_TURNING_TIME = 4
HUMAN_MIN_STOP_TIME = 2.5
SDC_MIN_STOP_TIME = 1


NORTH_CAR_PROBABILITY = 0.1
EAST_CAR_PROBABILITY = 0.1
```

```
SOUTH_CAR_PROBABILITY = 0.1
WEST_CAR_PROBABILITY = 0.1


TURN_PROBABILITY = 0.1
LEFT_TURN_PROBABILITY = 0.33
HUMAN_PROBABILITY = 0.9

#Classes

class Driver:

    def __init__(self, name, time, arrival_time, direction_from, direction_to, is_human):
        self.name = name
        self.is_human = is_human
        self.event = ARRIVAL
        self.start_time = time
        self.direction_from = direction_from
        self.direction_to = direction_to
        self.elapsed_time = 0
        self.busy_time = arrival_time

    def get_from_to(self):
        return [self.direction_from, self.direction_to]

class DriverQueue:

    def __init__(self):
        self.north, self.east, self.south, self.west = [], [], [], []
        self.north_stop, self.east_stop, self.south_stop, self.west_stop = [], [], [], []
        self.intersection = []

    def add_driver_arrivals(self, driver):
        if driver.direction_from == N:
            self.north.append(driver)
        elif driver.direction_from == E:
            self.east.append(driver)
        elif driver.direction_from == S:
            self.south.append(driver)
        elif driver.direction_from == W:
            self.west.append(driver)

    def add_driver_stop(self, driver):
        if driver.direction_from == N:
            self.north_stop.append(driver)
        elif driver.direction_from == E:
            self.east_stop.append(driver)
        elif driver.direction_from == S:
            self.south_stop.append(driver)
        elif driver.direction_from == W:
            self.west_stop.append(driver)

    def add_driver_intersection(self, driver):
        self.intersection.append(driver)

    def elapse_driver_time(self):
        for driver in self.north:
            driver.busy_time -= TIME_STEP
        for driver in self.east:
```

```
            driver.busy_time -= TIME_STEP
        for driver in self.south:
            driver.busy_time -= TIME_STEP
        for driver in self.west:
            driver.busy_time -= TIME_STEP
        for driver in self.north_stop:
            driver.busy_time -= TIME_STEP
        for driver in self.west_stop:
            driver.busy_time -= TIME_STEP
        for driver in self.south_stop:
            driver.busy_time -= TIME_STEP
        for driver in self.east_stop:
            driver.busy_time -= TIME_STEP
        for driver in self.intersection:
            driver.busy_time -= TIME_STEP

    def get_next_driver(self):
        min_busy_time = 0
        driver = None
        if len(self.north_stop) != 0 and self.north_stop[0].busy_time < min_busy_time:
            min_busy_time = self.north_stop[0].busy_time
            driver = self.north_stop[0]
        if len(self.east_stop) != 0 and self.east_stop[0].busy_time < min_busy_time:
            min_busy_time = self.east_stop[0].busy_time
            driver = self.east_stop[0]
        if len(self.south_stop) != 0 and self.south_stop[0].busy_time < min_busy_time:
            min_busy_time = self.south_stop[0].busy_time
            driver = self.south_stop[0]
        if len(self.west_stop) != 0 and self.west_stop[0].busy_time < min_busy_time:
            min_busy_time = self.west_stop[0].busy_time
            driver = self.west_stop[0]
        return driver

    def reset_busy_time(self, direction):
        if direction == N:
            for driver in self.north_stop:
                if driver.busy_time < 0:
                    driver.busy_time = 0
        elif direction == E:
            for driver in self.east_stop:
                if driver.busy_time < 0:
                    driver.busy_time = 0
        elif direction == S:
            for driver in self.south_stop:
                if driver.busy_time < 0:
                    driver.busy_time = 0
        elif direction == W:
            for driver in self.west_stop:
                if driver.busy_time < 0:
                    driver.busy_time = 0



class Simulation:

    def __init__(self, total_cars):
        self.num_cars = 0
```

```
        self.total_cars = total_cars
        self.clock = 0

        self.intersection_free = True
        self.driver_queue = DriverQueue()
        self.generate_arrivals()
        self.completed_cars = []

    def run(self):
        while len(self.completed_cars) < self.total_cars:
            #print("The current time is ", self.clock)
            self.execute_events()
            self.driver_queue.elapse_driver_time()
            self.clock += TIME_STEP

    def execute_events(self):
        for driver in self.driver_queue.intersection:
            if driver.busy_time <= 0:
                self.execute_clear(driver)
        for driver in self.driver_queue.north:
            if driver.busy_time <= 0:
                self.execute_arrival(driver)
        for driver in self.driver_queue.west:
            if driver.busy_time <= 0:
                self.execute_arrival(driver)
        for driver in self.driver_queue.south:
            if driver.busy_time <= 0:
                self.execute_arrival(driver)
        for driver in self.driver_queue.east:
            if driver.busy_time <= 0:
                self.execute_arrival(driver)

        driver = self.driver_queue.get_next_driver()
        if driver != None:
            self.execute_stop(driver)

        if self.num_cars < self.total_cars:
            self.generate_arrivals()

    def execute_arrival(self, driver):
        desire = driver.get_from_to()

        if desire[0] == N:
            driver.event = STOP
            if driver.is_human:
                driver.busy_time = HUMAN_MIN_STOP_TIME
            else:
                driver.busy_time = SDC_MIN_STOP_TIME
            self.driver_queue.north.pop(0)
            self.driver_queue.add_driver_stop(driver)
        elif desire[0] == S:
            driver.event = STOP
            if driver.is_human:
                driver.busy_time = HUMAN_MIN_STOP_TIME
            else:
                driver.busy_time = SDC_MIN_STOP_TIME
            self.driver_queue.south.pop(0)
            self.driver_queue.add_driver_stop(driver)
        elif desire[0] == E:
            driver.event = STOP
            if driver.is_human:
```

```
                driver.busy_time = HUMAN_MIN_STOP_TIME
            else:
                driver.busy_time = SDC_MIN_STOP_TIME
            self.driver_queue.east.pop(0)
            self.driver_queue.add_driver_stop(driver)
        elif desire[0] == W:
            driver.event = STOP
            if driver.is_human:
                driver.busy_time = HUMAN_MIN_STOP_TIME
            else:
                driver.busy_time = SDC_MIN_STOP_TIME
            self.driver_queue.west.pop(0)
            self.driver_queue.add_driver_stop(driver)


    def generate_arrivals(self):
        time = self.clock

        r = random.random()
        car_id = self.num_cars
        if r < NORTH_CAR_PROBABILITY and len(self.driver_queue.north) == 0: #From North
            r = random.random()
            if r < TURN_PROBABILITY:
                r = random.random()
                if r < LEFT_TURN_PROBABILITY:
                    direction_to = E
                else:
                    direction_to = W
            else:
                direction_to = S
            r = random.random()
            if r < HUMAN_PROBABILITY:
                is_human = True
            else:
                is_human = False
            #print("Driver ", car_id, " from the North is going to the ", direction_to)
            self.driver_queue.add_driver_arrivals(Driver(car_id, time, ARRIVAL_TIME, N, direction_to, is_human))
            self.num_cars += 1

        r = random.random()
        car_id = self.num_cars
        if r < EAST_CAR_PROBABILITY and len(self.driver_queue.east) == 0: #From East
            r = random.random()
            if r < TURN_PROBABILITY:
                r = random.random()
                if r < LEFT_TURN_PROBABILITY:
                    direction_to = S
                else:
                    direction_to = N
            else:
                direction_to = W
            r = random.random()
            if r < HUMAN_PROBABILITY:
                is_human = True
            else:
                is_human = False
```

```
            #print("Driver ", car_id, " from the East is going to the ", direction_to)
            self.driver_queue.add_driver_arrivals(Driver(car_id, time, ARRIVAL_TIME, E, direction_to, is_human))
            self.num_cars += 1

        r = random.random()
        car_id = self.num_cars
        if r < SOUTH_CAR_PROBABILITY and len(self.driver_queue.south) == 0: #From South
            r = random.random()
            if r < TURN_PROBABILITY:
                r = random.random()
                if r < LEFT_TURN_PROBABILITY:
                    direction_to = W
                else:
                    direction_to = E
            else:
                direction_to = N
            r = random.random()
            if r < HUMAN_PROBABILITY:
                is_human = True
            else:
                is_human = False
            #print("Driver ", car_id, " from the South is going to the ", direction_to)
            self.driver_queue.add_driver_arrivals(Driver(car_id, time, ARRIVAL_TIME, S, direction_to, is_human))
            self.num_cars += 1

        r = random.random()
        car_id = self.num_cars
        if r < WEST_CAR_PROBABILITY and len(self.driver_queue.west) == 0: #From West
            r = random.random()
            if r < TURN_PROBABILITY:
                r = random.random()
                if r < LEFT_TURN_PROBABILITY:
                    direction_to = N
                else:
                    direction_to = S
            else:
                direction_to = E
            r = random.random()
            if r < HUMAN_PROBABILITY:
                is_human = True
            else:
                is_human = False
            #print("Driver ", car_id, "from the West is going to the ", direction_to)
            self.driver_queue.add_driver_arrivals(Driver(car_id, time, ARRIVAL_TIME, W, direction_to, is_human))
            self.num_cars += 1

    def execute_clear(self, driver):
        driver.elapsed_time = self.clock - driver.start_time
        self.completed_cars.append(driver)
        self.driver_queue.intersection.pop(0)
        #print("Driver ", driver.name, " just left the intersection after ", driver.elapsed_time, "seconds")
        if len(self.driver_queue.intersection) == 0:
```

```
            self.intersection_free = True

    def execute_stop(self, driver):
        desire = driver.get_from_to()
        if not self.intersection_free:
            return


        if desire[0] == N and self.intersection_free:
            if desire[1] == S:
                driver.event = CLEAR
                if driver.is_human:
                    driver.busy_time = HUMAN_CLEAR_TIME
                else:
                    driver.busy_time = SDC_CLEAR_TIME
                self.driver_queue.north_stop.pop(0)
                self.driver_queue.reset_busy_time(N)
                self.driver_queue.add_driver_intersection(driver)
                self.intersection_free = False
            else:
                driver.event = CLEAR
                if driver.is_human:
                    driver.busy_time = HUMAN_TURNING_TIME
                else:
                    driver.busy_time = SDC_TURNING_TIME
                self.driver_queue.north_stop.pop(0)
                self.driver_queue.reset_busy_time(N)
                self.driver_queue.add_driver_intersection(driver)
                self.intersection_free = False
        elif desire[0] == S and self.intersection_free:
            if desire[1] == N:
                driver.event = CLEAR
                if driver.is_human:
                    driver.busy_time = HUMAN_CLEAR_TIME
                else:
                    driver.busy_time = SDC_CLEAR_TIME
                self.driver_queue.south_stop.pop(0)
                self.driver_queue.reset_busy_time(S)
                self.driver_queue.add_driver_intersection(driver)
                self.intersection_free = False
            else:
                driver.event = CLEAR
                if driver.is_human:
                    driver.busy_time = HUMAN_TURNING_TIME
                else:
                    driver.busy_time = SDC_TURNING_TIME
                self.driver_queue.south_stop.pop(0)
                self.driver_queue.reset_busy_time(S)
                self.driver_queue.add_driver_intersection(driver)
                self.intersection_free = False
        elif desire[0] == E and self.intersection_free:
            if desire[1] == W:
                driver.event = CLEAR
                if driver.is_human:
                    driver.busy_time = HUMAN_CLEAR_TIME
                else:
                    driver.busy_time = SDC_CLEAR_TIME
                self.driver_queue.east_stop.pop(0)
                self.driver_queue.reset_busy_time(E)
                self.driver_queue.add_driver_intersection(driver)
                self.intersection_free = False
```

```
            else:
                driver.event = CLEAR
                if driver.is_human:
                    driver.busy_time = HUMAN_TURNING_TIME
                else:
                    driver.busy_time = SDC_TURNING_TIME
                self.driver_queue.east_stop.pop(0)
                self.driver_queue.reset_busy_time(E)
                self.driver_queue.add_driver_intersection(driver)
                self.intersection_free = False
        elif desire[0] == W and self.intersection_free:
            if desire[1] == E:
                driver.event = CLEAR
                if driver.is_human:
                    driver.busy_time = HUMAN_CLEAR_TIME
                else:
                    driver.busy_time = SDC_CLEAR_TIME
                self.driver_queue.west_stop.pop(0)
                self.driver_queue.reset_busy_time(W)
                self.driver_queue.add_driver_intersection(driver)
                self.intersection_free = False
            else:
                driver.event = CLEAR
                if driver.is_human:
                    driver.busy_time = HUMAN_TURNING_TIME
                else:
                    driver.busy_time = SDC_TURNING_TIME
                self.driver_queue.west_stop.pop(0)
                self.driver_queue.reset_busy_time(W)
                self.driver_queue.add_driver_intersection(driver)
                self.intersection_free = False

    def output_times(self):
        times = []
        for car in self.completed_cars:
            times.append(car.elapsed_time)
        print(times)
        return sum(times)/len(times)


    def output_to_CSV(self):
        f = open("output.csv", 'w')
        f.write("Name,Type,Start Time,Elapsed Time,Start Direction,End Direction\n")
        for car in self.completed_cars:
            f.write(str(car.name) + "," + str(car.is_human) + "," + str(car.start_time) + "," + str(car.elapsed_time) + "," + str(car.direction_from) + "," + str(car.direction_to) + "\n")
        f.close()



def main():
    sim = Simulation(1000)
    sim.run()
    sim.output_times()
    print(sim.output_times())
    sim.output_to_CSV()

main()
```

Client Request 3: Measure estimated emissions for human-driven and human-driven

```
import random
import matplotlib.pyplot as plt

class Vehicle:
    def __init__(self,sdv_probability):
        r = random.random()
        if r < sdv_probability:
            self.is_self_driving = True
        if r > sdv_probability:
            self.is_self_driving = False

        if self.is_self_driving:
            self.speed = random.randint(50,80)
        else:
            self.speed = random.randint(10,100)

    def get_emissions(self):
        emission = round(0.0019 * self.speed**2 - 0.2506* self.speed + 13.74, 3)

        return emission

#car1 = Vehicle(0.2)
#print(car1.get_emissions())

def average_calculate(k):
    cars = []
    emissions = []

    for i in range(10000):
        cars.append(Vehicle(k))
    for j in cars:
        emissions.append(j.get_emissions())
    average = (sum(emissions)/len(emissions))
    return average

average = []
for i in range(1,11):
    average.append(average_calculate(i/10))

plt.plot([0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1],average, label = "Freq:5")
plt.ylabel('emission[L/100Km]')
plt.xlabel('sdv probability[%]')
plt.show()
```

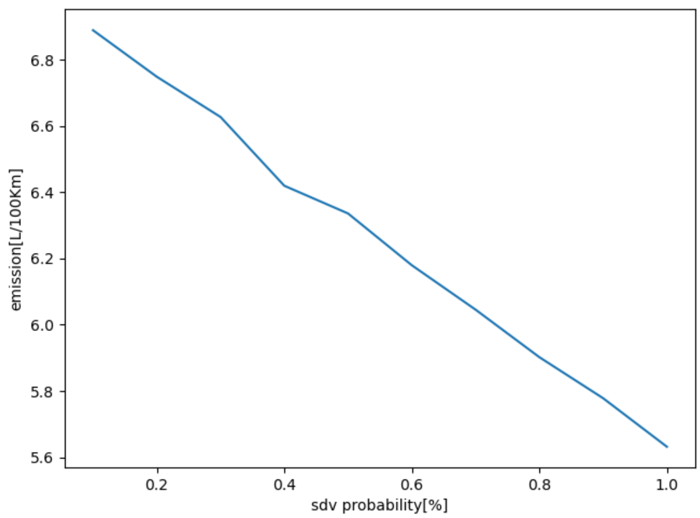


Figure 1 emission vs self-driving probability

Our simulation code:

Curve Class:

```python
def curve_points(start, end, control, resolution=5):
    # If curve is a straight line
    if (start[0] - end[0])*(start[1] - end[1]) == 0:
        return [start, end]

    # If not return a curve
    path = []

    for i in range(resolution+1):
        t = i/resolution
        x = (1-t)**2 * start[0] + 2*(1-t)*t * control[0] + t**2 *end[0]
        y = (1-t)**2 * start[1] + 2*(1-t)*t * control[1] + t**2 *end[1]
        path.append((x, y))

    return path

def curve_road(start, end, control, resolution=15):
    points = curve_points(start, end, control, resolution=resolution)
    return [(points[i-1], points[i]) for i in range(1, len(points))]

TURN_LEFT = 0
TURN_RIGHT = 1
def turn_road(start, end, turn_direction, resolution=15):
    # Get control point
    x = min(start[0], end[0])
    y = min(start[1], end[1])

    if turn_direction == TURN_LEFT:
        control = (
            x - y + start[1],
            y - x + end[0]
```

```
        )
    else:
        control = (
            x - y + end[1],
            y - x + start[0]
        )

    return curve_road(start, end, control, resolution=resolution)
```

Road Class:

```
from scipy.spatial import distance
from collections import deque

class Road:
    def __init__(self, start, end):
        self.start = start
        self.end = end

        self.vehicles = deque()

        self.init_properties()

    def init_properties(self):
        self.length = distance.euclidean(self.start, self.end)
        self.angle_sin = (self.end[1]-self.start[1]) / self.length
        self.angle_cos = (self.end[0]-self.start[0]) / self.length
        # self.angle = np.arctan2(self.end[1]-self.start[1], self.end[0]-self.start[0])
        self.has_traffic_signal = False

    def set_traffic_signal(self, signal, group):
        self.traffic_signal = signal
        self.traffic_signal_group = group
        self.has_traffic_signal = True

    @property
    def traffic_signal_state(self):
        if self.has_traffic_signal:
            i = self.traffic_signal_group
            return self.traffic_signal.current_cycle[i]
        return True

    def update(self, dt):
        n = len(self.vehicles)

        if n > 0:
            # Update first vehicle
            self.vehicles[0].update(None, dt)
            # Update other vehicles
            for i in range(1, n):
                lead = self.vehicles[i-1]
                self.vehicles[i].update(lead, dt)

             # Check for traffic signal
            if self.traffic_signal_state:
                # If traffic signal is green or doesn't exist
                # Then let vehicles pass
                self.vehicles[0].unstop()
```

```
                for vehicle in self.vehicles:
                    vehicle.unslow()
            else:
                # If traffic signal is red
                if self.vehicles[0].x >= self.length - self.traffic_signal.slow_distance:
                    # Slow vehicles in slowing zone
                    self.vehicles[0].slow(self.traffic_signal.slow_factor*self.vehicles[0]._v_max)
                if self.vehicles[0].x >= self.length - self.traffic_signal.stop_distance and\
                   self.vehicles[0].x <= self.length - self.traffic_signal.stop_distance / 2:
                    # Stop vehicles in the stop zone
                    self.vehicles[0].stop()
```

Simulation Class:

```
from .road import Road
from copy import deepcopy
from .vehicle_generator import VehicleGenerator
from .traffic_signal import TrafficSignal

class Simulation:
    def __init__(self, config={}):
        # Set default configuration
        self.set_default_config()

        # Update configuration
        for attr, val in config.items():
            setattr(self, attr, val)

    def set_default_config(self):
        self.t = 0.0            # Time keeping
        self.frame_count = 0    # Frame count keeping
        self.dt = 1/60          # Simulation time step
        self.roads = []         # Array to store roads
        self.generators = []
        self.traffic_signals = []

    def create_road(self, start, end):
        road = Road(start, end)
        self.roads.append(road)
        return road

    def create_roads(self, road_list):
        for road in road_list:
            self.create_road(*road)

    def create_gen(self, config={}):
        gen = VehicleGenerator(self, config)
        self.generators.append(gen)
        return gen
```

```python

    def create_signal(self, roads, config={}):
        roads = [[self.roads[i] for i in road_group] for road_group in roads]
        sig = TrafficSignal(roads, config)
        self.traffic_signals.append(sig)
        return sig

    def update(self):
        # Update every road
        for road in self.roads:
            road.update(self.dt)

        # Add vehicles
        for gen in self.generators:
            gen.update()

        for signal in self.traffic_signals:
            signal.update(self)

        # Check roads for out of bounds vehicle
        for road in self.roads:
            # If road has no vehicles, continue
            if len(road.vehicles) == 0: continue
            # If not
            vehicle = road.vehicles[0]
            # If first vehicle is out of road bounds
            if vehicle.x >= road.length:
                # If vehicle has a next road
                if vehicle.current_road_index + 1 < len(vehicle.path):
                    # Update current road to next road
                    vehicle.current_road_index += 1
                    # Create a copy and reset some vehicle properties
                    new_vehicle = deepcopy(vehicle)
                    new_vehicle.x = 0
                    # Add it to the next road
                    next_road_index = vehicle.path[vehicle.current_road_index]
                    self.roads[next_road_index].vehicles.append(new_vehicle)
                # In all cases, remove it from its road
                road.vehicles.popleft()
        # Increment time
        self.t += self.dt
        self.frame_count += 1


    def run(self, steps):
        for _ in range(steps):
            self.update()
```

Traffic Signal:

```python
class TrafficSignal:
    def __init__(self, roads, config={}):
        # Initialize roads
        self.roads = roads
        # Set default configuration
        self.set_default_config()
        # Update configuration
        for attr, val in config.items():
```

```
                setattr(self, attr, val)
        # Calculate properties
        self.init_properties()

    def set_default_config(self):
        self.cycle = [(False, True), (True, False)]
        self.slow_distance = 50
        self.slow_factor = 0.4
        self.stop_distance = 15

        self.current_cycle_index = 0

        self.last_t = 0

    def init_properties(self):
        for i in range(len(self.roads)):
            for road in self.roads[i]:
                road.set_traffic_signal(self, i)

    @property
    def current_cycle(self):
        return self.cycle[self.current_cycle_index]

    def update(self, sim):
        cycle_length = 30
        k = (sim.t // cycle_length) % 2
        self.current_cycle_index = int(k)
```

Vehicle Generator class:

```
from .vehicle import Vehicle
from numpy.random import randint

class VehicleGenerator:
    def __init__(self, sim, config={}):
        self.sim = sim

        # Set default configurations
        self.set_default_config()

        # Update configurations
        for attr, val in config.items():
            setattr(self, attr, val)

        # Calculate properties
        self.init_properties()

    def set_default_config(self):
        """Set default configuration"""
        self.vehicle_rate = 20
        self.vehicles = [
            (1, {})
        ]
        self.last_added_time = 0

```

```
    def init_properties(self):
        self.upcoming_vehicle = self.generate_vehicle()

    def generate_vehicle(self):
        """Returns a random vehicle from self.vehicles with random proportions"""
        total = sum(pair[0] for pair in self.vehicles)
        r = randint(1, total+1)
        for (weight, config) in self.vehicles:
            r -= weight
            if r <= 0:
                return Vehicle(config)

    def update(self):
        """Add vehicles"""
        if self.sim.t - self.last_added_time >= 60 / self.vehicle_rate:
            # If time elasped after last added vehicle is
            # greater than vehicle_period; generate a vehicle
            road = self.sim.roads[self.upcoming_vehicle.path[0]]
            if len(road.vehicles) == 0\
               or road.vehicles[-1].x > self.upcoming_vehicle.s0 + self.upcoming_vehicle.l:
                # If there is space for the generated vehicle; add it
                self.upcoming_vehicle.time_added = self.sim.t
                road.vehicles.append(self.upcoming_vehicle)
                # Reset last_added_time and upcoming_vehicle
                self.last_added_time = self.sim.t
            self.upcoming_vehicle = self.generate_vehicle()
```

Vehicle Class:

```
import numpy as np

class Vehicle:
    def __init__(self, config={}):
        # Set default configuration
        self.set_default_config()

        # Update configuration
        for attr, val in config.items():
            setattr(self, attr, val)

        # Calculate properties
        self.init_properties()

    def set_default_config(self):
        self.l = 4
        self.s0 = 4
        self.T = 1
        self.v_max = 16.6
        self.a_max = 1.44
        self.b_max = 4.61

        self.path = []
        self.current_road_index = 0
```

```
        self.x = 0
        self.v = self.v_max
        self.a = 0
        self.stopped = False

    def init_properties(self):
        self.sqrt_ab = 2*np.sqrt(self.a_max*self.b_max)
        self._v_max = self.v_max

    def update(self, lead, dt):
        # Update position and velocity
        if self.v + self.a*dt < 0:
            self.x -= 1/2*self.v*self.v/self.a
            self.v = 0
        else:
            self.v += self.a*dt
            self.x += self.v*dt + self.a*dt*dt/2

        # Update acceleration
        alpha = 0
        if lead:
            delta_x = lead.x - self.x - lead.l
            delta_v = self.v - lead.v

            alpha = (self.s0 + max(0, self.T*self.v + delta_v*self.v/self.sqrt_ab)) / delta_x

        self.a = self.a_max * (1-(self.v/self.v_max)**4 - alpha**2)

        if self.stopped:
            self.a = -self.b_max*self.v/self.v_max

    def stop(self):
        self.stopped = True

    def unstop(self):
        self.stopped = False

    def slow(self, v):
        self.v_max = v

    def unslow(self):
        self.v_max = self._v_max
```

Window Class:

```
import pygame
from pygame import gfxdraw
import numpy as np

class Window:
    def __init__(self, sim, config={}):
        # Simulation to draw
```

```
        self.sim = sim

        # Set default configurations
        self.set_default_config()

        # Update configurations
        for attr, val in config.items():
            setattr(self, attr, val)

    def set_default_config(self):
        """Set default configuration"""
        self.width = 1400
        self.height = 900
        self.bg_color = (250, 250, 250)

        self.fps = 60
        self.zoom = 5
        self.offset = (0, 0)

        self.mouse_last = (0, 0)
        self.mouse_down = False


    def loop(self, loop=None):
        """Shows a window visualizing the simulation and runs the loop function."""

        # Create a pygame window
        self.screen = pygame.display.set_mode((self.width, self.height))
        pygame.display.flip()

        # Fixed fps
        clock = pygame.time.Clock()

        # To draw text
        pygame.font.init()
        self.text_font = pygame.font.SysFont('Lucida Console', 16)

        # Draw loop
        running = True
        while running:
            # Update simulation
            if loop: loop(self.sim)

            # Draw simulation
            self.draw()

            # Update window
            pygame.display.update()
            clock.tick(self.fps)

            # Handle all events
            for event in pygame.event.get():
                # Quit program if window is closed
                if event.type == pygame.QUIT:
                    running = False
                # Handle mouse events
                elif event.type == pygame.MOUSEBUTTONDOWN:
                    # If mouse button down
                    if event.button == 1:
                        # Left click
```

```
                        x, y = pygame.mouse.get_pos()
                        x0, y0 = self.offset
                        self.mouse_last = (x-x0*self.zoom, y-y0*self.zoom)
                        self.mouse_down = True
                    if event.button == 4:
                        # Mouse wheel up
                        self.zoom *=  (self.zoom**2+self.zoom/4+1) / (self.zoom**2+1)
                    if event.button == 5:
                        # Mouse wheel down
                        self.zoom *= (self.zoom**2+1) / (self.zoom**2+self.zoom/4+1)
                elif event.type == pygame.MOUSEMOTION:
                    # Drag content
                    if self.mouse_down:
                        x1, y1 = self.mouse_last
                        x2, y2 = pygame.mouse.get_pos()
                        self.offset = ((x2-x1)/self.zoom, (y2-y1)/self.zoom)
                elif event.type == pygame.MOUSEBUTTONUP:
                    self.mouse_down = False

    def run(self, steps_per_update=1):
        """Runs the simulation by updating in every loop."""
        def loop(sim):
            sim.run(steps_per_update)
        self.loop(loop)

    def convert(self, x, y=None):
        """Converts simulation coordinates to screen coordinates"""
        if isinstance(x, list):
            return [self.convert(e[0], e[1]) for e in x]
        if isinstance(x, tuple):
            return self.convert(*x)
        return (
            int(self.width/2 + (x + self.offset[0])*self.zoom),
            int(self.height/2 + (y + self.offset[1])*self.zoom)
        )

    def inverse_convert(self, x, y=None):
        """Converts screen coordinates to simulation coordinates"""
        if isinstance(x, list):
            return [self.convert(e[0], e[1]) for e in x]
        if isinstance(x, tuple):
            return self.convert(*x)
        return (
            int(-self.offset[0] + (x - self.width/2)/self.zoom),
            int(-self.offset[1] + (y - self.height/2)/self.zoom)
        )


    def background(self, r, g, b):
        """Fills screen with one color."""
        self.screen.fill((r, g, b))

    def line(self, start_pos, end_pos, color):
        """Draws a line."""
        gfxdraw.line(
            self.screen,
            *start_pos,
            *end_pos,
            color
```

```
            )

    def rect(self, pos, size, color):
        """Draws a rectangle."""
        gfxdraw.rectangle(self.screen, (*pos, *size), color)

    def box(self, pos, size, color):
        """Draws a rectangle."""
        gfxdraw.box(self.screen, (*pos, *size), color)

    def circle(self, pos, radius, color, filled=True):
        gfxdraw.aacircle(self.screen, *pos, radius, color)
        if filled:
            gfxdraw.filled_circle(self.screen, *pos, radius, color)



    def polygon(self, vertices, color, filled=True):
        gfxdraw.aapolygon(self.screen, vertices, color)
        if filled:
            gfxdraw.filled_polygon(self.screen, vertices, color)

    def rotated_box(self, pos, size, angle=None, cos=None, sin=None, centered=True, color=(0, 0, 255), filled=True):
        """Draws a rectangle center at *pos* with size *size* rotated anti-clockwise by *angle*."""
        x, y = pos
        l, h = size

        if angle:
            cos, sin = np.cos(angle), np.sin(angle)

        vertex = lambda e1, e2: (
            x + (e1*l*cos + e2*h*sin)/2,
            y + (e1*l*sin - e2*h*cos)/2
        )

        if centered:
            vertices = self.convert(
                [vertex(*e) for e in [(-1,-1), (-1, 1), (1,1), (1,-1)]]
            )
        else:
            vertices = self.convert(
                [vertex(*e) for e in [(0,-1), (0, 1), (2,1), (2,-1)]]
            )

        self.polygon(vertices, color, filled=filled)

    def rotated_rect(self, pos, size, angle=None, cos=None, sin=None, centered=True, color=(0, 0, 255)):
        self.rotated_box(pos, size, angle=angle, cos=cos, sin=sin, centered=centered, color=color, filled=False)

    def arrow(self, pos, size, angle=None, cos=None, sin=None, color=(150, 150, 190)):
        if angle:
            cos, sin = np.cos(angle), np.sin(angle)

        self.rotated_box(
            pos,
            size,
```

```python
                cos=(cos - sin) / np.sqrt(2),
                sin=(cos + sin) / np.sqrt(2),
                color=color,
                centered=False
            )

            self.rotated_box(
                pos,
                size,
                cos=(cos + sin) / np.sqrt(2),
                sin=(sin - cos) / np.sqrt(2),
                color=color,
                centered=False
            )


        def draw_axes(self, color=(100, 100, 100)):
            x_start, y_start = self.inverse_convert(0, 0)
            x_end, y_end = self.inverse_convert(self.width, self.height)
            self.line(
                self.convert((0, y_start)),
                self.convert((0, y_end)),
                color
            )
            self.line(
                self.convert((x_start, 0)),
                self.convert((x_end, 0)),
                color
            )

        def draw_grid(self, unit=50, color=(150,150,150)):
            x_start, y_start = self.inverse_convert(0, 0)
            x_end, y_end = self.inverse_convert(self.width, self.height)

            n_x = int(x_start / unit)
            n_y = int(y_start / unit)
            m_x = int(x_end / unit)+1
            m_y = int(y_end / unit)+1

            for i in range(n_x, m_x):
                self.line(
                    self.convert((unit*i, y_start)),
                    self.convert((unit*i, y_end)),
                    color
                )
            for i in range(n_y, m_y):
                self.line(
                    self.convert((x_start, unit*i)),
                    self.convert((x_end, unit*i)),
                    color
                )

        def draw_roads(self):
            for road in self.sim.roads:
                # Draw road background
                self.rotated_box(
                    road.start,
                    (road.length, 3.7),
                    cos=road.angle_cos,
                    sin=road.angle_sin,
                    color=(180, 180, 220),
```

```
                centered=False
            )
            # Draw road lines
            # self.rotated_box(
            #     road.start,
            #     (road.length, 0.25),
            #     cos=road.angle_cos,
            #     sin=road.angle_sin,
            #     color=(0, 0, 0),
            #     centered=False
            # )

            # Draw road arrow
            if road.length > 5:
                for i in np.arange(-0.5*road.length, 0.5*road.length, 10):
                    pos = (
                        road.start[0] + (road.length/2 + i + 3) * road.angle_cos,
                        road.start[1] + (road.length/2 + i + 3) * road.angle_sin
                    )

                    self.arrow(
                        pos,
                        (-1.25, 0.2),
                        cos=road.angle_cos,
                        sin=road.angle_sin
                    )



            # TODO: Draw road arrow

    def draw_vehicle(self, vehicle, road):
        l, h = vehicle.l,  2
        sin, cos = road.angle_sin, road.angle_cos

        x = road.start[0] + cos * vehicle.x
        y = road.start[1] + sin * vehicle.x

        self.rotated_box((x, y), (l, h), cos=cos, sin=sin, centered=True)

    def draw_vehicles(self):
        for road in self.sim.roads:
            # Draw vehicles
            for vehicle in road.vehicles:
                self.draw_vehicle(vehicle, road)

    def draw_signals(self):
        for signal in self.sim.traffic_signals:
            for i in range(len(signal.roads)):
                color = (0, 255, 0) if signal.current_cycle[i] else (255, 0, 0)

                for road in signal.roads[i]:
                    a = 0
                    position = (
                        (1-a)*road.end[0] + a*road.start[0],
                        (1-a)*road.end[1] + a*road.start[1]
                    )
                    self.rotated_box(
                        position,
```

```
                        (1, 3),
                        cos=road.angle_cos, sin=road.angle_sin,
                        color=color)

    def draw_status(self):
        text_fps = self.text_font.render(f't={self.sim.t:.5}', False, (0, 0, 0))
        text_frc = self.text_font.render(f'n={self.sim.frame_count}', False, (0, 0, 0))

        self.screen.blit(text_fps, (0, 0))
        self.screen.blit(text_frc, (100, 0))


    def draw(self):
        # Fill background
        self.background(*self.bg_color)

        # Major and minor grid and axes
        # self.draw_grid(10, (220,220,220))
        # self.draw_grid(100, (200,200,200))
        # self.draw_axes()

        self.draw_roads()
        self.draw_vehicles()
        self.draw_signals()

        # Draw status info
        self.draw_status()

```

Project code:

```
import numpy as np
from trafficSimulator import *

# Create simulation
sim = Simulation()

# Add multiple roads
sim.create_roads([
  #Positive Y Part
    #Left North to South
    ((-5, -100), (-5, -10)),    #0
    #Middle North to South
    ((0, -100), (0, -5)),       #1
    #Right South to North
    ((5, -10), (5, -100)),      #2

  #Negative Y Part
    #Left North to South
    ((-5, 10),(-5, 100)),       #3
    #Middle South to North
    ((0, 100), (0, 5)),         #4
    #Right South to North
```

```
    ((5, 100), (5, 10)),        #5

  #Positive X part
    #Top East to West
    ((100, -5), (10, -5)),      #6
    #Middle East to West
    ((100, 0), (5, 0)),         #7
    #Down West to East
    ((10, 5), (100, 5)),        #8

  #Negative X part
    #Top East to West
    ((-10, -5), (-100, -5)),    #9
    #Middle West to East
    ((-100, 0), (-5, 0)),       #10
    #Down West to East
    ((-100, 5), (-10, 5)),      #11


  #Intersection
   #Straight
    #Negative Y to Positive Y
    ((5, 10), (5, -10)),        #12
    #Postive Y to Negative Y
    ((-5, -10), (-5, 10)),      #13
    #Negative X to Positive X
    ((-10, 5), (10, 5)),        #14
    #Positive X to Negative X
    ((10, -5), (-10, -5)),      #15

   #Turns
    #Negative Y to Positive X Right turn
    *turn_road((5, 10), (10, 5), TURN_RIGHT, 20),       #16
    #Negative Y to Negative X Left turn
    *turn_road((0, 5), (-10, -5), TURN_LEFT, 20),       #16 + n

    #Positive X to Postive Y Right turn
    *turn_road((10, -5), (5, -10), TURN_RIGHT, 20),     #16 + 2n
    #Positive X to Negative Y Left turn
    *turn_road((5, 0), (-5, 10), TURN_LEFT, 20),        #16 + 3n

    #Postive Y to Negative X Right turn
    *turn_road((-5, -10), (-10, -5), TURN_RIGHT, 20),   #16 + 4n
    #Positive Y to Postivie X Left turn
    *turn_road((0, -5), (10, 5), TURN_LEFT, 20),        #16 + 5n

    #Negative X to Negative Y Right turn
    *turn_road((-10, 5), (-5, 10), TURN_RIGHT, 20),     #16 + 6n
    #Negative X to Postivie Y Left turn
    *turn_road((-5, 0), (5, -10), TURN_LEFT, 20),       #16 + 7n

])

def road(a): return range(a, a+20)

sim.create_gen({
'vehicle_rate': 40,
'vehicles':[
    [3, {'path': [5, 12, 2]}],
    [1, {'path': [5, *road(16), 8]}],
    [1, {'path': [4, *road(16 + 20), 9]}],
```

```

        [3, {'path': [6, 15, 9]}],
        [1, {'path': [6, *road(16 + 2*20), 2]}],
        [1, {'path': [7, *road(16 + 3*20), 3]}],

        [3, {'path': [0, 13, 3]}],
        [1, {'path': [0, *road(16 + 4*20), 9]}],
        [1, {'path': [1, *road(16 + 5*20), 8]}],

        [3, {'path': [11, 14, 8]}],
        [1, {'path': [11, *road(16 + 6*20), 3]}],
        [1, {'path': [10, *road(16 + 7*20), 2]}],



    ]})

    # Traffic signal
    sim.create_signal([[0, 5, 1, 4,], [6, 11, 7, 10,]])


    # Start simulation
    win = Window(sim)
    win.run(steps_per_update=5)
```

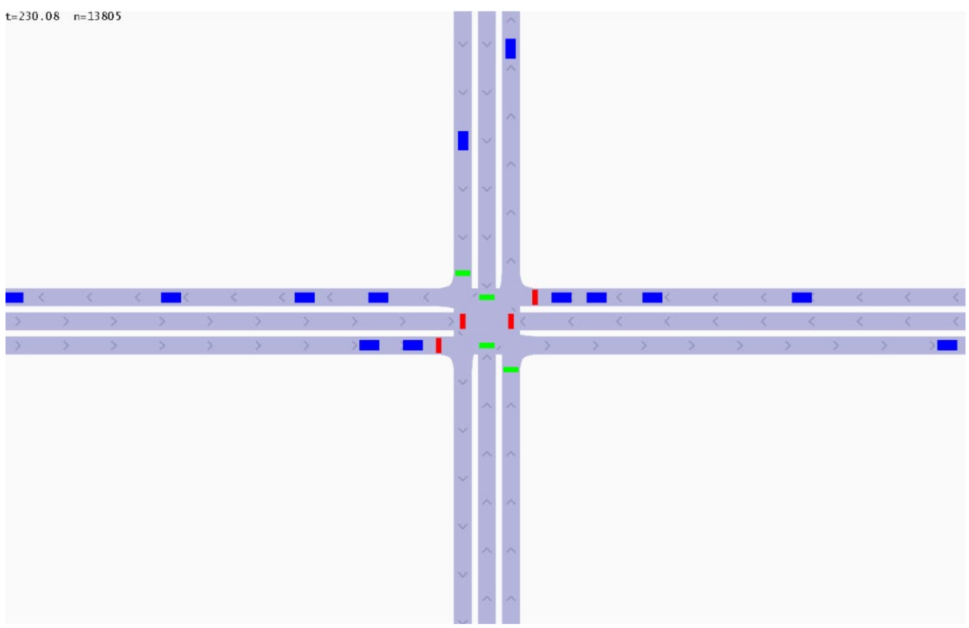


Figure 2 Simulation Output

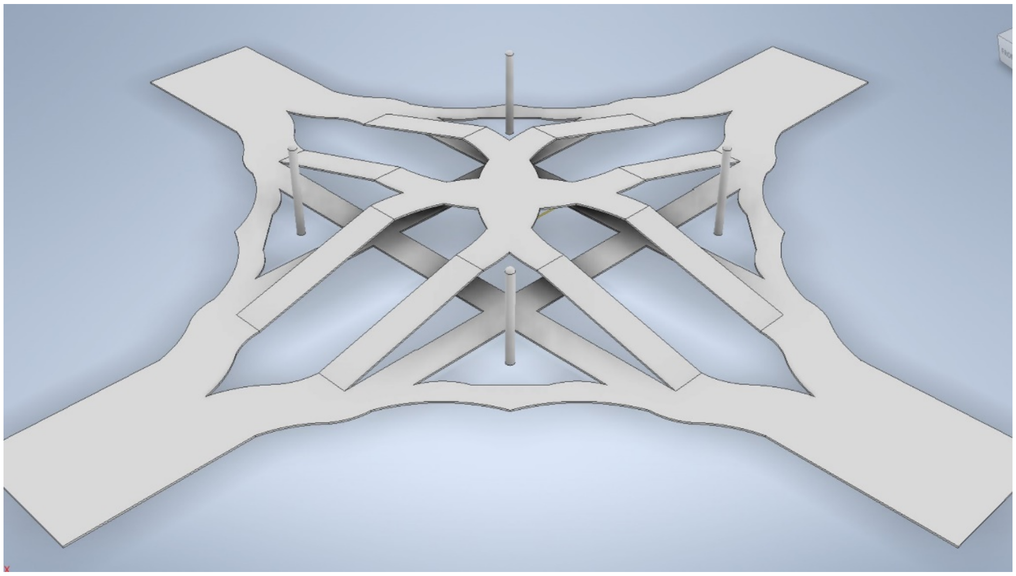


Figure 3 Intersection Modelling

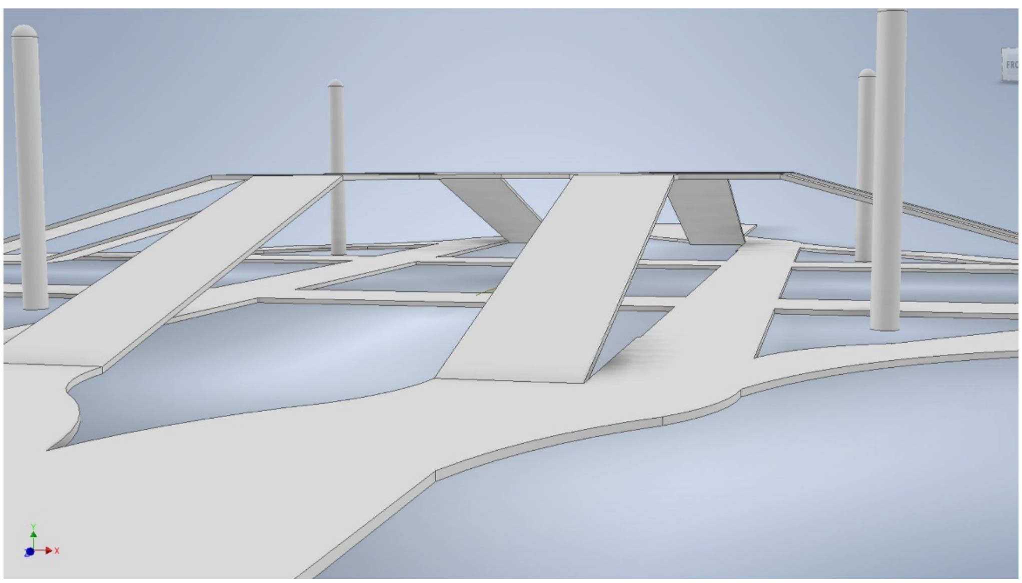
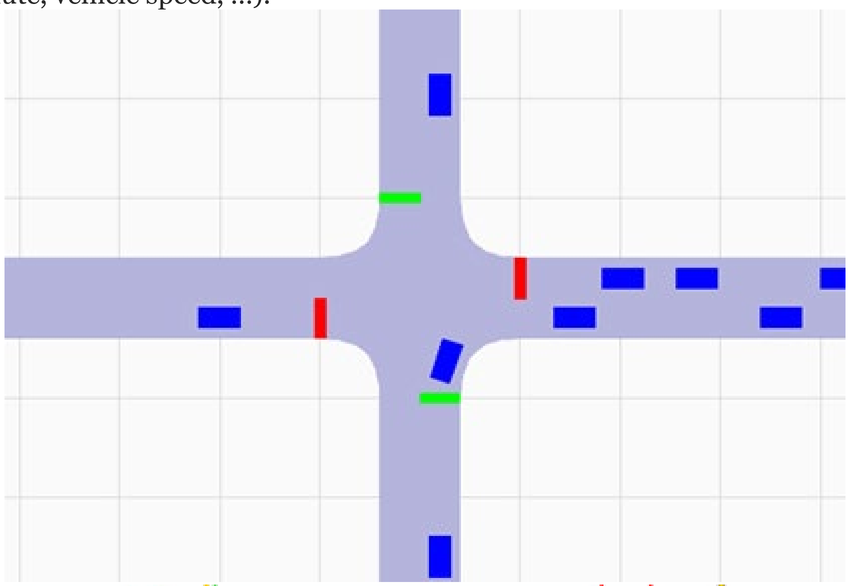


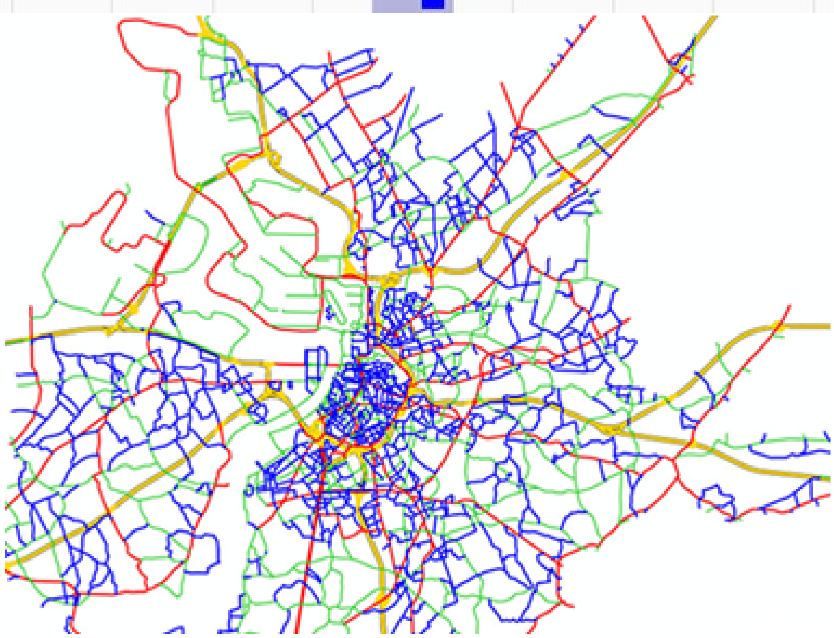
Figure 4 Intersection Modelling 2

The math related to the simulations:

Modeling

To analyze and optimize traffic systems, we first have to model a traffic system mathematically. Such a model should realistically represent traffic flow based on input parameters (road network geometry, vehicles per minute, vehicle speed, ...).





Traffic system models are generally classified into three categories, depending on what level they are operating on:

- **Microscopic models:** represent every vehicle separately and attempt to replicate driver behavior.
- **Macroscopic models:** describe the movement of vehicles as a whole in terms of traffic density (vehicle per km) and traffic flow (vehicles per minute). They are usually analogous to fluid flow.
- **Mesoscopic models:** are hybrid models that combine the features of both microscopic and macroscopic models; They model flow as "packets" of vehicles.

In this article, I will use a microscopic model.

Microscopic Models

A microscopic driver model describes the behavior of a single driver/vehicle. As a consequence, it must be a multi-agent system, that is, every vehicle operates on its own using input from its environment.

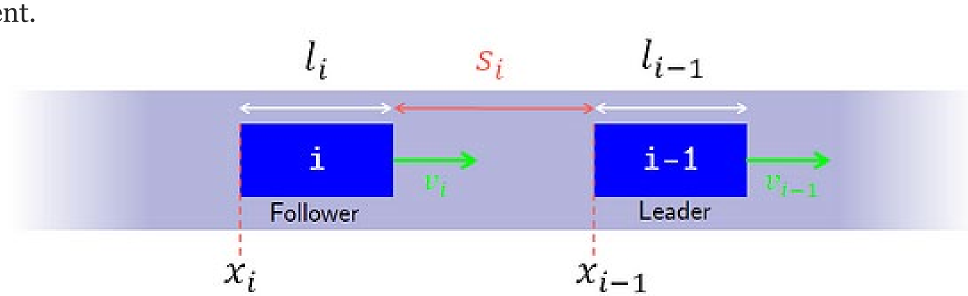


Image by Author.

In microscopic models, every vehicle is numbered a number *i*. The *i*-th vehicle follows the (*i*-*1*)-th vehicle. For the *i*-th vehicle, we will denote by $\mathbf{x}_i$ its position along the road, $\mathbf{v}_i$ its speed, and $\mathbf{l}_i$ its length. And this is true for every vehicle.

$$s_i = x_i - x_{i-1} - l_i$$
$$\Delta v_i = v_i - v_{i-1}$$

We will denote by $\mathbf{s_i}$ the bumper-to-bumper distance and $\mathbf{\Delta v_i}$ the velocity difference between the *i*-th vehicle and the vehicle in front of it (vehicle number *i-1*).

Intelligent Driver Model (IDM)

In 2000, Treiber, Hennecke et Helbing developed a model known as the Intelligent Driver Model. It describes the acceleration of the *i*-th vehicle as a function of its variables and those of the vehicle in front of it. The dynamics equation is defined as:

$$\frac{dv_i}{dt} = a_i\left(1 - \left(\frac{v_i}{v_{0,i}}\right)^{\delta} - \left(\frac{s^*(v_i, \Delta v_i)}{s_i}\right)^2\right)$$

$$s^*(v_i, \Delta v_i) = s_{0,i} + v_i T_i + \frac{v_i \Delta v_i}{\sqrt{2 a_i b_i}}$$

Before I explain the intuition behind this model, I should explain what some symbols represent.

We have talked about $\mathbf{s_i}$, $\mathbf{v_i}$, and $\Delta\mathbf{v_i}$. The other parameters are:

- $\mathbf{s_{0i}}$: is the minimum desired distance between the vehicle *i* and *i-1*.
- $\mathbf{v_{0i}}$ **:** is the maximum desired speed of the vehicle *i*.
- $\boldsymbol{\delta}$ **:** is the acceleration exponent and it controls the “smoothness” of the acceleration.

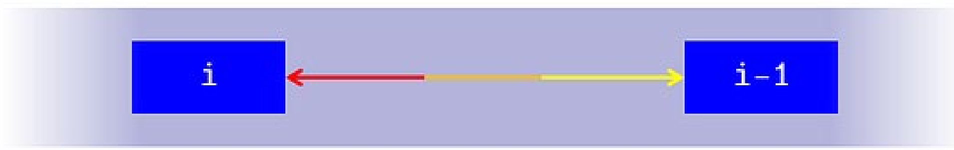
- **T_i :** is the reaction time of the *i*-th vehicle's driver.

- **a_i :** is the maximum acceleration for the vehicle *i*.

- **b_i :** is the comfortable deceleration for the vehicle *i*.

- **s*** : is the actual desired distance between the vehicle *i* and *i-1*.

First, we will look at **s***, which is a distance and it is comprised of three terms.



$$s^*(v_i, \Delta v_i) = s_{0,i} + v_i T_i + \frac{v_i \Delta v_i}{\sqrt{2a_i b_i}}$$

Image by Author.

- **s_{0_i} :** as said before, is the minimum desired distance.

- **$v_i T_i$:** is the reaction time safety distance. It is the distance the vehicle travels before the driver reacts (brakes).

  Since speed is distance over time, distance is speed times time.

$$v = \frac{d}{T} \implies d = vT$$

- **$(v_i \Delta v_i)/\sqrt{(2a_i b_i)}$:** this is a bit more complicated term. It's a speed-difference-based safety distance. It represents the distance it will take the vehicle to slow down (without hitting the vehicle in front), without braking too much (the deceleration should be less than **b_i**).

How The Intelligent Driver Model Works

Vehicles are assumed to be moving along a straight path and assumed to obey the following equation:

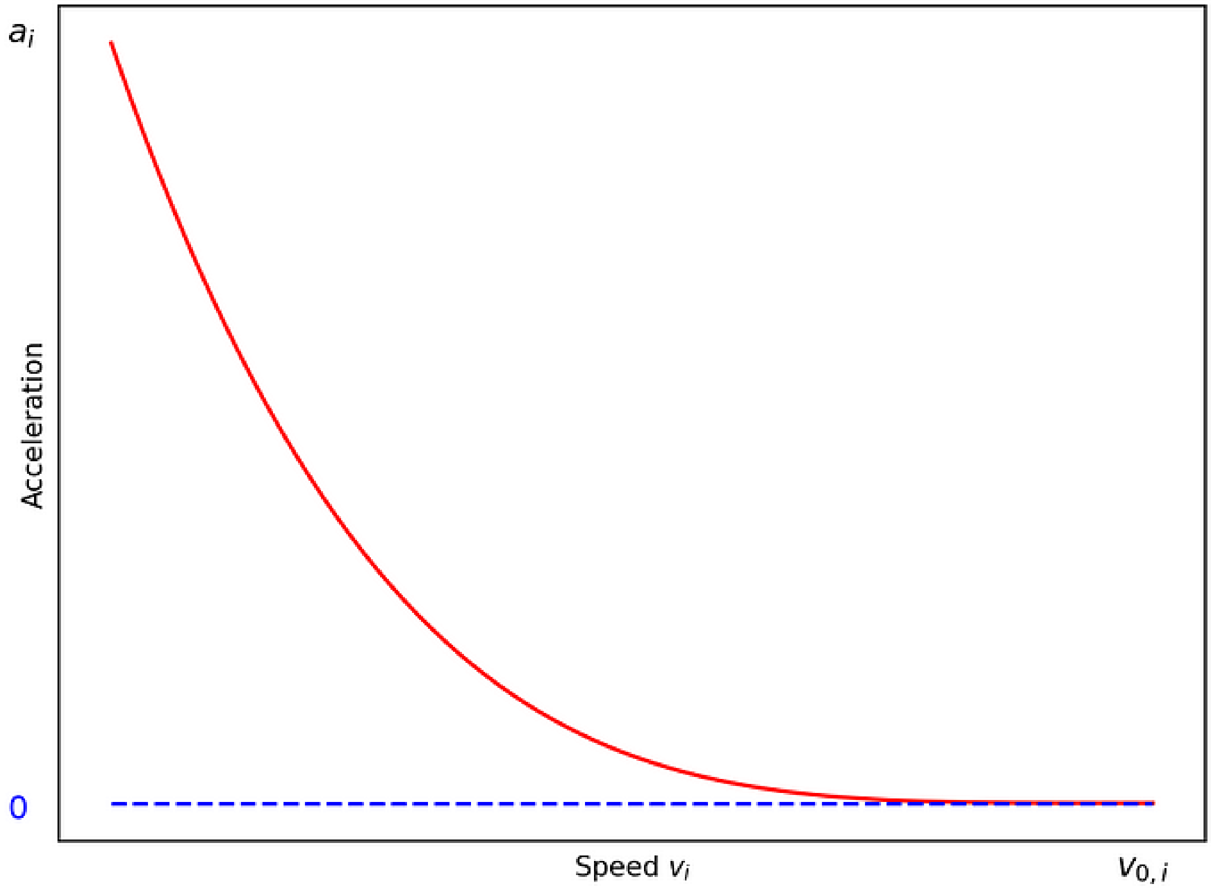
$$\frac{dv_i}{dt} = a_{free\ road} + a_{interaction}$$

$$\begin{cases} a_{free\ road} = a_i\left(1 - \left(\frac{v_i}{v_{0,i}}\right)^{\delta}\right) \\ a_{interaction} = -a_i\left(\frac{s^*(v_i, \Delta v_i)}{s_i}\right)^2 \end{cases}$$

To get a better understanding of the equation, we can divide its terms in two. We have a **free road acceleration** and an **interaction acceleration**.

$$a_{free\ road} = a_i\left(1 - \left(\frac{v_i}{v_{0,i}}\right)^{\delta}\right)$$

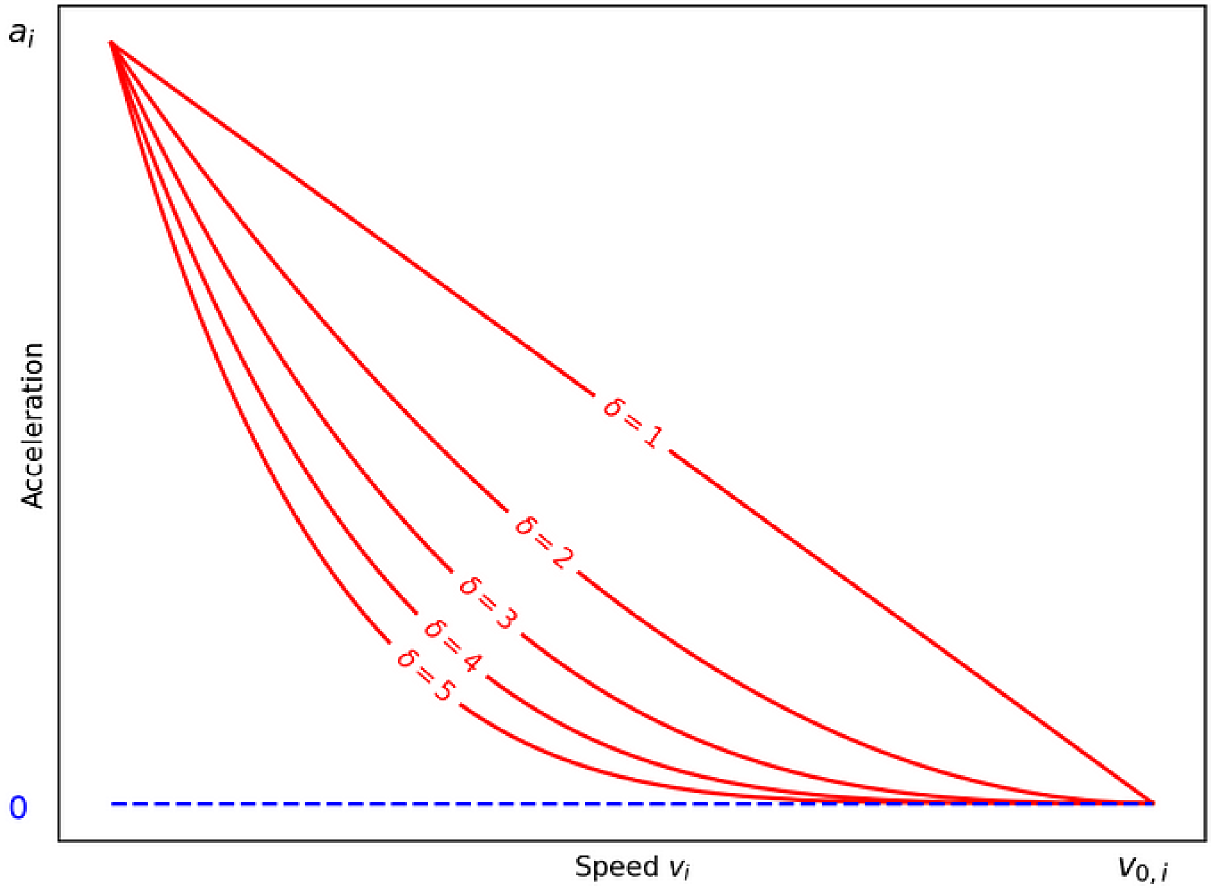
The **free road acceleration** is the acceleration on a free road, that is, an empty road with no vehicles ahead. If we plot the acceleration as a function of speed $\mathbf{v_i}$ we get**:**



Acceleration as a function of speed. Image by Author.

We notice that when the vehicle is stationary (**v_i=0**) the acceleration is maximal. When the vehicle speed approaches the maximum speed **$v0_i$** the acceleration becomes 0. This indicates that the **free road acceleration** will accelerate the vehicle to the maximum speed.

If we plot the v-a diagram for different values of **δ,** we notice that it controls how quickly the driver decelerates when approaching the maximum speed. Which in turn controls the smoothness of the acceleration/deceleration/



Acceleration as a function of speed. Image by Author.

$$a_{interaction} = -a_i \left(\frac{s^*(v_i, \Delta v_i)}{s_i} \right)^2 = -a_i \left(\frac{s_{0,i} + v_i T_i}{s_i} + \frac{v_i \Delta v_i}{2 s_i \sqrt{a_i b_i}} \right)^2$$

The **interaction acceleration** is linked to the interaction with the vehicle in front. To better understand how it works, let's consider the following situations:

- **On a free road (s_i >> s^*):**
  When the vehicle in front is far away, that is the distance **s_i** is dominates the desired distance **s^***, the interaction acceleration is almost 0.
  This means that vehicle will be governed by the free road acceleration.

$$\frac{dv_i}{dt} \approx a_{free\,road} = a_i\left(1 - \left(\frac{v_i}{v_{0,i}}\right)^{\delta}\right) \quad ; \quad \left(\frac{s^*(v_i, \Delta v_i)}{s_i}\right)^2 \approx 0$$

- **At high approach rates (Δv_i):**

  When the speed difference is high, the interaction acceleration tries to compensate for that by braking or slowing down using the $(v_i \Delta v_i)^2$ term in the numerator but too hard. This is achieved through the denominator $4b_i s_i^2$. (I honestly have no idea how does it limit the deceleration to exactly b_i).

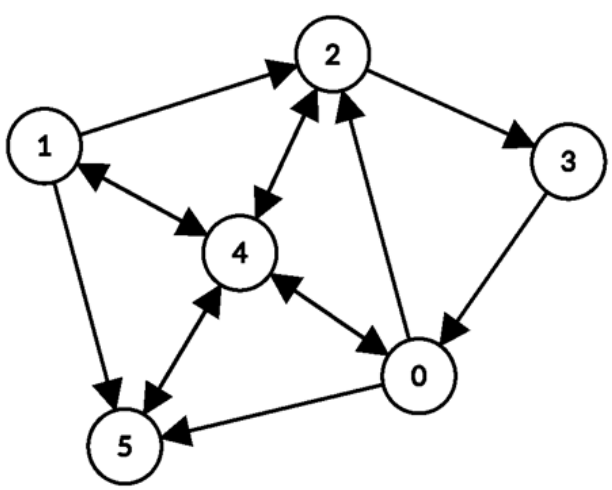
$$a_{interaction} = -a_i\left(\frac{s_{0,i} + v_i T_i}{s_i} + \frac{v_i \Delta v_i}{2s_i\sqrt{a_i b_i}}\right)^2 \approx -\frac{(v_i \Delta v_i)^2}{4b_i {s_i}^2}$$

- **At small distance difference ($s_i << 1$ and $\Delta v_i \approx 0$):**

  The acceleration becomes a simple repulsive force.

$$a_{interaction} = -a_i\left(\frac{s_{0,i} + v_i T_i}{s_i} + \frac{v_i \Delta v_i}{2s_i\sqrt{a_i b_i}}\right)^2 \approx -a_i \frac{\left(s_{0,i} + v_i T_i\right)^2}{{s_i}^2}$$

Traffic Road Network Model



$$V = \{0, 1, 2, 3, 4, 5\}$$

$$E = \begin{Bmatrix} (0,2),(0,4),(0,5), \\ (1,2),(1,4),(1,5), \\ (2,3),(2,4),(3,0), \\ (4,0),(4,1),(4,2), \\ (4,5),(5,4) \end{Bmatrix}$$

Example of a directed graph. Diagram (left) Sets (right)

We need to model a network of roads. To do this, we will use a **directed graph G=(V, E)**. Where:

- **V** is the set of vertices (or nodes).
- **E** is the set of edges that represent roads.

Every vehicle is going to have a path consisting of multiple roads (edges). We will apply the Intelligent Driver Model for vehicles in the same road (same edge). When a vehicle reaches the end of the road, we remove it from that road and append it to its next road.

In the simulation we won’t keep a set (array) of nodes, instead, every road is going to be explicitly defined by the values of its start and end nodes.

Stochastic Vehicle Generator

In order to add vehicles to our simulation we have two options:

- Add every vehicle manually to the simulation by creating a new `Vehicle` class instance and adding it to the list of vehicles.
- Add vehicles stochastically according to pre-defined probabilities.

For the second option, we have to define a stochastic vehicle generator.

A stochastic vehicle generator is defined by two constraints:

- **Vehicle generation rate (τ):** (in vehicles per minute) describes how many vehicles should be added to the simulation, on average, per minute.
- **Vehicle configuration list(L):** A list of tuples containing the configuration and probability of vehicles.
  $L = [(p_1, V_1), (p_2, V_2), (p_3, V_3), \ldots]$

The stochastic vehicle generator generates the vehicle $\mathbf{V_i}$ with probability $\mathbf{p_i}$.

Traffic Light

Image by Author.

Traffic lights are placed at vertices and are characterized by two zones:

- **Slow down zone:** characterized by a *slow down distance* and a *slow down factor*, is a zone in which vehicles slow down their maximum speed using the slow down factor.

$$v_{0,i} := \alpha\, v_{0,i} \text{ with } \alpha < 1$$

- **Stop zone:** characterized by a *stop distance*, is a zone in which vehicles stop. This is achieved using a damping force through this dynamics equation:

$$\frac{dv_i}{dt} = -b_i \frac{v_i}{v_{0,i}}$$