Eindhoven University of Technology
Department of Mathematics & Computer Science

Platoon forming algorithms for centrally controlled autonomous vehicles

Thomas van Kaathoven Waseem Khan David Meijll

Contents

1 Introduction

Although currently still in its infancy, self-driving cars have the potential to change the traffic landscape in a major way. For instance, the use of autonomous cars can eliminate the element of human error, which often leads to accidents and delay, and therefore also a lot of frustration in people.
Properly implementing self-driving cars requires communication between vehicles, and it requires some central controller that can determine the speed of the cars, to ensure traffic flow is optimal.
In this report, we investigate the working of this scenario on a two-lane intersection using computer simulation. As can be imagined, once cars enter the intersection area from multiple lanes, the controller will have to decide which car can go first. This decision can be split into two parts; the first regards *platoon forming*, and the second decision regards *speed profiling*, which will be explored in this report. The combination of the two then decides the trajectory of each vehicle on the intersection.

Our goal in this report is to simulate the intersection, and compute and visualise the trajectories of a number of vehicles arriving on both 'axes' of the intersection. To do this, we first describe the modeling specifications, and then use Python to perform the calculations and simulations. Finally, we will present and discuss our findings.

2 Model description

Since we are considering a hypothetical, ideal situation, we need to make a number of assumptions that might not reflect reality. These are, however, needed to simplify the modeling process, allowing us to obtain meaningful results. For instance, we assume the presence of autonomous cars and a central controller has eliminated all traffic lights and signs.
To start off, we are interested in some general parameters that are used in the model.

1. the intersection consists of two lanes, which we will call lane 0 and lane 1 from now on.
2. The *control region* that surrounds the intersection extends 300 meters from the intersection in all directions.
3. The maximum speed inside the control region is $v_m = 13$ m/s.
4. The acceleration and deceleration rate inside the control region is $a_m = 3$ m/s^2.

Furthermore, we assume the vehicles arrive such that their interarrival times follow a bunched exponential distribution, with parameters α and μ that need to be estimated.
Then, we make some assumptions regarding the controller.

5. The controller has full control over the speed of the vehicles.
6. The controller knows the arrival time of each vehicles entering the control region.
7. The controller ensures all vehicles enter the actual intersection a full speed.

As mentioned earlier, the first decision of the controller had something to do with platoon forming. This means that a number of vehicles on the same lane will form groups, platoons, in which they can drive very close to each other. The controller can then consider this platoon as a single entity that needs to cross the intersection. Again, this requires some assumptions.

8. The vehicles in a platoon drive at $B = 1$ second from each other, measured from the front of one car to the front of the next.
9. The time between a platoon from one lane crossing the intersection and a platoon from another lane crossing needs to be at least $S = 2.4$ seconds.

Having defined platoons, we can now determine the crossing times, the time when a vehicle starts crossing the intersection at full speed, from the arrival times. This is done using *Platoon forming algorithms*, as introduced by Timmerman and Boon in 2021 [1]. They view the controller process as a problem in queueing theory.
When looking at the situation as a queue, we can distinguish two cases: either the controller implements a *first-come-first-served* (FCFS) policy, or a *Cyclic exhaustive service* (CES). In the first case, the controller keeps track of which vehicle entered the control region first, and lets them cross in that order. In the second case, the controller only considers the second queue once the first queue is completely empty.

The second decision of the controller regards speed profiling. This means that the speed of the vehicles can be adjusted in multiple ways to ensure they cross the intersection at maximum speed at the right time. These speed profiles are grouped into two main categories. In the first, there is a time t_{dec} at which the vehicles in a platoon decelerate, a time t_{stop} at which the vehicles come to a complete stop, and times t_{acc} and t_{full} that signify the vehicle reaching full speed again. In the second case, the vehicles do not come to a complete stop, but only decelerate to a lower velocity before speeding up again. Based on the arrival time in the control region, the length of the control region, and the time of crossing, we can then use *Speed profiling algorithms* described by Timmerman and Boon to find the trajectory of all vehicles. This definition assumes that there can only be one phase of acceleration and deceleration, meaning that the controller cannot make any mistakes.

2.1 Data

Although we will generate cars by simulating the intersection, we are also given a dataset that contains the arrival times of a number of vehicles on both lanes. Using this data, we can estimate the parameters of the bunched exponential distribution described above.

3 Implementation

As stated in the introduction, we are interested in simulating the autonomous intersection, and computing the trajectories of each vehicle inside the control region. To accomplish this goal, we use object-oriented programming in Python. All code can be found in Appendix A, and all results will be discussed in section 4.

3.1 Estimation of parameters

We first use the given data to estimate parameters α and μ. In order to do this, we first import the file and define the initial parameters $B = 1$, $S = 2.4$. Following this, we create a class `Lane` that takes the number of lanes, the file containing arrival times, and B as an input. We first define the constructor, which includes the following attributes:

- `n`: the number of lanes;
- `times`: the array of arrival times for both lanes;
- `b`: the time between vehicles in a platoon;
- `len`: the amount of cars that arrive in a lane throughout the control period;
- `deltas`: an array containing the time between each arrival;
- `offDeltas`: `deltas` with `b` subtracted;

We then estimate α in the following way. We define a new attribute `alpha`, equal to $1 - M$, where M is the fraction of arrivals such that `offDeltas` is equal to 0. In other words, M denotes the fraction of cars that are not in the same platoon as the previous vehicle.
Now, μ is estimated using a number of samples, in this case we use 20 samples. For each sample i, we then compute

$$\mu_i = \frac{\log\left(\frac{-\alpha}{\text{mean}(\texttt{offDeltas } \leq t)-1}\right)}{t},$$

where

$$t = \frac{\frac{i}{2}\max\{\texttt{Deltas}\}}{\texttt{len}}$$

To then obtain our estimate, we take the mean of all samples μ_i. Additionally, we compute the standard deviation, to find a confidence interval for μ.
Having obtained the estimates, we define a test for α and μ within the class `Lane`. This test plots the bunched exponential distribution with our estimators as parameters, and then plots a histogram of the samples based on the input data. This is done for both lanes. Finally, the class contains a method that prints the estimates for α and for μ, including a 95%-confidence interval, for both lanes.

3.2 Simulation - Global FCFS

There are two ways to simulate the intersection. Our first model will assume the controller uses a FCFS-method to let cars cross. We made use of several classes using Future Event Scheduling to create an Object Oriented Programming approach. We used 3 sub classes, namely the `Car`, `Event` and `FES` classes, as objects used in our simulation. Using a class named `SimResults`, we save the waiting and queuing times of the simulation.
Importing these 4 classes into our main class, `FCFSSimulation`, we tie the classes together to output a table of the arrival and departure times for the respective lanes and the class `SimResults` containing all the results.

We will next explain in detail the methods of the 5 classes used in our simulation. The classes `Car`, `Event`, `SimResults` and `FES` are used with the Cyclic Exhaustive service which we will describe later.
The first class, `Car`, creates the car object, initiated with the lane and arrival time at the crossing. We then implement a setter method to set the departure time of the car.
Next, we create the `Event` class, which we use to set the arrivals and departures of the cars. We start by setting `Arrival` to 0, and `Departure` equal to 1. If the event is an arrival, it will get the time of the car arriving during this event. On the other hand, if the event is departure, the event gets the time of the car departing during this event. To sort the events in order, we compare the time of the current event to the time of another event.
For the third class `FES`, Future Events Set, we use the Python heapq module, which is an implementation of the heap queue algorithm, or priority queue algorithm, and add 4 methods. These methods are firstly a heappush to add events, then a heappop to get the next event, thirdly a boolean `isEmpty` function which checks if the number of events is 0, and finally a function that returns the number of events in the heap.

The 4th sub class we use is the `SimResults` class. This class is used to keep track of the queue and waiting times of the simulation. The methods are as follows.

- `registerQueueLength` keeps track of the queue length at the start of each run.
- `registerWaitingTime` keeps track of the waiting times of the cars, and is used whenever a car departs, where waiting time is defined as the depTime-arrTime.
- `getMeanQueueLength` returns the sum of the queue lengths over the duration of the run.
- `getMeanWaitingTime` returns the sum of the waiting times over the count.
- `getVarianceWaitingTime` returns the variance of the waiting times.
- `getQueueLengthHistogram` returns the values of the histogram of the queue lengths.
- `getWaitingTimes` returns the waiting times.
- `getConfidenceInterval` returns the 95% confidence interval of the waiting times.
- `histQueueLength` returns a histogram of the queue lengths.
- `histWaitingTimes` returns a histogram of the waiting times.

Now that we have explained the 4 subclasses used, we will explain the FCFS algorithm. The FCFS algorithm takes in the parameters `lanes`, `alpha`, `mu`, `b`, `s`, `vm` and `x0`, and is initialised with these variables. We then simulate the run. First we create the variables to store the data using the classes we defined earlier. We create a list of deques to represent the lanes called `laneQueue`, a deque to represent all cars at the crossing called `crossingQueue`, a list of the class

SimResults called res, a table to store the respective lanes, arrival and departure times of the cars, and we create the FES. We then initialise 1 car for each lane and determine which car goes first.
With a while loop condition, we iterate the simulation until the number of cars simulated is equal to the total number of cars input as a parameter for the simulation. At the start of each run, we register the queue lengths for each lane. We then get the next event and the car associated with that event, labelling it as oldCar.
If the event is an arrival, we add the oldCar to the crossingQueue and respective lane in laneQueue. We then check if the car is the only car in the crossingQueue and if its arrival time is less than the previous departure time plus end of service depending on the previous lane. We then schedule a departure and add that event to the FES. We also schedule the next arrival event.
If the event is a departure, we remove the oldCar from the crossingQueue and its respective lane in the laneQueue. We then register the waiting time of the oldCar , taking t - the arrival time of the oldCar. Now that the oldCar has moved off, we can append its lane arrival time and departure time to the table of values. We also increase the number of cars that have run through the simulation by 1. If the length of the crossing queue is more than zero, i.e. there are cars at the crossing, then we take the first car to arrive at that junction, first come first serve, as car. Checking if the car came from the same lane or not, we add the respective service time and create a departure event for the car. We then add it to the FES and set the previous departure time to t. Thus concludes our FCFS Algorithm.

3.3 Simulation - Cyclic Exhaustive Service

Next, we assume the controller uses Cyclic Exhaustive service, as described in section 2. Since the only difference with the FCFS-method is the simulation, we can reuse our Car, Event, FES, and SimResults classes that were described earlier. However, the simulation process is different. The algorithm initialises with the same parameters, lanes, alpha, mu, b, vm and x0, and based on these it first computes $\frac{x_0}{v_m}$ and initialises the number of cars to 0. We then start the simulation by creating a queue using deques, a table to store the lanes, arrival and departure times, a list res to store the queue length and waiting times, and a list of FES for each lane. We then generate a Car, similarly to the FCFS simulation.
First, we check if the current lane is empty and if there are cars in other lanes. If so, using a while loop, we then check whether each lane is empty to decide which car gets to cross the intersection, using the method isEmpty. If the lane is empty, the controller moves over to the next lane. If all lanes are empty, we return to the initial lane. Next we take the first car in the queue and schedule a departure for it. We check if there are any more cars in that queue and schedule departures for those cars.
Next if there are no cars or there are events that happen before the events that are scheduled to happen in the current lane, we get the next event from the FES and append the car to the queue. If the lane is the same as the current lane and the length of the queue is only 1, we schedule the departure of the next car. We also create an arrival of a car and add it to FES.
Lastly, with an else statement, we get the next event and register the values in our res list and table. If there are any cars left in the current lane we schedule their departures. Concluding our cyclic algorithm, we return the table and res.

3.4 Trajectories

In order to visualise the trajectories of each car passing the control region, we first need to compute them. We do this following the method described in the assignment. The trajectories are stored in an array such that it can be converted to a spreadsheet with columns arrival time lane 0; departure time lane 0; arrival time lane 1; departure time lane 1, with the cars chrono-

logically ordered.
Then, we plot the trajectories as lines that start at either $y = -300$ for lane 0 or $y = 300$ for lane 1, and end at the origin, using the first 100 vehicles from the generated array as an input.

4 Results

In this section we will discuss the results generated by the implementation described in the previous section. Firstly, we estimated parameters α and μ for the bunched exponential distribution of the interarrival times. This was done using the class `Lane`. After defining this class and importing the input data, the program returned the following result lines:

```
"For lane 0, alpha = 0.5700000000000001 and mu = 0.5081058847543559 ± 0.00795..."
"For lane 1, alpha = 0.585 and mu = 0.38808240107704883 ± 0.00685..."
```

For both lane 0 and lane 1, $\alpha \approx 0.575$, but there is a clear gap between both μ's. Taking the 95%-confidence region into account, which currently displays five significant digits for readability, instead of the original 15 we produce, does not decrease this gap. We can explain this by behaviour on both lanes apparently being different, meaning that another input file might not have this gap between lanes.
However, we cannot form any final conclusions before checking the accuracy of our estimation. Figure 1 shows the comparison between the plot of the bunched exponential distribution and the histograms based on samples for both lanes.

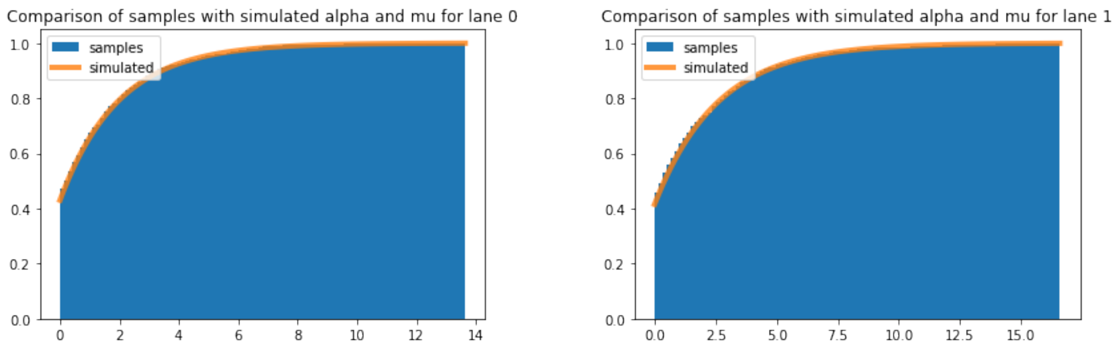


Figure 1: Comparison of simulated estimation and samples for lane 0 and lane 1.

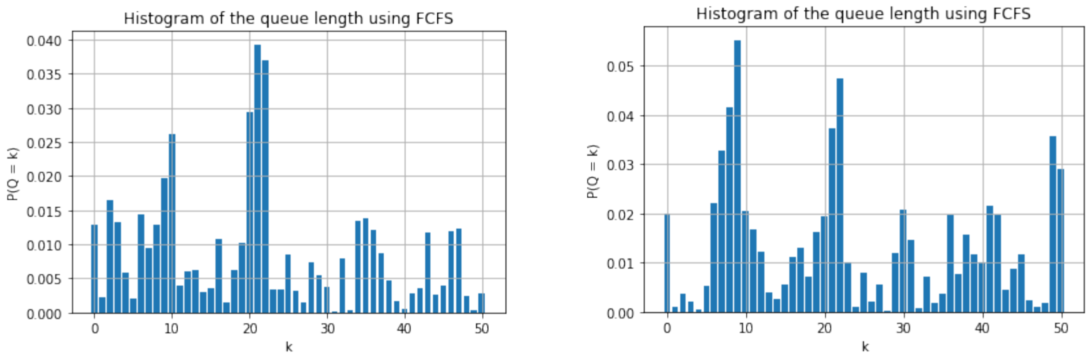
In both cases, we see that the histogram and line graph show a similar trajectory, implying that our estimations for α and μ are accurate.

Our next results are produced by the implementation of the FCFS-queue. Recall that in this case, the central controller clears the queues by looking at the arrival times of each vehicle, and then lets them cross the intersection in order of who arrived first. We are interested in investigating the queue length and waiting time for each lane, in the case that we simulate 100 vehicles entering the control region.
These results are shown in Table 1. Interesting to note is that there is a difference between waiting time for both lanes, although their 95%-confidence regions do overlap. As can be expected, the mean queue lengths differ in a similar fashion, since they are directly related.

	Lane 0	Lane 1
Mean queue length	53.165	38.076
Variance queue length	992.50	645.37
Mean waiting time	57.356	43.907
Variance waiting time	879.49	687.65
Confidence region waiting time	(49.764, 64.949)	(35.414, 52.401)

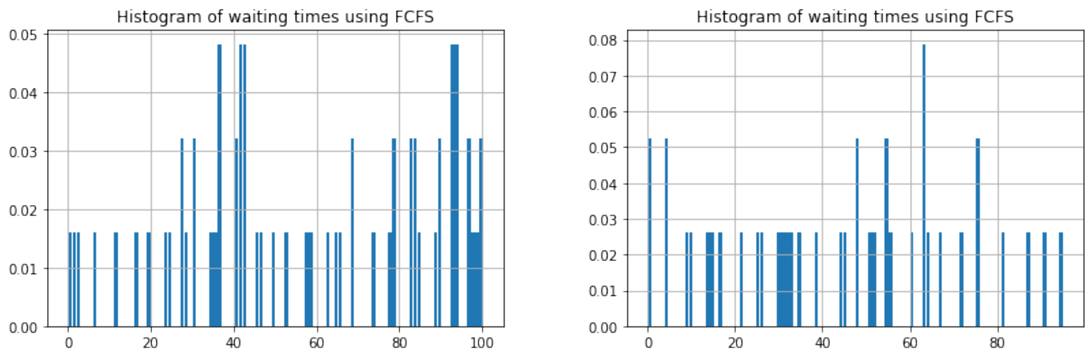
Table 1: Statistics for both lanes using the Global FCFS-method.

Apart from the table, Figure 2 depicts a histogram of the queue lengths in each lane. For any queue length k, this plot shows the fraction of times the queue had length k. We see that for lane 0, the largest peak occurs near $k = 20$, with a smaller peak around $k = 10$. In the other lane, we see similar behaviour at $k = 10$ and $k = 20$, although the peaks are higher. For instance, the peak queue length on lane 0 has a probability of about 4% of occurring, while on lane 1, the peak probability is over 5%. Additionally, we notice a smaller peak near $k = 50$ on lane 1, while this peak is completely absent on lane 0.



(a) Lane 0 (b) Lane 1

Figure 2: Histogram of queue lengths for lane 0 and lane 1 (FCFS).



(a) Lane 0 (b) Lane 1

Figure 3: Histogram of waiting times for lane 0 and lane 1 (FCFS).

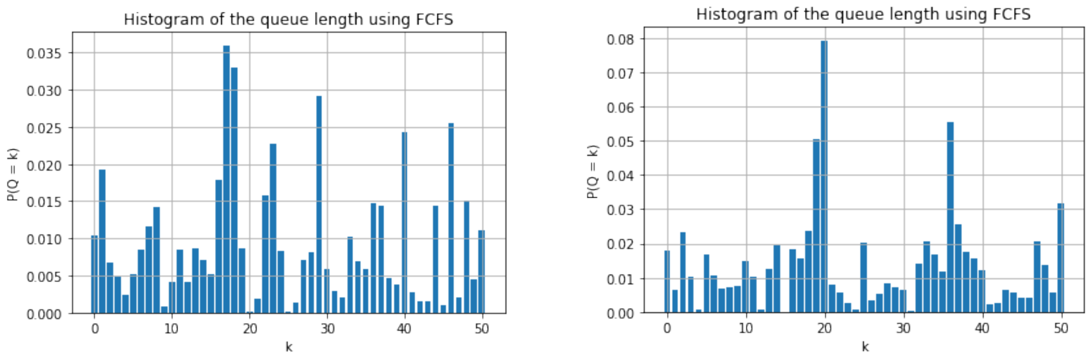
When we compare the waiting times of both lanes, which are shown in Figure 3, we notice a much larger difference between the two. Lane 0 is shown to have peak waiting times around time unit 40 and 95, while lane 1 has peaks near 0, 55 and 75. Furthermore, we see that overall, the waiting times in lane 1 are more spread out than in lane 0.

Following the results using the Global FCFS, we now repeat this process for the Cyclic exhaustive service, in which the central controller clears the vehicles one queue at a time. The mean queue length and waiting time can be found in Table 2. We notice that the queue length for lane 1 is much smaller on average, while the waiting times are a lot closer.

	Lane 0	Lane 1
Mean queue length	48.383	35.776
Variance queue length	683.00	434.87
Mean waiting time	58.538	54.207
Variance waiting time	1049.6	1059.7
Confidence region waiting time	(50.172, 66.905)	(43.805, 64.610)

Table 2: Statistics for both lanes using the Cyclic Exhaustive Service method.

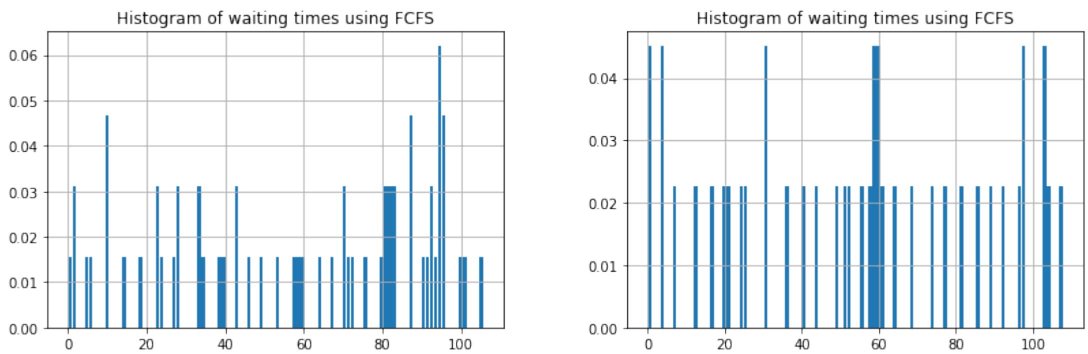
Again, we produce the same histograms as for the Global FCFS-strategy. In Figure 4, the distribution of queue lengths can be seen for both lanes. Just as with FCFS, we see some common peaks, but again, the peaks are higher for lane 1. This time however, when compared to lane 1, lane 0 is a lot more even across the board. When ignoring the two main outliers for lane 1 near $k = 20$ and $k = 35$, the two do behave in a similar way when it comes to distribution.



(a) Lane 0 (b) Lane 1

Figure 4: Histogram of queue lengths for lane 0 and lane 1 (CES).

Figure 5 shows the waiting time distribution for both lanes using CES. We instantly see that for lane 0, there is a large peak near 95 seconds, which is near the end of the spectrum. Lane 1 shows a similar, but much less intense peak. We can also see that both lanes have a large number of waiting times close to 0, which would be optimal.



(a) Lane 0 (b) Lane 1

Figure 5: Histogram of waiting times for lane 0 and lane 1 (CES).

As a final results, we plot the trajectories of 100 vehicles simulated with CES. The results can be seen in Figure 6. As expected, we see that there are some areas where most cars experience less delay. This can be explained by the fact that with CES, the traffic flow in one lane will be nearly optimal at some times, whenever the controller is clearing that specific lane. However, when both lanes are quite full, like near time unit 50, we see that some vehicles experience quite some delay.

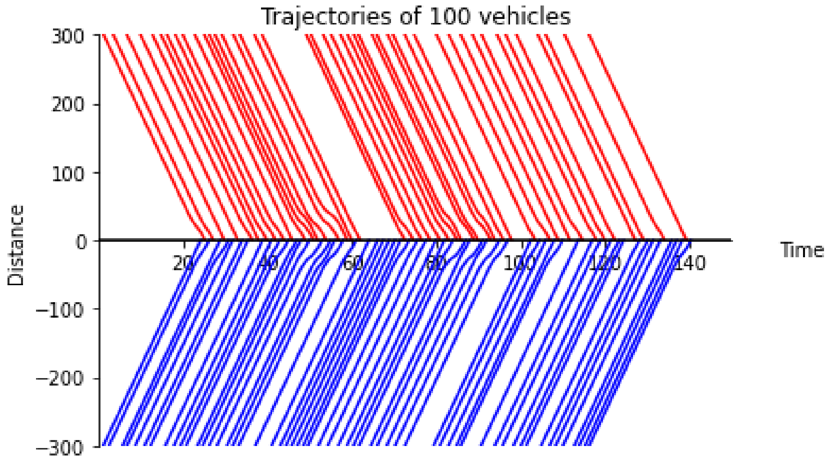


Figure 6: Trajectories of the first 100 vehicles using CES.

5 Conclusion

In this report, we have considered a scenario in which self-driving cars, controlled by a central controller, approach a two-lane intersection in groups, with lanes 0 and 1. Using computer simulation, we investigated multiple interesting properties of this scenario.

We started by estimating parameters that define the distribution of cars arriving to the intersection. By including confidence intervals and comparing simulation based on given input data, to a number of samples, we ultimately found that the first parameter, α, was approximately equal to 0.570 for lane 0, and approximately equal to 0.585 for lane 1. Parameter μ was found to be around 0.508 for lane 0, and 0.388 on lane 1, with a very tight 95%-confidence region. Since the histogram of samples and the line plot of the simulated distribution follow the same path when compared to each other, we can conclude that these estimators are accurate to use for simulation.

Next, we simulated the intersection scenario using two different methods of priority for groups of cars. The first method takes a first-come-first-served approach. We found that the average queue length for this method was approximately 53.165 on lane 0, and approximately 38.076 on lane 1. Additionally, the mean waiting time was found to be around 57.356 seconds on lane 0, and 43.907 seconds on lane 1, with 95%-confidence regions of $(49.764, 64.949)$ and $(35.414, 52.401)$ respectively. By comparing these numbers and the histograms of the queue length and waiting time on both lanes, we see that on average, the queues and waiting time are shorter in lane 1 with this method.
For the second method, the Cyclic exhaustive service method, the controller clears the queue one lane at a time, starting from lane 0. We found that the average queue length using this method was around 48.383 on lane 0, and approximately 35.776 on lane 1. The mean waiting time is around 58.538 seconds on lane 0, and 54.207 seconds on lane 1, with confidence regions (50.172, 66.905) and (43.805, 64.610) respectively. Therefore, by comparing these values to the FCFS ones, we can conclude that the queue lengths are shorter using CES, but the waiting times are better with FCFS.

Finally, we computed and plotted the trajectories of 100 simulated cars, using the Cyclic exhaustive method. The results show a clear distinction between platoons of cars on both lanes, and when compared to the examples shown in the assignment, we come to the conclusion that we have succeeded in accurately simulating cars on this intersection and visualising their trajectories.

6 Discussion

Although some elements of the discussion have already been included in the conclusion, we will use this section to address accuracy and realism of the report and scenario. Firstly, this scenario is purely theoretical. Even if fully autonomous vehicles do get integrated in traffic in such a way as presented in this report, some assumptions are made that decrease the realism of the study. For instance, we assume some exact driving speeds and distances between vehicles that cannot be as exact in the real world. We do not take into account any type of error, apart from a confidence region for the current situation with our assumptions.
Overall, our findings are accurate within the boundaries of the scenario. Even lifting some of the restrictions, our program would still work for any number of lanes. It would however still be interesting to expand our model to work for different types of roads, like a roundabout, or a system of intersections. Another extension could be to play around with the assumed parameters and controller strategies to find an ideal set that optimises traffic flow.

A Python Code

A.1 Classes

```
import numpy as np
from matplotlib import pyplot as plt
import copy
import scipy.stats as st
import heapq
import pandas as pd
from collections import deque
from numpy.ma.core import zeros

# Creating path for input data
from google.colab import drive
drive.mount('/content/gdrive')
cd /content/gdrive/MyDrive

-------------------------------------------------------------------------------
# Class to generate lanes and estimate their parameters for when new cars arrive
class Lane:

    def __init__(self, n, times, b):
        self.n = n
        self.times = times
        self.b = b
        self.len = np.alen(times)
        self.deltas = np.zeros(self.len)
        self.deltas[0] = times[0]
        self.deltas[1:] = times[1:] - times[:-1]
        self.offDeltas = self.deltas - b
        self.alpha = 1 - np.mean(self.offDeltas == 0)
        muSamples = 20
        muPoints = np.zeros(muSamples)
        for i in range(np.alen(muPoints)):
            t = (i + 1) * (np.amax(self.deltas) / 2) / np.alen(muPoints)
            muPoints[i] = np.log(-self.alpha / (np.mean(self.offDeltas <= t) - 1)) / t
        self.mu = np.mean(muPoints)
        self.muSTD = np.std(muPoints)
        self.muCI = 1.96 * self.muSTD / np.sqrt(muSamples)

    def testAlphaMu(self):
        plt_points = 100
        sim_input = np.linspace(0, np.amax(self.offDeltas), plt_points)
        sim_dist = 1 - self.alpha * np.exp(-self.mu * sim_input)
        plt.hist(self.offDeltas, bins=plt_points, label='samples', cumulative=True,
                 weights=[1 / self.len] * self.len)
        plt.plot(sim_input, sim_dist, alpha=0.8, label='simulated', linewidth=4)
        plt.legend()
        plt.show()

    def printAlphaMu(self):
        out = 'For lane ' + str(self.n) + ', alpha = ' + str(self.alpha) + ' and mu = ' + str(self.mu) + '±' + str(self.muCI)
        print(out)

-------------------------------------------------------------------------------
class Car:

    def __init__(self, arrTime, lane):
```

```
        self.arrTime = arrTime
        self.lane = lane

    def setDepTime(self, depTime):
        self.depTime = depTime

-------------------------------------------------------------------------------
class Event:

    ARRIVAL = 0
    DEPARTURE = 1

    def __init__(self, typ, car):
        self.type = typ
        self.car = car
        if self.type == self.ARRIVAL:
            self.time = self.car.arrTime
        elif self.type == self.DEPARTURE:
            self.time = self.car.depTime

    def __lt__(self, other):
        return self.time < other.time

-------------------------------------------------------------------------------
class FES:

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

    def add(self, event):
        heapq.heappush(self.events, event)

    def next(self):
        return heapq.heappop(self.events)

    def isEmpty(self):
        return len(self.events) == 0

    def getNumberOfEvents(self):
        return len(self.events)

-------------------------------------------------------------------------------
class SimResults:
  MAX_QL = 10000 # maximum queue length that will be recorded
  def __init__(self):
    self.sumQL = 0
    self.sumQL2 = 0
    self.oldTime = 0
    self.queueLengthHistogram = zeros(self.MAX_QL + 1)
    self.sumW = 0
    self.sumW2 = 0
    self.nW = 0
    self.waitingTimes = deque()

  def registerQueueLength(self, time, ql):
    self.sumQL += ql * (time - self.oldTime)
    self.sumQL2 += ql * ql * (time - self.oldTime)
    self.queueLengthHistogram[min(ql, self.MAX_QL)] += (time - self.oldTime)
    self.oldTime = time

  def registerWaitingTime(self, w):
    self.waitingTimes.append(w)
    self.nW += 1
    self.sumW += w
```

```
    self.sumW2 += w * w

  def getMeanQueueLength(self):
    return self.sumQL / self.oldTime
  def getVarianceQueueLength(self):
    return self.sumQL2 / self.oldTime - self.getMeanQueueLength()**2

  def getMeanWaitingTime(self):
    return self.sumW / self.nW
  def getVarianceWaitingTime(self):
    return self.sumW2 / self.nW - self.getMeanWaitingTime()**2

  def getQueueLengthHistogram(self) :
    return [x/self.oldTime for x in self.queueLengthHistogram]
  def getWaitingTimes(self):
    return self.waitingTimes

  def getConfidenceInterval(self):
    return (st.t.interval(confidence=0.95, df=len(self.waitingTimes)-1, loc=np.
   mean(self.waitingTimes), scale=st.sem(self.waitingTimes)))

  def __str__(self):
    s = 'Mean queue length: '+str(self.getMeanQueueLength()) + '\n'
    s += 'Variance queue length: '+str(self.getVarianceQueueLength()) + '\n'
    s += 'Mean waiting time: '+str(self.getMeanWaitingTime()) + '\n'
    s += 'Variance waiting time: '+str(self.getVarianceWaitingTime()) + '\n'
    s += 'Confidence Interval for the waiting time is' +str(self.
   getConfidenceInterval()) +'\n'
    return s

  def histQueueLength(self, maxq=50):
    ql = self.getQueueLengthHistogram()
    maxx = maxq + 1
    plt.figure()
    plt.bar(range(0, maxx), ql[0:maxx])
    plt.ylabel('P(Q = k)')
    plt.xlabel('k')
    plt.title('Histogram of the queue length using FCFS')
    plt.grid()
    plt.show()

  def histWaitingTimes(self, nrBins=100):
    plt.figure()
    plt.hist(self.waitingTimes, bins=nrBins, rwidth=0.8, density=True)
    plt.title('Histogram of waiting times using FCFS')
    plt.grid()
    plt.show()

-------------------------------------------------------------------------------
class FCFSSimulation:

    def __init__(self, lanes, alphaLst, muLst, b, s, vm, x0):
        self.lanes = lanes
        self.alphaLst = alphaLst
        self.muLst = muLst
        self.b = b
        self.s = s
        self.minT = x0 / vm
        self.nrCars = 0

    def simulate(self, totalCars):
        laneQueue = [] #Queues for each lane
        table = np.zeros((totalCars, 3))
        res = []
```

```
        crossingQueue = deque() #Queue for all lanes at crossing
        previousDepartureTime = 0

        for lane in range(self.lanes):
          laneRes = SimResults () # simulation results
          res.append(laneRes)
          laneQueue.append(deque())

        t = 0
        fes = FES()
        firstCarArrTime = 1000 #set random high number
        for lane in range(self.lanes):
            arrTime = t + self.b + rng.exponential(scale = 1/self.muLst[lane]) * self.alphaLst[lane]
            car = Car(arrTime, lane)
            arr = Event(Event.ARRIVAL, car)
            fes.add(arr)
            if arrTime < firstCarArrTime : #get first lane car moves from
                firstCarArrTime = arrTime
                currentLane = lane


        while self.nrCars < totalCars:
            e = fes.next()
            t = e.time
            lane = e.car.lane

            for l in range(self.lanes):
                res[l].registerQueueLength(t, len(laneQueue[l])) #register queue lengths for all lanes

            if e.type == Event.ARRIVAL:
                oldCar = e.car
                laneQueue[lane].append(oldCar)
                crossingQueue.append(oldCar)
                if len(crossingQueue) == 1: #only one lane can have cars moving so there is only 1 server
                    #if there is a free server,
                    res[lane].registerWaitingTime(t - oldCar.arrTime) #register car waiting time when car departs , if q is 0 waiting time shd be 0
                    if oldCar.lane == currentLane :
                        if oldCar.arrTime - previousDepartureTime <1: #if moving from same lane, time must be more than or equal to 1
                            depTime = previousDepartureTime +1
                        else:
                            depTime = t
                        car.setDepTime(depTime)
                        dep =Event(Event.DEPARTURE, oldCar)
                        previousDepartureTime = t
                    else:
                        if oldCar.arrTime - previousDepartureTime <2.4: #if moving from a different lane, time must be more than or equal to 2.4
                            depTime = previousDepartureTime +2.4
                        else:
                            depTime = t
                        oldCar.setDepTime(depTime)
                        dep =Event(Event.DEPARTURE, oldCar)
                        previousDepartureTime = depTime
                        currentLane =  oldCar.lane
                    fes.add(dep)

                if len(crossingQueue) != 0:
                    arrTime = t + rng.exponential(scale=1 / self.muLst[lane]) * self.alphaLst[lane]
```

```
                    car = Car(arrTime, lane)
                    arr = Event(Event.ARRIVAL, car)
                    fes.add(arr)

            elif e.type == Event.DEPARTURE:

                previousDepartureTime = t
                oldCar = e.car
                laneQueue[oldCar.lane].remove(oldCar)
                crossingQueue.remove(oldCar)
                res[oldCar.lane].registerWaitingTime(t - oldCar.arrTime) #car has moved off, so register its waiting time and remove it from queue

                table[self.nrCars, 0] = oldCar.lane
                table[self.nrCars, 1] = oldCar.arrTime
                table[self.nrCars, 2] = oldCar.depTime
                self.nrCars += 1

                if len(crossingQueue) > 0:
                    car = crossingQueue[0]

                    if car.lane == currentLane:
                        depTime = previousDepartureTime+self.b
                    else:
                        depTime = previousDepartureTime+self.s
                    car.setDepTime(depTime)

                    dep = Event(Event.DEPARTURE, car)
                    fes.add(dep)
                    currentLane = car.lane
                    previousDepartureTime = t

        return table , res

-------------------------------------------------------------------------------
class CyclicSimulation:

    def __init__(self, lanes, alphaLst, muLst, b, s, vm, x0):
        self.lanes = lanes
        self.alphaLst = alphaLst
        self.muLst = muLst
        self.b = b
        self.s = s
        self.minT = x0 / vm
        self.nrCars = 0

    def simulate(self, totalCars):
        queue = [deque()] * self.lanes
        table = np.zeros((totalCars, 3))
        res = []
        z = 0
        t = 0
        currentLane = 0
        fes = [[]] * (self.lanes + 1)
        fes[self.lanes] = FES()
        #create FES for each lane
        for lane in range(self.lanes):
            res.append(SimResults()) #simulation results
            fes[lane] = FES()
            arrTime = t + self.b + rng.exponential(scale = 1/self.muLst[lane]) * self.alphaLst[lane]
            car = Car(arrTime, lane)
            arr = Event(Event.ARRIVAL, car)
            fes[self.lanes].add(arr)
```

```python

        while self.nrCars < totalCars:
            for l in range(self.lanes):
                res[l].registerQueueLength(t, len(queue[l])) #register queue lengths for all lanes

            prevLane = currentLane
            notAllEmpty = 0
            for lane in range(self.lanes):
                #check which lane is empty
                notAllEmpty += not fes[lane].isEmpty()
            if fes[currentLane].isEmpty() and bool(notAllEmpty):
                #if current lane is empty and there are cars in other lanes, change to other lanes
                while fes[currentLane].isEmpty():
                    currentLane += (currentLane + 1) % self.lanes
                    if prevLane == currentLane:
                        break
                    #if all lanes are empty return to inital current lane
                car = queue[currentLane][0]#get first car in lane
                depTime = max(t + self.s, car.arrTime + self.minT) #add departure event of car
                car.setDepTime(depTime)
                dep = Event(Event.DEPARTURE, car)
                fes[currentLane].add(dep)
                oldCar = car
                if len(queue[currentLane]) > 1: #if there are more cars in the lane schedhule departure events of all teh cars
                    for carEntry in range(1, len(queue[currentLane])):
                        car = queue[currentLane][carEntry]
                        depTime = max(oldCar.depTime + self.b, car.arrTime + self.minT)
                        car.setDepTime(depTime)
                        dep = Event(Event.DEPARTURE, car)
                        fes[currentLane].add(dep)
                        oldCar = car

            if not bool(notAllEmpty) or fes[self.lanes].events[0].time < fes[currentLane].events[0].time:
                # if there are no cars or there are events that occur before the current lane,
                e = fes[self.lanes].next()
                t = e.time
                car = e.car
                lane = car.lane
                queue[lane].append(car)
                if lane == currentLane and len(queue[lane]) == 1:
                    #if same lane and there is only one car, schedhule departure of the car
                    depTime = t + self.minT
                    car.setDepTime(depTime)
                    dep = Event(Event.DEPARTURE, car)
                    fes[lane].add(dep)
                arrTime = t + self.b + rng.exponential(scale=1 / self.muLst[lane]) * self.alphaLst[lane]
                #create arrival of car
                car = Car(arrTime, lane)
                arr = Event(Event.ARRIVAL, car)
                fes[self.lanes].add(arr)

            else:
                e = fes[currentLane].next()
                t = e.time
                oldCar = e.car
```

```
                    res[oldCar.lane].registerWaitingTime(t - oldCar.arrTime)
                    queue[currentLane].remove(oldCar)
                    table[self.nrCars, 0] = oldCar.lane
                    table[self.nrCars, 1] = oldCar.arrTime
                    table[self.nrCars, 2] = oldCar.depTime
                    self.nrCars += 1

                    if len(queue[currentLane]) > 0:
                        car = queue[currentLane][0]
                        waitTime = self.b
                        depTime = max((car.arrTime + self.minT), (oldCar.depTime +
    waitTime))
                        car.setDepTime(depTime)
                        dep = Event(Event.DEPARTURE, car)
                        fes[currentLane].add(dep)

        return table,res
```

A.2 Executables

```
--------------------------------------------------------------------------------
# Estimate alpha and mu from the arrival data

timeLst = pd.read_excel('arrivals22.xlsx', header = None).to_numpy()
nCarPerLane, lanes = np.shape(timeLst)
b = 1
s = 2.4
laneLst = []
alphaLst = np.zeros(lanes)
muLst = np.zeros(lanes)
for n in range(lanes):
    laneLst.append(Lane(n, timeLst[:,n], b))
    laneLst[n].printAlphaMu()
    laneLst[n].testAlphaMu()
    alphaLst[n] = laneLst[n].alpha
    muLst[n] = laneLst[n].mu

--------------------------------------------------------------------------------
# Simulation using FCFS

vm = 13
x0 = 300
sim = FCFSSimulation(lanes, alphaLst, muLst, b, s, vm, x0)
table , results = sim.simulate(100)
_, counts = np.unique(table[:, 0], return_counts = True)
carLstOut = np.zeros((max(counts), 2 * lanes))
for n in range(lanes):
    laneArray = table[table[:, 0] == n]
    laneArray = np.pad(laneArray, ((0, max(counts) - counts[n]), (0, 0)), '
    constant', constant_values=0)
    carLstOut[:, 2 * n] = laneArray[:, 1]
    carLstOut[:, 1 + 2 * n] = laneArray[:, 2]

for lane in range(lanes):
    results[lane].histQueueLength() # plot of the queue length
    results[lane].histWaitingTimes() # histogram of waiting times
    print(str(results[lane])) #confidence interval of waiting times

--------------------------------------------------------------------------------
# Simulation using CES

vm = 13
x0 = 300
```

```
sim = CyclicSimulation(lanes, alphaLst, muLst, b, s, vm, x0)
table, res = sim.simulate(100)
_, counts = np.unique(table[:, 0], return_counts = True)
carLstOutC = np.zeros((max(counts), 2 * lanes))
for n in range(lanes):
    laneArray = table[table[:, 0] == n]
    laneArray = np.pad(laneArray, ((0, max(counts) - counts[n]), (0, 0)), 'constant', constant_values=0)
    carLstOutC[:, 2 * n] = laneArray[:, 1]
    carLstOutC[:, 1 + 2 * n] = laneArray[:, 2]

for lane in range(lanes):
    results[lane].histQueueLength() # plot of the queue length
    results[lane].histWaitingTimes() # histogram of waiting times
    print(str(results[lane])) #confidence interval of waiting times

-------------------------------------------------------------------------------
# Compute and plot trajectories of simulated cars

carLst = carLstOutC    #array Cyclic method

#setting variables according to assignment
nCarPerLane, lanes = np.shape(carLst)
lanes = int(lanes / 2)
am = 3
vm = 13
x0 = 300
b = 1
minT = x0 / vm
stopDist = b * vm
prevCar = CarTrajectory(-minT, 0, am, vm, x0)

#plot settings
fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)
ax.spines['bottom'].set_position('center')
ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')
ax.xaxis.set_ticks_position('bottom')
ax.yaxis.set_ticks_position('left')
ax.set_ylabel('Distance')
ax.set_xlabel('Time')
ax.xaxis.set_label_coords(1.11, 0.5)

for i in range(lanes):
    for n in range(nCarPerLane):
        if n > 0 and carLst[n, 2 * i] < (prevCar.dep - minT): # pass data on previous car if too close
            car = CarTrajectory(carLst[n, 2 * i], carLst[n, 1 + 2 * i], am, vm, x0, stopDist, prevCar.xFullEnd)
        else:
            car = CarTrajectory(carLst[n, 2 * i], carLst[n, 1 + 2 * i], am, vm, x0)
        prevCar = car
        #get time and trajectory output for specific car at roughly 0.5s intervals (makes them shorter if nessecary to also fit both the start and end point)
        time, distance = car.trajectory(0.5)
        #generate plot line from trajectory for said car
        if i == 1:
            distance = distance * -1
            clr = 'r'
        else:
            clr = 'b'
```

```
        plt.plot(time, distance, color=clr)

#more plot settings
plt.title('Trajectories of 100 vehicles')
plt.axis([0.1, 220, -300, 300])
plt.axhline(0, color='k')
plt.show()
```

References

[1] R.W. Timmerman and M.A.A. Boon. "Platoon forming algorithms for intelligent street intersections". In: *Transportmetrica A: Transport Science* 17.3 (2021), pp. 278–307. DOI: 10.1080/23249935.2019.1692962.