Investigating the Effects of Ramp Metering on Traffic Flow in Complex Traffic Systems

Team 43

Kiarash Ahmadi (kahmadi3)

Ryan Cooper (rcooper62)

Jason Lu (jlu367)

Github Repo: https://github.gatech.edu/rcooper62/cse6730project

Table of contents

Introduction

Traffic congestion serves as a paramount issue in a society that is growing exponentially more advanced in the 21st century. The city of Atlanta itself is ranked as the 10th most congested city in the U.S. The majority of traffic congestion occurs in or within a major metropolitan area, namely where intersections of major interstates and/or highways occur. To alleviate this issue, the city of Atlanta undertook a procedure to install ramp meters in designated areas in and around the city. A ramp meter is a traffic signal that regulates the flow of traffic entering freeways according to on-demand traffic conditions. Its purpose is to alleviate congestion that could occur in areas nearby.

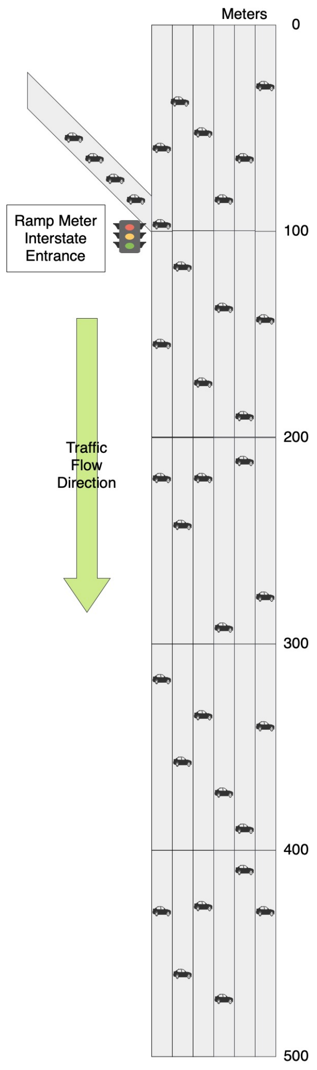
For our project, we will look further into the effects of ramp metering on an advanced system and observe its impact on traffic flow across an intersection of two major interstates. The basis behind our simulation consists of applying a microscopic traffic flow model that is based on dynamic parameters such as acceleration and velocity of a single vehicle. From there, our simulation incorporates lane-changing by applying a safety and advantage gain criteria. The safety criterion is based on maintaining a deceleration below a certain threshold and the advantage gain criterion assesses the gain that would come about from a driver if they change lanes and if that gain is marginal, they will not change lanes. From there, traffic flow data is retrieved via the Georgia Department of Transportation and inputted within our simulation model.

Description of System

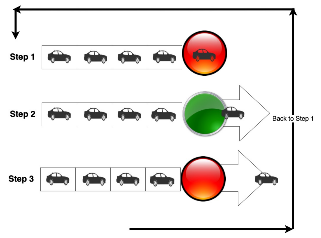
The system for the scope of this project focuses on the impact of traffic ramp meters on the overall traffic flow and congestion. The area of focus is the I-75 and I-285 intersection where the traffic flow is analyzed when assessing varying behavior from the ramp meters. This site was selected as it had readily available data at multiple points in the intersection and there are ramp meters at certain points in the intersection. The data is from the Georgia Department of Transportation and it is used in order to gauge and have more accurate traffic flow and speed data. For the intersection, there are 4 distinct areas of input and output: I-75 North, I-75 South, I-285 East (also called I-285 North) , and I-285 West (also called I-285 South), each of which has vehicles heading in opposing directions. A vehicle can also reach any of these outputs from any of the inputs, as the intersection is designed in a way to reach any designated input/output area. There will be two ramp meters located at designated spots on the interchange. With this traffic flow network, and with a plethora of data available, we are able to analyze our system with efficiency and accuracy.

Conceptual Model of System

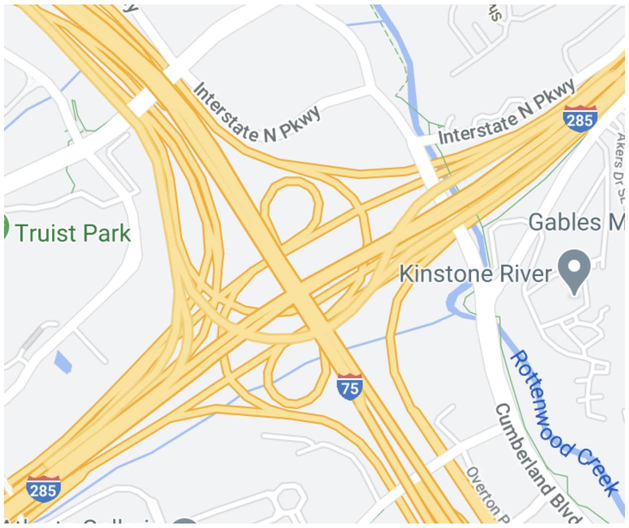
Our system is designed with the intent of producing a Discrete Event Simulation. The simulation will analyze the flow of vehicles in discrete time stepped patterns, with each time step representing a particular instance of the model. Here, we will look a look of a system of a freeway, which will serve as a proof-of-concept to how the interchange system will work. In total, the freeway of our proof-of-concept study area will consist of six lanes and will last 500 meters long. Vehicles will enter from any of the lanes at a random arrival rate followed by a normal distribution and will not be analyzed after they leave the system once they reach the end of the 500 meter study area. Each vehicle's velocity will change along the system depending on the it's surroundings. Vehicles may change lanes at anytime while being analyzed in the system. The ramp meter will be located at the 100 meter mark of the system and vehicles will enter the freeway from the right. This concept is visualized in Figure 1. There exists a discrete flow model of the system given a particular instance in time. Vehicles continue to travel down the freeway until it exits the system. The ramp meter represents a server which will process vehicles wanting to enter the system. Vehicles will be before the ramp meter and processed sequentially one at a time onto the freeway. The ramp meter interstate entrances will be represented by a queuing model. A queue will hold all the vehicles that are waiting to be processed and enter the free way. The arrival rates of the vehicles to the queue will be determined by a normal distribution. At the ramp meter light, one car will be processed at a time, and the rate at which the vehicles are processed will be deterministic at a designated service rate. Once processed, vehicles will exit the queue and enter the freeway along the direction of traffic flow. The figure below helps to visualize this process



The figure below gives a visual example of the 3 steps for which the vehicles will be processed. The red light represents that a vehicle may not pass through the ramp meter yet. The green light represents an all clear to enter the freeway.



The actual system we will be studying will take this proof-of-concept and apply it to a 4-way interchange. This interchange will have traffic flow in opposing directions along each of the four ways, and will also be analyzed on a discrete time-stepped basis. The following two figures below provide a map and satellite view of the system, respectively. The ramp meters for the interchange will be located on three designated spots and will likewise be modeled as it was in the proof-of-concept.



CUMBERLAND
ttery Atlanta
75
285
Cumberland Blvd

This simulation draws inspiration from a piece of literature that focuses on the simulation of freeway and urban traffic via a time-continuous microscopic traffic flow model or better known as the Intelligent Drivel Model (IDM). This approach helps to describe the dynamic behavior that occurs during traffic flow via the position and speed for each respective vehicle. The following equations are used to represent the dynamic position and speed of each vehicle.

$$\dot{x}_i = \frac{dx_i}{dt} = v_i$$

$$\dot{v}_i = \frac{dv_i}{dt} = a(1 - (\frac{v_i}{v_0})^\gamma - (\frac{s^*(v_i, \Delta v_i)}{s_i})^2)$$

Note that acceleration can be separated to a free road term and an interaction term where:

$$\dot{v}_i^{free} = a(1 - (\frac{v_i}{v_0})^\gamma)$$

$$\dot{v}_i^{int} = -a(\frac{s^*(v_i, \Delta v_i)}{s_i})^2)$$

In addition, s represents the deceleration that is a function as seen below:

$$s^*(v_i, \Delta v_i = s_0 + v_i T + \frac{v_i \Delta v_i}{2\sqrt{ab}}$$

Where v_0, s_0, T, a, and b are parameters within the model that represent the following:

- Desired velocity (v_0): the velocity of the vehicle if there was no traffic
- Minimum spacing (s_0): a minimum desired net distance
- Desired time headway (T): the minimum possible time to the vehicle in front
- Acceleration (a): the maximum acceleration of a car
- Braking deceleration (b): a positive number that represents the maximum deceleration of a car

This method of developing a simulation model works to improve the weaknesses of previous methods by addressing the loss of realistic properties in the deterministic limit. These limitations are addressed by applying a microscopic traffic flow model which aims to simulate single vehicle-driver units, so the dynamic variables of the models represent microscopic properties like the position and velocity of single vehicles. The implementation of this method can be seen below.

Imports, Parameters Initialization, and Implementations

In [1]:

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import csv
import os

from tqdm.notebook import tqdm

%matplotlib inline
```

In [2]:

```python
# IDM Params
DELTA = 4.0
MAIN_BSAVE = 9. # max deceleration

# Lane Change Params
MAIN_SMIN = 5. # gap for change lane

# Car Factory Params
V0_INIT = 120 # this is in KMH
S0_INIT = 2 # minimum distance allowed
T_REACTION = 2.3 # reaction time
A_INIT = 3.0 # acceleration factor
B_INIT = 3.0 # braking factor
P_FACTOR = 0.1 # Politeness
DB = 0.3 # change lane threshold

# Car Params
SPEED_LIMIT_KMH = 112.654 # kmh
T_DELAY_CHANGE = 1.6 # time to check lane change
LEN_CAR = 3 # car length
```

```
# Street Params
TIME_STEP = 0.5
BOUNDARY_DISTANCE = 200
INSERT_GAP = 7
```

In [3]:

```
class IDM():
    ismax = 100 # ve(s) = ve(ismax) if s > ismax

    def __init__(self, v0, a, b, s0=2, T=1.5):
        '''
        :param v0: max speed of the car
        :param delta: technical term in the acc calculation
        :param a: max acceleration m/s^2
        :param b: normal deceleration m/s^2
        :param s0: default 2
            least safe distance between two cars
            --> exceed this means need to brake immediately
        :param T:  default 1.5 human reaction time T for s*
        '''
        self.v0, self.delta, self.a, self.b, self.s0, self.T = v0, DELTA, a, b, s0, T
        self.sqrtab = np.sqrt(a*b)
        self.veq_table = np.zeros(self.ismax + 1)
        self.initialize() #generate equilibrium velocity table


    def initialize(self):
        dt = 0.5 #relaxation timestep 0.5s
        kmax = 20 #number of iteration in relaxation
        for s in range(1, self.ismax+1):
            Ve = self.veq_table[s-1]
            for k in range(0, kmax):
                s_star = self.s0 + Ve*self.T
                acc = self.a * (1.-np.power(Ve/self.v0, self.delta) - (s_star**2) / (s*
                Ve += acc * dt
                Ve = max(Ve, 0) # can't be lower
            self.veq_table[s] = Ve

    def Veq(self, dx):
        '''
        function for equilibrium velocity using veq_table;
            ve(s>ismax)=ve(ismax)
        this value is used to set up initial speed of vehicle
        :param dx: the distance between the current and forward car
        :return: velocity in m/s
        '''
        s = int(np.floor(dx))
        V = 0
        if s < 0:
            pass # V=0
        elif s < self.ismax:
            rest = dx - s
            V = (1-rest) * self.veq_table[s] + rest * self.veq_table[s+1]
        else:
            V = self.veq_table[self.ismax]
        return V
```

```python
    def calc_acc(self, bwd, fwd):
        '''
        :param bwd: Moveable, The current vehicle
        :param fwd: Moveable, The vehicle in the forward vehicle
        :return: acceleration m/s^2
        '''
        delta_v = bwd.vel - fwd.vel
        s = bwd.distance_to(fwd)
        vel = bwd.vel
        s_star_raw = self.s0 + vel * self.T\
                        + (vel * delta_v) / (2 * self.sqrtab)
        s_star = max(s_star_raw, self.s0)
        acc = self.a * (1 - np.power(vel / self.v0, self.delta) - (s_star **2) / (s**2)
        acc = max(acc, -6)
        return acc

    def get_v0(self):
        '''
        get maximum designed speed of this vehicle
        '''
        return self.v0
```

The next piece of literature that we drew inspiration from focuses more on the process behind creating a general lane-changing model. This is done through the development of a set lane-changing rules for discretionary and mandatory (depending on the traffic) lane changes for various car-following models. These rules are derived from the assessment of the utility of a given lane and the risk associated with a lane change using the common parameters in a car-following model such as acceleration, velocity, velocity-differences, and space between cars in a lane. Within this model, a safety criterion is applied to prevent collisions that assess the effect on the upstream vehicle in the target lane. The criterion is outlined below:

$$\tilde{a_n} \geq -b_{safe}$$

Where a_n is deceleration of the successor and b_{safe} is a safety limit. This criterion has flexibility in it all the information provided by a longitudinal car-following model. Beyond this safety criterion, the incentive criterion takes into account the advantages and disadvantages for other drivers associated with a single car's lane change via a politeness factor. This factor allows one to vary the motivation behind a lane change from one that is purely self-interested to a more cooperative behavior. The criterion is outline as seen below:

$$\tilde{a_c} - a_c + p(\tilde{a_n} - a_n + \tilde{a_o} - a_o) > \Delta a_{th}$$

Where

- a_c and $\tilde{a_c}$ represent the old and new acceleration of the driver
- a_n and $\tilde{a_n}$ represent the old and new acceleration of the new follower
- a_o and $\tilde{a_o}$ represent the old and new acceleration of the old follower
- p represents the politeness factor

- Δa_{th} represents an acceleration threshold that prevents lane changes if the overall advantage is marginal.

Incorporating this criterion makes for a more holistic analysis of whether to perform a lane change in that it depends on if the advantage gained is high enough or marginal depending on the threshold and also assesses each driver's appetite for this lane change based on the politeness factor. What this piece of literature did next was apply this model to a traffic simulation of cars using the intelligent driver model like previously discussed. In addition, an open system with ramp was studied to investigate the lane-changing rate as a function of the overall traffic flow in the system. Their findings were that the lane-changing rate is mainly determined by the politeness factor but also on the considered location of the road. The next step for this study is to research the empirical justification of their system and to validate and calibrate their model in the real world. Our implementation for this approach can be seen below.

In [4]:
```
class LaneChange():
    '''
    Implementation of the lane-changing model MOBIL "Minimizing Overall Brakings Induce
    LEFT = -1
    RIGHT = 1
    '''
    def __init__(self, p, db, gap_min=MAIN_SMIN, bsave=MAIN_BSAVE, bias_right=0):
        '''
        :param p: politeness factor
        :param db: change lane incentive penalty
        :param gap_min: max safe distance
        :param bsave: max safe braking deceleration
        :param bias_right: bias (m/s^2) to drive right
        '''
        self.p, self.db, self.gap_min, self.bsave, self.bias_right = p, db, gap_min, bs

    def set_bias_right(self, bias):
        self.bias_right = bias


    def change_ok(self, me, f_old, b_old, f_new, b_new):
        '''
        :param me: the current car
        :param f_old: forward car old
        :param b_old: .. (maybe useless but keep for further use)
        :param f_new: ..
        :param b_new: ..
        :return: bool
            change or not
        '''
        # is 1 if new lane is on the right, else 0
        is_right = int(f_new.lane > me.lane)
        if (me.distance_to(f_new) <= self.gap_min or
                b_new.distance_to(me) <= self.gap_min):
            return False

        # check safety criterion (a > -bsave)
        b_new_acc = b_new.model.calc_acc(b_new, me)
        me_new_acc = me.model.calc_acc(me, f_new)
        if (b_new_acc < -self.bsave or me_new_acc < -self.bsave):
            return False

        # my advantage of acceleration on lane change
        me_acc_adv = me_new_acc - me.model.calc_acc(me, f_old) + \
                     is_right * self.bias_right
        b_new_acc_disadv = b_new.model.calc_acc(b_new, f_new) - b_new_acc
        if b_new_acc_disadv < 0:
            b_new_acc_disadv = 0
        return me_acc_adv - self.p * b_new_acc_disadv > self.db
```

Given that we are modeling traffic using a discrete time simulation, we define a car as a object that defines its state variables at a particular time in the simulation. We also define a variety of update functions based on IDM and MOBIL that compute the acceleration, velocity, and lane change timings. This class is fairly straightforward and is the main actor used in the simulation.

In [5]:

```
class Car():
    def __init__(self, x, v, lane, model: IDM, lane_change: LaneChange, length):
        self.pos = x
        self.vel = v # vel is in m/s
        self.lane = lane
        self.model = model
        self.lane_change = lane_change
        self.length = length
        self.acc = 0  # current acceleration
        self.acc_history = 0 # acc 1 calculation before
        self.tdelay = 0  # cumulative waiting time
        self.Tdelay = T_DELAY_CHANGE  # time to check whether change lane or not

    def __copy__(self):
        return Car(self.pos, self.vel, self.lane, self.model, self.lane_change, self.le

    @property
    def vel(self):
        return self.__vel

    @vel.setter
    def vel(self, v):
        if v > SPEED_LIMIT_KMH / 3.6:
            self.__vel = SPEED_LIMIT_KMH / 3.6

        elif v <= 0:
            self.__vel = SPEED_LIMIT_KMH

        else:
            self.__vel = v

    def time_to_change(self, dt):
        self.tdelay += dt
        if self.tdelay > self.Tdelay:
            self.Tdelay -= self.tdelay
            return True
        return False

    def translate(self, dt):
        self.pos += dt * self.vel

    def accelerate(self, dt, fwd=None):
        assert (fwd == None)
        if fwd != None:
            self.acceleration(fwd)
        self.vel += self.acc * dt
        if (self.vel < 0.):
            self.vel = 0.

    def acceleration(self, fwd=None):
        if fwd == None:
            return self.acc
        else:
            return self.model.calc_acc(self, fwd)

    def distance_to(self, fwd):
        return fwd.pos - self.pos - self.length

    def change(self, f_old, b_old, f_new, b_new):
        return self.lane_change.change_ok(self, f_old, b_old, f_new, b_new)
```

Extending the general Car framework, we define a human car, that is that reaction time is not instantaneous and velocity is calculated as a function of delta t, the purpose being to add smoothing to the acceleration and make things less rigid with acceleration computations.

In [6]:

```
class CarHuman(Car):
    acc_history = 0

    def accelerate(self, dt, fwd=None):
        if fwd != None:
            self.acceleration(fwd)
        self.vel += self.acc_history * dt
        self.acc_history = self.acc
        if (self.vel < 0.):
            self.vel = 0.
```

We also define boundary cars to be used as ghost objects in the simulation since we naively assume that we can compute acceleration, lane change, and distance to next as being always defined. These objects are to be generated at each update step in the simulation at the boundaries of the road segment to ensure all actual cars can correctly compute their update steps.

In [7]:

```
class BCCar():
    def __init__(self, x, v, lane, model, length=0):
        self.pos, self.vel, self.lane, self.model, self.length = x, v, lane, model, len

    @property
    def lane_change(self):
        return None

    @lane_change.setter
    def lane_change(self, lanechange):
        pass
    def time_to_change(self, dt):
        return False

    def translate(self, dt):
        pass

    def accelerate(self, dt, fwd=None):
        pass

    def acceleration(self, fwd=None):
        return 0.

    def distance_to(self, fwd):
        return fwd.pos - self.pos - self.length

    def change(self, f_old, b_old, f_new, b_new):
        pass
```

Car Factory is the main object generator passed into the class we have yet to discuss. This class is responsible for defining the IDM parameters for each car and generating car objects correcting given the parameters of the current state.

In [8]:

```
class CarFactory():
    def __init__(self):
        self.car_human_IDM = IDM(v0 = V0_INIT, a = A_INIT, b = B_INIT, s0 = S0_INIT, T
        self.lane_change_human = LaneChange(p = P_FACTOR, db = DB, gap_min = MAIN_SMIN,

    def create_vehicle(self, position, initial_gap, lane):
        IDM = self.car_human_IDM
        lc = self.lane_change_human
        v = IDM.Veq(initial_gap)

        return CarHuman(position, v, lane, IDM, lc, LEN_CAR)
```

The main driver in the simulation is the street itself. We treat this class as the one who is responsible for handling cars entering and exiting its segment. The way that this class works is pretty straightforward, at each iteration (delta t time), some number of cars are entered into the vehicle queue, that is, the non-existent space before the road segment where a potentially infinite number of cars can exist. We utilizing queuing heavily here since cars can't be inserted into any lane unless there is room for it (based on the defined space needed for a car to take up). We also define another queuing mechanism which we refer to as a ramp queue, that is, again, another non-existent road segment where cars are inserted only on to the right lane at some specified distance up. We do this to ensure that cars entering the road segment from a ramp are correctly inserted into the right lane (assuming there is room). This queuing method is naive in some ways, but ultimately, is fairly accurate in representing reality as we will demonstrate.

As was mentioned before, the Street class handles updating all cars that exist on it before moving them to another road segment or destroying them. In each update, we first created boundary cars to ensure all cars in the segment can properly compute their values, then we accelerate every car based on current position. Next, we prompt each car to change lane if they desire and then remove the boundary cars. Finally, we move each car's position given its new acceleration and then compute the new cars we need to dequeue in this step and remove cars past the road boundary. One other thing this class does is recording position and velocity states at each iteration; which makes metric computation extremely easy for evaluation.

This design of modeling individual road segments allows us to link flow between segments, where some road segments could be considered as sources, others as sinks, and other that just process flow through them. Given that each road segment has varying size and parameters, we believe this modular design is inherently powerful for traffic simulation.

In [9]:
```
class Street():
    def __init__(self, num_lane, road_length, car_factory, dt = TIME_STEP):
        self.num_lane, self.road_length, self.carfactory, self.dt= num_lane, road_lengt
        self.street = [] # positions sorted in decreasing order
        self.time, self.flow_out_speed, self.flow_in_speed = 0, 0, 0
        self.vehicle_wait, self.vehicle_in, self.vehicle_out = 0, 0, 0
        self.vehicle_wait_ramp = 0

        self.road_state = []
        self.road_state_vel = []
        self.cars_out = []

    def update(self, q_in, position = 0, insert_on_last_lane = False, change_lanes = Tr
        self.time += self.dt
        self.insert_BC() # add boundary
        self.accelerate() # calculate new velocity
        if change_lanes:
            self.change_lanes()
        self.clear_BC() # remove boundary

        self.translate() # pos += vel * dt
        self.sort() # derease order of car.pos

        self.io_flow(q_in, position=position, insert_on_last_lane=insert_on_last_lane)

        if self.time % 100 == 0:
            self.calc_io_flow()


        state = []
        vel_state = []
        for car in self.street:
            state.append((car.lane, car.pos))
            vel_state.append((car.lane, car.vel))

        self.road_state.append(state)
        self.road_state_vel.append(vel_state)

    def report(self):
        ''' Debug report for lane states '''
        self.vehicle_in, self.vehicle_out = 0, 0

        vels = [car.vel for car in self.street]
        lane_count = np.zeros(self.num_lane)
        for car in self.street:
            lane_count[car.lane] += 1
        print("-----------------------------------------------------------------")
        print ("time = {:5.2f}".format(self.time))
        print("total vehicle: {:4}, average speed {:4.2f}, cars in queue: {:3.2f} vehic
              .format(len(self.street), np.average(vels), self.vehicle_wait, self.vehic
        print("\t num cars in each lane {}".format(lane_count))
        print("-----------------------------------------------------------------")

    @staticmethod
    def compute_average_lane_dist(states, num_lane, road_length):
        distance = np.zeros((len(states), num_lane))
        for i, state in enumerate(states):
            x = np.array(state)

            for lane in range(num_lane):
                cars = []
                for car in x:
                    if car[0] == lane:
                        cars.append(car[1])

                cars = sorted(cars)
                cars = np.diff(np.array(cars)) - 5
                distance[i,lane] = np.mean(cars)

        distance= np.nan_to_num(distance, nan=road_length)
        return distance.mean(axis=0)

    def calc_io_flow(self):
        self.flow_in_speed = self.vehicle_in / (self.dt * 100)
        self.flow_out_speed = self.vehicle_out / (self.dt * 100)

    def first_index_on_lane(self, lane):
        if lane >= self.num_lane:
            raise IndexError("max lane = {}".format(lane))
        for idx in range(len(self.street)):
            if self.street[idx].lane == lane:
                return idx
        return -1

    def last_index_on_lane(self, lane):
        if lane >= self.num_lane:
            raise IndexError("max lane = {}".format(lane))
        for idx in range(len(self.street)-1, -1, -1):
            if self.street[idx].lane == lane:
                return idx
        return -1

    def next_index_on_lane(self, lane, idx):
        if lane >= self.num_lane:
            raise IndexError("max lane = {}".format(lane))
        for idx in range(idx-1, -1, -1):
            if self.street[idx].lane == lane:
                return idx
        return -1

    def prev_index_on_lane(self, lane, idx):
        if lane >= self.num_lane:
            raise IndexError("max lane = {}".format(lane))
```

```python
        for idx in range(idx+1, len(self.street)):
            if self.street[idx].lane == lane:
                return idx
        return -1

    def sort(self):
        sorted = False
        while not sorted:
            sorted = True
            for i in range(1, len(self.street)):
                if self.street[i-1].pos < self.street[i].pos:
                    self.street[i-1], self.street[i] = \
                        self.street[i], self.street[i-1]
                    sorted = False

    def translate(self):
        for car in self.street:
            car.translate(self.dt)

    def accelerate(self):
        for idx in range(len(self.street)):
            fwd = self.next_index_on_lane(self.street[idx].lane, idx)
            self.street[idx].acceleration(self.street[fwd])
        for car in self.street:
            car.accelerate(self.dt)

    def insert_BC(self):
        dx = BOUNDARY_DISTANCE
        for lane in range(self.num_lane):
            id_first = self.first_index_on_lane(lane)
            id_last = self.last_index_on_lane(lane)
            if id_first == -1:
                first_vel = 0
            else:
                first_vel = self.street[id_first].vel
            if id_last == -1:
                last_vel = 0
            else:
                last_vel = self.street[id_last].vel
            self.street.insert(0, BCCar(self.road_length+dx, first_vel, lane, self.carf
            self.street.insert(len(self.street), BCCar(0 - dx, last_vel, lane, self.car

    def clear_BC(self):
        self.street = [car for car in self.street if not isinstance(car, BCCar)]

    def change_lanes(self):
        for idx in range(len(self.street)):
            new_lane = []
            car = self.street[idx]
            # add possible new lanes, otherwise next loop do nothing
            if car.time_to_change(self.dt):
                if car.lane - 1 >= 0:
                    new_lane.append(car.lane - 1) # left is shadowed if right is True
                if car.lane + 1 < self.num_lane:
                    new_lane.append(car.lane + 1) # right first
            for lane in new_lane:
                f_old = self.street[self.next_index_on_lane(car.lane, idx)]
                b_old = self.street[self.prev_index_on_lane(car.lane, idx)]
                f_new = self.street[self.next_index_on_lane(lane, idx)]
                b_new = self.street[self.prev_index_on_lane(lane, idx)]
                if car.change(f_new=f_new, f_old=f_old, b_new=b_new, b_old=b_old):
                    self.street[idx].lane = lane

    def io_flow(self, q_in, position = 0, insert_on_last_lane=False):
        self.o_flow()
        self.i_flow(q_in, position = position, insert_on_last_lane = insert_on_last_lan

    def i_flow(self, q_in, position = 0, insert_on_last_lane=False):
        # in
        self.vehicle_wait += q_in * self.dt  # add to waitlist
        lanes = np.arange(self.num_lane)

        if insert_on_last_lane:
            lane = lanes[-1]
            if self.vehicle_wait_ramp > 1:
                self.vehicle_wait_ramp -= 1
                idx_fwd = self.last_index_on_lane(lane)
                if idx_fwd == -1:
                    distance = self.road_length
                else:
                    distance = self.street[idx_fwd].pos

                if distance >= INSERT_GAP:
                    self.street.append(self.carfactory.create_vehicle(position, distanc
                    self.vehicle_in += 1

        if insert_on_last_lane:
            lanes = lanes[:-1]

        np.random.shuffle(lanes)
        for lane in lanes:
            if self.vehicle_wait > 1:
                self.vehicle_wait -= 1
                idx_fwd = self.last_index_on_lane(lane)
                if idx_fwd == -1:
                    distance = self.road_length
                else:
                    distance = self.street[idx_fwd].pos

                if distance >= INSERT_GAP:
                    self.street.append(self.carfactory.create_vehicle(position, distanc
                    self.vehicle_in += 1

    def o_flow(self):
        # out
        origin = len(self.street)
        self.street = [car for car in self.street if car.pos < self.road_length]
        self.vehicle_out += origin - len(self.street)
        self.cars_out.append(origin - len(self.street))
```

Simulation Overview and Assumptions

That concludes the model definition! We have defined a discrete time simulator using Car objects in a Street controller, where the Street is responsible for managing and updating its children Cars.

We make a few naive assumptions that we will identify here. Firstly, we normally distribute all of the constant parameters about car objects in order to introduce variance into the simulation. This can cause things like long trailing backups, frequent lane changes, or other nuances that are hard to model explicitly. Second, the queuing model in our roads is two folded, both queuing into any lane and queuing onto the roads "ramp". Although this may seem correct, consider a situation where we have the joining of two road segments, one in which the inflow is far greater than the outflow. In the inflow road, it will appear as though traffic is flowing normally and all is good, but in the outflow road, it will be congested with an ever-growing queue. In essence, our model lacks feedback to previous segments. We get around this by verifying that our road segments are not creating massive queues and that out flow and in flow are preserved.

The simulations taking place will utilize the parameters above as necessary, each of which is commented concisely for your understanding. A series of simulation models will then be implemented and run accordingly.

The first simulation presented below demonstrates a simply two-lane road and the vehicle attributes on it over a series of time steps for a Discrete Event Simulation Model. This simulated was designed for the purpose of showing how the vehicles are set up to act and how the vehicles interact with each other.

Note that in all simulations below, the units for velocity are specified in meters per second. Unless otherwise specified, assume the speed limit for the road is 70 MPH, or 112.654 KPH.

In [10]:
```
# Simulation parameters

num_lane = 2
road_length = 500 # 5 km
flow_in = 1 # vehicle per second
dt = 0.1
iters = 10000

seed = 1111
np.random.seed(seed=seed)
```

In [11]:
```
cf = CarFactory()

road = Street(num_lane, road_length, cf, dt=dt)

for i in  tqdm(range(iters)):
    q_in = np.random.normal(flow_in, flow_in/2 ** 1/5, (1))[0]

    road.update(q_in, insert_on_last_lane=False)

#     if i % 1000 == 0:
#         road.report()
```

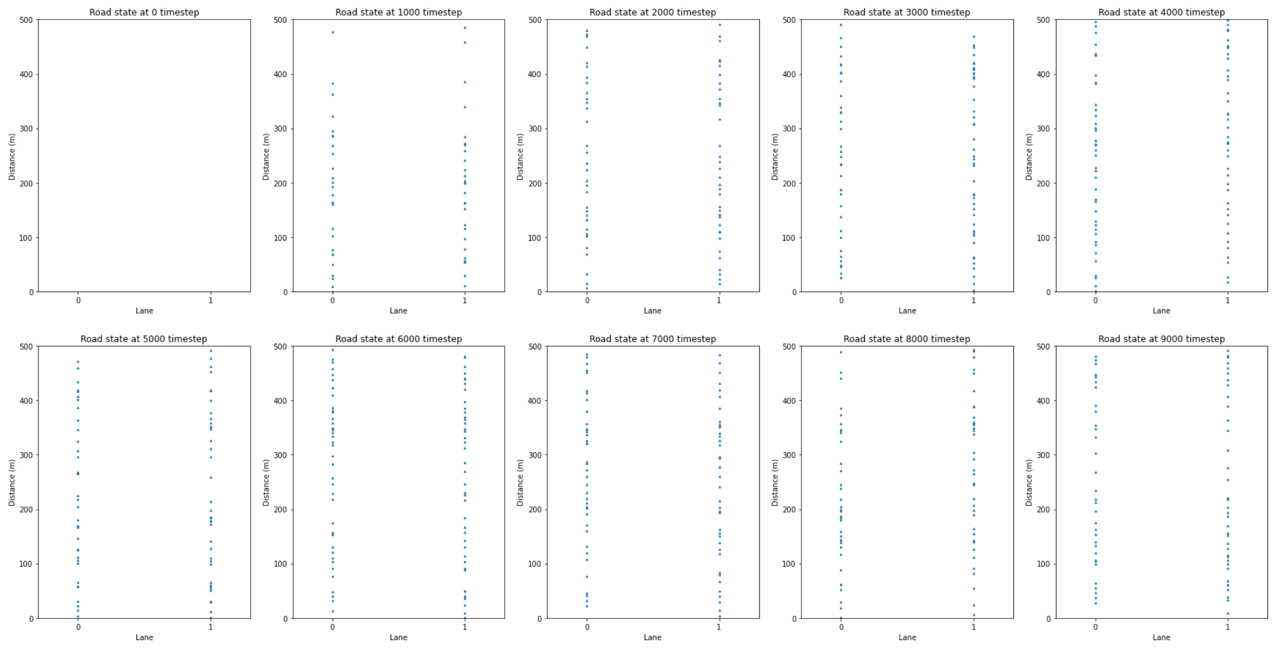
Out[11]:

The following figures help to visualize the system that is created where each marker represents a car, the distance on the road is defined by the y-axis, and the lanes are defined by the x-axis.

In [12]:

```
fig, axs = plt.subplots(2, 5, figsize=(30,15))

for i, state_idx in enumerate(range(0, iters, 1000)):
    state = road.road_state[state_idx]
    state_x = [s[0] for s in state]
    state_y = [s[1] for s in state]
    eps = 0.3
    axs[int(i/5), i%5].set_title('Road state at {} timestep'.format(state_idx))
    axs[int(i/5), i%5].set_xlabel('Lane')
    axs[int(i/5), i%5].set_ylabel('Distance (m)')
    axs[int(i/5), i%5].set_xlim(0 - eps, num_lane - 1 + eps)
    axs[int(i/5), i%5].set_xticks(range(0, num_lane))
    axs[int(i/5), i%5].set_ylim(0, road_length)
    axs[int(i/5), i%5].scatter(state_x, state_y, s=6, marker='^')
```

Out[12]:



Show that cars are entering from right lane when in ramp queue

The simulation presented demonstrates the right lane of the simple road, with the purpose of showing how the lane creates a build up when taking in vehicles from the ramp queue.

In [13]:

```
# Simulation parameters

num_lane = 3
road_length = 500 # 5 km
flow_in = 0.5 # vehicle per second
dt = 0.1
iters = 10000
```

```
seed = 1111
np.random.seed(seed=seed)
```

In [14]:

```
cf = CarFactory()

road = Street(num_lane, road_length, cf, dt=dt)

for i in tqdm(range(iters)):
    q_in = np.random.normal(flow_in, flow_in/2 ** 1/5, (1))[0]

    road.vehicle_wait_ramp += q_in * dt

    road.update(0, insert_on_last_lane=True, change_lanes=False)
```

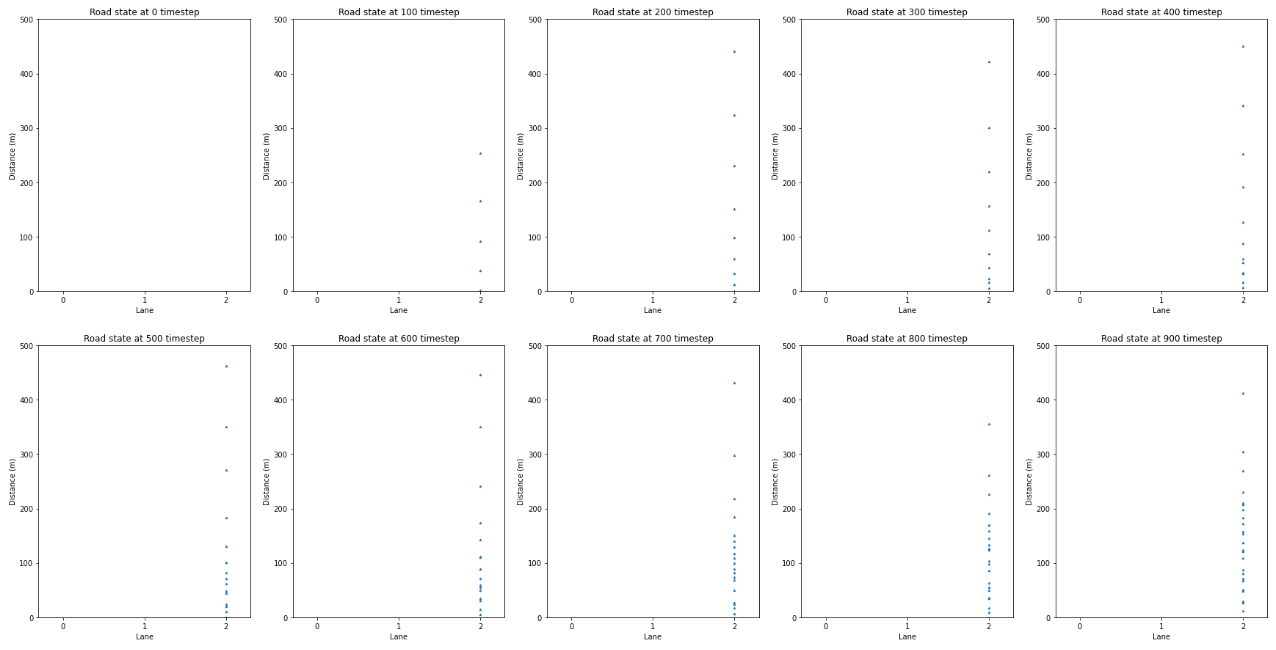
Out[14]:

The following figure helps to visualize the behavior of the simulation where each car enters from the right lane.

In [15]:

```
fig, axs = plt.subplots(2, 5, figsize=(30,15))

for i, state_idx in enumerate(range(0, 1000, 100)):
    state = road.road_state[state_idx]
    state_x = [s[0] for s in state]
    state_y = [s[1] for s in state]
    eps = 0.3
    axs[int(i/5), i%5].set_title('Road state at {} timestep'.format(state_idx))
    axs[int(i/5), i%5].set_xlabel('Lane')
    axs[int(i/5), i%5].set_ylabel('Distance (m)')
    axs[int(i/5), i%5].set_xlim(0 - eps, num_lane - 1 + eps)
    axs[int(i/5), i%5].set_xticks(range(0, num_lane))
    axs[int(i/5), i%5].set_ylim(0, road_length)
    axs[int(i/5), i%5].scatter(state_x, state_y, s=6, marker='^')
```

Out[15]:



Simulating a ramp

The simulation below demonstrates a six-lane freeway to serve as one of the simulations utilizing the proof-of-concept simulation model for the interchange. This serves as a more complex model compared to a road, and more reflective of the lanes of the interchange. This specific simulation below utilizes a ramp on its own without any ramp metering system implemented with the purpose of showing the effects of free ramping. This will serve as the ground basis proof-of-concept for comparison to the ramp metering systems to come.

In [2]:

```
# Simulation parameters

num_lane = 6
road_length = 500 # 5 km
ramp_length = 30 # 30m
ramp_position = 100 # place ramp 100m into the road
r_prev = 1
dt = 0.1

k_r = 70 / 3600

highway_flow_in = 10
ramp_flow_in = 0.5

seed = 1111
np.random.seed(seed=seed)
```

In [17]:

```
%%time

cf = CarFactory()

pre_ramp_highway = Street(num_lane, ramp_position, cf, dt=dt)
post_ramp_highway = Street(num_lane, road_length - ramp_position, cf, dt=dt)
ramp = Street(1, 30, cf, dt=dt)

ramp_queue = []

for i in tqdm(range(36000)):
    highway_q_in = np.random.normal(highway_flow_in, highway_flow_in/2 ** 1/5, (1))[0]
    ramp_q_in = np.random.normal(ramp_flow_in, ramp_flow_in/2 ** 1/5, (1))[0]

    pre_ramp_highway.update(highway_q_in, insert_on_last_lane=False)
    post_ramp_highway.vehicle_wait += pre_ramp_highway.vehicle_out
    pre_ramp_highway.vehicle_out -= pre_ramp_highway.vehicle_out

    ramp.update(ramp_q_in)
    post_ramp_highway.vehicle_wait_ramp += ramp.vehicle_out
    ramp.vehicle_out -= ramp.vehicle_out

    post_ramp_highway.update(0, insert_on_last_lane=True)

    ramp_queue.append((i, ramp.vehicle_out))
```

```
#     if i > 1000 and i % 1000 == 0:
#         post_ramp_highway.report()
#         print('cars at ramp', ramp.vehicle_out, 'cars waiting to be dequeued', post_r
```

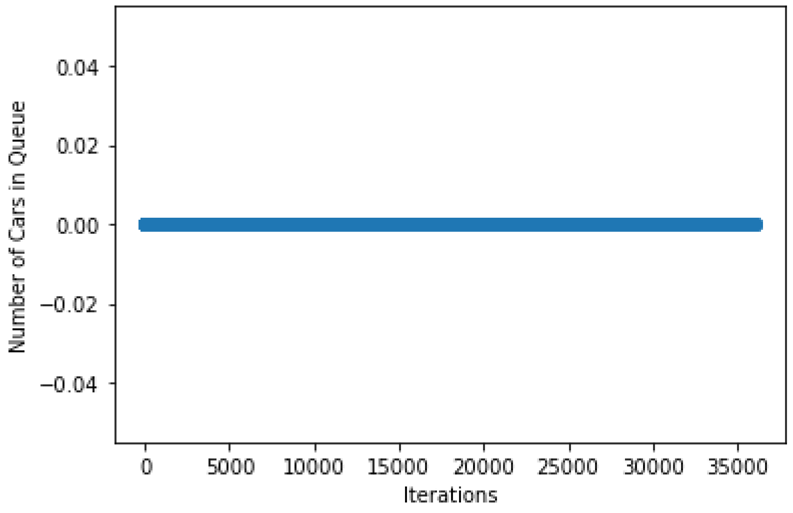
Out[17]:

```
Wall time: 5min 16s
```

In [18]:

```
plt.close()
plt.scatter(np.array(ramp_queue)[:,0], np.array(ramp_queue)[:,1])
plt.xlabel('Iterations')
plt.ylabel('Number of Cars in Queue')
plt.show()
print(np.array(ramp_queue)[:,1].mean())
```

Out[18]:



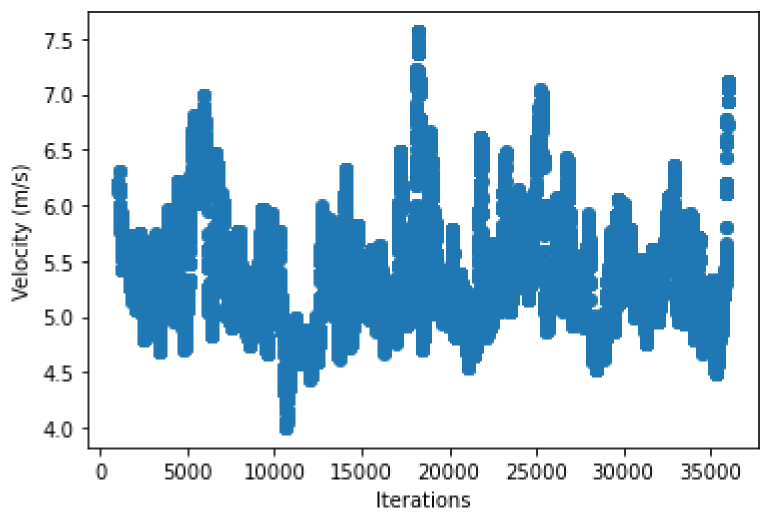
```
0.0
```

Here we observe that the ramp queue is constantly at 0 cars for the entire simulation, which is expected. With this simulation we want to show the effect of not applying a ramp metering policy as a baseline for later comparison.

In [19]:

```
vels = []
for i, state in enumerate(post_ramp_highway.road_state_vel):
    if len(state) == 0:
        continue
    state_vel = list(np.array(state)[:,1])
    vels.append((i, np.array(state_vel).mean()))
plt.close()
plt.scatter(np.array(vels)[1000:,0], np.array(vels)[1000:,1])
plt.xlabel('Iterations')
plt.ylabel('Velocity (m/s)')
plt.show()
np.array(vels)[:,1].mean()
```

Out[19]:



```
Out[19]: 5.563748972261837
```

It is observed that velocity oscillates around the mean of 5.56 m/s over the time steps, with a few spikes of relatively higher or lower velocities compared to the mean. From here, it is evidenced that at a slower velocity level that traffic is relatively dense.

ALINEA + simple example

ALINEA served as a ramp metering strategy that utilized an efficient and robust implementation for our simulation. It's main motivation lies in obtaining a desired freeway occupancy. The following equation was utilized for ALINEA in obtaining a ramp metering rate $r(t)$ for a control period t:

$$r(t) = r(t-1) + K_r[O - O_{out}(t)]$$

O represents the desired occupancy threshold, and $O_{out}(t)$ represents the measured occupancy at time t. K_r signifies a constant that acts as a regulatory parameter. Below is the function implemented to utilize ALINEA.

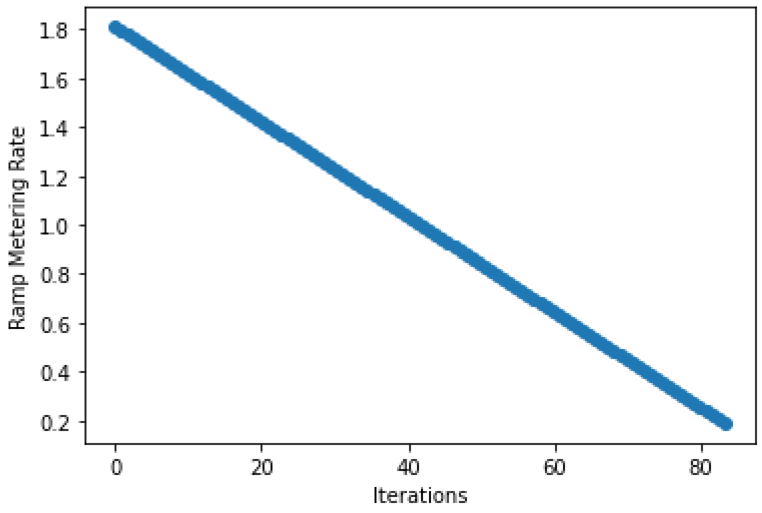
```
In [20]:
# r_prev : previous iteration's ramp metering rate in vehicles/second
# k_r : regulatory rate for smoothing. Recommended value of 70/3600 vehicles per second
# o_thres: desired occupancy threshold (vehicles/meter)
# o_out: measure occupancy in vehicles per meter

#return value: the ramp metering rate in vehicles/second
def ALINEA(r_prev, k_r, o_thres, o_out):
    r_new = r_prev + k_r*(o_thres - o_out)
    return r_new if r_new > 0 else 0
```

The ALINEA forumula is linear in nature. SO in turn, we should see a linear relation between the desired occupancy threshold, $O_{out}(t)$, and the ramp metering rate $r(t)$. An increase in the value of $O_{out}(t)$ should lead to a decrease in $r(t)$ as reflected in the formula, and vice versa. A plot can reflect this negative linear relationship.

```
In [21]:
o_thres = road_length / (LEN_CAR * S0_INIT * 2)
```

```
o_out = len(pre_ramp_highway.road_state[1000]) +len(post_ramp_highway.road_state[1000])

x = np.linspace(0, o_thres *2, 100)
y = [ALINEA(1, 70/3600, o_thres, pt) for pt in x]
plt.xlabel('Iterations')
plt.ylabel('Ramp Metering Rate')

plt.scatter(x, y);
```

Out[21]:



The behavior of this figure is expected and helps to confirm prior assumptions from looking at the analytical representation of ALINEA that it is linear. This in turn could potentially lead to a build up of of cars that are on the ramp during simulation due to the fact that the ramp metering rate decreases over time.

The simulation below will confirm this hypothesis by implementing ALINEA unto a simple 6-lane road with arbitrary flow data.

In [22]:

```
%%time
# Simulation parameters

num_lane = 6
road_length = 500 # 5 km
ramp_length = 30 # 30m
ramp_position = 100 # place ramp 100m into the road
r_prev = 1
dt = 0.1

k_r = 70 / 3600

highway_flow_in = 10
ramp_flow_in = 0.5

seed = 1111
np.random.seed(seed=seed)

cf = CarFactory()

pre_ramp_highway = Street(num_lane, ramp_position, cf, dt=dt)
post_ramp_highway = Street(num_lane, road_length - ramp_position, cf, dt=dt)
```

```
ramp = Street(1, 30, cf, dt=dt)

o_thres = 18.5

r_hist = []
ramp_queue = []

for i in tqdm(range(36000)):
    highway_q_in = np.random.normal(highway_flow_in, highway_flow_in/2 ** 1/5, (1))[0]
    ramp_q_in = np.random.normal(ramp_flow_in, ramp_flow_in/2 ** 1/5, (1))[0]

    pre_ramp_highway.update(highway_q_in, insert_on_last_lane=False)
    post_ramp_highway.vehicle_wait += pre_ramp_highway.vehicle_out
    pre_ramp_highway.vehicle_out -= pre_ramp_highway.vehicle_out

    ramp.update(ramp_q_in)

    try:
        b = np.array(post_ramp_highway.road_state[i-1])
        b = b[np.where(b[:,0] == max(b[:, 0]))]
        o_out = len(b)
    except:
        o_out = r_prev


    if i > 1000 and i % 50 == 0:
        r_new = ALINEA(r_prev, k_r, o_thres, o_out)
        r_hist.append((i, r_new))
        r_prev = r_new

    if ramp.vehicle_out >= r_prev * dt:
        ramp.vehicle_out -= r_prev * dt
        post_ramp_highway.vehicle_wait_ramp += r_prev * dt
    post_ramp_highway.update(0, insert_on_last_lane=True)

    ramp_queue.append((i, ramp.vehicle_out))


#     if i > 1000 and i % 1000 == 0:
#         post_ramp_highway.report()
#         print('cars at ramp', ramp.vehicle_out, 'cars waiting to be dequeued', post_r
```

Out[22]:

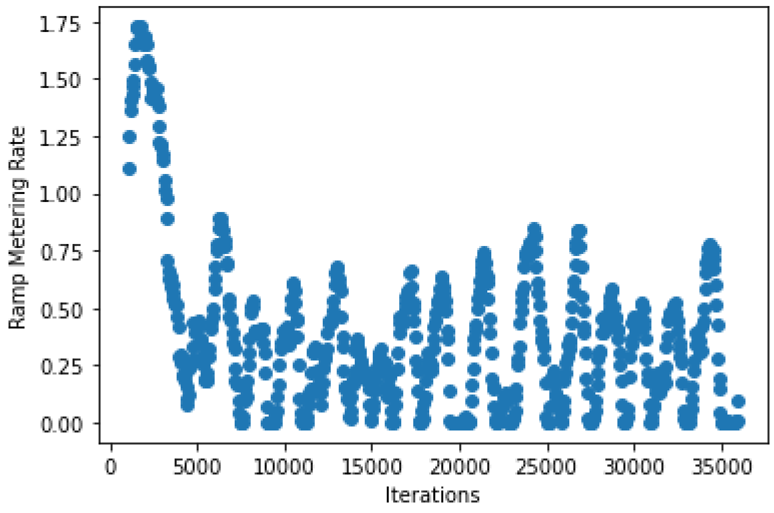
```
Wall time: 5min 45s
```

To look into analysis of the execution, the ramp metering rate over time was plotted through the simulation time. It can be observed that the ramp metering rate decreased overtime during the initial period of the simulation until it entered a steady oscillation at a lower ramp metering rate.

In [24]:

```
plt.close()
plt.scatter(np.array(r_hist)[:,0], np.array(r_hist)[:,1])
plt.xlabel('Iterations')
plt.ylabel('Ramp Metering Rate')
plt.show()
print(np.array(r_hist)[:,1].mean())
```

Out[24]:



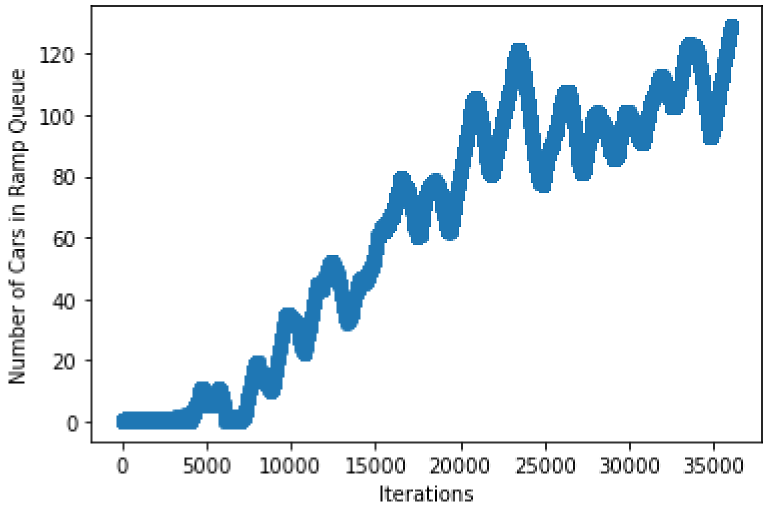
```
0.3758941344778255
```

Let's next analyze the number of vehicles in the ramp queue over time to observe the behavior.

In [25]:

```
plt.close()
plt.scatter(np.array(ramp_queue)[:,0], np.array(ramp_queue)[:,1])
plt.xlabel('Iterations')
plt.ylabel('Number of Cars in Ramp Queue')
plt.show()
print(np.array(ramp_queue)[:,1].mean())
```

Out[25]:



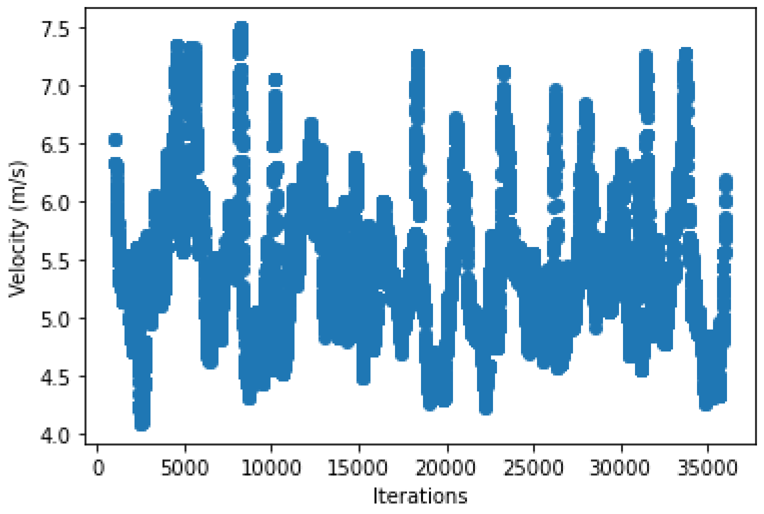
```
61.69712367669829
```

It is observed that right as the time the ramp metering rate decreased to an oscillating steady low state, the queue of the ramp increases in size significantly and continues to increase. This is in line with our theoretical assumption. As the arrival rate of cars lies in a normal distribution, the arrival rate shouldn't deviate from the amount expected significantly. With a lower ramp metering rate, it should be seen that the queue increases as vehicles continue to enter the ramp system and the exit rate from the system becomes lower.

In [26]:

```
vels = []
for i, state in enumerate(post_ramp_highway.road_state_vel):
    if len(state) == 0:
        continue
    state_vel = list(np.array(state)[:,1])
    vels.append((i, np.array(state_vel).mean()))

plt.close()
plt.scatter(np.array(vels)[1000:,0], np.array(vels)[1000:,1])
plt.xlabel('Iterations')
plt.ylabel('Velocity (m/s)')
plt.show()
np.array(vels)[:,1].mean()
```

Out[26]:



Out[26]:

```
5.633168752446347
```

The velocities of the vehicles on the highway system tend to follow a fluctuated rate centered at around the mean velocity of the vehicles of the system: 5.633168752446347 meters per second. At a relatively slow rate, it can be seen that the traffic is rather dense and congested. This was the intention based on our parameters, as we wanted to evaluate the buildup in the traffic from a dense crowd of vehicles. Hence, the value for the arrival flow rate into the highway is relatively high at 10 vehicles/second.

ALINEA with Ramp Queue Consideration + Simple Simulation Example

The following is a modified version of the ALINEA implementation where a new set of variables are introduced into ALINEA that depend on the number of cars in the ramp queue. This in turn will cause an increase in the ramp metering rate if the number of cars in the ramp queue is above a certain threshold that we define.

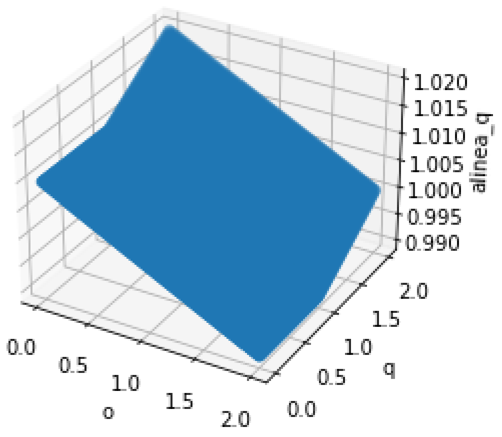
In [27]:

```
#return value: the ramp metering rate in vehicles/second
def ALINEA_q(r_prev, k_r, o_thres, o_out, q_thresh, q_out):
    q_weight = q_out - q_thresh # num cars over threshold, need to queue them
    if q_weight < 0:
        q_weight = 0
    r_new = r_prev + k_r*((o_thres - o_out) + q_weight) / 2
    return r_new if r_new > 0 else 0
```

```
dat = []
x_o = np.linspace(0, 2, 100)
x_q = np.linspace(0, 2, 100)

for o in x_o:
    for q in x_q:
        dat.append((o, q, ALINEA_q(1, 70/3600, 1, o, 1, q)))

dat = np.array(dat)

fig = plt.figure()
ax = plt.axes(projection='3d')
ax.scatter(dat[:,0], dat[:,1], dat[:,2])
ax.set_xlabel('o')
ax.set_ylabel('q')
ax.set_zlabel('alinea_q')
```

Out[27]:
```
Text(0.5, 0, 'alinea_q')
```

Out[27]:



From the 3-D visualization above, we can see that based on the ALINEA formula we present, despite being at full occupancy value in the threshold we defined, vehicles will still be let in through the ramp in order to alleviate problems that could occur with a substantially longer queue size. Let's now implement the modified ALINEA method onto the same simulation scenario as before.

In [28]:
```
%%time
# Simulation parameters

num_lane = 6
road_length = 500 # 5 km
ramp_length = 30 # 30m
ramp_position = 100 # place ramp 100m into the road
r_prev = 1
dt = 0.1

k_r = 70 / 3600

highway_flow_in = 10
ramp_flow_in = 0.5

seed = 1111
```

```
np.random.seed(seed=seed)

cf = CarFactory()

pre_ramp_highway = Street(num_lane, ramp_position, cf, dt=dt)
post_ramp_highway = Street(num_lane, road_length - ramp_position, cf, dt=dt)
ramp = Street(1, 30, cf, dt=dt)

o_thres = 18.5
q_thres = 5

r_hist = []
ramp_queue = []

for i in tqdm(range(36000)):
    highway_q_in = np.random.normal(highway_flow_in, highway_flow_in/2 ** 1/5, (1))[0]
    ramp_q_in = np.random.normal(ramp_flow_in, ramp_flow_in/2 ** 1/5, (1))[0]

    pre_ramp_highway.update(highway_q_in, insert_on_last_lane=False)
    post_ramp_highway.vehicle_wait += pre_ramp_highway.vehicle_out
    pre_ramp_highway.vehicle_out -= pre_ramp_highway.vehicle_out

    ramp.update(ramp_q_in)

    try:
        b = np.array(post_ramp_highway.road_state[i-1])
        b = b[np.where(b[:,0] == max(b[:, 0]))]
        o_out = len(b)
    except:
        o_out = r_prev


    if i > 1000 and i % 50 == 0:
        q_out = ramp.vehicle_out
        r_new = ALINEA_q(r_prev, k_r, o_thres, o_out, q_thres, q_out)
        r_hist.append((i, r_new))
        r_prev = r_new

    if ramp.vehicle_out >= r_prev * dt:
        ramp.vehicle_out -= r_prev * dt
        post_ramp_highway.vehicle_wait_ramp += r_prev * dt

    post_ramp_highway.update(0, insert_on_last_lane=True)

    ramp_queue.append((i, ramp.vehicle_out))


#     if i > 1000 and i % 1000 == 0:
#         post_ramp_highway.report()
#         print('cars at ramp', ramp.vehicle_out, 'cars waiting to be dequeued', post_r
```

Out[28]:

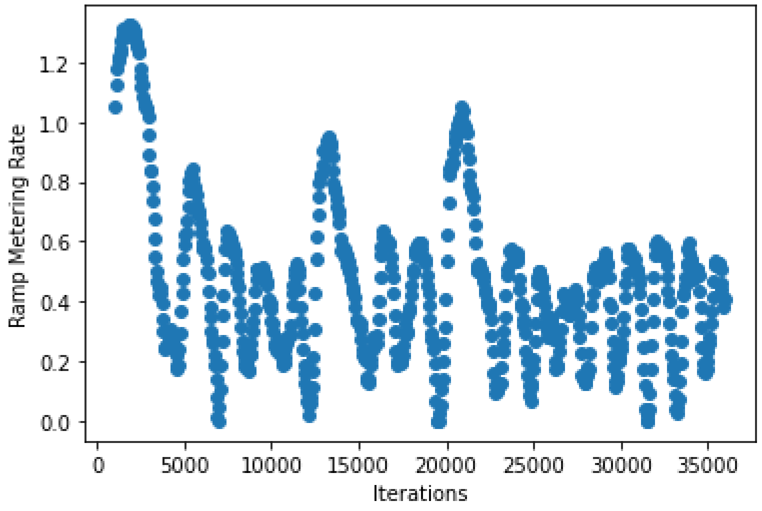
```
Wall time: 6min 1s
```

Now that the simulation is complete, let's now analyze the various parameters of focus and compare them to the original ALINEA method. The first parameter of focus is the ramp metering rate.

In [29]:

```
plt.close()
plt.scatter(np.array(r_hist)[:,0], np.array(r_hist)[:,1])
plt.xlabel('Iterations')
plt.ylabel('Ramp Metering Rate')
plt.show()
np.array(r_hist)[:,1].mean()
```

Out[29]:



Out[29]:

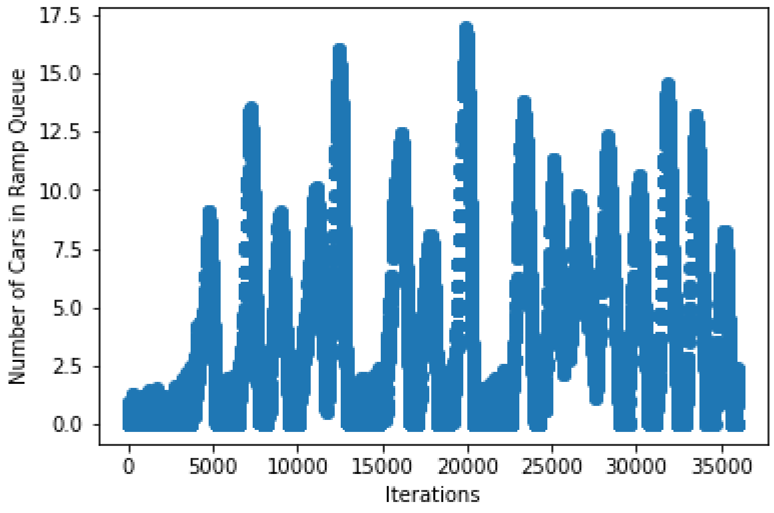
```
0.46594344044038616
```

As seen in the figure, the mean ramp metering rate increases by approximately 0.10. This is expected based on how the modified ALINEA method behaves. Let's next analyze the number of vehicles in the ramp queue to see if this modified method results in an improvement.

In [30]:

```
plt.close()
plt.scatter(np.array(ramp_queue)[:,0], np.array(ramp_queue)[:,1])
plt.xlabel('Iterations')
plt.ylabel('Number of Cars in Ramp Queue')
plt.show()
print(np.array(ramp_queue)[:,1].mean())
```

Out[30]:

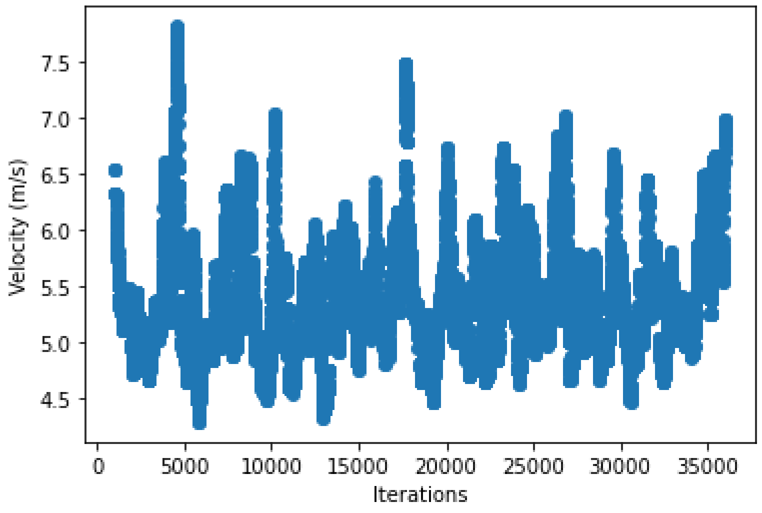


```
3.8945050179282017
```

This oscillatory behavior is an improvement from the original ALINEA method in that not only has the mean number of cars in queue decreased significantly but the ramp metering rate's dependency on the number of cars in the ramp queue allows for a more dynamic behavior that works to address the buildup that comes about in the ramp queue. Let's next analyze the velocity of the cars within this simulation with the modified ALINEA method.

In [31]:
```
vels = []
for i, state in enumerate(post_ramp_highway.road_state_vel):
    if len(state) == 0:
        continue
    state_vel = list(np.array(state)[:,1])
    vels.append((i, np.array(state_vel).mean()))
plt.close()
plt.scatter(np.array(vels)[1000:,0], np.array(vels)[1000:,1])
plt.xlabel('Iterations')
plt.ylabel('Velocity (m/s)')
plt.show()
np.array(vels)[:,1].mean()
```

Out[31]:



Out[31]:
```
5.624896329655928
```

As one can see from the figure above and the mean velocity, there is a slight decrease in velocity when implementing this modified method. However, this marginal decrease in velocity combined with the significant decrease in the number of cars in the ramp queue serves as evidence and confirmation that this modified ramp metering method has better performance when evaluating the flow of traffic. The modified ALINEA method will be utilized in the simulation for the real world data to come.

Observations and Discussion

Overall, the modified ALINEA method performs better on this simple example than the original ALINEA method when taking into account the number of cars in the ramp queue and the average velocity of cars in traffic. This is due to the fact that the modified ALINEA method has dynamic behavior that is dependent on the number of cars in the ramp queue. Thus when implementing these two ramp metering methods on the interchange, we expect that the modified ALINEA method will perform better.

Data Collection

The data collection process consisted of 3 key steps:

- Selecting the site to collect either volume or speed data from the site.
- Selecting a time-frame to retrieve data from depending on if the site is an active site that has data over a long period of time or a temporary site that has data over a 3-day period.
- Convert the file from ".xls" format to ".csv" to then parse through the data and collect an average value for either volume or speed for every hour

The main challenge that arose during this process is that ".xls" files are not very friendly programming-wise. This caused the need for a manual conversion of every file to ".csv" format. Once that process is over, a script is ran that sifts through the file and develops an average value for either speed of traffic volume depending on the file for each hour. In addition to splitting the data by the hour, the data is also split by direction. For example, a file can be split by the traffic volume in all directions, southbound, and northbound. The datatype is presented via double nested dictionary where the first key is the file name that identifies the site and whether it is a speed or traffic volume file. Once a dictionary is accessed via the filename key, the next keys are to identify the direction. The the values within that key are numpy arrays that represent either the flow or speed for each hour. The sites that data was retrieved from were specifically chosen in order to have an idea of traffic flow and speed in all four directions that come in and out of the intersection of I-75 and I-285. In addition, to analyzing the flow in and out of the intersection, they were also chosen to analyze the flow through the intersection as a function of the varying the number of cars that enter the road via the ramp meter. This allows us to vary how many cars can enter the ramp meter at a certain time and analyze how this affects flow and speed throughout the intersection. Below is the implementation of this data parsing.

In [32]:

```python
def data_parser(filename, by_sec=True):
    with open(filename) as csv_file:
        csv_reader = csv.reader(csv_file, delimiter=',')
        keywords = ['direction', 'West','North','South','East','NB','SB','EB','WB']
        names = []
        data = {}
        total = []
        it = 0
        once = 0
        drop = 'full'
        for row in csv_reader:
            x = row[0]
            if any(word in x for word in keywords):
                names.append(x)
            if '00:00' in x:
                vec = []
                for i in range(1,8):
                    if row[i] != '':
                        vec.append(float(row[i]))
                if vec != []:
                    total.append(np.mean(vec))
                elif once == 0:
                    drop = it
                    once = 1
                if '00:00:00' in x and 'speed' in filename:
                    it += 1
        total = np.array(total)
        if drop != 'full':
            del names[drop]
        total = np.array_split(total,len(names))

        for ndx,name in enumerate(names):
            data[name] = total[ndx]
        return data
```

In [33]:

```
files = ['traffic_files/' + f for f in os.listdir('traffic_files/') if not f.startswith

data = {}
for filename in files:
    file_data = data_parser(filename)
    data[filename.split('/')[1]] = file_data

checker = []
for key in data:
    for k in data[key]:
        checker.append(len(data[key][k]))
if len(set(checker)) == 1:
    print('The data is good!')

else:
    print('The data has instances where the volume/speed for a direction does not have
# os.chdir('/home/user/')
```

```
The data is good!
```

Input Analysis

The following code below sets the stage for the flow rates at each direction of the interchange. This is done through using the data provided by Georgia Department of Transportation where the flow rate selected for each direction of the interchange is at 5pm local time. Once the flow rates are extracted, they are then used to calculate the interchange probability so that it can be inputed into the model.

In [34]:

```
i285_south_in = data['067-2373_feb_2014.csv']['All Northbound'][17] / 3600
i285_south_out = data['067-2373_feb_2014.csv']['All Southbound'][17] / 3600
i285_north_in = data['121-5546_feb_2020.csv']['All Westbound'][17] / 3600
i285_north_out = data['121-5546_feb_2020.csv']['All Eastbound'][17] / 3600
i75_south_in = data['121-6370_feb_2020.csv']['All Northbound'][17] / 3600
i75_south_out = data['121-6370_feb_2020.csv']['All Southbound'][17] / 3600
i75_north_in = data['067-2738_feb_2016.csv']['All Southbound'][17] / 3600
i75_north_out = data['067-2738_feb_2016.csv']['All Northbound'][17] / 3600
i285_south_i75_north = data['067-9922_feb_2020.csv']['All Eastbound'][17] / 3600
i285_south_i75_south = data['067-r081_july_2020.csv']['All Eastbound'][17] / 3600
i285_south_i285_north = i285_south_in - i285_south_i75_south - i285_south_i75_north
i75_north_i285_north = data['067-r601_oct_2020.csv']['All Southbound'][17] / 3600
i75_north_i285_south = data['067-r712_oct_2019.csv']['All Northbound'][17] / 3600
i75_north_i75_south = i75_north_in - i75_north_i285_north - i75_north_i285_south
```

```
i75_south_i285_north = data['067-r630_oct_2020.csv']['All Southbound'][17] / 3600
i75_south_i285_south = data['067_r003_oct_2020.csv']['All Northbound'][17] / 3600
i75_south_i75_north = i75_south_in - i75_south_i285_north - i75_south_i285_south
i285_north_i285_south = i285_south_out - i75_north_i285_south - i75_south_i285_south
i285_south_north_i75_north = data['067-9006_feb_2020.csv']['All Northbound'][17] / 3600
i285_north_i75_north = i285_south_north_i75_north - i285_south_i75_north
i285_north_i75_south = i285_north_in - i285_north_i75_north - i285_north_i285_south

cumberland_ramp_in =  data['067_9109_feb_2020.csv']['All Eastbound'][17] / 3600
i75_south_in_ramp_in = 0.5
i75_north_in_ramp_in = 0.5
```

In [35]:
```
# Calculate interchange probability based on real-world densities.

# i75_north_out
i285_south_i75_north_prop = i285_south_i75_north / (i285_south_i75_north + i75_south_i7
i75_south_i75_north_prop = i75_south_i75_north / (i285_south_i75_north + i75_south_i75_
i285_north_i75_north_prop = i285_north_i75_north / (i285_south_i75_north + i75_south_i7
print('i75_north_out', i285_south_i75_north_prop, i75_south_i75_north_prop, i285_north_

# i75_south_out
i285_north_i75_south_prop = i285_north_i75_south / (i285_north_i75_south + i75_north_i7
i75_north_i75_south_prop = i75_north_i75_south / (i285_north_i75_south + i75_north_i75_
i285_south_i75_south_prop = i285_south_i75_south / (i285_north_i75_south + i75_north_i7
print('i75_south_out', i285_north_i75_south_prop, i75_north_i75_south_prop, i285_south_

# i285_north_out
i285_south_i285_north_prop = i285_south_i285_north / (i285_south_i285_north + i75_south
i75_south_i285_north_prop = i75_south_i285_north / (i285_south_i285_north + i75_south_i
i75_north_i285_north_prop = i75_north_i285_north / (i285_south_i285_north + i75_south_i
print('i285_north_out', i285_south_i285_north_prop, i75_south_i285_north_prop, i75_nort

# i285_south_out
i75_north_i285_south_prop = i75_north_i285_south / (i75_north_i285_south + i75_south_i2
i75_south_i285_south_prop = i75_south_i285_south / (i75_north_i285_south + i75_south_i2
i285_north_i285_south_prop = i285_north_i285_south / (i75_north_i285_south + i75_south_
print('i285_south_out', i75_north_i285_south_prop, i75_south_i285_south_prop, i285_nort

#cumberland_ramp_out
cumberland_ramp_i285_north_prop = i285_north_out / (i285_north_out + i75_south_out + i7
cumberland_ramp_i75_north_prop = i75_north_out / (i285_north_out + i75_south_out + i75_
cumberland_ramp_i75_south_prop = i75_south_out / (i285_north_out + i75_south_out + i75_
print('cumberland_ramp_out', cumberland_ramp_i285_north_prop, cumberland_ramp_i75_north
```

```
i75_north_out 0.08142744358831558 0.8323163827977257 0.08625617361395865
i75_south_out 0.3508656595058194 0.5669064188122538 0.0822279216819269
i285_north_out 0.7158097289105574 0.08751832022253463 0.19667195086690797
i285_south_out 0.35271011580919703 0.22492117600138462 0.4223687081894184
cumberland_ramp_out 0.29198258366405505 0.4277721904716021 0.2802452258643429
```

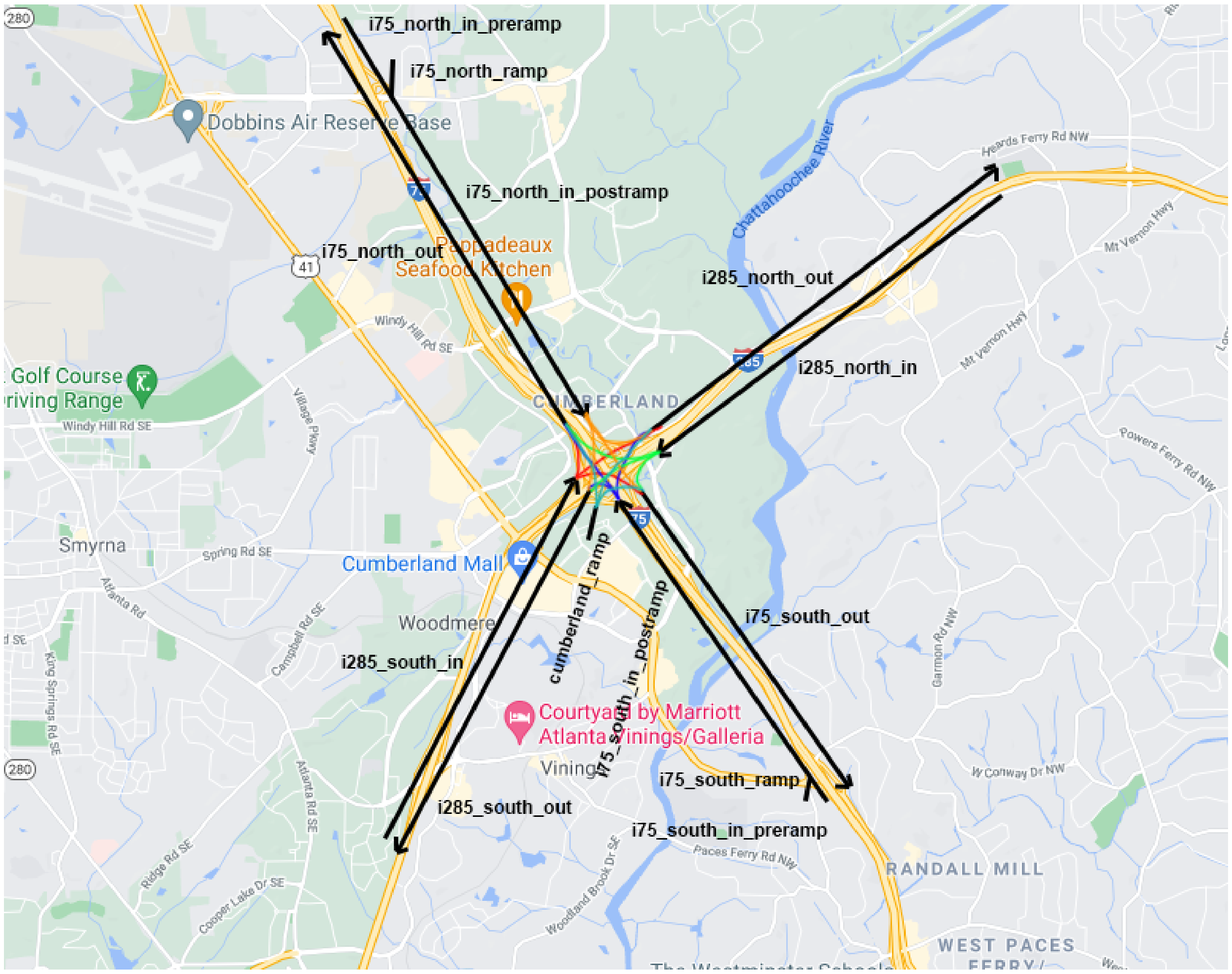
I-75/I-285 Interchange

The following simulations implement the interchange of I-75 and I-285. The first simulation merely just sets up the interchange using real world measured street lengths with no policy. From there the next simulation implements 3 ramps on the interchange and incoporates the ALINEA ramp metering method where the 3 ramps are located on I-75 Northbound , I-75 Southbound , and I-285 Northbound. The I-75 Northbound and I-75 Southbound ramps simply enter the cars in the respective highways. The I-285 Northbound ramp (Cumberland Ramp) however allows cars to enter into I-75 South, I-75 North, and I-285 North. The results from this simulation are then compared to a simulation that has the same scenario but uses the modified ALINEA method instead of the original ALINEA method.

Base Simulation: No Policy

Given the modular nature of our road simulator, it is quite extensible to very complex systems. Here we recreate from real measurements the general structure of the i75/i285 interchange. Becuase we are studying the effects of ramp metering on flow, we decided to take some simplifications on the amount of roads to simulate since we found that a completely accurate simulation would require 100+ road segments. So, we simplify the interchange into a form of black box, that is, each road segment prior to the interchange splits some of it out flow into the other 2 directions of the crossing highway, and preserves some of its flow to continue on through the interchange. We use numerically derived proportions here to ensure we stay true to the real world.

Below is a visualization of the implementation of the interchange in a simplistic way. We create 13 road segments and treat the interchange as a black box, that is, the outflow of the inputs is directly added to the outflow of the corresponding roads that cars could potentially travel. This allows us to create a simpler simulation while still staying true to the real world.



In [36]:
```
cf = CarFactory()
dt = 0.1

i285_south_in_road = Street(6, 3911, cf, dt=dt)
i285_south_out_road = Street(6, 3911, cf, dt=dt)

i285_north_in_road = Street(6, 1543, cf, dt = dt)
i285_north_out_road = Street(6, 1543, cf, dt = dt)

i75_south_in_preramp = Street(6, 568, cf, dt=dt)
i75_south_in_postramp = Street(6, 320, cf, dt=dt)
i75_south_in_ramp = Street(1, 50, cf, dt=dt)
i75_south_out_road = Street(6, 888, cf, dt = dt)

i75_north_in_preramp = Street(6,1018, cf, dt = dt)
i75_north_in_postramp = Street(6, 752, cf, dt = dt)
i75_north_in_ramp = Street(1, 50, cf, dt = dt)
i75_north_out_road = Street(6, 1770, cf, dt = dt)

cumberland_ramp = Street(1, 50, cf, dt = dt)
```

In [37]:
```
%%time

# Simulation time
for i in tqdm(range(36000)):
    # hw input
    i285_south_q_in = np.random.normal(i285_south_in, i285_south_in/2 ** 1/5, (1))[0]
    i285_north_q_in = np.random.normal(i285_north_in, i285_north_in/2 ** 1/5, (1))[0]
    i75_south_q_in = np.random.normal(i75_south_in, i75_south_in/2 ** 1/5, (1))[0]
    i75_north_q_in = np.random.normal(i75_north_in, i75_north_in/2 ** 1/5, (1))[0]

    # ramp input
    i75_south_in_ramp_q_in = np.random.normal(cumberland_ramp_in, cumberland_ramp_in/2
    i75_north_in_ramp_q_in = np.random.normal(i75_south_in_ramp_in, i75_south_in_ramp_i
    cumberland_ramp_q_in = np.random.normal(i75_north_in_ramp_in, i75_north_in_ramp_in/


    # i285 south
    i285_south_in_road.update(i285_south_q_in, insert_on_last_lane=False)

    # i285 north
    i285_north_in_road.update(i285_north_q_in, insert_on_last_lane=False)

    # i75 south
    i75_south_in_preramp.update(i75_south_q_in, insert_on_last_lane=False)
    i75_south_in_postramp.vehicle_wait += i75_south_in_preramp.vehicle_out
    i75_south_in_preramp.vehicle_out -= i75_south_in_preramp.vehicle_out

    i75_south_in_ramp.update(i75_south_in_ramp_q_in)
    i75_south_in_postramp.vehicle_wait_ramp += i75_south_in_ramp.vehicle_out
    i75_south_in_ramp.vehicle_out -= i75_south_in_ramp.vehicle_out

    i75_south_in_postramp.update(0, insert_on_last_lane=True)

    # i75 north
    i75_north_in_preramp.update(i75_north_q_in, insert_on_last_lane=False)
    i75_north_in_postramp.vehicle_wait += i75_north_in_preramp.vehicle_out
    i75_north_in_preramp.vehicle_out -= i75_north_in_preramp.vehicle_out

    i75_north_in_ramp.update(i75_north_in_ramp_q_in)
    i75_north_in_postramp.vehicle_wait_ramp += i75_north_in_ramp.vehicle_out
    i75_north_in_ramp.vehicle_out -= i75_north_in_ramp.vehicle_out

    i75_north_in_postramp.update(0, insert_on_last_lane=True)

    # cum ramp
    cumberland_ramp.update(cumberland_ramp_q_in)


    # i285 north out
    i285_north_out_road.vehicle_wait += i75_north_in_postramp.vehicle_out * i75_north_i
    i75_north_in_postramp.vehicle_out -= i75_north_in_postramp.vehicle_out * i75_north_

    i285_north_out_road.vehicle_wait += i75_south_in_postramp.vehicle_out * i75_south_i
    i75_south_in_postramp.vehicle_out -= i75_south_in_postramp.vehicle_out * i75_south_

    i285_north_out_road.vehicle_wait += i285_south_in_road.vehicle_out * i285_south_i28
    i285_south_in_road.vehicle_out -= i285_south_in_road.vehicle_out * i285_south_i285_

    # i285 south out
    i285_south_out_road.vehicle_wait += i75_north_in_postramp.vehicle_out * i75_north_i
    i75_north_in_postramp.vehicle_out -= i75_north_in_postramp.vehicle_out * i75_north_

    i285_south_out_road.vehicle_wait += i75_south_in_postramp.vehicle_out * i75_south_i
    i75_south_in_postramp.vehicle_out -= i75_south_in_postramp.vehicle_out * i75_south_

    i285_south_out_road.vehicle_wait += i285_north_in_road.vehicle_out * i285_north_i28
    i285_north_in_road.vehicle_out -= i285_north_in_road.vehicle_out * i285_north_i285_

    # i75 north out
    i75_north_out_road.vehicle_wait += i285_north_in_road.vehicle_out * i285_north_i75_
    i285_north_in_road.vehicle_out -= i285_north_in_road.vehicle_out * i285_north_i75_n

    i75_north_out_road.vehicle_wait += i75_south_in_postramp.vehicle_out * i75_south_i7
    i75_south_in_postramp.vehicle_out -= i75_south_in_postramp.vehicle_out * i75_south_

    i75_north_out_road.vehicle_wait += i285_south_in_road.vehicle_out * i285_south_i75_
    i285_south_in_road.vehicle_out -= i285_south_in_road.vehicle_out * i285_south_i75_n

    # i75 south out
    i75_south_out_road.vehicle_wait += i285_north_in_road.vehicle_out * i285_north_i75_
    i285_north_in_road.vehicle_out -= i285_north_in_road.vehicle_out * i285_north_i75_s

    i75_south_out_road.vehicle_wait += i75_north_in_postramp.vehicle_out * i75_north_i7
    i75_north_in_postramp.vehicle_out -= i75_north_in_postramp.vehicle_out * i75_north_

    i75_south_out_road.vehicle_wait += i285_south_in_road.vehicle_out * i285_south_i75_
    i285_south_in_road.vehicle_out -= i285_south_in_road.vehicle_out * i285_south_i75_s

    # cumberland ramping

    i75_north_out_road.vehicle_wait += cumberland_ramp.vehicle_out * cumberland_ramp_i7
    cumberland_ramp.vehicle_out -= cumberland_ramp.vehicle_out * cumberland_ramp_i75_no

    i75_south_out_road.vehicle_wait += cumberland_ramp.vehicle_out * cumberland_ramp_i7
    cumberland_ramp.vehicle_out -= cumberland_ramp.vehicle_out * cumberland_ramp_i75_so

    i285_north_out_road.vehicle_wait += cumberland_ramp.vehicle_out * cumberland_ramp_i
    cumberland_ramp.vehicle_out -= cumberland_ramp.vehicle_out * cumberland_ramp_i285_n


    # remaining flow

    i285_south_out_road.vehicle_wait += i285_north_in_road.vehicle_out
    i285_north_in_road.vehicle_out -= i285_north_in_road.vehicle_out

    i285_north_out_road.vehicle_wait += i285_south_in_road.vehicle_out
    i285_south_in_road.vehicle_out -= i285_south_in_road.vehicle_out

    i75_south_out_road.vehicle_wait += i75_north_in_postramp.vehicle_out
    i75_north_in_postramp.vehicle_out -= i75_north_in_postramp.vehicle_out

    i75_north_out_road.vehicle_wait += i75_south_in_postramp.vehicle_out
    i75_south_in_postramp.vehicle_out -= i75_south_in_postramp.vehicle_out

    # update out roads

    i285_south_out_road.update(0, insert_on_last_lane=False)
    i285_north_out_road.update(0, insert_on_last_lane=False)
    i75_south_out_road.update(0, insert_on_last_lane=False)
    i75_north_out_road.update(0, insert_on_last_lane=False)



#     if i > 1000 and i % 1000 == 0:
#         i285_south_out_road.report()
#         i285_north_out_road.report()
#         i75_south_out_road.report()
#         i75_north_out_road.report()
```

Out[37]:

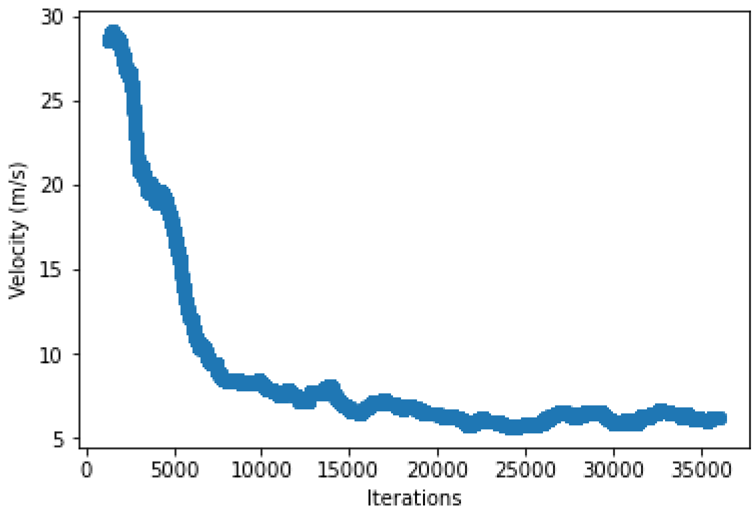
```
Wall time: 2h 7s
```

Let's now evaluate the velocities of the cars in all 4 directions at the interchange.

In [38]:

```
vels = []
for i, state in enumerate(i285_south_out_road.road_state_vel):
    if len(state) == 0:
        continue

    state_vel = list(np.array(state)[:,1])
    vels.append((i, np.array(state_vel).mean()))
plt.close()
plt.scatter(np.array(vels)[1000:,0], np.array(vels)[1000:,1])
plt.xlabel('Iterations')
plt.ylabel('Velocity (m/s)')
plt.show()
np.array(vels)[:,1].mean()
```

Out[38]:



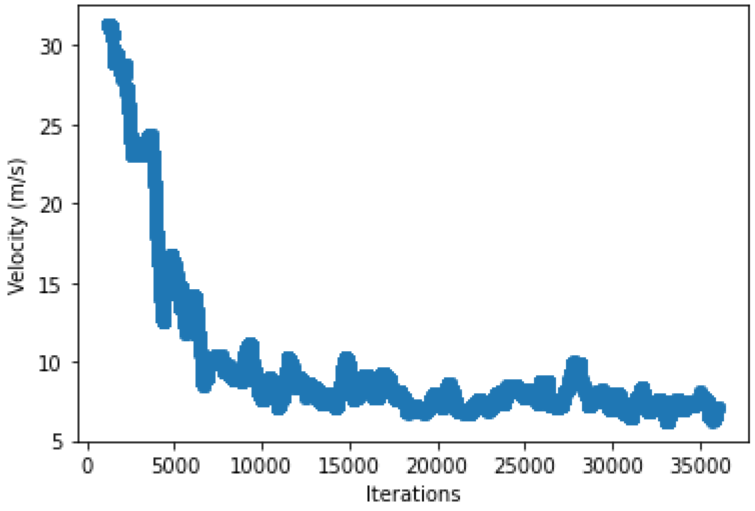
Out[38]:

```
9.356918872637648
```

In [39]:

```
vels = []
for i, state in enumerate(i285_north_out_road.road_state_vel):
    if len(state) == 0:
        continue

    state_vel = list(np.array(state)[:,1])
    vels.append((i, np.array(state_vel).mean()))
plt.close()
plt.scatter(np.array(vels)[1000:,0], np.array(vels)[1000:,1])
plt.xlabel('Iterations')
plt.ylabel('Velocity (m/s)')
plt.show()
np.array(vels)[:,1].mean()
```

Out[39]:



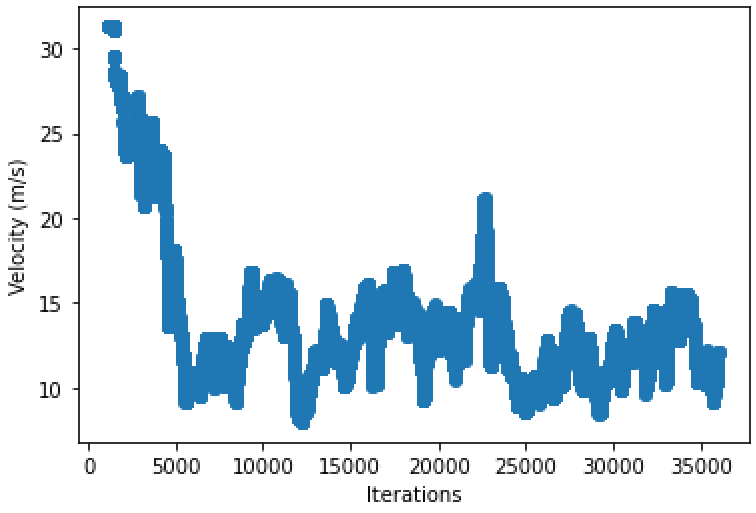
Out[39]:

```
10.486369814724696
```

In [40]:

```
vels = []
for i, state in enumerate(i75_south_out_road.road_state_vel):
    if len(state) == 0:
        continue

    state_vel = list(np.array(state)[:,1])
    vels.append((i, np.array(state_vel).mean()))
plt.close()
plt.scatter(np.array(vels)[1000:,0], np.array(vels)[1000:,1])
plt.xlabel('Iterations')
plt.ylabel('Velocity (m/s)')
plt.show()
np.array(vels)[:,1].mean()
```

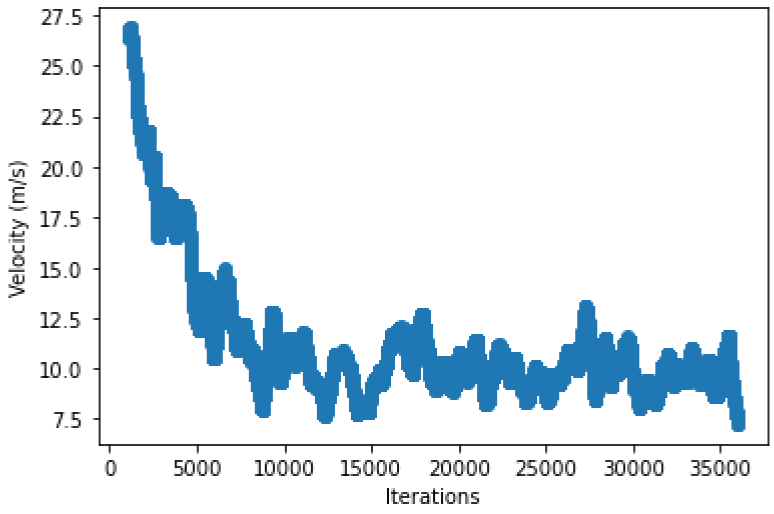
Out[40]:



```
Out[40]: 14.29628234346243
```

```
In [41]:
vels = []
for i, state in enumerate(i75_north_out_road.road_state_vel):
    if len(state) == 0:
        continue

    state_vel = list(np.array(state)[:,1])
    vels.append((i, np.array(state_vel).mean()))
plt.close()
plt.scatter(np.array(vels)[1000:,0], np.array(vels)[1000:,1])
plt.xlabel('Iterations')
plt.ylabel('Velocity (m/s)')
plt.show()
np.array(vels)[:,1].mean()
```

Out[41]:



```
Out[41]: 11.71392498743949
```

These velocities serve as a baseline for when the ramp metering policies are implemented to then compare the predicted improvement that comes about when placing ramp meters into the interchange.

Simulate with ALINEA

Below demonstrates the interchange simulation utilizing the original ALINEA formulation, which is applied to the three ramp meters.

In [42]:

```
# Params
cf = CarFactory()
dt = 0.1

# Road Definitions
i285_south_in_road = Street(6, 3911, cf, dt=dt)
i285_south_out_road = Street(6, 3911, cf, dt=dt)

i285_north_in_road = Street(6, 1543, cf, dt = dt)
i285_north_out_road = Street(6, 1543, cf, dt = dt)

i75_south_in_preramp = Street(6, 568, cf, dt=dt)
i75_south_in_postramp = Street(6, 320, cf, dt=dt)
i75_south_in_ramp = Street(1, 50, cf, dt=dt)
i75_south_out_road = Street(6, 888, cf, dt = dt)

i75_north_in_preramp = Street(6,1018, cf, dt = dt)
i75_north_in_postramp = Street(6, 752, cf, dt = dt)
i75_north_in_ramp = Street(1, 50, cf, dt = dt)
i75_north_out_road = Street(6, 1770, cf, dt = dt)

cumberland_ramp = Street(1, 50, cf, dt = dt)


# ALINEA Params
k_r = 70 / 3600

i75_south_o_thres = 18.5
i75_south_r_prev = 1
i75_south_r_hist = []
i75_south_ramp_queue = []

i75_north_o_thres = 18.5
i75_north_r_prev = 1
i75_north_r_hist = []
i75_north_ramp_queue = []

cumberland_o_thres = 18.5
cumberland_r_prev = 1
cumberland_r_hist = []
cumberland_ramp_queue = []


seed = 1111
np.random.seed(seed=seed)
```

In [43]:

```
%%time

# Simulation time
for i in tqdm(range(36000)):
    # hw input
    i285_south_q_in = np.random.normal(i285_south_in, i285_south_in/2 ** 1/5, (1))[0]
    i285_north_q_in = np.random.normal(i285_north_in, i285_north_in/2 ** 1/5, (1))[0]
    i75_south_q_in = np.random.normal(i75_south_in, i75_south_in/2 ** 1/5, (1))[0]
    i75_north_q_in = np.random.normal(i75_north_in, i75_north_in/2 ** 1/5, (1))[0]

    # ramp input
    i75_south_in_ramp_q_in = np.random.normal(cumberland_ramp_in, cumberland_ramp_in/2
    i75_north_in_ramp_q_in = np.random.normal(i75_south_in_ramp_in, i75_south_in_ramp_i
    cumberland_ramp_q_in = np.random.normal(i75_north_in_ramp_in, i75_north_in_ramp_in/


    # i285 south
    i285_south_in_road.update(i285_south_q_in, insert_on_last_lane=False)

    # i285 north
    i285_north_in_road.update(i285_north_q_in, insert_on_last_lane=False)

    # i75 south
    i75_south_in_preramp.update(i75_south_q_in, insert_on_last_lane=False)
    i75_south_in_postramp.vehicle_wait += i75_south_in_preramp.vehicle_out
    i75_south_in_preramp.vehicle_out -= i75_south_in_preramp.vehicle_out

    i75_south_in_ramp.update(i75_south_in_ramp_q_in)

    try:
        b = np.array(i75_south_in_postramp.road_state[i-1])
        b = b[np.where(b[:,0] == max(b[:, 0]))]
        o_out = len(b)
    except:
        o_out = i75_south_r_prev


    if i > 1000 and i % 50 == 0:
        i75_south_r_prev = ALINEA(i75_south_r_prev, k_r, i75_south_o_thres, o_out)
        i75_south_r_hist.append((i, i75_south_r_prev))


    if i75_south_in_ramp.vehicle_out >= i75_south_r_prev * dt:
        i75_south_in_ramp.vehicle_out -= i75_south_r_prev * dt
        i75_south_in_postramp.vehicle_wait_ramp += i75_south_r_prev * dt

    i75_south_in_postramp.update(0, insert_on_last_lane=True)

    i75_south_ramp_queue.append((i, i75_south_in_ramp.vehicle_out))

    # i75 north
    i75_north_in_preramp.update(i75_north_q_in, insert_on_last_lane=False)
    i75_north_in_postramp.vehicle_wait += i75_north_in_preramp.vehicle_out
    i75_north_in_preramp.vehicle_out -= i75_north_in_preramp.vehicle_out

    i75_north_in_ramp.update(i75_north_in_ramp_q_in)

    try:
        b = np.array(i75_north_in_postramp.road_state[i-1])
        b = b[np.where(b[:,0] == max(b[:, 0]))]
        o_out = len(b)
```

```
except:
    o_out = i75_north_r_prev


if i > 1000 and i % 50 == 0:
    i75_north_r_prev = ALINEA(i75_north_r_prev, k_r, i75_north_o_thres, o_out)
    i75_north_r_hist.append((i, i75_north_r_prev))


if i75_north_in_ramp.vehicle_out >= i75_north_r_prev * dt:
    i75_north_in_ramp.vehicle_out -= i75_north_r_prev * dt
    i75_north_in_postramp.vehicle_wait_ramp += i75_north_r_prev * dt

i75_north_in_postramp.update(0, insert_on_last_lane=True)

i75_north_ramp_queue.append((i, i75_north_in_ramp.vehicle_out))


# cum ramp
cumberland_ramp.update(cumberland_ramp_q_in)

try:
    a = np.array(i75_north_out_road.road_state[i-1])
    a = a[np.where(a[:,0] == max(a[:, 0]))]

    b = np.array(i75_south_out_road.road_state[i-1])
    b = b[np.where(b[:,0] == max(b[:, 0]))]

    c = np.array(i285_north_out_road.road_state[i-1])
    c = a[np.where(c[:,0] == max(c[:, 0]))]

    o_out = cumberland_ramp_i75_north_prop * len(a) + cumberland_ramp_i75_south_pro
except:
    o_out = cumberland_r_prev


if i > 1000 and i % 50 == 0:
    cumberland_r_prev = ALINEA(cumberland_r_prev, k_r, cumberland_o_thres, o_out)
    cumberland_r_hist.append((i, cumberland_r_prev))


# i285 north out
i285_north_out_road.vehicle_wait += i75_north_in_postramp.vehicle_out * i75_north_i
i75_north_in_postramp.vehicle_out -= i75_north_in_postramp.vehicle_out * i75_north_

i285_north_out_road.vehicle_wait += i75_south_in_postramp.vehicle_out * i75_south_i
i75_south_in_postramp.vehicle_out -= i75_south_in_postramp.vehicle_out * i75_south_

i285_north_out_road.vehicle_wait += i285_south_in_road.vehicle_out * i285_south_i28
i285_south_in_road.vehicle_out -= i285_south_in_road.vehicle_out * i285_south_i285_

# i285 south out
i285_south_out_road.vehicle_wait += i75_north_in_postramp.vehicle_out * i75_north_i
i75_north_in_postramp.vehicle_out -= i75_north_in_postramp.vehicle_out * i75_north_

i285_south_out_road.vehicle_wait += i75_south_in_postramp.vehicle_out * i75_south_i
i75_south_in_postramp.vehicle_out -= i75_south_in_postramp.vehicle_out * i75_south_
```

```
    i285_south_out_road.vehicle_wait += i285_north_in_road.vehicle_out * i285_north_i28
    i285_north_in_road.vehicle_out -= i285_north_in_road.vehicle_out * i285_north_i285_

    # i75 north out
    i75_north_out_road.vehicle_wait += i285_north_in_road.vehicle_out * i285_north_i75_
    i285_north_in_road.vehicle_out -= i285_north_in_road.vehicle_out * i285_north_i75_n

    i75_north_out_road.vehicle_wait += i75_south_in_postramp.vehicle_out * i75_south_i7
    i75_south_in_postramp.vehicle_out -= i75_south_in_postramp.vehicle_out * i75_south_

    i75_north_out_road.vehicle_wait += i285_south_in_road.vehicle_out * i285_south_i75_
    i285_south_in_road.vehicle_out -= i285_south_in_road.vehicle_out * i285_south_i75_n

    # i75 south out
    i75_south_out_road.vehicle_wait += i285_north_in_road.vehicle_out * i285_north_i75_
    i285_north_in_road.vehicle_out -= i285_north_in_road.vehicle_out * i285_north_i75_s

    i75_south_out_road.vehicle_wait += i75_north_in_postramp.vehicle_out * i75_north_i7
    i75_north_in_postramp.vehicle_out -= i75_north_in_postramp.vehicle_out * i75_north_

    i75_south_out_road.vehicle_wait += i285_south_in_road.vehicle_out * i285_south_i75_
    i285_south_in_road.vehicle_out -= i285_south_in_road.vehicle_out * i285_south_i75_s

    # cumberland ramping
    if cumberland_ramp.vehicle_out >= cumberland_r_prev * dt:
        cumberland_ramp.vehicle_out -= cumberland_r_prev * dt
        i75_north_out_road.vehicle_wait_ramp += cumberland_r_prev * dt * cumberland_ram
        i75_south_out_road.vehicle_wait_ramp += cumberland_r_prev * dt * cumberland_ram
        i285_north_out_road.vehicle_wait_ramp += cumberland_r_prev * dt * cumberland_ra

    cumberland_ramp_queue.append((i, cumberland_ramp.vehicle_out))

    # remaining flow

    i285_south_out_road.vehicle_wait += i285_north_in_road.vehicle_out
    i285_north_in_road.vehicle_out -= i285_north_in_road.vehicle_out

    i285_north_out_road.vehicle_wait += i285_south_in_road.vehicle_out
    i285_south_in_road.vehicle_out -= i285_south_in_road.vehicle_out

    i75_south_out_road.vehicle_wait += i75_north_in_postramp.vehicle_out
    i75_north_in_postramp.vehicle_out -= i75_north_in_postramp.vehicle_out

    i75_north_out_road.vehicle_wait += i75_south_in_postramp.vehicle_out
    i75_south_in_postramp.vehicle_out -= i75_south_in_postramp.vehicle_out

    # update out roads

    i285_south_out_road.update(0, insert_on_last_lane=False)
    i285_north_out_road.update(0, insert_on_last_lane=False)
    i75_south_out_road.update(0, insert_on_last_lane=False)
    i75_north_out_road.update(0, insert_on_last_lane=False)


#     if i > 1000 and i % 1000 == 0:
#         i285_south_out_road.report()
#         i285_north_out_road.report()
#         i75_south_out_road.report()
#         i75_north_out_road.report()
```

Out[43]:

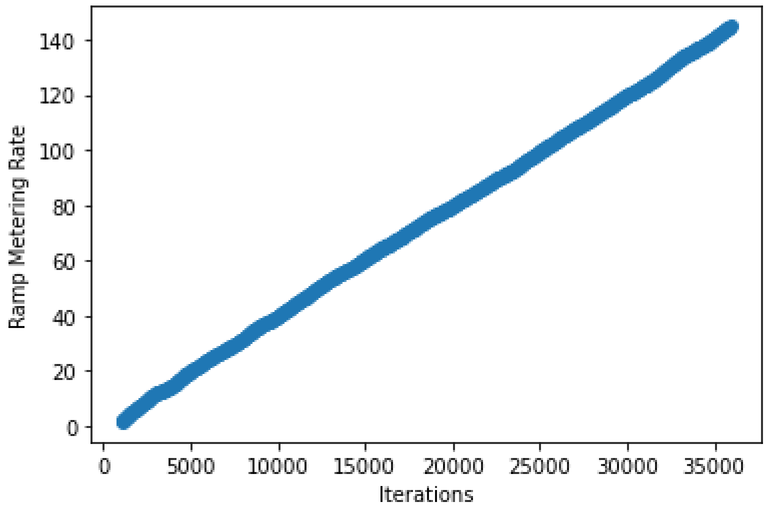
```
Wall time: 1h 58min 49s
```

Below we can visualize the ramp metering rate at each respective time step on the ramp located at I-75 South of the interchange.

In [44]:

```
plt.close()
plt.scatter(np.array(i75_south_r_hist)[:,0], np.array(i75_south_r_hist)[:,1])
plt.xlabel('Iterations')
plt.ylabel('Ramp Metering Rate')
plt.show()
```

Out[44]:

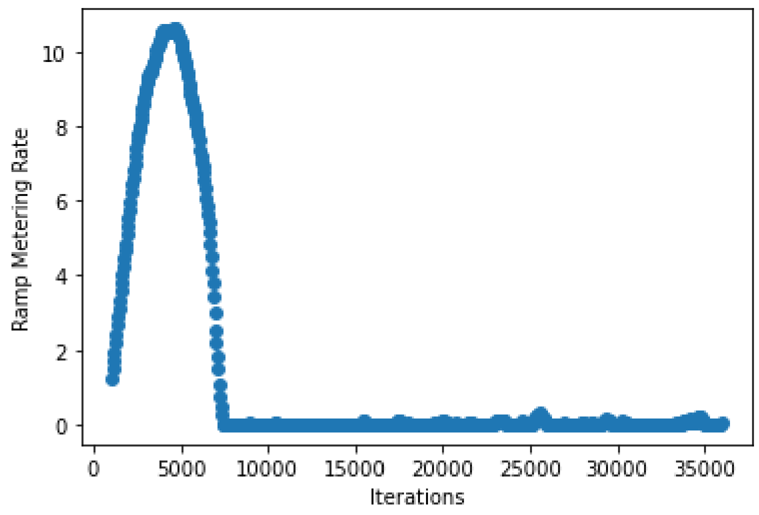


The scatterplot shows a highly linear relationship between ramp metering rate and time. This shows that the ramp metering rate goes unbounded as time inreases. Now let's look at the ramp metering rate for the ramp located at I-75 North of the interchange.

In [45]:

```
plt.close()
plt.scatter(np.array(i75_north_r_hist)[:,0], np.array(i75_north_r_hist)[:,1])
plt.xlabel('Iterations')
plt.ylabel('Ramp Metering Rate')
plt.show()
```

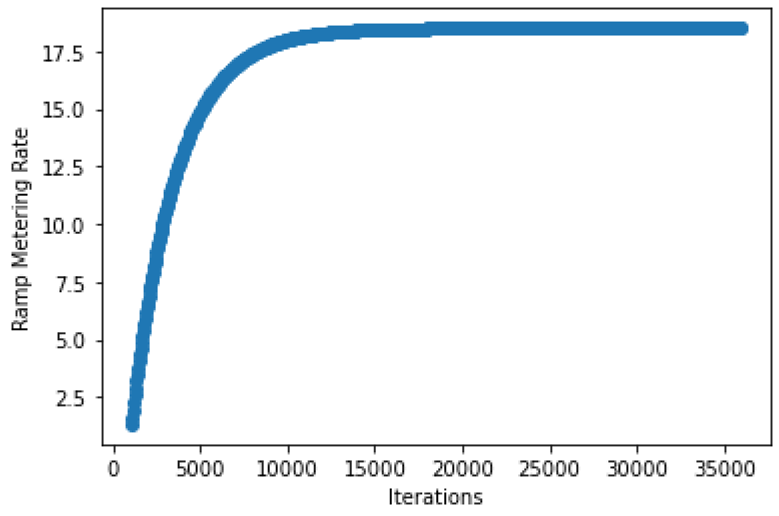
Out[45]:



From the visualization, we can see that ramp metering follows a parabolic pattern, with a peak right before 5000 time steps, until it flattens out at 0 for the remainder of the simulation. This is indicative that the ramp meter was unable to maintain the threshold and this likely caused a backup at the ramp, which we will see below. Now we can take a look at the ramp located at I-285 South of the interchange, also known as the Cumberland Ramp.

In [46]:
```
plt.close()
plt.scatter(np.array(cumberland_r_hist)[:,0], np.array(cumberland_r_hist)[:,1])
plt.xlabel('Iterations')
plt.ylabel('Ramp Metering Rate')
plt.show()
```

Out[46]:

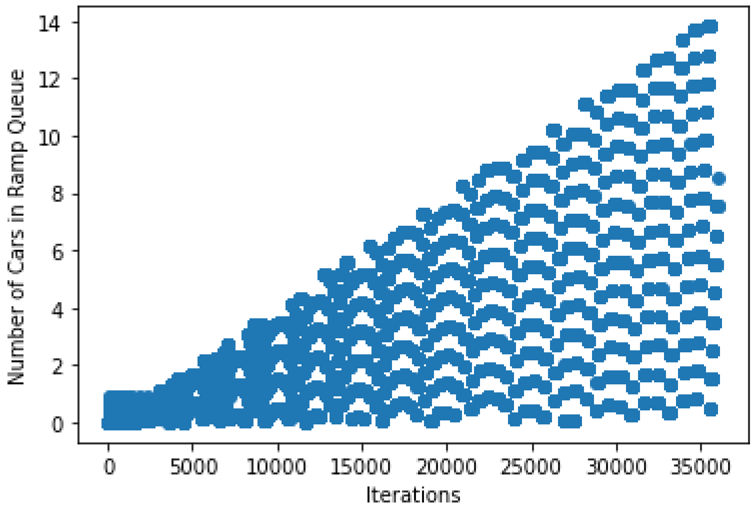


We can see a consistent dramatic increase in the ramp metering rate until around time step 8000, at which the ramp metering rate flattens out.

Now let's look at the queue lengths for each of the ramps, in the respective order the ramp metering rates were visualized.

In [47]:
```
plt.close()
plt.scatter(np.array(i75_south_ramp_queue)[:,0], np.array(i75_south_ramp_queue)[:,1])
plt.xlabel('Iterations')
plt.ylabel('Number of Cars in Ramp Queue')
plt.show()
print(np.array(i75_south_ramp_queue)[:,1].mean())
```

Out[47]:



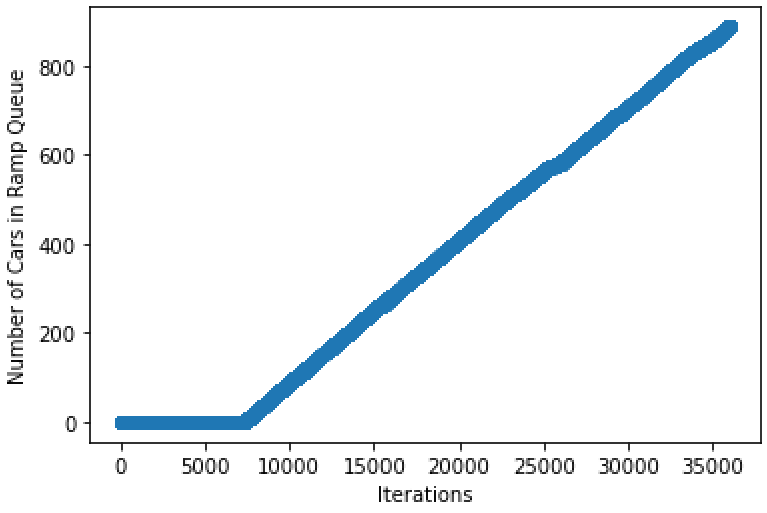
```
3.5700247569446177
```

The I-75 ramp metering queue shows relatively consistent overall increase throuhgout the simulation as time goes on, so there's overall large amounts of backup. However, large fluctuations occur throughout the simulation.

In [48]:

```
plt.close()
plt.scatter(np.array(i75_north_ramp_queue)[:,0], np.array(i75_north_ramp_queue)[:,1])
plt.xlabel('Iterations')
plt.ylabel('Number of Cars in Ramp Queue')
plt.show()
print(np.array(i75_north_ramp_queue)[:,1].mean())
```

Out[48]:



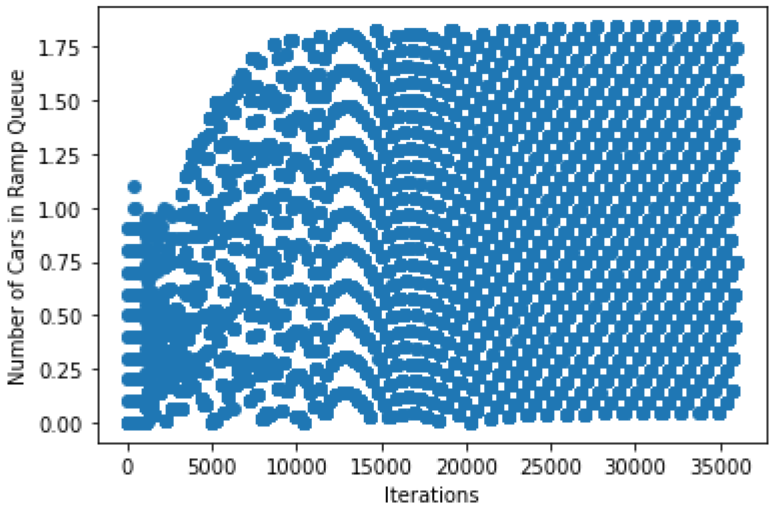
```
360.9239655053931
```

On the ramp located at I-75 North of the interchange, it is clear that the ramp queue is nonexistent until it steadily increases overtime. This follows the pattern of the ramp metering rate visualization, as the ramp metering rate rests at a negligible rate right as the queue length starts to build up.

In [49]:

```
plt.close()
plt.scatter(np.array(cumberland_ramp_queue)[:,0], np.array(cumberland_ramp_queue)[:,1])
plt.xlabel('Iterations')
plt.ylabel('Number of Cars in Ramp Queue')
plt.show()
print(np.array(cumberland_ramp_queue)[:,1].mean())
```

Out[49]:



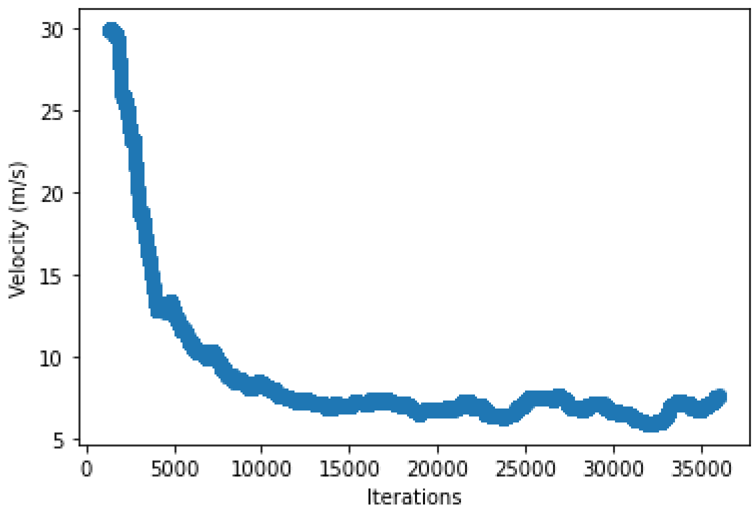
```
0.8394344398709073
```

On the Cumberland Ramp, we can see more feedback patterns, much like on the ramp on I-75 North of the interchange, but an overall consistent low value of the ramp queue. As evidenced the ramp metering rate for this ramp, the metering rate flattens out at a consistent rate value. So, it is sensible for the queue to flatten out likewise as evidenced here.

In [50]:

```
vels = []
for i, state in enumerate(i285_south_out_road.road_state_vel):
    if len(state) == 0:
        continue

    state_vel = list(np.array(state)[:,1])
    vels.append((i, np.array(state_vel).mean()))
plt.close()
plt.scatter(np.array(vels)[1000:,0], np.array(vels)[1000:,1])
plt.xlabel('Iterations')
plt.ylabel('Velocity (m/s)')
plt.show()
np.array(vels)[:,1].mean()
```

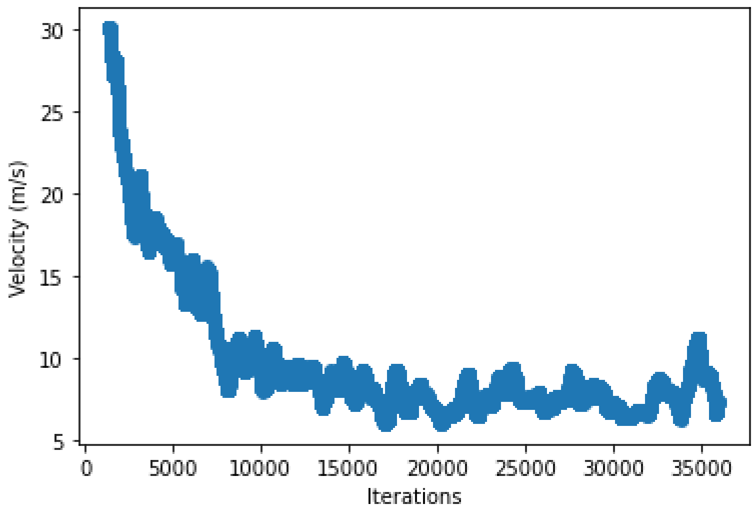
Out[50]:



Out[50]: 9.357609086394884

In [51]:
```
vels = []
for i, state in enumerate(i285_north_out_road.road_state_vel):
    if len(state) == 0:
        continue

    state_vel = list(np.array(state)[:,1])
    vels.append((i, np.array(state_vel).mean()))
plt.close()
plt.scatter(np.array(vels)[1000:,0], np.array(vels)[1000:,1])
plt.xlabel('Iterations')
plt.ylabel('Velocity (m/s)')
plt.show()
np.array(vels)[:,1].mean()
```

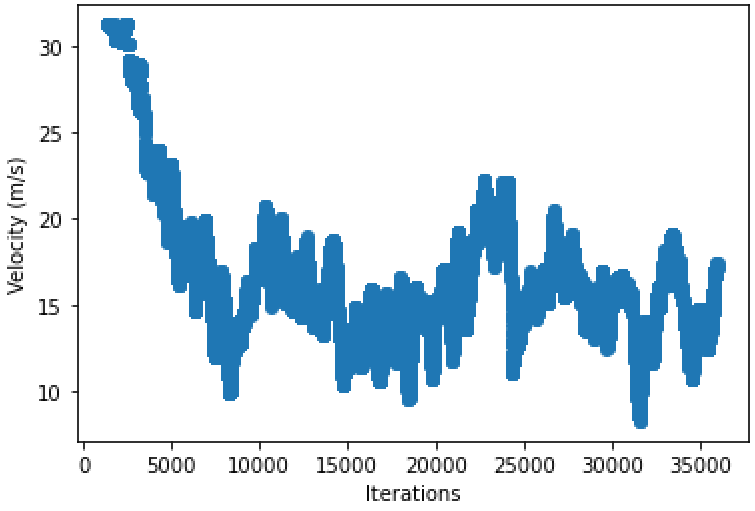
Out[51]:



Out[51]: 10.451989883480541

In [52]:
```
vels = []
for i, state in enumerate(i75_south_out_road.road_state_vel):
    if len(state) == 0:
        continue

    state_vel = list(np.array(state)[:,1])
    vels.append((i, np.array(state_vel).mean()))
plt.close()
plt.scatter(np.array(vels)[1000:,0], np.array(vels)[1000:,1])
plt.xlabel('Iterations')
plt.ylabel('Velocity (m/s)')
plt.show()
np.array(vels)[:,1].mean()
```

Out[52]:



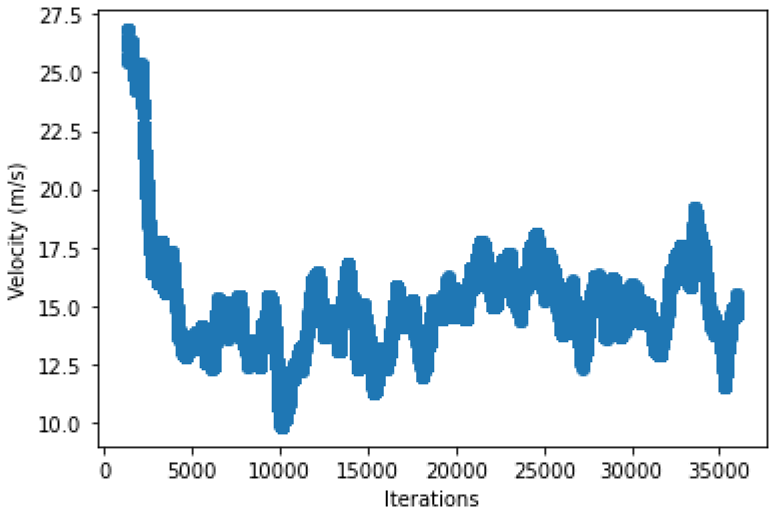
Out[52]:

```
17.106791485092007
```

In [53]:

```
vels = []
for i, state in enumerate(i75_north_out_road.road_state_vel):
    if len(state) == 0:
        continue

    state_vel = list(np.array(state)[:,1])
    vels.append((i, np.array(state_vel).mean()))
plt.close()
plt.scatter(np.array(vels)[1000:,0], np.array(vels)[1000:,1])
plt.xlabel('Iterations')
plt.ylabel('Velocity (m/s)')
plt.show()
np.array(vels)[:,1].mean()
```

Out[53]:



```
Out[53]: 15.395179497747161
```

As the average velocities indicate, the implementation of ALINEA did lead to an overall increase in average velocity and thus better traffic flow. However, this was at the expense of number of cars in queue increasing. Thus, this next simulation focuses on addressing the ramp queue buildup by implementing a modified ALINEA method. We expect that this will decrease the number of cars in the ramp queue drastically while maintain a marginal difference in average velocity.

Simulate with ALINEA_q

The simulation of the interchange below utilizes the modified ALINEA formulation our group designed. This formulation is applied to each of the ramp meters.

```
In [54]:
# Params
cf = CarFactory()
dt = 0.1

# Road Definitions
i285_south_in_road = Street(6, 3911, cf, dt=dt)
i285_south_out_road = Street(6, 3911, cf, dt=dt)

i285_north_in_road = Street(6, 1543, cf, dt = dt)
i285_north_out_road = Street(6, 1543, cf, dt = dt)

i75_south_in_preramp = Street(6, 568, cf, dt=dt)
i75_south_in_postramp = Street(6, 320, cf, dt=dt)
i75_south_in_ramp = Street(1, 50, cf, dt=dt)
i75_south_out_road = Street(6, 888, cf, dt = dt)

i75_north_in_preramp = Street(6,1018, cf, dt = dt)
i75_north_in_postramp = Street(6, 752, cf, dt = dt)
i75_north_in_ramp = Street(1, 50, cf, dt = dt)
i75_north_out_road = Street(6, 1770, cf, dt = dt)

cumberland_ramp = Street(1, 50, cf, dt = dt)


# ALINEA Params
k_r = 70 / 3600

i75_south_o_thres = 18.5
i75_south_r_prev = 1
i75_south_q_thres = 5
i75_south_r_hist = []
i75_south_ramp_queue = []

i75_north_o_thres = 18.5
i75_north_r_prev = 1
i75_north_q_thres = 5
i75_north_r_hist = []
i75_north_ramp_queue = []

cumberland_o_thres = 18.5
cumberland_r_prev = 1
```

```
cumberland_q_thres = 5
cumberland_r_hist = []
cumberland_ramp_queue = []


seed = 1111
np.random.seed(seed=seed)
```

In [55]:

```
%%time

# Simulation time
for i in tqdm(range(36000)):


    # hw input
    i285_south_q_in = np.random.normal(i285_south_in, i285_south_in/2 ** 1/5, (1))[0]
    i285_north_q_in = np.random.normal(i285_north_in, i285_north_in/2 ** 1/5, (1))[0]
    i75_south_q_in = np.random.normal(i75_south_in, i75_south_in/2 ** 1/5, (1))[0]
    i75_north_q_in = np.random.normal(i75_north_in, i75_north_in/2 ** 1/5, (1))[0]

    # ramp input
    i75_south_in_ramp_q_in = np.random.normal(cumberland_ramp_in, cumberland_ramp_in/2
    i75_north_in_ramp_q_in = np.random.normal(i75_south_in_ramp_in, i75_south_in_ramp_i
    cumberland_ramp_q_in = np.random.normal(i75_north_in_ramp_in, i75_north_in_ramp_in/


    # i285 south
    i285_south_in_road.update(i285_south_q_in, insert_on_last_lane=False)

    # i285 north
    i285_north_in_road.update(i285_north_q_in, insert_on_last_lane=False)

    # i75 south
    i75_south_in_preramp.update(i75_south_q_in, insert_on_last_lane=False)
    i75_south_in_postramp.vehicle_wait += i75_south_in_preramp.vehicle_out
    i75_south_in_preramp.vehicle_out -= i75_south_in_preramp.vehicle_out

    i75_south_in_ramp.update(i75_south_in_ramp_q_in)

    try:
        b = np.array(i75_south_in_postramp.road_state[i-1])
        b = b[np.where(b[:,0] == max(b[:, 0]))]
        o_out = len(b)
    except:
        o_out = i75_south_r_prev


    if i > 1000 and i % 50 == 0:
        i75_south_r_prev = ALINEA_q(i75_south_r_prev, k_r, i75_south_o_thres, o_out, i7
        i75_south_r_hist.append((i, i75_south_r_prev))


    if i75_south_in_ramp.vehicle_out >= i75_south_r_prev * dt:
        i75_south_in_ramp.vehicle_out -= i75_south_r_prev * dt
        i75_south_in_postramp.vehicle_wait_ramp += i75_south_r_prev * dt
    i75_south_in_postramp.update(0, insert_on_last_lane=True)

    i75_south_ramp_queue.append((i, i75_south_in_ramp.vehicle_out))

    # i75 north
    i75_north_in_preramp.update(i75_north_q_in, insert_on_last_lane=False)
    i75_north_in_postramp.vehicle_wait += i75_north_in_preramp.vehicle_out
    i75_north_in_preramp.vehicle_out -= i75_north_in_preramp.vehicle_out

    i75_north_in_ramp.update(i75_north_in_ramp_q_in)

    try:
        b = np.array(i75_north_in_postramp.road_state[i-1])
        b = b[np.where(b[:,0] == max(b[:, 0]))]
        o_out = len(b)
    except:
        o_out = i75_north_r_prev


    if i > 1000 and i % 50 == 0:
        i75_north_r_prev = ALINEA_q(i75_north_r_prev, k_r, i75_north_o_thres, o_out, i7
        i75_north_r_hist.append((i, i75_north_r_prev))


    if i75_north_in_ramp.vehicle_out >= i75_north_r_prev * dt:
        i75_north_in_ramp.vehicle_out -= i75_north_r_prev * dt
        i75_north_in_postramp.vehicle_wait_ramp += i75_north_r_prev * dt

    i75_north_in_postramp.update(0, insert_on_last_lane=True)

    i75_north_ramp_queue.append((i, i75_north_in_ramp.vehicle_out))


    # cum ramp
    cumberland_ramp.update(cumberland_ramp_q_in)

    try:
        a = np.array(i75_north_out_road.road_state[i-1])
        a = a[np.where(a[:,0] == max(a[:, 0]))]

        b = np.array(i75_south_out_road.road_state[i-1])
        b = b[np.where(b[:,0] == max(b[:, 0]))]

        c = np.array(i285_north_out_road.road_state[i-1])
        c = a[np.where(c[:,0] == max(c[:, 0]))]

        o_out = cumberland_ramp_i75_north_prop * len(a) + cumberland_ramp_i75_south_pro
    except:
        o_out = cumberland_r_prev


    if i > 1000 and i % 50 == 0:
        cumberland_r_prev = ALINEA_q(cumberland_r_prev, k_r, cumberland_o_thres, o_out,
        cumberland_r_hist.append((i, cumberland_r_prev))


    # i285 north out
    i285_north_out_road.vehicle_wait += i75_north_in_postramp.vehicle_out * i75_north_i
    i75_north_in_postramp.vehicle_out -= i75_north_in_postramp.vehicle_out * i75_north_

    i285_north_out_road.vehicle_wait += i75_south_in_postramp.vehicle_out * i75_south_i
    i75_south_in_postramp.vehicle_out -= i75_south_in_postramp.vehicle_out * i75_south_

    i285_north_out_road.vehicle_wait += i285_south_in_road.vehicle_out * i285_south_i28
    i285_south_in_road.vehicle_out -= i285_south_in_road.vehicle_out * i285_south_i285_

    # i285 south out
    i285_south_out_road.vehicle_wait += i75_north_in_postramp.vehicle_out * i75_north_i
    i75_north_in_postramp.vehicle_out -= i75_north_in_postramp.vehicle_out * i75_north_

    i285_south_out_road.vehicle_wait += i75_south_in_postramp.vehicle_out * i75_south_i
    i75_south_in_postramp.vehicle_out -= i75_south_in_postramp.vehicle_out * i75_south_

    i285_south_out_road.vehicle_wait += i285_north_in_road.vehicle_out * i285_north_i28
    i285_north_in_road.vehicle_out -= i285_north_in_road.vehicle_out * i285_north_i285_

    # i75 north out
    i75_north_out_road.vehicle_wait += i285_north_in_road.vehicle_out * i285_north_i75_
    i285_north_in_road.vehicle_out -= i285_north_in_road.vehicle_out * i285_north_i75_n

    i75_north_out_road.vehicle_wait += i75_south_in_postramp.vehicle_out * i75_south_i7
    i75_south_in_postramp.vehicle_out -= i75_south_in_postramp.vehicle_out * i75_south_

    i75_north_out_road.vehicle_wait += i285_south_in_road.vehicle_out * i285_south_i75_
    i285_south_in_road.vehicle_out -= i285_south_in_road.vehicle_out * i285_south_i75_n

    # i75 south out
    i75_south_out_road.vehicle_wait += i285_north_in_road.vehicle_out * i285_north_i75_
    i285_north_in_road.vehicle_out -= i285_north_in_road.vehicle_out * i285_north_i75_s

    i75_south_out_road.vehicle_wait += i75_north_in_postramp.vehicle_out * i75_north_i7
    i75_north_in_postramp.vehicle_out -= i75_north_in_postramp.vehicle_out * i75_north_

    i75_south_out_road.vehicle_wait += i285_south_in_road.vehicle_out * i285_south_i75_
    i285_south_in_road.vehicle_out -= i285_south_in_road.vehicle_out * i285_south_i75_s

    # cumberland ramping
    if cumberland_ramp.vehicle_out >= cumberland_r_prev * dt:
        cumberland_ramp.vehicle_out -= cumberland_r_prev * dt
        i75_north_out_road.vehicle_wait_ramp += cumberland_r_prev * dt * cumberland_ram
        i75_south_out_road.vehicle_wait_ramp += cumberland_r_prev * dt * cumberland_ram
        i285_north_out_road.vehicle_wait_ramp += cumberland_r_prev * dt * cumberland_ra

    cumberland_ramp_queue.append((i, cumberland_ramp.vehicle_out))

    # remaining flow

    i285_south_out_road.vehicle_wait += i285_north_in_road.vehicle_out
    i285_north_in_road.vehicle_out -= i285_north_in_road.vehicle_out

    i285_north_out_road.vehicle_wait += i285_south_in_road.vehicle_out
    i285_south_in_road.vehicle_out -= i285_south_in_road.vehicle_out

    i75_south_out_road.vehicle_wait += i75_north_in_postramp.vehicle_out
    i75_north_in_postramp.vehicle_out -= i75_north_in_postramp.vehicle_out

    i75_north_out_road.vehicle_wait += i75_south_in_postramp.vehicle_out
    i75_south_in_postramp.vehicle_out -= i75_south_in_postramp.vehicle_out

    # update out roads

    i285_south_out_road.update(0, insert_on_last_lane=False)
    i285_north_out_road.update(0, insert_on_last_lane=False)
    i75_south_out_road.update(0, insert_on_last_lane=False)
    i75_north_out_road.update(0, insert_on_last_lane=False)


#     if i > 1000 and i % 1000 == 0:
#         i285_south_out_road.report()
#         i285_north_out_road.report()
#         i75_south_out_road.report()
#         i75_north_out_road.report()
```

Out[55]:

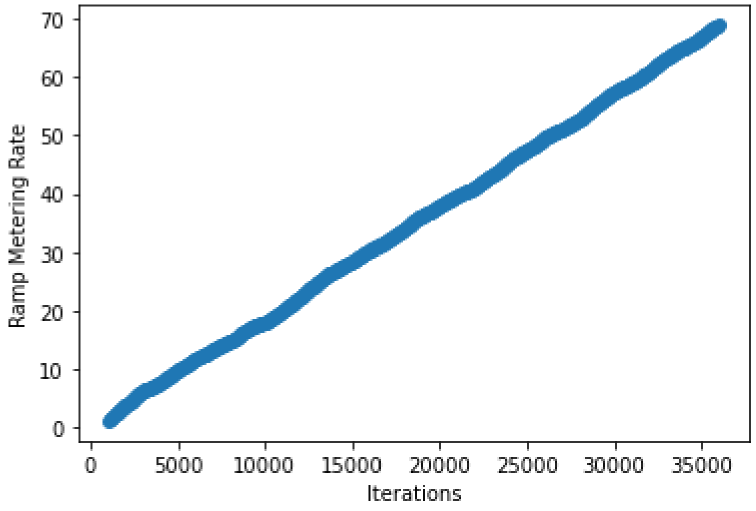
```
Wall time: 1h 55min 32s
```

Now we can look into the ramp metering rates for the ramps located at I-75 South of the interchange, I-75 North of the interchange, and the Cumberland Ramp, respectively. It'll be interesting to see the effects of ramp metering from the modified ALINEA formulation.

In [56]:

```
plt.close()
plt.scatter(np.array(i75_south_r_hist)[:,0], np.array(i75_south_r_hist)[:,1])
plt.xlabel('Iterations')
plt.ylabel('Ramp Metering Rate')
plt.show()
```

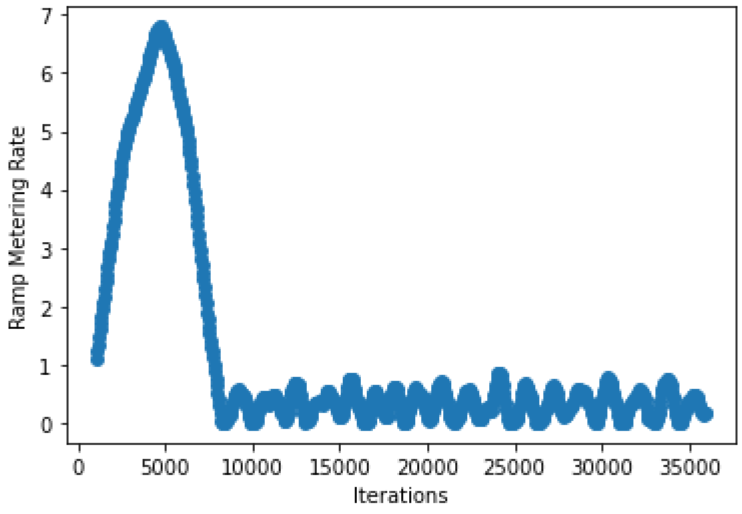
Out[56]:



Like in the original ALINEA formulation for I-75 South of the interchange, the modified ALINEA formulation, ALINEA_Q, demonstrates a positive linear relation between metering rate and time in simulation.

In [57]:
```python
plt.close()
plt.scatter(np.array(i75_north_r_hist)[:,0], np.array(i75_north_r_hist)[:,1])
plt.xlabel('Iterations')
plt.ylabel('Ramp Metering Rate')
plt.show()
```

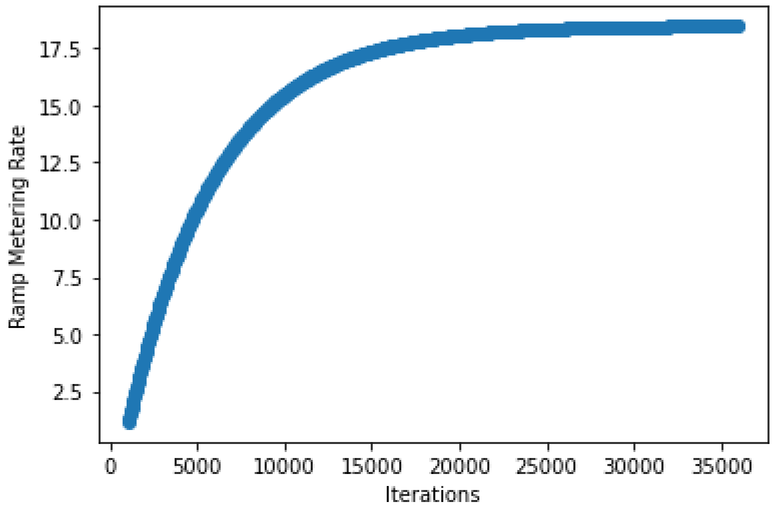
Out[57]:



The trends for the ramp metering rate of I-75 North of the interchange follow generally along the lines with the one from the original ALINEA formulation, with one notable exception. Instead of flattening out at a negligible value, the ramp metering rate for the modified ALINEA formulation oscillates at a continuous low value. This reflects a goal of the modified ALINEA formulation: to avoid deadlocks that can occur in the ramp metering rate, and thus an absurd queue size should be eliminated.

In [58]:
```python
plt.close()
plt.scatter(np.array(cumberland_r_hist)[:,0], np.array(cumberland_r_hist)[:,1])
plt.xlabel('Iterations')
plt.ylabel('Ramp Metering Rate')
plt.show()
```

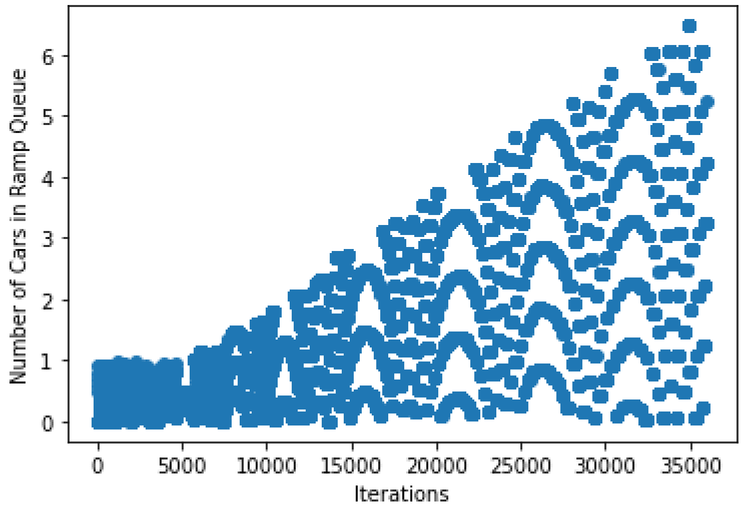
Out[58]:



As in the original ALINEA formulation, the modified ALINEA formulation displays a nearly identical trend in ramp metering rate. This is most likely due to the fact that this ramp can enter a vehicle into 3 different sections of the interchange and thus it is less dependent on the occupancy parameter of a single interstate. The next 3 figures focus on evaluating the number of cars in the ramp queue at each respective ramp.

In [59]:
```
plt.close()
plt.scatter(np.array(i75_south_ramp_queue)[:,0], np.array(i75_south_ramp_queue)[:,1])
plt.xlabel('Iterations')
plt.ylabel('Number of Cars in Ramp Queue')
plt.show()
print(np.array(i75_south_ramp_queue)[:,1].mean())
```

Out[59]:

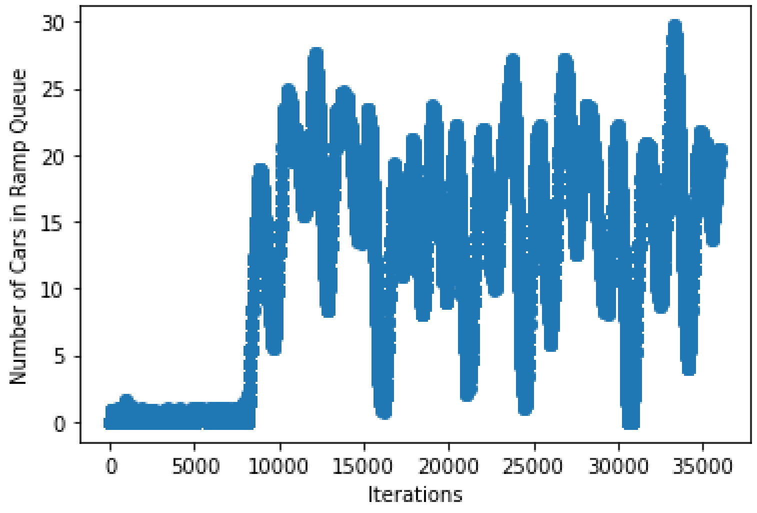


```
1.6752500462896731
```

The results from this are as expected in that the modified ALINEA method led to a significant decrease in the average number of cars in the ramp queue. More specifically the average number of cars in ramp queue decreased from 3.57 to 1.63.

In [60]:
```
plt.close()
plt.scatter(np.array(i75_north_ramp_queue)[:,0], np.array(i75_north_ramp_queue)[:,1])
plt.xlabel('Iterations')
plt.ylabel('Number of Cars in Ramp Queue')
plt.show()
print(np.array(i75_north_ramp_queue)[:,1].mean())
```

Out[60]:

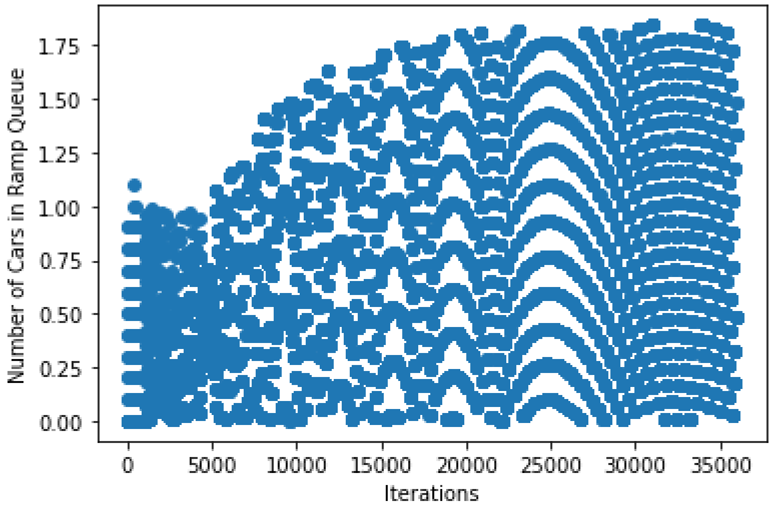


```
12.06918476242405
```

The modified ALINEA method led to a drastic decrease in number of cars in the ramp queue at this ramp in that the average number of cars in the ramp queue with the original method was around 360.93 and that value is 13.05 for this modified method.

In [61]:
```
plt.close()
plt.scatter(np.array(cumberland_ramp_queue)[:,0], np.array(cumberland_ramp_queue)[:,1])
plt.xlabel('Iterations')
plt.ylabel('Number of Cars in Ramp Queue')
plt.show()
print(np.array(cumberland_ramp_queue)[:,1].mean())
```

Out[61]:

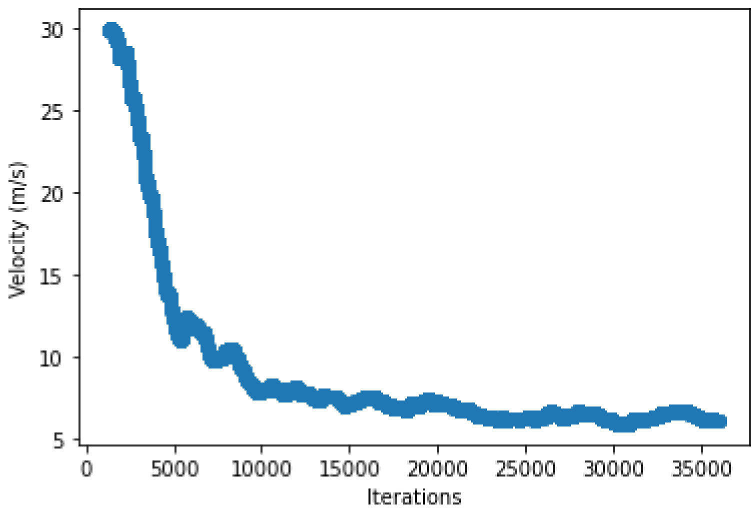


```
0.7637076526056988
```

The decrease in number of cars in the ramp queue is the least with respect to the other two ramps. This is expected due the the ramp metering rate being very similar to the original method and the fact that this ramp allows cars to enter 3 different sections of the interchange. Nevertheless, the average number of cars in the ramp queue did decrease from 0.84 to 0.78. Let's next analyze the velocities when implementing this new method.

In [62]:
```
vels = []
for i, state in enumerate(i285_south_out_road.road_state_vel):
    if len(state) == 0:
        continue

    state_vel = list(np.array(state)[:,1])
    vels.append((i, np.array(state_vel).mean()))
plt.close()
plt.scatter(np.array(vels)[1000:,0], np.array(vels)[1000:,1])
plt.xlabel('Iterations')
plt.ylabel('Velocity (m/s)')
plt.show()
np.array(vels)[:,1].mean()
```

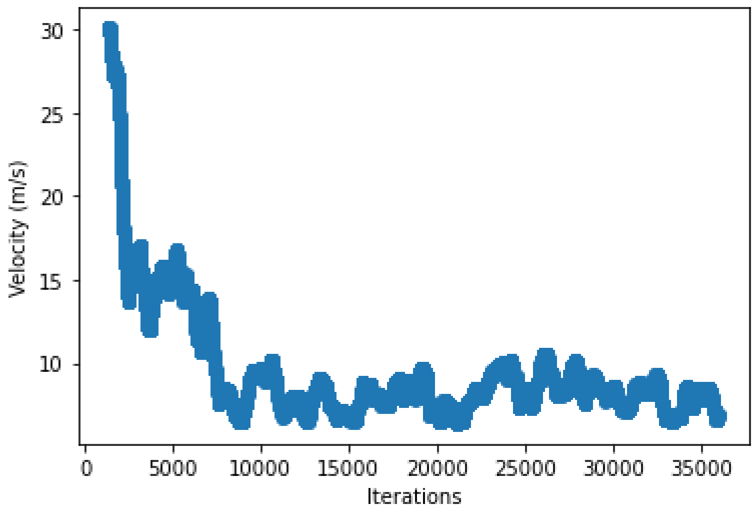
Out[62]:



Out[62]:
```
9.609670010296103
```

In [63]:
```python
vels = []
for i, state in enumerate(i285_north_out_road.road_state_vel):
    if len(state) == 0:
        continue

    state_vel = list(np.array(state)[:,1])
    vels.append((i, np.array(state_vel).mean()))
plt.close()
plt.scatter(np.array(vels)[1000:,0], np.array(vels)[1000:,1])
plt.xlabel('Iterations')
plt.ylabel('Velocity (m/s)')
plt.show()
np.array(vels)[:,1].mean()
```

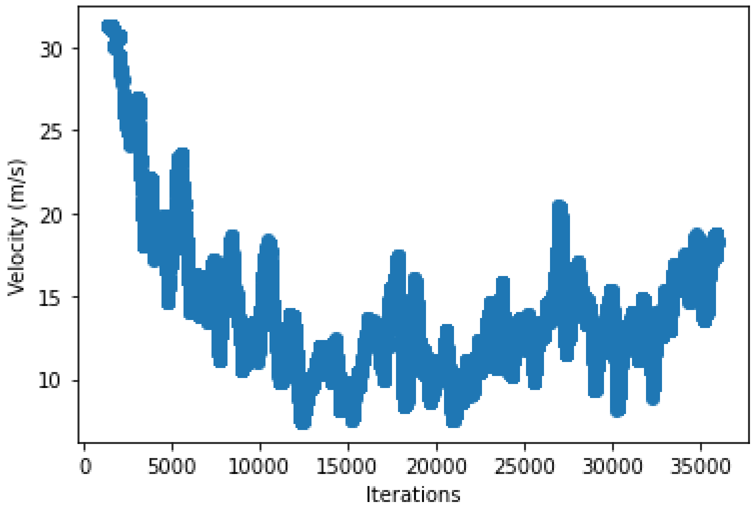
Out[63]:



Out[63]:
```
10.081965430734698
```

In [64]:
```python
vels = []
for i, state in enumerate(i75_south_out_road.road_state_vel):
    if len(state) == 0:
        continue

    state_vel = list(np.array(state)[:,1])
    vels.append((i, np.array(state_vel).mean()))
plt.close()
plt.scatter(np.array(vels)[1000:,0], np.array(vels)[1000:,1])
plt.xlabel('Iterations')
plt.ylabel('Velocity (m/s)')
plt.show()
np.array(vels)[:,1].mean()
```

Out[64]:



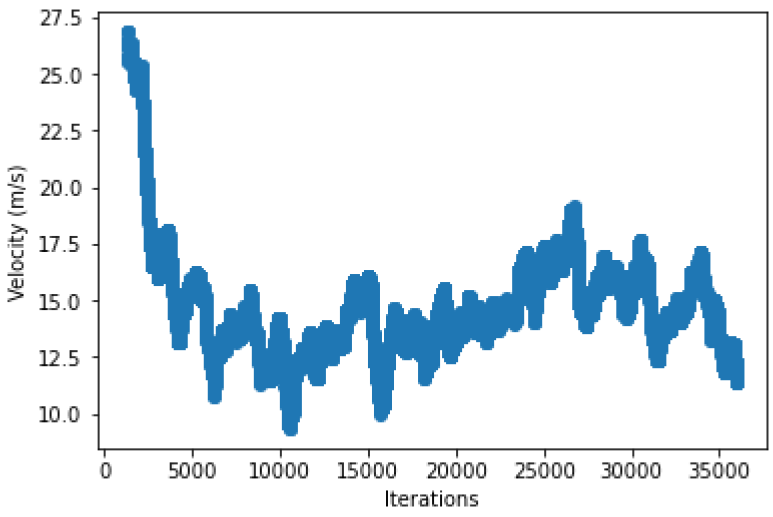
Out[64]:

```
14.601119431975848
```

In [65]:

```
vels = []
for i, state in enumerate(i75_north_out_road.road_state_vel):
    if len(state) == 0:
        continue

    state_vel = list(np.array(state)[:,1])
    vels.append((i, np.array(state_vel).mean()))
plt.close()
plt.scatter(np.array(vels)[1000:,0], np.array(vels)[1000:,1])
plt.xlabel('Iterations')
plt.ylabel('Velocity (m/s)')
plt.show()
np.array(vels)[:,1].mean()
```

Out[65]:



```
Out[65]: 15.052264850277965
```

As the figures indicate, there is a change in the average velocities at each section of the interchange. This change is either a decrease or increase depending on the section. A comparative analysis of these changes will be made in the observations and discussion sections as seen below.

Observations and Discussion

We ran these interchange simulations on an i7-5960x which took the following amount of time to compute one hour's worth of simulation time at 0.1 second resolution:

Simulation	Time hh:mm:ss
Base, no policy	02:00:07
ALINEA	02:04:48
ALINEA_q	02:15:36

The following tables illustrate the effects of the implementation of ALINEA and ALINEA_q onto the simulation of the interchange when evaluating the average velocity and number of cars in the the ramp queue.

	No Policy	ALINEA	ALINEA_q
i75_south_out average velocity	9.356	9.357	9.609
i75_north_out average velocity	10.486	10.452	10.081
i285_south_out average velocity	14.296	17.106	14.601
i285_north_out average velocity	11.713	15.395	15.052

	No Policy	ALINEA	ALINEA_q
i75_south_in_ramp average queue size	0	3.570	1.675
i75_north_in_ramp average queue size	0	360.92	12.069
cumberland_ramp average queue size	0	0.839	0.763

Looking at the base model, that is, the interchange without any form of ramp policy; we notice that the average speed is fairly low at the four exit roads in the simulation. This is expected since there is no form of control for queuing cars into the system. With the application of ALINEA, we see similar results to what we observed in the simple example where the average speed greatly increases but at the cost of an unbounded ramp queue. This is problematic for a variety of reasons, and the speed increase is ultimately artificial since it is simply blocking cars from entering the road. Finally, with ALINEA_q, we notice a balance of the ramp queue length and velocity when comparing all of the roads. We notice two special output cases, both i75_south_out and i75_north_out, where a majority of the speed remains unchanged. We hypothesize that this invariance is due to how congested these roads already are, and the few cars added by ramping does not change the inherent flow very much. In the i285 case, we notice that the velocities vary greatly, as the addition of a few cars can drastically reduce the velocity.

ALINEA_q outperforms ALINEA and the no policy model in balancing average speed and cars on the road, essentially maximixing throughput.

Conclusion

With transportation systems becoming increasingly more congested in recent years, it is imperative to find solutions to accomodate for a growing society. Ramp metering serves as not only a solution but also as the optimal stepping stone towards ultimate free flow efficiency during all times of the day. To analyze the effects of ramp metering, we undertook one of the most important aspects of computational engineering: simulation analysis and design.

We introduce an object oriented design for accurate traffic simulation in order to model vehicle ramping strategies. We identify an existing ramping policy, ALINEA, and propose a modification and show empirical support for our modification. We analyze both the simulation and the policy on a complex basis across the I-75/I-285 Interchange, which will each hold important attributes for recording and analyzation. From there, the proof of concept demonstrated to you a simple instance of a freeway to foreshadow the implementation of the interchange. The main portion of our report demonstrates three models for experimentation on the effects of ramp metering: no ramp policy, the original ALINEA formulation, and the modified ALINEA_Q formulation, using real-world data that was publicly available to us in order to accurately undergo the simulations.

Our results were recorded and from our findings, we hope that the information presented can be of beneficial use to transportation departments to make the most optimal use of ramp meters to alleviate the problem of traffic congestion that has become so common in highly urbanized areas across the United States.

More importantly, our group has reflected on a few aspects of the model which could be further improved upon for future work. Our interchange is a simplified version of the actual interchange, with all the road segments that are part of the interchange simplified to the 13 fundamental road segments. Ideally, we would need around 100 total road segments in order to develop the most accurate interchange and build a sophisticated representation of the flow of vehicles. This was relatively unfeasible for our group due to the extreme amounts of computational power required that was not readily avaliable to us. In addition, express lanes were ignored in our model but are part of the real interchange, including functions to account for an express lane would further exalt the simulation to a higher level of precision.

With all aspects considered, we as a group are proud of the efforts expended to undergo this project, and we hope that not only will we take what we learned in the process of creating this application towards future endeavors, but that the application itself would be beneficial towards the betterment of the general public.

References

1. A. Kesting, M. Treiber, and D. Helbing, “General lane-changing model mobil forcar-following models,”Transportation Research Record, vol. 1999, no. 1, pp. 86–94,2007. [Online]. Available: https://doi.org/10.3141/1999-10[2]M.
2. Treiber, A. Hennecke, and D. Helbing, “Congested traffic states in empiricalobservations and microscopic simulations,”Physical Review E, vol. 62, no. 2, p.1805–1824, Aug 2000. [Online]. Available: http://dx.doi.org/10.1103/PhysRevE.62.1805[3]L.
3. Tang, X. Luo, P. Zhai, and X. Gao, “A density-based ramp meteringmodel considering multilane context in urban expressways,”MathematicalProblems in Engineering, vol. 2017, p. 4909363, Mar 2017. [Online]. Available:https://doi.org/10.1155/2017/49093636

Contributions

Name	Contribution
Ryan Cooper	Simulation Design and Implementation Proof of Concept Experiments Interchange Implementation Experiment Design Analysis
Kiarash Ahmadi	IDM/MOBIL Research Data Collection Data Preparation Interchange Implementation Experiment Design Analysis
Jason Lu	Interchange Identification and Design Interchange Implementation Experiment Design Analysis