HANDS-ON TUTORIALS

Simulating Traffic Flow in Python

Implementing a microscopic traffic model

Bilal Himite Sep 5, 2021 · 14 min read ★

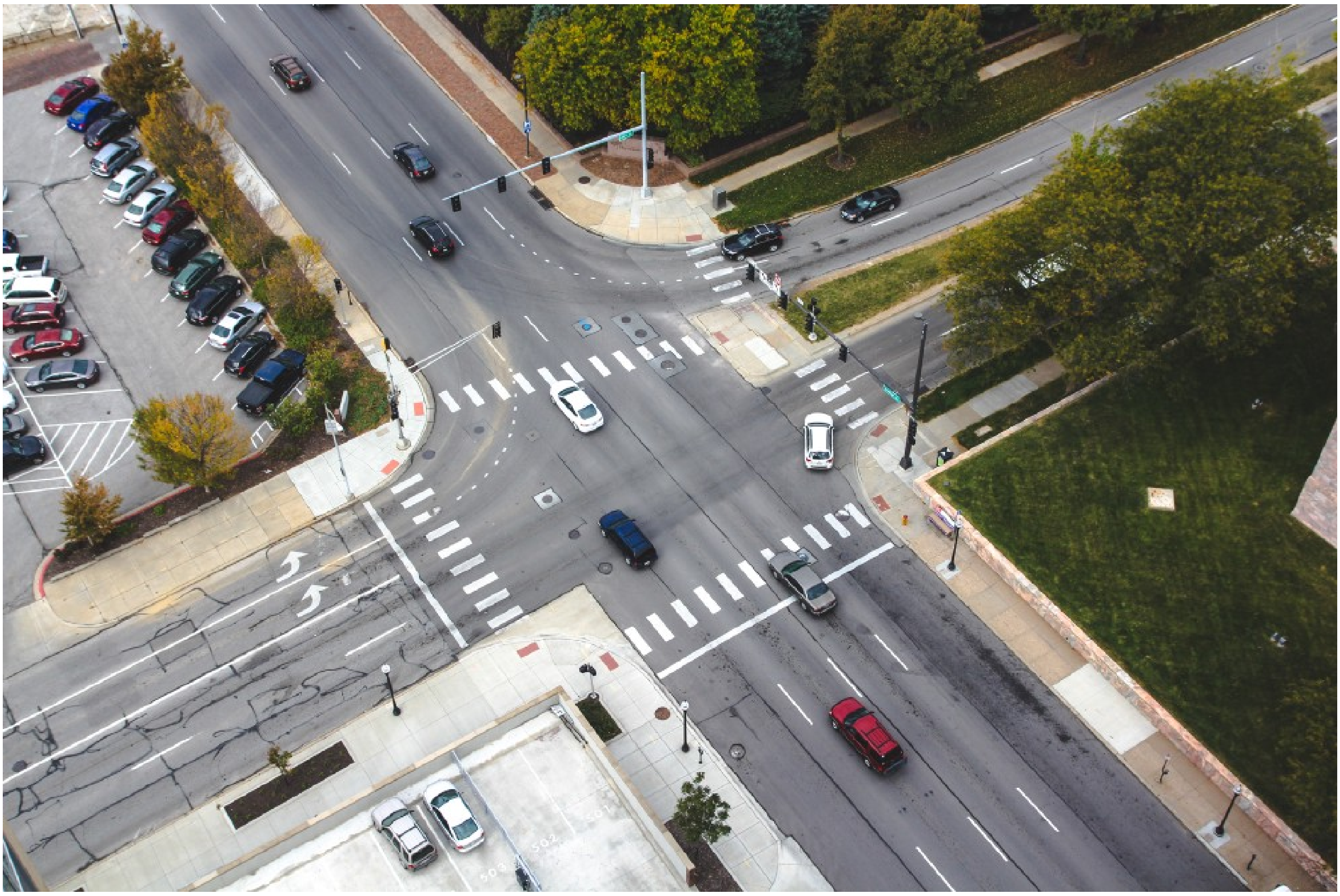


Photo by John Matychuk on Unsplash

Although traffic doesn't always flow smoothly, cars seamlessly crossing intersections and turning and stopping at traffic signals can look quite magnificent. This contemplation got me thinking of how important traffic flow is for human civilization.

After this, the nerd inside of me couldn't resist thinking of a way to simulate traffic flow. I spent a couple of weeks working on an undergraduate project involving traffic flow. I looked in-depth into different simulation techniques and I settled for one.

In this article, I will explain why traffic simulation is important, compare different methods possible to model traffic, and present my simulation (along with the source code).

Why simulate traffic flow?

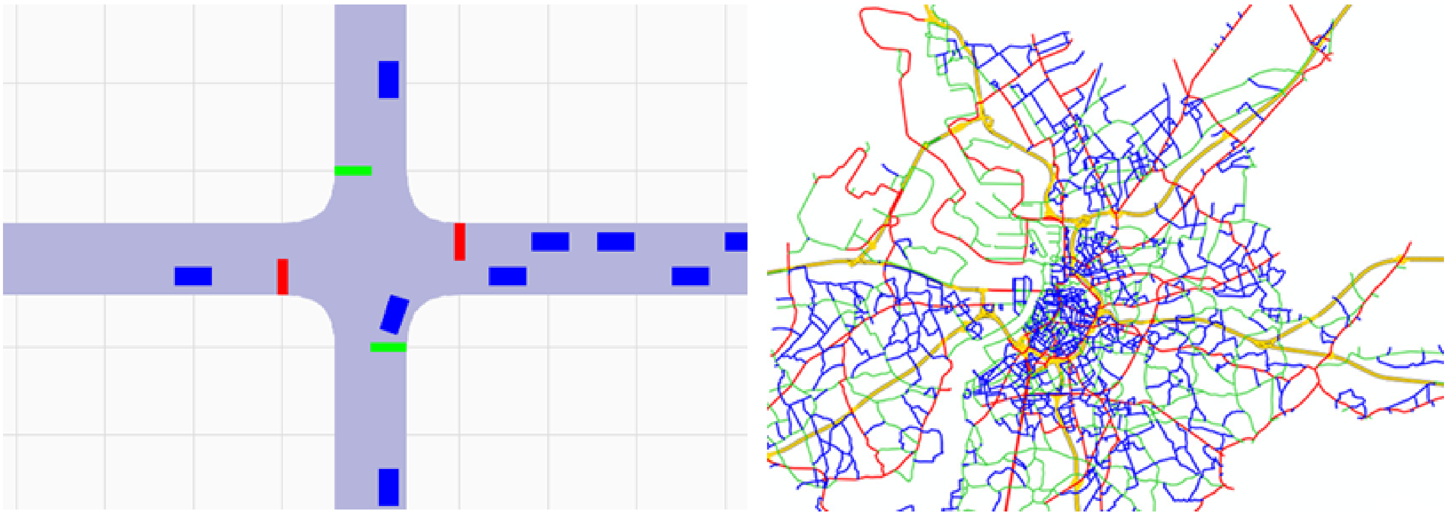
The main reason behind simulating traffic is generating data without the real world. Instead of testing new ideas on how to manage traffic systems in the real world or collect data using sensors, you can use a model run on software to predict traffic flow.

This helps accelerate the optimization and data gathering of traffic systems. Simulation is a much cheaper and faster alternative to real-world testing.

Training machine learning models requires huge datasets that can be difficult and costly to gather and process. Generating data procedurally by simulating traffic can be easily adapted to the exact type of data needed.

Modeling

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



Microscopic Model (left) Macroscopic Model (right). Image by Author.

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

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

In this article, I will use a microscopic model.

Microscopic Models

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

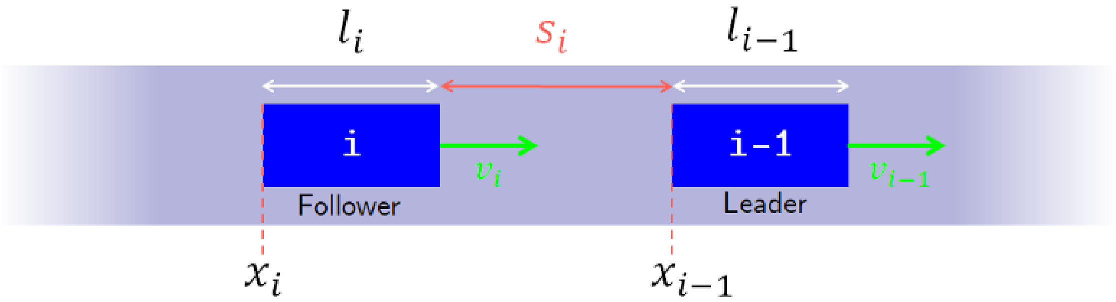


Image by Author.

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

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

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

Intelligent Driver Model (IDM)

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

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

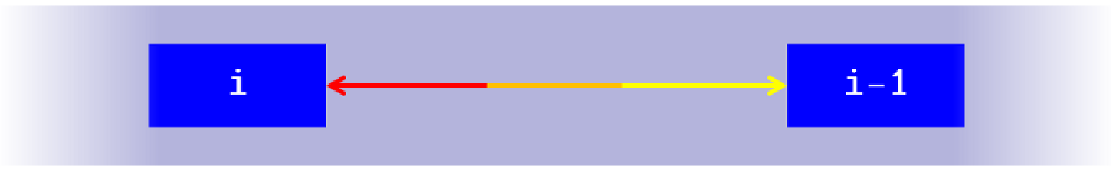
$$s^*(v_i, \Delta v_i) = s_{0,i} + v_i T_i + \frac{v_i \Delta v_i}{\sqrt{2 a_i b_i}}$$

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

We have talked about **s_i**, **v_i**, and **Δv_i**. The other parameters are:

- **so_i** : is the minimum desired distance between the vehicle *i* and *i-1*.
- **vo_i :** is the maximum desired speed of the vehicle *i*.
- **δ :** is the acceleration exponent and it controls the "smoothness" of the acceleration.
- **T_i :** is the reaction time of the *i*-th vehicle's driver.
- **a_i :** is the maximum acceleration for the vehicle *i*.
- **b_i :** is the comfortable deceleration for the vehicle *i*.
- **s*** : is the actual desired distance between the vehicle *i* and *i-1*.

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



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

Image by Author.

- **so_i :** as said before, is the minimum desired distance.
- **$v_i T_i$:** is the reaction time safety distance. It is the distance the vehicle travels before the driver reacts (brakes).
  Since speed is distance over time, distance is speed times time.

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

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

How The Intelligent Driver Model Works

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

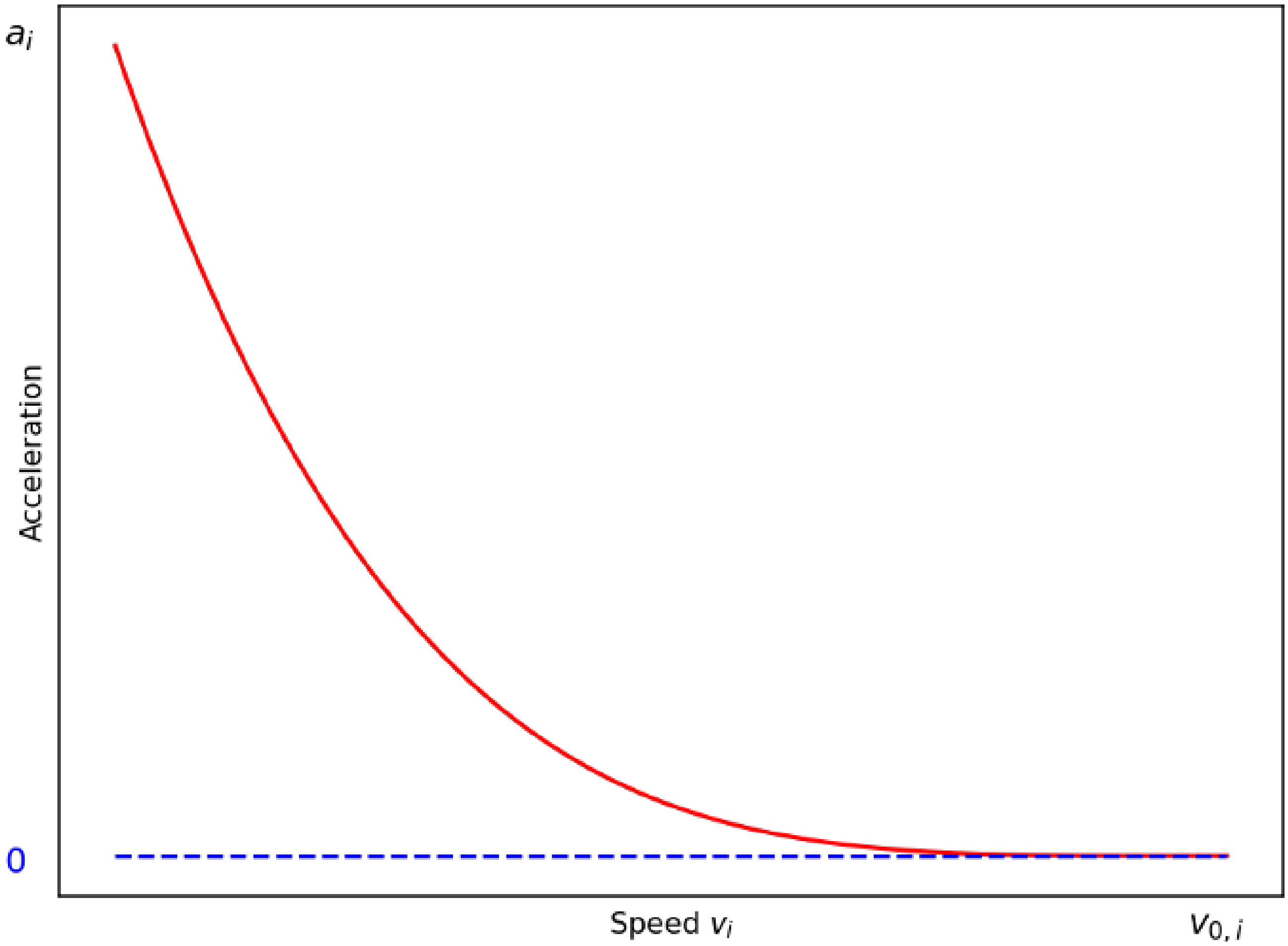
$$\frac{dv_i}{dt} = a_{free\,road} + a_{interaction}$$

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

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

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

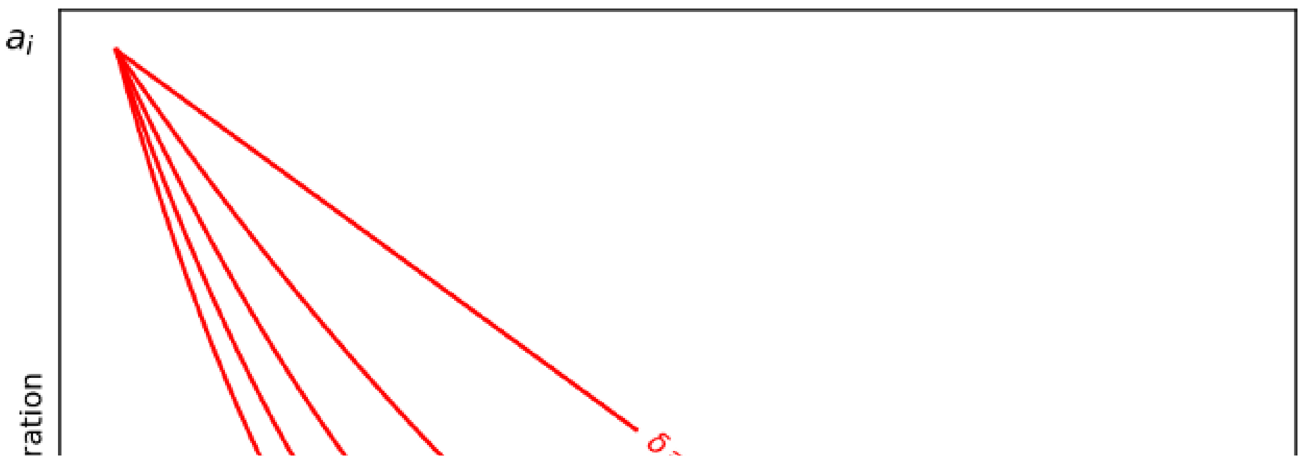
The **free road acceleration** is the acceleration on a free road, that is, an empty road with no vehicles ahead. If we plot the acceleration as a function of speed v_i we get**:**

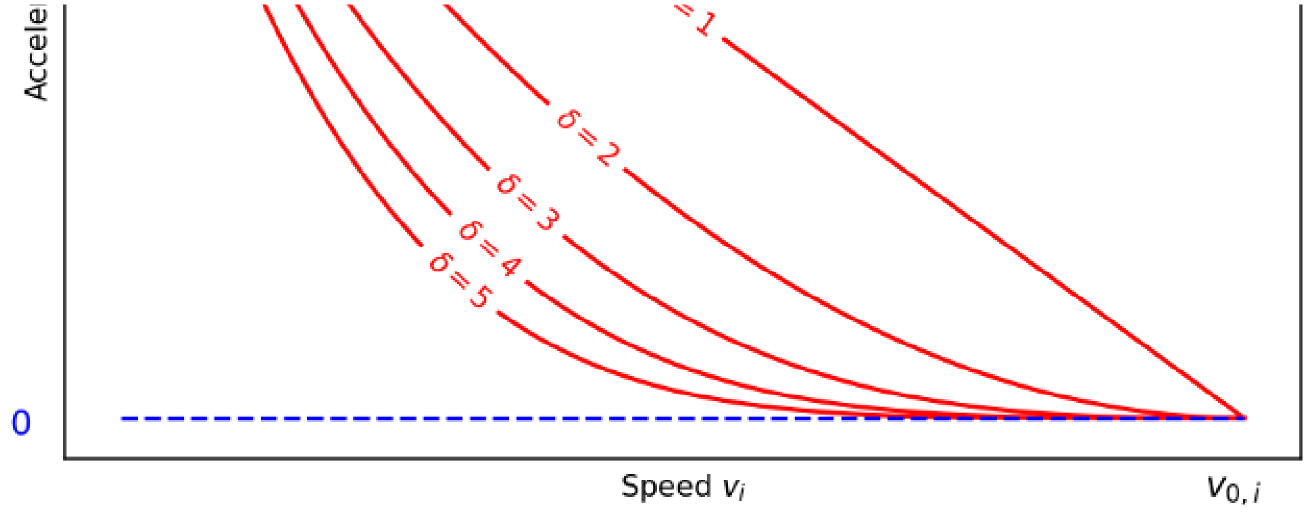


Acceleration as a function of speed. Image by Author.

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

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





Acceleration as a function of speed. Image by Author.

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

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

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

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

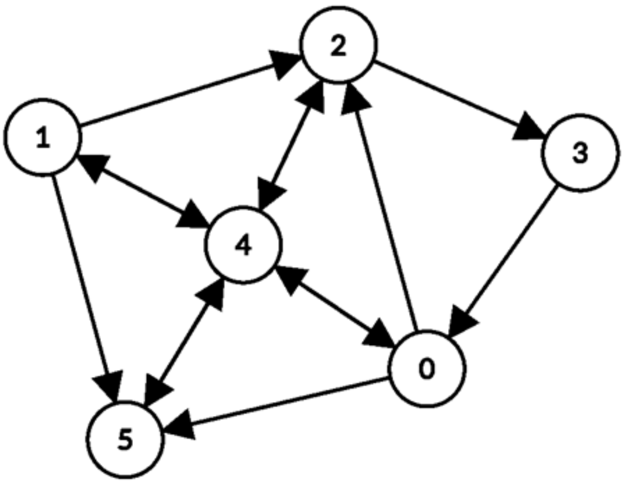
- **At high approach rates (Δv_i):**
When the speed difference is high, the interaction acceleration tries to compensate for that by braking or slowing down using the $\mathbf{(v_i \Delta v_i)^2}$ term in the numerator but too hard. This is achieved through the denominator $\mathbf{4 b_i s_i^2}$. (I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

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

- **At small distance difference ($s_i << 1$ and $\Delta v_i \approx 0$):**
  The acceleration becomes a simple repulsive force.

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

Traffic Road Network Model



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

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

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

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

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

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

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

Stochastic Vehicle Generator

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

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

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

A stochastic vehicle generator is defined by two constraints:

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

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

Traffic Light

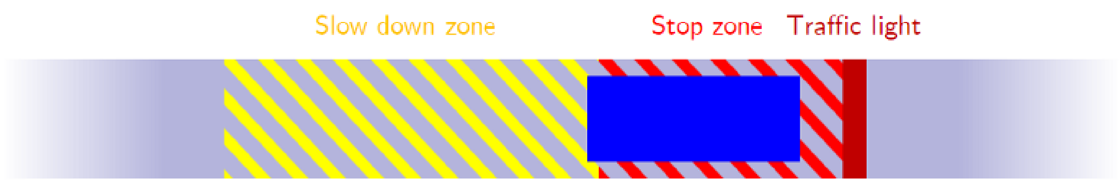


Image by Author.

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

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

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

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

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

Simulation

We will adopt an object-oriented approach. Every vehicle and road is going to be defined as a class.

We will use the following `__init__` function repeatedly in many upcoming classes. It sets the default configuration of the current class through a function `set_default_config`. And expects a dictionary and sets every property in the dictionary as a property to the current class instance. This way, we don't have to worry about updating `__init__` functions of different classes or about changes in the future.

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

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

traffic_init.py hosted with ❤ by GitHub view raw

Road

We will create a `Road` class:

```python
from scipy.spatial import distance

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

        self.init_porperties()
```

```

    def init_properties(self):
        self.length = distance.euclidean(self.start, self.end)
        self.angle_sin = (self.end[1]-self.start[1]) / self.length
        self.angle_cos = (self.end[0]-self.start[0]) / self.length
```

traffic_road.py hosted with ❤ by GitHub view raw

We will need the road's `length` and the cosine and sine of its angle when drawing it on the screen.

Simulation

And a `Simulation` class. I added some methods to add roads to the simulation.

```
from .road import Road

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

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

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

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

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

traffic_simulation.py hosted with ❤ by GitHub view raw

We have to display our simulation on the screen in real-time. To do this, we will use `pygame`. I will create a `Window` class that expects a `Simulation` class as a parameter.

I defined multiple drawing functions that help in drawing basic shapes.

The `loop` method creates a `pygame` window and calls the `draw` method and the `loop` parameter every frame. This will become useful when our simulation needs to be updated every frame.

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

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

        # Set default configurations
        self.set_default_config()

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

    def set_default_config(self):
        """Set default configuration"""
        self.width = 1400
        self.height = 1000
        self.bg_color = (250, 250, 250)

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

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


    def loop(self, loop=None):
        """Shows a window visualizing the simulation and runs the loop function."""
        # Create a pygame window
        self.screen = pygame.display.set_mode((self.width, self.height))
```

```
        self.screen = pygame.display.set_mode((self.width, self.height))
        pygame.display.flip()

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

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

        # Draw loop
        running = True
        while not self.sim.stop_condition(self.sim) and running:
            # Update simulation
            if loop: loop(self.sim)

            # Draw simulation
            self.draw()

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

            # Handle all events
            for event in pygame.event.get():
                # Handle mouse drag and wheel events
                ...


    def convert(self, x, y=None):
        """Converts simulation coordinates to screen coordinates"""
        ...

    def inverse_convert(self, x, y=None):
        """Converts screen coordinates to simulation coordinates"""
        ...


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

    def line(self, start_pos, end_pos, color):
        """Draws a line."""
        ...

    def rect(self, pos, size, color):
        """Draws a rectangle."""
```

```python
        ...

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

    def circle(self, pos, radius, color, filled=True):
        """Draws a circle"""
        ...

    def polygon(self, vertices, color, filled=True):
        """Draws a polygon"""

    def rotated_box(self, pos, size, angle=None, cos=None, sin=None, centered=True, co
        """Draws a filled rectangle centered at *pos* with size *size* rotated anti-cl

    def rotated_rect(self, pos, size, angle=None, cos=None, sin=None, centered=True, c
        """Draws a rectangle centered at *pos* with size *size* rotated anti-clockwise


    def draw_axes(self, color=(100, 100, 100)):
        """Draw x and y axis"""

    def draw_grid(self, unit=50, color=(150,150,150)):
        """Draws a grid"""

    def draw_roads(self):
        """Draws every road"""

    def draw_status(self):
        """Draws status text"""


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

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

        # Draw roads
        self.draw_roads()

        # Draw status info
        self.draw_status()

```

traffic_window.py hosted with ❤ by GitHub view raw

I combined every file in a folder named `trafficSimulator` with an `__init__.py` file importing all the class names.

```
from .road import *
from .simulation import *
from .window import *
```

__init__.py hosted with ❤ by GitHub view raw

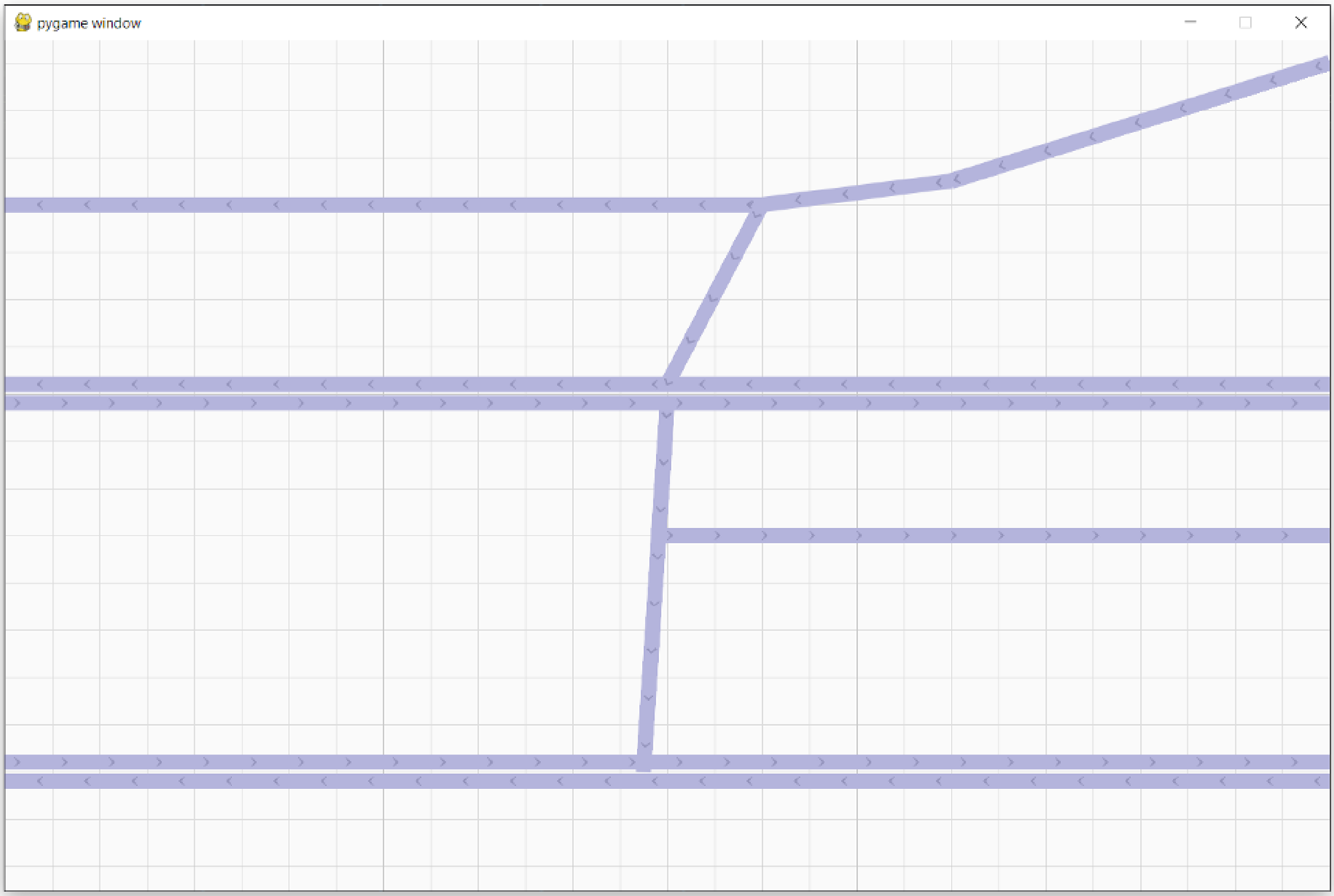
Whenever a new class is defined, it should be imported in this file

Placing the `trafficSimulator` folder in our project folder will let us use the module.

```
from trafficSimulator import *

# Create simulation
sim = Simulation()

# Add one road
sim.create_road((300, 98), (0, 98))

# Add multiple roads
sim.create_roads([
    ((300, 98), (0, 98)),
    ((0, 102), (300, 102)),
    ((180, 60), (0, 60)),
    ((220, 55), (180, 60)),
    ((300, 30), (220, 55)),
    ((180, 60), (160, 98)),
    ((158, 130), (300, 130)),
    ((0, 178), (300, 178)),
    ((300, 182), (0, 182)),
    ((160, 102), (155, 180))

])
```

```

# Start simulation
win = Window(sim)
win.loop()
```

test_1.py hosted with ❤ by GitHub view raw



Simulation test. Image by Author.

Vehicles

Now, we have to add vehicles.

We will use Taylor series to approximate the solution of the dynamics equations discussed in the modeling part of this article.

Taylor series expansion for an infinitely differential function `f` is:

$$f(x) = f(a) + \frac{df}{dt}(a) \cdot (\mathrm{x} - \mathrm{a}) + \frac{d^2 f}{dt^2}(a) \cdot \frac{(x-a)^2}{2} + \cdots$$

Substituting `a` by `▲x` ,and `x` by `x+▲x` we get:

$$f(x + \Delta x) = f(x) + \frac{df}{dt}(x) \cdot \Delta\mathrm{x} + \frac{d^2 f}{dt^2}(x) \cdot \frac{\Delta\mathrm{t}^2}{2} + \cdots$$

Replacing `f` by the position `x` :

$$x(t + \Delta t) = x(t) + \frac{dx}{dt}(t) \cdot \Delta\mathrm{t} + \frac{d^2 x}{dt^2}(t) \cdot \frac{\Delta\mathrm{t}^2}{2} + \cdots$$

As an approximation, we will stop at order 2 for position, since acceleration is the highest-order derivative. We get equation **(2)**:

$$x(t + \Delta t) \approx x(t) + v(t) \cdot \Delta\mathrm{t} + a(t) \cdot \frac{\Delta\mathrm{t}^2}{2} \qquad (2)$$

For speed, we will substitute `x` by `v` :

$$v(t + \Delta t) = v(t) + \frac{dv}{dt}(t) \cdot \Delta\mathrm{t} + \frac{d^2 v}{dt^2}(t) \cdot \frac{\Delta\mathrm{t}^2}{2} + \cdots$$

We will stop at order 1, since the highest-order derivative we have is acceleration (order 1 for speed). Equation **(2)**:

$$v(t + \Delta t) \approx v(t) + a(t) \cdot \Delta\mathrm{t} \qquad (1)$$

In every iteration (or frame), after calculating the acceleration using the IDM formula, we will update position and speed using these two equations:

$$v(t + \Delta t) \approx v(t) + a(t) \cdot \Delta\mathrm{t} \qquad (1)$$
$$x(t + \Delta t) \approx x(t) + v(t) \cdot \Delta\mathrm{t} + a(t) \cdot \frac{\Delta\mathrm{t}^2}{2} \qquad (2)$$

In code this looks like this:

```
self.a = ...        # IDM formula
self.v += self.a*dt
self.x += self.v*dt + self.a*dt*dt/2
```

numerical_apprx.py hosted with ❤ by GitHub view raw

Since this is only an approximation, the speed can become negative at times (but the model does not allow for that). An instability arises when the speed is negative, and the position and speed diverge into negative infinity.

To overcome this problem, whenever we predict a negative speed we will set it equal to zero and work out way from there:

$$\begin{cases} v(t+\Delta t) \approx v(t) + a(t)\cdot \Delta t \\ v(t+\Delta t) = 0 \end{cases} \implies v(t) + a(t)\cdot \Delta t = 0$$

$$\implies a(t) = -\frac{v(t)}{\Delta t}$$

$$\begin{cases} x(t+\Delta t) \approx x(t) + v(t)\cdot \Delta t + a(t)\cdot \frac{\Delta t^2}{2} \\ a(t) = -\frac{v(t)}{\Delta t} \end{cases} \implies x(t+\Delta t) \approx x(t) + v(t)\cdot \Delta t - \frac{v(t)}{\cancel{\Delta t}}\cdot \frac{\Delta t^{\cancel{2}}}{2}$$

$$\implies x(t+\Delta t) \approx x(t) + v(t)\cdot \Delta t - v(t)\cdot \frac{\Delta t}{2}$$

$$\implies x(t+\Delta t) \approx x(t) + \frac{1}{2} v(t)\cdot \Delta t$$

$$\implies \boxed{x(t+\Delta t) = x(t) - \frac{1}{2}\frac{v^2(t)}{a(t)}}$$

In code, this is implemented as follows:

```
if self.v + self.a*dt < 0:
  self.x -= 1/2*self.v*self.v/self.a
  self.v = 0
else:
  self.v += self.a*dt
  self.x += self.v*dt + self.a*dt*dt/2
```

negative_speed.py hosted with ❤ by GitHub view raw

To calculate the IDM acceleration, we will denote the lead vehicle as `lead` and calculate the interaction term (denoted `alpha`) when `lead` is not `None` .

```
alpha = 0
if lead:
  delta_x = lead.x - self.x - lead.l
  delta_v = self.v - lead.v
  alpha = (self.s0 + max(0, self.T*self.v + delta_v*self.v/self.sqrt_ab)) / delta_x
self.a = self.a_max * (1-(self.v/self.v_max)**4 - alpha**2)
```

lead.py hosted with ❤ by GitHub view raw

If the vehicle is stopped (e.g. at a traffic light), we will use the damping equation:

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

damping.py hosted with ❤ by GitHub view raw

Then, we combine everything together in an `update` method inside a `Vehicle` class:

```
import numpy as np

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

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

        # Calculate properties
        self.init_properties()

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

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

```

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

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

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

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

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

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

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

    def stop(self):
        self.stopped = True

    def unstop(self):
        self.stopped = False

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

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

vehicle.py hosted with ❤ by GitHub view raw

In the `Road` class, we will add a `deque` (double-ended queue) to keep track of vehicles. A queue is a better data structure to store vehicles because the first vehicle in the queue is the farthest one down the road, it is the first one that can be removed from the queue. To remove the first item from a `deque` , we can use `self.vehicles.popleft()` .

We will add an `update` method in the `Road` class:

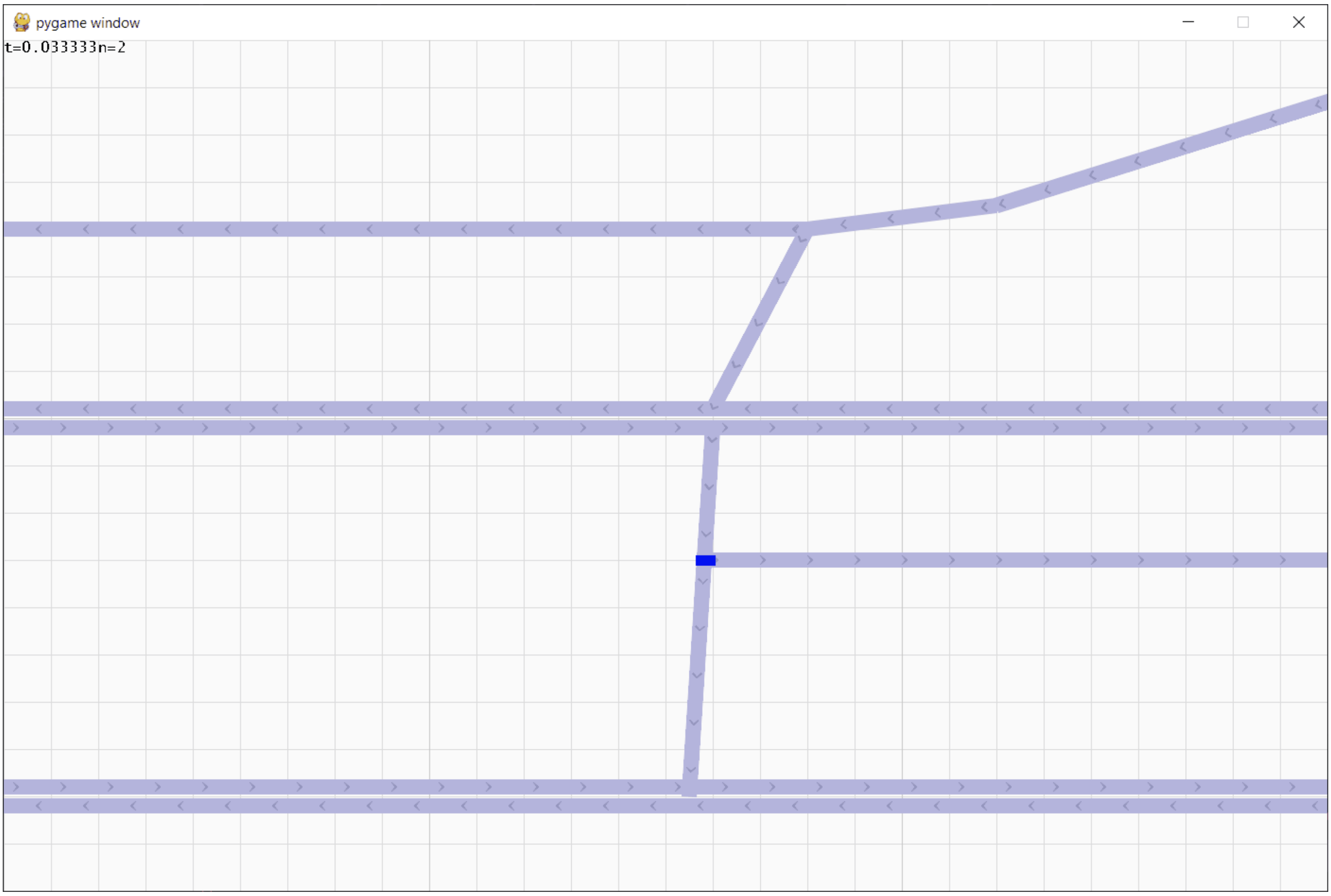
```
def update(self, dt):
  n = len(self.vehicles)

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

road.py hosted with ❤ by GitHub view raw

And an `update` method in the `Simulation` class:

Going back to the `Window` class, I added a `run` method to update the simulation in real-time:

For now, we will add vehicles manually:



Vehicles are moving! Image by Author.

Vehicle Generators

A `VehicleGenerator` has an array of tuples containing `(odds, vehicle)` .

The first element of the tuple is the weight (not probability) of generating the vehicle in the same tuple. I used weights because they are easier to work with since we can just use integers.

For example, if we have 3 vehicles with weights `1` , `3` , `2` . This corresponds to `1/6` , `3/6` , `2/6` with `6=1+3+2`.

To implement this, we use the following algorithm

- Generate a number **r** between 1and the sum of all weights.
- While **r** is non-negative:
  Loop through all possible vehicles and subtract its weight in every iteration.
- Return the last used vehicle.

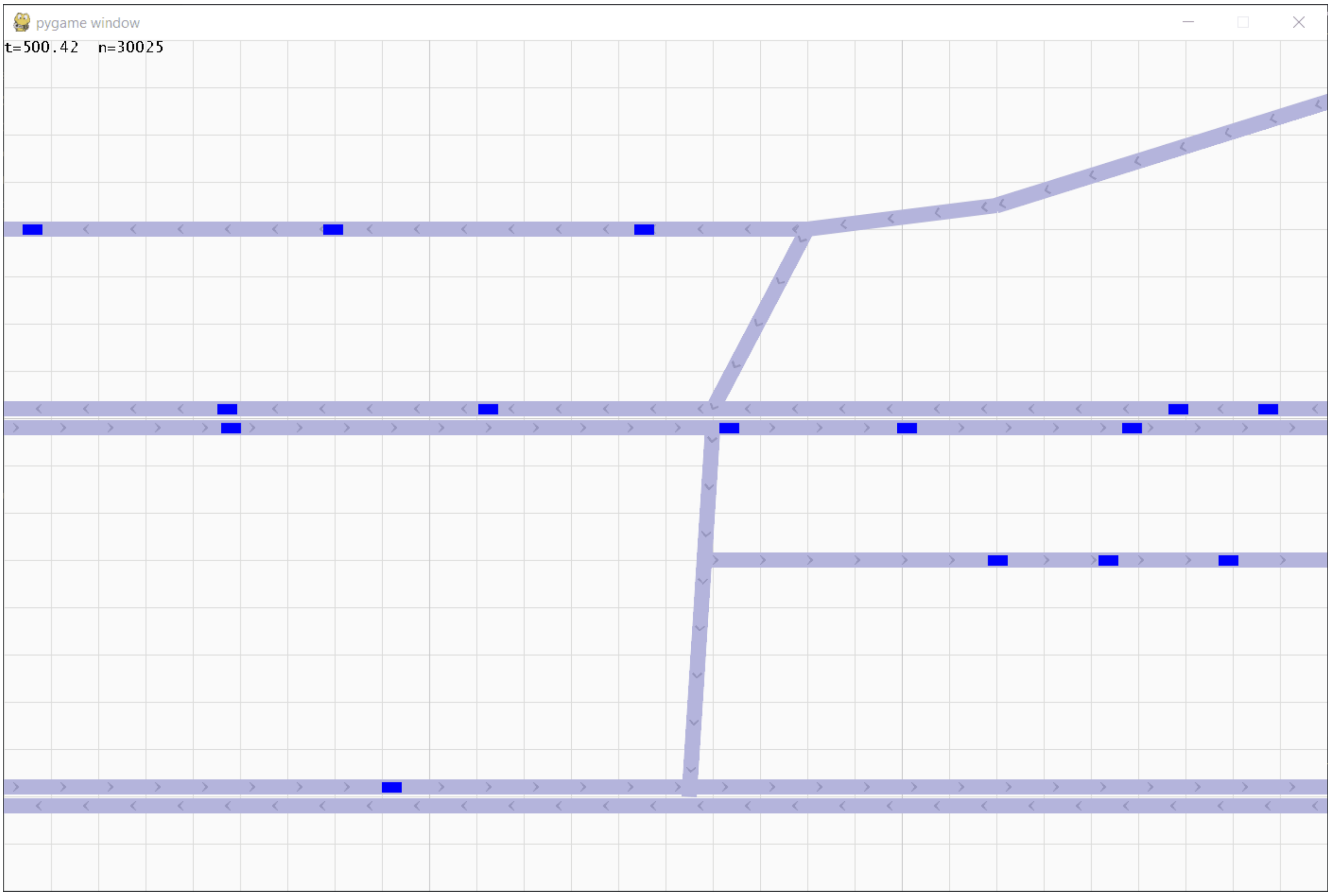
If we have weights: **W1**, **W2**, **W3**. This algorithm allocates numbers between **1** and **W1** to the first vehicle, numbers between **W1** and **W1+W2** to the second vehicle, and numbers between **W1+W2+W3** to the third vehicle.

As to when to add a vehicle, a property called `last_added_time` is updated to the current time every time the generator adds a vehicle. When the time duration between the current time and `last_added_time` is greater than the period of vehicle generation, a vehicle is added.

The period of adding vehicles is `60/vehicle_rate` , because `vehicle_rate` is in *vehicles per minute* and `60` is 1 minute or 60 seconds.

We also have to check whether the road has any space left to add the upcoming vehicle. We do this by checking the distance between the last vehicle in the road and the sum of the length and safety distance of the upcoming vehicle.

Finally, we should `update` vehicle generators by calling the `update` method from `Simulation` ‘s `update` method.



Vehicles are spawning! Image by Author.

Traffic Lights

The default properties for a traffic signal are:

`self.cycle` is an array of tuples containing the states (`True` for green and `False` for red) for every road set in `self.roads` .

In the default configuration, `(False, True)` means the first set of road is red and second is green. `(True, False)` is the opposite.

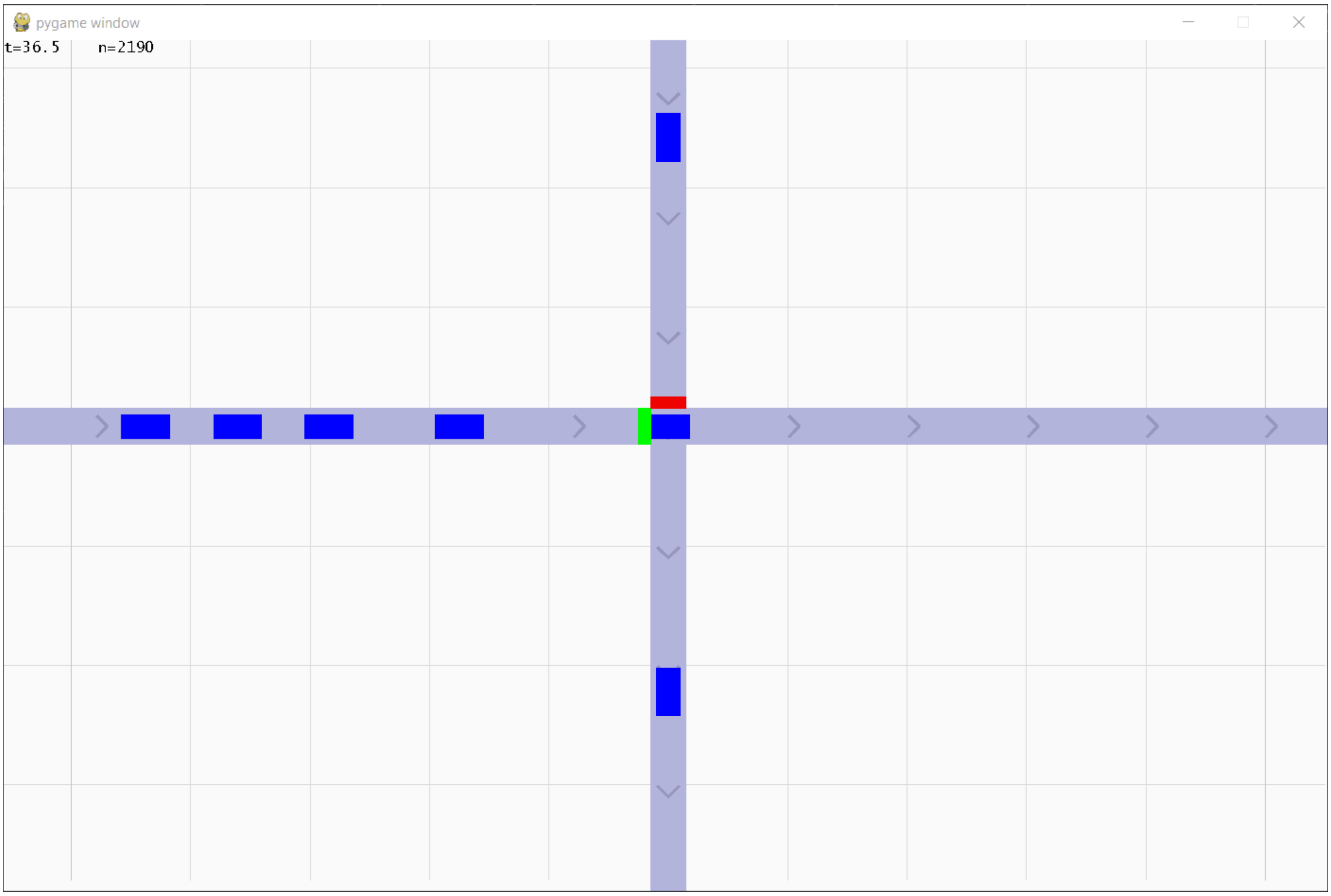
This approach is used because it is easily scalable. We create traffic lights that include more than 2 roads, traffic lights with separate signals for right and left turns, or even for synchronized traffic signals across multiple intersections.

The `update` function of a traffic signal is supposed to be customizable. Its default behavior is symmetric fixed-time cycling.

We need to add these methods to the `Road` class:

And this, in the `update` method of `Road` .

And check for traffic light state in `Simulation` ‘s `update` method:



Stop! Image by Author.

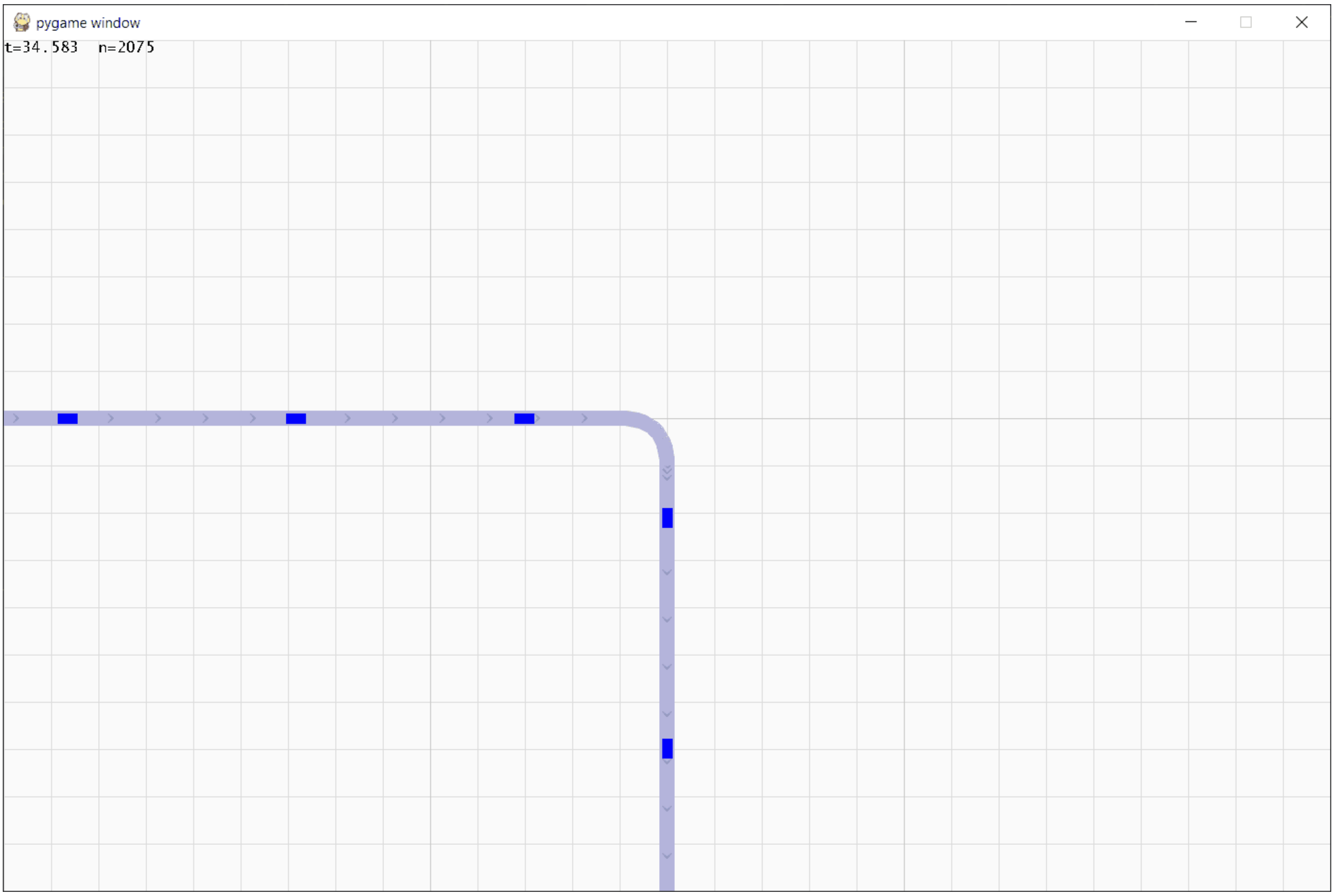
Curves

In the real world, roads have curves. While we can, technically, create curves in this simulation by hand-writing the coordinates of a lot of roads to approximate a curve, we can do the same thing procedurally.

I will use Bézier curves for this.

I created a `curve.py` file that contains functions helping in creating curves and referencing them by their road indices.

Test:

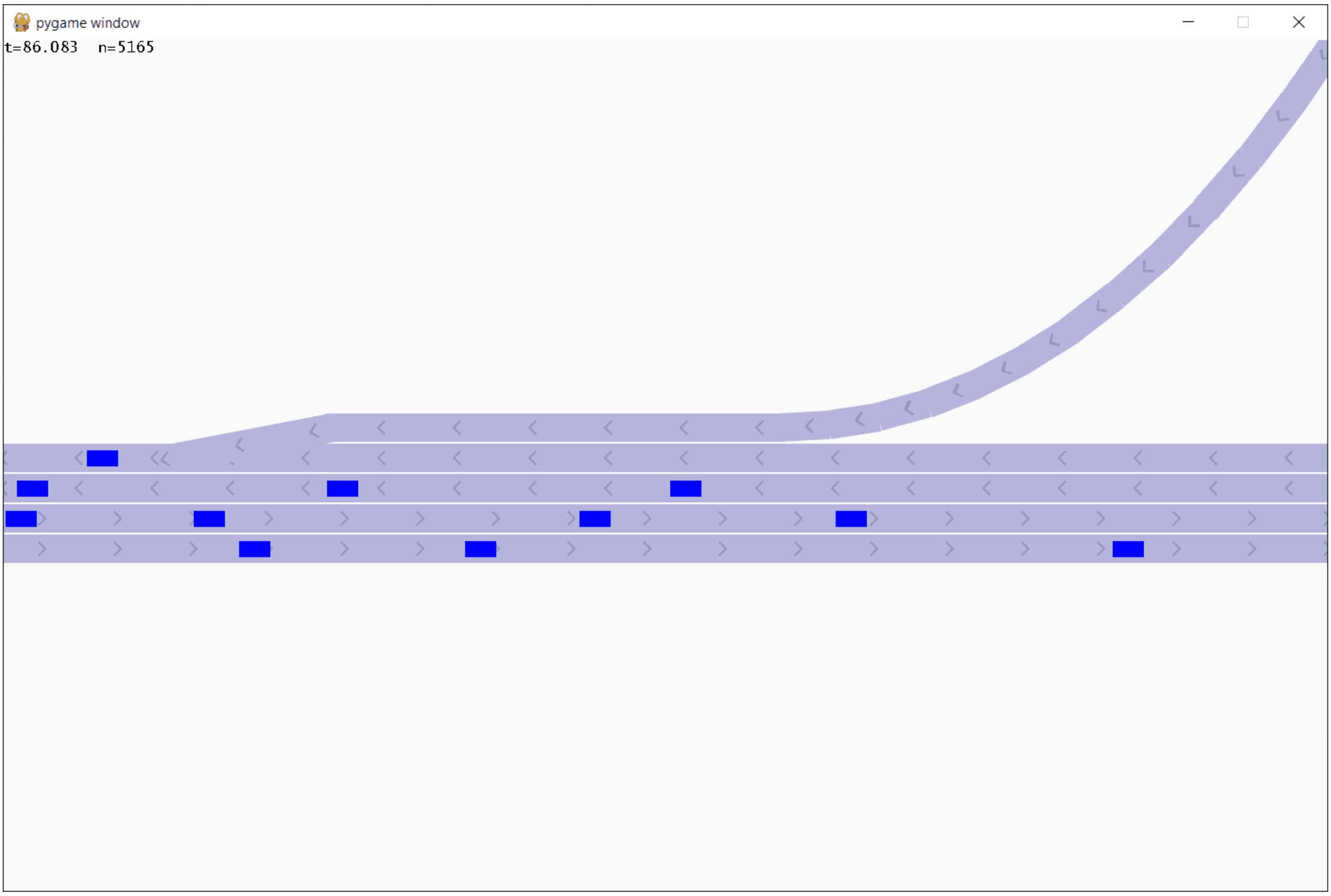


Beautiful curves. Image by Author.

Examples

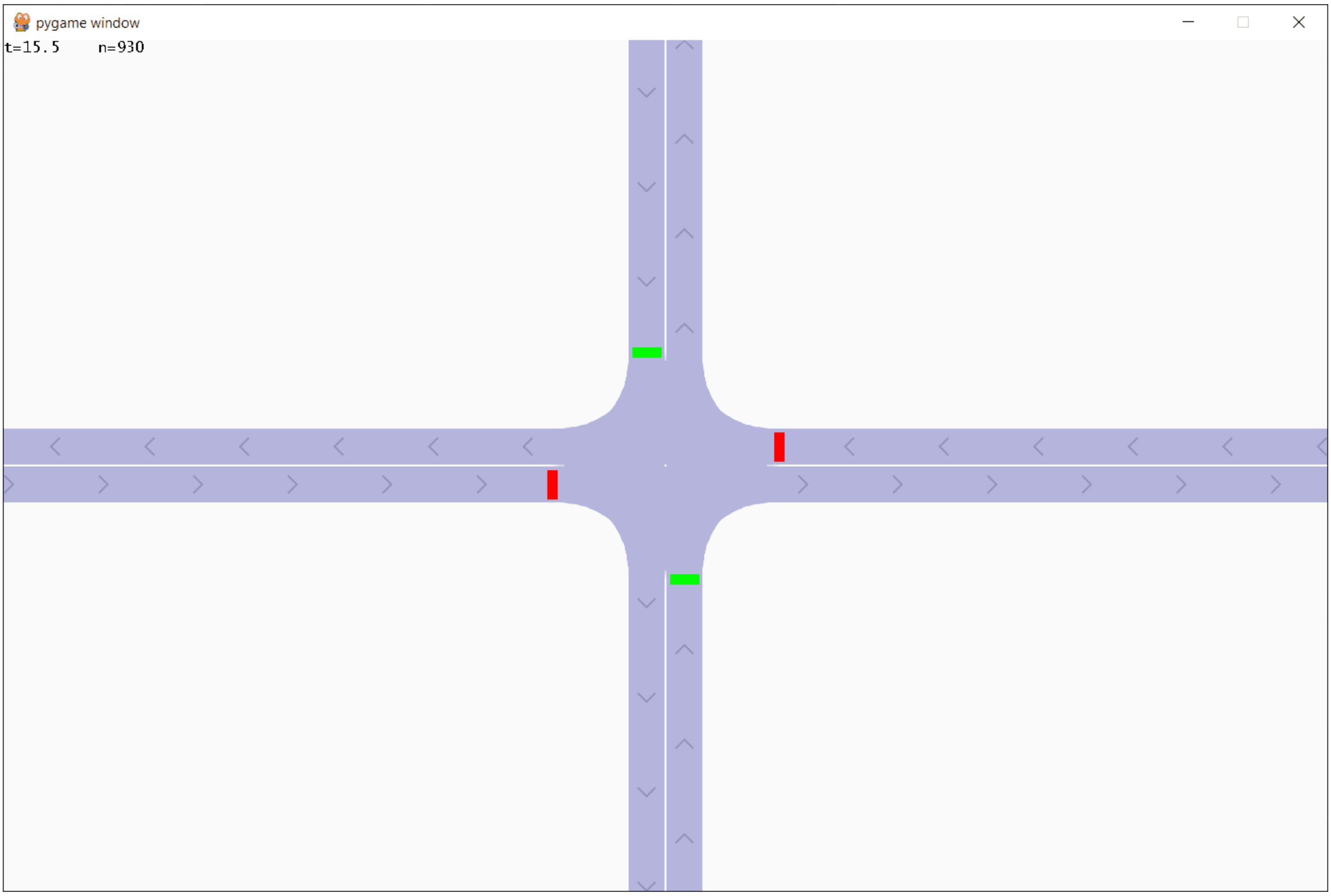
The code for these examples is available in the Github repository linked at the bottom of the article.

Highway Onramp



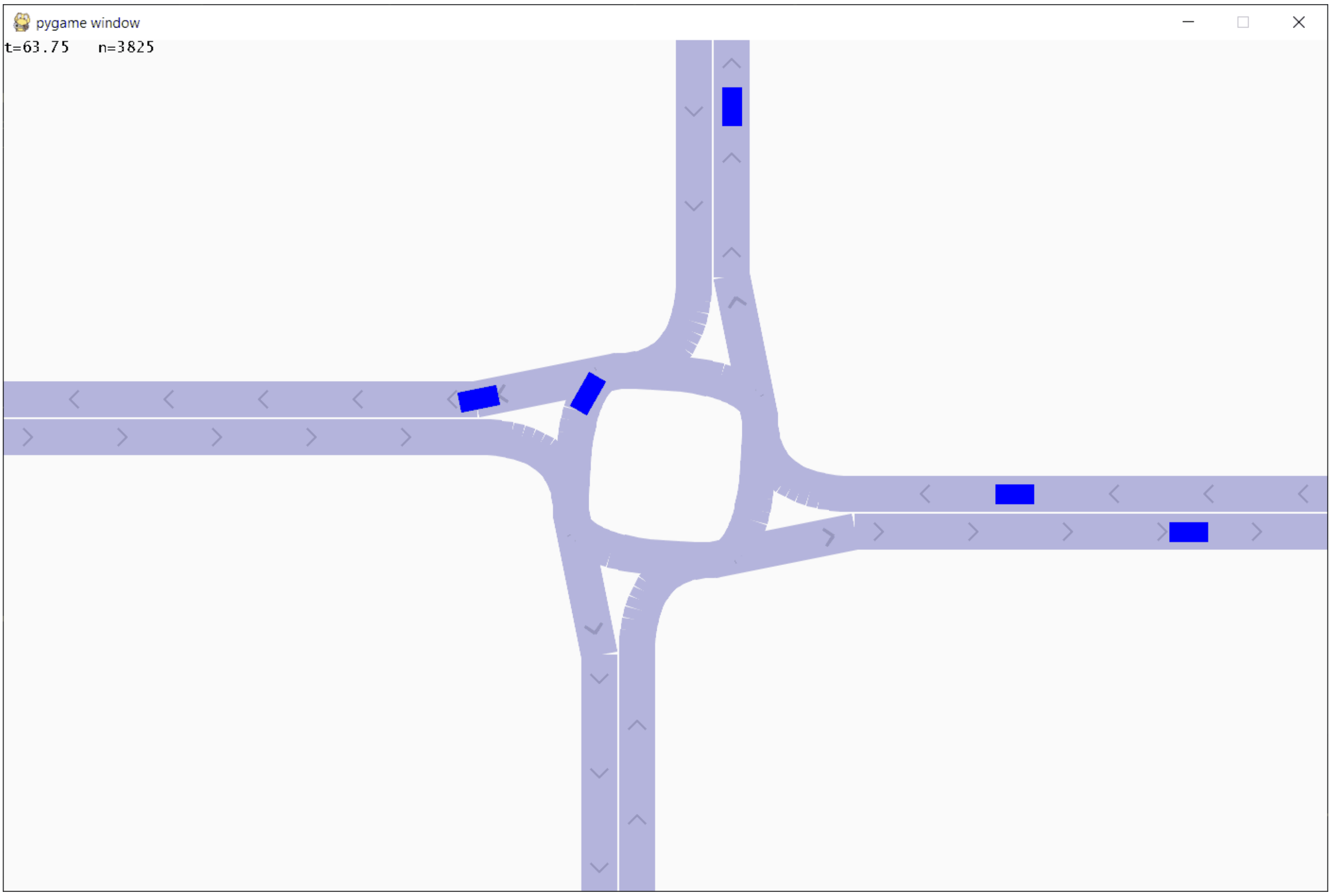
Highway onramp. Image by Author.

Two-Way Intersection



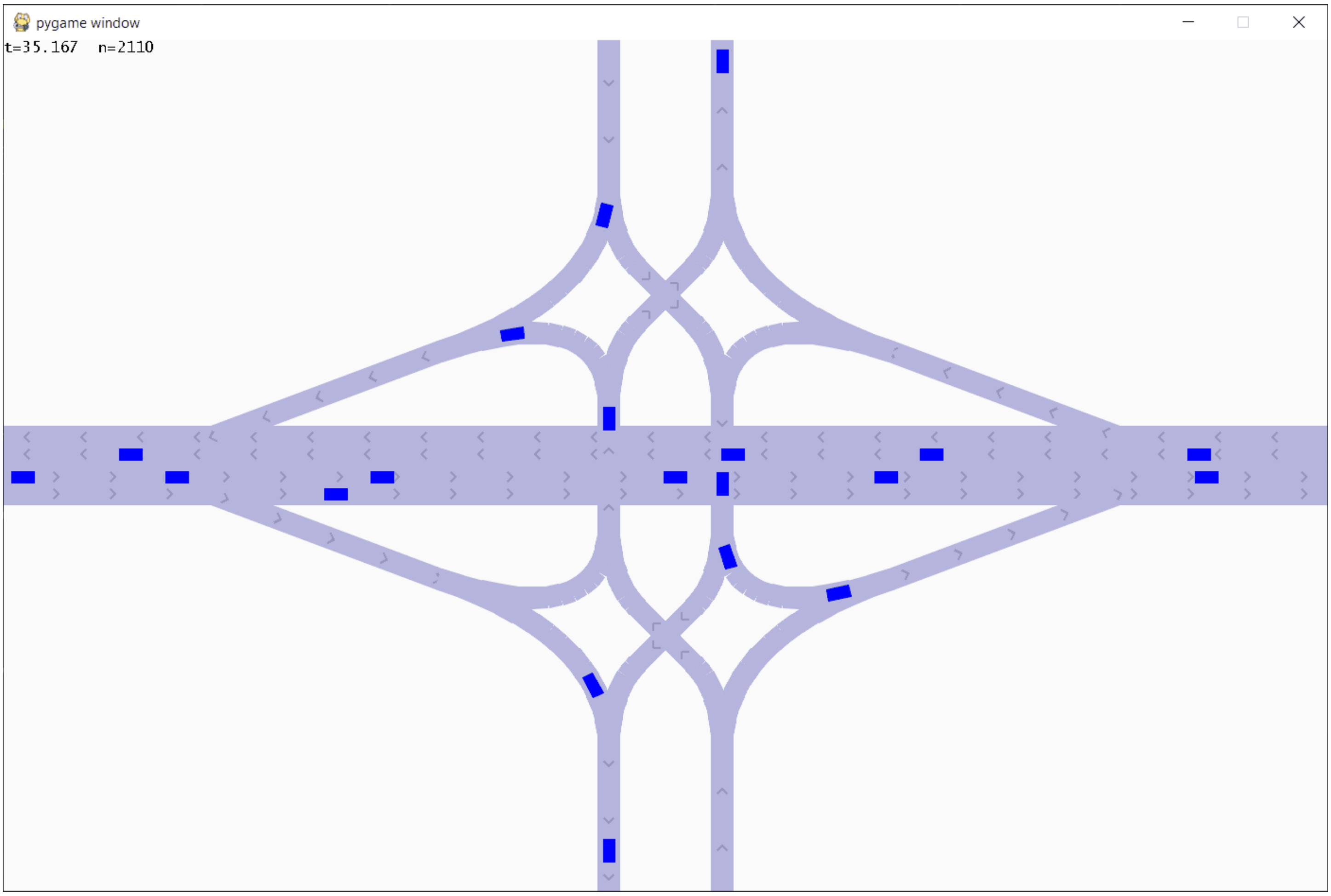
Two-way intersection. Image by Author.

Roundabout



Roundabout. Image by Author.

Diverging Diamond Interchange



Diverging diamond interchange. Image by Author.

Limitations

While we can modify the `Simulation` class to store data about our simulation that we can use later, it would be better if the data gathering process was more streamlined.

This simulation is still lacking a lot. The implementation of curves is bad and inefficient and causes problems with interactions between vehicles and traffic signals.

While some may consider the Intelligent Driver Model a little bit overkill, it is important to have a model that can replicate real-world phenomena like traffic waves (a.k.a. ghost traffic snakes) and the effects of the reaction time of drivers. For this reason, I opted to use the Intelligent Driver Model. But for simulation where accuracy and extreme realism are not important, like in video games, the IDM could be substituted by a simpler logic-based model.

Relying fully on simulation-based data increases the risk of over-fitting. Your ML model could be optimizing for treats present only in the simulation and absent in the real world.

Conclusion

Simulation is an important part of data science and machine learning. Sometimes, gathering data from the real world is either not possible or costly. And generating data helps build huge datasets at a somewhat better price. Simulation can also help fill in the gaps in real-world data. On some occasions, real-word datasets lack edge cases that may be critical to the developed model.

This simulation was part of an undergraduate school project I worked on. The purpose was to optimize traffic signals in urban intersections. I made this simulation as a way to test and validate my optimization methods.

I never thought about publishing this article until I was watching Tesla's AI day, in which they talked about how they use simulation to generate data for edge cases.

Source Code And Contributions

Here is a link to a **Github repository** with all the code in this article including the examples.

If you have any questions or issues with the code, don't hesitate to contact me or submit a pull request or an issue on GitHub.

Data Science Programming Python Editors Pick Hands On Tutorials