ENGG1001: Programming for Engineers

> Assignment 3: Traffic simulation

Due: 1600, 28 Oct, 2022

Aim

The aim of this assignment is to synthesise your understanding of NumPy `ndarrays` and object oriented programming by applying them in an engineering problem, namely the simulation of traffic in a roadway section. Your solutions are to be formally implemented in Python.

Learning Objectives

This assignment supports the following learning objectives, as listed in the Electronic Course Profile:

1 Differentiate and apply program constructs such as variables, selection, iteration and sub-routines.

2 Recognise and apply basic object-oriented design methodologies, along with associated concepts such as classes, instances and method.

4 Interpret an engineering problem and design an algorithmic solution to the problem.

6 Apply techniques for program testing and debugging.

7 Apply sound programming techniques to the solution of real world engineering problems.

8 Analyse and visualise engineering data.

Background

Traffic simulation is essentially a mathematical model of real-world traffic. Traffic simulation models are useful to test alternative scenarios (infrastructure plans, control policies, etc.), to understand travel behaviour and to forecast traffic.

Traffic simulation can be implemented at different modelling levels. Macroscopic models incorporate simplified flow dynamics and allow modelling of large networks with high computational efficiency. Microscopic models emulate the dynamics of individual vehicles in a detailed network representation, based on car-following, lane changing behaviour. In this project, we will focus on car-following behaviour of individual vehicles on a roadway section.

The Problem

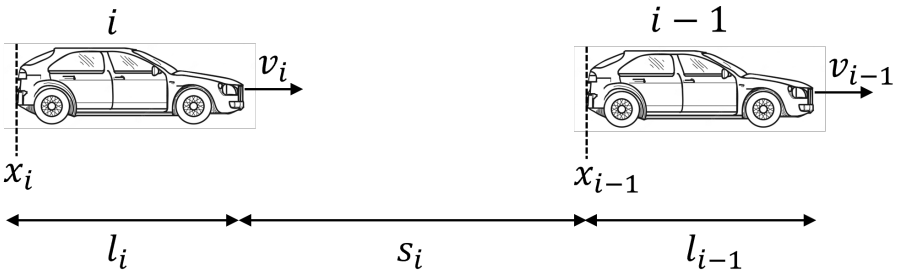


Figure 1: *Sketch of a car-following model with associated variables.*

A microscopic traffic model describes the behaviour of individual drivers/vehicles. The i-th vehicle follows the (i-1)-th vehicle. For the i-th vehicle, denote the position of the rear bumper by x_i, its speed by v_i, and its

length by l_i. Also, denote by s_i the bumper-to-bumper distance or spacing and Δv_i the velocity difference between the i-th vehicle and the vehicle in front of it (vehicle number i-1).

$$s_i = x_{i-1} - x_i - l_i \tag{1}$$

$$\Delta v_i = v_i - v_{i-1} \tag{2}$$

Treiber et al. (2000) [1] has developed a model to describe the car following behaviour, named Intelligent Driver Model (IDM). This model describes the acceleration of the i-th vehicle as a function of its own variables and those of the vehicle in front of it. The dynamics equation is defined as:

$$\frac{\mathrm{d}v_i}{\mathrm{d}t} = a_{0,i}\left(1 - \left(\frac{v_i}{v_{0,i}}\right)^4 - \left(\frac{s^*(v_i, \Delta v_i)}{s_i}\right)^2\right) \tag{3}$$

where

$$s^*(v_i, \Delta v_i) = s_{0,i} + v_i T_i + \frac{v_i \Delta v_i}{2\sqrt{a_{0,i} b_i}} \tag{4}$$

where

- $s_{0,i}$: the minimum desired distance between vehicle i and $i-1$
- $v_{0,i}$: the maximum desired speed of vehicle i
- $a_{0,i}$: the maximum acceleration of vehicle i
- T_i: the reaction time of vehicle i
- b_i: the maximum comfortable deceleration of vehicle i

When the vehicle in front is far away, that is the distance s_i becomes much larger than the desired distance s*, the last term in Eq. 3 goes to zero. Therefore, for the lead vehicle on a roadway section, Eq. 3 can be simplified as:

$$\frac{\mathrm{d}v_i}{\mathrm{d}t} = a_{0,i}\left(1 - \left(\frac{v_i}{v_{0,i}}\right)^4\right) \tag{5}$$

Additionally, when vehicle i enters a stopping zone, where they are expected to come to a full stop because of traffic lights or traffic signs, the change in speed is given as:

$$\frac{\mathrm{d}v_i}{\mathrm{d}t} = -b_i \frac{v_i}{v_{0,i}} \tag{6}$$

The equations above describe the continuous car-following behaviour of vehicles. An integration scheme is necessary for an approximate numerical solution of the system of equations. Assuming a constant time step Δt, a simple but efficient approximation can be calculated. The position and speed of vehicle i in the next time step is approximated as:

$$x_i(t + \Delta t) \approx x_i(t) + v_i(t)\Delta t + a_i(t)\frac{\Delta t^2}{2} \tag{7}$$

$$v_i(t + \Delta t) \approx v_i(t) + a_i(t)\Delta t \tag{8}$$

Since this is an approximation, it may lead to conditions in which speed (i.e., $v(t + \Delta t)$) becomes negative. To avoid these conditions, whenever we predict a negative speed value, we will set it to zero and recalculate

[1] Treiber, M., Hennecke, A., Helbing, D. (2000). Congested traffic states in empirical observations and microscopic simulations. Physical review E, 62(2), 1805.

the position as follows:

$$\begin{cases} v_i(t+\Delta t) < 0 \\ \implies v_i(t+\Delta t) = 0 \\ \implies v_i(t) + a_i(t)\Delta t = 0 \\ \implies a_i(t) = -\frac{v_i(t)}{\Delta t} \end{cases} \tag{9}$$

$$x_i(t+\Delta t) \approx \begin{cases} x_i(t) + v_i(t)\Delta t + a_i(t)\frac{\Delta t^2}{2} \\ x_i(t) + v_i(t)\Delta t - \frac{v_i(t)}{\Delta t}\frac{\Delta t^2}{2} \\ x_i(t) + \frac{1}{2}v_i(t)\Delta t \\ x_i(t) - \frac{1}{2}\frac{v_i^2(t)}{a_i(t)} \end{cases} \tag{10}$$

Implementation

The implementation of the assignment consists of two parts; the first part will only involve procedural programming and NumPy `ndarrays`, while the second part will be the implementation of the same problem through object oriented programming. Note that the second part of the assignment will be mostly about reuse of the existing functions in the context of object oriented programming.

Task 1

Write a function vehicle_update that accepts the parameters of vehicle i and vehicle in front $i-1$ as well as time step dt (i.e., Δt) and the simulation time ts. The function should compute and return the parameters of vehicle i in the next time step. This function should satisfy the criteria described in the bullet points below.

```
>>> def vehicle_update(x: float, v: float, a: float, front_x: float, front_v: float, dt:
↪   float, ts: float):
    """
    Parameters:
        x: position of vehicle i [m]
        v: speed of vehicle i [m/s]
        a: acceleration of vehicle i [m/s2]
        front_x: position of the vehicle in front, i-1 [m]
        front_v: speed of the vehicle in front, i-1 [m/s]
        dt: time step [s]
        ts: time of simulation [s]
    Returns:
        x: position of vehicle i in the next time step [m]
        v: speed of vehicle i in the next time step [m/s]
        a: acceleration of vehicle i in the next time step[m/s2]
    """
    l = 6.0 # vehicle length [m]
    s0 = 4.0 # max desired spacing [m]
    T = 1.0 # reaction time [s]
    v_max = 19.44 # maximum desired speed [m/s]
    a_max = 1.50 # maximum acceleration [m/s2]
    b_max = 4.10 # maximum comfortable deceleration [m/s2]

```

- The order in which the position, velocity and acceleration are updated is crucial. First, the position should be updated based on Eq. 7. Second, the speed should be updated based on Eq. 8. Third, the acceleration should be computed with the updated position and velocity values according to Eq. 3 - 5.

```
>>> x = 100; v = 16; a = 0.5; front_x = 130; front_v = 19.44; dt = 0.1; ts = 30;
>>> x, v, a = vehicle_update(x, v, a, front_x, front_v, dt, ts)
>>> x, v, a
```

```
(101.6025, 16.05, 0.5565165058179474)
```

- If vehicle i is the lead vehicle on the roadway (i.e., no vehicle further ahead), the corresponding parameters for vehicle $i-1$ should be np.nan.

```
>>> x = 100; v = 16; a = 0.5; front_x = np.nan; front_v = np.nan; dt = 0.1; ts = 20;
>>> x, v, a = vehicle_update(x, v, a, front_x, front_v, dt, ts)
>>> x, v, a
(101.6025, 16.05, 0.8030423912930567)
```

- Additionally, in this scenario, we assume that the lead vehicle on the roadway enters a stopping zone associated with a traffic signal at t=30s, and the traffic signal turns green at t=60s.

```
>>> x = 300; v = 19.44; a = 0; front_x = np.nan; front_v = np.nan; dt = 0.1; ts = 30;
>>> x, v, a = vehicle_update(x, v, a, front_x, front_v, dt, ts)
>>> x, v, a
(301.944, 19.44, -4.1)
```

Task 2

Write a function road_update that computes and updates the position, velocity and acceleration of all vehicles on the roadway section. The function should accept the parameters of vehicles in the current time t, time step Δt, the time of simulation and roadway length. In this new function, you should call the previous function that you have built. This function should satisfy the criteria described in the bullet points below.

```
>>> def road_update(vehs_x: np.array, vehs_v: np.array, vehs_a: np.array, dt: float, ts:
↪ float, l_road: float):
    """
    Parameters:
        vehs_x: array of vehicle positions [m] - (Nveh x 1)
        vehs_v: array of vehicle speeds [m/s] - (Nveh x 1)
        vehs_a: array of vehicle accelerations [m/s2] - (Nveh x 1)
        dt: time step [s]
        ts: time of simulation [s]
        l_road: roadway length [m]
    Returns:
        none
    """
```

- The position, velocity and acceleration arrays should be sorted with respect to the vehicle positions, where the first entry belongs to the lead vehicle on the road and the last entry belongs to the last vehicle on the section. Similarly, the position, velocity and acceleration attributes should be updated with respect to the vehicle positions on the road; i.e. the lead vehicle should be updated first, the second vehicle should be updated next, etc.

```
>>> vehs_x = np.array([115.0, 85.0, 45.0]); vehs_v = np.array([19.44, 18.0, 16.0]);
↪ vehs_a = np.array([0, 0.5, 1]); dt = 0.1; ts = 100; l_road = 200
>>> road_update(vehs_x, vehs_v, vehs_a, dt, ts, l_road)
>>> vehs_x, vehs_v, vehs_a
(array([116.944 ,  86.8025,  46.605 ]), array([19.44, 18.05, 16.1 ]), array([ 0.
↪ , -0.35790762,  0.55110751]))
```

- If a vehicle position exceeds the roadway length, then the corresponding outputs should be np.nan.

```
>>> vehs_x = np.array([199.0, 85.0, 45.0]); vehs_v = np.array([19.44, 18.0, 16.0]);
 ↪  vehs_a = np.array([0, 0.5, 1]); dt = 0.1; ts = 100; l_road = 200
>>> road_update(vehs_x, vehs_v, vehs_a, dt, ts, l_road)
>>> vehs_x, vehs_v, vehs_a
(array([    nan, 86.8025, 46.605 ]), array([  nan, 18.05, 16.1 ]), array([  nan,
 ↪  0.38515358, 0.55110751]))
```

Task 3

Write a function named `simulation` that computes and returns positions and velocities of all vehicles at all time steps. The function should accept the time step Δt, total simulation duration T_{sim}, total number of vehicles to enter the roadway N_{veh}, headway between vehicles $hway$, and length of road l_{road}. Note that $v_{0,i}$ or `v_max` is given as a hard-coded value as described below.

```
>>> def simulation(dt: float, Tsim: float, Nveh: int, hway: int, l_road: float):
    """
    Parameters:
        dt: time step  [s]
        Tsim: total simulation time [s]
        Nveh: total number of vehicles [veh]
        hway: time gap between vehicle entries [number of time steps]
        l_road: roadway length [m]
    Returns:
        pos: array of updated vehicle positions at all time steps [m] - (Nveh x Nstep)
        speed: array of updated vehicle speeds at all time steps [m/s] - (Nveh x Nstep)
    """
    v_max = 19.44 # maximum desired speed [m/s]
```

- The vehicles will enter the roadway at $x = 0$ and with constant headways of $hway$. Note that the unit for $hway$ is the number of time steps Δt, not seconds. That means, at each $hway$ steps, a new vehicle will enter the roadway. Assume $v_i(t) = v_{0,i}$ and $a_i(t) = 0$ for all new vehicles $\forall i$ at the entry time t.
- Only N_{veh} vehicles can enter the roadway, and the first vehicle enters at $t = 0$.
- The size of *pos* and *speed* arrays is Nveh x Nstep, where Nstep is the total number of time steps needed for the simulation time T_{sim}.
- The corresponding entries in *pos* and *speed* arrays for the vehicles that have not yet entered the roadway or that have left the roadway should be `np.nan`.

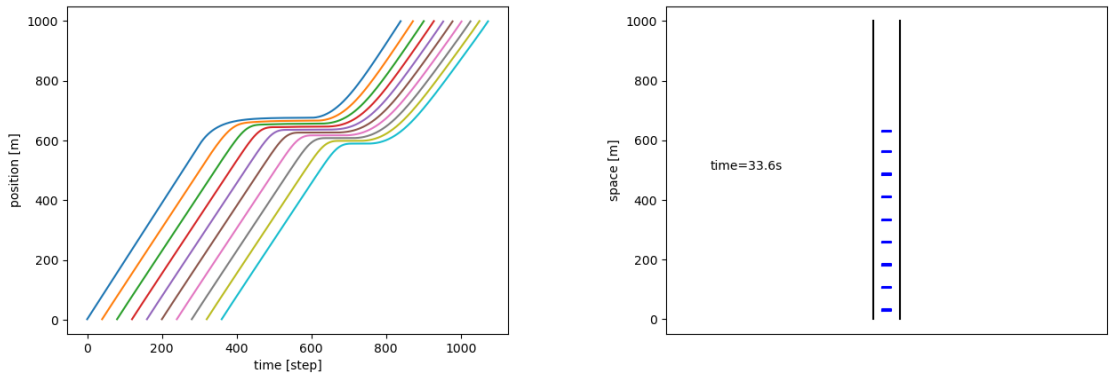
Task 4

Write a function named `plot_positions` that accepts the array *pos*. The plotting function does not return anything, but should produce a plot of the vehicle positions over time.

```
>>> def plot_positions(pos: np.array)::
    """
    Parameters:
        pos: array of updated vehicle positions at all time steps [m] - (Nveh x Nstep)
    """
```

```
>>> dt = 0.1; Tsim = 120; Nveh = 10; hway = 40; l_road = 1000;
>>> pos, speed = simulation(dt, Tsim, Nveh, hway, l_road)
>>> plot_positions(pos)
>>> fig, ax = plt.subplots()
>>> create_animation(pos, ax)
```

The script above should create a plot of vehicle positions over time as shown in Figure 2(a) and an animation of traffic simulation, an instance of which is shown in Figure 2(b). Note that `create_animation` function is already provided to you.



(a) vehicle positions over time

(b) an instance from traffic simulation

Figure 2: *Example plots from simulation.*

Discussion

At this point, you are actually done with the development of the traffic simulation model. However, we will proceed to develop this in a more rigorous manner using object oriented programming.

In terms of code design, you might notice up to this point that there has been a lot of repetition of the simulation parameters (e.g., dt, ts) as arguments to the functions. Many other parameters have been hard-coded to the functions to avoid repetition. Additionally, this code design does not offer flexible changes. For example, all the vehicles are subject to the same parameters (e.g., desired maximum speed, max acceleration, etc), which is not realistic. This is a hint to us that we could capture those parameters as the state of some object. Then we would be able to pass a single object to functions rather than a long list of parameters. An object-oriented design would help here. In the next part of this assignment, you will rebuild part of the code in object-oriented form. You can and should leave your existing functions in place.

Task 5

Write a class called `Vehicle` to represent the vehicle. At a minimum, your class should contain the following methods. You may write additional helper methods to aid your program design.

`__init__()` an initialiser method to store the parameters of the vehicle class (as given below) and the initial values for position, speed and acceleration at the entry to the roadway section. Additionally, this class should include a boolean variable that represents whether a vehicle is in a stopping zone or not, and this should be initially set to `False`. Assume $v_i(t) = v_{0,i}$ and $a_i(t) = 0$ for all vehicles $\forall i$ at the entry $x = 0$. Differently from the previous implementation, assume $v_{0,i} = v_0 + U(-3, 3)$ where v_0 is equal to `v_max` and $a_{0,i} = a_0 + U(-0.5, 0.5)$ where a_0 is equal to `a_max` in the script below and U stands for uniform distribution. Note that it would have been much more difficult to add such randomness to the individual vehicle parameters in the procedural programming settings.

`update()` a method to update the position, speed and acceleration of vehicles. You can copy and adapt your `vehicle_update()` function from Task 1 and convert to a method. Unlike the `vehicle_update()` function, this method does not need to consider the time interval where the traffic signal is red (between ts=300 and ts=600). Instead, the stopping mechanism should be implemented considering the boolean variable that represents the stopping status of the vehicle. Additionally, `front` should be set to `None` if there is no vehicle in front.

`stop()` a method to change the stopping status of the vehicle (i.e., the boolean variable) to `True`. This method should be called when the lead vehicle enters the stopping zone.

`unstop()` a method to change the stopping status of the vehicle (i.e., the boolean variable) to `False`. This method should be called when the lead vehicle can start moving again.

The remaining methods below are `getter` methods returning the corresponding attributes as described below.

```python
class Vehicle:
    def __init__(self, vehid: int, l: float, s0: float, T: float, v_max: float, a_max:
    ↪ float, b_max: float):
    """
    Parameters:
        l: vehicle length [m]
        s0: max desired spacing [m]
        T: reaction time [s]
        v_max: maximum desired speed [m/s]
        a_max: maximum acceleration [m/s2]
        b_max: maximum comfortable deceleration [m/s2]
        vehid: vehicle id based on the order of entry,
               e.g., the id of the first vehicle is 0, the second is 1, etc.
    Returns:
        none
    """

    def update(self, front: Vehicle, dt: float):
    """
    Parameters:
        front (vehicle): vehicle in front,
        dt: time step [s]
    Returns:
        none
    """

    def stop(self):
    """
    Returns:
        none
    """

    def unstop(self):
    """
    Returns:
        none
    """

    def get_pos(self):
    """
    Returns:
        (float): position of the vehicle
    """

    def get_speed(self):
    """
    Returns:
        (float): speed of the vehicle
    """

    def get_length(self):
    """
    Returns:
        (float): length of the vehicle
    """

```

```
    def get_id(self):
    """
    Returns:
        (int): id of the vehicle
    """
```

Task 6

Write a class called `Road` to represent the road. At a minimum, your class should contain the following methods. You may write additional helper methods to aid your program design.

`__init__()` an initialiser method to store the length of the road and the `list` of vehicles that are currently on the road. This `list` should initially be empty and it should be resized every time a vehicle enters or leaves the road.

`update()` a method to update the vehicle objects that are currently on the roadway section. You can copy and adapt your `road_update()` function from Task 2 and convert to a method. While `road_update()` function makes use of `ndarrays` to store position, speed and acceleration values, `update()` method should consider the `list` of vehicles that consists of `Vehicle` instances.

`add_vehicle()` a method to add a `Vehicle` instance to the `list` of vehicles that are currently on the road. The new vehicle should be added to the end of `list`.

```
class Road:
    def __init__(self, length: float):
    """
    Parameters:
        length: length of roadway section [m]
    Returns:
        none
    """

    def update(self, dt: float):
    """
    Parameters:
        dt: time step [s]
    Returns:
        none
    """

    def add_vehicle(self, veh: vehicle):
    """
    Parameters:
        veh: vehicle object
    Returns:
        none
    """
```

Task 7

Write a class called `Simulation` to represent the simulation environment. At a minimum, your class should contain the following methods. You may write additional helper methods to aid your program design.

`__init__()` an initialiser method to store the simulation parameters as given below. This method should also initialize a `road` object with the given length and append it to the `Simulation` class.

`run()` a method to run the simulation and return the position and speed of all vehicles at all time steps. You can copy and adapt your `simulation()` function from Task 3 and convert to a method. Differently from the `simulation()` function, this method should alter the stopping status of the lead vehicle

assuming it enters a stopping zone associated with a traffic signal at t=30s and the traffic signal turns green at t=60s.

```
class Simulation:
    def __init__(self, dt: float, Tsim: float, Nveh: int, hway: int, l_road: float, l:
    ↪ float, s0: float, T: float, v_max: float, a_max: float, b_max: float):
    """
    Parameters:
        dt: time step  [s]
        Tsim: total simulation time [s]
        Nveh: total number of vehicles [veh]
        hway: time gap between vehicle entries [number of time steps]
        l_road: roadway length [m]
        l: vehicle length [m]
        s0: max desired spacing [m]
        T: reaction time [s]
        v_max: maximum desired speed [m/s]
        a_max: maximum acceleration [m/s2]
        b_max: maximum comfortable deceleration [m/s2]
    Returns:
        none
    """

    def run(self):
    """
    Parameters:
        none
    Returns:
        pos (ndarray): array of updated vehicle positions at all time steps [m] - (Nveh x
    ↪ Tstep)
        speed (ndarray): array of updated vehicle speeds at all time steps [m/s] - (Nveh x
    ↪ Tstep)
    """

```

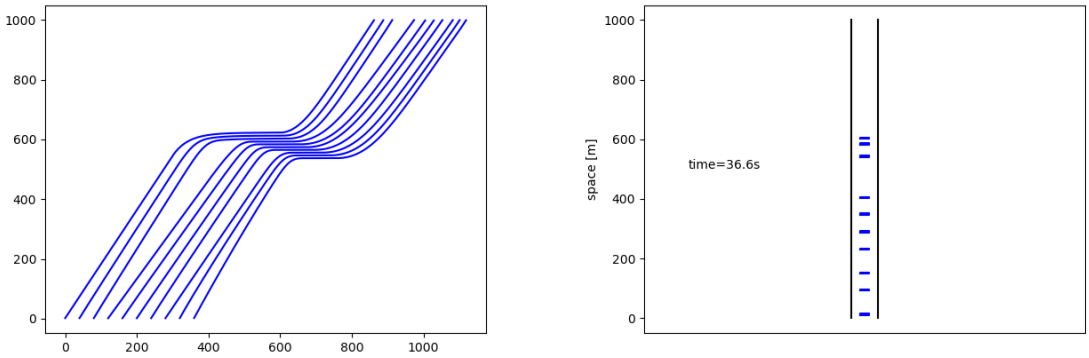
```
>>> dt = 0.1; Tsim = 120; Nveh = 10; hway = 40; l_road = 1000;
>>> l = 4; s0 = 4; T = 1; v_max = 19.44; a_max = 1.5; b_max = 4.1;
>>> sim = Simulation(dt, Tsim, Nveh, hway, l_road, l, s0, T, v_max, a_max, b_max)
>>> pos, speed = sim.run()
>>> plot_positions(pos)
>>> fig, ax = plt.subplots()
>>> create_animation(pos, ax)
```

The script above should create a plot of vehicle positions over time as shown in Figure 3(a) and an animation of traffic simulation, an instance of which is shown in Figure 3(b). Note that your output may not be the same, as the desired maximum speed and maximum acceleration of each vehicle is randomized with a uniform distribution. As expected, the distance between vehicles is not fully homogeneous in this scenario.

Assignment Submission

You must submit your completed assignment electronically through Gradescope. The only file you submit should be a single Python file called **a3.py** (use this name, all in lower case). This should be uploaded to Gradescope at Blackboard>Assessment>Assignment 3. You may submit your assignment multiple times before the deadline, as only the last submission will be marked.

Late submission of the assignment will **not** be accepted. In the event of exceptional personal or medical circumstances that prevent you from handing in the assignment on time, you may submit a request for an extension. See the course profile for details of how to apply for an extension.



(a) vehicle positions over time

(b) an instance from traffic simulation

Figure 3: *Example plots from object oriented programming implementation.*

Requests for extensions **must** be made no later than 48 hours prior to the submission deadline. The application and supporting documentation (e.g. medical certificate) **must** be submitted to the ITEE Coursework Studies office (78-425) or by email to enquiries@itee.uq.edu.au. If submitted electronically, you **must** retain the original documentation for a minimum period of six months to provide as verification should you be requested to do so. Refer to the Electronic Course Profile for more detail.