Embouteillages

Projet informatique

Rapport 2e partie

Vellu Clément – Quici Mipam

Table des matières

Introduction

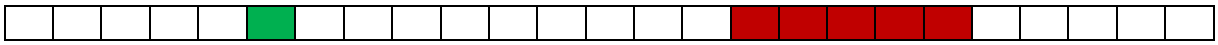
Les embouteillages sont des phénomènes ennuyants que l'on rencontre très couramment sur nos routes et peuvent, dans certains cas, prendre des proportions gigantesques.

Le but de ce projet est donc de modéliser une route avec différents véhicules dessus. La simulation devra, en laissant évoluer assez de véhicules, créer un embouteillage et montrer en même temps l'effet « accordéon » dans le mouvement de masse des véhicules. Il devra aussi être possible de déclencher un embouteillage en insérant un élément perturbateur, mais aussi de trouver des méthodes de résolution de ceux-ci, notamment en permettant aux voitures de changer de voies.

La simulation se basera dans un premier temps sur des modèles simplifiés de la physique, avec une modélisation particulière de la vitesse et de la position des véhicules par exemple, avant d'améliorer le modèle pour qu'il soit plus fiable et proche de la réalité.

1 Pistes et hypothèses

Nous avons envisagé dans un premier temps de modéliser la route par une liste, mais avons retenu l'utilisation des tableau numpy afin de profiter des fonctions de manipulations des tableaux plus avancées et faciles d'utilisation. Les tableaux et l'affectation d'une seule position par véhicule discrétise leur position, éloignant ainsi le modèle de la réalité. Nous avons utilisé ce modèle-ci pour faciliter la création et le fonctionnement de la simulation. Néanmoins, nous utilisons les cellules comme unité de mesure en considérant qu'une cellule mesure environ 2 mètres.



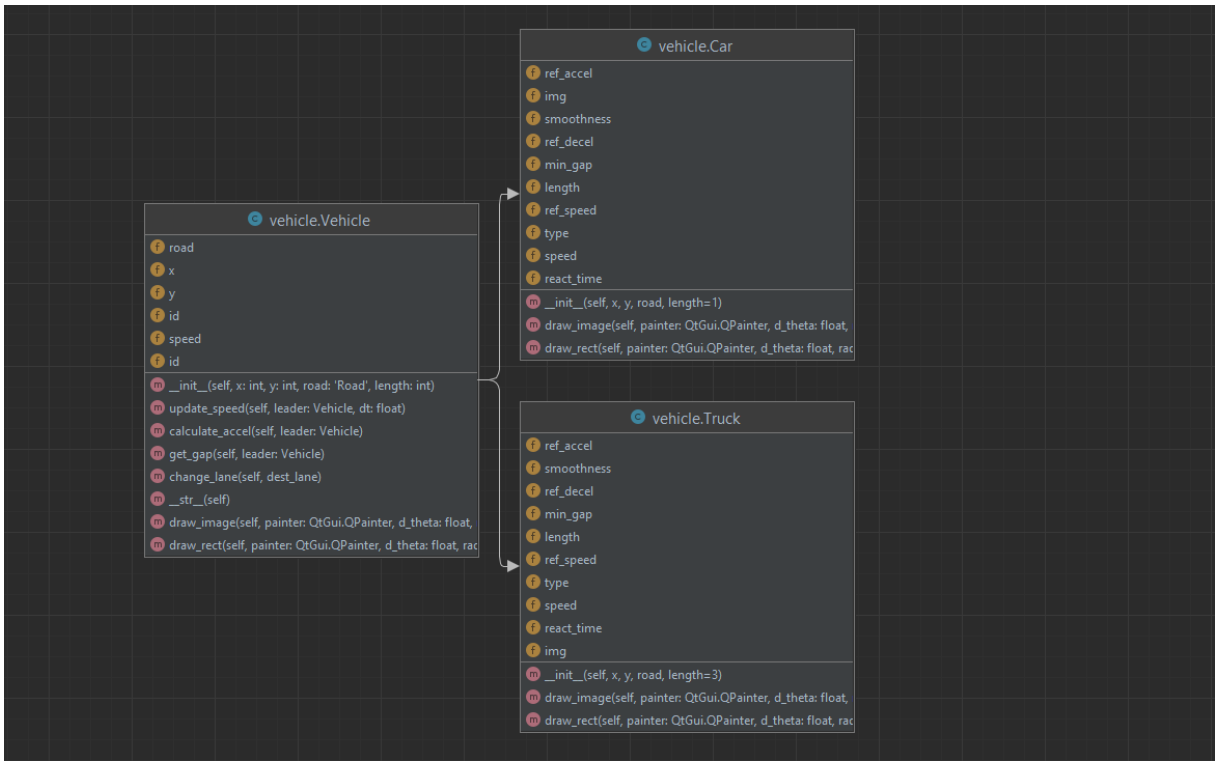
Nous pouvons bien distinguer les cellules du tableau sur le schéma de la route ci-dessus. La modélisation actuelle des véhicules est celle de la case verte, chaque véhicule occupe une seule case. Les 5 cases rouges représentent aussi un seul véhicule mais dont la longueur est plus importante. En pratique, chaque véhicule est stocké dans une seule case.

2 Présentation des classes

2.1 La classe *Vehicle* et les sous-classes *Car* et *Truck*

La classe *Vehicle* est une classe abstraite qui recense les caractéristiques et le comportement des différents véhicules présents sur la route. Le module possède également deux sous-classes qui héritent de *Vehicle* pour les deux types de véhicules : Car et Truck. Les deux types de véhicule ont les caractéristiques communes de la classe *Vehicle*, comme une paire de coordonnée pour leur position. Ils ont néanmoins quelques attributs différents (comme la distance minimale) et des fonctions d'affichages différentes.

La classe *Vehicle* comporte de nombreuses fonctions de calculs car ceux-ci sont propres aux paramètres intrinsèques de chaque véhicule.



2.1.1 La méthode *calculate_accel()*

Cette méthode permet de calculer l'accélération d'un véhicule à partir d'un modèle physique, l'Intelligent Driver Model (IDM). En considérant A la nouvelle accélération, s0 l'écart minimal entre les véhicules, s l'écart normal, v la vitesse, $v0$ la vitesse de référence (80 km/h pour les camions, 130 km/h pour les voitures), Δv l'écart de vitesse, T le temps de réaction, a et b les références de l'accélération et de la décélération et enfin Δ le coefficient d'amortissement, l'accélération est calculée de la manière suivante :

$$S = s0 + \max\left(0; v.T + v.\frac{\Delta v}{2 \times \sqrt{a.b}}\right)$$

Pour arriver à :

$$A = a\left(1 - \left(\frac{v}{v0}\right)^{\Delta} - \left(\frac{S}{s}\right)^{2}\right.$$

Ce modèle physique a pour objectif de prendre en compte les facteurs humains dans la conduite afin de modéliser les effets d'anticipation ou au contraire, de manque d'anticipation.

2.1.2 La méthode *update_speed()*

Cette méthode appelle *calculate_accel()* pour récupérer l'accélération et calcul la nouvelle vitesse ainsi que la nouvelle position d'un véhicule. En considérant dv la nouvelle accélération, v la vitesse d'un véhicule, nv sa nouvelle vitesse, dt une incrémentation de temps, y la position actuelle d'un véhicule, ny sa nouvelle position, L la longueur de la route, la vitesse mise à jour est calculée de la manière suivante :

$$nv = v + dv \times dt$$

Si la vitesse mise à jour est négative elle sera considérée comme nulle. Enfin, nouvelle position est calculée comme indiqué ci-dessous, avec % l'opérateur qui calcul le reste de la division euclidienne.

$$ny = \left\lfloor y + v \times dt + \frac{1}{2} \times dv \times dt^2 \right\rfloor \% L$$

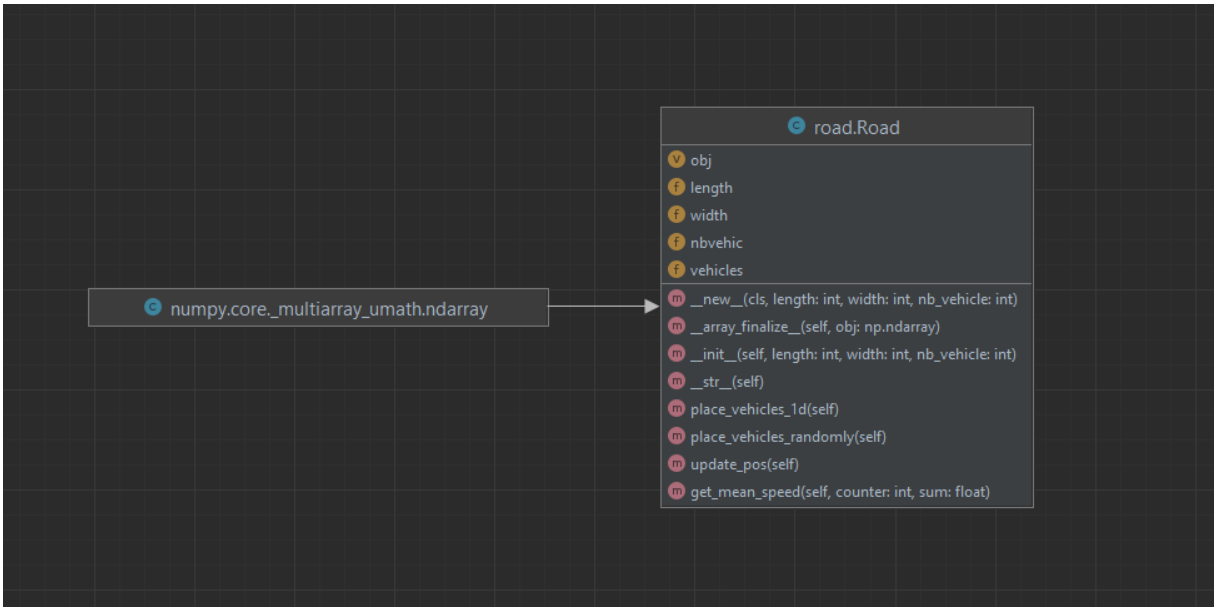
Ces équations découlent de l'approximation d'Euler (aussi appelée approximation balistique) qui est suffisamment précise pour notre problème car il possède une résolution temporelle (incrément dt) relativement faible au regard des valeurs mises en jeu. Il n'y a donc pas de problème de divergence.

2.2 La classe *Road*

La classe *Road* est l'écosystème qui modélise la route sur laquelle les véhicule vont évoluer, elle hérite d'un tableau de numpy. La classe prend en argument la longueur de la route, la largeur qui représente le nombre de voies, et le nombre de véhicules présents sur la route.

Chaque instance possède également une liste des *Vehicle* instanciés lors de l'initialisation (par la méthode place_vehicles_1d) ordonnée dans le sens croissant des index de la position sur la route à l'instant initial.

Ci-dessous le diagramme de classe de *Road* :



2.2.1 La méthode *__new__()*

Cette méthode créer un objet Road et surcharge le constructeur ndarray pour ajouter les variables d'instances dont nous avons besoin pour notre projet. Il prend en argument et ajoute en variables d'instances le nombre de véhicules présents sur la route, la longueur de la route, la largeur (nombre de voies), et une liste vide de véhicules.

2.2.2 La méthode *__array_finalize__()*

Cette méthode permet, lorsqu'un nouvel objet est créé lors de la manipulation des ndarrays, de s'assurer que ce nouvel objet, qui devient l'objet principal, reçoive aussi les variables d'instances.

2.2.3 La méthode *place_vehicles_1d()*

Cette méthode est appelée à chaque fois qu'il faut générer la route avant une simulation. Elle utilise les paramètres de la simulation, la longueur de la route, le nombre de voies et le nombre de véhicules, pour les placer sur la route de façon à les répartir avec le plus d'espace possible entre eux. C'est aussi avec cette

méthode que les véhicules vont se diviser en voitures et en camions. Il est estimé que trois véhicules sur dix sur les autoroutes sont des camions. Donc si le résultat d'un randint(1, 10) est supérieur à 3 le véhicule est placé sur la route en tant que voiture, et à l'inverse il est placé en tant que camion.

2.2.4 La méthode *update_pos()*

Cette méthode utilise la nouvelle position de chaque véhicule, calculée dans la classe *Vehicle* et met à jour la position de ses propres véhicules dans le tableau qui représente la route. Afin d'éviter les dépassements de tableau, lorsque la nouvelle position calculée dépasse la taille du tableau, le véhicule est replacé au début du tableau. On utilise un modulo pour conserver la cohérence de la simulation.

2.2.5 La méthode *get_mean_speed()*

Cette méthode calcule de façon récursive la vitesse moyenne des véhicules en circulation. Cette donnée est cruciale car elle permet de quantifier l'évolution d'un embouteillage et son intensité. En outre, elle permet de constater que certaines configurations sont plus propices à la génération d'embouteillages que d'autres. Par exemple, la concentration de camion dans une zone réduite semble réduire la vitesse moyenne et augmente le risque de formation d'un bouchon.

2.3 Le module Simulation

Ce module permet de lancer la simulation du mouvement de tous les véhicules sur la route en mode « console ». Initialement utilisé pour la simulation, ce module est moins utilisé mais permet d'effectuer des essais sans charger l'interface. Il contient une fonction qui est réutilisée dans cette dernière.

Cette fonction prend en argument la route avec les véhicules déjà placés et une incrémentation de temps. Elle appelle les méthodes *update_speed()* et *update_pos()* pour mettre à jour la vitesse des véhicules et leur position. Elle permet donc d'effectuer un tour de la simulation.

3 Interface Homme-Machine

L'IHM permet à un utilisateur de visualiser la route et de lancer la simulation, il peut choisir lui-même le nombre de véhicules sur la route, la longueur de la route, le nombre de voies et la vitesse de la simulation. Elle a été créée avec le logiciel Qt Designer et se lance en exécutant le fichier « main_ihm.py ».

3.1 Les différents actionneurs

3.1.1 Les actionneurs qui génèrent la route

Lorsque l'utilisateur ouvre la fenêtre de la simulation, il a le choix entre plusieurs actionneurs. Un premier *spinbox* permet de choisir le nombre de véhicules que l'on veut générer sur la route, la valeur de défaut est 40 véhicules, le minimum étant 1 véhicule et le maximum dépend de la taille de la route. Un deuxième permet donc de modifier la longueur de la route. Après avoir entré les différents paramètres, l'utilisateur peut générer la route en appuyant sur le bouton « Générer ».

3.1.2 Les actionneurs utiles pour la simulation

Après avoir générer la route et les véhicules, l'utilisateur peut lancer la simulation en appuyant sur le bouton « Simuler ». À tout moment de la simulation, il peut modifier la vitesse de celle-ci avec le *slider_vitesse*, il peut aussi changer l'affichage des voitures et camions en rectangles. Il y a aussi un bouton « Pause » permettant d'interrompre la simulation, et la reprendre en appuyant de nouveau sur « Simuler ».

Un bouton « Quitter » permet aussi de quitter la fenêtre de la simulation à n'importe quel moment. La vitesse moyenne des véhicules est affichée sur le « LCD_vitesse_moy ».

3.2 L'affichage

Les véhicules apparaissent sur une route à la forme circulaire qui restitue l'aspect « infini » de la simulation. L'utilisateur peut choisir entre deux modes d'affichage : le mode « Images » qui affiche des images représentant les véhicules en fonction de leur nature, et le mode « Rectangles » qui est plus léger et affiche uniquement la forme des véhicules. Plus le véhicule roule vite (par rapport à sa vitesse de consigne), plus sa couleur se rapproche du vert.

4 Figures imposées

Voici une brève description des figures imposées.

Factorisation du code

Le projet est composé de six modules, dont trois dédiés à la simulation, deux dédiés à l'affichage et à l'IHM, et un dédié aux tests unitaires. Hors interface, le projet est composé de six classes.

Documentation du code

Le code est commenté et documenté. La documentation est disponible en ouvrant le fichier « index.html » situé dans le dossier « docs/_build/html »

Tests unitaires

Un fichier « test.py » recense les différents tests unitaires effectués.

Création d'un type d'objet

Le projet comporte plusieurs types d'objets avec plusieurs variables d'instances, comme par exemple la classe abstraite *Vehicle* qui possède quatre variables d'instances (road, x, y) et une variable de classe (id).

Héritage entre deux types créés

Nous avons choisi d'utiliser deux types de véhicules dans notre simulation : les voitures et les camions. Ainsi les classes Car et *Truck* héritent de notre classe *Vehicule*.

Héritage depuis un type intégré

Notre classe *Road*, qui modélise la route, hérite d'un ndarray numpy. Si ce type n'est pas à proprement parlé *intégré*, il est très couramment utilisé et est semblable aux list python. Compte tenu de sa popularité, nous avons fait le choix de considérer que cet héritage répondait aux conditions imposées.

Fonction récursive

La méthode *get_mean_speed()* qui permet de calculer la moyenne de vitesse des véhicules, dans la classe *Road*, est une fonction récursive.

5 Limites et amélioration

Notre simulateur possède quelques défauts, essentiellement dus à un manque de temps. Parmi ceux-ci nous pouvons citer le manque d'une deuxième voie, voire d'une troisième voie sur la route simulée. Nous avons été contraints d'abandonner la réalisation de ce point car cela nécessitait d'utiliser un autre modèle physique, le modèle MOBIL, et nous demandait de revoir en profondeur la structure de notre code, ce qui était impossible compte tenu du temps restant.

Une version améliorée de notre simulateur pourrait également inclure la possibilité de modifier les variables comportementales du modèle, afin de pouvoir constater les comportements nuisibles et observer les bonnes pratiques en embouteillages.

6 Sources utilisées pour ce projet

Au cours de ce projet, nous nous sommes inspirés d'un certain nombre de simulateurs semblables au notre, sans toutefois se contenter d'une simple recopie. Nous nous sommes particulièrement inspirés du design d'un simulateur extrêmement complet utilisant JavaScript [1]. Pour l'élaboration et la mise en œuvre du modèle physique de l'IDM, nous nous sommes appuyés sur divers documents scientifiques dont une thèse et un mémoire [2][3]

[1] Simulateur de trafic routier dans différents scénarios : https://traffic-simulation.de/

[2] M. Mastio, « Modèles de distribution pour la simulation de trafic multi-agent », Theses, Université Paris-Est, 2017. [En ligne]. Disponible sur : https://hal.archives-ouvertes.fr/tel-01582943

[3] F. Savy, K. Trolès, « Simulation du trafic routier », Projet final au Cycle Préparatoire de Bordeaux, 2019, Disponible sur : http://fsavy.vvv.enseirb-matmeca.fr/reports/memoire_cpbx_savy_troles_trafic.pdf