



ÉCOLE POLYTECHNIQUE DE L'UNIVERSITÉ DE TOURS

64, Avenue Jean Portalis

37200 TOURS, FRANCE

Tél. (33)2-47-36-14-14

Fax (33)2-47-36-14-22

www.polytech.univ-tours.fr

Parcours des Écoles d'Ingénieurs Polytech

Rapport de projet S4

Modélisation de trafic et microsimulation

Auteur(s)

Samuel Bamba

[\[samuel.bamba@etu.univ-tours.fr\]](mailto:samuel.bamba@etu.univ-tours.fr)

Encadrant(s)

Emmanuel Néron

[\[emmanuel.neron@univ-tours.fr\]](mailto:emmanuel.neron@univ-tours.fr)

Polytech Tours
Département Informatique

Table des matières

Introduction	1
1 Présentation du IDM et partie mécanique	2
1.1 IDM	2
1.2 Partie mécanique	3
2 Le code et son évolution	5
2.1 1 ^{ere} version : Déplacement véhicule en ligne droite	5
2.2 2 nd version : Ajout d'un feu contrôlé par l'utilisateur	8
2.3 3 ^{eme} version : Ajout de curseurs modulables en temps réel	10
2.4 4 ^{eme} version : Véhicules multiples	11
2.5 5 ^{eme} version ? : Limites et idées d'amélioration du code	13
Conclusion	15
Annexes	16
A Liens utiles	17
B Fiche de suivi de projet PeiP	18

Introduction

Dans le cadre du projet de quatrième semestre nous avons eu pour objectif de développer une application simulant le déplacement de véhicules lors d'un trajet en ligne droite. Le langage que nous avons utilisé est python car il nous paraissait plus familier et adapté à ce projet. Nous avons plusieurs objectifs et avons donc amélioré progressivement notre code étapes par étapes, en se fixant à chaque fois un nouvel objectif. Nous avons ainsi développé 4 versions de la simulation, chaque version se basant sur la précédente tout en rajoutant une touche.

Dans ce compte rendu de projet nous parlerons dans un premier temps du modèle IDM¹ puis nous nous pencherons sur la partie mécanique qui est derrière les fonctions d'accélération et de décélération de notre code. Enfin, dans un second chapitre nous détaillerons et expliquerons le fonctionnement du code de chacune des 4 versions.

Une dernière partie sera consacré à une auto-critique du code.

1. Intelligent-Driver Model

Chapitre 1

Présentation du IDM et partie mécanique

1.1 IDM

Nous nous sommes inspiré du IDM (intelligent driver model) pour simuler des mouvements de véhicules lors d'un trafic routier.

L'IDM est une simulation de trafic à véhicules multiples. Développé en 2000 par D. Helbing, A. Hennecke, et M. Treiber dans le but d'améliorer les résultats obtenus par d'autres modèles existants, ce modèle vise un meilleur réalisme dans les simulations de longues durées. Le but de telles simulations est de déterminer les endroits de la route où les véhicules ont le plus de chances de se concentrer à un grand nombre, il est ainsi possible de déterminer une meilleure géométrie pour une route ou un carrefour ou encore la durée de feu rouge optimale pour les véhicules et les piétons.

Pour un véhicule α , x_α désigne sa position à un temps t , v_α sa vitesse et $\alpha - 1$ désigne le véhicule qui le précède. D'après l'IDM les dynamiques du véhicule α sont régies par les deux équations différentielles suivantes :

$$\begin{aligned}\dot{x}_\alpha &= \frac{dx_\alpha}{dt} = v_\alpha \\ \dot{v}_\alpha &= \frac{dv_\alpha}{dt} = a \left(1 - \left(\frac{v_\alpha}{v_0} \right)^\delta - \left(\frac{s^*(v_\alpha, \Delta v_\alpha)}{s_\alpha} \right)^2 \right) \\ \text{with } s^*(v_\alpha, \Delta v_\alpha) &= s_0 + v_\alpha T + \frac{v_\alpha \Delta v_\alpha}{2\sqrt{ab}} \\ s_\alpha &:= x_{\alpha-1} - x_\alpha - l_{\alpha-1} \\ \Delta v_\alpha &:= v_\alpha - v_{\alpha-1}\end{aligned}$$

FIGURE 1.1 – Equations différentielles qui régissent IDM

- v_0 : désigne la "vélocité voulue", soit la vitesse que le véhicule aurait si il n'y avait pas d'autres véhicules,
- s_0 : désigne la distance minimale autorisée entre le véhicule et celui devant lui,
- T : désigne l'intervalle de temps désiré entre le véhicule et celui devant lui,
- a : désigne l'accélération maximale du véhicule,
- b : désigne la décélération lors du freinage,
- δ : est une constante souvent égale à 4 et liée à a .

1.2 Partie mécanique

Pour notre simulation nous avons fait le choix de ne pas utiliser les mêmes équations différentielles que pour l'IDM. Par conséquent nous avons créé seulement nos fonctions `accelerate()` et `decelerate()`. Notre programme repose sur l'évaluation de la distance de freinage par rapport à la distance avant l'obstacle, il nous fallait donc dans un premier temps retrouver (sur internet) la formule de calcul d'une distance de freinage en fonction de la vitesse du véhicule. Le site <http://www.csgnetwork.com/stopdistcalc.html> propose une explication pour la détermination de la formule et un simulateur ; il nous a fourni la formule que nous avons utilisée :

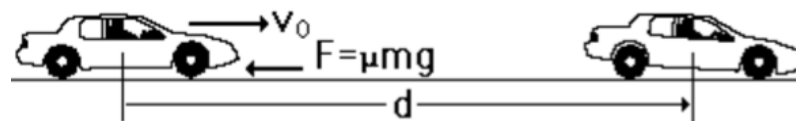
$$d = \frac{v_0^2}{2\mu g}$$


FIGURE 1.2 – Formule de calcul d'une distance de freinage

- v_0 : est la vitesse initiale
- μ : est le coefficient de friction entre les pneus et la route (0.8 pour un sol sec et des pneus en bon état)
- g : constante d'accélération de la pesanteur à la surface de la Terre (environ $9,81 \text{ m}\cdot\text{s}^{-2}$)

Nous avons donc pu déterminer un des tests fondamentaux de notre code à savoir le test de décélération/freinage :

```
58  #if distance au mur > distance de freinage + Distance de sécurité :
59  if (WIDTH-BALL_RADIUS-x)/10 > (((SPEED)**2)/(2*FRICTION*9.81))+BALL_RADIUS/10 and a == 0:
60      accelerate() # Accélérer
61  else:
62      decelerate()
```

Le véhicule va donc accélérer jusqu'à ce qu'il atteigne la distance de freinage, il va alors se mettre à décélérer. Expliquons à présent le fonctionnement des fonctions `accelerate()` et `decelerate()`.

Pour connaître l'accélération optimale par rapport à la taille de la fenêtre de l'application il nous fallait effectuer des tests. Le site http://www.engineeringtoolbox.com/car-acceleration-d_1309.html possède un simulateur qui après plusieurs tests nous a permis d'établir que nous voulions une accélération de 1.39 m/s^2 (ACCELERATION) pour notre fenêtre de $1000 \text{ px} = 100 \text{ m}$. - avec $a = 1.39 \text{ m/s}^2$ une voiture de taille moyenne parcourt $69,4 \text{ m}$ pour atteindre les 50 km/h -

La fonction `accelerate()` :

```
28 def accelerate():
29     global SPEED
30     if SPEED + ACCELERATION*0.02 <= SPEED_LIMIT/3.6 : # si le véhicule ne va pas dépasser la limite de vitesse
31         SPEED += ACCELERATION*0.02 # la boucle s'actualise toute les 20 ms d'où le *0.02
32     elif SPEED > SPEED_LIMIT/3.6: # si le véhicule a dépassé la vitesse limite il freine
33         SPEED -= ACCELERATION*0.02 # ralentir
```

On introduit la variable globale de la vitesse du véhicule `SPEED` qui est initialisée au choix de l'utilisateur mais souvent initialisée à 0. Le véhicule avance toutes les 20 ms par conséquent on augmente la `SPEED` du véhicule de `ACCELERATION * 0.02` toute les 20 ms.

- 1. **30, if** : Si on ne va pas augmenter la vitesse du véhicule au dessus de la limite de vitesse (on fait une division par 3.6 pour convertir les km/h en m/s) => Accélérer.
- 1. **32, elif** : Sinon si la vitesse du véhicule est déjà au dessus de la limite de vitesse => Ralentir.
- (**else**) : Sinon, on ne peut plus accélérer sans dépasser la limite de vitesse => Conserver la même vitesse.

La fonction `decelerate()` :

```
35 def decelerate():
36     global SPEED, DECELERATION, DFA
37     if DFA == 0: #Pour n'affecter a Deceleration une valeur une seule fois
38         DECELERATION = (SPEED*SPEED)/(2*((WIDTH-BALL_RADIUS-x)/10)) # a = v²/2x
39         DFA = 1
40         SPEED -= DECELERATION*0.02
41     else:
42         SPEED -= DECELERATION*0.02
```

La fonction `decelerate()` utilise un critère de test (changé en booléen dans les dernières versions du programme) : DFA (Distance de Freinage Atteinte), celui ci est égal à 0 (False) si la distance de freinage n'a pas encore été atteinte et 1 (True) sinon. C'est `decelerate()` qui le modifie et l'utilise, en effet on ne veut calculer la vitesse de décélération qu'une seule fois : une fois que la distance de freinage à été atteinte; puis utiliser la même DECELERATION durant tout le processus de décélération. Après pas mal de recherches (sur internet) nous avons pu trouver une formule qui nous donne la DECELERATION d'un véhicule :

$$DECELERATION = \frac{SPEED^2}{2 * d_a}$$

où d_a est la distance entre le véhicule et l'endroit où on veut qu'il s'arrête (le mur dans cette version, DAA¹ dans les suivantes). On soustrait la décélération de la même manière qu'on ajoute l'accélération.

- 1. **37, if** : Si on a pas encore affecté une valeur à DECELERATION => Affecter une valeur à DECELERATION et soustraire cette décélération à `SPEED` (On divise par 10 pour convertir les pixels en m).
- 1. **41, else** : sinon DFA = 1, c'est-à-dire que la valeur de DECELERATION à déjà été calculée => Soustraire cette décélération à `SPEED`.

1. Distance Avant Arrêt

Chapitre 2

Le code et son évolution

2.1 1^{ère} version : Déplacement véhicule en ligne droite

Après avoir appris comment fonctionnait la création d'objet en python il nous fallait à présent animer notre petit cercle bleu.

Le squelette de la première version repose en grande partie sur une page wikibook intitulée "Apprendre à programmer avec Python/Utilisation de fenêtres et de graphismes" (cf [Bibliographie](#)) Sur cette page on apprend à modifier les coordonnées d'un objet à l'aide d'une récursivité tout en gardant le contrôle sur l'animation à l'aide d'un "flag".

Le programme possède 3 fonctions : `start_it()` ayant pour action de démarrer l'animation, `stop_it()` qui la stoppe et `moveCircle()`. Cette dernière est la fonction principale, elle sera détaillé plus loin mais en attendant il est important de savoir que `moveCircle()` est récursive et se relance toutes les 20 ms grâce à cette ligne :

```
75 |         if flag > 0:
76 |             root.after(20,moveCircle) # répétition de la boucle si le critere d'arret est bon
```

La fonction `stop_it()` :

```
23 |     flag = 0                # critère d'arret
24 |     x = 0                   # position du véhicule
25 |     a = 0                   # distance de freinage atteinte ?
28 |     def stop_it():
29 |         "arret de l'animation"
30 |         global flag
31 |         flag = 0
```

L'explication dernière la variable "flag" est très simple, le "flag" est un critère d'arrêt qui a pour fonction de pauser l'animation lorsque sa valeur devient 0 (voir plus haut le `if flag > 0` : continuer la récursivité), et comme vous l'avez sûrement constaté, c'est `stop_it()` qui une fois enclenchée donne à "flag" la valeur 0.

La fonction `start_it()` :

```
33 |     def start_it():
34 |         global flag, x, END
35 |         "démarrage de l'animation"
36 |         if END == True:
37 |             x = 0
38 |             END = False
39 |         if flag == 0:        # pour ne lancer qu'une seule boucle
40 |             flag = 1
41 |             moveCircle()
```

La fonction `start_it()` prend en paramètres globaux le "flag", la position x du véhicule et un booléen critère d'arrêt de l'animation "END" qui est initialisé `False` et deviendra `True` lorsque le véhicule attendra le mur de fin.

- 1. 36, `if` : le critère "END" est testé; si le véhicule a atteint le mur (`END = True`), on reset la position du véhicule à 0, le véhicule n'est alors plus au mur et on modifie END en `False`.
- 1. 39, `if` : on vérifie que le "flag" est égal à 0 se qui veut soit dire que l'initialisation vient d'avoir lieu (premier lancement de l'animation) ou alors que la fonction `stop_it()` à été enclenchée (pause de l'animation) puis on le passe à 1 pour éviter des enclenchements multiples du Second. `if` de la fonction `start_it()`. Puis, lance le déplacement du véhicule avec `moveCircle()`.

La fonction `moveCircle()` :

```

59 def moveCircle():
60     global flag, x, END, DECELERATION, DFA, SPEED
61     if SPEED != 0:
62         # si le démarrage ne se fait pas à vitesse nulle
63         #if distance au mur > distance de freinage + Distance de sécurité :
64         if (WIDTH-BALL_RADIUS-x)/10 > (((SPEED)**2)/(2*FRICTION*9.81))+BALL_RADIUS/10 and a == 0:
65             accelerate() #accélérer
66         else:
67             decelerate() #ralentir
68     else:
69         print ("Speed 1 ",SPEED)
70         accelerate()
71
72     if x<WIDTH-BALL_RADIUS and SPEED > 0: # si la voiture n'a pas atteint la fin et n'est pas arrêtée
73         x+=SPEED/5
74     else:
75         END = True
76         SPEED = StartSPEED
77         DFA= 0
78         stop_it()
79     myCan.coords(i, x-BALL_RADIUS, HEIGHT/2-BALL_RADIUS, x+BALL_RADIUS, HEIGHT/2+BALL_RADIUS)
80     if flag >0:
81         root.after(20,moveCircle) # répétition de la boucle si le critere d'arret est bon

```

Nous avons déjà expliqué la partie 1. 61 à 1. 71, voyons ensemble la fin de cette fonction :

- 1. 72, `if` : Si la voiture n'a pas atteint le mur ($x < WIDTH - BALL_RADIUS$) et n'est pas à l'arrêt (`SPEED = 0`) => Ajouter la vitesse à la position du véhicule (on divise par 5 car l'ajout se fait 50 fois par seconde et `SPEED` est en dam/s).
- 1. 74, `else` : Sinon le véhicule a atteint le mur => Réinitialiser "END", `SPEED` et DFA et mettre le programme en pause (`stop_it()`).
- 1. 79 : Sert à modifier les coordonnées de l'objet véhicule.
- 1. 80, `if` : Si le programme n'est pas en pause ("flag" = 0) => `root.after(20, moveCircle)` <=> relancer `moveCircle` après une pause de 20 ms.

Parlons interface :

```

89 newBtn1 = Button(root,text="GO",command=start_it)
90 newBtn1.pack()
91
92 newBtn2 = Button(root, text='Arrêter', command=stop_it)
93 newBtn2.pack()
94
95 newBtn3 = Button(root, text='Quit', command=root.quit)
96 newBtn3.pack()

```

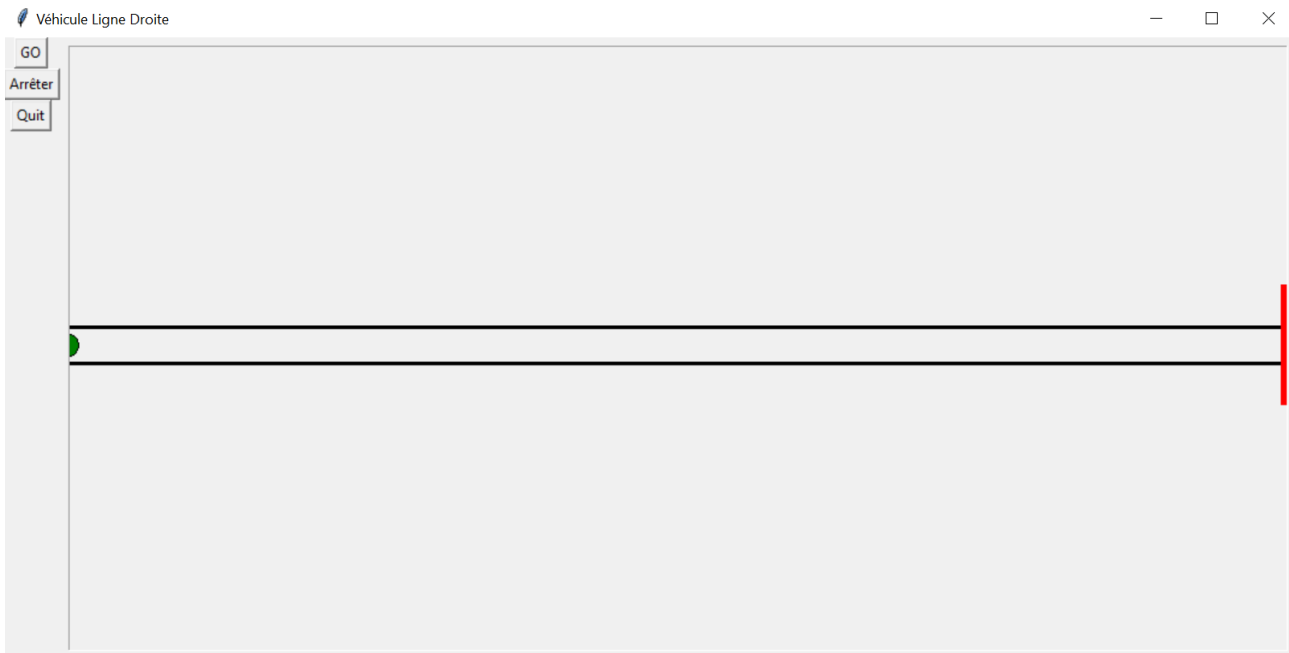



FIGURE 2.1 – Capture d'écran de l'interface de l'application avec les boutons

Il nous alors fallu un moyen de lancer les fonctions `start_it()` et `stop_it()` et qui de mieux placé que l'utilisateur pour une telle responsabilité. Par le moyen de communication machine à homme qu'est : un bouton ; nous avons pu associer `start_it()` à un bouton "GO", `stop_it()` à un bouton "Arrêter" et avons même rajouté un bouton "Quit" qui permet de quitter l'animation sans même cliquer la petite croix rouge.

2.2 2nd version : Ajout d'un feu contrôlé par l'utilisateur

Dans cette seconde version un feu est ajouté à l'interface :

```

93 def change_feu():
94     global FRouge, SPEED, END, flag
95     if FRouge == False:
96         myCan.itemconfigure(f, fill="red")
97         FRouge = True
98     else:
99         myCan.itemconfigure(f, fill="green")
100        FRouge = False
101        if END == False: # si le vehicule n'est pas arrêté au mur de fin
102            start_it()
123 f = myCan.create_line(WIDTH-DFM,HEIGHT-280,WIDTH-DFM,HEIGHT-220, width=5, fill="green")
133 newBtn3 = Button(root, text='Feu', command=change_feu)
134 newBtn3.pack()

```

Le feu est un widget ligne f, qui peut passer du vert au rouge et du rouge au vert à l'aide de la fonction `change_feu()` associée au bouton "Feu". On remarquera au passage qu'un critère booléen `FRouge` change de `False` à `True` en même temps que le feu passe du `vert` au `rouge`. Le deuxième `if` suivi du `start_it()` est présent pour faire redémarrer le véhicule instantanément après que le feu soit repassé au vert.

L'implantation de la Distance Avant Arrêt (DAA) multiple :

```

26 DFA = False # test: distance de freinage atteinte ?
27 FRouge = False # test: feu rouge ?
28 DAA = 0
36 def moveCircle():
37     global flag, x, END, DECELERATION, DFA, SPEED
38     if SPEED != 0: # si le demarrage ne se fait pas à vitesse nulle
45         """ (...) """
46         if FRouge == True and x < WIDTH-DFM: # si le feu est rouge et que le vehicule ne l'a pas dépassé
47             global DAA
48             DAA = (WIDTH-DFM - BALL_RADIUS-x)/10 # La distance avant arret est celle du vehicule au feu
49         else:
50             DAA = (WIDTH-BALL_RADIUS-x)/10 # La distance avant arret est celle du vehicule au mur
51
52         #if Distance Avant Arret > distance de freinage + Distance de sécurité:
53         if DAA > ((SPEED**2)/(2*FRICTION*9.81))+BALL_RADIUS/10 and DFA == False:
54             if SPEED + ACCELERATION*0.02 <= SPEED_LIMIT/3.6 : # si le vehicule ne va pas dépasser la limite de vitesse (km/h vers m/s: /3.6)
55                 SPEED += ACCELERATION*0.02 # la boucle s'actualise toute les 20 ms d'où le *0.02
56             elif DAA <= ((SPEED**2)/(2*FRICTION*9.81))+BALL_RADIUS/10: # sinon on freine
57                 if DFA == False: #Pour n'affecter a Deceleration une valeur une seule fois
58                     DECELERATION = (SPEED*SPEED)/(2*(DAA)) # a = v^2/2x
59                     print ("Decel ", DECELERATION)
60                     DFA = True
61                     SPEED -= DECELERATION*0.02
62                 else:
63                     SPEED -= DECELERATION*0.02
64             else: #si le feu repasse au vert
65                 if SPEED + ACCELERATION*0.02 <= SPEED_LIMIT/3.6 : # si le vehicule ne va pas dépasser la limite de vitesse (km/h vers m/s: /3.6)
66                     SPEED += ACCELERATION*0.02 # la boucle s'actualise toute les 20 ms d'où le *0.02
67                 """ (...) """

```

L'idée des multiples DAA repose sur le fait qu'on a voulu garder le même système de freinage que la V1. Que le véhicule s'arrête au mur de fin ou au feu, la seule chose qui change entre ces deux situations est la distance que le véhicule aura à parcourir avant de devoir s'arrêter.

- 1. 38, `if` : Si la vitesse n'est pas nulle (car il y aurait de la division par zéro dans les calculs qui vont suivre) on fait les tests sinon on accélère le véhicule.
- 1. 46, `if` : On décide de la valeur de la variable DAA, si `FRouge = True`, le feu est rouge, le véhicule doit donc s'y arrêter si il ne l'a pas encore dépassé (`x` : position véhicule < `WIDTH` : longueur de la route - Distance Feu-Mur) et si ces deux condition sont réunies DAA prend la valeur : $x_{feu} - \text{rayonVehicule} - x_{vehicule}$.

- 1. 49, **else** soit **FRouge** est **False** ou le véhicule a dépassé le feu et dans ce DAA prend la valeur : $x_{mur} - \text{rayonVéhicule} - x_{vehicule}$.

Après le choix de DAA on a le même test que avec la V1 avec la formule de calcul de distance de freinage et un critère test DFA (Distance Freinage Atteinte (?)). Ce qui est nouveau dans cette version c'est qu'on retest DAA dans le **elif** avec la formule de freinage de manière à vérifier qu'il n'a pas changé depuis la dernière boucle ; un changement de DAA indiquerait que le feu est repassé au vert, le **elif** devient faux et le **else** prend alors le relais en ré-accélérant (pour éviter que le véhicule ne continue sa décélération pour s'arrêter à un feu passé au vert).

2.3 3^{eme} version : Ajout de curseurs modulables en temps réel

De manière à se rapprocher d'avantage de l'IDM et de faciliter l'implantation des paramètres par l'utilisateur nous avons décider de rajouter des curseurs. Ceux-ci fonctionnent en deux parties :

La partie interface :

```

164 # Création d'un widget Scale
165 Acc = StringVar()
166 Acc.set(ACCELERATION)
167 echelle = Scale(root, from_=0, to=3, resolution=0.01, orient=HORIZONTAL, length=200, width=20, label="Acceleration (m.s²)", tickinterval=0.5, variable=Acc, command=majAcc)
168 echelle.pack(padx=1, pady=1)
169
170 SpeedL = StringVar()
171 SpeedL.set(SPEED_LIMIT)
172 echelle = Scale(root, from_=20, to=40, resolution=1, orient=HORIZONTAL, length=200, width=20, label="Limite de vitesse (km/h)", tickinterval=5, variable=SpeedL, command=majSpeedL)
173 echelle.pack(padx=1, pady=1)

```

Les deux paramètres qui ont été choisi pour ces curseurs sont l'accélération du véhicule (de base à $1,39m.s^2$) et limite de vitesse maximale (de base à 40 km/h). La partie interface fonctionne en parallèle avec une partie fonctions de mises-à-jour. La partie fonctions maj :

```

125 def majAcc(nouvelleValeur):
126     global ACCELERATION
127     # nouvelle valeur en argument
128     ACCELERATION = float(nouvelleValeur)
129     print("Acceleration ", ACCELERATION)
130
131 def majSpeedL(nouvelleValeur):
132     global SPEED_LIMIT
133     # nouvelle valeur en argument
134     SPEED_LIMIT = float(nouvelleValeur)
135     print("Vitesse Limit ", SPEED_LIMIT)

```

Les deux fonctions `majAcc()` et `majSpeedL()` permettent de mettre à jour les valeurs des variables globales `ACCELERATION` et `SPEED_LIMIT`, cela permet une mise à jour en temps réel qui peut s'observer directement sur l'animation.

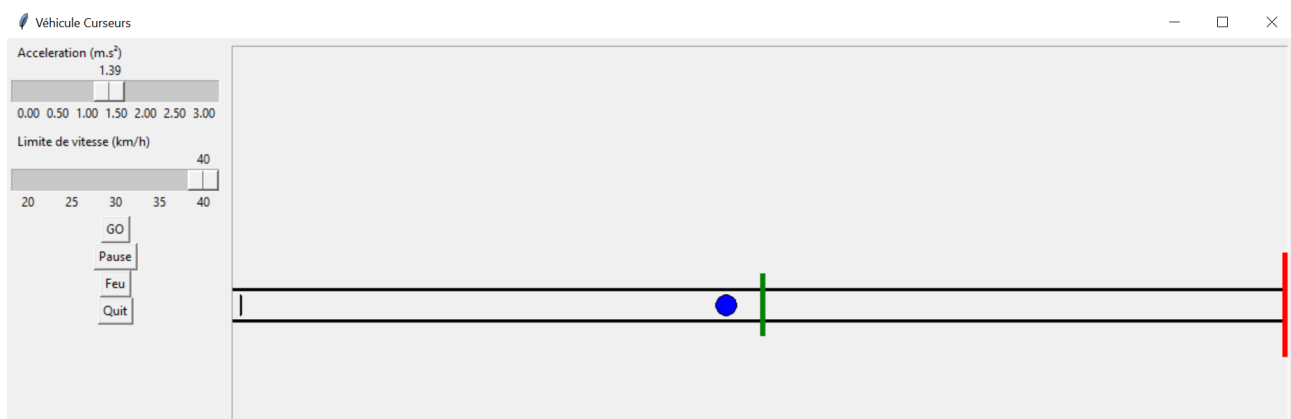


FIGURE 2.2 – Capture d'écran de l'interface de l'application avec les curseurs

2.4 4^{ème} version : Véhicules multiples

La quatrième et dernière version permet de simuler un trafic de plusieurs véhicules tout en conservant les fonctions des versions précédentes. Cette version a probablement été la plus dure à développer, dans un premier temps il a fallu redéfinir l'objet "véhicule" comme un objet a paramètres propres et individuels, puis ranger chaque véhicule dans une liste `i` :

```
187 i = list(range(BALLS))
188 for k in range(BALLS): # on crée une liste qui contient tous les véhicules
189     i[BALLS-1-k] = [myCan.create_oval(BALL_RADIUS, HEIGHT/2-BALL_RADIUS, BALL_RADIUS, HEIGHT/2+BALL_RADIUS, fill = Colors[k])
190                     ,0+k*(BALL_RADIUS*2+BALL_RADIUS/10),StartSPEED,False,False,WIDTH-BALL_RADIUS, 0]
191     # Véhicule i = (objet, position, speed, End: Move: Le véhicule a atteint le mur ?,
192     # DFA: Move: Le véhicule a atteint la distance de freinage ?,
193     # DAA: Distance avant que le véhicule doive s'arrêter, Vitesse de decélération)
```

`i` est une liste de longueur BALLS (où BALLS est le nombre de véhicules qu'on a choisi d'animer), ainsi `i[0]` désigne le véhicule en tête de file qui est lui aussi une liste de paramètres :

`i[j] = véhicule = [(0.), (1.), (2.), (3.), (4.), (5.), (6.)]`

1. `i[j][0]` [(ovale, de rayon BALL_RADIUS, de couleur Colors[j]),
2. `i[j][1]` = (position x), : pour les positions de départ on a le BALLS - 1^{ème} placé en $x = 0$, le $x_{BALLS-2ème} = 2 * BALL_RADIUS + BALL_RADIUS/10 \leq \Rightarrow$ distance entre deux véhicules + distance de sécurité,..., le $x_{0ème} = BALLS - 1 * (BALL_RADIUS * 2 + BALL_RADIUS/10)$,
3. `i[j][2]` = (vitesse), : initialisée à 0,
4. `i[j][3]` = (END), : chaque véhicule a son propre critère booléen pour savoir si il a atteint le mur, initialisé à **False**,
5. `i[j][4]` = (DFA), : pareil pour le critère de distance de freinage atteinte, il est propre à chaque véhicule et est initialisé à **False**,
6. `i[j][5]` = (DAA), : la distance avant arrêt est un critère important qui est initialisé à WIDTH - BALL_RADIUS $\leq \Rightarrow$ distance avant le mur en partant de 0,
7. `i[j][6]` = (Vitesse de décélération), : et enfin puisque les véhicules ne vont pas faire face aux mêmes obstacles et ne vont pas forcément décélérer ni en même temps, ni à la même vitesse, initialisé à 0.

Dans le cas de véhicules multiples la récurrence fonctionne en faisant une mise à jour des paramètres des différents véhicules, pour permettre cela nous avons créé la fonction `changeCircle()` :

```
104 def changeCircle():
105     global j, i, TabMove
106     if j == BALLS-1: # Si on était en train de faire bouger le dernier véhicule
107         j = 0 # On retourne au premier
108     else: j += 1 # Sinon on passe au suivant
```

Après la pression de "GO", la fonction `moveCircle()` s'exécute une fois et met à jour les paramètres du premier véhicule, puis contrairement aux versions précédentes `changeCircle()` est lancé avant de répéter `moveCircle()`. `changeCircle()` est une fonction très simple, elle fait défiler `j`; si on est à l'indice `j` maximal = BALLS - 1 on retourne à `j = 0` sinon on fait `+1`. Une fois que

le prochain indice à été déterminé `moveCircle()` se répète comme toute bonne fonction récursive mais cette fois ci on mettra à jour les coordonnées du nouveau véhicule.

Pour éviter que les véhicules ne se rentrent dedans ou passent les uns à travers les autres nous avons rajouté une option supplémentaire à la séquence de `tests` pour déminer la Distance Avant Arrêt (`i[j][5]`) du véhicule. Cette séquence de `tests` possède dorénavant trois possibilités d'affectation de DAA (`i[j][5]`) :

- ① $i[j][5] = (\text{WIDTH-DFM} - \text{BALL_RADIUS} - i[j][1])/10 \Leftrightarrow$ distance entre le véhicule et le feu.
- ② $i[j][5] = (\text{WIDTH-BALL_RADIUS} - i[j][1])/10 \Leftrightarrow$ distance entre le véhicule et le mur.
- ③ $i[j][5] = (i[j-1][1] - i[j][1] - \text{BALL_RADIUS}*2)/10 \Leftrightarrow$ distance entre le véhicule et celui de devant.

Etudions la séquence de `tests` :

```

36 def moveCircle():
37     global flag, j, i, BALLS
38
39     if i[j][3] == False :
40         if FRouge == True and i[j][1] < WIDTH-DFM : # si le feu est rouge et que le vehicule ne l'a pas dépassé
41             if i[j-1][1] > WIDTH-DFM or j == 0 :
42                 i[j][5] = (WIDTH-DFM - BALL_RADIUS-i[j][1])/10 # La distance avant arret est celle du vehicule
43             else: # si le vehicule n'est pas premier
44                 i[j][5] = (i[j-1][1] - i[j][1] - BALL_RADIUS*2)/10 # La distance avant arret est celle du vehicule
45         elif j != 0: # si le vehicule n'est pas premier
46             i[j][5] = (i[j-1][1] - i[j][1] - BALL_RADIUS*2)/10 # La distance avant arret est celle du vehicule
47         else:
48             i[j][5] = (WIDTH-BALL_RADIUS-i[j][1])/10 # La distance avant arret est celle du vehicule au mur

```

- 1. 39, `if` : On teste "END" (`i[j][3]`), si le véhicule a déjà atteint le mur on ignore toute la séquence de déplacement de `moveCircle()` et on passe au prochain `j` avec `changeCircle()` :
- 1. 40, `if` : Si le feu est rouge et (`i[j][1]`) < WIDTH-DFM \Leftrightarrow le véhicule n'a pas dépassé le feu :
- 1. 41, `if` : Si (`i[j][1]`) > WIDTH-DFM \Leftrightarrow le véhicule de devant a dépassé le feu ou `j == 0` \Leftrightarrow le véhicule est le premier \Rightarrow ①
- 1. 43, `else` : Sinon le véhicule a un véhicule devant lui \Rightarrow ③
- 1. 45, `elif` : Sinon si le véhicule n'est pas premier \Rightarrow ③
- 1. 47, `else` : Sinon le véhicule est premier et n'a pas d'obstacles devant lui \Rightarrow ②

2.5 5^{ème} version ? : Limites et idées d'amélioration du code

La 4^{ème} version pourrait donner naissance à une centaine d'autres. Par manque de compétence mais surtout de temps nous ne pouvons que lister des idées d'amélioration de notre code qu'il sera possible d'établir plus tard.

Les problèmes de la V4 :

- Le lag : au bout de 5 véhicules l'aspect récurssif de la fonction `moveCircle()` engendre un lag, les mise-à-jour des positions ne se font pas assez rapidement et on voit les véhicules se déplacer par à-coups. Ce n'est pas tant l'aspect "laid" du lag qui est dérangentant mais le fait qu'il altère le réalisme de la simulation, en effet plus l'animation lague plus les véhicules sont lents, par conséquent sa vitesse et son accélération ne correspondent plus aux chiffres rentrés par l'utilisateur.

Pour régler ce problème j'ai tenté de créer une variable `FRAMES_PER_SECOND` qui serait contrôlé par l'utilisateur à l'aide d'un curseur :

```
153 def majFPS(nouvelleValeur):
154     global FRAMES_PER_SEC
155     # nouvelle valeur en argument
156     if flag == 0: # Pour ne pas changer les FPS pendant l'animation
157         FRAMES_PER_SEC = float(nouvelleValeur)
158     print ("FPS ", FRAMES_PER_SEC)
202 Fps = StringVar()
203 Fps.set(FRAMES_PER_SEC)
204 echelle = Scale(root,from=40,to=100,resolution=1,orient=HORIZONTAL,length=200,width=20,
205                 label="Frame Per Second",tickinterval=10,variable=Fps,command=majFPS)
206 echelle.pack(padx=1,pady=1)
```

Mais encore la il y a le problème que plus on augmente les fps plus les véhicules se déplacent rapidement et se n'est pas logique. Pour garder le réalisme de la simulation il faudrait déterminer une variable **Y** qui dépendrait de `FRAMES_PER_SECOND` et la multiplier par `ACCELERATION` pour que plus il y ait de boucles par seconde moins les véhicule avancent à chacune de ces boucles.

- Nombre de véhicules : Le nombre de véhicules est limité au nombre de véhicule qu'on peut mettre avant le feu. Deux solutions pour remédier à se problème :
 - Avoir un écran plus grand ;
 - Modifier le point de départ des véhicules.

Dans la version actuelle lors de l'initialisation et la création des objets véhicules On place chaque véhicule à une distance $BALL_RADIUS * 2 + BALL_RADIUS/10$ du précédent :

```
182 i = list(range(BALLS))
183 for k in range(BALLS): # on crée une liste qui contient tous les véhicules
184     i[BALLS-1 - k] = [myCan.create_oval(BALL_RADIUS, HEIGHT/2-BALL_RADIUS, BALL_RADIUS, HEIGHT/2+BALL_RADIUS
185     ,0+k*(BALL_RADIUS*2+BALL_RADIUS/10),StartSPEED,False,False,WIDTH-BALL_RADIUS, 0)]
186 # Véhicule i = (objet, Position, speed, End: Move: Le véhicule a atteint le mur ?,
```

Il faudrait faire démarrer les véhicules superposés les uns sur les autres puis lorsque les véhicules 1 et 2 ne sont plus confondus le 2 démarre, puis le 3 et ainsi de suite de façon à pouvoir théoriquement une infinité de véhicules. Pour en avoir une infinité il faudrait que les

véhicules se superposent aussi à l'arrivée ou alors qu'ils sortent de l'écran... ce qui nous amène à la seconde partie de cette section 2.5 : les idées d'amélioration.

- Voix multiples : Rajouter des voix n'est pas tellement un problème mais gérer les dépassement entre véhicules pourrait être très difficile.
- Route circulaire : Une route circulaire permettrait à l'animation de pouvoir tourner en continue, de plus on pourrait observer des effets de circulation intéressants comme "l'effet accordéon" lors d'un embouteillage (cf [Liens utiles](#)). Une autre alternative à la route circulaire serait de téléporter les véhicules au point de départ lorsqu'ils arrivent au mur, le but étant de donner une illusion de fluidité.
- Identité des véhicules : Malgré que chaque véhicule ait sa propre vitesse, couleur ou décélération cela n'est pas encore suffisant pour leur permettre d'avoir une identité propre, ressemblante à celle de vrai conducteurs. Plusieurs paramètres pourrait être rajoutés dans ce but :
 - différents modèles de véhicules avec une accélération, une masse, et donc une capacité à freiner différente.
 - un type de conduite altérant l'accélération, les distances de sécurité et la volonté à doubler dans le cas d'une simulation à plusieurs voix
 - ...

Conclusion

Dans le cadre du projet de quatrième semestre j'avais pour objectif de développer une application simulant le déplacement de véhicules lors d'un trajet en ligne droite et voilà chose faite. La création de cette simulation fut une parfaite expérience en territoire inconnu à la fois enrichissante et difficile parfois. Il est intéressant de voir le fruit de plusieurs heures de travail se concrétiser sur un écran et malgré que la simulation soit loin d'être parfaite et clairement ~~laide~~ simpliste, ce travail reste avant tout une fierté méritant sa place sur un CV. Cette simulation mérite d'être développée d'avantage pour trouver une réelle utilité.

Bibliographie

- [1] https://fr.wikibooks.org/wiki/Apprendre_%C3%A0_programmer_avec_Python/Utilisation_de_fen%C3%AAtres_et_de_graphismes#Animation_automatique_-_R.C3.A9cursivit.C3.A9
- [2] <http://www.traffic-simulation.de/>
- [3] <http://www.csgnetwork.com/stopdistcalc.html>
- [4] http://www.engineeringtoolbox.com/car-acceleration-d_1309.html
- [5] https://en.wikipedia.org/wiki/Intelligent_driver_model
- [6] <http://rsta.royalsocietypublishing.org/content/368/1928/4585.short>

Annexe A

Liens utiles

Voici une petite liste d'url intéressantes au sujet de ce projet :

— [https://fr.wikipedia.org/wiki/Embouteillage_\(route\)#Ph.C3.A9nom.C3.A8ne](https://fr.wikipedia.org/wiki/Embouteillage_(route)#Ph.C3.A9nom.C3.A8ne)

Annexe B

Fiche de suivi de projet PeiP

Séance n° 1 du 13/01/2016	prise en compte du sujet, établissement des objectif avec le professeur encadrant
Séance n° 2 du 20/01/2016	prise en main des widgets python et première lignes de code
Séance n° 3 du 27/01/2016	tests et recherches concernant la mécanique
Séance n° 4 du 04/02/2016	détermination de formules et implantation dans le code
Séance n° 5 du 11/02/2016	dernières retouche à l'interface de la V1, réflexion sur la V2
Séance n° 6 du 18/02/2016	décision sur l'implantation d'un feu dans la V2 et programmation de l'interface visuelle du feu
Séance n° 7 du 25/02/2016	rajout des critères d'arrêt pour le feu
Séance n° 8 du 02/03/2016	fnitions sur la V2
Séance n° 9 du 09/03/2016	développement de l'idée de la V3 et étude du fonctionnement des curseurs
Séance n° 10 du 16/03/2016	implantation des curseurs à la V3 et tests avec différents paramètres, 2 sont retenus
Séance n° 11 du 06/04/2016	développement de la V4
Séance n° 12 du 13/04/2016	fnitions et lissage du code

Modélisation de trafic et microsimulation

Rapport de projet S4

Résumé : Compte rendu de projet sur le développement d'une simulation de trafic en python inspiré de l'IDM

Mots clé : informatique, simulation, IDM

Abstract : Project record about the developpement of a simulation of cars in python inspired by IDM

Keywords : informatic, simulation, IDM

Auteur(s)

Samuel Bamba

[samuel.bamba@etu.univ-tours.fr]

Encadrant(s)

Emmanuel Néron

[emmanuel.neron@univ-tours.fr]

**Polytech Tours
Département Informatique**

Ce document a été formaté selon le format EPUProjetPeiP.cls (N. Monmarché)

École Polytechnique de l'Université de Tours
64 Avenue Jean Portalis, 37200 Tours, France
<http://www.polytech.univ-tours.fr>