



PAr 126 : Modélisation et visualisation des équilibres de transport d'un réseau routier



Auteurs :

Clément BORNE
Fabien DURANSON

Table des matières

1 Extraction des données XML	4
1.1 Format des données	4
1.2 Méthodologie générale	4
2 Visualisations statiques	6
2.1 Cartes Origine-Destination	6
2.1.1 Extraction et création de la matrice OD	6
2.1.2 Trajets en lignes droites	7
2.1.3 Cercles	7
2.1.4 Cartes internes	8
2.2 Visualisation des trajets	9
2.2.1 Vitesse limite	9
2.2.2 Encombrement des tronçons	10
2.3 MotionRugs	10
2.3.1 De la 2D à la 1D	11
2.3.2 Visualisation et interprétation	12
3 Visualisations dynamiques	13
3.1 Replay	13
3.1.1 Avantages	14
3.1.2 Limites	14
3.2 Densité de véhicules par gradient	14
4 Bibliographie	16
5 Annexes : Exemples de Codes	17
5.1 Replays	17
5.2 Cercles	18

Résumé

Ce document traite du projet de recherche commandité par les chercheurs Romain VUILLEMOT et Ludovic LECLELRC. Il concerne la visualisation de données de simulation de trafic. Ce projet a été réalisé entre octobre 2018 et avril 2019. Il a abouti sur 12 codes Python permettant de créer des visualisations à partir des données de simulation fournies par un simulateur de l'ENTPE.

Ce rapport explique le fonctionnement des codes Python et fournit une interprétation aux visualisations obtenues. Ces dernières sont séparées en 2 catégories : les visualisations statiques et les visualisations dynamiques. Les visualisations statiques sont des images obtenues rapidement et les visualisations dynamiques sont des vidéos et prennent donc plus de temps à être créées.

Tous les codes et résultats sont disponibles sur GitHub.

Dans ce rapport se trouvent les descriptions des codes permettant d'obtenir les visualisations suivantes :

- Carte OD sous forme de cercles
- Carte OD sous forme de lignes droites
- Carte OD sous forme de cartes imbriquées
- MotionRugs
- Encombrement des tronçons sur la totalité de la simulation
- Vidéo rejouant la simulation
- Visualisation dynamique sous forme de zones de chaleur

Abstract

This document is about a project backened by Romain VUILLEMOT and Ludovic LECLERC. The topic is the visualisation of traffic equilibrium. The project has been carried out between october 2018 and april 2019. It results on 12 Python programs enabling one to create visualizations from data given by the ENTPE's simulator.

This report explains how the Python programs work and gives an interpretation to the visualizations. These are separated in two categories : static and dynamic visualizations. Static visualizations are images and dynamic visualizations are videos that require more calculation.

All codes and results are available on GitHub.

In this document one can find descriptions of the programs that have created the following visualisations :

- OD Map with circles
- OD Map with straight lines
- OD Map with nested maps
- MotionRugs
- Congestion of sections over the whole simulation
- Replay of the simulation
- Dynamic visualisation with heat areas

Glossaire

Visualisation

Domaine informatique pluri-disciplinaire dont l'objet d'étude est la représentation visuelle de données sur une interface graphique.

Réseau routier

Ensemble des voies de circulation terrestres permettant le transport par véhicules routiers. Dans le cadre du projet, le réseau est l'ensemble des voies de circulations du 6^{me} arrondissement de Lyon.

O-D

Sigle employé dans ce rapport comme abréviation de "Origine-Destination". Ce sigle renvoie à une forme d'information présentant uniquement l'origine et la destination des utilisateurs d'un **Réseau Routier**, en faisant abstraction du trajet parcouru.

Trafic routier

Déplacement de véhicules automobiles sur une route. Dans ce document, on désignera par trafic la situation globale des positions des véhicules sur le réseau.

Tronçon

Portion de route délimitée de part et d'autre par une intersection (rond point, carrefour,...) ou par une sortie du réseau (bordure du réseau ou impasse). Un tronçon est défini par une **position de départ**, une **position de fin**, et un certain nombre de **voies**.

Congestion

État d'un lieu du réseau fortement encombré.

Trajet

Ensemble des tronçons parcourus par son véhicule, et par abus de langage, le trajet d'un utilisateur pourra aussi être la trajectoire de son véhicule.

Code

Texte qui représente les instructions d'un programme telles qu'elles ont été écrites dans un langage de programmation sous une forme humainement lisible par un programmeur. Les codes sont ici les fichiers en format ".py" créés à partir de *Spyder* et exécutés par Python 3.0. La distinction entre **programme** et **code** n'étant pas pertinente à l'échelle de notre projet, nous utiliserons indifféremment l'un et l'autre pour désigner le code informatique.

Algorithm

Ensemble de règles opératoires dont l'application permet de résoudre un problème énoncé au moyen d'un nombre fini d'opérations. Un algorithme peut être transcrit dans un **code** afin d'être exécuté par le langage de programmation.

XML

Acronyme de eXtensible Markup Language (langage de balisage extensible). C'est un langage informatique servant à enregistrer des données textuelles. C'est un langage très répandu ayant l'avantage d'être très lisible humainement grâce à son système de balisage.

Arbre/Noeuds/Fils/Père

En informatique, un **Arbre** est une structure de données qui peut se représenter sous la forme d'une hiérarchie dont chaque élément est appelé **Noeud**. Dans un arbre, chaque **Noeud** possède un ou des **Fils**, dont il est le **Père**.

Introduction

La visualisation des données permet de représenter sous forme graphique des résultats scientifiques. C'est donc via cette dernière que ce fait l'interface entre l'homme et la machine. Elle permet via la couleur, les formes ou encore le mouvement de faire passer des informations qui retrouvent au plus juste la réalité. On ne peut généralement pas visualiser la totalité des éléments nécessaires sur un seul et même graphe, il faut donc réfléchir à plusieurs techniques, qui une fois combinées permettront d'avoir un point de vue global.

Contexte

Dans cette optique, la visualisation de réseaux routiers est très intéressante puisqu'elle permet d'accéder rapidement à de nombreuses données et de trouver des solutions à un problème fixé. On peut par exemple s'intéresser aux routes les plus empruntées et donc anticiper leur usure. On peut également observer les tronçons les plus congestionnés, afin de réfléchir à une solution, trouver la source du problème. On peut aussi s'intéresser aux trajets les plus fréquemment empruntés, etc

Dans le cadre de notre PAr, on s'intéresse à la visualisation de données issues d'un simulateur de l'Ecole Nationale des Travaux Publics de l'Etat. Ce dernier simule l'évolution du réseau routier de Lyon 6 et de Villeurbanne. On ne cherche pas à expliquer le fonctionnement de la simulation, mais seulement à visualiser les données brutes afin qu'elles puissent être exploitées par les chercheurs de l'ENTPE.

1 Extraction des données XML

On s'intéresse dans cette partie à la forme et au traitement des données issues de la simulation.

1.1 Format des données

Les données se présentent sous la forme d'un arbre, on s'intéressera au noeud <INSTANTS> qui a pour fils les différents instants composant la simulation. Chaque instant a proprement parler à lui même plusieurs fils :

- Les entrées de véhicules sur le réseau
- Les sorties de véhicules du réseau
- Les trajectoires des véhicules sur le réseau (position, vitesse, distance parcourue)

```
<INSTANTS>
  <INST nbVeh="0" val="1.00">=
  <INST nbVeh="0" val="2.00">=
  <INST nbVeh="0" val="3.00">=
  <INST nbVeh="0" val="4.00">=
  <INST nbVeh="2" val="5.00">=
  <INST nbVeh="2" val="6.00">=
  <INST nbVeh="7" val="7.00">=
  <INST nbVeh="8" val="8.00">=
  <INST nbVeh="9" val="9.00">=
  <INST nbVeh="9" val="10.00">
  <CREATIONS>
    <CREATION entree="E_82607284" id="9" sortie="S_545413229_T_61615192_FRef" type="VL"/>
    <CREATION entree="E_762997770_T_549690255_toRef" id="10" sortie="S_762997770_T_549690255_FRef" type="VL"/>
  </CREATIONS>
  <SORTIES/>
  <TRAJS>
    <TRAJ abs="843092.51" acc="0.00" dst="17.29" id="0" ord="6519881.85" tron="T_799708557_FRef" type="VL" vit="18.00" voie="1" z="0.00"/>
    <TRAJ abs="842986.09" acc="0.00" dst="48.36" id="1" ord="6520859.28" tron="T_549694276_toRef" type="VL" vit="18.00" voie="2" z="0.00"/>
    <TRAJ abs="844386.83" acc="0.50" dst="27.43" id="2" ord="6520635.52" tron="T_58230558_FRef" type="VL" vit="14.50" voie="1" z="0.00"/>
    <TRAJ abs="843398.64" acc="-1.00" dst="75.00" id="3" ord="6519869.24" tron="T_58232701_FRef" type="VL" vit="13.00" voie="1" z="0.00"/>
    <TRAJ abs="844421.66" acc="0.00" dst="90.00" id="4" ord="6520733.78" tron="T_825496565_FRef" type="VL" vit="18.00" voie="1" z="0.00"/>
    <TRAJ abs="844391.88" acc="-1.00" dst="71.12" id="5" ord="6519879.10" tron="T_818614510_FRef" type="VL" vit="16.00" voie="1" z="0.00"/>
    <TRAJ abs="844533.48" acc="-1.00" dst="19.45" id="6" ord="6520215.31" tron="T_1089867061_FRef" type="VL" vit="13.00" voie="1" z="0.00"/>
    <TRAJ abs="843037.78" acc="0.00" dst="70.91" id="7" ord="6519883.07" tron="T_58258527_FRef" type="VL" vit="18.00" voie="2" z="0.00"/>
    <TRAJ abs="842919.10" acc="0.00" dst="46.67" id="8" ord="6520861.37" tron="T_61615192_toRef" type="VL" vit="18.00" voie="1" z="0.00"/>
    <TRAJ abs="842983.28" acc="0.00" dst="16.36" id="9" ord="6519880.91" tron="T_58258527_FRef" type="VL" vit="18.00" voie="2" z="0.00"/>
    <TRAJ abs="844325.38" acc="0.00" dst="16.36" id="10" ord="6520941.29" tron="T_549690255_toRef" type="VL" vit="18.00" voie="2" z="0.00"/>
  </TRAJS>
```

FIGURE 1: Format des données XML

1.2 Méthodologie générale

Traiter des données de ce volume demande une bonne méthodologie pour réduire le temps de calcul. C'est pourquoi nous avons procédé comme indiqué en *Figure 2*.



FIGURE 2: Méthodologie générale de traitement de données

Nous avons décomposé la création des visualisations en deux étapes :

- Extraction des données intéressantes du fichier XML et stockage sous une forme qui sera exploitable efficacement par un algorithme informatique.
- Récupération des données importantes à partir de cette forme de stockage afin de l'exploiter efficacement et de créer la visualisation recherchée.

La méthode générale d'extraction de données XML en Python nous a été fournie par Ruiwei CHEN, notre contact spécialiste à l'ENTPE. Il s'agit d'une librairie qui facilite la recherche de clés en transformant le fichier XML en arbre.

Ces méthodes restent cependant longues en raison de la taille des données : la création de l'arbre et l'extraction de données peuvent prendre jusqu'à 10 minutes.

C'est pour cette raison que nous avons décidé de stocker les données extraites du fichier XML dans un fichier texte. La lecture de fichier texte est en effet plus rapide, car ce dernier est créé de manière à ne contenir que les données utiles à la visualisation.

2 Visualisations statiques

Une visualisation statique est une image, ou un tableau. Elle ne varie pas dans le temps, mais permet de représenter l'évolution d'un système entre les instants t_1 et t_2 . Il s'agit d'un bilan de ce qu'il s'est passé au cours de la simulation.

2.1 Cartes Origine-Destination

Une matrice Origine-Destination (OD) est un tableau qui, pour chaque véhicule présent sur le réseau, donne son point d'entrée et son point de sortie (ces points peuvent être internes au réseau). Dans toute cette partie on utilise uniquement les instants t_0 et t_f de la modélisation : on ne s'intéresse pas au trajet emprunté par les utilisateurs.

2.1.1 Extraction et création de la matrice OD

A la fin des données de la simulation se trouve un résumé du trajet de chaque véhicule. Ce dernier permet de récupérer le tronçon d'entrée et le tronçon d'arrivée de chaque véhicule. On a donc également besoin des informations sur les tronçons afin d'avoir accès à des coordonnées spatiales. La fonction représentée en *Figure 3* permet l'extraction des résumés des trajets des véhicules et des tronçons existants afin de réunir les informations pour avoir les coordonnées des origines et des destinations.

```
15
16 def ODMatFromXML(path):
17     input_xml = path
18     input_tree = ET.parse(input_xml)
19     input_root = input_tree.getroot()
20
21     list_veh = input_tree.findall("./VEHS/VEH")
22     nbr_veh = len(list_veh) # total number of links
23     list_troncons = input_tree.findall("./RESEAU/TRONCONS/TRONCON")
24
25     dlist = []
26     aList = []
27
28     for i in range(nbr_veh):
29         veh = list_veh[i].attrib
30         if 'itineraire' in veh.keys():
31             itineraire = veh['itineraire'].split()
32             troncon_depart = itineraire[0]
33             troncon_arrivee = itineraire[-1]
34             depart_str = find_origin(troncon_depart, list_troncons).split()
35             arrivee_str = find_end(troncon_arrivee, list_troncons).split()
36             depart = (float(depart_str[0]), float(depart_str[1]))
37             arrivee = (float(arrivee_str[0]), float(arrivee_str[1]))
38             dlist.append(depart)
39             aList.append(arrivee)
40
41     D = np.eye(len(dList))
42
43     departs = []
44     arrives = []
45
46     for d in dList:
47         if d not in departs:
48             departs.append(d)
49     for a in aList:
50         if a not in arrives:
51             arrives.append(a)
52
53     ODMat = [[0 for _ in range(len(arrives))] for _ in range(len(departs))]
54
55     for depart in departs:
56         for i in range(len(dList)):
57             if dList[i]==depart:
58                 a = find_index(aList[i],arrives)
59                 d = find_index(dList[i],departs)
60                 ODMat[d][a]+=1
61
62     return {'mat' : ODMat, 'origins' : departs, 'destinations' : arrives}
```

FIGURE 3: Code de ODMatFromXML

Cette fonction renvoie le dictionnaire contenant la matrice et les coordonnées géographiques correspondantes à toutes les lignes et colonnes ('origins' et 'destinations').

Cette fonction est coûteuse en temps (plus d'une minute). Les fonctions représentées en *Figure 4* permettent de créer un fichier texte afin d'avoir un accès immédiat aux données, sans repasser par l'arbre.

```

64
65 def save(ODmat,name):
66     ODMat = ODmat['mat']
67     origins = ODmat['origins']
68     destinations = ODmat['destinations']
69     file = open(name,"w")
70     for l in ODMat:
71         s=""
72         for e in l:
73             s+=str(e)
74             s+=" "
75             s+="\n"
76             file.write(s)
77             file.write("\n")
78         for o in origins:
79             s = str(o[0]) + " " + str(o[1]) + "\n"
80             file.write(s)
81             file.write("\n")
82         for d in destinations:
83             s = str(d[0]) + " " + str(d[1]) + "\n"
84             file.write(s)
85             file.write("\n")
86     file.close()
87
88 def ODMatFromTxt(name):
89     file = open(name)
90     f = file.readlines()
91     ODM_s,origins_s,destinations_s = split_l(f)
92     ODMat = []
93     origins = []
94     destinations = []
95     for l in ODM_s:
96         lp = []
97         lf = l.split()
98         lf.pop(-1)
99         for e in lf:
100            lp.append(int(e))
101        ODMat.append(lp)
102    for o in origins_s:
103        origin = o.split()
104        origins.append((float(origin[0]),float(origin[1])))
105    for d in destinations_s:
106        dest = d.split()
107        destinations.append((float(dest[0]),float(dest[1])))
108    return {'mat' : ODMat, 'origins' : origins, 'destinations' : destinations}
109
110

```

FIGURE 4: Codes de ODMatFromTxt et save

La première sauvegarde les données créées grâce à l'extraction du document XML dans un fichier texte. On contrôle alors complètement la manière de lire les données présentes dans le fichier texte. La lecture est assurée par la fonction *ODMatFromTxt*. Elle reprend la structure de la fonction *save*, et permet d'obtenir le même résultat que la fonction *ODMatFromXML* mais en un temps plus court. La création du fichier texte est longue, mais les données étant figées, une fois que ce dernier

2.1.2 Trajets en lignes droites

La première visualisation OD réalisée (*Figure 5*) représente les trajets des véhicules et permet de se faire sous forme de lignes droites une idée des couples Origine-Destination les plus souvent appariés.

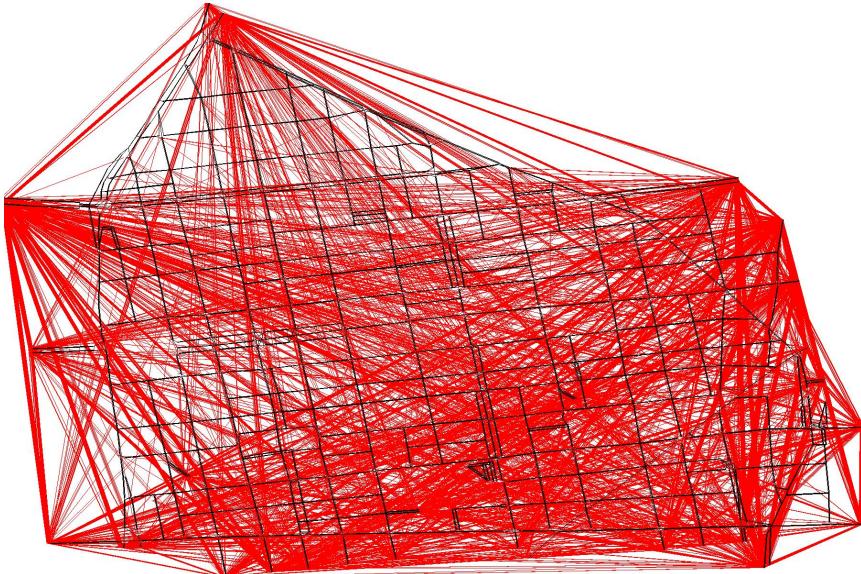


FIGURE 5: Carte OD avec lignes droites

Cette visualisation devient cependant très vite illisible s'il y a trop d'utilisateurs, ou de couples Entrée-Sortie.

2.1.3 Cercles

La seconde visualisation OD réalisée (*Figure 6*) représente le nombre de véhicules entrants et sortants.

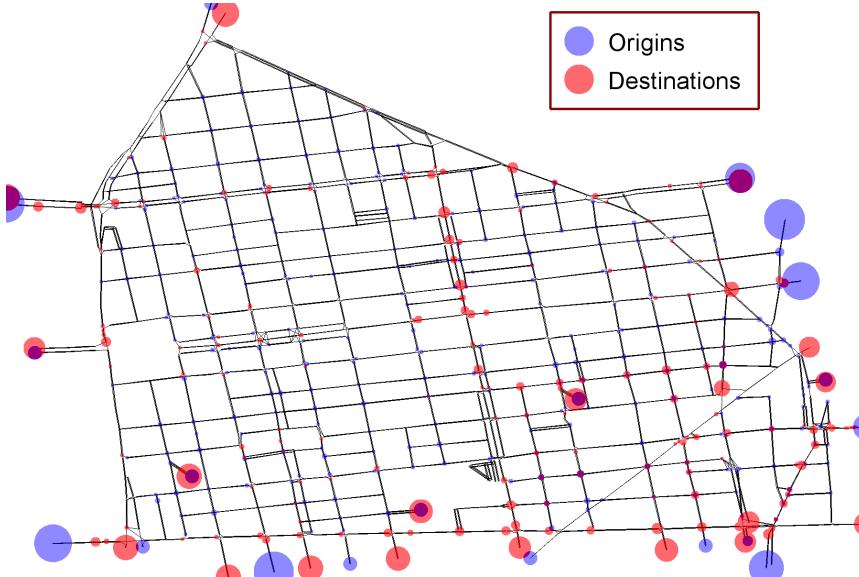


FIGURE 6: Carte OD avec cercles

Dans la figure ci-dessus chaque disque bleu correspond à un point d'entrée et chaque disque rouge à un point de sortie. Le rayon du disque varie en fonction du nombre d'utilisateurs : plus il est grand, plus il y a d'utilisateurs. Grâce à cette représentation, on accède directement aux points d'entrée-sortie les plus usités du réseau, mais on ne sait pas où les utilisateurs vont ni d'où ils viennent.

2.1.4 Cartes internes

Un bon compromis qui regroupe les deux méthodes précédentes est la visualisation par cartes internes. Pour cette méthode on divise la carte en n^2 secteurs, et dans chaque secteur, on représente de nouveau la carte de base divisée en ces mêmes n^2 secteurs.

On peut ensuite définir $N_{i,j}$, le nombre d'utilisateurs entrants dans le secteur i et sortants par le secteur j . Dans le secteur i , on représente les $N_{i,j}$ pour $j \in [1, n^2]$.

En associant une couleur à chaque nombre entre $\min(N_{i,j})$ et $\max(N_{i,j})$ selon un dégradé on obtient des images comme la *Figure 7* :



FIGURE 7: Carte OD avec "cartes internes"

On peut ainsi visualiser les entrées et sorties du réseau, mais également se faire une idée des trajets empruntés. Cette visualisation est plus complexe visuellement mais permet d'obtenir rapidement et clairement beaucoup d'informations sur les destinations privilégiées des utilisateurs.

On peut faire varier le paramètre n afin d'avoir plus ou moins de précision (voir *Figure 8* et *9*).

Exemple pour $n = 9$:



FIGURE 8: Carte OD avec "cartes internes" 9x9

Exemple pour $n = 23$:

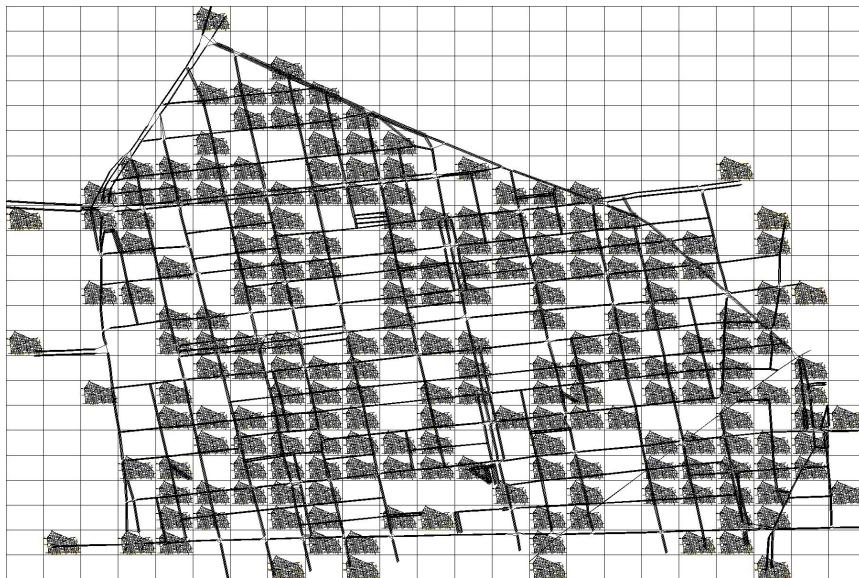


FIGURE 9: Carte OD avec "cartes internes" 23x23

2.2 Visualisation des trajets

Après avoir visualisé les Entrées et Sorties, on s'intéresse maintenant aux trajets empruntés par les différents utilisateurs. Ces données sont plus complètes et nécessitent donc souvent des visualisation plus complexes.

2.2.1 Vitesse limite

La première visualisation effectuée n'est basée sur des données du simulateur mais sur un autre jeu de données donnant les limitations de vitesses de Lyon 6 et Villeurbanne.

En *Figure 10* les tronçons sont colorés suivant leur limite de vitesse.



FIGURE 10: Carte colorée en fonction des limitations de vitesse

Le code couleur s'étend du rouge au vert, le rouge correspondant aux tronçons de plus faible limite.

2.2.2 Encombrement des tronçons

On a ensuite exploité les données fournies par l'ENTPE afin de tracer un bilan de l'encombrement total des routes durant la simulation. Le résultat de la visualisation est donné en *Figure 11*.



FIGURE 11: Coloration des voies en fonction de leur encombrement

Bien que le résultat de cette visualisation soit assez simple à appréhender, il n'est pas exploitable. En effet certains tronçons arrivent à "saturation" assez vite et on ne voit pas de différences entre un tronçon moyennement emprunté et un tronçon très emprunté. Cette visualisation permettrait tout au plus de repérer les tronçons qui ne sont pas empruntés.

2.3 MotionRugs

La visualisation par MotionRugs permet de représenter une donnée dépendant de la position et du temps (comme par exemple la moyennes des vitesses instantanées sur un tronçon, le nombre de véhicules à un endroit et un instant donné...). Cette visualisation se fait sous la forme d'un "tapis" 2D.

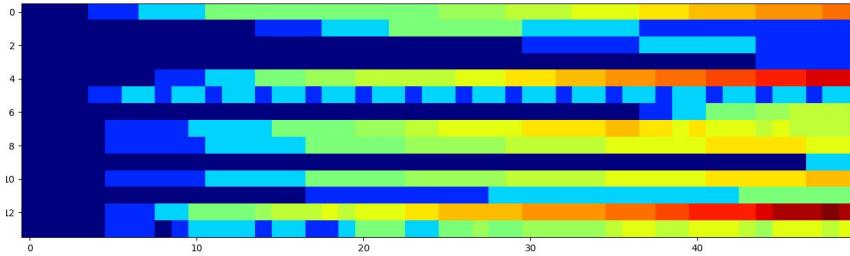


FIGURE 12: MotionRugs

En *Figure 12*, on représente l'écart relatif entre la vitesse maximale autorisée et la vitesse réelle (*bleu = normal, rouge = lent*). L'axe des abscisses représente le temps, et l'axe des ordonnées une représentation en 1D de l'espace 2D (on expliquera plus en détail dans la partie ci-après).

2.3.1 De la 2D à la 1D

Le principal enjeu de cette visualisation est le passage de deux dimensions à une seule dimension pour l'espace. En effet on a au dessus qu'on représentait l'espace en ordonnée mais dans notre cas l'espace est un plan à deux dimensions (x,y).

La première chose à faire est de discréteriser l'espace : on le découpe en $n \times n$ sections. On a alors une matrice $A = (a_{i,j})_{(i,j) \in \llbracket 1; n \rrbracket^2}$ de sections. On approxime alors chaque section à un point (tous les véhicules situés dans ce secteur sont au même point). Pour linéariser l'espace on pourrait tout d'abord remplacer cette matrice par une liste en accolant chaque ligne ou chaque colonne à la suite. Mais on aurait alors une discontinuité de l'espace et le tapis présenterait lui aussi des discontinuités.

Pour palier à ce problème on utilise la courbe de Hilbert : c'est une courbe continue qui remplit un carré. Elle est définie comme $\lim_{n \rightarrow \infty} H_n$. On visualise en *Figure 13* les H_n pour $n \in \llbracket 1; 6 \rrbracket$

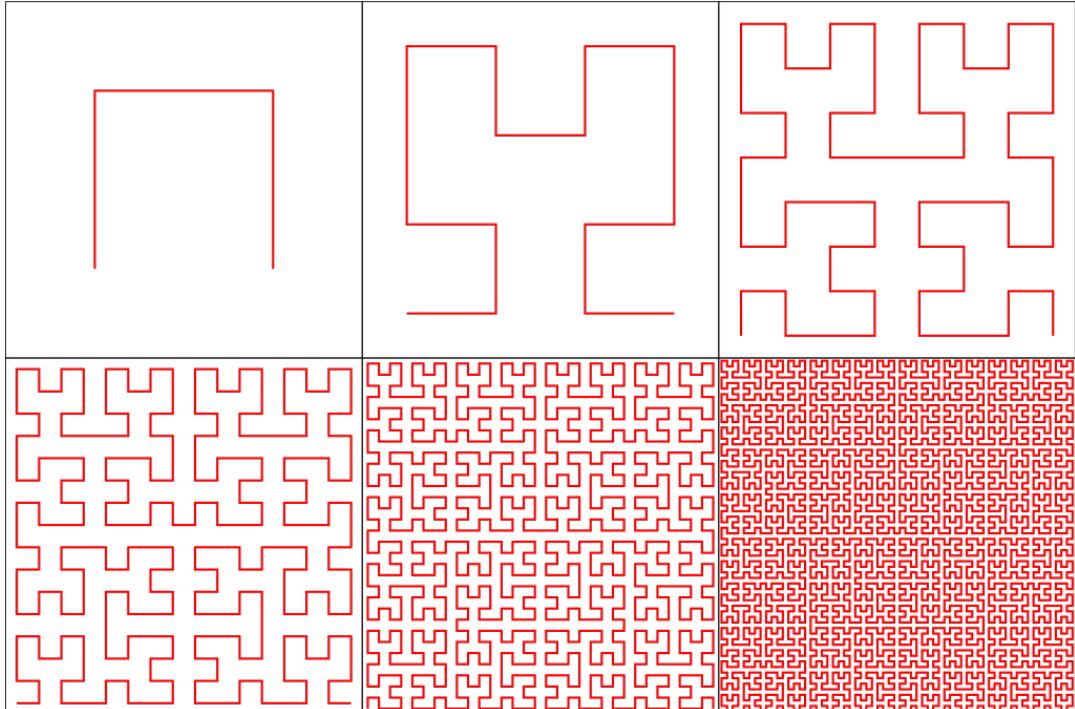


FIGURE 13: 6 premières étapes de la construction de la courbe de Hilbert

H_n divise l'espace en $2^n \times 2^n$ sections : on approxime chaque point du plan au point le plus proche de la courbe.

Il suffit ensuite d'"étirer" la courbe pour aboutir à une visualisation de l'espace en une seule dimension.

2.3.2 Visualisation et interprétation

La visualisation en *Figure 12* est difficilement lisible. On remarque que des tronçons sont encombrés (en rouge), mais on ne sait pas lesquels. La représentation en 1D de l'espace 2D n'est pas intuitive et ne permet pas une lecture directe du graphe. Il faudrait revoir la fonction de répartition afin qu'elle soit personnalisée pour notre cas d'étude et qu'elle soit plus adaptée, en mettant par exemple les tronçons principaux côte à côte.

3 Visualisations dynamiques

Les visualisations dynamiques sont des animations : l'image n'est pas fixe sur l'interface machine-utilisateur, et ne permet donc pas une compréhension instantanée. Elles sont plus coûteuses en temps de calcul et en espace mémoire, mais permettent d'accéder à des représentations plus intuitives des données de la simulation.

Le passage d'une visualisation statique à une visualisation dynamique permet d'ajouter une dimension à la visualisation. Le plus naturel est d'ajouter la dimension temporelle de la simulation mais on peut imaginer d'autres dimensions (représentation 3D grâce à une animation mobile par exemple). On se contentera de la dimension temporelle dans le cadre de notre projet.

3.1 Replay

Cette visualisation est la plus naturelle que l'on puisse imaginer : il s'agit de placer les véhicules présents à un instant dans le réseau sur une image puis de recommencer à l'instant suivant.

On obtient alors une reproduction de la simulation (*Figure 14 et 15*), les points rouges sont les véhicules.

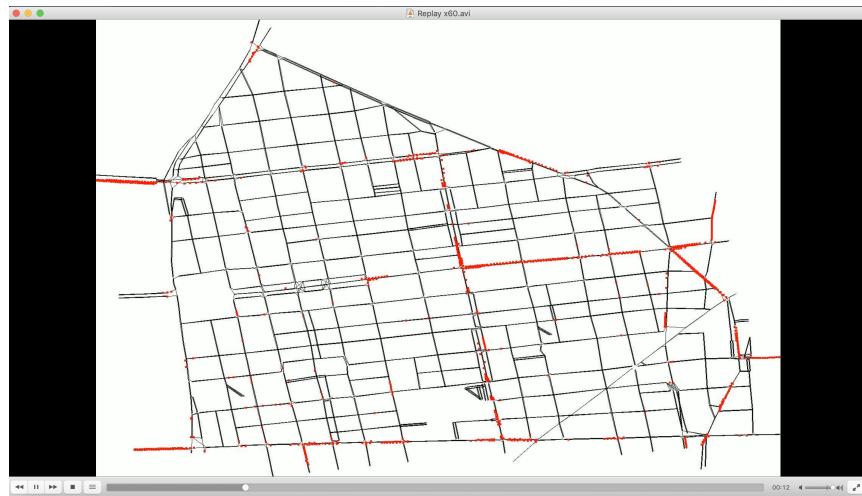


FIGURE 14: Premier arrêt sur image d'un Replay x60

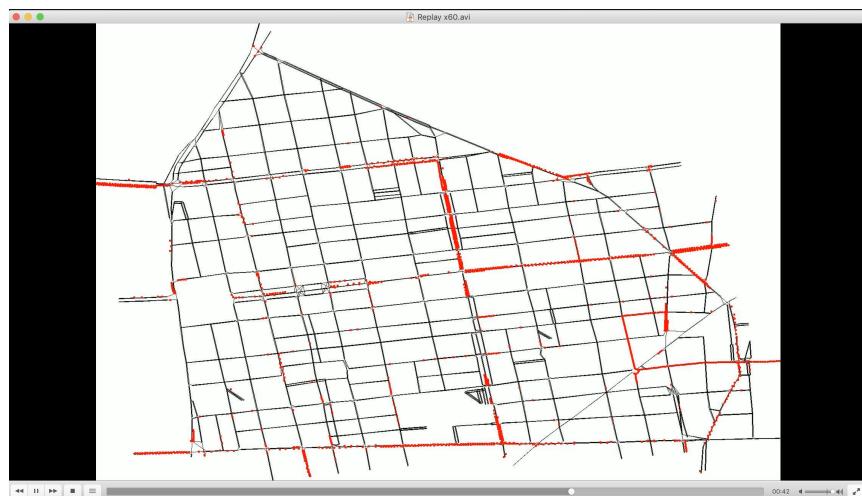


FIGURE 15: Deuxième arrêt sur image d'un Replay x60

On peut jouer sur la vitesse de défilement en tant que paramètre de calcul suivant la précision voulue.

3.1.1 Avantages

Avec cette visualisation, on peut facilement repérer les congestions sur le réseau. Elle permet aussi de vérifier que la simulation s'est bien déroulée en observant les comportements des différents usagers.

3.1.2 Limites

Cette visualisation a deux principaux défauts. Premièrement, le temps de calcul est assez conséquent : une vidéo de 1 minute nécessite 6 ou 7 minutes de calcul. Secondement, le format vidéo de la visualisation prend beaucoup plus d'espace mémoire qu'une visualisation statique.

3.2 Densité de véhicules par gradient

Cette visualisation a pour but de visualiser la densité de véhicules au cours du temps.

Pour cela on définit une fonction f_t telle que

$$f_t(x, y) = \sum_i^n e^{-\alpha \cdot d_i(x, y)} \quad (1)$$

Où n est le nombre de véhicules sur le réseau, $d_i(x, y)$ la distance entre le véhicule i et le point (x, y) à l'instant t .

$f_t(x, y)$ donne alors une idée du nombre de véhicules aux alentours du point (x, y) à l'instant t .

On norme ensuite l'ensemble des fonctions obtenues de la manière suivante :

$$f_0 = \max_{t \in \text{temps}} (\max_{(x, y) \in \text{espace}} (f_t(x, y))) \quad (2)$$

On peut associer une couleur à chaque $f_t(x, y)$, de manière à obtenir un dégradé de rouge à bleu en passant par le blanc selon qu'il y ait plus ou moins de véhicules. On note $c_t(x, y)$ la couleur, en RVB, du point (x, y) à l'instant t . On a alors :

$$c_t(x, y) = \begin{cases} (255, 255 \cdot 2 \frac{f_t(x, y)}{f_0}, 255 \cdot 2 \frac{f_t(x, y)}{f_0}) & \text{si } \frac{f_t(x, y)}{f_0} < \frac{1}{2} \\ (255 \cdot 2(1 - \frac{f_t(x, y)}{f_0}), 255 \cdot 2(1 - \frac{f_t(x, y)}{f_0}), 255) & \text{si } \frac{f_t(x, y)}{f_0} > \frac{1}{2} \end{cases} \quad (3)$$

Une fois qu'on a accès à $c_t(x, y) \forall x, y, t$ on peut créer l'animation.

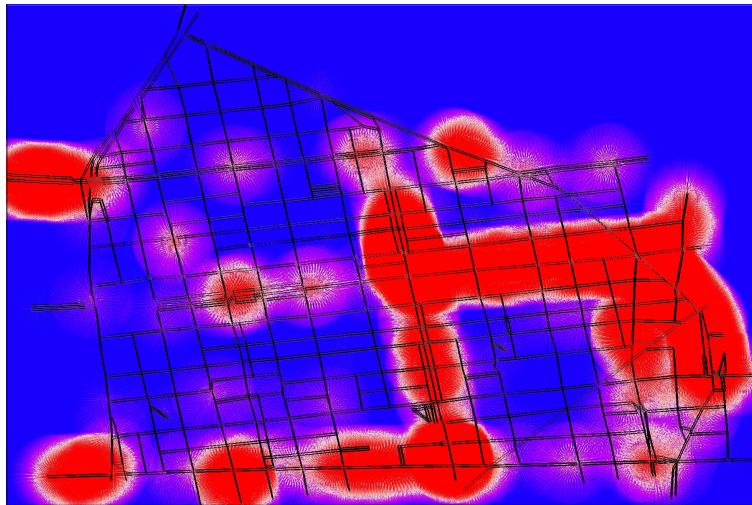


FIGURE 16: Capture d'un instant de la visualisation

On observe en *Figure 16* la capture d'un instant de la visualisation. Le paramètre α a été fixé arbitrairement à $\alpha = 0.02$. La visualisation permet d'obtenir un aperçu instantané des zones de fort trafic mais elle prend beaucoup trop de temps à être obtenue : nous n'avons pas pu obtenir de vidéo complète pour toute la durée de la simulation. Il faudrait donc revoir la méthode de calcul et/ou la résolution graphique souhaitée.

Conclusion

En explorant de nombreuses pistes de visualisation, nous avons obtenu de nombreux résultats très variés. Cependant, certains ne sont pas à la hauteur de nos espérances en terme de lisibilité ou de facilité d'obtention. Nous avons documenté nos codes pour que nos contacts à l'ENTPE puissent les utiliser facilement et efficacement. Certains de nos codes ne sont pas fonctionnels, le facteur limitant a souvent été le temps. Nous n'avons obtenus aucun résultats complet pour la vidéo à zones de chaleur car le calcul est trop complexe et il aurait fallu repenser complètement notre manière de calculer la valeur des pixels de chaque image.

Les codes que nous avons fournis permettent d'exploiter les données XML sortants du simulateur, elles permettent de visualiser de nouvelles données afin de comparer les valeurs de leurs attributs.

4 Bibliographie

Guillaume Costeseque, *Modélisation du trafic routier : passage du microscopique au macroscopique*, 2012

C. Buisson, J.B. Le Sort, *Comprendre le trafic routier*

H. Guo, Z. Wang, B. Yu, H. Zhao, X. Yuan, *TripVista : Triple Perspective Visual Trajectory Analytics and Its Application on Microscopic Traffic Data at a Road Intersection*, 2011

J. Buchmüller, *MotionRug : Visualizing Collective Trends in Space and Time*, 2018

Y. Yang, T. Dwyer, S. Goodwin, K. Marriott, *Many-to-Many Geographically-Embedded Flow Visualisation : An Evaluation*

J. Wood, J. Dykes, A. Slingsby, *Visualization of Origins, Destinations and Flows with OD Maps*, 2010

N. Ferreira, J. Poco, H. Vo, J. Freire, C. Silva, *TaxiVis*, 2013

P. Riviere, *MotionRugs*, 2019

5 Annexes : Exemples de Codes

Chaque visualisation est associée à un code Python par soucis de clarté. Dans une optique de lisibilité et de déboguage efficace, chaque code est autant que possible décomposé en fonctions auxiliaires.

5.1 Replays

```
9 import sys
10 import Trace_carte
11 import Save_Instants
12 import cv2
13 from PIL import Image, ImageDraw
14 import time
15 import os
16
17
18 path_to_XML = '/Users/fabienduranson/Desktop/Pouge/PAr/Data/SymuviaOut_000000_010000_traf_SO_media'
19 path_to_network = '/Users/fabienduranson/Desktop/Pouge/PAr/Data/networkLyon6.xml'
20 saving_path = '/Users/fabienduranson/Desktop/Pouge/PAr/Results/Dynamic/Replays/'
21
22 def create_blancl():
23     w,h = 1800,1200
24     Lyon6x3 = Trace_carte.Network('Lyon mamene')
25     Lyon6x3.from_XML(path_to_network)
26     im = Image.new('RGBA',(w,h), (255,255,255,0))
27     draw = ImageDraw.Draw(im)
28     lim = Lyon6x3.limits()
29     l = 2
30     Trace_carte.trace(draw,w,lim,(0,w,0,h),l)
31     im.save('Blanc.jpg','JPEG')
32     return im
33
34
35 def frame(ind,video,instant):
36     im = Image.open('Blanc.jpg')
37     vehs = instant['vehs']
38     for veh in vehs:
39         place(veh,im)
40     name = 'frame{}.jpg'.format(ind)
41     im.save(name,'JPEG')
42     video.write(cv2.imread(name))
43     os.remove(name)
44
45 def place(veh,im):
46     x,y = veh['x'],veh['y']
47     for i in range(-2,3):
48         for j in range(-2,3):
49             im.putpixel((max(0,min(x+i,1799)),max(0,min(y+j,1199))), (255,0,0))
50
51 def temps(t):
52     h = t//3600
53     m = (t%3600)//60
54     s = t%60
55     return '{} h {} min et {} secondes'.format(h,m,s)
56
57 def play(fact):    #1 = normal speed; 10 = x10 speed
```

FIGURE 17: Code partiel de création de Replays

La démarche adoptée ici est de créer une image par "instant" du fichier XML et de créer la vidéo en concaténant les images.

La librairie PIL permet de créer des images en formats '.bmp' grâce à la fonction `putpixel` du module `ImageDraw`. Cette fonction permet de créer une image en définissant la couleur de chaque pixel. La fonction `trace` de `Trace_carte` crée une image 'Blanc.jpg'. C'est la carte vierge de Lyon6. La fonction `frame` change

5.2 Cercles

```

50
51 def valide(pt):
52     x,y = pt
53     return (0<x and x<w and 0<y and y<h)
54
55 global w
56 global h
57 w,h = 1800,1200
58 Lyon6x3 = Trace_carte.Network('Lyon mamene')
59 Lyon6x3.from_XML(path_to_network)
60 im = Image.new('RGBA',(w,h), (255,255,255,0))
61 draw = ImageDraw.Draw(im)
62 lim = Lyon6x3.limits()
63 l = 2
64 Trace_carte.trace(draw,h,w,lim,(0,w,0,h),l)
65
66 #im,draw = basic_frame(w,h,"circles",[])
67
68 ODm = ExtractODMat.ODMatFromTxt('ODmat.txt')
69
70 ODmat = ODm['mat']
71 origins = ODm['origins']
72 destinations = ODm['destinations']
73
74 nbr_origin = []
75 nbr_dest = []
76
77
78 for i in range(len(ODmat)):
79     nbr_origin.append((sum(ODmat[i]),origins[i]))
80 for i in range(len(ODmat[0])):
81     d = [ODmat[j][i] for j in range(len(ODmat))]
82     nbr_dest.append((sum(d),destinations[i]))
83
84
85 max_o = max([i[0] for i in nbr_origin])
86 max_d = max([i[0] for i in nbr_dest])
87
88 #for ori in nbr_origin:
89 #    n,Centre = ori
90 #    centre = scl(Centre)
91 #    r = rayon(n,max_o)
92 #    couleur = (0,0,255)
93 #    circle(im,centre,r,couleur)
94
95 for dest in nbr_dest:
96     n,Centre = dest
97     centre = scl(Centre)
98     r = rayon(n,max_d)
99     couleur = (255,0,0)
100    circle(im,centre,r,couleur)
101
102 ...

```

FIGURE 18: Code partiel de création de Cercles - Corps

Le programme se base sur le fichier ExtractODMat.py. Ce dernier permet d'obtenir la matrice O-D de la simulation. Le programme va ensuite exploiter cette matrice et croiser les informations afin de placer des cercles rouges et bleus aux bons endroits.