

RECONSTITUTION 3D ET DÉTECTION D'OBJETS EN MILIEU URBAIN

Projet INF552
Rapport Final

Pour le 8 Décembre 2017

Romain LOISEAU
Lucas BROUX



Table des matières

1	Introduction	3
2	Algorithmes utilisés et raisonnements	4
2.1	Reconstitution de nuage 3D	4
2.2	Détection du sol	4
2.3	Détection d'objets verticaux	5
2.3.1	Variante de l'algorithme RANSAC	5
2.3.2	Algorithme des k-moyennes	5
3	Résultats, performance, perspectives d'amélioration	7
3.1	Exemples de résultats	7
3.1.1	Nuage de points 3D	7
3.1.2	Détection de la route et des éléments verticaux par RANSAC	7
3.1.3	Détection d'objets par clustering	9
3.2	Analyse de la performance	10
3.3	Perspectives d'amélioration	14
4	Conclusion	15

1 Introduction

Ce projet, réalisé dans le cadre du cours *INF552 - Analyse d'Images et Vision par Ordinateur* consiste en l'analyse des images de la base de données *Cityscapes* (<https://www.cityscapes-dataset.com/>) par les méthodes et algorithmes vus en cours. Cette base de données consiste en des images stéréo d'environnements urbains, prises par deux caméras installées sur le capot d'une voiture, dont on connaît à tout instant les matrices. Les auteurs de la base de données fournissent les images prises par la caméra de gauche, la disparité calculée avec l'image de droite, ainsi que les informations géométriques des deux caméras.

Notre objectif est le suivant : étant donnés une image de gauche, la disparité correspondante, ainsi que les données géométriques des deux caméras, reconnaître des objets importants du paysages en n'utilisant que des algorithmes et méthodes étudiés en cours.

Notre approche est la suivante : dans un premier temps, nous utilisons ces informations pour reconstituer un nuage de point 3D correspondant à la prise de vue ; puis nous appliquons l'algorithme RANSAC pour détecter le plan principal de ce nuage de points, le sol ; enfin, nous avons développé plusieurs approches (RANSAC, *clustering*) pour détecter les objets verticaux i.e. orthogonaux à ce plan.

Dans ce rapport, nous présentons tout d'abord notre approche et précisons les algorithmes employés. Nous présentons ensuite les résultats obtenus et analysons la performance de nos algorithmes. Enfin, nous concluons en présentant les enseignements acquis au cours de ce projet.

Le code source du projet est disponible sur ce repository github : <https://github.com/lucas-broux/Projet-Inf552>

2 Algorithmes utilisés et raisonnements

2.1 Reconstitution de nuage 3D

Nous nous donnons l'image prise par l'appareil de gauche, marquée par les coordonnées x et y , ainsi que l'image correspondant à la disparité $d(x, y)$, et les données géométriques des deux caméras i.e. leur paramètres intrinsèques et extrinsèques.

Soit X, Y , et Z les coordonnées 3D correspondant au point x, y sur l'image de gauche, et $d := d(x, y)$. On note aussi t la valeur de translation horizontale entre les deux caméras. Les équations de projection s'écrivent alors :

$$\left\{ \begin{array}{l} s * \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \\ s * \begin{bmatrix} x - d \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & t \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \end{array} \right.$$

à résoudre en s, X, Y , et Z . L'inversion du système donne :

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = -\frac{t}{d} \underbrace{\begin{bmatrix} 1 & 0 & -c_x \\ 0 & f_x & -f_x c_y \\ 0 & f_y & f_y \\ 0 & 0 & f_x \end{bmatrix}}_{:=N} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Ainsi, le calcul des points 3D revient à une simple multiplication matricielle.

2.2 Détection du sol

Pour détecter le sol, nous cherchons le plan principal du nuage de points 3D, à l'aide de l'algorithme RANSAC.

Cet algorithme consiste à tirer aléatoirement un grand nombre de fois trois points dans le nuage de points, puis de regarder combien de points sont dans le plan défini par ces trois points. A l'issue du processus, on retient le plan contenant le plus de points.

Nous avons besoin de deux paramètres : l'épaisseur d'un plan et le nombre d'itérations de l'algorithme.

L'épaisseur d'un plan est défini à partir d'une longueur caractéristique du nuage de point. En effet, nous pouvons essayer de calculer la distance moyenne entre deux points voisins afin d'obtenir une longueur caractéristique. Or calculer pour chaque point la distance à son plus proche voisin à une complexité en n^2 avec la structure de donnée que nous utilisons (Rappel : nous utilisons un vecteur de *Vec3d* pour stocker notre nuage de point). Comme notre nuage de point à de l'ordre du million de point, le calcul exact de cette grandeur serait trop gourmand pour être exécuté en un temps raisonnable. Pour faire face à ce problème, nous tirons aléatoirement 1000 points pour lesquels nous cherchons la plus petite distance avec 10000 autres points tirés aléatoirement. Nous faisons donc de l'ordre de $1000 * 10000 = 10000000$ d'opérations, ce qui est soutenable par la machine. En contrepartie,

nous n'obtenons pas la grandeur désirée mais une approximation supérieur à cette grandeur qui suffira pour les besoins du projet. Dans l'algorithme, tout point qui sera à une distance inférieure à cette grandeur caractéristique du nuage de point du plan sera considéré comme appartenant au plan.

Le second paramètre à calculer est le nombre d'itérations suffisant. La page wikipédia suivante explique comment calibrer ce paramètre à notre problème.

Désignons par w la probabilité qu'un point appartienne au plan. Cette grandeur n'est en général pas connue, mais nous pouvons en donner une approximation. Dans notre cas, nous avons estimé que $w = 0.2$. Comme il faut 3 points pour définir le plan, $1 - w^3$ est la probabilité qu'au moins un point parmi ceux tirés n'appartienne pas au plan. En désignant par p la probabilité de réussite de l'algorithme attendue et par k le nombre d'itérations du modèle, nous en déduisons que $1 - p = (1 - w^3)^k$. Cela implique donc que :

$$k = \frac{\log 1 - p}{\log(1 - w^3)}$$

Le paramètre entré par l'utilisateur est donc une probabilité de réussite de l'algorithme p . A cela nous ajoutons deux fois l'écart type de k afin de s'assurer de la réussite de l'algorithme. Cet écart type vaut $\frac{\sqrt{1-w^3}}{w^3}$.

2.3 Détection d'objets verticaux

Nous proposons deux approches pour réaliser la détection des objets verticaux sur la route.

2.3.1 Variante de l'algorithme RANSAC

Notre première idée est d'appliquer l'algorithme RANSAC dans le but de rechercher des lignes dans l'espace ; en ajoutant une contrainte sur leur verticalité. Pour cela, nous utilisons le même algorithme que précédemment en l'adaptant aux lignes. De plus, nous recherchons plusieurs lignes. De plus, nous ajoutons une contrainte sur la verticalité des lignes détectées en s'assurant que l'angle entre la normale au plan de la route et la direction de la ligne trouvée soit inférieur à une constante fixée au préalable.

Cette méthode est assez mauvaise dans la détection des objets verticaux. En effet, dans le nuage de points on ne distingue pas vraiment de lignes, et il semble donc compliqué pour notre algorithme de fonctionner efficacement sur ce type de données.

2.3.2 Algorithme des k-moyennes

Notre seconde approche est fondée sur l'algorithme des k-moyennes. Nous commençons par choisir k centres de *clusters* aléatoirement dans le nuage de point. Ensuite nous calculons les *clusters* en assignant chaque point au barycentre dont il est le plus proche. Nous itérons ensuite jusqu'à ce que le modèle converge. Voici les quelques spécificités de notre algorithme :

- Le nombre d'itérations est calculé de telle sorte que l'algorithme s'arrête lorsque les barycentres se stabilisent. Dans notre cas, lorsque la variation de la variation de barycentre moyenne est inférieure à 1% de la variation de barycentre moyenne du pas précédent. Pour éviter toutefois de boucler indéfiniment, nous avons ajouté un seuil à ne pas dépasser pour le nombre d'itérations.
- Nous changeons également la topologie de l'espace des positions. En effet, les objets que nous souhaitons détecter sont des objets verticaux. Ils sont donc par nature étirés selon l'altitude. Pour que l'algorithme les détecte plus facilement, nous comprimons l'axe vertical afin de rapprocher les points selon cet axe. Ainsi, cela revient à avoir un algorithme qui regarde le nuage de points

du ciel. Cela se fait en deux étapes. Tout d'abord, nous faisons un changement de base en fixant l'axe vertical du monde réel comme égal au vecteur normal au plan trouvé précédemment. Ensuite, nous utilisons trois variables constantes spécifiant le degré de compression selon chacun des axes.

- const double *SCALER_X* = 0.5;
- const double *SCALER_Y* = 1;
- const double *SCALER_Z* = 1;

Ainsi, à chaque calcul de distance, nous multiplierons chacune des dimensions par le coefficient adéquat

- Enfin, nous avons la possibilité d'utiliser l'algorithme en prenant en entrée la couleur de chacun des points. Cela nous permet de prendre en compte le fait que lorsqu'on veut détecter un objet vertical, on a de bonnes raisons de penser que sa couleur va être homogène. Ainsi, nous pouvons réaliser l'algorithme en utilisant non plus trois dimensions, mais six. Nous commençons donc par définir un paramètre permettant de définir l'importance de la couleur dans l'algorithme. A l'image des trois paramètres utilisés précédemment, cela permet de changer la topologie de l'espace pour donner plus ou moins d'importance à la couleur. Nous utilisons ensuite ce paramètre en le définissant comme le produit de deux termes
 - Le rapport de l'écart-type en position et de l'écart-type en couleur. Cela permet de procéder à une normalisation des couleurs par rapport aux positions. En effet, il n'y a pas de raison que les valeurs de position et de couleurs évoluent dans le mêmes ordres de grandeur.
 - Un paramètre ressemblant aux paramètres nommés précédemment qui permet de donner arbitrairement plus ou moins d'importance à la couleur dans le calcul des k-moyennes.
- Le score que nous utilisons pour évaluer la performance est un score nommé *silhouette score*. Il permet de s'assurer que l'algorithme cherche le bon nombre de *clusters*. Il s'intéresse à la différence entre la distance d'un point à son *cluster* et la distance du même point au *cluster* le plus proche dont il ne fait pas partie.

3 Résultats, performance, perspectives d'amélioration

3.1 Exemples de résultats

Voici quelques visualisations des résultats obtenus :

3.1.1 Nuage de points 3D



FIGURE 1 – Image d'origine et sa disparité

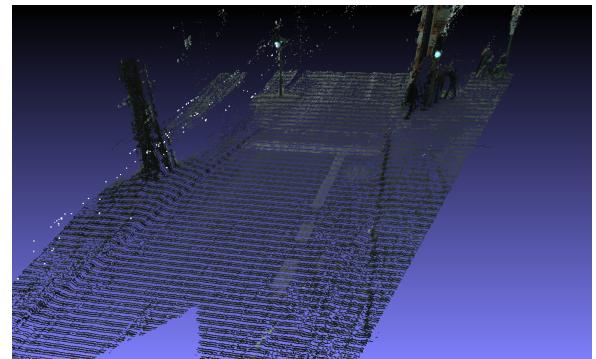
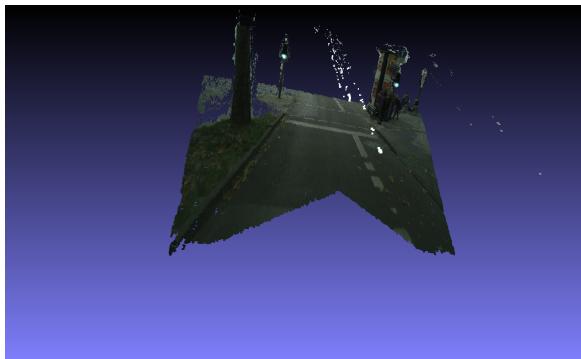


FIGURE 2 – Nuage de point 3d obtenu (Visualisation sur MeshLab)

3.1.2 Détection de la route et des éléments verticaux par RANSAC

Voilà les résultats obtenus sur une dizaine d'images pour notre implémentation de l'algorithme de RANSAC. Nous représentons en rouge la route, en vert les éléments verticaux.



FIGURE 3 – Algorithme RANSAC



FIGURE 4 – Algorithme RANSAC



FIGURE 5 – Algorithme RANSAC



FIGURE 6 – Algorithme RANSAC



FIGURE 7 – Algorithme RANSAC

Nous remarquons que l'algorithme de détection de la route est très précis, retournant le plan correct sur chacune des images.

En revanche, si nous constatons que l'algorithme est efficace pour repérer les éléments verticaux (poteaux, passants, arbres, ...), il ne permet pas de distinguer certains objets plus aplatis ou plans (voitures, murs, ...) et renvoie des résultats aberrants dans le cas où aucun objet vertical n'est présent sur l'image. Cela est du au fait que notre algorithme est conçu pour renvoyer un nombre fixe de lignes verticales.

3.1.3 Détection d'objets par clustering

Voilà les résultats obtenus sur une dizaine d'images pour notre algorithme de *clustering*.



FIGURE 8 – Algorithme des k-moyennes



FIGURE 9 – Algorithme des k-moyennes



FIGURE 10 – Algorithme des k-moyennes



FIGURE 11 – Algorithme des k-moyennes

Nos résultats sont mitigés. Les *clusters* renvoyés par l'algorithme sont constitués en réalité de pans de l'espace agglomérés ensemble, et comportent parfois de nombreux objets distincts ou coupent ces objets en deux. Nous pensons que cela est dû à plusieurs facteurs :

- D'une part, les algorithmes mis en oeuvre reposent sur un certain nombre d'hyperparamètres qu'il faudrait analyser plus en détail pour pouvoir en comprendre l'impact et optimiser l'algorithme.
- En outre, l'algorithme choisi pour réaliser le *clustering* (algorithme des k-moyennes) est relativement basique et possède les inconvénients suivants vis à vis de nos données :
 - Il est basé sur l'hypothèse principale selon laquelle les *clusters* à trouver sont globulaires, ce qui a pour effet de couper en plusieurs parties certains *clusters* naturels qui ne le sont pas (haies, arbres, ...).
 - Il prend en paramètre le nombre voulu de *clusters*, qui n'est pas facile à estimer. Pour pallier ce problème, nous appliquons plusieurs fois l'algorithme en faisant varier la valeur de ce paramètre et nous appliquons un score (lui aussi assez simple) pour quantifier la pertinence du *clustering*. Toutefois ce procédé contribue à une forte augmentation du temps de calcul.
 - Enfin, l'algorithme ne prend pas en compte le bruit dans les données du problème, qui est pourtant bien présent à cause des légères imperfections dans les images de disparité fournies par le *cityscapes dataset*.

Nous pensons que d'autres algorithmes que nous n'avons pas pu implémenter pour des raisons de temps - par exemple l'algorithme DBSCAN, qui résout les 3 inconvénients ci-dessus - pourraient être plus efficaces et précis.

3.2 Analyse de la performance

Pour juger la performance temporelle de nos algorithmes, nous avons réalisé une plateforme de tests, que nous avons appliquée à 57 images, sur un ordinateur portable standard.

Voici les histogrammes des temps de calcul des différents algorithmes employés :

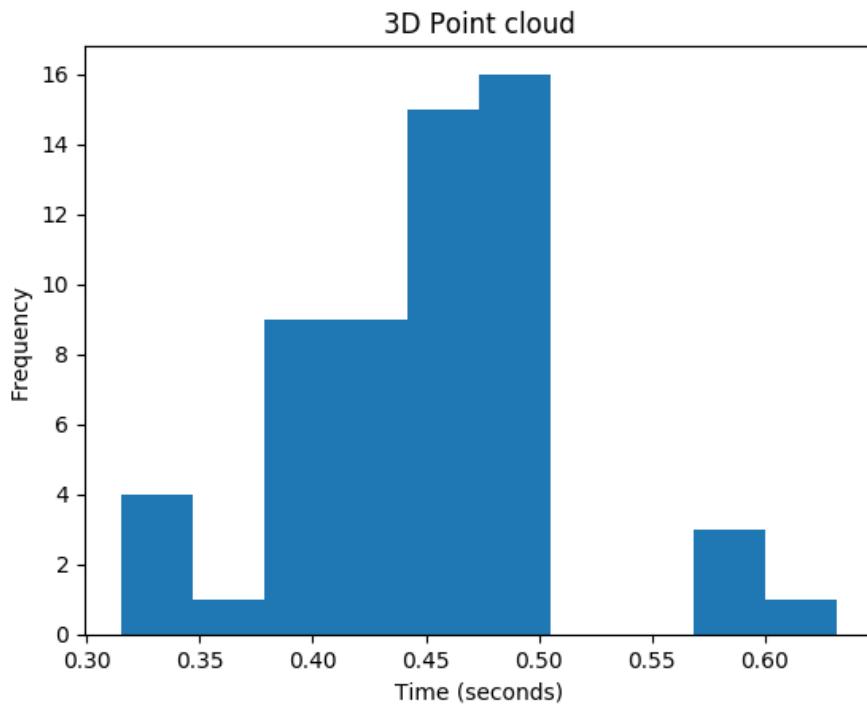


FIGURE 12 – Temps de calcul du nuage de points 3d

Nous constatons que le calcul du nuage de points 3D est relativement rapide (moins d'une demi-seconde quasiment systématiquement), en particulier étant donné le nombre de sommets calculés, qui est de l'ordre du million.

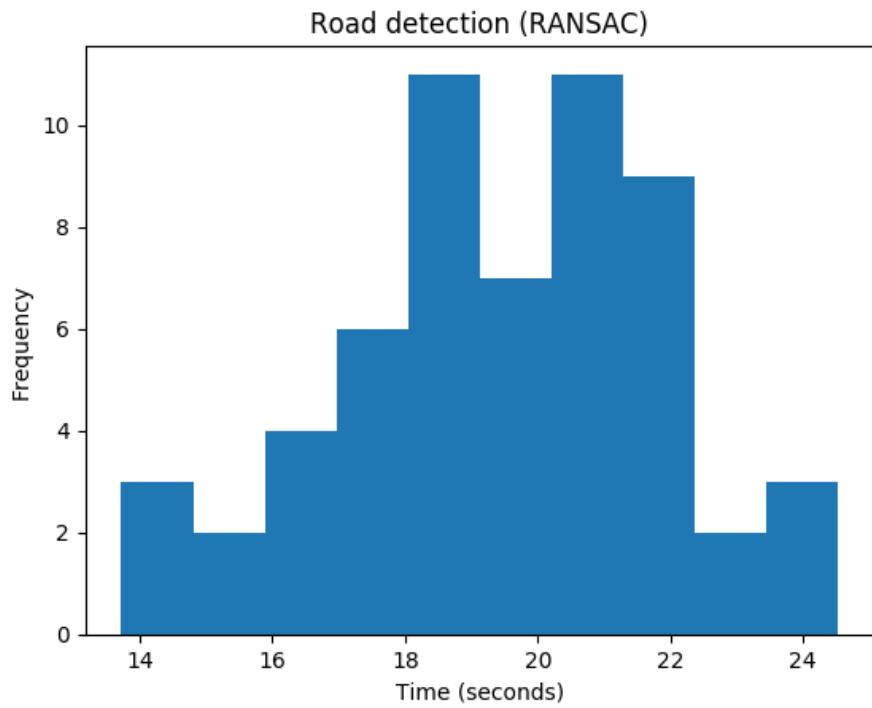


FIGURE 13 – Temps de calcul pour la détection de la route

L’application de l’algorithme RANSAC pour la détection de la route est ainsi particulièrement long (de l’ordre de 20 secondes en moyenne). C’est donc une opération limitante d’un point de vue temporel.

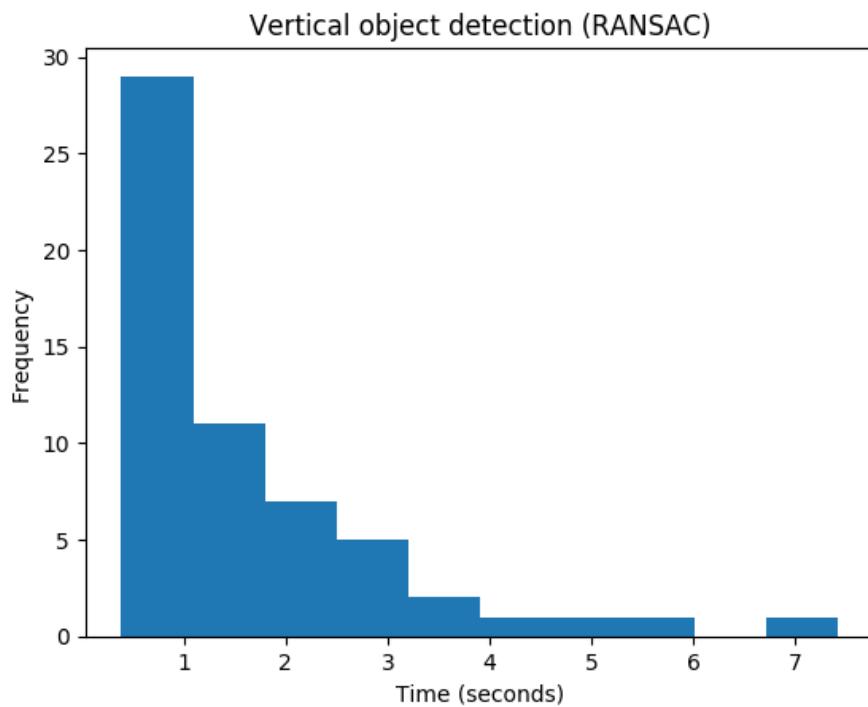


FIGURE 14 – Temps de calcul pour la détection des objets verticaux (RANSAC)

Pour la détection des objets verticaux, l'algorithme RANSAC est donc relativement efficace, mais tout de même de l'ordre de plusieurs secondes en moyenne.

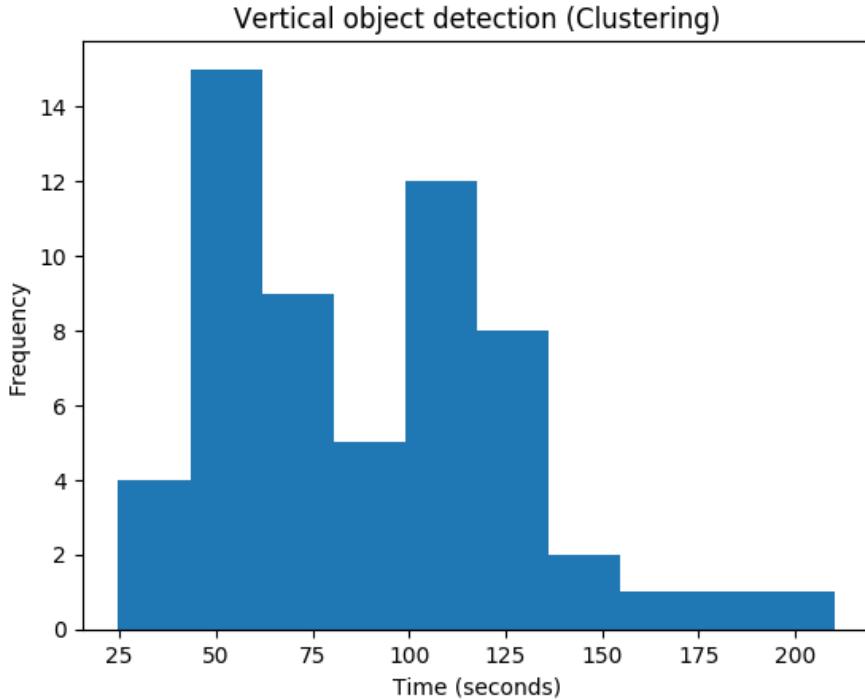


FIGURE 15 – Temps de calcul pour la détection des objets verticaux (k-moyennes)

Nous constatons que l'algorithme de *clustering* choisi est particulièrement lent. Comme présenté précédemment, ceci est en partie dû au fait que nous l'appliquons plusieurs fois pour trouver le nombre optimal de *clusters*, sachant que les données à analyser sont assez nombreuses (~ 1 million de points). Cela est peut-être aussi dû à nos choix d'implémentation : des structures de données plus efficaces et une meilleure prise en compte des possibilités d'optimisation du compilateur pourraient sûrement améliorer cette performance.

3.3 Perspectives d'amélioration

Ainsi, nous sommes parvenus à des résultats satisfaisants sur certains points :

- Le calcul du nuage de points 3D.
- La précision de la détection de la route.
- La précision des algorithmes de détection des objets.

En revanche, nous pensons qu'il est possible d'encore améliorer :

- La précision et la performance de l'algorithme de détection d'objets par *clustering*,
 - Soit en trouvant une combinaison d'hyperparamètres plus efficace,
 - Soit en appliquant d'autres algorithmes plus adaptés à nos données. Nous pensons particulièrement à l'algorithme DBSCAN, qui permettrait de contourner plusieurs difficultés et inconvénients de notre algorithme actuel. Toutefois, une implémentation efficace de cet algorithme n'est pas aisée puisqu'il repose sur des structures de données particulières.
- Le temps de calcul de certains algorithmes, en relâchant certains seuils d'algorithmes, mais aussi en optimisant les structures de données utilisées et en exploitant encore plus les possibilités du compilateur et du langage C++.

4 Conclusion

Nous avons appliqué différentes méthodes de vision par ordinateur à l'analyse du *cityscapes dataset*. Nous sommes parvenus à reproduire un nuage de point 3D à partir des informations données, à reconnaître le plan correspondant à la route, et - dans une certaine mesure - à reconnaître des objets verticaux du paysage.

Nous avons toutefois proposé plusieurs pistes d'amélioration :

- D'une part pour améliorer la précision de la méthode de détection d'objets verticaux.
- D'autre part pour améliorer la performance temporelle des algorithmes mis en œuvre.

Outre les résultats obtenus, ce projet nous a permis de mettre en application différents outils étudiés durant le cours : reconstitution 3D, algorithme RANSAC, bibliothèque OpenCv ... Cela a également été l'occasion pour nous de nous familiariser avec l'outil de gestion de versions *git*, qui nous a permis une grande souplesse dans la répartition du travail. Enfin, nous avons pu apprêhender la nécessité de bien organiser et documenter un projet, en particulier en C++.