MASTER 2 CALCUL HAUTE PERFORMANCE ET SIMULATION (CHPS)

RAPPORT DE STAGE DE FIN D'ÉTUDES

EDGE COMPUTING : DÉTECTION DE DEFORMATION ROUTIÈRE

Zhiyuan LIU

Responsable Entreprise :

M. Jean-Michel BATTO

Enseignant Référent :

Mme. Soraya ZERTAL

Présenté le 8 septembre 2021
Année universitaire 2020-2021

Remerciements

Avant le début de ce rapport, je voudrais adresser mes remerciements et ma reconnaissance à toutes les personnes qui m'ont accompagné et apporté leur aide et soutien au long de cette période.

Tout d'abord, je remercie **Jean-Michel BATTO**. Comme encadrant au sein de l'entreprise, il est toujours patient et disponible quand je rencontre des difficultés techniques. Il m'a apporté beaucoup d'indications, des conseils et de la confiance pour que je puisse aller jusqu'à la fin de ce sujet.

Je remercie aussi **Sara MOUKIR**, pour son aide et son soutien dans ma présence en entreprise et pour la réalisation de ce projet.

Je remercie **Thomas LACAZE**, pour son aide technique et la coopération pour le benchmark sur la plateforme Raspberry Pi.

Je remercie **Louiza HANIS** pour ses conseils utiles sur la rédaction de ce rapport.

Je remercie **Stéphane BAUDELOCQ**, Directeur de Forclum Numérique, pour son accueil, qui a rendu ce stage possible. Ainsi, je remercie toute l'équipe d'Eiffage Energie Systèmes pour son accueil et ses précieux conseils, qui m'ont beaucoup aidé pendant ce stage.

Enfin, je voudrais aussi remercier toute l'équipe des **enseignants du master CHPS** pour les connaissances et le professionnalisme qu'ils m'ont apportés pendant les deux ans de ce master.

Abstract

This report is about the work during my internship at Eiffage Energie Système. This internship is a part of the Master Degree in high performance computing and simulation (calcul de haute performance et simulation, CHPS).

This report presents a simple and low-cost method to detect deformations on a road surface. This method requires 3d point data taken by an Intel Realsense depth camera or lidar as input. By using the Realsense2 SDK provided by intel, we made a 3d point collector which acquire data in a high-speed manner. Then we build a data reader for ply format point cloud files. It is the only one implemented in native Golang until now.

To analyze the data, we produced a program based on the RANSAC algorithm. The main idea is to describe the 3d points in a non-topological manner. This approach adjusts the planes from the 3d point cloud and then measures angular between those. Several benchmarks and tests were done. According to the result of profiling, we performed some optimizations and achieved a whole better performance. The program is also tested on a low power ARM device, to deploy a solution on Edge Computing.

Résumé

Ce rapport concerne le déroulement de travail effectué durant mon stage chez Eiffage Energie Système. Ceci est le stage de fin d'étude pour le master calcul de haute performance (CHPS).

Ce rapport présente une méthode simple et d'un coût abordable pour détecter les déformations d'une surface routière. Les données requises sont un nuage de points capturé par un appareil de profondeur. En utilisant le Realsense2 SDK d'Intel, on a fabriqué un programme qui collecte les données de nuage de points d'une vitesse élevée depuis une caméra de profondeur ou un Lidar d'Intel. On a développé un package (bibliothèque de Golang) qui lit les données de format ply. Il est le seul package existant qui sont 100% de Golang sans appel à une bibliothèque de C externe.

Le programme qui analyse les données est basé sur la méthode de RANSAC. L'idée principale est de décrire le nuage de points de manière non topologique. Avec le programme, on extrait les plans depuis le nuage de points puis on mesure leurs angles. On a réalisé plusieurs tests et profilings. On a réalisé plusieurs optimisations qui ont abouti à une meilleure performance. Le programme est aussi testé sur une plateforme d'architecture ARM pour déployer notre solution en Edge Computing.

Sommaires

1. Environnement du stage

1.1. Information de l'entreprise Eiffage

Eiffage est un grand groupe français qui se trouve principalement dans le secteur de construction et de génie civil. Il est fondé en 1993 à la suite de la fusion de plusieurs entreprises, Fougerolle et SAE (Société Auxiliaire d'Entreprises électriques et de travaux publics). Fondée en 1844, Fougerolle est aussi une entreprise des travaux publics. Eiffage est le 3ème groupe du secteur de construction en France, après les groupes Vinci et Bouygues.

Le nombre d'employés d'Eiffage s'élève à 72 500 aujourd'hui. Le chiffre d'affaires de l'année 2019 est 18.1 milliards d'euros. Les activités se répartissent dans le monde entier, majoritairement en Europe et sud Afrique.

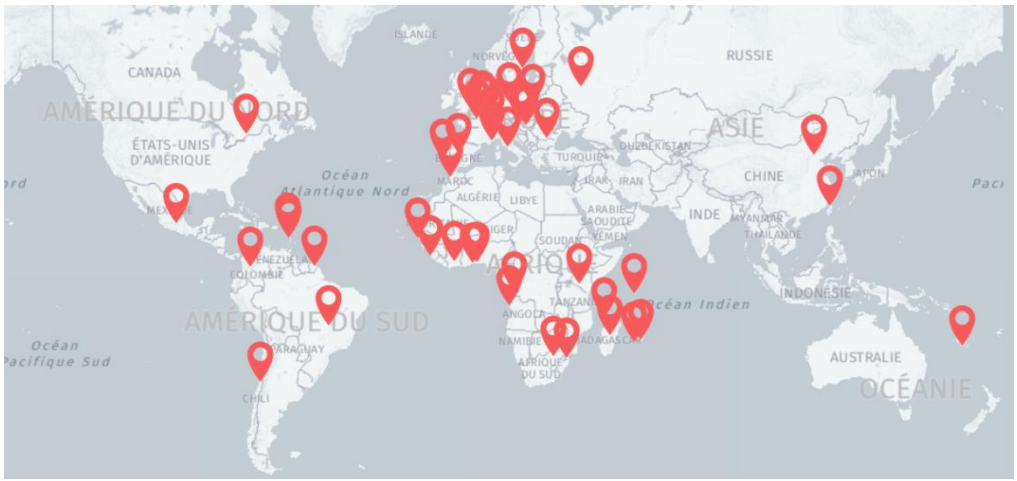


Figure 1 Carte des implantations d'Eiffage dans le monde entier

Le groupe Eiffage est composé d'un ensemble d'entreprises et chacune entre eux fonctionne de manière indépendante, décentralisée par rapport au reste du groupe. Les connexions et les communications entre les filiales permettent de regrouper plusieurs métiers et savoir-faire complémentaires entre eux. Un exemple est le département de développement durable, un département de la filiale Eiffage Construction, qui lance régulièrement des appels à projet interne parmi toutes les branches. L'association des entités permet de mieux répondre à des appels externes. Chaque entité peut aussi se positionner sur un projet indépendamment des autres entités.

Les 4 branches du groupe Eiffage concernent divers métiers, à savoir :

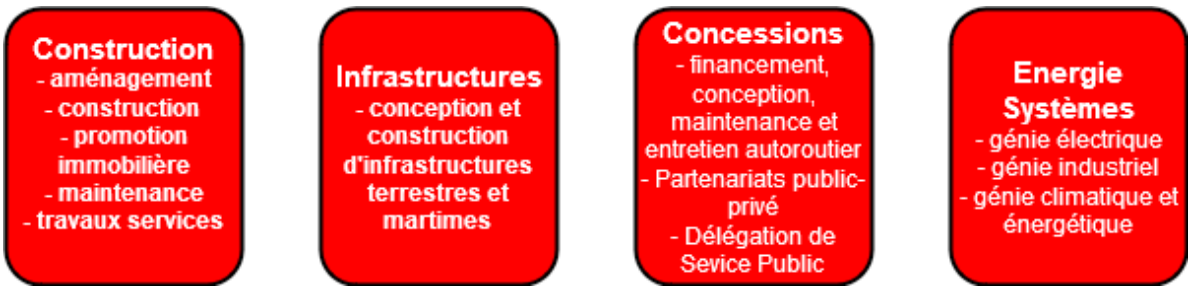


Figure 2 Activités d'Eiffage

1.2. Eiffage Energies Systèmes

Etant une branche indépendante, Eiffage Energie Systèmes fait partie du groupe Eiffage. Ses domaines d'activités sont le génie électrique, le génie industriel et le génie climatique et énergétique. La branche contient elle-même plusieurs marques, qui permettent de répondre aux demandes variées du marché.

Figure 3 Activités d'Eiffage Energies Systèmes

Eiffage Energies Systèmes est composée de 5 marques, leur but est de regrouper et représenter les savoir-faire liés à chaque secteur d'activité avec un nom, sans forcer le marché à tenir compte de la complexité de l'organisation.

1.3. Forclum Numérique

Forclum Numérique est une entreprise filiale du groupe Eiffage appartenant à la branche Eiffage Energie Systèmes. Son domaine d'activité a été historiquement orienté dans les télécoms. Aujourd'hui, elle adopte un mode de start-up et emploie une dizaine de personnes grâce à la forte indépendance des entités dans le groupe. Elle a une autonomie de réalisation et une expertise interne sur les technologies informatiques. Elles permettent à Forclum Numérique de répondre rapidement à un large éventail de projets techniques.

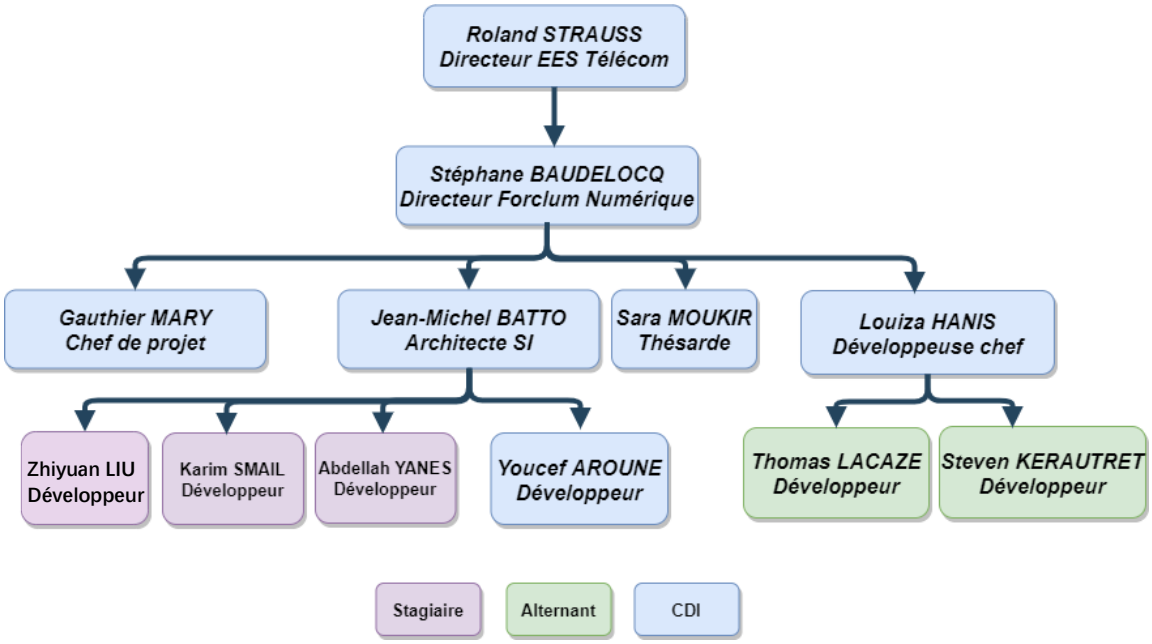


Figure 4 Organisation interne de Forclum Numérique

C'est donc l'entité juridique dans laquelle j'ai effectué mon stage.

2. Etat de l'art et enjeux de recherche

Dans cette partie, on s'intéressera au projet collaboratif HERMES (Heterogeneous Extensive Road Monitoring and Evaluation System), dans lequel mon projet s'insère. Puis on présentera les méthodes numériques adoptées et le langage de programmation choisi.

2.1. Projet HERMES

Dans le cadre de l'Appel à Projet 2016 « Route du Futur I-STREET » organisé par l'ADEME (Agence de la Transition Ecologique), le projet HERMES s'inscrit dans le programme d'innovation I-STREET. Il met à profit les possibilités offertes par les nouvelles technologies de l'information et de la communication pour mieux gérer la conception, la construction, l'exploitation et surtout l'entretien des routes.

I-STREET (Innovations systémiques au service des transitions écologiques et énergétiques dans les infrastructures routières de transport) a été lancé en septembre 2017. Il s'agit d'une route du futur intelligente, mise au point par Eiffage, en partenariat avec Total, l'IFSTTAR et la PME Olikrom. Ce projet vise à améliorer la conduite, réduire ses impacts sur l'environnement et intégrer les nouvelles technologies numériques dans les différents maillons de la chaine de valeur. Par conséquent, sont concernées ici des solutions industrielles (fabrication d'enrobés sous forme de granulés, par exemple) comme des produits et services (chaussée urbaine démontable, peintures innovantes ou intégration de capteurs dans les chaussées). (ADEME, 2017)

Contexte

Grâce aux progrès constants dans le domaine des télécoms, les systèmes de transports intelligents se concrétisent dans notre quotidien. On parle aujourd'hui de mobilité 2.0 voire 3.0 avec à la clé le développement d'une filière industrielle. La route de demain sera donc résolument intelligente et sera équipée de dispositifs permettant d'échanger des informations avec ses usagers, les véhicules qui la parcourent ou encore les différents centres qui la gèrent donnant naissance à la notion d'infrastructure digitale ou numérique (cf. Figure 5).

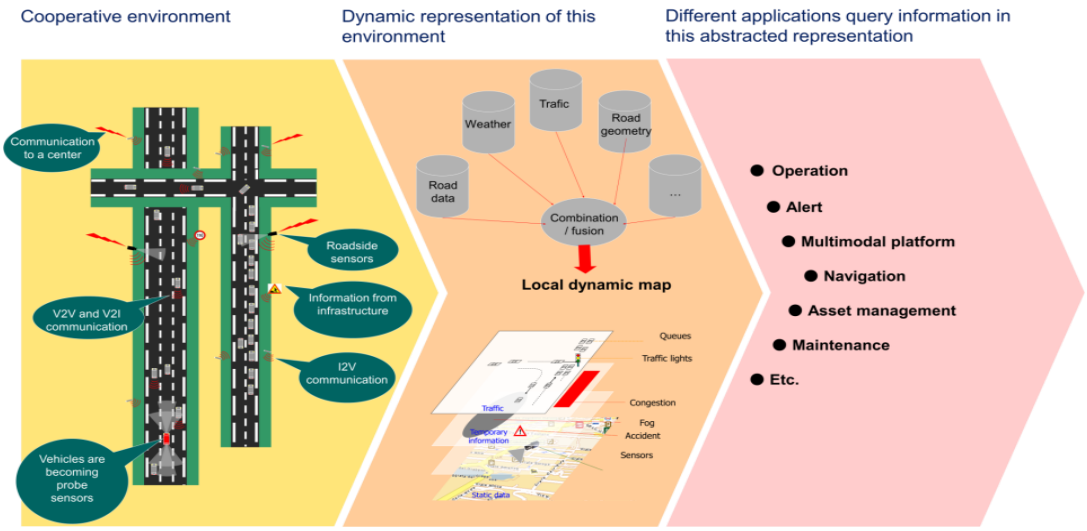


Figure 5 Schématisation de l'infrastructure numérique

Pour faciliter l'interopérabilité et le déploiement des Systèmes de Transport Intelligents, des cadres d'architectures ont été établis dans les différentes régions du globe. En France, il s'agit d'ACTIF (Aide à la conception de systèmes de transports interopérables en France). Ils partagent tous le cadre d'architecture Américain, célèbre pour son « sausage diagram » que nous présentons en Figure 6. Leur principe est de classer les systèmes en quatre grandes familles : les voyageurs, les centres, le bord de voies et les véhicules. Chaque famille de système communique avec l'autre avec le média de télécommunication le plus adapté.

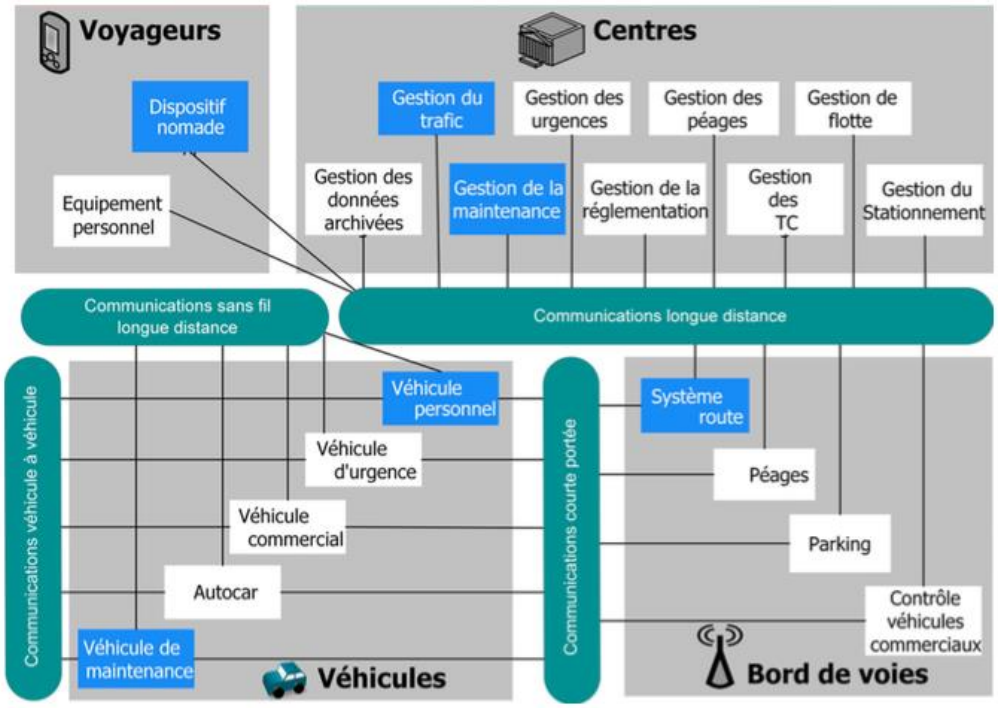


Figure 6 Architecture cadre des systèmes de transports intelligents

Selon la dynamique temporelle de ces systèmes, on parle de systèmes interactifs, de systèmes coopératifs, de sécurité active ou passive (voir Figure 6).

Enjeux de recherche

Dans le projet HERMES, on prévoit une tâche de prototypage d'UBR (Unités de Bord de Route), unités autonomes. Elles sont indispensables pour fournir l'énergie aux capteurs, donner l'accès au réseau de collecte et servir comme ressources de calculs pour le prétraitement des informations. Un UBR est un nœud de calcul Edge. Mon stage a donc été orienté par la cible des traitements et les contraintes de performance associées.

2.2. Edge Computing

Edge Computing est une façon de traiter les données collectées en proximité physique du site de production de celles-ci. L'Edge Computing est adossé au Cloud qui va permettre la mise à disposition des données.

L'un des avantages principaux de l'Edge Computing est d'éviter le téléchargement des données massives qui induit une saturation potentielle de la bande passante. L'impact de la fluctuation de la connexion, notable pour les réseaux radio, est aussi limité.

Un autre avantage est le court délai entre l'obtention des données et le résultat du calcul, ceci n'est pas notre cas mais plutôt pour les plateformes comme des véhicules ou des avions ayant un besoin des résultats de calculs immédiats. (Weisong Shi)

Edge computing pour la métrologie sur site

Il s'agit de construire un système de collecte et de traitements de données du type nuage de points sur l'edge (l'UBR), générées par une caméra lidar d'Intel. L'analyse concernée est de reconnaître les plans depuis un fichier de nuages de points fournis par la caméra lidar puis donner leurs équations standards.

On simule une route usée par une piste et un manège de fatigue qui roule dessus. L'UBR et les capteurs sont posés à proximité de la route.

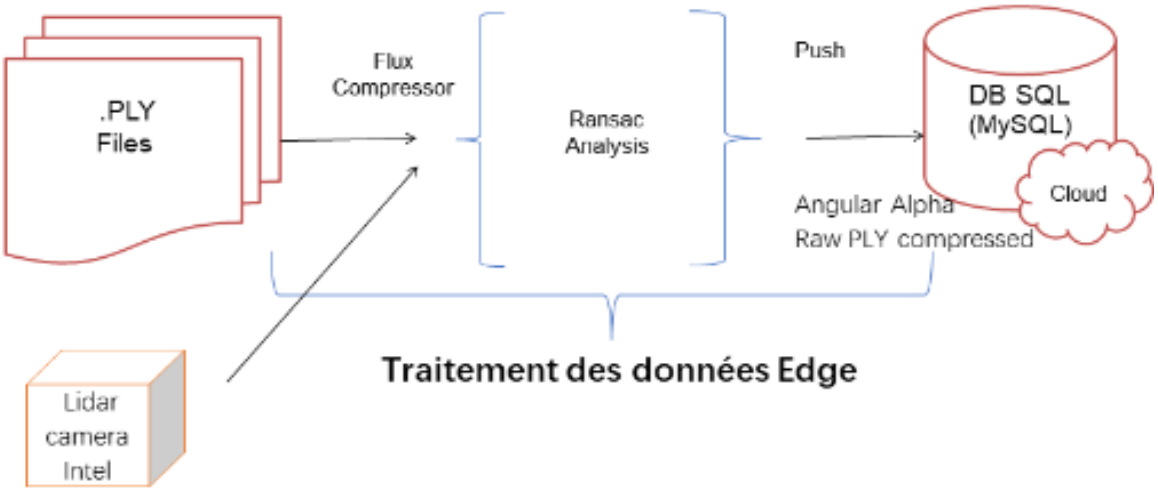


Figure 7 Traitement des données Edge : nuages de points

Edge computing pour l'analyse de données massives d'un capteur de déformation

Les données concernées dans cette partie sont générées par les capteurs de déformation placés sous la piste du manège de fatigue, encodées en format base16 dans un fichier json. Le but est de décoder les données puis les analyser pour trouver des capteurs qui présentent une panne en cherchant une anomalie des signaux reçus (il s'agit d'une approche de signature dynamique).

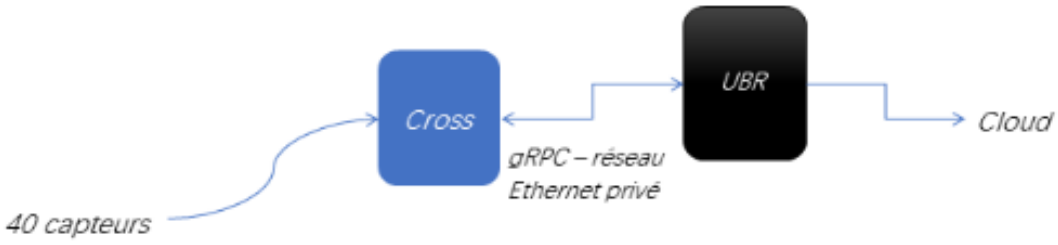


Figure 8 Traitement des données Edge : signaux de déformation

Figure 9 Manège de fatigue

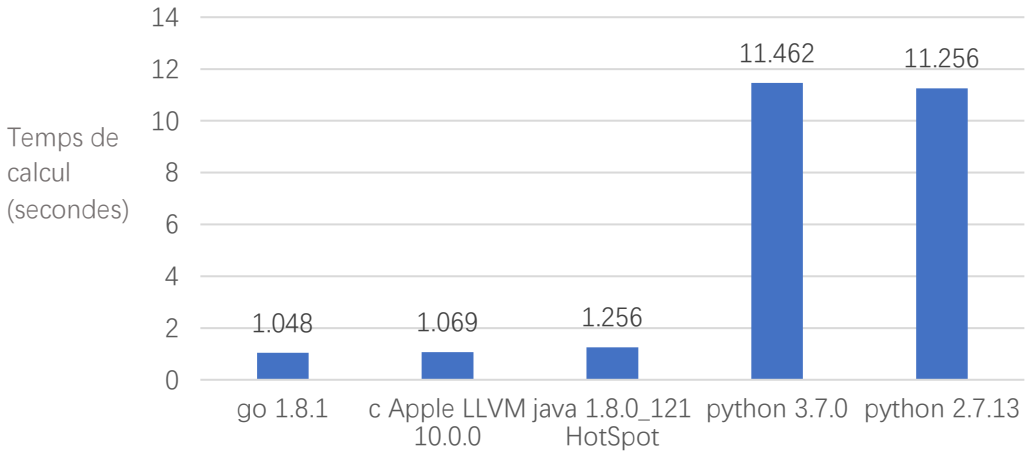
2.3. Langage principal de travail : Golang

Le langage de programmation le plus utilisé dans ce stage est le Golang, qui est aussi le cas dans le département de l'entreprise.

Caractéristiques et avantages

Né en 2009, Golang est un langage de programmation performant dans beaucoup de sens. Etant un langage compilé, il a environ la même vitesse d'exécution pour une pièce de code par rapport au C ou C++. Néanmoins, il est beaucoup plus facile à utiliser et manipuler, vu qu'il n'y a pas de génériques, pas de modèles et pas de bibliothèques d'exécution séparées. De plus, la gestion automatique de mémoire et la concurrence légère facilitent la production des codes adaptés aux processeurs multi-coeurs et au final augmente l'efficacité de développement.

La figure suivante montre un benchmark de calcul de modulo à partir de 2 jusqu'à un nombre premier très grand. (https://www.jianshu.com/p/d60eff598aa0)



Graphique 1 Comparaison de vitesse entre plusieurs langages de programmation

2.4. Méthodes Numériques utilisées

Dans cette partie on s'intéressera aux méthodes adoptées pour analyser les plans depuis un fichier de nuage de points issus de la caméra Intel. La méthode RANSAC est utilisée pour distinguer les plans différents dans le nuage. A l'intérieur de chaque itération de RANSAC, on ajuste un plan à partir d'un groupe de points. Pour cela, on a utilisé 2 moyens différents, dont la méthode des moindres carrés (Least Square) et la décomposition en valeurs singulières (Singular Value Decomposition).

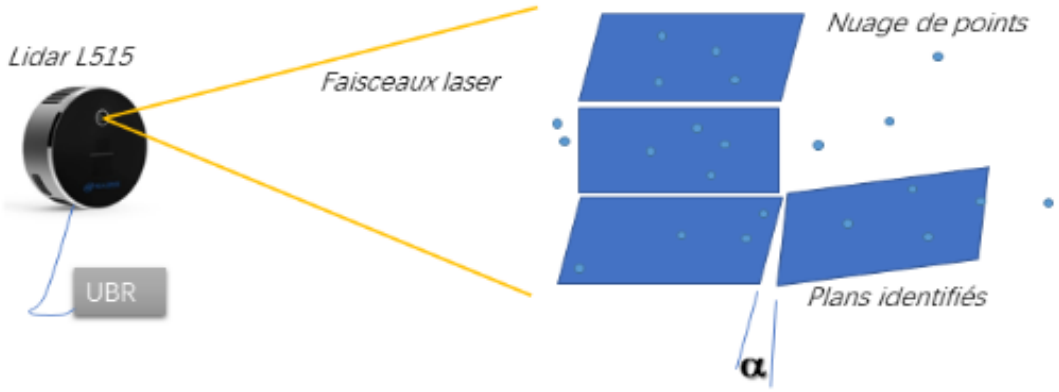


Figure 10 Flux de traitement

Une fois que tous les plans ont été identifiés, on calcule les angles entres eux pour obtenir la distribution qui décrit le nuage de points.

RANSAC

Présentée par Fischler et Bolles en 1981, RANSAC (RANdom SAmple Consensus) est une méthode itérative pour construire un modèle mathématique à partir d'un groupe des données (échantillon) en examinant la validité des données choisies aléatoirement. (Martin A. Fischler, 1981) Il est simple à comprendre, implémenter et utiliser.

On suppose que le modèle optimal contient une partie de données, appelées ***inliers***, et le reste des données, dont les ***outliers***, seront exclus pour ce modèle. Le but est de distinguer les ***inliers*** et les ***outliers***. Les méthodes d'ajustement de modèles traditionnels, comme le moindre carré, prennent le groupe de données dans sa totalité comme input. RANSAC vise à approcher l'optimalité locale en détectant puis en excluant les ***outliers***.

Cependant, la méthode présente quelques inconvénients. D'abord, la méthode est non-déterministe. Les résultats obtenus et le temps de calcul dépendent des données choisies au début de chaque itération donc sont peu reproductible. D'autre part, la méthode est fortement sensible aux paramètres. Ceux-ci ont de fortes influences aussi sur les résultats et le temps de calcul.

Dans le projet du stage, le problème est ramené à calculer les plans à partir d'un nuage de points.

Méthode des moindres carrés directe

Supposons qu'il y a un groupe de données, la méthode des moindres carrés permet de trouver un modèle qui le résume le mieux. Son principe est de minimiser la différence entre les valeurs empiriques est les valeurs théoriques. La fonction qui mesure cette valeur est aussi appelée la fonction de perte (Loss Function) dans le domaine d'apprentissage automatique.

On va présenter cette méthode en prenant son usage dans ce projet : Ajuster un plan à partir d'un groupe de points.

En méthode directe, il s'agit de construire un système linéaire surdéterminé à partir de la fonction du plan cherché $ax + by + c = z$ avec les points donnés $(x_i, y_i, z_i)_n$. On a :

$$\begin{bmatrix} x_1 & y_1 & 1 \\ \dots & \dots & \dots \\ x_n & y_n & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} z_1 \\ \dots \\ z_n \end{bmatrix} \Rightarrow \begin{bmatrix} x_1 a + y_1 b + c - z_1 \\ \dots \\ x_n a + y_n b + c - z_n \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \quad (1)$$

On écrit cette équation sous forme de produit $Ax = b$. Les lignes de la matrice A sont indépendantes l'une à l'autre car il n'existe pas 2 points exactement identiques dans le nuage obtenu. Par conséquent, on ne peut que chercher une solution approchée à l'aide de la méthode des moindres carrés.

Pour chaque ligne i de la matrice A, on prévoit une erreur ε_i générée par la solution approchée. Puis on construit la fonction de perte en sommant les carrés de cette erreur :

$$L = \sum_i^n (\varepsilon_i)^2 = \sum_i^n (x_i a + y_i b + c - z_i)^2 \quad (2)$$

Une condition nécessaire pour que la fonction atteigne son minium est que les dérivées partielles s'annulent :

$$\frac{\partial L}{\partial a} = \frac{\partial L}{\partial b} = \frac{\partial L}{\partial c} = 0 \Rightarrow \begin{cases} 2\sum_i^n x_i(x_i a + y_i b + c - z_i) = 0 \\ 2\sum_i^n y_i(x_i a + y_i b + c - z_i) = 0 \\ 2\sum_i^n x_i a + y_i b + c - z_i = 0 \end{cases} \Rightarrow \begin{cases} 2\sum_i^n x_i(x_i a + y_i b + c) = \sum_i^n x_i z_i \\ 2\sum_i^n y_i(x_i a + y_i b + c) = \sum_i^n y_i z_i \\ 2\sum_i^n c(x_i a + y_i b + c) = \sum_i^n z_i \end{cases} \quad (3)$$

On peut écrire les sommes ci-dessus sous forme de multiplication matricielle. En multipliant à gauche par une matrice inverse on trouve les résultats, les coefficients de l'équation du plan.

$$A^T Ax = A^T b \Rightarrow x = (A^T A)^{-1} A^T b \quad (4)$$

La méthode directe est simple à comprendre et à implémenter. L'inversion de la matrice a une complexité en $O(n^3)$, les calculs sont tous vectoriels donc facile à optimiser pour le compilateur Golang. De plus, la décomposition LU basée sur la méthode gaussienne est possible à paralléliser. On peut donc prévoir que la performance sera optimale.

Estimation des normes : Décomposition SVD

Dans cette partie, on essaie d'ajuster un plan par une estimation de la norme commune des n points donnés $(x_i, y_i, z_i)_n$. (Hoppe, 1992)

On suppose que l'équation du plan cherché est de la forme $ax + by + cz = 0$. Le principe est de ramener le point d'origine au barycentre des points donnés par $\bar{x} = \sum_i^n x_i$. On obtient $\bar{y}, \bar{z}$ de la même manière. La nouvelle équation du plan qui passe par le barycentre est :

$$a(x - \bar{x}) + b(y - \bar{y}) + c(z - \bar{z}) = 0 \tag{5}$$

On construit la fonction de perte pour la méthode des moindres carrés :

$$L = \sum_i^n (\varepsilon_i)^2 = \sum_i^n (x_i a + y_i b + z_i c)^2 \tag{6}$$

On obtient la matrice à décomposer par l'écriture des dérivées sous la forme de $Ax = 0$

$$\frac{\partial L}{\partial a} = \frac{\partial L}{\partial b} = \frac{\partial L}{\partial c} = 0 \Rightarrow \begin{cases} 2\sum_i^n x_i(x_i a + y_i b + z_i c) = 0 \\ 2\sum_i^n y_i(x_i a + y_i b + z_i c) = 0 \\ 2\sum_i^n z_i(x_i a + y_i b + z_i c) = 0 \end{cases} \Rightarrow \begin{bmatrix} \sum {x_i}^2 & \sum x_i y_i & \sum x_i z_i \\ \sum x_i y_i & \sum {y_i}^2 & \sum y_i z_i \\ \sum x_i z_i & \sum y_i z_i & \sum {z_i}^2 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} = 0 \tag{7}$$

On réalise la décomposition en valeurs singulières pour la matrice A : $A = UDV^T$, dont U et V sont matrices orthonormales. La solution du système est le vecteur propre correspondant à la valeur minimale dans la matrice D, qui se trouve à la troisième colonne de la matrice V.

3. Système de capture de données avec les appareils Intel

La caméra de profondeur L515 et le LiDAR D455 fabriqués par l'Intel sont les deux appareils utilisés pour récupérer les données et pour alimenter notre programme d'analyse. Les données enregistrées sont au format standard .ply, c'est-à-dire celui de nuages de points en 3D.

3.1. Contexte et objectif

Les appareils d'Intel répondent bien à la demande de créer un système de surveillance de déformation de route, sous la contrainte de minimiser le coût. Leurs prix sont assez limités, par rapport aux autres laser scanners terrestres (€400 contre €18000+). Ils sont compacts, donc faciles à installer dans divers environnements comme un véhicule ou un drone.

Figure 11 Appareil de profondeur Intel L515 et D455

Un autre avantage de ces minis appareils est la vitesse d'acquisition des points par seconde. Cela peut aller jusqu'à 600 000, ce qui dépasse déjà certains modèles très hauts de gamme. Les inconvenients de cette solution à faible coût sont aussi évidents. L'un d' eux est la distance maximale de détection à environ 9 mètres. Pour le LiDAR, le résultat risque de se dégrader, si l'environnement est très lumineux ou la réflectivité de l'objet très faible (i.e. un objet noir).

Parmi ces deux appareils Intel, les technologies adoptées sont différentes. Le L515 est un LiDAR basé sur la technique TOF (time of flight), qui mesure les distances par différence entre l'instant d'émission d'un faisceau lumineux et celui de la réception du reflet. L'appareil D455 est basé sur la lumière structurée (structured light), il calcule la distance selon les caractéristiques structurales des faisceaux lumineux réfectés par les objets à différentes distances.

Il est possible que cette hétérogénéité technologique nécessite une préoccupation particulière dans les futurs usages (i.e. méthode d'installation correspondante à la réflectivité de la surface de route).

L'objectif de notre approche est d'implémenter un programme qui traite les nuages de points depuis un appareil d'une façon continue, efficace et robuste.

3.2. Implémentation

L'implémentation de ce programme est basée sur Intel® RealSense™ SDK 2.0, un kit de développement pour établir les flux d'informations entre la caméra et un système d'opération. Plusieurs langages sont supportés comme C++, java et python, parmi lesquels on choisit le C++ dû à sa performance élevée par rapport aux autres langages interprétés.

On a commencé le travail depuis un programme d'exemple, **rs-save-to-disk.cpp**[1], fourni dans le SDK par RealSense. Ce programme enregistre 3 images et 1 fichier de points de nuage. Les 3 images sont : une photo réelle, une photo infrarouge et une photo RGB qui exprime les profondeurs par différentes couleurs. Les métadonnées correspondantes à chaque frame sont enregistrées dans un fichier type texte.

[1] https://github.com/IntelRealSense/librealsense/tree/master/examples/save-to-disk

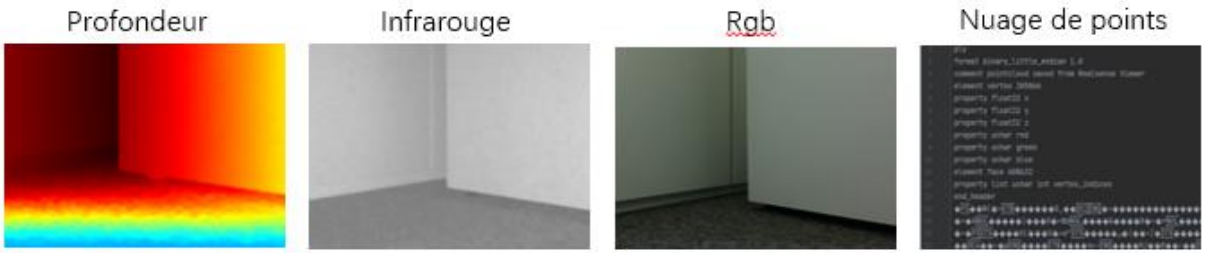


Figure 12 Données récupérées à un instant par le LiDAR Intel L515

L'entrée de paramètres se fait à l'aide de la bibliothèque ***boost/program_options***, qui donne accès à une gestion de paramètres de façon structurée, systématique et efficace.

Les fonctions principales sont ci-dessous :

- ***void SaveOneFrame(const BoostConfig& config)*** :

  La fonction de l'entrée qui sert à créer l'horodatage, les flux de données et le fichier contenant tous les paramètres entrés. Le nombre d'enregistrements voulu est contrôlé par un paramètre, la fonction ***SaveInfiniteFrames*** sera appelée si un nombre est détecté. Dans le cas contraire, ***SaveKernel*** sera appelée pour itérativement récupérer les données.

- ***void SaveInfiniteFrames(const rs2::colorizer& color_map, const rs2::pipeline& pipe, int interv, int ifPly, int ifDepth, int ifImages, int ifInfr, int ifColor, struct timeval t1, const std::string& direction)*** :

  Cette fonction contient une boucle infinie qui appelle ***SaveKernel*** afin de récupérer les données dans un mode dit infini, jusqu'à un arrêt manuel.

- ***void SaveKernel(const rs2::colorizer& color_map, const rs2::pipeline& pipe, int ifPly, int ifDepth, int ifImages, int ifInfr, int ifColor, const std::string& direction)*** :

  Elle reçoit les frames envoyés par la caméra, génère les nuages de points puis les écrits dans le disque. Si les images sont demandées, elles seront alignées avec les nuages de points puis exportées.

Le problème principal est la vitesse d'acquisition sur un UBR de l'architecture ARM. Sa caractéristique est une consommation énergétique très limitée, mais aussi pour la performance. Selon le premier benchmark, le temps d'acquisition s'élève à 3 s pour chaque groupe de donnée sur une plateforme x86. D'autres expériences prévoient un ralentissement jusqu'à 40 fois sur un Raspberry Pi[2]. Cette vitesse est trop lente. On cherche à réduire ce temps, notre approche est décrite dans les paragraphes suivants.

[2] https://github.com/IntelRealSense/librealsense/tree/master/examples/save-to-disk

3.3. Optimisation

Le premier benchmark avant toute optimisation est présenté dans le paragraphe suivant avec les autres benchmarks.

Les optimisations dans cette partie concernent principalement la procédure et la fonctionnalité de programme car il s'agit plutôt d'un programme de pilotage d'un hardware. Ils sont les optimisations de niveau application et système.

Désactivation des flux de données non nécessaires

Les données exigées par l'étape suivante du projet sont les nuages de points. Par conséquent, on pourra chercher à augmenter la vitesse d'acquisition des données sans enregistrer les images de format png.

Cette opération est possible car chaque capteur établit son propre flux de données par la fonction ***rs2::config.enable_stream()***. En ajoutant cette fonction par exemple, on a ***RS2_STREAM_DEPTH*** où sont transmis les nuages de points ; ***RS2_STREAM_INFRARED*** où est transmise l'image d'infrarouge et ***RS2_STREAM_COLOR*** où est transmise l'image prise par une caméra normale. Ces flux de données peuvent être désactivés manuellement.

Cela permet de réduire le temps d'acquisition d'une façon remarquable. Tout d'abord, le temps consacré à l'autocalibration de l'exposition et de la balance des blancs pour la caméra sera supprimé. De plus, la production de l'image profondeur nécessite d'aligner un nuage de points à une image RGB, qui consomme aussi de la capacité de calcul.

Baisse de la résolution de nuage de points

Les nuages de points enregistrés sous le mode par défaut sont tous de la résolution de 640*480. Cela peut être modifié par la valeur de résolution souhaitée dans la fonction qui établit le flux de données. Les résolutions supportées par le capteur de résolution sont comme suit :

	LOW		MID		HIGH	
D455	424*240	480*270	640*360	640*480	848*480	1280*720
L515	320*240		640*480		1024*768	

Tableau 1 Résolutions supportées

On priorise l'usage de la résolution la plus basse pour chercher une meilleure performance. Cela divise le nombre de points à traiter par 4, par rapport à la résolution défaut. Cela va aussi diminuer le temps de l'analyse RANSAC dans la prochaine étape.

Modification de la structure de données

Cette modification concerne la structure du fichier de nuage de points. La structure détaillée sera présentée

dans le chapitre suivant.

Dans un fichier généré par défaut, un point est représenté par ses coordonnées en 3 dimensions, plus sa couleur du pixel correspondant en RGB. Cette dernière information n'est pas nécessaire pour l'algorithme d'analyse, mais il augmente la quantité de données traitées. On essaie d'éviter de prendre en compte cette partie redondante.

La modification est réalisée par l'ajout d'une fonction `void points::export_to_ply_notexture(const std::string& fname)` qui omet les informations concernant les informations texturales, y compris l'allocation de mémoire, l'écriture du header et les entiers qui représentent les valeurs de RGB.

Cette opération équivaut à ne pas ajouter la deuxième de la fonction par défaut : `void points::export_to_ply(const std::string& fname, const frame_holder& texture)`.

3.4. Benchmark avant et après optimisations

Optimisation du programme

Le benchmark de ce programme consiste à acquérir 100 nuages de points successivement, sans attente entre deux enregistrements. Le benchmark est réalisé à la fois sur une plateforme X86 et une autre de type UBR dont le processeur est de structure ARM. Seuls les fichiers de nuages de points sont enregistrés. Le compilateur est (GCC) g++ 9.3.0

- **Plateforme X86 :**

  Système : Ubuntu 20.04.2 LTS 5.4.0-70-generic ; gcc version 9.3.0

  Hardware : Intel(R) Core(TM) i5-1035G7 CPU @ 1.20GHz ; NVMe device Micron CT1000P1SSD8

- **Plateforme ARM :**

  Système : Raspbian GNU/Linux 10 (buster) 5.10.17-v7l+ ; gcc version 9.3.0

  Hardware : ARMv7 rev3 (v7l) ; SSD SATA 500gb Samsung 860 Evo

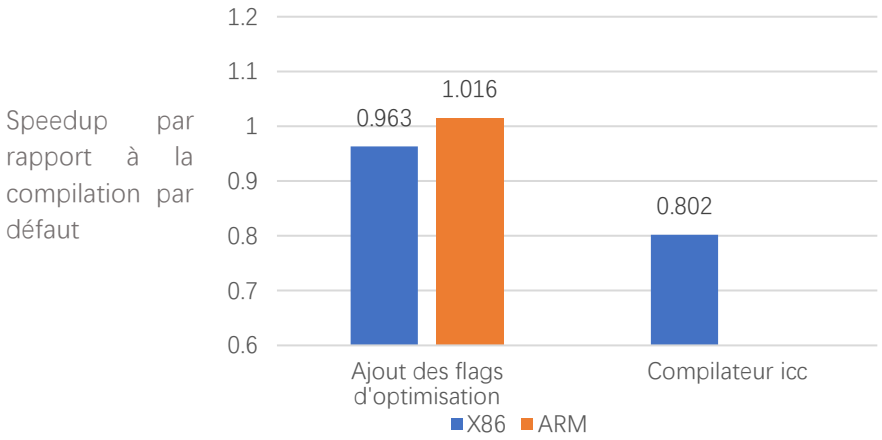
	Avant optimisation	Après optimisation	Plateforme ARM (Pour information)
Bande passante	5.01 MB/s	25.11 MB/s	6.10 MB/s
Temps par fichier	1.87 s	0.11 s	0.33 s

Tableau 2 Benchmark du collecteur de données

On constate une amélioration énorme du temps d'acquisition des données. Le speed up s'élève à 5.01 si on regarde la bande passante avant et après l'optimisation. Il permet de récupérer un nuage de points en 0.11 seconde, ce qui est largement suffisant par rapport la demande du projet.

Compilateur

- Dans cette partie on cherche l'influence des flags de compilation pour les différents compilateurs - selon la plateforme d'exécution. On ajoute d'abord des flags d'optimisation pour la plateforme X86 et ARM, puis on essaie le compilateur icc pour X86. Vu que la cross-compilation vers ARM n'est pas supportée, le résultat est absent pour icc.
- Flags d'optimisation ajoutés sur X86 : ***-g -O3 -march=icelake-client -funroll-loops -Wall -Wextra***
- Flags d'optimisation ajoutés sur ARM : ***-mcpu=cortex-a72 -mtune=cortex-a72 -mfpu=neon-fp-armv8 -g -O3***



Graphique 2 Speedup par rapport à la compilation par défaut

On constate que l'influence de ces optimisations est très limitée. Les deux modifications sur la plateforme X86 ont eu même des effets de ralentissements. Cela peut vient du fait qu'il n'y a pas de calcul massif dans un programme de pilotage comme celui que l'on utilise.

3.5. Mise en production : un programme robuste

Le but de cette partie est de rendre le programme plus robuste dans un mode de production, face aux divers imprévus ou accidents. Le programme est à ajouter dans ***/etc/rc.local***, pour un démarrage automatique (en Linux). On a implémenté un programme en language **bash**, ses fonctions principales sont les suivantes :

- **Surveillant** : Surveiller les processus dans le background par la commande ***grep*** et relancer le programme si aucun processus n'est détecté.

  Il s'agit de reprendre la collection des données après un incident (i.e. coupure de courant ou un crash du système et etc.), sans aucune intervention. Cette fonction permet de reprendre la mission non terminée automatiquement, qui assure la continuité des données, surtout pour le mode dit permanent (infini).

- **Chargeur** : Charger les paramètres automatiquement par un fichier extérieur.

Cette fonction assure que les données générées dans une mission reprise se conforment aux spécifiques requises initialement. Pour une mission dont le nombre de données est fixé, il sera recalculé lors de la reprise selon le nombre de fichiers déjà enregistrés. C'est-à-dire que le programme mémorise ce nombre et les paramètres donnés au début du programme.

Ce programme est testé de manière suivante :

- Le nombre total de fichiers demandés : 100
- Rupture brutale du programme et reprise (par exemple) : Après 26 fichiers sauvegardés
- Reprise (par exemple) : Continuer à sauvegarder les 74 fichiers
- Types de rupture testé : Ctrl -C / Eteinte brutale d'alimentation

Le test est répété plusieurs fois. On constate qu'à chaque fois le programme reprend la collection des données en mémorisant les paramètres correctement.

3.6. Conclusion

Dans cette partie on a développé un programme qui utilise les caméras de profondeur fabriquées par Intel pour récupérer les points de nuage. Le programme est facile à utiliser. Les contenus et les méthodes de l'enregistrement sont personnalisables grâce aux paramètres. Le programme est maintenant accessible sur mon Github.

4. Analyse des plans : méthode RANSAC

L'objectif de ce chapitre est de se servir des nuages de points récupérées depuis les appareils. L'idée générale est d'extraire les équations de tous les plans dans le nuage de points. Ensuite, on étudie la distribution des valeurs d'angles formés par ces plans. Le but est de chercher une méthode pour interpréter la condition de la surface représentée par ce nuage de points.

4.1. Implémentation de package de lecture des données

La première étape consiste à construire un lecteur pour les nuages de points de format ***ply***. Pour éviter les travaux redondants, on a cherché puis trouvé un package en Golang qui s'appelle ***plyfile***. Ce package présente des problèmes. Tout d'abord, la compilation échoue et on constate que le package n'a eu aucun renouvellement ni maintenance pendant une très longue période (5 ans). D'autre part, ses fonctions principales sont toutes implémentées dans une bibliothèque en C, qui est reliée via ***cgo***. Cela apporte plusieurs influences néfastes (Cheney, 2016) :

- **Cross compilation :** Vu que l'on envisage un usage des unités de calcul de l'architecture ARM, la cross compilation devient nécessaire. Elle ne supporte pas officiellement le cgo, qui rend le test et le déploiement plus compliqués.
- **Outils de profiling :** Le ***pprof*** qui analyse l'usage de la mémoire et du processeur n'est pas supporté.
- **Performance :** Le temps pour la compilation sera plus long et les codes binaires compilés seront plus volumineux. Les appels des fonctions et passages des données coûtent cher. Différents mécanismes de recyclage de mémoire pourront apporter un risque de fuite de mémoire et etc.

Par conséquent, on décide d'implémenter un package d'origine ***Golang*** qui n'utilise pas de bibliothèque externe de C. On est basé sur l'ancienne structure de ***plyfile***. Comme montré ci-dessous, un nuage de points de format ***ply*** est composé de deux partie. Le header, qui contient les informations générales du fichier y compris la méthode de codage (binaire ou ASCII), le nombre de points et les types de données (entier, flottant, chaîne de caractères et etc.). Tous les points sont regroupés dans la dernière ligne de façon l'une après l'autre.

```
ply
format binary_little_endian 1.0
comment pointcloud saved from Realsense Viewer
element vertex 95760
property float32 x
property float32 y
property float32 z
element face 189776
property list uchar int vertex_indices
end_header
```

Figure 13 Exemple d'un nuage de points contenant 95760 points et 189776 surfaces dont chacune est composée de 3 points

L'entrée du programme est la fonction ***ReadPLY*** qui prend le nom du fichier à lire comme le paramètre et

retourne deux slices (un slice pour ***Golang*** est comme un array pour ***C***).

Un nuage de points de format ***ply*** est traité selon les étapes suivantes :

1. **Parsing du header :** Le header d'un nuage codé en ASCII, encadré par les signes ***ply*** et ***end_header***. Il regroupe toutes les indications nécessaires pour comprendre la composition du fichier et le type des données, y compris les informations complémentaires comme des commentaires. La fonction ***PlyGetElementDescription*** analyse le header, extrait les informations puis les retourne dans la structure ***PlyFile***.

2. **Lecture des éléments :** La lecture des données se fait itérativement vers un slice de conteneur. On cherche les données pour chaque élément pour aller dans le fichier et le récupérer. A la fin du programme, on retourne les slices contenant les informations désirées. Prenons l'exemple du nuage de points ci-dessus, on va obtenir deux slices. L'un contient les vertex, c'est-à-dire les cordonnées de chaque point, et l'autre contient les surfaces dont chacune est composée par 3 points de proches voisins.

3. **Localisation des données :** On a besoin de localiser où un élément se termine dans le nuage de points pour pouvoir entamer le prochain. C'est-à-dire pour commencer lire les informations des surfaces, il va falloir localiser la fin des informations de sommets, qui sont écrites avant les surfaces dans le nuage de points. Il s'agit d'identifier le nombre de bits selon le type de donnée indiqué dans le header. Dans l'exemple, on remarque que le type vertex occupe 32*3*286194 bits, comme un point contient 3 coordonnées de flottant de 32 bits.

Une fois que toutes les données sont lues, le slice de cordonnées des points sera transmis à l'étape suivante.

4.2. Recherche préliminaire : la complexité

Pour étudier la complexité du RANSAC, on suppose que le nombre total d'échantillons contenus par le système (ici le nombre de points) est n. Le nombre de modèles du système (ici les plans dans le nuage de points) est m. La complexité idéale dans le meilleur cas est de $O(mn)$ car pour chaque modèle on calcule la fonction de perte au moins n fois. Vu que le nombre de modèle est souvent un entier fixé qui est inférieur à n, cette complexité peut s'écrire comme $O(n)$.

Néanmoins, le RANSAC n'est pas un algorithme déterministe. La probabilité de choisir les **inliers** affecte le temps de calcul. Plus il est facile de trouver les **inliers** d'un modèle, moins on risque de faire des calculs inutiles.

Un autre facteur est la condition d'arrêt. Le nombre de points qui ne sont pas classés comme **inliers** pour aucun plan indique la pertinence de la représentation de auquel niveau le nuage de points est expliqué par le modèle des plans. Le nombre de plans idéal est un inconnu pour l'algorithme. On décide de prendre ce nombre de plans cible comme le paramètre qui contrôle l'arrêt du programme. Si ce paramètre est trop bas, on risque d'omettre les petits plans. Dans le cas contraire, on risque de tomber dans une itération infinie car l'algorithme n'arrive pas à sortir un plan dont il y a assez d'**inliers** parmi les points restants.

4.3. Implémentation

Les étapes principales de l'algorithme sont données ci-dessous. On considère un point appartenant à un plan comme son **inlier**. Sinon, il sera considéré comme un **outlier**. Les étapes principales sont ci-dessous :

1. **Sélection d'un batch :** On choisit un sous-ensemble parmi tous les points aléatoirement. Cette opération va réduire la quantité des points sur lesquels on applique la procédure de RANSAC. Ainsi, les tests de distance ne se répètent que sur un petit groupe de points au lieu de tous les points.

2. **Processus de RANSAC :** On applique la méthode de RANSAC sur le batch choisi. C'est-à-dire on ajuste un plan avec 3 points aléatoires, puis on étudie les distances entre chaque point et ce plan ajusté. Si la distance est inférieure à un seuil donné, le point correspondant sera considéré comme un **inlier**, dans le cas contraire, un **outlier**. Si le nombre d'**inliers** dépasse un seuil donné, on passe à l'étape suivante et sinon on recommence cette étape.

3. **Réajustement :** On recalcule les distances pour choisir les **inliers** parmi tous les points. Une fois que l'on aura obtenu les **inliers**, on réajuste un nouveau plan et on passe à l'étape suivante.

4. **Contrôle de redondance :** Cette étape est pour contrôler si on obtient deux plans différents qui représentent la même réalité. On calcule d'abord l'angle entre ces deux plans. Ils seront considérés comme parallèles si cet angle est inférieur à un seuil donné. Pour les deux plans parallèles, on calcule la distance entre eux. Si cette distance est encore inférieure à un seuil donné, on fusionne ces 2 plans : on reprend les points contributeurs et on ajuste un nouveau plan qui se substitue aux 2 précédents.

5. **Condition d'arrêt :** En général, un algorithme itératif s'arrête si le résultat obtenu est assez satisfaisant ou les itérations sont déclarées infinies. Dans notre cas, ces deux conditions sont appréciées avec un maximum du nombre d'itérations et un seuil du nombre de points qui n'appartient à aucun plan. L'algorithme s'arrête puis retourne le slice qui contient les équations des plans si l'une des deux conditions est vérifiée, sinon on passe à la prochaine itération à partir de l'étape 1.

Les pseudo codes sont comme suit :

Algorithme : Analyse des plans via RANSAC

```
1  func planeRansac (VertexList, BatchSize, LimitDistance, LimitPercentage, LimitAngle, LimitIterations,
2  LimitPoints)
3      PoolOfPlane := []
4      IfContinue := true
5      IterCount := 0
6      while IfContinue :
7          Batch := chooseBatch (VertexList, Batchsize)
8          Plane := choosePlane (Batch)
9          if validatePlane (Plane, Batch, LimitDistance, LimitPercentage) != true
10             continue
11         Inliers := computeInliers (Batch, VertexList)
```

```
12      NewPlane := estimatePlane (Inliers)
13      if redundantPlane (NewPlane) := true
14          AnotherPlane := reEstimate (NewPlane, ···)
15          NewPlane := AnotherPlane
16      PoolOfPlane = append (PoolOfPlane, NewPlane)
17      IterCount++
18      if exitCondition (IterCount, LimitIterations, LimitDistance, LimitPercentage, LimitAngle, LimitIteration,
19      LimitPoints) == true
20          IfContinue = false
21          return PoolOfPlane
```

Code 1 Analyse des plans via RANSAC

Les usages des paramètres dans le pseudocode sont précisés ci-dessous :

- **VertexList :** La variable type le slice des points qui contient le nuage des points
- **BatchSize :** La taille de batch dans lequel on réalise les itérations de RANSAC. Un petit batch réduit la quantité de calcul, mais est plus vulnérable aux bruits locaux, générés à cause de la précision limitée des caméras utilisées.
- **LimitDistance :** La distance maximale tolérée entre un point et un plan pour que le point soit considéré comme un **inlier** de ce plan. Ce paramètre est utilisé pour qualifier les **inliers** d'un plan. S'il est très grand, la qualité des plans ajustés peut baisser. Dans le cas contraire, on risque d'augmenter le nombre d'itérations car il est plus difficile de trouver assez d'**inliers**.

  Ce paramètre est aussi utilisé pour calculer la distance entre deux plans considérés a parallèles.
- **LimitPercentage :** La proportion minimale d'**inliers** nécessaires parmi un batch pour qu'un plan soit considéré comme validé. C'est-à-dire si le nombre de points considérés comme **inliers** d'un plan est au-dessus de ce seuil, le plan sera validé. Avec le paramètre précédent, les deux forment un arbitrage entre la qualité des plans et le temps de calcul.
- **LimitAngle :** L'angle maximal toléré entre deux plans pour qu'ils soient considérés parallèles. Dans ce cas, si la distance est inférieure au paramètre ci-dessus, on fusionnera les deux plans.
- **LimitIterations :** Le nombre maximal des itérations avant la fin du programme. Le programme s'arrête une fois que ce seuil est atteint.
- **LimitPoints :** Le nombre maximal des points qui n'appartiennent à aucun plan, pour continuer les itérations. S'il est inférieur au seuil donné, cela veut dire que le système est bien analysé et on pourra terminer le programme.

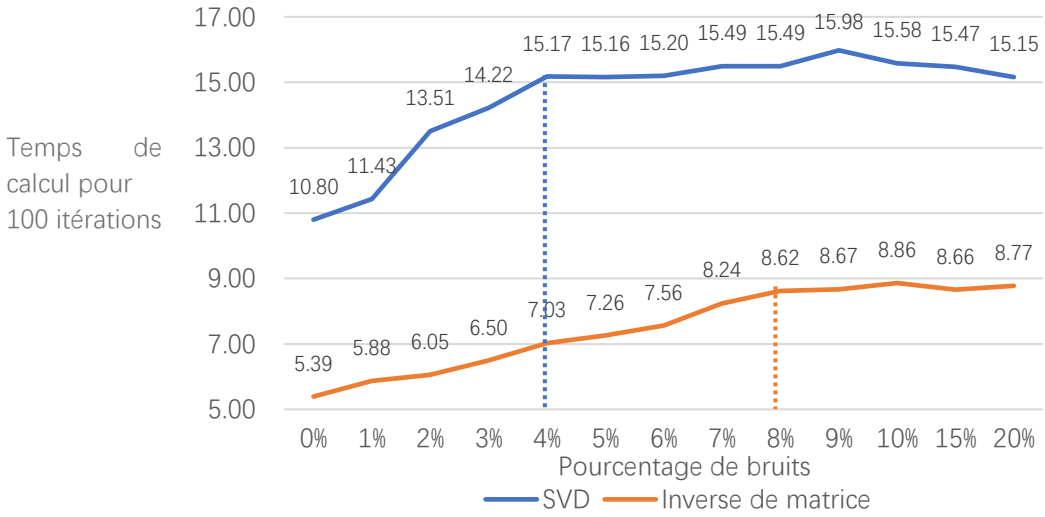
4.4. Comparaison des méthodes numériques : la robustesse

Dans ce chapitre on s'intéresse à la performance des deux méthodes d'ajustement des plans présentées dans le paragraphe 2.4. Elles sont la méthode des moindres carrés (méthode 1) et la décomposition de SVD (méthode 2). On voudrait comparer la performance de ces deux méthodes et leur robustesse face à un bruit dans les nuages de points.

Pour cela, le programme principal est testé sur le même jeu de données (un nuage composé d'environ 95000 points appartenant au même plan) en utilisant à la fois deux méthodes différentes. L'ajout de bruits est fait par une fonction qui prend un pourcentage et un intervalle comme les paramètres. Le pourcentage désigne le nombre de points qui seront ajouté d'un bruit et l'intervalle contrôle le minimum et le maximum du bruit aléatoire.

- Intervalle qui encadre la valeur d'un bruit : $[0.05, 0.1[$
- Pourcentage de sommets ajoutés d'un bruit : de 0% à 20%

Ralentissement de calcul



Graphique 3 Temps de calcul des deux méthodes avec l'introduction des bruits

Le graphique montre que le temps de calcul augmente lorsque l'on augmente le nombre de points concerné par l'ajout de bruits. Le temps de calcul arrête d'augmenter à un moment donné (à partir de 4% pour la méthode SVD et 8% pour l'inversement de matrice), puis il reste stable à ce niveau.

Le taux de corrélation s'élève à 0.864, ce qui signifie que les influences sur le temps de calcul sont fortement corrélées.

	SVD	Inverse de matrice
Temps de calcul à 0% bruits	10.80	5.39
Temps de calcul à 10% bruits	15.58	8.86
Ralentissement	1.44	1.64

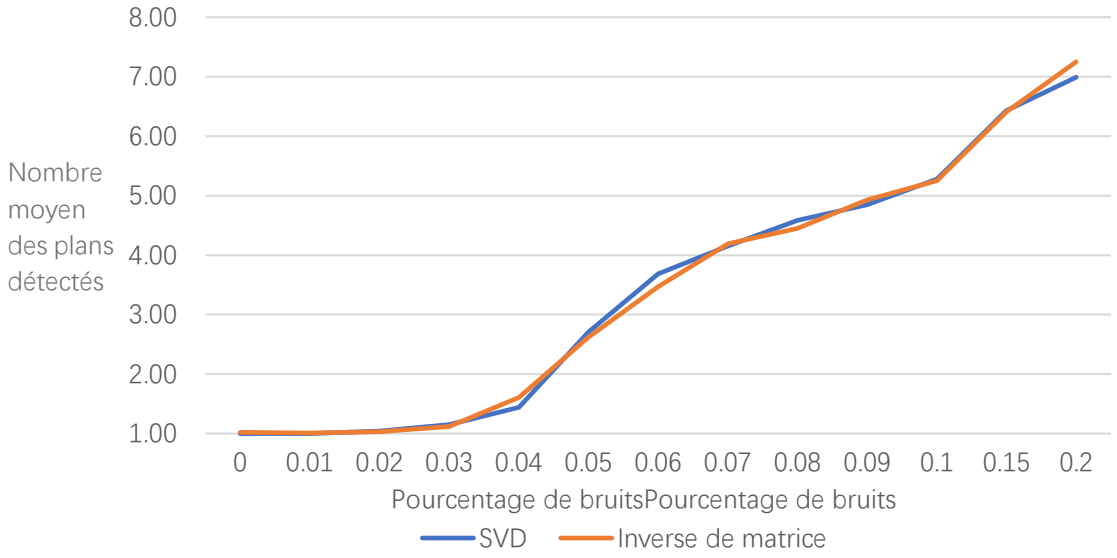
Tableau 3 Comparaison de deux méthodes

La performance de la méthode d'inverse de matrice est meilleure. Elle est 100.3% plus rapide que la méthode de SVD sans bruit, et 75.85% plus rapide que la SVD à 10% de bruits. Le ralentissement proportionnel face à

une introduction de bruits est aussi plus élevé.

Erreurs générées par les bruits

En gardant le pourcentage des bruits comme l'invariant et faisant varier leur intervalle encadrant, on constate que l'influence sur le temps de calcul est très limitée :



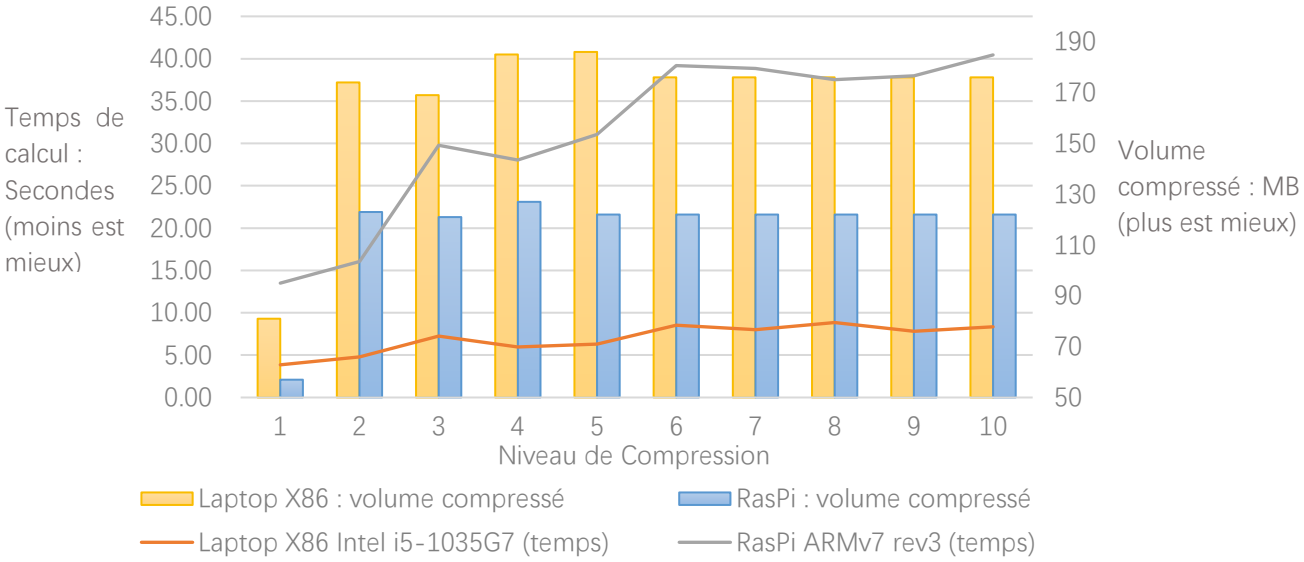
Graphique 4 Variation d'Erreur de calcul de deux méthodes

On constate que les résultats d'analyse peuvent être erronés si la quantité de bruit continue à augmenter. Le graphique montre que l'erreur commence à apparaître à partir de 3% de bruits, si on ne touche pas les paramètres du RANSAC. Le taux de corrélation est à 1.0, qui signifie que les résultats de ces deux groupes sont très fortement corrélés.

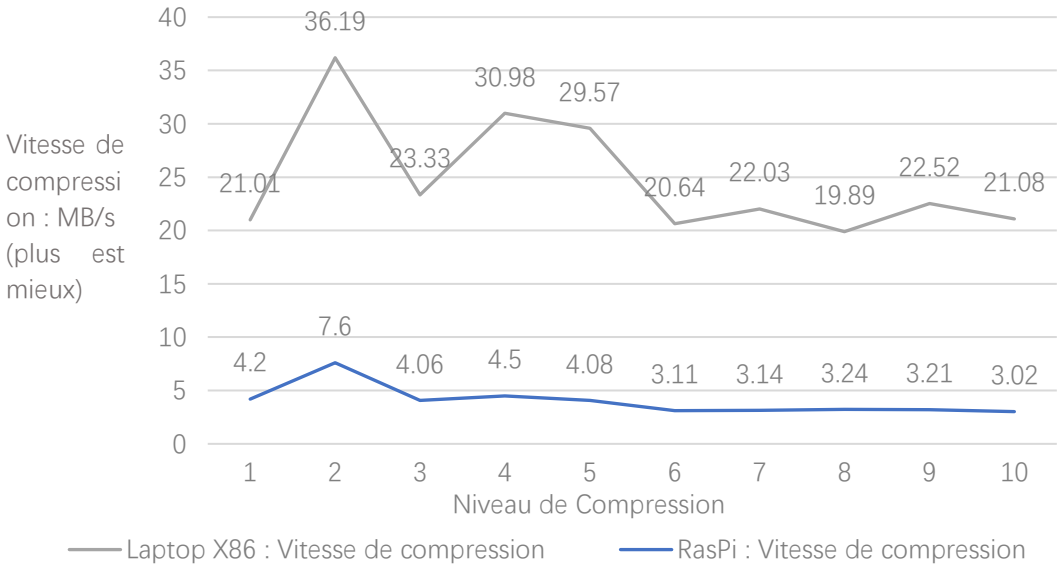
4.5. Compression

A la suite de l'algorithme d'analyse, on veut compresser les nuages de points afin de les envoyer vers le Cloud, pour avoir un backup des données originales. On a réalisé cette fonction par le package ***gzip***. Ce package permet d'archiver puis compresser des fichiers sous le format ***.tar.gz.*** La fonction prend le niveau de compression comme le paramètre. Plus ce niveau est élevé, plus le temps de compression sera prolongé. La taille après compression sera aussi moins volumineuse théoriquement.

Néanmoins, ce n'est pas le cas en réalité. On a trouvé le paramètre le plus approprié en mesurant les temps pris par chaque niveau de compression en X86 et ARM. La configuration de hardware est identique que le chapitre 3.4. On obtient les résultats suivants :



Graphique 5 Relation entre le niveau de compression et le résultat



Graphique 6 La vitesse de compression

On constate que l'augmentation du niveau de compression conduit à une augmentation du temps de calcul et du volume compressé avant le niveau 5. A partir du niveau 6, le volume compressé ne varie plus et le temps d'exécution continue à augmenter. Un niveau de compression plus élevé n'aboutit pas toujours à une meilleure performance.

En regardant la vitesse de compression, on trouve que le niveau 2 est le plus efficace pour les 2 plateformes. Par conséquence, on l'adopte comme paramètre par défaut dans les travaux suivants.

4.6. Expérience sur le terrain et analyse des données

Dans cette partie, on teste le programme avec des exemples réels. On cherche puis récupère les nuages de points des surfaces où se trouve une déformation. On les fournit à ce programme puis on cherche à interpréter

les résultats obtenus.

Pour chaque nuage de points, on obtient deux types de résultats dont le nombre de plans et une liste des angles formés par ces plans deux à deux. Vu que l'algorithme de RANSAC n'est pas déterministe, on somme les résultats des 1000 répétitions pour soulager l'influence des facteurs aléatoires.

Jeux de données

Les jeux de données sont présentés ci-dessous. Les images en couleur permettent de mieux comprendre le jeu de données, qui sont en réel les nuages de points. Pour chaque groupe de données, on essaie de prédire puis vérifier le résultat d'analyse des plans.

Figure 14 Jeu de donnée utilisé

- **Enrobé lisse :**

Enrobé lisse est le groupe de témoin, qui est une surface normale sans aucune déformation. On prévoit qu'un seul plan sera détecté.

- **Enrobé bombé :**

- **Enrobé très bombé :**

Enrobé bombé et **enrobé très bombé** sont des ralentisseurs de vitesse de parking. Ils représentent des surfaces convexes similaires à une ornière creuse mais dans le sens opposé.

On constate que le groupe **enrobé très bombé** présente plus de surfaces irrégulières que le groupe **enrobé bombé**. On prévoit de trouver des surfaces dont les angles entre eux sont plutôt aigus, c'est-à-dire plus proches de zéro dans l'intervalle de 0 à 0.5π.

- **Enrobé bouche :**

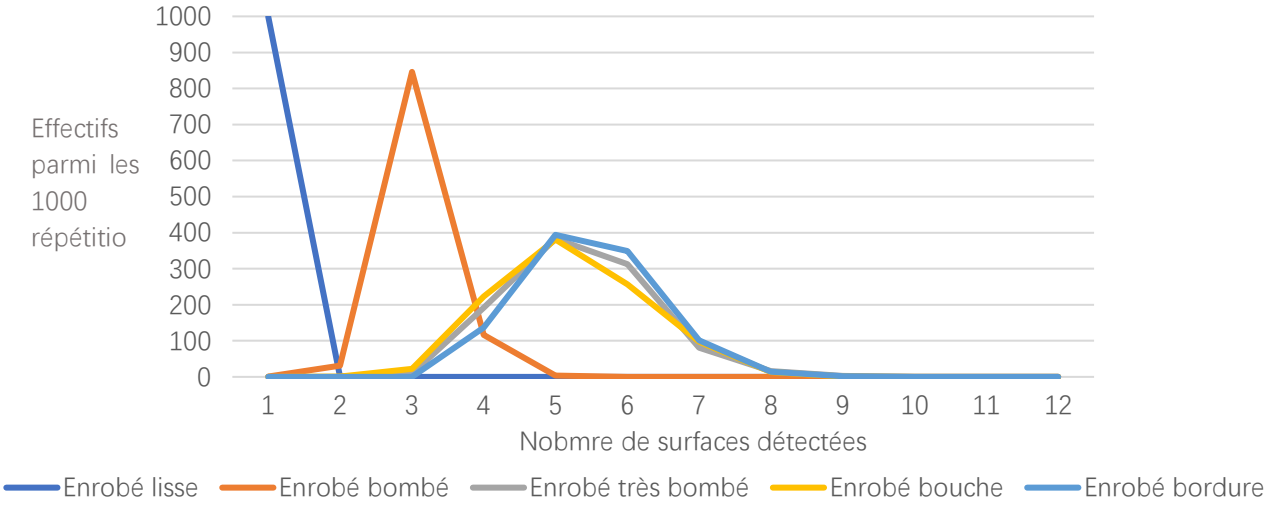
Enrobé bouche est une bouche à eau au milieu de la route. Vu que la frontière entre la bouche et la route est composée des surfaces courbes, on prévoit que les angles soient plus grands par rapport aux jeux de données précédents.

- **Enrobé bordure :**

Enrobé bordure contient un obstacle et la bordure de la route. On constate qu'il y a beaucoup d'angles perpendiculaires. Par conséquent, le résultat sera proche du groupe **enrobé lisse, enrobé bombé et enrobé très bombé**, avec une quantité considérable d'angles perpendiculaires.

Tableau 4 Jeux de données pour le test sur le terrain

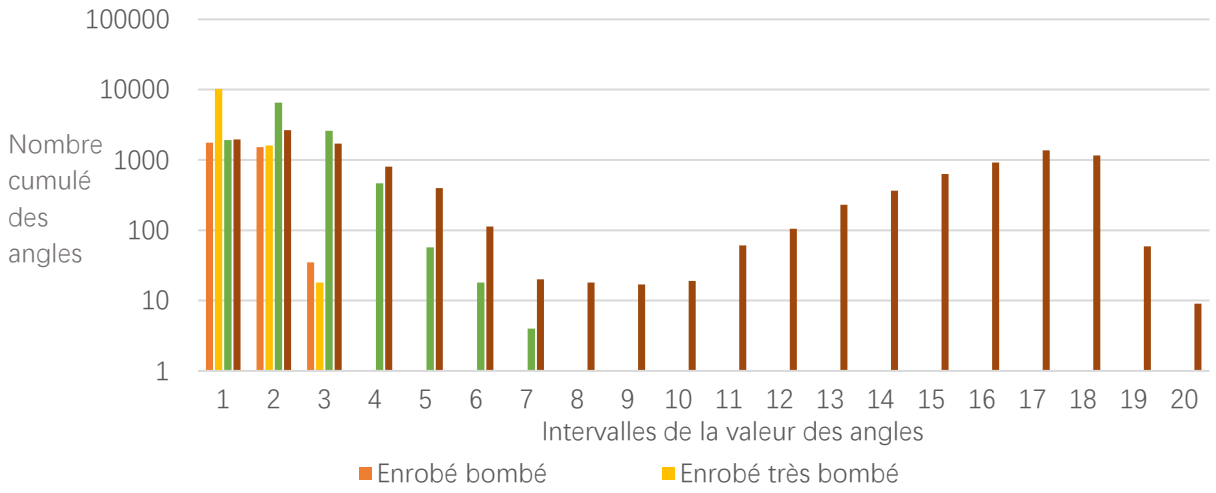
Analyse des résultats



Graphique 7 La distribution de nombre des surfaces détectées

Le graphique 5 montre pour chaque groupe de données, combiens de surfaces sont détectées dans les 1000 itérations. Le graphique 6 est une distribution de tous les angles sur les 20 intervalles subdivisés de 0 à 0.5π.

Par exemple, le 1 sur l'abscisse signifie l'intervalle $[0, \frac{1}{40}\pi[$, et le 2 signifie $[\frac{1}{40}\pi, \frac{2}{40}\pi[$ etc...



Graphique 8 Distribution des angles sur 20 sous intervalles de $[0, \frac{1}{2}\pi[$

Pour le groupe témoin (enrobé lisse), le programme ne détecte toujours qu'une surface, qui est identique à la réalité. Ainsi, aucun angle n'est détecté donc le groupe est absent du graphique 6.

Pour le groupe **enrobé bombé**, on détecte exactement 3 surfaces dans plus de 80% des cas et tous les angles sont inférieurs à 0.05π. Le ralentisseur courbe est traité comme une intersection des plans. On voit que la courbe est assez plate, donc elle est interprétée par des angles aigus.

Le groupe **enrobé très bombé** est similaire au groupe **enrobé bombé**, mais la surface autour du ralentisseur ont plus de déformations le groupe **enrobé bombé**, cependant elle sont très légères. Cela s'interprète par une augmentation très forte des angles aigus. Dans ces deux groupes, aucun angle supérieur à 0.05π n'est détecté.

On constate des angles jusqu'à 0.125π dans le groupe 4, qui proviennent probablement des surfaces connectant le fond et la route.

Le dernier groupe est le plus complexe. D'abord, on constate des angles proches de 0.5π, comme ce que l'on prévoit. La surface de la route est légèrement plus basse que celle où se base l'obstacle. Cette différence peut générer des angles au-dessus de 0.1π (5 et 6 en abscisse).

Tests statistiques

Vu que les données **Enrobé bouche** présentent une forme similaire à la fonction de distribution de la loi gamma, on décide de faire un test de Kolmogorov-Smirnov pour les données de ces deux groupes. Ce test statistique consiste à déterminer si un groupe de données suit une loi décrite par sa fonction de distribution. On a utilisé la fonction ***kstest*** en Matlab pour réaliser ce test. Les hypothèses utilisées sont les suivantes :

- H0 : Les données suivent la loi donnée
- H1 : Les données ne suivent pas la loi donnée

Les données sont traitées de manière similaire que le graphique 8. On divise l'intervalle $[0, \frac{1}{2}\pi[$ en 360 sous

intervalles, puis on prend la fréquence de présence des données dans chaque sous-intervalle pour la fournir à l'algorithme suivant :

	Algorithme : Test de Kolmogorov-Smirnov via Matlab
1	Fitted = *gamfit*(data, 0.1)
2	DistriFunction = *gamcdf*(data, Fitted(1), Fitted(2))
3	[H, alpha] = *kstest*(data, [data, DistriFunction])

Code 2 Test de Kolmogorov-Smirnov pour la loi gamma

L'hypothèse H0 n'est pas rejetée à une signifiance statistique de 5%. Néanmoins la valeur p s'élève à 0.408, ce qui est très élevé pour que l'on accepte la H0 comme le résultat.

On relance le même test avec la loi normale. La H0 est rejetée à une signifiance statistique de 5%, la valeur p est de 0.012.

La conclusion est qu'il y a plus de chance que les données du groupe **Enrobé bouche** suivent une loi gamma qu'une loi normale.

4.7. Passage vers le Raspberry Pi (RasPi) et autres microcontrôleurs

Dans ce chapitre on présente brièvement les travaux dédiés à installer et faire fonctionner les programmes sur la plateforme ARM – les codes ont d'abord été développés sur x86.

Comme le modèle prévu d'un UBR, le RasPi est déjà utilisé pendant les benchmarks dans les chapitres précédents. Les spécificités de hardware ont été présentés dans **le chapitre 4**. Pour compiler le programme exécutable sur une machine de structure ARM, on utilise la commande :

- ***GOARCH=arm GOARM=7 go build***

La valeur de GOARM dépend du processeur de la plateforme. Par exemple, on utilise ***GOARM=6*** dans les autres benchmarks avec la machine virtuelle de RasPi émulée dans une plateforme de x86.

	RasPi ARM	**Plateforme x86**
Bande passante	14.99 MB/s	2.96 MB/s

Tableau 5 Benchmark du programme traitement des données

Le benchmark est réalisé par le traitement successif de 100 images. On constate que le traitement sur un RasPi est beaucoup moins rapide que celui sur une plateforme x86. De plus, il est aussi moins rapide pour la récupération de données (2.96 MB/s contre 6.10 MB/s). Par conséquent, si on connecte deux appareils ou plus d'appareils à un même RasPi, les données non traitées risquent de s'accumuler au fur et à mesure. Il est donc pertinent de chercher des optimisations.

5. Optimisation du programme RANSAC

Dans cette partie on présente plusieurs optimisations réalisées et testées pour notre programme. Une optimisation du programme dans un chapitre est appliquée dans les prochaines. Par exemple, tous les tests dans le chapitre 5.5 sont basés sur l'optimisation de la fonction de lecture du chapitre 5.2.

La plateforme ARM utilisée dans ce chapitre est une machine virtuelle RasPi émulée sur la plateforme de x86. Son processeur virtuel est un ARM1176 et la mémoire est de 512Mbytes.

5.1. Profiling

Le profiling est une méthode efficace pour examiner le programme et connaître son ou ses goulots d'étranglement (bottleneck) pour que l'on puisse y réaliser des optimisations et améliorer la performance. Pour le Golang, il existe un outil de profiling, ***pprof***, qui est à la fois performant et simple à utiliser. Il analyse un programme dans trois aspects :

- **La performance du processeur** : Il s'agit de suspendre le runtime pour profiler la pile et le tas de chaque thread, à un rythme fixé (en général tous les 10 millisecondes). Cela permet d'identifier les hotspots dans le code.
- **La performance de la mémoire** : Il s'agit de profiler l'allocation de la mémoire à un rythme fixé. Il s'intéresse aux les allocations sur le tas. Cela permet d'identifier une consommation anomale de la mémoire.
- **La performance de mutex** : Il s'agit de profiler les attentes des goroutines pour avoir les accès aux ressources. Il est utile pour chercher les goulots pour un code de haut parallélisme ou concurrence.

On va se concentrer principalement sur la performance du processeur car elle est directement liée au temps de calculs. La plateforme du profiling est celle de X86 pendant le benchmark. On obtient un premier framegraph ci-dessous :

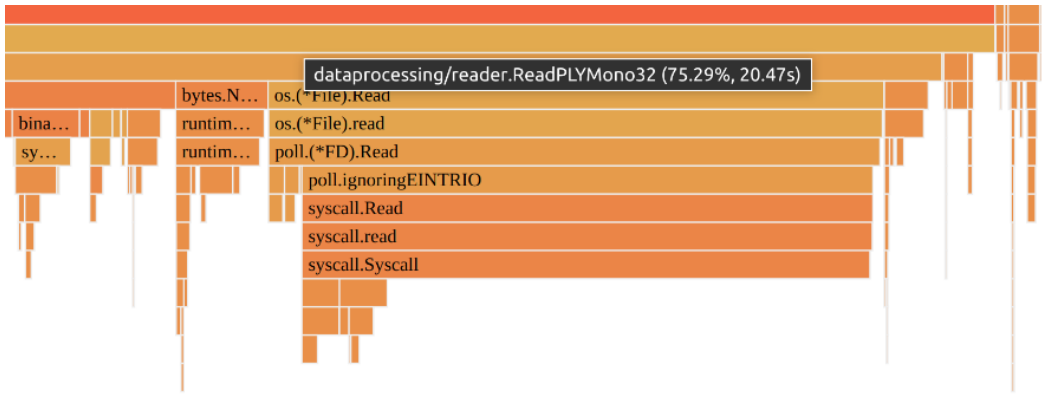


Figure 15 Résultat du premier profiling sur la plateforrme X86

On constate que la fonction qui lit les nuages de points occupe 75.29% du temps de calcul total (la barre jaune

au-dessous de ***main.main***). Ceci n'est pas normal car la plupart des calculs se trouvent dans la fonction d'ajustement des plans qui nécessite d'inverser les matrices. On vise à chercher et localiser les lignes de codes qui causent le problème pour y réaliser une optimisation.

5.2. Optimisation de la fonction de lecture

Algorithme : Lecture des fichiers avant l'optimisation

```
...
1  if type == "vertex"
2      for i := 0; i < NumberOfVertex; i++
3          readAVertex(FilePointer)          # Communication avec le stockage
4          decode(ThisVertex)                # Communication avec la mémoire
...
```

Code 3 Lecture des fichiers avant l'optimisation

On constate que dans les codes originaux, la fonction de lecture est comprise dans les itérations. Elle est appelée une fois pour chaque point dans le nuage de points. Cela conduit à un coût très élevé de communication entre le processeur et le stockage. Les appels des fonctions de système qui lit un seul fichier avec un pointeur est aussi coûteux. Par conséquent, le programme devient très lourd quand on veut lire un nuage de points dont le nombre de points est énorme.

Une solution pratique est de réduire la fréquence de communication avec le stockage externe et les appels de fonctions du système. On est arrivé aux codes suivants :

Algorithme : Lecture des fichiers après l'optimisation

```
...
1  if type == "vertex"
2          readAllVertices(FilePointer)      # Communication avec le stockage
3      for i := 0; i < NumberOfVertex; i++
4          decode(ThisVertex)                # Communication avec la mémoire
...
```

Code 4 Lecture des fichiers après l'optimisation

Dans la nouvelle version, on lit tous les points dans un seul appel de la fonction de lecture. Ainsi, on réduit le nombre d'appel des fonctions de système. Vu que l'on doit itérer pour décoder chaque point, on déplace cette itération vers la mémoire. Ainsi, on supprime l'attente de la communication entre le processeur et le stockage externe.

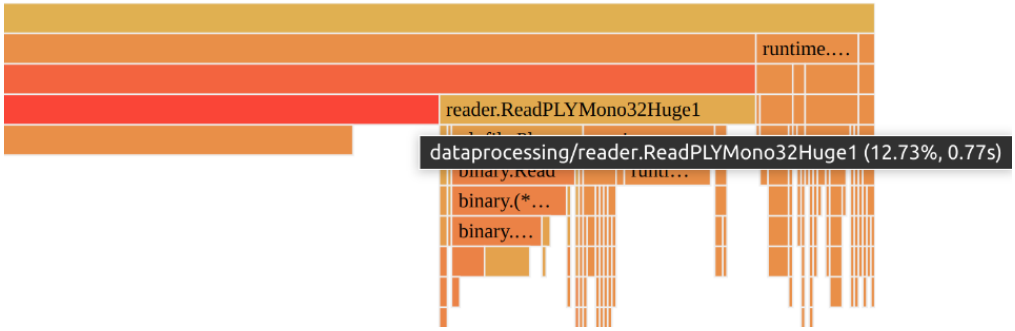
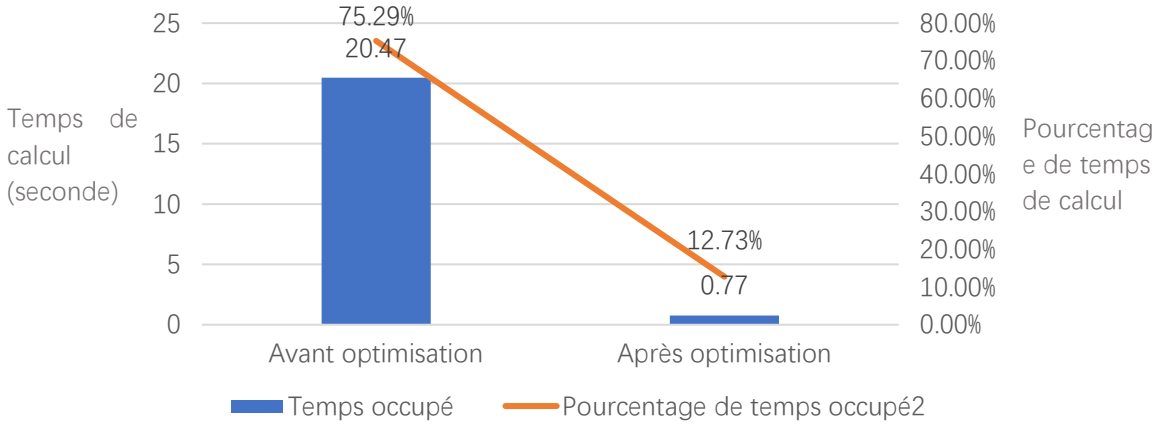


Figure 16 Profiling après l'optimisation de la fonction de lecture

On constate que le pourcentage de temps occupé par la partie de code optimisé a descendu jusqu'à

12.73%. Par conséquent, le temps de calcul baisse de 20.47 secondes à 0.77 seconde.



Graphique 9 Benchmark de l'optimisation sur la plateforme X86

On a aussi fait un benchmark sur la machine virtuelle de Raspi. Le tableau suivant montre que même sur une plateforme dont la performance est très limitée, l'amélioration apportée par cette optimisation est remarquable. Le speedup s'élève à 3.865.

	Avant optimisation	**Après optimisation**
Bande passante	0.104 MB/s	0.402 MB/s

Tableau 6 Benchmark de l'optimisation sur la Raspberry virtuelle ARM

5.3. Distance euclidienne entre un point et un plan

Pour le point $X(x_0, y_0, z_0)$ et le plan $A(ax + by + cz + d) = 0$, la distance euclidienne entre eux est calculée par la formule :

$$d(X, A) = \frac{|ax_0 + by_0 + cz_0 + d|}{\sqrt{a^2 + b^2 + c^2}} \quad (8)$$

La formule contient une opération de l'inverse d'une racine. Elle est appelée massivement par la méthode de RANSAC. Or ces deux opérations sont couteuses par rapport à l'addition et multiplication, dont le nombre de cycle sont 10 à 100 fois plus élevé. On cherche un moyen alternatif qui peut améliorer la performance avec moins de cycles de processeur.

Optimisations

L'implémentation par défaut est un appel direct aux fonctions de la bibliothèque math :

	Algorithme : Calcul de distance version originale
1	**func** *dist* (x, y, z, a, b, c, d)
2	**return** *math.Abs*(x * a + y * b + z * c + d) / *math.Sqrt*(a * a + b * b + c * c)

Code 5 Calcul de distance version originale

Le premier essai consiste à supprimer l'opération de racine, en utilisant une méthode basée sur un calcul approximatif de l'itération de Newton. Elle utilise un « nombre magique » (LEMONT) Cette méthode est utilisée dans beaucoup de jeux vidéo dans les années de 2000 pour calculer les reflets des faisceaux de lumières. Le code est comme suit :

```
Algorithme : Calcul de distance alternatif 1
1  func dist1 (x, y, z, a, b, c, d)
2     buff = a * a + b * b + c * c
3     X = buff * 0.5
4     I = 0x5fe6eb50c7aa19f9 – (math.Float64bits(buff) >> 1)
5     Y = math.Float64frombits (I)
6     Y = Y * (1.5 – X * Y * Y)
7     Y = Y * (1.5 – X * Y * Y)
8     return math.Abs(x * a + y * b + z * c + d) * Y
```

Code 6 Calcul de distance alternatif (1) : Fast square root

La deuxième alternative est de continuer à utiliser les fonctions standard de la bibliothèque, mais sortir les multiplications en dehors. On crée deux variables temporaires :

```
Algorithme : Calcul de distance alternatif 2
1  func dist2 (x, y, z, a, b, c, d)
2     temp1 = x * a + y * b + z * c + d
3     temp2 = a * a + b * b + c * c
4     return math.Abs(temp1) / math.Sqrt(temp2)
```

Code 7 Calcul de distance alternatif (2)

On fait un benchmark par la bibliothèque standard du Golang sur la plateforme X86 :

	Version originale	Alternatif (1) Fast square root	Alternatif (2)
Résultat d'un calcul	14.6699317	14.6699035	14.6699317
Temps moyen par opération (ns/op)	0.3003	5.070	0.2830

Tableau 7 Benchmark du calculateur de distance

Le temps moyen par opération de l'alternative 2 est légèrement mieux que la version originale en utilisant deux variables temporaires. Cette valeur est très élevée pour l'alternative 1 qui utilise la méthode de fast square root. Il risque d'y avoir une erreur. On étudie la performance de cette méthode par un autre benchmark dans lequel on calcule les racines carrées depuis 1 jusqu'à 10^e7.

	fast square root	math.Sqrt()
Temps pour les racines carrés de 1 à 10^e7 (ms)	22.24	56.44

Tableau 8 Performance réelle du fast square root

On constate que la méthode de fast square root a une meilleure performance en réel. Le speedup est de 2.538. Avec une erreur minuscule, il sera logique de substituer cette méthode à la fonction interne de la bibliothèque math.

On a aussi étudié la nécessité de la deuxième itération de la méthode de Newton. La méthode du benchmark

est comme ci-dessus. On constate qu'une amélioration de la précision est gagnée à un prix assez minuscule. Le taux d'erreur est 34.2 fois moins élevée avec un temps de calcul qui est 33.94% plus long.

	1 itération	2 itérations
Valeur finale (racine carrée de 10000)	99.829	99.995
Taux d'erreur	0.171%	0.005%
Temps de calcul (ms)	17.15	22.97

Tableau 9 Performance de fast fast square root avec 1 et 2 itérations de la méthode de Newton

Analyse des codes assembleurs

```
   .     4bf241: SHRQ $0x1, AX                          ;main.go:20
  10ms   4bf244: MOVQ $0x5fe6eb50c7aa19f9, CX           ;main.DistPointPlane1 main.go:20
   .     4bf24e: SUBQ AX, CX                            ;main.go:20
   .     4bf251: MOVQ CX, X5                            ;unsafe.go:29
   .     4bf256: MOVUPS X2, X6                          ;main.go:22
   .     4bf259: MULSD X5, X2
   .     4bf25d: MULSD X5, X2
   .     4bf261: MOVSD_XMM $f64.3ff8000000000000(SB), X7
   .     4bf269: SUBSD X2, X7
  50ms   4bf26d: MULSD X5, X7                           ;main.DistPointPlane1 main.go:22
   .     4bf271: MULSD X7, X6                           ;main.go:23
   .     4bf275: MULSD X7, X6
   .     4bf279: MOVSD_XMM $f64.3ff8000000000000(SB), X2
  10ms   4bf281: SUBSD X6, X2                           ;main.DistPointPlane1 main.go:23
   .     4bf285: MULSD X7, X2                           ;main.go:23
  80ms   4bf289: MOVSD_XMM 0x8(SP), X5                  ;main.DistPointPlane1 main.go:24
  80ms   4bf28f: MULSD X5, X1
   .     4bf293: MOVSD_XMM 0x10(SP), X5                 ;main.go:24
   .     4bf299: MULSD X5, X3
   .     4bf29d: ADDSD X1, X3
   .     4bf2a1: MOVSD_XMM 0x18(SP), X1
  10ms   4bf2a7: MULSD X1, X4                           ;main.DistPointPlane1 main.go:24
   .     4bf2ab: ADDSD X3, X4                           ;main.go:24
```

Figure 17 Codes assembleurs de l'alternatif (1)

On constate que le temps de calcul est dispersé à chaque ligne de code. Selon l'image 6, ce qui consomme le plus sont les lignes 22 – 24. Les lignes 22 – 23 répètent 2 fois l'itération de Newton pour augmenter la précision. La ligne 24 calcule puis retourne le résultat final.

```
   .     4bef95: MOVQ AX, 0x40(SP)
   .     4bef9a: MOVB $0x3, 0x3f(SP)
   .     4bef9f: XORL AX, AX
   .     4befa1: JMP 0x4befa6                           ;main.go:50
2.78s    4befa3: INCQ AX                                ;main.main main.go:50
480ms    4befa6: MOVQ $0x2540be400, CX
 90ms    4befb0: CMPQ CX, AX
   .     4befb3: JL 0x4befa3                            ;main.go:50
   .     4befb5: MOVB $0x1, 0x3f(SP)                    ;main.go:54
   .     4befba: CALL runtime/pprof.StopCPUProfile(SB)
   .     4befbf: MOVB $0x0, 0x3f(SP)
   .     4befc4: MOVQ 0x48(SP), AX
   .     4befc9: MOVQ AX, 0(SP)
   .     4befcd: CALL os.(*File).Close(SB)
```

Figure 18 Codes assembleurs de la version originale

Néanmoins, le ***pprof*** ne peut pas analyser plus précisément les codes de l'implémentation originale et

l'alternative (2). Selon l'image 7, ce qui consomme le plus est un déplacement de donnée type ***qword*** entre les registres. Or un déplacement ne peut pas être si coûteux. On a essayé d'augmenter la fréquence d'échantillonnage de ***pprof*** jusqu'à 500Hz et le résultat ne change pas.

Une explication possible est que la fonction de conversion de type que l'on utilise est couteuse. Après avoir consulté le code de la fonction ***math.Sqrt()***, on découvre que la fonction utilise la réalisation de la racine carré du système s'il en existe une. Cela peut expliquer pourquoi l'échantillonnage s'arrête en dehors de la fonction noyaux.

5.4. Format de données 32 / 64 bits

Pendant les calculs précédents, les données sont transformées en flottant de 64 bits pour passer au package ***Gonum***. On cherche une amélioration potentielle en utilisant les flottants de 32bits. Pour cela on est obligé de faire toutes les opérations mathématiques en 32 bits, ce qui est difficile pour la méthode d'inverse de matrice car le package ***Gonum*** est en 64bits.

On a réussi à transférer la méthode SVD en 32 bits avec succès. Par conséquent, les tests suivants utilisent la méthode de SVD en 32 et 64 bits. Le programme reste strictement identique sauf la longueur des données. On répète la lecture des données et l'ajustesment des plans 100 fois et on calcule le temps passé. Le jeu de données utilisé est une surface simple, pour réduire l'influence aléatoire de RANSAC. Le résultat du programme est exactement 1 plan, ce qui est correcte.

	Temps de calcul (s)	**Consommation de mémoire (KBytes)**
32bits	21 196	18.88
64bits	36 752	29.23

Tableau 10 Résultats des benchmarks 32/64bits sur la platforme x86

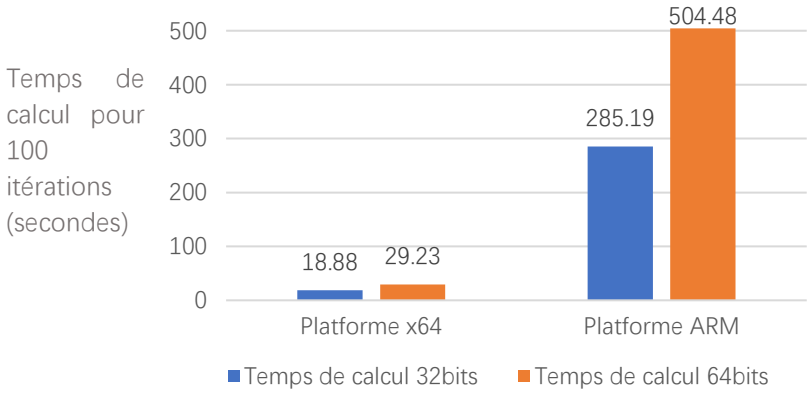
La configuration de la plateforme x86 correspond au chapitre 3.4. Le programme est chronométré par le package ***time***. La consommation de mémoire est identifiée par la commande ***grep***. On constate que pour la plateforme x86, la version de 32 bits consomme moins 42.51% de temps et 35.41% moins de mémoire que celle de 64 bits.

On utilise la machine virtuelle de RasPi pour le test sur la machine virtuelle ARM. L'émulation est sur la plateforme x86 ci-dessus.

	Temps de calcul (s)
32bits	285.19
64bits	504.48

Tableau 11 Tableau 6 Résultats des benchmarks 32/64bits sur la machine virtuelle ARM

On constate que pour la plateforme ARM, la version de 32 bits est 43.47% plus rapide que la version de 64bits. Le profiling par pprof donne des flamegraphs et codes assembleurs similaires. On pourra conclure que cette différence vient exactement des différentes longueurs de données.



Graphique 10 Temps de calcul sur différentes plateforme, 32bits et 64bits

5.5. Parallélisation via goroutines

A priori, le traitement d'un nuage de points permet une approche fortement parallélisable. L'accumulation de données à traiter peut-être supérieure à la vitesse de traitement. Si on utilise une machine Raspberry Pi pour traiter les données collectées, les données s'accumuleront forcément de manière supérieure au traitement sur x86.

Les goroutines sont une fonctionnalité du Golang qui nous permettent de réaliser la concurrence de manière simple. On cherche à paralléliser ce programme en utilisant les goroutines pour faire face à cette accumulation des données. Dans l'exemple qui suit, on génère une liste contenant les noms des fichiers existants à traiter. Une goroutine sera lancée pour traiter le fichier à chaque itération. A la fin du traitement, les résultats seront enregistrés dans le logger. Un lock est utilisé pour éviter les conflits d'écritures en lignes 9 à 11. On attend que toutes les goroutines soient terminées en ligne 14.

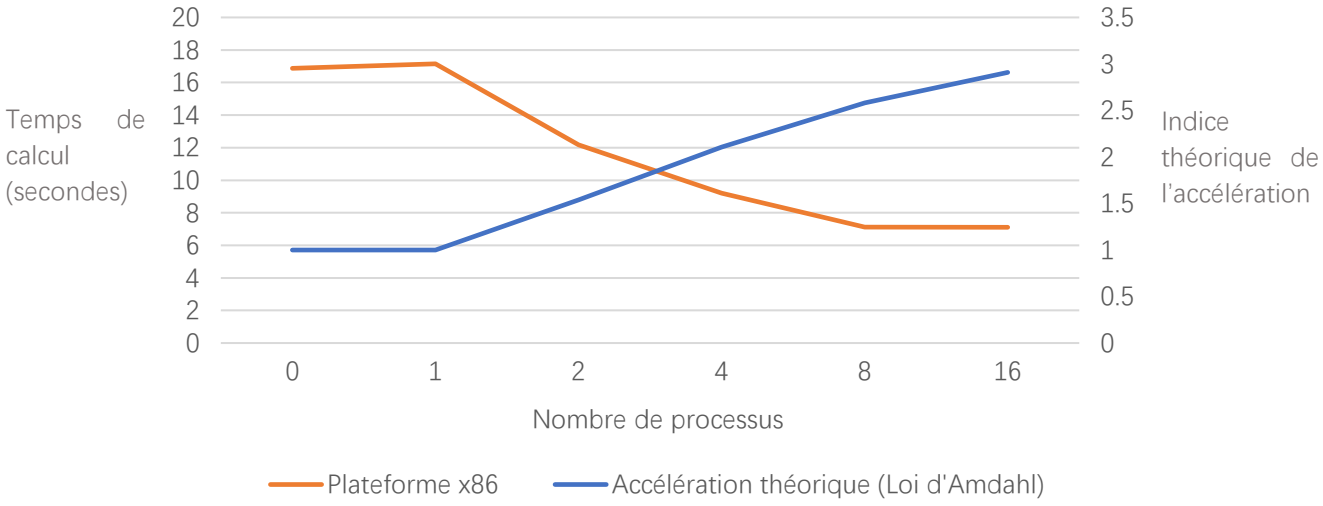
Algorithme : Version parallèle pour le traitement des nuages de points

```
...
1   wg := sync.Waitgroup{}
2   wg.add(length)
3   var mutex sync.Mutex
4
5   for i := 0; i < length; i++
6       go func(filename)
7           results := RANSAC(filename)
8           mutex.Lock()
9           log.Println(results)
10          mutex.Unlock
11          wg.Done()
12
13  wg.Wait()
...
```

Code 8 Traitements parallèles des nuages de points

On vérifie la performance obtenue d'après la loi d'Amdahl : $Acc = \frac{1}{1-p+\frac{p}{s}}$ où :

- Acc est l'accélération obtenue de la parallélisation.
- p est le pourcentage de la partie parallélisable. On l'estime par le pourcentage de la lecture et le traitement de données. Ce pourcentage s'élève à 0.7 selon le flamegraph en chapitre 5.1.
- s est le nombre de threads, ici on utilise le nombre des cœurs logiques du processeur.

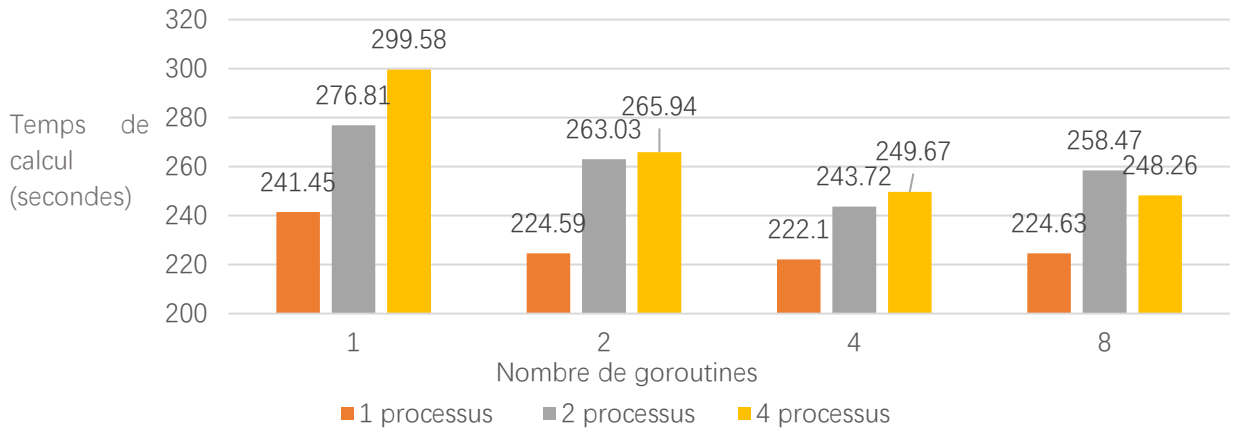


Graphique 11 L'accélération de la parallélisation sur la plateforme x86

Le nombre de cœurs 0 correspond à la version non parallélisée. On fait varier le paramètre ***GOMAXPROCS,*** qui désigne le nombre de cœurs du processeur accordés au programme. Ce paramètre force le programme à réaliser la vraie parallélisation dans les différents cœurs du processeur utilisés, à la place du concurrency sur le même cœur.

On constate que la performance (le temps de calcul) est positivement corrélée au speedup théorique. A partir de 8 cœurs, le gain devient minuscule car la machine de test est équipée d'un processeur de 4 cœurs physiques dont 8 cœurs logiques. Quand le nombre de processus est supérieur au nombre de cœurs, on risque d'avoir un gain très faible voir même négatif.

Dans l'étape suivante on passe au test sur la machine virtuelle de la machine virtuelle ARM. A cause de sa performance (fréquence du processeur et la mémoire) limité, on ajoute une autre variable, qui est le nombre de goroutines en exécution au même temps. Cela est réalisée par un channel qui accepte une variable booléenne au début de chaque itération puis le relâche à la fin d'un goroutine. La taille du channel correspond à ce nombre.



Graphique 12 L'accélération de la parallélisation sur la machine virtuelle ARM

On constate que pour un nombre de processus donné, le temps d'exécution baisse jusqu'à 4 goroutines. Le nombre maximal possible de goroutines est 8 à cause de la mémoire limitée. La valeur optimale est 4. Si on regarde les différents nombres de processus, on constate que la performance optimale est déjà atteinte à 1 processus, alors que l'on obtient un gain de performance jusqu'à 8 processus en x86. Cela correspond au fait que le processeur émulé est du cœur unique et d'une performance limitée.

Ainsi, on prévoit une amélioration de performance plus grande sur un modèle de RasPi 3 ou plus récent grâce à son processeur de quad-cœurs.

5.6. Conclusion

Dans ce chapitre, on a réalisé plusieurs optimisations dont chacune a apporté une amélioration de performance considérable. La modification du lecteur apporte la plus grande accélération et pourrait servir comme un package dans autres projets. L'utilisation de la méthode fast square root présente une meilleure performance que celle intégrée de Golang, avec un compromis sur la précision de calcul. L'utilisation de calcul en 32 bits a amélioré la performance de la méthode SVD, alors qu'elle est toujours moins performante que la méthode des moindres carrés directe. La parallélisation via les goroutines permet de mieux exploiter un processeur à plusieurs cœurs, notamment pour un RasPi 3 ou plus récent.

6. Conclusion Générale et perspective

Dans ce rapport, on a présenté une méthode low-cost de détection de déformations routières basée sur les nuages de points. On a construit un système Edge computing pour faire fonctionner cette méthode. On a aussi réalisé plusieurs optimisations qui ont apporté des améliorations considérables. Les expériences sur le terrain ont montré que l'algorithme est capable de détecter les déformations de la surface routière.

Sur une note personnelle, ce stage à Forclum Numérique m'a permis de développer mes compétences en informatique. J'ai aussi découvert le monde de l'entreprise et les méthodes de travail. Grâce à ce stage, je suis mieux préparé pour partir dans le monde professionnel.

Il existe plusieurs orientations possibles pour les recherches futures concernant ce sujet.

Tout d'abord, les expériences avec les vraies déformations, causées par l'usage, sont à faire. L'algorithme est-il capable de détecter les déformations très légères ? Dans ce cas, quels paramètres faudra-t-il donner ? Ces questions sont à répondre. Une relation entre l'output de l'algorithme et la gravité de déformation est aussi à construire.

Puis, on pourra exploiter les photos prises par la caméra RGB sur le même appareil. Une possibilité est de faire une reconnaissance de caractéristique à l'aide d'un réseau de neurones. Les résultats peuvent aider l'algorithme à déterminer la gravité de déformation.

Il est aussi possible d'améliorer l'ajustement des plans. Le RANSAC est un algorithme efficace mais aussi primitif. On pourra le fusionner avec d'autres approches comme le region growing. Les méthodes d'apprentissage automatique peuvent aussi aider.

7. Bibliographie

ADEME. (2017). *I-street innovations systémiques au service des transitions écologiques et énergétiques dans les infrastructures routières de transport route à énergie positive.* www.ademe.fr/i-street.

Cheney, D. (2016). cgo is not Go.

Hoppe, H. D. (1992). Surface reconstruction from unorganized points. Dans *ACM SIGGRAPH Computer Graphics* (pp. 71-78).

https://www.jianshu.com/p/d60eff598aa0. (s.d.). *Comparaison de vitesse entre plusieurs langages de programmation.*

LEMONT, C. (s.d.). FAST INVERSE SQUARE ROOT.

Martin A. Fischler, R. C. (1981). Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. Dans *Communications of the ACM* (pp. 381–395).

Weisong Shi, J. C. (s.d.). *Edge Computing: Vision and Challenges.* Wayne State University.

8. Annexes

8.1. Expérience sur le microcontrôleur STM32L476

Etant équipé d'un processeur ARM d'ultra faible consommation, ce microcontrôleur de STM électroniques est aussi un micro-ordinateur comme un RasPi. Mais il est allé plus loin pour chercher une consommation énergétique autant moins élevée que possible, jusqu'au niveau du nanowatt en mode veille et µwatt en mode normal. Comparé à la puissance énergétique au niveau du watt de RasPi, ce type de microcontrôleur se trouve dans le monde des calculs à une puissance ultra-basse.

Néanmoins, cette puissance est liée à une performance très limitée par rapport à RasPi, comme suit :

	RasPi 4	**STM32L476**
Puissance	2.7 W – 6.4 W	200 nW – 360 µW
Processeur	Arm® 64-bit Cortex®-A72, 1.5GHz	Arm® 32-bit Cortex®-M4, 80MHz
Mémoire	Jusqu'à 4 GB	128 KB
Stockage	Carte SD / SSD externe	1 MB

Tableau 12 Configuration des hardwares ARM

Les microcontrôleurs sont beaucoup utilisés dans les contrôles industriels, les embarqués et l'IoT. Mais on constate qu'il n'est pas assez performant pour le projet. Le stockage est très limité pour installer le driver fabriqué par la bibliothèque Realsense2.

On a aussi installé le système d'opération ***RT-Thread*** sur ce microcontrôleur. Il est un système compact et intégré, pour les plateformes de l'IoT et des embarqués. On a réussi à implémenter un programme d'essai qui réalise des outputs simples. Mais il est difficile d'y développer des fonctionnements plus sophistiqués car la plupart des bibliothèque et fonctions utiles ne sont pas supportées.