Smagghe Cyril
Tournier Jean-Michel

Rapport de projet (décembre) : P25 Architecture ROS pour des véhicules autonomes intelligents

IMA 5 2015-2016
Tuteurs : Vincent Coelen et Rochdi Merzouki

Réalisé du 21/09/2015 au 14/12/2015

Table des matières

1 Introduction **3**

2 Présentation du projet **4**
2.1 Contexte 4
2.2 État de l'art 5
2.3 Cahier des charges 6

3 Présentation des outils de développements **8**
3.1 ROS 8
3.2 Matlab-Simulink 8
3.3 PURE 9

4 Travail réalisé **11**
4.1 Corrections de la direction et de la traction 11
4.2 Serveur PURE 12
4.3 Laser et évitement d'obstacle 12
4.4 Map 13
4.5 Path finding 15

5 Prochaines étapes **17**
5.1 Localisation 17
5.2 GPS-Odométrie 17
5.3 Suivi de chemin 17
5.4 IHM et convertisseur 17

6 Conclusion **18**

Liste des figures **19**

Remerciements

Nous tenons à remercier M. Merzouki Rochdi et M. Coelen Vincent pour nous avoir proposé ce PFE à la fois enrichissant et multidisciplinaire. Plus particulièrement, nous remercions l'expertise de M. Coelen qui permet au projet de prendre forme au fil des semaines.

Nous remercions M. Pollart Michel pour sa disponibilité lors d'un besoin matériel pour, par exemple, avoir mis à notre disposition le laser ou prochainement le récepteur GPS.

Nous remercions tous les acteurs qui ont contribué au projet InTraDE et notamment ceux dont nous réutilisons les travaux.

1 Introduction

Les véhicules autonomes prennent de plus en plus d'importance depuis la dernière décennie. Dans le cadre de notre projet de fin d'étude, nous sommes en train de rendre le déplacement d'un véhicule électrique autonome. Bien évidemment, de part le manque de temps et de moyens, nous ne pouvons réaliser un véhicule capable des mêmes prouesses. Cependant, au terme du projet, le véhicule devra pouvoir se déplacer sans intéraction humaine directe dans le campus universitaire de Lille 1. Ce PFE s'inscrit dans le cadre du projet industriel InTraDE.

Ce rapport de mi-parcours va donc introduire au lecteur les différents aspects de ce projet ainsi que les divers outils de développement à disposition qui permettront au projet d'aboutir. Le travail accompli sera ensuite développé dans le rapport. Enfin, les prochaines étapes à réaliser pour le PFE seront présentées.

FIGURE 1 – robuCAR dans le hall de Polytech Lille

2 Présentation du projet

2.1 Contexte

Au sein de l'école sont présents trois robuCARs, véhicules électriques pouvant transporter jusqu'à 400 kg et étant limité à la vitesse de 18 km/h. Chacun des véhicules possède une technologie de commande embarquée différente de l'autre. Le véhicule sur lequel l'essentiel du travail va être exécuté dispose d'une dSpace 1103.

Dans le cadre du projet InTraDE (Intelligent Transportation for Dynamic Environment), le RobuTainer se doit de se déplacer de façon autonome dans un environnement restreint en transportant sur son châssis un conteneur d'un point à un autre. En effet, avec la mondialisation, le commerce maritime s'est grandement développé. Afin d'améliorer la compétitivité de plusieurs ports européens tels que celui de Rouen, d'Ostende, de Dublin, ce véhicule se déplacera de façon autonome dans les ports grâce à sa batterie de capteurs : GPS, laser, etc. Il s'adapte à l'environnement existant tout en limitant les risques de dysfonctionnement grâce à ses 8 roues motorisées (soit 4 pour la traction et 4 pour la direction). Les roues peuvent prendre toutes les directions possibles, ce qui lui offre donc une grande maniabilité malgré sa taille.

FIGURE 2 – Schéma représentant le robuTAINER

Le projet InTraDE contribue à une gestion du trafic portuaire plus fluide et permet d'optimiser l'espace dans les zones confinées en développant un système de transport intelligent et écologique, cela offrant ainsi une meilleure sécurité pour les humains. Pour notre projet, il est ici proposé de travailler à l'aide d'un robuCAR pour le simple fait qu'il est bien moins encombrant et par conséquent plus adapté pour les tests d'algorithmes développés pour InTraDE. En effet, la taille du robuTAINER a empêché par le passé le test de projets, par manque d'espace et d'autorisations.

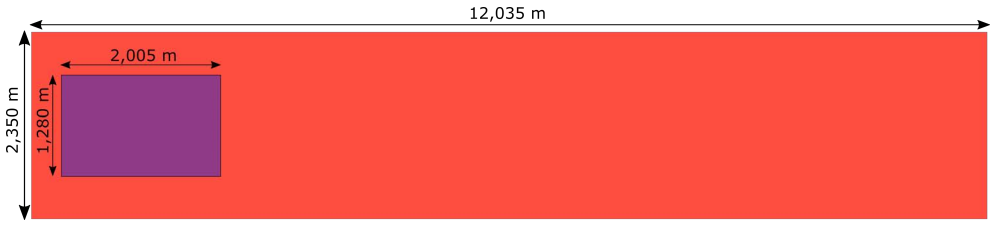


FIGURE 3 – Différence de taille entre le robuTAINER et le robuCAR

2.2 État de l'art

Le concept de voitures autonomes est né dans les années 1970. Mais c'est seulement à partir des années 1980 que nous pouvons constater de véritables avancées. En 1984, Mercedes-Benz teste une camionnette équipée de caméra sur un réseau routier sans trafic. En 1987, le projet EUREKA Prometheus est financé par la Commission Européenne pour développer des outils technologiques dédiés à la conduite automobile automatique. En 1994, en situation réelle de trafic, deux véhicules autonomes réalisent une démonstration de conduite en file, de changement de file et dépassement sur l'autoroute A1. En 2004, l'agence DARPA organise un concours réservé aux voitures autonomes, c'est le DARPA Grand Challenge. Le but est d'arriver en moins de 10h au bout d'un circuit d'une longueur de 240 km. Lors de la première édition, aucune équipe ne réussit le challenge. Mais, lors de l'édition suivante, cinq équipes réussirent à parcourir le circuit complet dont quatre dans la limite des 10 heures.

FIGURE 4 – Stanley, vainqueur du DARPA Grand Challenge 2005

En 2007, la DARPA lance une nouvelle compétition : la DARPA Urban Challenge. Les véhicules devaient s'intégrer au trafic routier tout en respectant le code de la route et remplir des missions d'approvisionnement sur une distance de près d'une centaine de kilomètres. En 2010, Google annonce l'arrivée des GoogleCars après avoir conçu

un système de pilotage automatique pour automobile sur 8 voitures (six Toyota Prius, une Audi TT et une Lexus RX-450h). Le système de pilotage utilise une caméra, des radars, un lidar, un récepteur GPS ainsi que des capteurs sur les roues motrices. De nos jours, de nombreux constructeurs automobiles travaillent sur des projets de voitures autonomes tels que Toyota, Audi, Renault, ... Nissan et Volvo ont d'ailleurs annoncé qu'ils souhaitaient commercialiser leurs premiers véhcules sans conducteur d'ici 2020.

Notre travail n'a donc pas l'ambition d'être au niveau de ses véhicules, mais de proposer une architecture simple, fonctionnelle et évolutive, avec les moyens à notre disposition. Ce projet pourra aussi servir de base pour des futurs travaux dans ce domaine.

2.3 Cahier des charges

Le projet consiste à automatiser le déplacement d'un véhicule électrique dans un environnement confiné. Tout le projet sera réalisé dans le campus universitaire de Lille 1. Cependant, il faudra faire en sorte que le système soit facilement adaptable pour un autre lieu. L'architecture ROS qui permettra aux robots d'être autonomes sera aussi générique que possible pour permettre sa réutilisation sur d'autres plates-formes mobiles. La synthèse de différents codes existants du projet InTraDE et son adaptation permettront au binôme de rassembler ces différentes briques dans le but de relier le tout ensemble.

Les différents objectifs du projet sont les suivants :

- Prendre connaissance de l'architecture matérielle et logicielle du robuCAR et réaliser les modifications et corrections nécessaires
- Réaliser une architecture ROS générique permettant de rendre autant que possible le déplacement autonome dans le campus
- Créer une interface homme machine pour le contrôle du robot

Choix techniques et matériels :

Le robuCAR dispose d'une dSpace 1103, carte gérant de nombreux modules d'entrées/sorties et d'une grande puissance de calcul pour le prototypage de lois de commande. La dSpace est contrôlée à l'aide de Matlab R2006a (Simulink) et de ControlDesk installés sur l'ordinateur fixe présent dans le robuCAR. A cette dSpace sont reliés le GPS ainsi que les données odométriques, ce qui permettra de réaliser la localisation.

Un PC portable avec un Ubuntu 14.04 LTS aura un laser Sick LMS221. A l'aide de ROS, il chargera la map, gérera la recherche de chemin, le suivi de chemin, l'évitement d'obstacle et l'interface homme machine.

Le schéma suivant reprend donc les éléments existants sur le robuCAR, ainsi que ceux à créer. Il représente aussi les différentes couches de transmissions des informations.

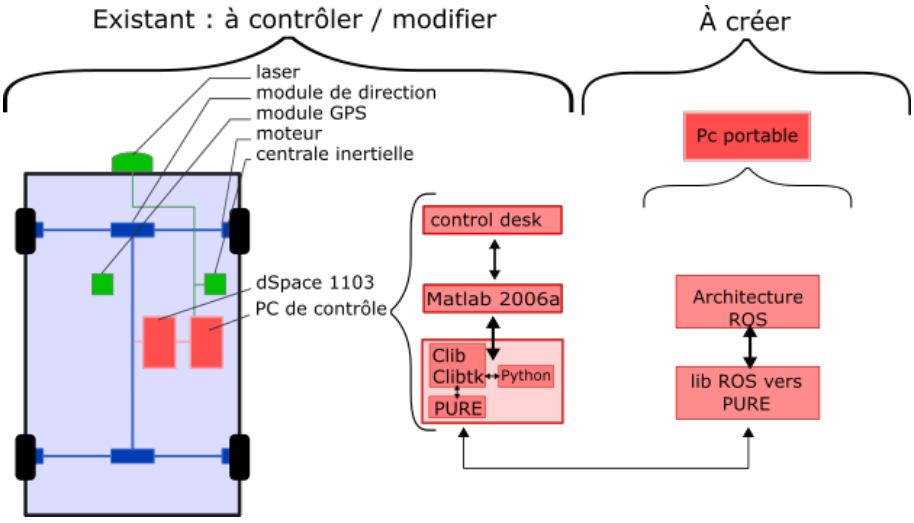


FIGURE 5 – Vue globale de l'architecture du système

3 Présentation des outils de développements

3.1 ROS

Robot Operating System (ROS) est une collection d'outils, de librairies et de conventions permettant l'écriture de logiciels de robots. ROS vise à simplifier la création d'un comportement de robot complexe et robuste et cela à travers une très grande variété de plates-formes robotiques. ROS a été conçu dans le but de permettre à de petits groupes de programmeurs de collaborer et de s'appuyer sur le travail de chacun afin d'obtenir un logiciel qui englobe tous les divers travaux.

Il existe certaines distributions ROS. Dans notre projet, nous utilisons la distribution Indigo Igloo car c'est celle qui est la plus récente et la plus stable pour Ubuntu 14.04. Il est possible de travailler soit en C++, soit en Python.

3.2 Matlab-Simulink

Simulink est un logiciel de modélisation intégré à Matlab. Un environnement graphique et une collection de bibliothèques contenant des blocs de modélisation permettent à l'utilisateur de simuler un système réel voire de contrôler le système en question. Étant donné que Simulink est intégré à Matlab, l'utilisateur a également accès aux outils de développement algorithmique, de visualisation et d'analyse de données de Matlab.

Simulink peut modéliser des données simples ou multicanaux, des composants linéaires ou non. Simulink peut simuler des composants numériques, analogiques ou mixtes. Il peut modéliser des sources de signaux et les visualiser. De plus, il est également possible de décomposer les diagrammes hiérarchiquement en emboitant des sous-systèmes dans des systèmes. Ce logiciel permet donc de régler le comportement du robot, en intégrant le programme compilé dans la dSpace, notre système de commande embarqué.

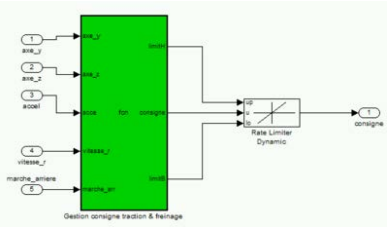


FIGURE 6 – Un des blocs Simulink que nous avons eu à modifier

3.3 PURE

Pure (Professional Universal Robot Engine) est un protocole développé et utilisé par robosoft, le fabriquant des robuCAR, robuTAINER ou encore RobuRIDE que l'on retrouve dans les locaux de Polytech. Il permet de contrôler chacune de ses plateformes robotiques de façon universelle. Il permet une certaine abstraction matérielle en communiquant des informations sur les caractéristiques du robot, en autorisant la récupération d'informations de ses capteurs ainsi que le contrôle des actionneurs de façon standard.

Ce protocole repose sur le principe d'instructions de type questions - réponses via paquets UDP, et les données du robot sont réparties dans des services. Les serveurs PURE implémentés dans les contrôleurs des robots de robosoft répondent alors aux instructions qui lui sont envoyées. Des services "system" permettent la découverte et la gestion des services "application".

Le service Directory répond par exemple à une demande avec la liste des services "applications" disponibles, tandis que le service Notification permet lui, de s'abonner aux informations disponibles par tel ou tel service. Enfin les services "application" tel que le service "Car", "Laser" ou "Localization" fournissent tout d'abord des informations sur le véhicule ou le capteur, puis via un abonnement aux notifications envoient leur état, soit périodiquement ou sur évènements.

Il est après possible de leur adresser aussi des notifications entrantes, afin de contrôler le déplacement du robot, en lui indiquant par exemple une vitesse de consigne et une direction.

Ainsi le schéma suivant résume une communication type avec un robot, sur lequel on obtient la liste des services, puis les propriétés du service “Car”, et enfin l'abonnement aux notifications de ce service afin d'obtenir régulièrement l'état des actionneurs en vitesse et direction. Une fois cela effectué le client reçoit alors les notifications sortantes, et peut modifier les consignes avec une notification entrante.

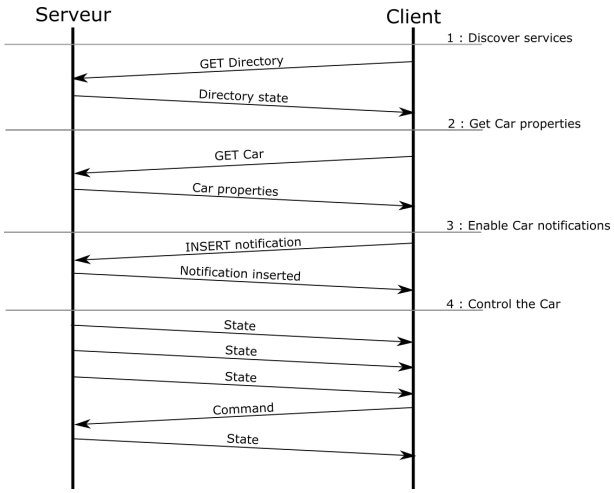


FIGURE 7 – Schéma d'un échange PURE type

4 Travail réalisé

4.1 Corrections de la direction et de la traction

A la suite de la lecture du rapport de projet du groupe de 2012/2013 ayant participé à l'intégration du système de commande embarqué dSpace, nous avons identifié un souci majeur, entrainant sous certaines conditions un blocage du train de direction du robuCAR. Nous avons donc commencé à prendre en main dès les premières semaines le robot, en lui faisant faire quelques tours de piste, afin de constater ce souci et d'identifier d'éventuels autres problèmes.
Le problème de blocage du train de direction est effectivement apparu, et nous avons aussi identifié des problèmes avec la marche arrière mettant trop de temps à démarrer, et le freinage en marche arrière, complètement inactif.

Pour le blocage du train de direction, il s'agissait en fait d'une limitation logicielle mise en place afin d'éviter d'emmener les trains de direction en butée physique. Malheureusement l'implémentation de cette limite entrainait une désactivation de toute commande sur le train de direction après un dépassement d'une valeur limite.
Un tel dépassement arrive régulièrement car si une consigne de direction n'emmenait jamais l'actionneur au delà de la valeur limite, la régulation ne permet pas de prévenir complètement l'action de l'effort de la route sur les pneus et donc sur la direction en virage. Cet effort physique était alors suffisant pour qu'en virage (notamment avec les deux trains de direction activés) la limite soit dépassée.
Après un bonne recherche dans le projet Simulink gérant le comportement du robuCAR, nous avons modifié le code et les blocs déterminants cette limite, et y avons intégré la prise en compte de la direction désirée par le conducteur. Ainsi les commandes de l'actionneur sont maintenant bloquées tant que la direction n'est pas changée, et l'on retrouve alors un comportement normal en sortie de virage.

De la même façon la gestion de la traction en mode marche arrière n'était pas correctement gérée et un bloc permettant le lissage de la consigne gardait les paramètres de traction en marche avant. Ce bloc, un "Rate Limiter Dynamic", se rapproche de la consigne pas à pas en fonction de l'état précédent et des limites hautes et basses. Il a l'avantage d'éviter les sauts de consigne sur les moteurs, mais les limites hautes et basses restaient les mêmes lors d'une marche arrière. Ainsi la limite basse négative qui permettait au robuCAR de s'arrêter en marche avant, le faisait rester en mouvement lors d'un freinage en marche arrière. Le même principe s'applique avec la limite haute, et entrainait le long délai entre l'appui sur la pédale et le démarrage en arrière.

Ces modifications qui n'étaient pas complexes à réaliser, ont tout de même mobilisé 2 à 3 semaines, afin de comprendre le fonctionnement du projet complet et la façon dont les choses ont été implémentées. La modification du projet Simulink devait aussi garder l'esprit initial du code, sans impacter le reste des fonctions, pouvant partager certaines variables.

4.2 Serveur PURE

Comme indiqué dans la section sur les outils de développement, le serveur PURE permet de contrôler les robots de robosoft. Mais étant donné que notre robot a été vendu avant la mise en place de ce protocole, et que le système de commande embarqué a été modifié, il n'est pas équipé de ce protocole.
Puisqu'un des objectifs de ce projet est de rendre compatible notre robuCAR avec des programmes destinés au robuTAINER, il faut donc que les commandes de notre architecture arrivent au robuCAR via un serveur PURE. Pour cela, un programme basique a été codé lors d'un précédent stage pour le laboratoire Cristal.

Cependant son implémentation a été faite de façon linéaire, et tout le processus de réception des paquets, test et réponse se fait dans le programme principal. Le programme ne gère que le service "car", et ne gère pas ou peu les services "Directory" et "Notifications". L'ajout de services y serait donc très laborieux, et pour la pérennité du projet, nous avons donc entrepris un re-codage de cette application. Celui-ci s'effectue en C++ sous Visual Studio, afin d'être exécuté sur le PC windows gérant la dSpace. En effet, il faut pouvoir, pour commander le robuCAR, modifier les cases mémoires contenant les valeurs consignes de vitesse et de direction par exemple. Pour cela, nous avons à notre disposition une bibliothèque : "Clib", couplé à un parseur en Python permettant de retrouver les zones mémoires.

Nous avons donc commencé à re-coder cette application, en utilisant une classe type définissant les services, et une classe fille pour chaque service que nous souhaitons, ou qui seront à ajouter. Ainsi, si un service est demandé, et que celui-ci n'est pas implémenté, la classe mère donnera une erreur sans pour autant interrompre le programme, et le service pourra être simplement développé par la suite. De plus, les classes "Directory" et "Notifications" joueront pleinement leur rôle, en indiquant la liste des services implémentés, et en gérant l'appel aux notifications sortantes de chaque service par exemple.

Cette partie n'est pas encore terminée, mais elle le sera dans les prochaines semaines, et permettra ainsi d'avoir une chaîne de transmission de l'information complète, entre notre architecture ROS et le robot, la partie ROS-Pure étant déjà codée et fonctionnelle (Réalisée lors d'une thèse sur le robuTAINER).

4.3 Laser et évitement d'obstacle

Nous avons extrait les données de notre laser SICK LMS221 sur un PC personnel. Il a juste fallu installer les drivers, connecter un convertisseur RS232-USB sur le PC et alimenter le laser avec 24 V continu. Ce laser est capable de voir de 10 cm à 81 m. Il effectue des mesures sur 180° tous les 1°. Il fonctionne à une vitesse de 38400 bauds.

FIGURE 8 – Notre laser avec son alimentation

A l'aide de rviz (logiciel de visualisation intégré à ROS), nous pouvons observer les différents points de mesure vus par le laser. Chacun des points représente un point de mesure du laser. Le laser se situe au centre de la map.

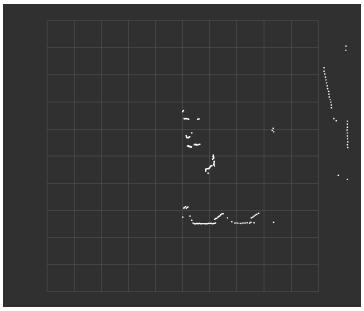


FIGURE 9 – Vue des données brutes du laser

Actuellement, il est prévu de réaliser un correcteur proportionnel pour éviter les obstacles. Si un obstacle se situe entre 2 m et 0,5 m, on diminue la vitesse proportionnellement. Si un obstacle est à moins de 50 cm, on stoppe le véhicule jusqu'à la disparition de l'obstacle. Ces distances sont paramétrables par l'utilisateur. De plus, l'utilisateur peut facilement paramétrer le nombre de points qu'il souhaite pour déterminer si l'obstacle est un obstacle ou non.

Après avoir récupérer les points de mesure, nous avons créé des obstacles. Les obstacles proches sont considérés comme dangereux, il faudra donc réduire la vitesse en leur présence voire s'arrêter totalement. Pour ceux qui ne sont pas dans la zone de danger, on peut considérer qu'ils ne sont pas prendre en compte. Nous avons réalisé une vidéo qui permet de visualiser les obstacles visibles par le laser. Elle est disponible à ce lien https://www.youtube.com/watch?v=9DunFgo-5iY .

4.4 Map

Afin de rendre le projet indépendant du lieu de travail, il nous faut pouvoir importer (ou réaliser nous-même puis importer) une carte du lieu en question. Étant donné qu'OpenStreetMap est open source et libre de droit, nous nous sommes donc orientés

vers cette solution. De plus, OSM encourage chaque internaute à réaliser les correctifs nécessaires pour avoir une carte libre du monde. Des données dans le monde entier sur les routes, les voies ferrées, les rivières, les forêts, les bâtiments et bien plus encore sont collectées. Les données cartographiques collectées sont ré-utilisables sous licence libre ODbL.

Certains modules de ROS permettent de convertir des fichiers avec l'extension OSM (fichiers exportés à l'aide de OpenStreetMap) en topics. Un utilisateur lambda peut facilement exporter une map de son choix avec son navigateur préféré. Pour cela, il n'a qu'à aller sur le site d'OSM, de cliquer sur le bouton "Exporter" et de sélectionner à l'aide du curseur la zone souhaitée. Cela génèrera un fichier map.osm.

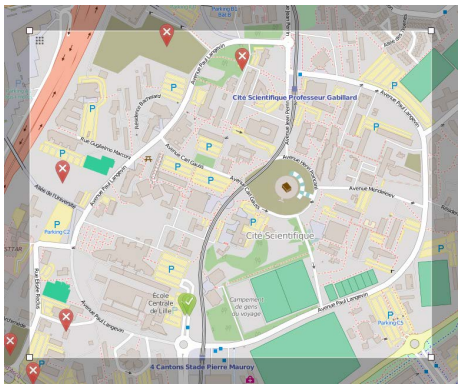


FIGURE 10 – Capture de l'interface d'exportation d'une map osm

Toutes les données brutes d'OSM (nœuds, voies, relations et propriétés/étiquettes) sont contenues dans un format XML. Nous avons pu assez rapidement visualiser une map du campus de Lille 1 avec rviz. Ci-dessous nous pouvons voir la visualisation avec rviz. De nombreuses informations ne nous sont pas utiles pour réaliser le path finding tels que les bâtiments, les chemins seulement accessibles aux piétons ou la ligne de métro. Il faudra donc pour la suite retirer ces données de notre map.

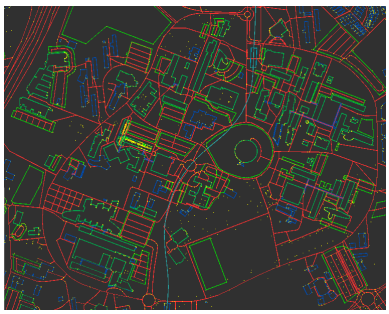


FIGURE 11 – Vue de la map exportée sous rviz

4.5 Path finding

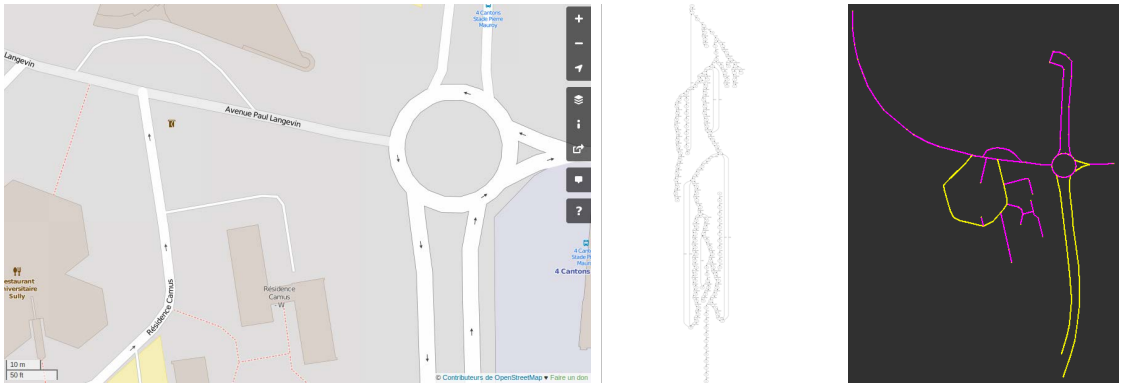
Bien que nous sachions visualiser n'importe quelle OpenStreetMap, les données qui transitent dans les topics ne permettent de réaliser la recherche de chemins. En effet, tout est envoyé sous forme d'UUID. Les UUID (Universally Unique IDentifier) permettent à des systèmes distribués d'identifier une information sans avoir besoin d'une très importante coordination centrale. Ils sont composés de 16 octets.

Nous avons donc cherché où la correspondance était faite entre les coordonnées GPS (WSG84) et les UUID. Du code Python permettait de réaliser cela, c'est pourquoi il a fallu apprendre quelques bases en Python pour comprendre le code et le modifier pour notre usage. Une fonction Python permet de convertir les UUID des points en coordonnées de projection. La projection en question est la Transverse Universelle de Mercator dans la zone 31. Ce qui permet donc d'avoir un système "rectangulaire" et métrique sur lequel on peut utiliser la géométrie de base pour se repérer, chercher un chemin, suivre un chemin, etc. De plus, les coordonnées UTM sont basées sur un système décimal alors que les coordonnées WSG84 sont basées sur un système sexagésimal, même s'il reste vrai qu'il est possible de travailler avec des degrés décimaux. Il faudra ne pas oublier que du coup la localisation réalisée sur la DSpace sera réalisée à l'aide des coordonnées UTM.

Il nous restait à présent à réaliser un graphe où chaque nœud correspond à un point et chaque arc correspond à un segment de route. A l'aide de la librairie boost, nous avons réalisé notre graphe et y avons appliqué l'algorithme A* pour rechercher rapidement le plus court chemin pour atteindre un objectif de notre choix. L'algorithme A* combine l'algorithme Dijkstra (on favorise les nœuds proches du point de départ) avec une heuristique (on favorise les nœuds proches de l'objectif). Dans la terminologie régulièrement utilisée, lorsqu'on parle de A*, g(n) représente le coût exact du chemin du point de départ jusqu'à n'importe quel point n. h(n) représente le coût estimé de l'heuristique du nœud n jusqu'à l'objectif. A chaque itération, A* examine le nœud n qui a le plus petit f(n) = g(n) +h(n).

Pour des raisons pragmatiques de visualisation et de calcul, nous avons travaillé sur une portion plus petite de la map. Ainsi, nous pouvons visualiser l'OSM, puis le graphe lié à notre map et la visualisation des routes sur rviz. Nous pouvons remarqué que sur rviz, nous avons les routes visibles jusqu'à ce qu'elles croisent d'autres routes. Cela est du au fait que chaque route dans OSM est une succession de points avec un point de début et un point de fin. Donc lors de l'importation, nous avons tous les points compris entre le point de début et le point de fin. Les sens uniques sont en jaune et les routes en double sens sont en violet. En observant la map sur rviz, nous pouvons distinguer deux problèmes :

- Les voies réservées aux piétons sont considérées comme des routes à part entière.
- Il manque des sens uniques : on ne peut prendre un rond-point que dans le sens trigonométrique



(a) zone exportée sous osm (b) représentation du graphe réalisé (c) vue du graphe sous rviz

FIGURE 12 – Les différentes représentations de notre graphe

En adaptant les modules présents dans ROS, nous avons réussi à ne récupérer que les routes destinées aux véhicules et à éliminer les voies réservées aux piétons et la ligne de métro encore présentes. De plus, pour pouvoir avoir une map correspondant à la réalité physique, il suffit juste d'éditer la map dans OpenStreetMap. En à peine 5 min, n'importe quel utilisateur peut créer ou modifier des routes de part l'interface très user-friendly. Au bout de quelques secondes, la map est chargée et il ne reste plus qu'à l'utilisateur de décider où il souhaite aller.

Le chemin est généré immédiatement, il n'y a aucun temps de latence. Il est visible en vert surbrillant. Dans le cas où aucun chemin ne peut être trouvé, aucun chemin n'est affiché et un message indique qu'il est impossible d'atteindre le point souhaité. Il faut à présent que la communication entre la dSpace et ROS (serveur PURE) soit fonctionnelle pour envoyer les coordonnées GPS du véhicule et pour pouvoir réaliser un path finding dynamique. Une vidéo permettant de visualiser la recherche de chemin est disponible à ce lien : https://www.youtube.com/watch?v=U97S_Z1HDLU.

Ci-contre, nous pouvons voir le chemin généré pour aller de Polytech à la résidence Galois.

FIGURE 13 – Visualisation du chemin généré sous rviz

5 Prochaines étapes

5.1 Localisation

Actuellement, la localisation prend en compte les données GPS et odométriques. Dans le cas où aucun signal GPS n'est capté, le robuCAR doit pouvoir continuer à se localiser. Un filtre de Kalman étendu est appliqué pour corriger la dérive odométrique. Les coordonnées GPS/WGS84 sont converties en coordonnées planes Lambert. Il faudra vérifier qu'elle fonctionne correctement et y appliquer des correctifs si nécessaire. De plus, il faudra réaliser la conversion des données GPS en coordonnées UTM (Transverse Universelle de Mercator).

5.2 GPS-Odométrie

Théoriquement, la réception des données GPS et odométriques est fonctionnelle. Cependant, à cause des nombreux bâtiments et arbres, il est des fois impossible de recevoir des données GPS, c'est pourquoi l'utilisation d'une balise RTK se révèlera nécessaire. Par ailleurs, le robuCAR, ayant des défauts dans la conception de sa direction et de sa traction, nous donne des données odométriques très imprécises, d'où l'utilisation de l'EKF pour la localisation et de la balise RTK pour le récepteur GPS.

5.3 Suivi de chemin

Cette partie concentrera la majeure partie du reste du projet. En effet, nous avons à présent un chemin global à suivre mais nous n'avons que très peu d'informations du point de vue local. Les lignes de parking ainsi que les lignes au centre de la route, lorsqu'elles sont présentes, pourraient permettre de pouvoir se localiser à l'aide d'une caméra. En outre, nous avons réfléchi à une autre approche. Avec le laser positionné et orienté d'une façon connue, nous pourrions essayer de détecter les bordures de trottoirs pour nous orienter. Enfin, rien ne nous garantir que les données GPS que nous recevrons seront en parfaite adéquation avec la localisation du véhicule sur la map.

5.4 IHM et convertisseur

L'interface homme machine se fera à l'aide de rqt (logiciel intégré à ROS contenant de nombreux outils GUI sous forme de plugins). rqt permet de visualiser plusieurs fenêtres notamment celle de rviz où nous verrons la map. On y ajoutera plusieurs boutons permettant de directement aller dans des lieux connus sans devoir chercher où c'est sur la map. Pour cela, un noeud jouera le rôle de convertisseur pour que la recherche de chemin puisse comprendre l'instruction. Un bouton permettant d'activer ou non la marche en mode manuelle sera disponible.

6 Conclusion

Lorsque la communication entre la dSpace et le PC externe sera fonctionnelle, ie que le serveur soit en place, il pourra être aisé de tester puis de valider la réception des données GPS et odométriques et la localisation. En effet, nous sommes à présent familier avec les outils de développement utilisés. Cela ne devrait prendre qu'environ deux semaines normalement. L'essentiel du travail sera ensuite consacré au suivi de chemin car l'IHM sera rapidement réalisée grâce à rqt. A ce stade, nous estimons que nous sommes à mi-chemin du PFE.

Le schéma suivant permet de récapituler le travail accompli (en vert), en cours (orange) et à faire (rouge).

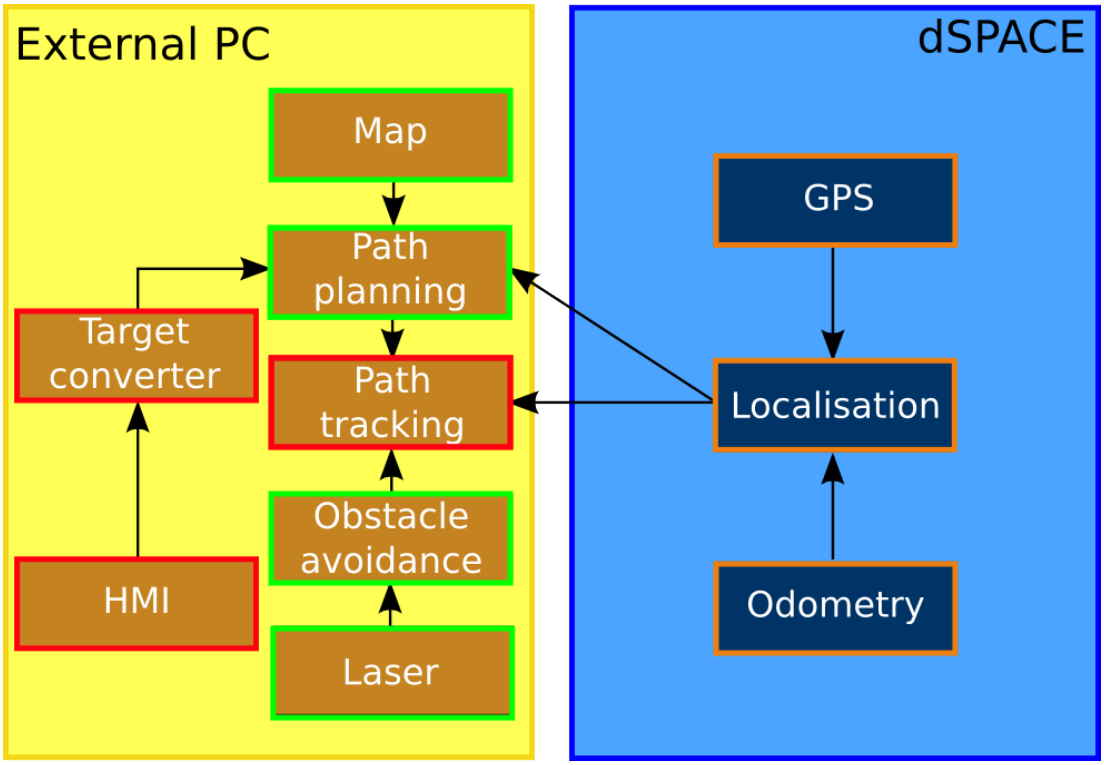


FIGURE 14 – Travail restant

Table des figures

1 robuCAR dans le hall de Polytech Lille 3
2 Schéma représentant le robuTAINER 4
3 Différence de taille entre le robuTAINER et le robuCAR 5
4 Stanley, vainqueur du DARPA Grand Challenge 2005 5
5 Vue globale de l'architecture du système 7
6 Un des blocs Simulink que nous avons eu à modifier 9
7 Schéma d'un échange PURE type 10
8 Notre laser avec son alimentation 13
9 Vue des données brutes du laser 13
10 Capture de l'interface d'exportation d'une map osm 14
11 Vue de la map exportée sous rviz 14
12 Les différentes représentations de notre graphe 16
13 Visualisation du chemin généré sous rviz 16
14 Travail restant 18