Smagghe Cyril
Tournier Jean-Michel

Rapport de projet : Vers un véhicule autonome

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

Réalisé du 21/09/2015 au 25/02/2016

Table des matières

Remerciements

Nous tenons à remercier M. Merzouki Rochdi et M. Coelen Vincent de nous avoir proposé ce PFE à la fois enrichissant et multidisciplinaire. Plus particulièrement, nous remercions l'expertise de M. Coelen qui a permis 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 ainsi que l'installation de la balise 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 interaction humaine directe dans le campus universitaire de Lille 1. Ce PFE s'inscrit dans le cadre du projet industriel InTraDE.

Ce rapport va donc introduire au lecteur les différents aspects de ce projet ainsi que les divers outils de développement à disposition qui ont permis au projet d'aboutir. Le travail accompli sera ensuite développé dans le rapport.

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.

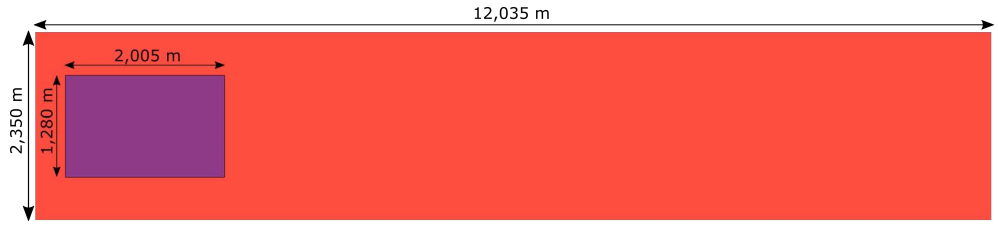


FIGURE 3 – Différence de taille entre le robuTAINER et le robuCAR

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 ces 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.

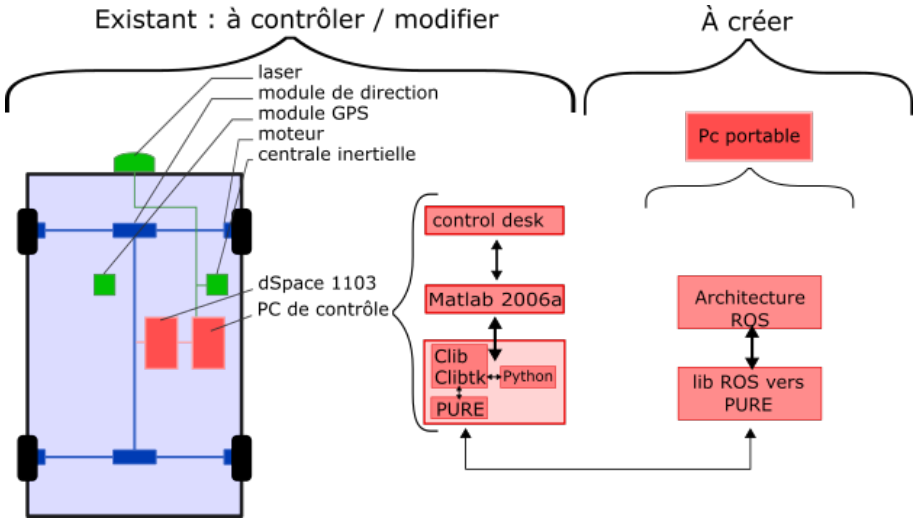


FIGURE 5 – Vue globale de l'architecture du système

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.

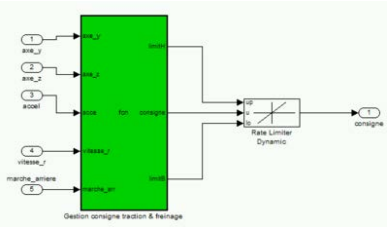


FIGURE 6 – Un des blocs Simulink que nous avons eu à modifier

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.

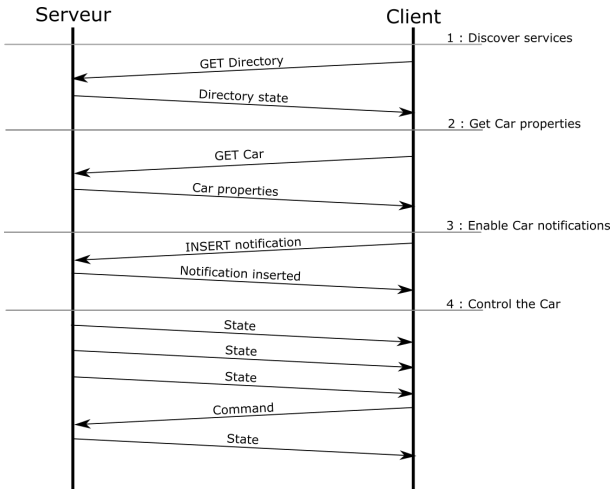


FIGURE 7 – Schéma d'un échange PURE type

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 re-codé 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. Le tout sera « encadré » par une classe de gestion UDP, qui aura pour tâche de recevoir et d'envoyer les paquets. Cette classe devra aussi, après une réception, retrouver à quel service est destiné le paquet et le lui transmettre.

Nous allons donc détailler les diverses classes développées et leurs fonctionnements, ainsi que leurs utilisations et la méthode pour créer de nouveaux services.

4.2.1 La classe Service

La classe service va servir de prototype pour tous les autres services. Comme indiqué précédemment, elle doit être incluse comme classe mère par les autres services et possède donc les définitions de toutes les fonctions représentant toutes les requêtes ou notifications (entrantes ou sortantes) possibles par le protocole PURE.

Le header contient donc la définition de cette classe avec en éléments protected (afin de pouvoir être hérité par les services fils) :

- Deux shorts pour contenir le type du service et le numéro d'instance et un string pour contenir une description textuelle du service. Ces 3 éléments doivent être initialisés par le constructeur du service fils (puisque différents à chaque instance de service).
- Deux éléments de type shared pointer, ici crée avec la bibliothèque boost car non disponible en std sous visual studio 2005. Ces deux shared pointeur vont contenir un pointeur vers l'objet ClibTk instancié à l'ouverture du serveur (qui permettra d'accéder à des variables de la dSpace) et un pointeur sur un objet de la classe udpManager, aussi instancié au démarrage du serveur et dont nous parlerons en section suivante.

```
class Service
{
  protected:
    short m_type; //MUST BE initialized with the soon constructor
    short m_instanceN; //MUST BE initialized with the soon constructor
    std::string m_description ; //MUST BE initialized with the soon
    constructor

    std::tr1::shared_ptr<ClibTk> m_robucar;
    std::tr1::shared_ptr<udpManager> m_udpCom;

    void actionNotSupported(char &identifier, char action);
```

On retrouve ensuite en public, le constructeur qui doit recevoir les shared pointer évoqués précédemment, quelques accesseurs et les prototypes des différentes fonctions, accessibles par le protocole PURE. Ces fonctions sont ici définies en virtual, et avec un code retournant un message d'erreur si jamais ce code ce retrouve appelé. Ainsi, si la fonction n'a pas été redéfinie dans le code du service, celui de ce prototype sera exécuté et permettra d'avertir l'utilisateur de cet appel, sans bloquer ou arrêter le fonctionnement du serveur.

```
1   public:
2
3
4     Service(std::tr1::shared_ptr<udpManager> udp, std::tr1::shared_ptr<ClibTk>
      robucarClib);
5
6     //Accessors for the directory service
7     short getType();
8     short getInstanceN();
9     std::string getDescription();
10    //Virtual definition of all the possible request and notif of a service
11    virtual void getRequest(char &identifier, Buffer& buffer)
12            {actionNotSupported(identifier, char(GET_CODE));}
13
14    virtual void queryRequest(char &identifier, Buffer& buffer)
15            {actionNotSupported(identifier, char(QUERY_CODE));}
16
17    virtual void insertRequest(char &identifier, Buffer& buffer)
18            {actionNotSupported(identifier, char(INSERT_CODE));}
19
20    virtual void deleteRequest(char &identifier, Buffer &buffer)
21            {actionNotSupported(identifier, char(DELETE_CODE));}
22
23    virtual void replaceRequest(char &identifier, Buffer &buffer)
24            {actionNotSupported(identifier, char(REPLACE_CODE));}
25
26    virtual void updateRequest(char &identifier, Buffer &buffer)
27            {actionNotSupported(identifier, char(UPDATE_CODE));}
28
29    virtual void inboundNotif(Buffer &buffer)
30            {std::cout<<"Instance n: "<< m_instanceN
31            << ", inboundNotif not implemented"<<std::endl;}
32    virtual void outboundNotif()
33            {std::cout<<"Instance n: "<< m_instanceN
34            << ", outboundNotif not implemented"<<std::endl;}
35 };
```

Enfin, dans le header on trouve aussi 3 enums, définissant tous les codes d'erreurs ou de retour du serveur PURE. Ainsi, ils seront accessible par tous les services et classes incluant ce header.

```
enum ACTION
{
  GET_CODE = 0x00,
  QUERY_CODE = 0x01,
  REPLACE_CODE = 0x02,
  UPDATE_CODE = 0x03,
  INSERT_CODE = 0x04,
  DELETE_CODE = 0X05
};
```

```
enum SERVICE
{
  DIRECTORY_CODE = 0x0000,
  NOTIFICATION_CODE = 0x0001,
  DIAGNOSTIC_CODE = 0x0002,
  LOGGER_CODE = 0x0003,
  ARM_CODE = 0xA002,
  BATTERY_CODE = 0x400D,
  CAR_CODE = 0x4003,
  DIFFERENTIAL_CODE = 0x4005,
  DRIVE_CODE = 0x4009,
  ENCODER_CODE = 0x400A,
  GPS_CODE = 0x400C,
  GYROSCOPE_CODE = 0x400E,
  I_O_CODE = 0x4001,
  LASER_CODE = 0x4004,
  LOCALIZATION_CODE = 0x8002,
  STEP_CODE = 0x8003,
  TELEMETER_CODE = 0x4008,
  TRAJECTORY_CODE = 0x8001
};
```

```
enum RESULT
{
  SUCCESS_CODE = 0x00,
  UNKNOWNTARGET_CODE = 0x01,
  ACTIONNOTSUPPORTED_CODE = 0x02,
  UNKNOWNACTION_CODE = 0x03,
  INVALIDLENGHT_CODE = 0x04,
  INVALIDDATA_CODE = 0x05
};
```

Dans le fichier de code, on ne retrouve uniquement que les définitions du constructeur, qui se contente de ne faire que la recopie des deux pointeurs, les 3 accesseurs et la fonction actionNotSupported, qui effectue l'affichage et retourne au client un message d'erreur.

4.2.2 La classe udpManager

Comme évoqué dans la classe Service, chaque objet possède un pointeur sur un objet de cette classe. Cela permettra à tous les différents services d'effectuer des envois de paquets.

On retrouve dans le header de cette classe, en private :

- quelques définitions de fonctions internes
- les variables permettant d'initialiser et de contrôler la socket (ouverte avec winsock2)
- un pointeur sur un objet de type CriticalSection, afin d'éviter tout conflit si le serveur fonctionne avec des threads
- un shared pointer sur un vecteur de Service*

Ce vecteur de pointeur de services, permettra à cette classe d'accéder aux différents services initialisés et donc d'appeler leurs fonctions lors de la réception de paquets leur étant destinés. Puisque les numéros d'instance se suivent et débutent à 0, l'accès au service voulu se fait donc très simplement, en allant chercher le ième élément du vecteur, avec i son numéro d'instance.

Dans le code de cette classe, on retrouve donc le constructeur, qui récupère en paramètre l'adresse et le port sur lesquels le serveur doit écouter, ainsi qu'une valeur de timeOut et l'objet CriticalSection. Il initialisera la socket, puis l'ouvrira, exécutera le bind et sauvegardera la valeur de timeOut et le CriticalSection dans les variables prévues à cet effet. La fonction setService permet quant à elle de recevoir et d'enregistrer la liste de service tel que décrite précédemment, une fois que celle-ci aura été initialisée.

La fonction receive lance un select sur la socket, afin de détecter une réception de données. C'est ici qu'intervient la variable de timeOut, qui permettra d'interrompre cette fonction si aucun paquet n'a été reçu dans le temps imparti. La fonction receive n'est donc pas bloquante, chose très importante, puisque l'utilisation de thread pour la partie émission ne semble pas fonctionner. Voir en section 5.4 pour plus d'informations.

Lorsqu'un paquet est reçu, celui-ci, si sa taille est convenable, est sauvegardé dans un buffer puis décodé. Son identifier (valeur choisie par le client et à réutiliser pour la réponse) est dé-

pilé, et l'on passe ce buffer et l'identifier à la fonction notifExec s'il s'agit d'une notification entrante, ou à requestExec s'il s'agit d'une action. Ces deux fonction appelleront donc la fonction inboundNotif ou celle correspondant à l'action du code contenu dans le paquet, en utilisant le vecteur de Service* ainsi : `(*m_services)[target]->getRequest(identifier,bufferIn);` Si jamais le code de l'action passé dans le paquet, ou le service cible n'est pas connu ou pas décodé, alors les fonctions unknownTarget et unknownAction se chargeront de renvoyer un paquet au client avec un code d'erreur adapté, ainsi que d'informer l'utilisateur sur la sortie standard.

La fonction send, utilisée par tous les services se contente de prendre en paramètre un objet de type Buffer, de le recopier dans un tableau de char, et de l'envoyer via la socket. Si une erreur a lieu, l'utilisateur sera averti.

Enfin la fonction closeCleanup permet comme son nom l'indique, de fermer proprement la socket et l'api Windows.

4.2.3 La classe Directory

Le service PURE Directory, supposé lister et donner quelques informations sur les services présents sur le robot est donc bien sur représenté par une classe, héritant de la classe service et donc de tous ses paramètres. Comme la classe udpManager, elle possède elle aussi un shared pointer d'un vecteur de Service*, initialisé par la fonction setServices identique à celle d'udpManager.

Son constructeur récupère les pointeurs sur l'objet udpManager et ClibTk, et les passe au constructeur de la classe mère : Service. Il instancie aussi les variables de type, de numéro d'instance et de description textuelle, avec les données spécifiques au service Directory (le numéro d'instance est aussi passé en paramètre).

La fonction getRequest est redéfinie, afin que le service Directory retourne le code de type et le numéro d'instance de tous les services initialisés. Pour cela, une boucle for est lancée sur la taille du vecteur de Service, et remplit le buffer de réponse des types et des numéros d'instances grâce aux accesseurs prévus dans la classe Service. Le buffer de réponse est ensuite envoyé, via l'objet udpManager.

La fonction queryRequest est elle aussi redéfinie, afin de renvoyer le champ de description textuel d'un service en particulier. Pour cela, elle dépile le buffer entrant afin de récupérer le numéro d'instance voulu, puis récupère le string grâce à la liste de service et à l'accesseur. Ensuite, le string est converti en tableau de char, afin d'être ajouté au buffer de sortie pour être envoyé. Dans l'éventualité où les données passées dans le buffer d'entrée ne contiendraient pas un numéro d'instance, ou que celui n'existerait pas, la fonction renverra alors au client un paquet avec le code d'erreur approprié.

4.2.4 La classe Notification

Comme pour la classe Directory, la classe Notification hérite de la classe mère service et possède le même vecteur de Service*. En effet, afin de jouer pleinement son rôle, elle doit retenir pour quels services les notifications sortantes ont été activées, et pouvoir appeler la fonction outboundNotif de chacun de ces services.

Dans le header de cette classe, on trouve une struct : NotificationEntry. Elle permet

de stocker le couple numéro de service, et fréquence désirée, tel que spécifié dans le manuel PURE. Cette struct possède donc un numéro d'instance, ainsi qu'un unsigned char contenant la valeur de la fréquence reçue. La struct possède aussi un constructeur afin de pouvoir créer des éléments aisément, ainsi qu'une redéfinition de l'opérateur « < ». Celle-ci permet l'utilisation d'un set de ces objets, en ne comparant que le champ serviceInstance. Ainsi on pourra gérer les abonnements sous la forme d'une liste triée ou les ajouts, recherches et suppressions seront aisés.

```
1  struct NotificationEntry
2  {
3    unsigned short serviceInstance;
4    unsigned char notifMode; //0 = on change, 1-255 = period
5    NotificationEntry(){}
6    NotificationEntry(short instanceN, char mode):
7      serviceInstance(instanceN), notifMode(mode){}
8
9    bool operator < (const NotificationEntry &other) const
10     { return serviceInstance < other.serviceInstance; }
11
12 };
```

La classe Notification possède donc un objet std : :set de NotificationEntry, ainsi qu'un pointeur d'objet CriticalSection. Celui-ci permet d'éviter tout problème si la réception et réponse au paquet entrant se fait parallèlement aux envois de notifications.

Les fonctions redéfinies ici sont les fonctions getRequest, insertRequest, et deleteRequest qui permettent respectivement de lister les abonnements actifs, d'en ajouter ou d'en supprimer.

La fonction insertRequest récupère donc le buffer reçu, vérifie que celui-ci est bien de taille suffisante et que le numéro d'instance à ajouter à la liste d'abonnements n'est pas trop élevé puis crée un nouvel objet NotificationEntry. Si la même instance de service ce trouvait déjà dans liste (recherche avec iterator), un message est affiché, et un paquet avec code d'erreur et renvoyé. Sinon le nouvel élément est ajouté et un message avec un code de succès est renvoyé.

De même avec deleteRequest, après vérification de la taille et de la validité du numéro d'instance, un nouvel élément est créé, sans que le champ de fréquence ne soit rempli et une recherche avec iterator est lancée. Si celle ci aboutie, alors l'entrée est supprimée et un code de succès et retourné. Sinon, comme précédemment, un message est affiché et un code d'erreur est renvoyé.

Enfin, la fonction getRequest se contente uniquement d'ajouter via un for sur la taille du set contenant les abonnements, les deux champs de chacune des entrées au buffer de réponse.

En plus d'une fonction pour l'affichage des abonnements en cours, une autre fonction a été ajoutée. callOutboundNotifs permet d'appeler successivement les fonctions outboundNotif de chacun des services ajoutés à la liste. Afin de gérer les différentes possibilités de fréquences de ces notifications, un compteur (callCount) s'incrémentant à chaque appel de la fonction est créé. En effectuant une opération de modulo entre le compteur et la valeur demandée on obtient alors une valeur à tester pour savoir si l'on doit ou non faire l'appel à l'outboundNotif de se service. L'implémentation de la boucle peut donc ce faire ainsi :

```
1  for (it=m_notifEntryList.begin(); it!=m_notifEntryList.end(); ++it)
2  {
3    if((callCount % it->notifMode)==0)
4      (*m_services)[it->serviceInstance]->outboundNotif();
5  }
```

4.2.5 La classe Car

La classe Car est un service plus standard comparé au services Directory et Notification. Celui-ci hérite donc uniquement des données contenues par la classe mère service, et son constructeur ne fait que passer les deux pointeurs au constructeur Service, ainsi qu'initialiser les champs d'informations.

Le header de ce service possède 3 structures : CarProperties, CarState et CarCommand. Ces structures, une fois instanciés et contenus en private dans la classe vont permettre de contenir les données propres au véhicule. L'objet CarProperties est initialisé par son propre constructeur et ne contient que des données d'information concernant la physique du véhicule. CarState contient les informations sur l'état actuel du véhicule, informations qui doivent être rafraichies avec un appel à la fonction updateState. De même, CarCommand contient les données de commande reçues par le serveur qui sont ensuite passées à la dSpace via la fonction writeCommand.

Ces deux fonctions, updateState et writeCommand utilisent l'objet ClibTk contenu par la classe mère Service, afin effectuer respectivement des read et des write sur des cases mémoires de la dSpace. Ces deux fonctions sont peu développées ici, car il s'agit simplement d'utiliser une blibliothèque déjà codée auparavant , et dont l'utilisation est assez simple.

Les fonctions redéfinies ici sont getRequest, qui recopie les données contenues par CarProperties dans le buffer de sortie, inboundNotif, qui enregistre les données de commande reçues dans CarCommand, puis effectue un appel à writeState et enfin outboundNotif, qui envoie les valeurs contenues par CarState après un updateState.

4.2.6 La classe Localization

Comme la classe Car, la classe Localization est un service simple qui repose sur le même principe. Une struct LocalizationState est présente dans le header, et une fonction updateState permet aussi de mettre à jour ses valeurs.

Les fonctions getRequest et outboundNotif redéfinies ont le même comportement : retourner l'état du service après un updateState. Les fonctions updateRequest et replaceRequest ont aussi été redéfinies, mais retournent uniquement un message sur la sortie standard. Leur comportement devrait permettre de transmettre au filtre de Kalman des données de localisation supplémentaire, ou d'effectuer une remise à zéro de son calcul. Mais ces fonctions n'étant pas disponibles sur le filtre que nous avons implémenté dans la dSpace, elles n'ont pas de raison de l'être ici.

4.2.7 Fonctionnement global

Le schéma UML suivant permet de visualiser les différentes classes abordées précédemment.

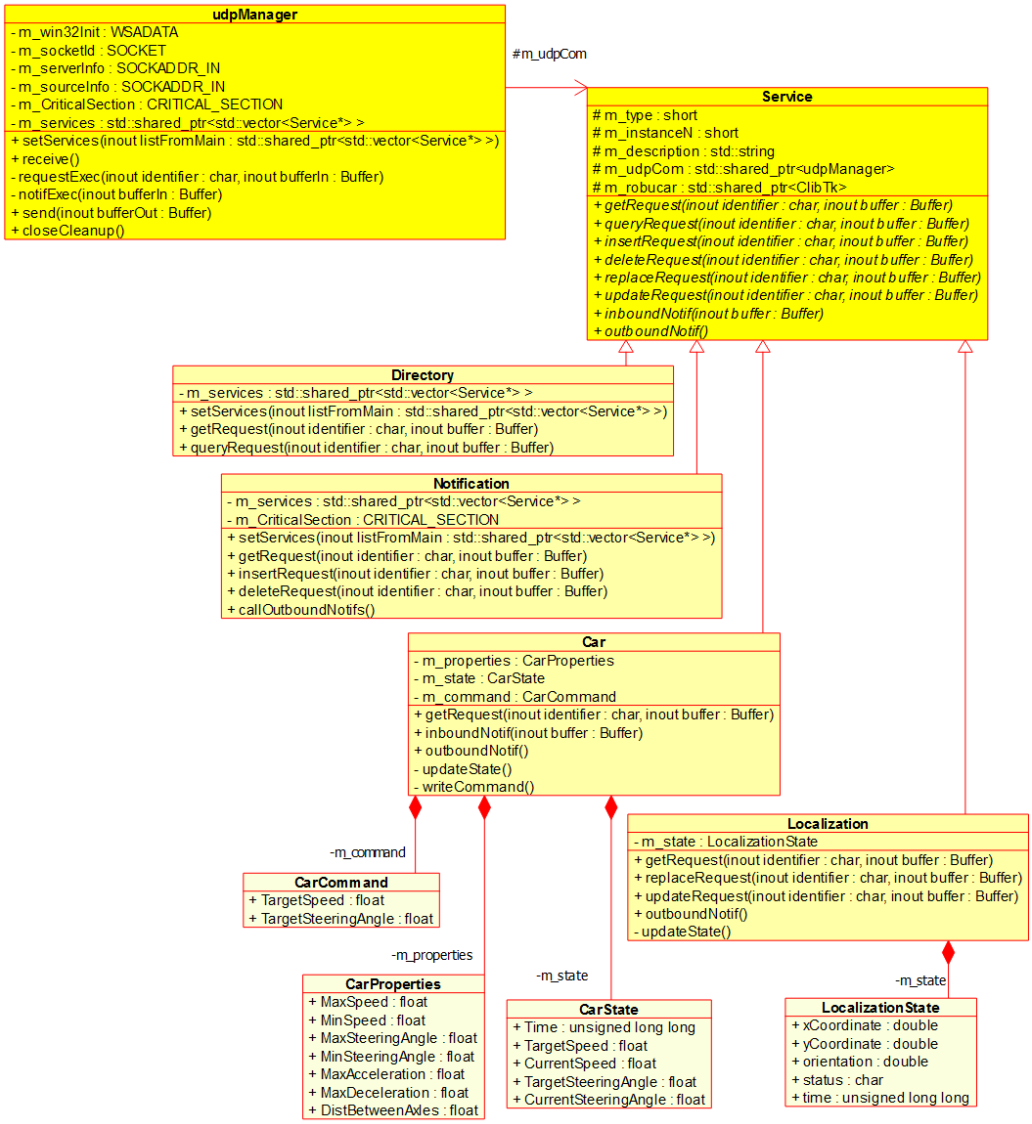


FIGURE 8 – Schéma UML des classes du serveur PURE

Le main inclut chacun des headers de nos classes précédemment vues.

Après avoir créé et initialisé l'objet CriticalSection, on crée dans un shared pointer un objet de type ClibTk, puis on effectue un appel à la fonction load afin de charger notre fichier de config. On fait de même pour un objet udpManager, en lui passant l'adresse et le port d'écoute, le timeOut en microsecondes et l'objet CriticalSection.

On peut ensuite construire le vecteur de Service, et le remplir en effectuant des push-back avec de nouveaux objets. Il faut cependant faire attention à ne pas faire de new pour les services Directory et Notification. En effet il faut les créer avant afin d'avoir toujours accès à leurs fonctions spécifiques et notamment setServices. En effet une fois dans le vecteur, les objets étant castés en Service, on ne peut plus accéder aux fonctions autres que définies dans la classe mère.

Une fois le vecteur remplit, on peut le passer aux objets udpManager, Directory et Notification via setServices, ce qui termine nos initialisations. Une boucle while fera alors alterner

les receive de l'objet udpManager avec les callOutboundNotifs de Notification. La fonction receive possédant un timeOut, la fréquence d'émission des notifications est bien fixe, et si le client effectue une régulation avec réception d'état / commande, le serveur restera synchronisé sur sa fréquence. Pour quitter la boucle while un appui sur n'importe quelle touche de clavier est possible, la condition de sortie étant : `while(!_kbhit())`
fonction intégrée dans <conio.h> sous Windows.

On va alors supprimer l'objet CriticalSection, puis exécuter la fonction close de ClibTk et enfin le closeCleanup de l'udpManager.

4.2.8 Ajout d'un nouveau service

Afin d'ajouter un nouveau service il faut ajouter dans son header, le fichier service.h, et ajouter « : public Service » à sa définition de classe. Il est conseillé d'utiliser une structure pour contenir les informations d'état ou de commande, comme c'est le cas pour les services Car et Localization, avec des fonctions utilisant l'objet ClibTk pour mettre ces structures à jour.

Le fichier cpp doit inclure le fichier udpManager.h, afin d'avoir accès à la fonction send de notre udpManager, et le constructeur doit repasser les pointeurs aux constructeur de service, et définir les champs de description, fait ici avec le service Car.

```
1 #include "car.h"
2 #include "udpManager.h"
3 Car::Car(int instanceN, std::tr1::shared_ptr<udpManager> udp,
4       std::tr1::shared_ptr<ClibTk> robucarClib):Service(udp,robucarClib)
5 {
6   m_type=CAR_CODE;
7   m_instanceN=instanceN;
8   m_description="car service, no more info";
9 }
```

Une fois cela fait, il suffit de redéfinir et de compléter le code des fonctions correspondant aux actions nécessaires au bon fonctionnement du service.

4.2.9 Implémentation dans le projet Simulink

Nous avons aussi modifié la façon dont les données étaient envoyés à la dSpace. À l'origine, la bibliothèque ClibTk allait écrire les valeurs de commande dans deux blocs de type constant, suivis d'un switch permettant de basculer entre les valeurs de commande manuelle ou celles venant du serveur. La conversion des unités utilisées par PURE (mètres par seconde et radians par seconde) en unités utilisées dans le projet Simulink (pourcentage de puissance moteur et données de capteur de directions) était effectuée par le serveur. Afin d'améliorer la compréhension du code du serveur, nous avons choisi de modifier ces blocs, et d'effectuer la conversion par la dSpace. Le bloc de commande de la direction se voit alors ajouter un bloc, qui permet de séparer la commande d'angle en radian passé par PURE en deux commandes utilisables par la régulation PID qui s'effectue ensuite. La capture suivante permet donc d'observer ce bloc chargé de la division, et de la conversion de la commande d'angle. Si l'on souhaite aussi que le robuCAR ne soit commandé en direction que sur le train avant, c'est donc sur ce bloc qu'il faut jouer.

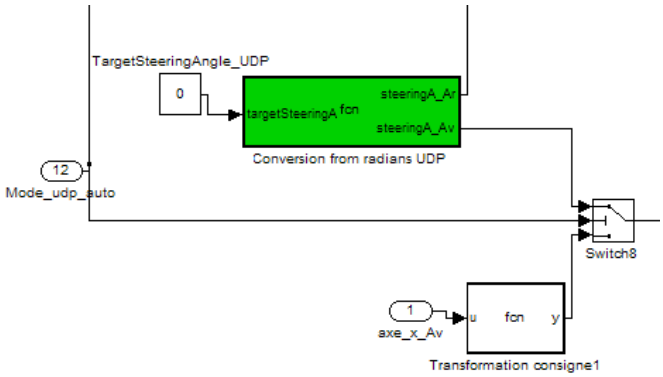


FIGURE 9 – Bloc de conversion de commande de direction.

Pour la commande de traction, celle-ci a été déplacée. Elle était à l'origine placée après le bloc de consigne de traction et freinage, que nous avions modifié en début de projet. Cette valeur ne bénéficiait alors pas du bloc rate limiter dynamic, ce qui résultait en une application brutale de la consigne de traction. Nous avons donc aussi créé un bloc emmbedded, qui convertit la consigne reçue par udp avant de l'appliquer dans le rate limiter dynamic. Ce bloc effectue la conversion en pourcentage,en considérant une vitesse maximale de 5m/s, et en limitant la commande à 80% maximum.

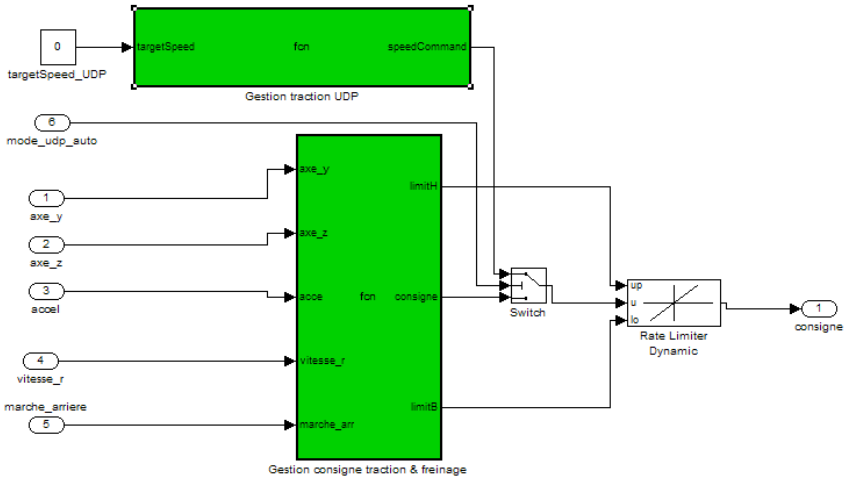


FIGURE 10 – Bloc de conversion de commande de traction.

Pour changer le mode de contrôle, une variable modifiée par deux boutons de Control Desk permet de basculer en mode UDP ou commande manuelle. Néanmoins cliquer sur ce bouton lorsque le véhicule est en déplacement autonome et que l'on souhaite l'arrêter rapidement n'est pas une chose aisée. Nous avons donc créé un bloc embedded, qui se place après le bloc attaqué par Control Desk et vient modifier le mode de contrôle, en prenant en compte la pédale de frein. Ainsi, si le robuCAR est en mode autonome et que la pédale de frein vient à être appuyé, le robucar repasse alors en mode manuel, et ce jusqu'à ce qu'un appui successif sur les boutons mode manuel puis mode UDP soit effectué. Cela permet d'interrompre et de

reprendre plus facilement les tests, car lorsqu'un arrêt d'urgence est enclenché, le redémarrage n'est pas toujours évident.

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 11 – 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.

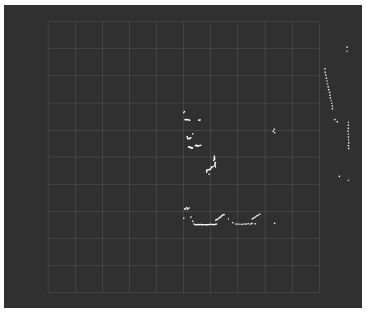


FIGURE 12 – Vue des données brutes du laser

Actuellement, le robuCAR s'arrête lorsqu'il y a au moins un obstacle dans la zone de danger. Le code est suffisamment modulaire pour permettre par la suite de réaliser un correcteur proportionnel pour éviter les obstacles. Si un obstacle se situe entre 3 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

L'illustration suivante montre les données visibles sous rviz, avec le robuCAR dans l'allée de parking de Polytech.

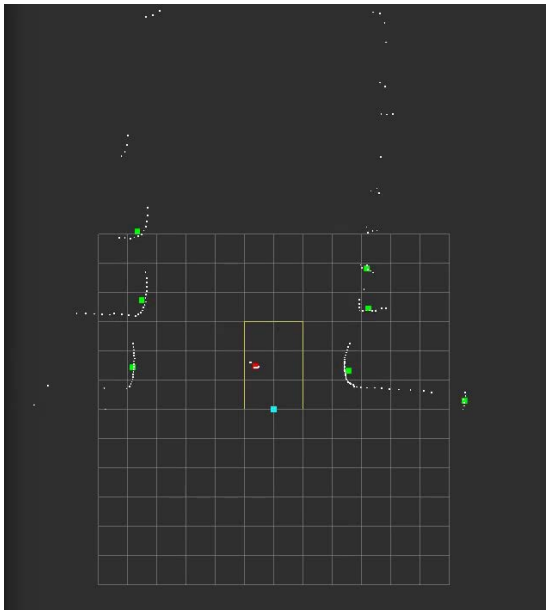


FIGURE 13 – Données laser en conditions réelles

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.

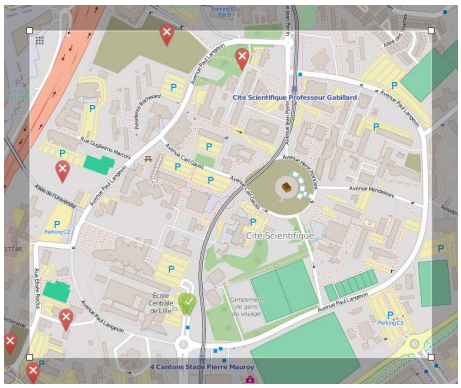


FIGURE 14 – Capture de l'interface d'exportation d'une map osm

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.

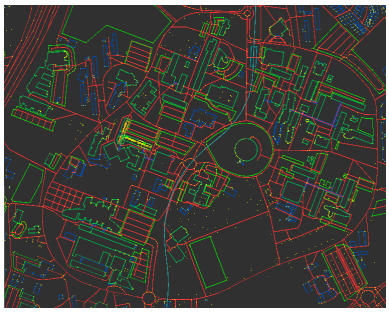


FIGURE 15 – Vue de la map exportée sous rviz

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.

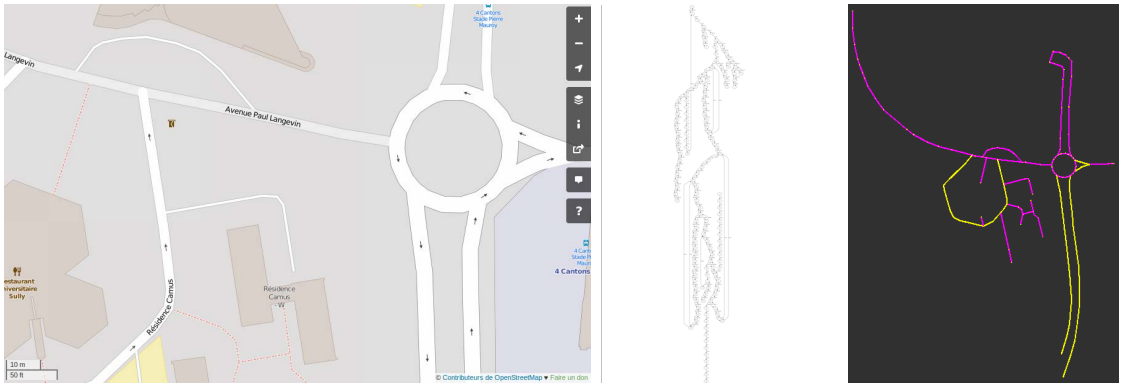
Nous avons donc cherché où la correspondance était faite entre les coordonnées GPS (WSG84) et les UUID. Du code Python permettait de réaliser cela, c'est pourquoi il a fallu apprendre quelques bases en Python pour comprendre le code et le modifier pour notre usage. Une fonction Python permet de convertir 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 16 – 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. Une vidéo permettant de visualiser la recherche de chemin est disponible à ce lien .

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

FIGURE 17 – Visualisation du chemin généré sous rviz

Afin de rendre la recherche de chemin dynamique, le noeud récupère les données de localisation. Ainsi, nous sélectionnons le point le plus proche des coordonnées actuelles du véhicule pour commencer la recherche de chemin. Cependant, nous nous sommes aperçu que ce critère de sélection de début de chemin (et par extension de fin de chemin) n'était pas viable. C'est pourquoi nous avons changé le critère de début et de fin de recherche de chemin. A présent, le début de recherche de chemin s'effectue sur le projeté orthogonal du point où est localisé le véhicule sur le segment le plus proche. De même, pour la fin de recherche de chemin, on ne prend plus le point le plus proche mais le projeté sur le segment le plus proche.

4.6 GPS-Odométrie

La réception des données GPS se faisait auparavant sur le PC externe. Pour limiter les retards occasionnés par les transmissions de données. La lecture des trames GPS est réalisée à présent sur la dSpace car toute la localisation est réalisée sur la dSpace. Nous sommes parvenus facilement à convertir les coordonnées WSG84 (latitude et longitude) en coordonnées UTM (x et y).

Nous avons modifié le programme Simulink afin de pouvoir récupérer les données GPS à travers le port RS232 de la dSpace. Un parser a été réalisé afin de pouvoir interpréter les différentes trames GPS (GPA et ZDA). Après quelques tests, nous avons été heurtés à différents problèmes notamment le fait qu'il faille travailler au plus bas niveau possible, c'est-à-dire récupérer les octets un à un puis de convertir les données grâce au tableau de correspondance ASCII. Après avoir modifier les paramètres de différents blocs Simulink, nous sommes arrivés à recevoir les données GPS trame par trame. Auparavant, on les recevait octet par octet et du coup nous étions incapables de récupérer une trame complète ou alors il aurait fallu augmenter la fréquence de fonctionnement de la dSpace. Avant de passer la trame GPS au parser, nous utilisons un bloc permettant de vérifier si la trame ne comporte pas d'erreur

car il y a régulièrement des erreurs de transmission. Hors le parser ne peut se permettre de recevoir des trames invalides, sinon nous obtenons des données erronées.

Étant donné que les arbres n'étaient plus couverts de feuilles, nous étions donc visibles par de nombreux satellites (14 et plus). Ce qui nous garantissait une précision décimétriques.

4.7 Localisation

Le travail réalisé précédemment permettait à la localisation de prendre en compte les données GPS et odométriques. Un filtre de Kalman étendu était appliqué pour corriger la dérive odométrique. Les coordonnées GPS/WGS84 étaient converties en coordonnées planes Lambert.

Le filtre de Kalman est une méthode permettant d'estimer des paramètres d'un système évoluant dans le temps à partir de mesures bruitées. Ce filtre peut permettre de prédire et de corriger les erreurs des capteurs. Il fonctionne en deux étapes : une étape de prédiction et une étape de mise à jour. L'étape de prédiction reprend l'estimation de l'étape précédente des paramètres et de l'erreur et détermine les nouveaux paramètres et la nouvelle erreur suivant la modélisation du système. La seconde étape va mettre en corrélation la prédiction faite avec la valeur des mesures. Cela permettra d'avoir une estimation des paramètres et de l'erreur. Dans le cas où le modèle comporte des petites erreurs, cette étape de mise à jour les rectifiera.

Pour pouvoir utiliser ce filtre, il faut avant tout réaliser le modèle du robuCAR. Un modèle avait déjà été réalisé auparavant mais il ne prenait en compte que la direction avant.

4.7.1 Modélisation du robuCAR

Afin de pouvoir fusionner et filtrer les données GPS et odométriques, nous avons réeffectué le modèle cinématique du robucar en prenant en compte non plus seulement l'angle de braquage du train avant mais également celui du train arrière. Ainsi quelle que soit la configuration du véhicule (train avant seul, train arrière seul, multidirectionnel ou parking), notre modèle fonctionnera. Nous avons décidé de réaliser le modèle d'un véhicule à deux roues et non à quatre roues car nous n'avons qu'un encodeur à l'avant pour déterminer l'angle de braquage avant et pour le train arrière la situation est identique. Le schéma ci-dessous permet de pouvoir de visualiser le modèle utilisé.

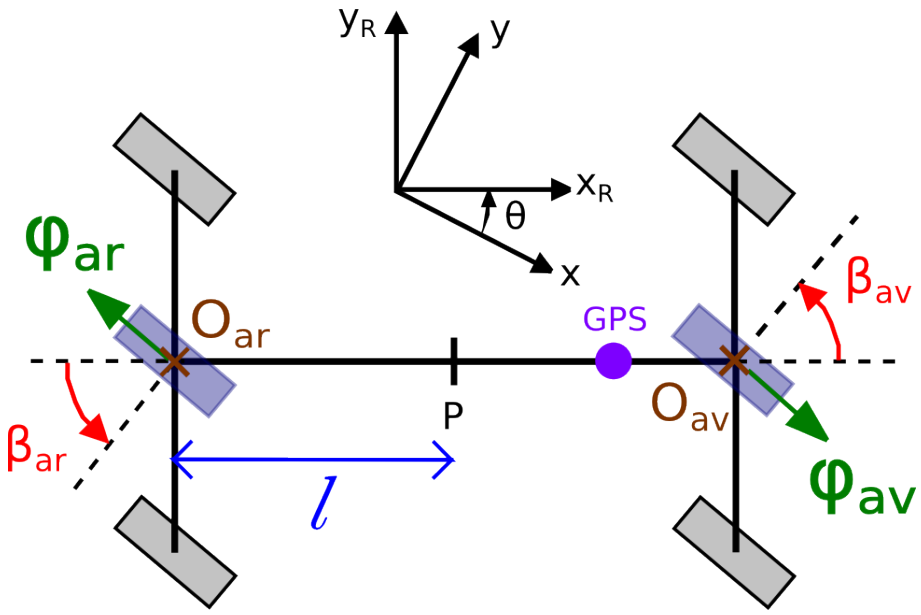


FIGURE 18 – Modélisation du robucar

θ : angle entre le repère local du robot (x_R,y_R) et le repère global (x,y)
r_{av},r_{ar} : rayons respectifs des roues du train avant et arrière
φ_{av},φ_{ar} : vitesses respectives des roues du train avant et arrière
β_{av}, β_{ar} : angles définis sur le schéma ci-dessus des trains avant et arrière

Pour déterminer notre modèle, nous avons utilisé les contraintes des équations de roulement et de glissement des deux roues du véhicule. On suppose qu'il n'y a pas de glissement et que l'on travaille à altitude constante. α_{av} est l'angle formé entre l'axe x_R et l'axe (PO_{av}), il vaut donc 0 rad. α_{ar} est l'angle formé entre l'axe x_R et l'axe (PO_{ar}), il vaut donc π rad.

Soit la matrice de rotation R telle que :

$$R = \begin{pmatrix} \cos(\theta) & \sin(\theta) & 0 \\ -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Les équations de roulement nous donnent :

$$\left\{ \begin{bmatrix} \sin(\alpha_{av}+\beta_{av}) & -\cos(\alpha_{av}+\beta_{av}) & -l.\cos(\beta_{av}) \\ \sin(\alpha_{ar}+\beta_{ar}) & -\cos(\alpha_{ar}+\beta_{ar}) & -l.\cos(\beta_{ar}) \end{bmatrix} . \; R . \begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{pmatrix} = \begin{pmatrix} r_{av}.\varphi_{av} \\ r_{ar}.\varphi_{ar} \end{pmatrix} \right.$$

Les équations de glissement nous donnent :

$$\left\{ \begin{bmatrix} \cos(\alpha_{av}+\beta_{av}) & \sin(\alpha_{av}+\beta_{av}) & l.\sin(\beta_{av}) \\ \cos(\alpha_{ar}+\beta_{ar}) & \sin(\alpha_{ar}+\beta_{ar}) & l.\sin(\beta_{ar}) \end{bmatrix} . \; R . \begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \right.$$

Soit la matrice G telle que :

$$G = \begin{pmatrix} \sin(\beta_{av}) & -\cos(\beta_{av}) & -l.\cos(\beta_{av}) \\ -\sin(\beta_{ar}) & \cos(\beta_{ar}) & -l.\cos(\beta_{ar}) \\ \cos(\beta_{av}) & \sin(\beta_{av}) & l.\sin(\beta_{av}) \\ -\cos(\beta_{ar}) & -\sin(\beta_{ar}) & l.\sin(\beta_{ar}) \end{pmatrix}$$

Le système global devient donc :

$$G\,.\,R\,.\begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{pmatrix} = \begin{pmatrix} r_{av}.\varphi_{av} \\ r_{ar}.\varphi_{ar} \\ 0 \\ 0 \end{pmatrix}$$

Afin de pouvoir résoudre le système, on multiplie les membres de chaque côté par la transposée de G :

$${}^{T}G.G.R.\begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{pmatrix} = {}^{T}G.\begin{pmatrix} r_{av}.\varphi_{av} \\ r_{ar}.\varphi_{ar} \\ 0 \\ 0 \end{pmatrix}$$

Etant que ${}^{T}G.G$ est une matrice diagonale avec tous ces éléments strictement positifs, on peut donc calculer facilement son inverse et on obtient le système suivant :

$$\begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{pmatrix} = R^{-1}.\left({}^{T}G.G\right)^{-1}.{}^{T}G.\begin{pmatrix} r_{av}.\varphi_{av} \\ r_{ar}.\varphi_{ar} \\ 0 \\ 0 \end{pmatrix}$$

Après détermination du système, nous obtenons :

$$\begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{pmatrix} = \frac{1}{2l}.\begin{pmatrix} l.(r_{av}.\varphi_{av}.(\cos(\beta_{av}).\sin(\theta)+\sin(\beta_{av}).\cos(\theta)) - r_{ar}.\varphi_{ar}.(\cos(\beta_{ar}).\sin(\theta)+\sin(\beta_{ar}).\cos(\theta))) \\ l.(r_{av}.\varphi_{av}.(-\cos(\beta_{av}).\cos(\theta)+\sin(\beta_{av}).\sin(\theta)) + r_{ar}.\varphi_{ar}.(\cos(\beta_{ar}).\cos(\theta)-\sin(\beta_{ar}).\sin(\theta))) \\ -r_{av}.\varphi_{av}.\cos(\beta_{av}) - r_{ar}.\varphi_{ar}.\cos(\beta_{ar}) \end{pmatrix}$$

A présent, nous pouvons déterminer les différents vecteurs pour notre filtre de Kalman étendu

Vecteur d'état ancien : $X_k = \begin{pmatrix} x_k \\ y_k \\ \theta_k \\ \dot{x}_k \\ \dot{y}_k \\ \dot{\theta}_k \end{pmatrix}$ Vecteur de commande : $U_k = \begin{pmatrix} \beta_{av} \\ \beta_{ar} \\ \varphi_{ar} \\ \varphi_{av} \end{pmatrix}$

Vecteur d'état nouveau : $X_{k+1} = f(X_k, U_k)$

avec $X_{k+1} = \begin{pmatrix} x_k + dt.\dot{x}_k \\ y_k + dt.\dot{y}_k \\ \theta_k + dt.\dot{\theta}_k \\ \frac{1}{2}(r_{av}.\varphi_{av}.(\cos(\beta_{av}).\sin(\theta_k)+\sin(\beta_{av}).\cos(\theta_k)) + r_{ar}.\varphi_{ar}.(-\cos(\beta_{ar}).\sin(\theta_k)-\sin(\beta_{ar}).\cos(\theta_k))) \\ \frac{1}{2}(r_{av}.\varphi_{av}.(-\cos(\beta_{av}).\cos(\theta_k)+\sin(\beta_{av}).\sin(\theta_k)) + r_{ar}.\varphi_{ar}.(\cos(\beta_{ar}).\cos(\theta_k)-\sin(\beta_{ar}).\sin(\theta_k))) \\ \frac{1}{2l}(-r_{av}.\varphi_{av}.\cos(\beta_{av}) - r_{ar}.\varphi_{ar}.\cos(\beta_{ar})) \end{pmatrix}$

Nous disposons d'une centrale inertielle permettant de mesurer $\dot{\theta}_k$. Nous avons également un GPS situé en (d,0) dans le repère robot sachant que le point P est l'origine du repère robot.

Vecteur de mesure : $Y_k = h(X_k) = \begin{pmatrix} x_k + d.cos(\theta_k) \\ y_k + d.sin(\theta_k) \\ \dot{\theta}_k \end{pmatrix}$

Les matrices A, B et H sont définies telles que :

$A_{ij} = \frac{\partial f_i(X_k,U_k)}{\partial x_j}$ avec A_{ij} élément(i,j) de la matrice A et x_j élément j du vecteur X_k

$B_{ij} = \frac{\partial f_i(X_k,U_k)}{\partial u_j}$ avec B_{ij} élément(i,j) de la matrice B et u_j élément j du vecteur U_k

$H_{ij} = \frac{\partial h_i(X_k)}{\partial x_j}$ avec H_{ij} élément(i,j) de la matrice H et x_j élément j du vecteur X_k

Après un certain nombre de tests, nous sommes parvenus à régler notre filtre de Kalman. Pour pouvoir analyser nos données odométriques et GPS , nous avons du réaliser des fichiers d'extension .mat pour Matlab et .kml pour Google Earth. Nous avons réalisé des simulations en modifiant les paramètres des matrices P (matrice de covariance du bruit de mesure) et Q (matrice de covariance du bruit de commande). Au final, les deux matrices sont des matrices diagonales. En effet, les valeurs mesurées par le GPS et la centrale inertielle ne dépendent pas entre elles. De même, les valeurs des vitesses des roues et du angles β ne dépendent pas entre elles. Ci-dessous nous pouvons observer l'exportation sous Matlab ainsi que sur Google Earth.

(a) Matlab (b) Google Earth

FIGURE 19 – Visualisations des trames GPS et du filtre

Le zoom ci-dessous permet de montrer que le filtre de Kalman ne se contente pas de suivre parfaitement les trames GPS, il fusionne l'odométrie et le GPS. Du coup, la trajectoire suivie par le filtre n'est pas chaotique comme celle du GPS, ce qui sera bien utile pour le suivi de trajectoire.

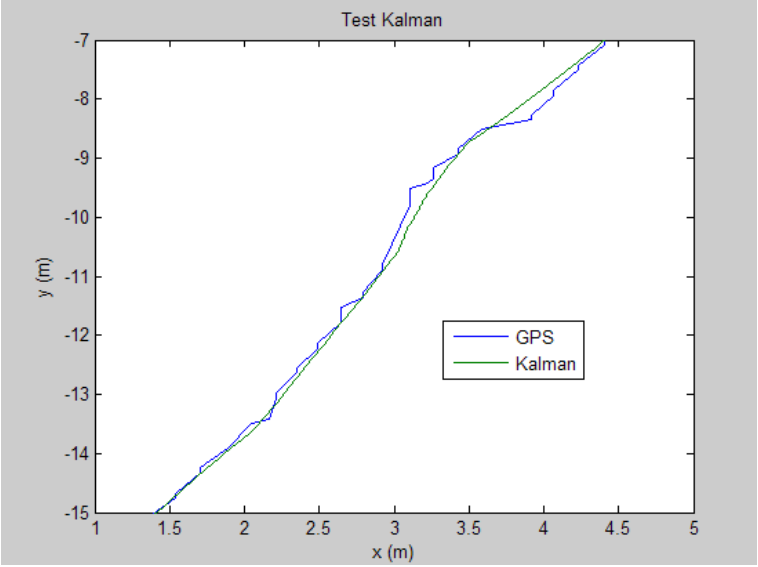


FIGURE 20 – Zoom de la visualisation sur Matlab

4.7.2 Inversion du modèle

Afin de pouvoir réaliser un suivi de trajectoire, nous allons déterminer le système inverse, c'est-à-dire déterminer β_{av}, β_{ar}, φ_{av},φ_{ar} en fonction de $\dot{x}$, $\dot{y}$, $\dot{\theta}$ et de θ. Nous reprenons les équations de glissement et de roulement :

$$\begin{cases} sin(\beta_{av}).(-\dot{x}.sin(\theta)+\dot{y}.cos(\theta)+l.\dot{\theta})+cos(\beta_{av}).(\dot{x}.cos(\theta)+\dot{y}.sin(\theta))=0 \\ sin(\beta_{ar}).(\dot{x}.sin(\theta)-\dot{y}.cos(\theta)+l.\dot{\theta})+cos(\beta_{ar}).(-\dot{x}.cos(\theta)-\dot{y}.sin(\theta))=0 \\ sin(\beta_{av}).(\dot{x}.cos(\theta)+\dot{y}.sin(\theta))+cos(\beta_{av}).(\dot{x}.sin(\theta)-\dot{y}.cos(\theta)-l.\dot{\theta})=r_{ar}.\varphi_{av} \\ sin(\beta_{ar}).(-\dot{x}.cos(\theta)-\dot{y}.sin(\theta))+cos(\beta_{ar}).(-\dot{x}.sin(\theta)+\dot{y}.cos(\theta)-l.\dot{\theta})=r_{ar}.\varphi_{ar} \end{cases}$$

Avec les équations de glissement (1 et 2), nous pouvons connaitre les valeurs respectives de β_{av} et β_{ar}, puis, avec les équations de roulement (3 et 4), les valeurs respectives de φ_{av} et φ_{ar}.

$$\begin{cases} tan(\beta_{av})=\frac{sin(\beta_{av})}{cos(\beta_{av})}=\frac{-\dot{x}.cos(\theta)-\dot{y}.sin(\theta)}{-\dot{x}.sin(\theta)+\dot{y}.cos(\theta)+l.\dot{\theta}} \\ tan(\beta_{ar})=\frac{sin(\beta_{ar})}{cos(\beta_{ar})}=\frac{\dot{x}.cos(\theta)+\dot{y}.sin(\theta)}{\dot{x}.sin(\theta)-\dot{y}.cos(\theta)+l.\dot{\theta}} \\ \varphi_{av}=\frac{sin(\beta_{av}).(\dot{x}.cos(\theta)+\dot{y}.sin(\theta))+cos(\beta_{av}).(\dot{x}.sin(\theta)-\dot{y}.cos(\theta)-l.\dot{\theta})}{r_{av}} \\ \varphi_{ar}=\frac{sin(\beta_{ar}).(-\dot{x}.cos(\theta)-\dot{y}.sin(\theta))+cos(\beta_{ar}).(-\dot{x}.sin(\theta)+\dot{y}.cos(\theta)-l.\dot{\theta})}{r_{ar}} \end{cases}$$

A l'aide de Matlab Simulink, nous avons testé l'inversion du système. β_{av} et β_{ar} sont compris dans l'intervalle $\left[\frac{\pi}{2} - \frac{\pi}{9} \quad ; \quad \frac{\pi}{2} + \frac{\pi}{9} \right]$. C'est pourquoi il est indispensable de multiplier β_{av} et β_{ar} par $k\pi$ avec $k \in \mathbb{Z}$ pour qu'ils soient dans l'intervalle. Il est à noter que l'utilisation de la fonction atan dans Matlab pour déterminer β_{av} et β_{ar} est à proscrire. Il est nécessaire d'utiliser la fonction atan2 sinon nous devons faire face à de nombreuses discontinuités. Ci-dessous, nous pouvons observer à gauche l'odométrie avec des valeurs mesurées. A droite, nous observons également l'odométrie mais là avec les valeurs déterminées par le calcul en fonction de $\dot{x}$, $\dot{y}$, $\dot{\theta}$ et de θ. Nous obtenons des courbes très similaires.

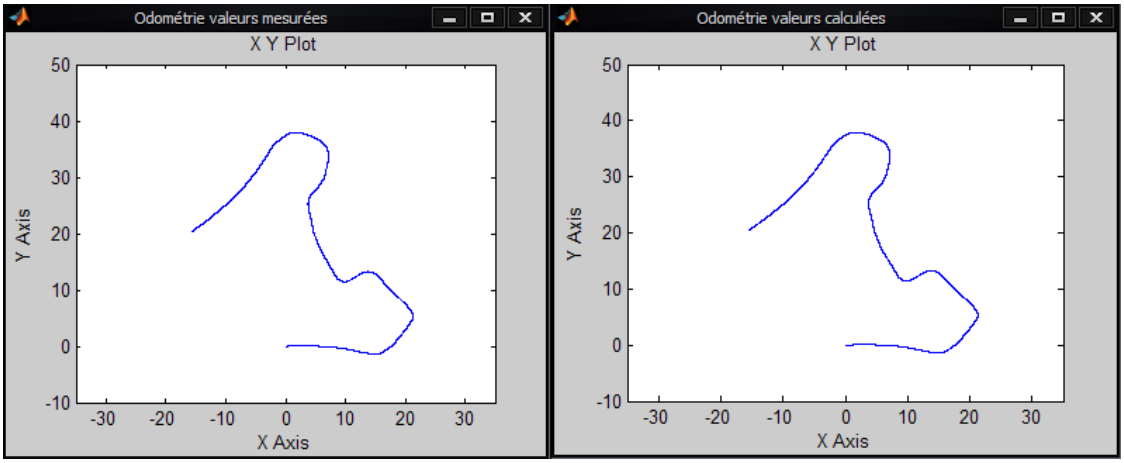


FIGURE 21 – Odométrie avec valeurs mesurées et valeurs calculées

L'écart suivant l'axe x ou l'axe y ne dépasse jamais 1 cm. Les déplacements sont pourtant de l'ordre de dizaines de mètres donc nous pouvons considérer les écarts négligeables.

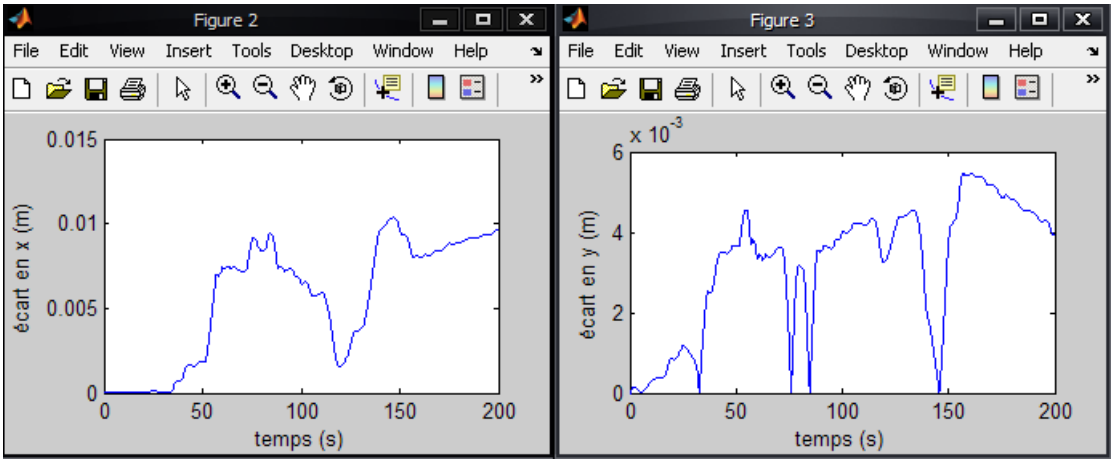


FIGURE 22 – Ecarts suivant l'axe x et suivant l'axe y

4.7.3 Modélisation du robuTAINER

Nous avons également établi le modèle cinématique du robutainer. Nous avons appliqué les contraintes des équations de roulement et de glissement des quatre roues.

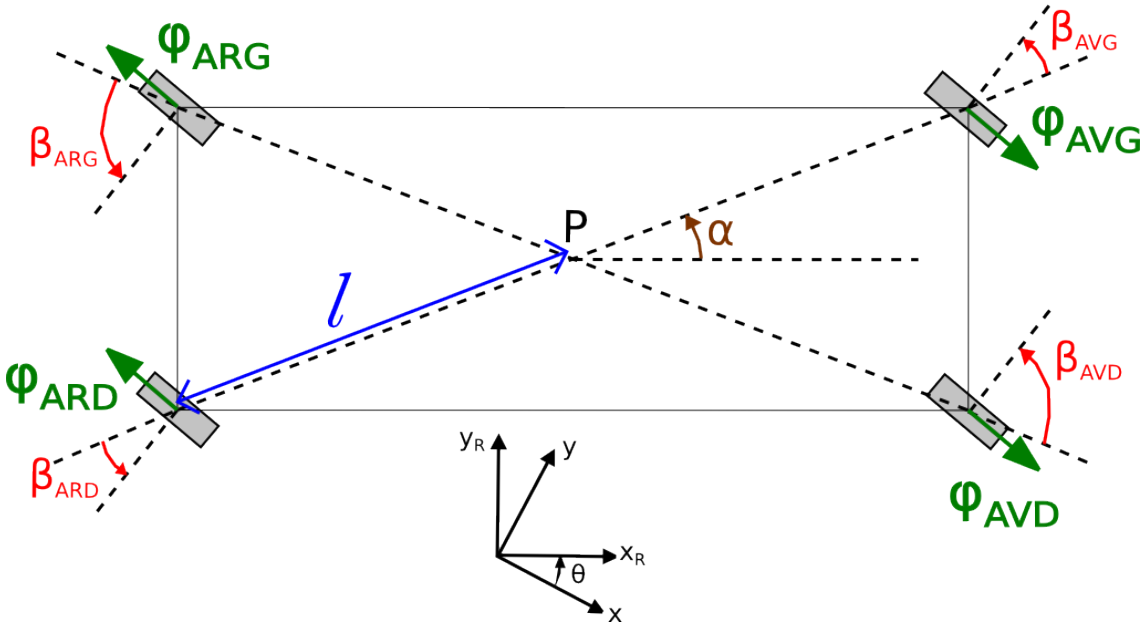


FIGURE 23 – Modélisation du robutainer

θ : angle entre le repère local du robot (x_R,y_R) et le repère global (x,y)
r_{AVD},r_{AVG},r_{ARD},r_{ARG} : rayons des roues
φ_{AVD},φ_{AVG},φ_{ARD},φ_{ARG} : vitesses respectives des roues
β_{AVD}, β_{AVG},β_{ARD}, β_{ARG} : angles définis sur le schéma ci-dessus
α : angle entre l'axe x_R et l'axe formé par le point P et le centre de la roue avant gauche

Les équations de roulement nous donnent :

$$\begin{cases} \begin{bmatrix} \sin(\alpha+\beta_{AVG}) & -\cos(\alpha+\beta_{AVG}) & -l.\cos(\beta_{AVG}) \end{bmatrix} \\ \begin{bmatrix} \sin(\pi-\alpha+\beta_{ARG}) & -\cos(\pi-\alpha+\beta_{ARG}) & -l.\cos(\beta_{ARG}) \end{bmatrix} \\ \begin{bmatrix} \sin(\pi+\alpha+\beta_{ARD}) & -\cos(\pi+\alpha+\beta_{ARD}) & -l.\cos(\beta_{ARD}) \end{bmatrix} \\ \begin{bmatrix} \sin(2\pi-\alpha+\beta_{AVD}) & -\cos(2\pi-\alpha+\beta_{AVD}) & -l.\cos(\beta_{AVD}) \end{bmatrix} \end{cases} . \; R . \begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{pmatrix} = \begin{pmatrix} r_{AVG}.\varphi_{AVG} \\ r_{ARG}.\varphi_{ARG} \\ r_{ARD}.\varphi_{ARD} \\ r_{AVD}.\varphi_{AVD} \end{pmatrix}$$

Les équations de glissement nous donnent :

$$\left\{ \begin{array}{c} \left[\begin{array}{ccc} \cos(\alpha+\beta_{AVG}) & \sin(\alpha+\beta_{AVG}) & l.\sin(\beta_{AVG}) \end{array} \right] \\ \left[\begin{array}{ccc} \cos(\pi-\alpha+\beta_{ARG}) & \sin(\pi-\alpha+\beta_{ARG}) & l.\sin(\beta_{ARG}) \end{array} \right] \\ \left[\begin{array}{ccc} \cos(\pi+\alpha+\beta_{ARD}) & \sin(\pi+\alpha+\beta_{ARD}) & l.\sin(\beta_{ARD}) \end{array} \right] \\ \left[\begin{array}{ccc} \cos(2\pi-\alpha+\beta_{AVD}) & \sin(2\pi-\alpha+\beta_{AVD}) & l.\sin(\beta_{AVD}) \end{array} \right] \end{array} \right. \cdot R \cdot \begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

Pour pouvoir les différentes vitesses du robutainer, nous avons utilisé la même méthode qu'avec le robucar. Nous pourrons noter que le modèle cinématique du robutainer est très proche sur la forme et sur le fond de celui du robucar.

$$\begin{pmatrix} \dot{x} \\ \\ \dot{y} \\ \\ \dot{\theta} \end{pmatrix} = \frac{1}{4} \cdot \begin{pmatrix} \varphi_{ARD}.r_{ARD}.(-cos(\alpha+\beta_{ARD}).sin(\theta) - cos(\theta).sin(\alpha+\beta_{ARD})) \\ +\varphi_{ARG}.r_{ARG}.(-cos(\alpha-\beta_{ARG}).sin(\theta) + cos(\theta).sin(\alpha-\beta_{ARG})) \\ +\varphi_{AVD}.r_{AVD}.(cos(\alpha-\beta_{AVD}).sin(\theta) - cos(\theta).sin(\alpha-\beta_{AVD})) \\ +\varphi_{AVG}.r_{AVG}.(cos(\alpha+\beta_{AVG}).sin(\theta) + cos(\theta).sin(\alpha+\beta_{AVG})) \\ \\ \varphi_{ARD}.r_{ARD}.(cos(\alpha+\beta_{ARD}).cos(\theta) - sin(\alpha+\beta_{ARD}).sin(\theta)) \\ +\varphi_{ARG}.r_{ARG}.(cos(\alpha-\beta_{ARG}).cos(\theta) + sin(\alpha-\beta_{ARG}).sin(\theta)) \\ +\varphi_{AVD}.r_{AVD}.(-cos(\alpha-\beta_{AVD}).cos(\theta) - sin(\alpha-\beta_{AVD}).sin(\theta)) \\ +\varphi_{AVG}.r_{AVG}.(-cos(\alpha+\beta_{AVG}).cos(\theta) + sin(\alpha+\beta_{AVG}).sin(\theta)) \\ \\ \frac{1}{l}(-\varphi_{ARD}.r_{ARD}.cos(\beta_{ARD}) - \varphi_{ARG}.r_{ARG}.cos(\beta_{ARG}) \\ -\varphi_{AVD}.r_{AVD}.cos(\beta_{AVD}) - \varphi_{AVG}.r_{AVG}.cos(\beta_{AVG})) \end{pmatrix}$$

4.8 Suivi de chemin

Pour le suivi de chemin , nous avons décidé d'utiliser une méthode permettant de prendre en compte l'erreur d'angle entre le véhicule et la trajectoire. Les calculs sont réalisés à l'aide de la fonction atan2. Comme il n'est pas toujours possible de pouvoir travailler sur le robuCAR, un suivi de chemin en simulation a été réalisé. Il permet à l'utilisateur de tester ainsi les algorithmes sans risquer de détériorer quoi que ce soit. L'utilisateur peut choisir de piloter manuellement le véhicule ou laisser le véhicule se déplacer de façon autonome. La capture suivante permet d'observer l'interface de visualisation, avec la flèche représentant le véhicule sur le chemin généré. On peut aussi observer les objets détectés par le laser.

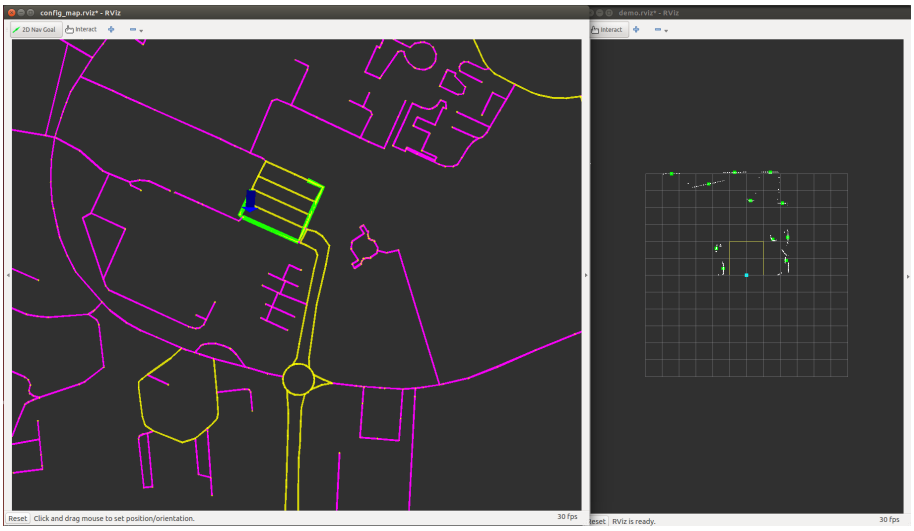


FIGURE 24 – Visualisation du suivi de trajectoire

Ci-dessous, nous pouvons observer l'erreur en position en simulation ainsi que l'erreur en angle. L'énorme écart en position de plus d'un mètre ainsi que l'écart en angle sont dus à un virage à 90 degrés. Les résultats en simulation sont plutôt probants. En effet, en régulant juste sur l'angle, le véhicule suit proprement la trajectoire.

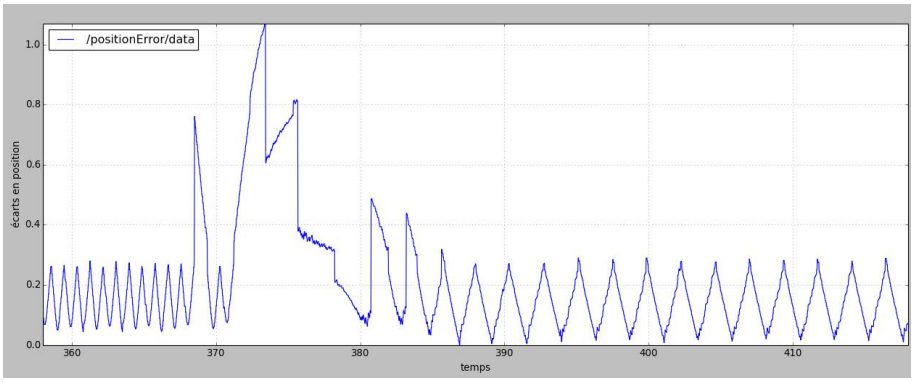


FIGURE 25 – Erreur en position en simulation

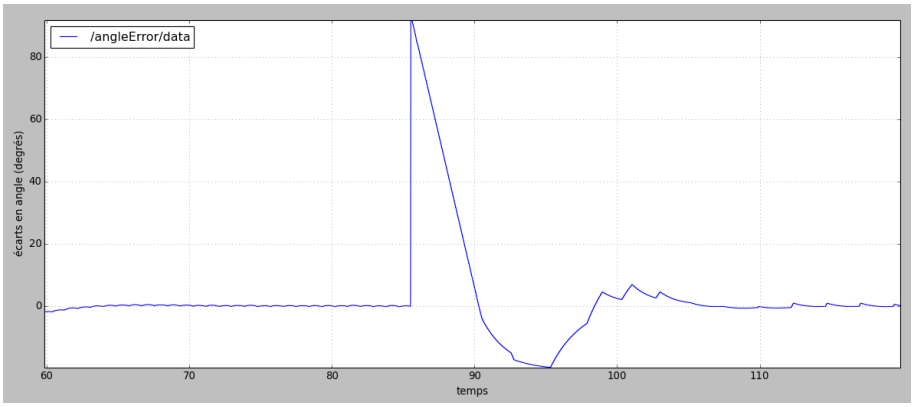


FIGURE 26 – Erreur en angle en simulation

Une vidéo permettant de voir le véhicule suivre une trajectoire choisie par l'utilisateur est disponible au lien suivant :

5 Améliorations envisageables

5.1 Suivi de chemin

Actuellement, le suivi de chemin ne s'effectue qu'autour de la map générée. Il n'y a aucun respect du Code de la route. Deux caméras permettraient par exemple de faire du suivi de ligne, de détecter les panneaux de signalisation, les bandes de stop ... De plus, la redondance de laser permettrait de faire des marches arrière, car pour l'instant, le véhicule n'a aucune connaissance de ce qui pourrait se trouver derrière lui.

5.2 Évitement d'obstacle

Le robuCAR est capable de s'arrêter si un objet se trouve dans la zone de danger délimitée devant le véhicule. Cependant, cette zone est statique. Une amélioration possible serait de la rendre dynamique en fonction des commandes de vitesse et de braquage ainsi que des valeurs de mesures de vitesse et de braquage. De plus, le véhicule pourrait réellement "éviter" l'obstacle, c'est-à-dire le contourner au lieu de rester immobile en attendant que l'objet disparaisse.

5.3 Localisation

Lors de nos tests, nous avons rencontré un comportement anormal au passage d'une bordure (trottoir). Cela est probablement dû au fait que l'on considère le tangage et le roulis nuls pour déterminer l'angle de lacet. Le "choc" détecté par la centrale inertielle est alors interprété par notre filtre comme une brusque accélération. Cela conduit alors à une mauvaise estimation de la vitesse suivant ϑ, à un instant t.

5.4 Serveur Pure

Si la commande fonctionne correctement et que le serveur communique correctement avec les différents clients codés auparavant, quelques modifications peuvent être à envisager.

- Initialement, le serveur était prévu pour fonctionner avec un thread et le code de celui-ci est toujours disponible, en commentaire dans le main. Ce thread devait s'occuper de l'émission périodique de messages de notifications, pendant que le main restait en attente de messages entrants. Cela permettait de réduire le risque de messages manqués, lorsqu'un client se reconnecte au serveur après avoir déjà établi une communication et des abonnements. Malheureusement, la bibliothèque Clib n'a pas fonctionné lorsque l'appel venait du thread, alors que la même fonction appelée depuis le processus principal ne posait aucun souci. Par manque de temps, nous n'avons pu chercher s'il était possible de résoudre ce problème, par exemple si la création du thread se fait d'une façon différente (librairie boost maintenant disponible).
- Si le problème de thread ne peut être résolu, il est aussi possible d'envisager une fonction de reconnexion automatique après des erreurs d'envois sur la socket. La fonction pourrait par exemple être appelée depuis la fonction send d'udpManager, lorsque l'appel à sendto retourne erreur, et pourrait exécuter successivement un closeCleanup,

puis une nouvelle initialisation et enfin un receive afin d'attendre la connexion d'un nouveau client.
- Les boutons de ControlDesk n'ont pas été liés à l'exécutable compilé de notre serveur, cela permettrait de rendre le lancement plus aisé.
- Le champ status retourné par le service Localization ne tient pas compte de la réception GPS, ou de l'état du filtre de Kalman. Un bloc pourrait donc être créé dans le projet Simulink afin de modifier cette variable selon les critères de PURE, en fonction du nombre de fix GPS obtenu, ou du temps depuis lequel le filtre Kalman fonctionne.
- Enfin, le gain sur les régulations des trains de direction pourrait être ajusté afin d'obtenir une réponse plus rapide, car celle-ci est assez lente, et encore plus lorsque les batteries deviennent plus faibles.

5.5 Divers

Dans les améliorations générales, on pourrait rendre le robuCAR étanche afin de pouvoir travailler lorsqu'il fait mauvais temps. En effet, il est impossible de réaliser des tests sur le robuCAR lorsqu'il pleut par exemple. Par ailleurs, le changement des batteries serait bienvenu afin de pouvoir faire durer les tests, car nous étions très limités en temps lors des différents tests. Une amélioration de la régulation de la vitesse de braquage permettrait de rendre plus réactif le suivi de trajectoire. En effet une fois les batteries légèrement déchargées, la réponse aux consignes de directions devient encore plus lente. Enfin, il serait bien de pouvoir réguler en boucle fermée la vitesse des roues. Cela éviterait les lenteurs lors de l'application d'une nouvelle consigne de vitesse, ou lors d'un ralentissement pour présence d'obstacle.

6 Conclusion

Au terme du projet, nous sommes parvenus à faire fonctionner toutes les parties ensemble. Ainsi, le suivi de trajectoire (PC externe) prend en compte les données du laser et du suivi de chemin afin de réaliser la trajectoire demandée par l'utilisateur. La localisation (dSpace) fusionne les données odométriques et GPS à l'aide du filtre de Kalman. Le serveur PURE, quant à lui, rend possible la transmission de données entre le PC externe et la dSpace. L'interface à laquelle l'utilisateur est confronté est facile d'utilisation, il n' a qu'à cliquer à l'endroit où il veut se diriger pour lancer le déplacement du véhicule. Les différentes parties réalisées sur le PC externe pourront être ré-utilisées pour le robuTAINER.

Ce projet nous aura permis de travailler sur de multiples interfaces (Matlab-Simulink,Visual Studio++, ROS, ControlDesk) et avec divers langages de programmation. Contrairement à ce que nous avons l'habitude, nous avons été confrontés à de nombreux problèmes matériels tels que le dysfonctionnement des équipements du robuCAR. De plus, il fallait faire avec le mauvais temps qui nous empêchait de sortir le robuCAR de part son manque d'étanchéité et de robustesse au froid (condensation).

Bien qu'il reste certains points à améliorer, nous pouvons considérer que le projet est fonctionnel dans sa globalité.Une vidéo de tout l'ensemble fonctionnant de concert sera bientôt disponible sur le Wiki du projet.

Annexes

Liste des paquets à installer

```
ros-indigo-gegraphic-info
ros-indigo-geographic-msgs
ros-indigo-osm-cartography
ros-indigo-sicktoolbox
ros-indigo-sicktoolbox-wrapper
ros-indigo-route-network
```

Matrices A,B et H

$A_{ij} = \frac{\partial f_i(X_k,U_k)}{\partial x_j}$ avec A_{ij} élément(i,j) de la matrice A et x_j élément j du vecteur X_k

$$A = \begin{pmatrix} 1 & 0 & 0 & dt & 0 & 0 \\ 0 & 1 & 0 & 0 & dt & 0 \\ 0 & 0 & 1 & 0 & 0 & dt \\ 0 & 0 & \frac{1}{2}(r_{av}.\varphi_{av}.(\cos(\beta_{av}).\cos(\theta_k) - \sin(\beta_{av}).\sin(\theta_k)) \\ & & +r_{ar}.\varphi_{ar}.(-\cos(\beta_{ar}).\cos(\theta_k) + \sin(\beta_{ar}).\sin(\theta_k))) & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{2}(r_{av}.\varphi_{av}.(\cos(\beta_{av}).\sin(\theta_k) + \sin(\beta_{av}).\cos(\theta_k)) \\ & & +r_{ar}.\varphi_{ar}.(-\cos(\beta_{ar}).\sin(\theta_k) - \sin(\beta_{ar}).\cos(\theta_k))) & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

$B_{ij} = \frac{\partial f_i(X_k,U_k)}{\partial u_j}$ avec B_{ij} élément(i,j) de la matrice B et u_j élément j du vecteur U_k

$$B = \frac{1}{2}\begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ r_{av}.\varphi_{av}.(-sin(\beta_{av}).\sin(\theta_k) & r_{ar}.\varphi_{ar}.(sin(\beta_{ar}).\sin(\theta_k) & r_{ar}.(-\cos(\beta_{ar}).\sin(\theta_k) & r_{av}.(\cos(\beta_{av}).\sin(\theta_k) \\ +cos(\beta_{av}).\cos(\theta_k)) & -cos(\beta_{ar}).\cos(\theta_k)) & -\sin(\beta_{ar}).\cos(\theta_k)) & +\sin(\beta_{av}).\cos(\theta_k)) \\ r_{av}.\varphi_{av}.(sin(\beta_{av}).\cos(\theta_k) & r_{ar}.\varphi_{ar}.(-sin(\beta_{ar}).\cos(\theta_k) & r_{ar}.(\cos(\beta_{ar}).\cos(\theta_k) & r_{av}.(-\cos(\beta_{av}).\cos(\theta_k) \\ +cos(\beta_{av}).\sin(\theta_k)) & -cos(\beta_{ar}).\sin(\theta_k)) & -\sin(\beta_{ar}).\sin(\theta_k)) & +\sin(\beta_{av}).\sin(\theta_k)) \\ \frac{1}{l}(r_{av}.\varphi_{av}.sin(\beta_{av})) & \frac{1}{l}(r_{ar}.\varphi_{ar}.sin(\beta_{ar})) & \frac{1}{l}(-r_{av}.\cos(\beta_{av})) & \frac{1}{l}(-r_{ar}.\cos(\beta_{ar})) \end{pmatrix}$$

$H_{ij} = \frac{\partial h_i(X_k)}{\partial x_j}$ avec H_{ij} élément(i,j) de la matrice H et x_j élément j du vecteur X_k

$$H = \begin{pmatrix} 1 & 0 & -d.sin(\theta_k) & 0 & 0 & 0 \\ 0 & 1 & d.cos(\theta_k) & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Table des figures