

École polytechnique de Louvain

Robotran embarqué : Robot Operating System

Validation et application sur un simulateur de
véhicules à retour haptique

Auteur: **Théo TUERLINCKX**
Promoteurs: **Paul FISETTE, Nicolas DOCQUIER**
Lecteurs: **Philippe CHATELAIN, Sébastien TIMMERMANS**
Année académique 2019–2020
Master [120] : ingénieur civil mécanicien

L'industrie actuelle est composée de nombreux systèmes mécaniques (tels les robots, les voitures, les trains, les bras mécaniques des chaînes de production, ...). Ces dispositifs tendent maintenant à utiliser des systèmes de simulation en temps-réel embarqués pour prévoir, analyser et ensuite modifier leurs comportements. L'objectif est alors d'augmenter les performances, l'automation ou encore la sécurité de ces mécanismes.

Ce travail présente l'utilisation du logiciel Robotran pour simuler la cinématique et dynamique d'un dispositif mécanique. Il associe ce logiciel à l'exploitation de ROS (i.e. Robot Operating System) afin que la simulation puisse interagir en temps-réel avec son environnement physique par le biais de capteurs et actionneurs. La fusion de ces deux logiciels est validée à l'aide de cas simples et ensuite appliquée à un simulateur de véhicules à retour haptique.

Remerciements

Ce mémoire a pu être réalisé grâce à l'aide et l'expérience de personnes que je tenais tout d'abord à remercier.

Je remercie le Professeur Paul Fisette pour sa confiance et son soutien dans ce travail ainsi que pour l'apprentissage reçu lors de ses cours à l'UCLouvain.

Je remercie Sébastien Timmermans pour tout le temps qu'il a investi dans ce projet tout au long de l'année. Ce travail n'aurait pu se faire sans son expérience, son aide et sa disponibilité apportées dans les différentes étapes à réaliser. Je le remercie également pour la confiance et le temps qu'il a consacré en laboratoire pour permettre l'aboutissement de ce travail.

Je remercie également Nicolas Docquier pour ses précieux nombreux conseils et avis dans les choix d'orientation et d'application de ce mémoire.

Je remercie aussi ces trois personnes pour les démarches réalisées afin d'adapter la conception de ce projet malgré la crise sanitaire engendrée par le COVID-19.

Finalement, je remercie ma famille pour son soutien ainsi que le temps consacré à la relecture de ce mémoire.

Table des matières

Remerciements	5
Liste des figures	10
Liste des tableaux	11
Introduction	13
1 Robotran	15
1.1 Fonctionnement	15
1.1.1 Modélisation d'un multicorps : MBsysPAD	15
1.1.2 Génération des équations du mouvement : MBsysTran	16
1.1.3 Résolutions des équations du mouvement : MBsysC	17
1.2 Architecture d'un projet Robotran	19
1.3 Modèles multicorps utilisés	19
1.3.1 Voiture	20
1.3.2 Kart	26
2 Robot Operating System	29
2.1 Fonctionnement	29
2.1.1 La Plomberie	30
2.1.2 Les Outils	31
2.1.3 Les Capacités	31
2.1.4 L'Ecosystème	32
2.2 Architecture d'un projet ROS	32
3 Intégration de Robotran dans ROS	35
3.1 Objectif et contraintes	35
3.2 Intérêts de ROS	36
3.2.1 Utilisation d'autres composants	36
3.2.2 Distribution des tâches sur différents composants	37
3.2.3 Réseau ROS universel et flexible	37
3.2.4 Gestion et automatisation du procédé	38
3.3 Fonctionnement	38
3.3.1 Paquet <i>ros_rob</i>	38
3.3.2 Projet Robotran	38

4	Validation sur des cas unitaires	41
4.1	Actionneur extérieur	41
4.1.1	Architecture de l'hardware	41
4.1.2	Architecture du software	42
4.1.3	Architecture de contrôle	43
4.1.4	Résultats	45
4.2	Capteurs extérieurs et simulation anticipée	52
4.2.1	Architecture de l'hardware	53
4.2.2	Architecture du software	55
4.2.3	Architecture de contrôle	57
4.2.4	Dynamique de l'évitement	59
4.2.5	Résultats	63
5	Simulateur à retour haptique	67
5.1	Présentation du simulateur	68
5.2	Architecture de l'hardware	71
5.2.1	Architecture globale	71
5.2.2	Module multicorps	72
5.2.3	Module de transmission	72
5.2.4	Module actionneur	74
5.3	Architecture du software et protocoles de communication	75
5.3.1	CANopen	75
5.3.2	Topic	79
5.4	Architecture de contrôle	79
5.4.1	Fréquence des boucles des noeuds ROS	80
5.4.2	Initialisation et utilisation du servo-variateur	81
5.5	Expérimentations et résultats	82
5.5.1	Validation	82
5.5.2	Application sur des cas simples	89
5.6	Conclusion et améliorations	94
	Conclusion	97
A	Capteurs et actionneurs pour Raspberry	101
B	Tutoriel pour le simulateur à retour haptique	103

Table des figures

1.1	Modules de Robotran pour une implémentation en C	15
1.2	Boucle temporelle de Robotran pour la résolution d'un système multicorps avec contraintes	18
1.3	Arborescence d'un projet Robotran	19
1.4	Vue 3D de la voiture dans MBSysPAD	20
1.5	Angle de chasse et traînée mécanique	21
1.6	Angle de pivot et rayon de frottement	21
1.7	Schéma cinématique du système de direction	23
1.8	Ratio variable du système de direction	23
1.9	Couple d'auto-alignement des roues en virage	24
1.10	Vue 3D du kart dans MBSysPAD	26
1.11	Système de direction "à manivelle"	27
2.1	Concepts généraux de ROS	30
2.2	Plomberie de ROS	30
2.3	Exemple de la plomberie ROS affichée par rqt_graph	31
2.4	Arborescence d'un paquet ROS	32
2.5	Arborescence de l'espace de travail	33
3.1	Multithreading dans la main Robotran	40
4.1	Raspberry Pi 2 Model B	41
4.2	Architecture physique globale	42
4.3	Architecture globale de ROS	43
4.4	Architecture du protocole de communication	45
4.5	Interface graphique d'une simulation temps-réel Robotran	46
4.6	Section de circuit pour les mesures de fréquence	47
4.7	Variation du pas de temps pour l'analyse de différentes fréquences	48
4.8	Pas de temps optimal pour la simulation ROS-Robotran	48
4.9	Comparaison d'une simulation en ligne droite ou sur circuit pour $\Delta_{t,sim}=0.75ms$	49
4.10	Lien entre la baisse de fréquence et la dynamique du circuit pour $dt=0.75ms$	49
4.11	Lien entre la baisse de fréquence et la résolution des contraintes pour $\Delta_{t,sim}=0.75ms$	50
4.12	Schéma des différentes fréquences mesurées pour $\Delta_{t,sim}=0.82ms$	51
4.13	Mise en contexte d'un scénario d'évitement	53
4.14	Capteur ultrason HC-SR04	54
4.15	Circuit électronique pour le branchement du sonar au Raspberry	54
4.16	Architecture physique globale	55
4.17	Architecture globale de ROS	55

4.18	Protocole de communication pour un évitement	57
4.19	Force longitudinale en fonction du glissement longitudinale	60
4.20	Couple de freinage maximum	61
4.21	Distance de freinage sur sol sec	62
4.22	Angle de braquage du volant en fonction de la vitesse du véhicule	62
4.23	Echelle du temps du protocole d'esquive sur le côté d'un obstacle	63
4.24	Trajectoires pour l'esquive d'un obstacle à du 70 km/h	64
5.1	Illustration des 5 composants principaux du simulateur de véhicule	68
5.2	Vue éclatée de l'assemblage mécanique sous <i>SolidWorks</i>	70
5.3	Vue d'ensemble du simulateur	70
5.4	Architecture physique globale	71
5.5	Carte PIKAN CAN-Bus pour Raspberry Pi	73
5.6	Servo-variateur LXM32AD30M2	74
5.7	Servo-moteur BMH1401P12A2A	74
5.8	Protocole de communication CANopen	76
5.9	Architecture de KaCanOpen	77
5.10	Architecture globale de ROS pour le simulateur haptique	79
5.11	Architecture du protocole de communication pour le simulateur haptique	80
5.12	Données sur la position du volant pour la voiture	84
5.13	Données sur la vitesse du volant pour la voiture	84
5.14	Position du volant pour la voiture	85
5.15	Vitesse du volant pour la voiture	86
5.16	Vitesse du volant pour le kart	86
5.17	Données sur le couple pour la voiture	87
5.18	Coupe dans le volant pour la voiture	88
5.19	Couple dans le volant pour le kart	88
5.20	Zoom du couple dans le volant pour la voiture	89
5.21	Zoom du couple dans le volant pour le kart	89
5.22	Couple de retour du volant : chasse mécanique de 42.02 mm	90
5.23	Trajectoire du véhicule : chasse mécanique de 42.02 mm	90
5.24	Couple de retour du volant : chasse mécanique de 38.02 mm	91
5.25	Trajectoire du véhicule : chasse mécanique de 38.02 mm	91
5.26	Couple de retour du volant : chasse variable et contre-braquage	91
5.27	Trajectoire du véhicule : chasse variable et contre-braquage	92
5.28	Trajectoire du véhicule lors de l'esquive	92
5.29	Couple de retour du volant lors de l'esquive	93
5.30	Position du volant lors de l'esquive	93
A.1	Capteurs et actionneurs pour Raspberry	102

Liste des tableaux

1.1	Valeurs usuelles pour la géométrie des roues	21
1.2	Géométrie des roues de la voiture	22
1.3	Répartition de la masse du véhicule	23
4.1	Spécifications du Raspberry	42
4.2	Spécifications de l'ordinateur	42
4.3	Facteur de vitesse	46
4.4	Pas de temps minimal et fréquence maximale d'une boucle Robotran seule	47
4.5	Fonctionnalité des noeuds ROS pour l'application avec les sonars	56
4.6	Fonctionnalité des topics ROS pour l'application avec les sonars	57
4.7	Résultats des simulations ROS-Robotran avec sonars	65
5.1	Tableau résumant le simulateur à retour haptique réalisé par <i>Nicola De Lellis, Quentin Matérne et Romain Poletti</i>	69
5.2	Type de connexion et protocole de communication des liaisons physiques	71
5.3	Caractéristiques du servo-moteur	74
5.4	Caractéristiques du codeur	75
5.5	Temps d'exécution des fonctions de <i>Transmission_mod</i>	81
5.6	Ordre de grandeur des temps d'exécution des fonctions de l'ancien module de trans- mission	81
5.7	Fréquence et vitesse de communication des topics	83

Introduction

Au début de ce troisième millénaire, les systèmes multicorps (tels les robots, les voitures, les trains, les bras mécaniques des chaînes de production, ...) font désormais partie de notre quotidien. L'étude de leurs mouvements a mené à l'élaboration de logiciels informatiques simulant leur cinématique et dynamique avec précision. Un programme de ce genre, *Robotran*, a vu le jour au sein de l'Université catholique de Louvain-la-Neuve et plus particulièrement dans le département des systèmes mécatroniques, dynamiques et d'énergie électrique (MEED).

Lors d'une expérimentation en temps-réel, prévoir le comportement d'un dispositif mécanique afin de l'analyser et d'en déduire les modifications à y apporter permettrait d'augmenter les performances, l'automation ou encore la sécurité de ce mécanisme. Prenons l'exemple de la voiture autonome qui est au centre des avancées technologiques actuelles. Grâce à un logiciel de modélisation de multicorps embarqué, calculer à l'avance la trajectoire du véhicule en fonction de telle ou telle commande de vitesse et de direction offrirait la possibilité de choisir la meilleure manoeuvre d'évitement d'un obstacle par exemple. De plus, la révolution industrielle moderne pousse à l'automation de ces systèmes mécaniques. Cela entraîne la création de simulations impliquant des modules électroniques, électriques, mécaniques... tels les capteurs, actionneurs et effecteurs donnant ainsi naissance au terme "jumeau numérique".[1]

Il s'agit de créer une réplique numérique d'un objet, processus ou système réel. Cette réplique se met à jour et se modifie en fonction de l'interaction de son jumeau réel avec son environnement. Le jumeau numérique intègre également une intelligence artificielle et un apprentissage automatique qui ne sont pas couverts par ce travail. Ce mémoire se penche sur l'analyse de données d'un jumeau numérique à savoir, créer une interaction en temps-réel avec son environnement physique en utilisant des capteurs pour se repérer dans son milieu et des actionneurs pour permettre à l'humain d'interagir avec lui.

Le logiciel *Robotran* offre la possibilité de réaliser des simulations en temps-réel d'un système multicorps et d'interagir avec le modèle numérique à l'aide d'un clavier ou d'un joystick connecté à l'ordinateur réalisant la simulation. En revanche, exploiter les données de l'environnement du multicorps réel à l'aide de ses capteurs pour ensuite en modifier la configuration réelle par le biais de ses actionneurs n'est pas encore simulable en temps-réel par *Robotran*.

C'est ici qu'intervient le logiciel *Robot Operating System* (i.e. *ROS*). Ce système, fortement utilisé en robotique, permet l'exploitation de ces modules en temps-réel afin d'automatiser et contrôler un dispositif mécanique. En effet, il possède de nombreux outils pour manipuler des capteurs et publier leurs données sur un réseau et inversement il est possible de lire les données sur ce réseau pour les appliquer sur un actionneur. De plus, l'utilisation de *ROS* offre de nombreux autres intérêts comme la possibilité de distribuer les tâches nécessitant une utilisation élevée des processeurs sur plusieurs composants différents ou encore la possibilité de modifier facilement l'hardware du système sans devoir apporter de modification majeure au software.

L'objectif de ce mémoire est de permettre à *Robotran* de publier/souscrire aux données de ce réseau *ROS* afin de pouvoir simuler, en temps-réel, des dispositifs mécaniques automatisés ou non ainsi que de mettre à disposition de tous un système embarqué permettant de prévoir, analyser et contrôler le comportement d'un multicorps. Cette intégration ROS-Robotran se doit d'être généralisée afin de s'adapter à tout type de plateforme, véhicules, capteurs, actionneurs, ...

Ce mémoire introduit d'abord en deux chapitres le fonctionnement des deux logiciels *Robotran* et *ROS* en insistant sur les fonctionnalités utilisées dans le cadre de ce projet.

Le troisième chapitre se penche sur l'intégration de *Robotran* dans *ROS*. Il présente les contraintes de cette fusion et détail les intérêts de l'utilisation de *ROS*. Il se termine par l'explication du fonctionnement d'un projet *ROS-Robotran*.

Le quatrième chapitre applique l'utilisation de ce nouvel outil sur des cas simples afin d'en mesurer les performances et de prouver la capacité d'adaptation à différents actionneurs et capteurs.

Enfin, le dernier chapitre présente l'utilisation de *ROS-Robotran* dans le simulateur haptique de véhicules de l'UCLouvain conçu par *Nicola De Lellis* et *Quentin Materne* [2] et amélioré par *Romain Poletti* [3].

Finalement, ce travail se conclut par un bilan général sur l'implémentation et l'utilisation d'un projet *ROS-Robotran*.

Chapitre 1

Robotran

Robotran est un logiciel créé par le Centre de Recherche en Mécatroniques (*CEREM*) qui lui-même fait part de l'institut d'Ingénierie Mécanique, Matériel et Civile (*iMMC*) de l'Université Catholique de Louvain-la-Neuve (*UCLouvain*). Ce logiciel se base sur l'approche dite "symbolique" pour générer les équations du mouvement d'un modèle multicorps. Ces dernières sont ensuite résolues afin d'étudier le comportement cinématique et dynamique du système.

Robotran

1.1 Fonctionnement

La simulation d'un modèle multicorps passe par trois grandes étapes. Le système multicorps est d'abord modélisé dans l'éditeur graphique *MBsysPAD*, les équations du mouvement sont ensuite générées symboliquement par le module *MBsysTran* en langage *C*, *Python* ou *Matlab*, et enfin, résolues par le module *MBsysC*, *MBsysLab* ou *MBsysPy*.

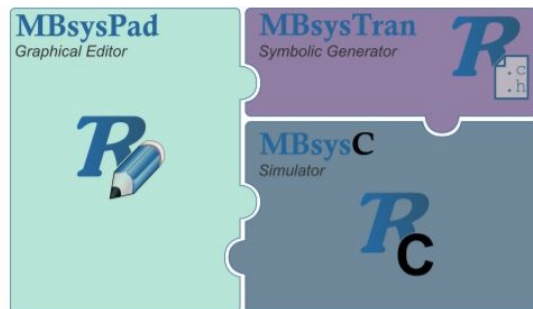


FIGURE 1.1 – Modules de Robotran pour une implémentation en C [4]

1.1.1 Modélisation d'un multicorps : MBsysPAD

Premièrement le modèle est créé sur l'éditeur graphique *MBsysPad*. On y définit différents corps rigides et ponctuels avec leur masse, leur matrice d'inertie et le centre de masse. Ces corps sont ensuite liés par une ou plusieurs articulations rotatives ou prismatiques. On peut également y spécifier des forces extérieures qui peuvent simuler un contact pneu-sol, roue-rail, un vent latéral sur un véhicule, etc ou des liens qui simulent une force d'action-réaction entre deux corps tels les ressorts, les amortisseurs, etc.

Robotran se base sur une structure du multicorps dite "arborescente". C'est-à-dire une structure où les corps ne forment pas de boucles fermées. Lorsque le système présente de telles boucles fermées, Robotran utilisera une procédure permettant de restaurer une structure arborescente en coupant un corps en deux parties ou en coupant un joint (ces *cuts* devront alors être déterminés dans MBsysPad).

Enfin, MBsysPad possède également un éditeur 3D offrant la possibilité de dessiner un modèle virtuel et de visualiser une animation de la simulation en temps-réel ou non.

1.1.2 Génération des équations du mouvement : MBsysTran

Deuxièmement, le module *MBsysTran* génère les équations du mouvement à partir du modèle dessiné sur MBsysTran à l'aide du calcul symbolique. Les intérêts du calcul symbolique sont nombreux. Spécialement dédié aux modèles multicorps, il permet une génération des équations du mouvement en quelques millisecondes en simplifiant les expressions et la trigonométrie et en éliminant les équations superflues. Il offre également une bonne portabilité aux équations générées, permettant de les utiliser dans d'autres domaines scientifiques. Les équations du mouvement peuvent alors être produites sous différentes formes :

La formulation semi-explicite ou dynamique directe Les équations du mouvement prennent la forme :

$$M(q, \delta)\ddot{q} + c(q, \dot{q}, \delta, frc, trq, g) = Q(q, \dot{q}) \quad (1.1)$$

avec pour n articulations :

- M : la matrice de masse, de taille $[n * n]$
- c : le vecteur dynamique nonlinéaire (comprenant les effets gyroscopique, centrifuge et de gravité ainsi que les contributions des forces et couples externes), de taille $[n * 1]$
- q : les coordonnées relatives généralisées (en position, vitesse et accélération), de taille $[n * 1]$
- δ : les paramètres dynamiques des corps (masse, centre de masse et matrice d'inertie), de taille $[10n * 1]$
- Q : les forces généralisées des articulations (force ou couple), de taille $n * 1$

La formulation semi-explicite ou dynamique directe avec contraintes L'équation (1.1) est valable pour un système multicorps sans contraintes (c'est-à-dire en structure arborescente et sans contraintes ajoutées par l'utilisateur). Cependant ce n'est pas le cas de la plupart des systèmes multicorps, il faut donc ajouter ces contraintes (ainsi que leurs dérivées première et seconde) aux équations du mouvement :

$$M(q)\ddot{q} + c(q, \dot{q}, g) = Q(q, \dot{q}) + J^T \lambda \quad (1.2)$$

$$h(q) = 0 \quad (1.3)$$

$$\dot{h}(q, \dot{q}) = J(q)\dot{q} = 0 \quad (1.4)$$

$$\ddot{h}(q, \dot{q}, \ddot{q}) = J(q)\ddot{q} + \dot{J}\dot{q}(q, \dot{q}) = 0 \quad (1.5)$$

avec pour m contraintes :

- $J = \frac{\partial h}{\partial q^t}$: la matrice Jacobienne des contraintes, de taille $[m * n]$
- h : les équations des contraintes que doivent satisfaire les coordonnées généralisées des articulations (q) à chaque pas de temps, cela comprend les contraintes géométriques imposées par le modèle et les contraintes algébriques spécifiées par l'utilisateur (pour par exemple lier le mouvement de translation et de rotation d'un système pignon-crémaillère)
- λ : les multiplicateurs de Lagrange associés aux contraintes, de taille $[m * 1]$

La formulation implicite ou dynamique inverse avec contraintes La dynamique inverse consiste à calculer les forces (ou couples) généralisées des articulations du multicorps pour une certaine configuration de celui-ci en position, vitesse et accélération et ainsi qu'en fonction des forces (et/ou couples) externes qui lui sont appliquées. Cette formulation est fortement utilisée en robotique pour par exemple calculer les couples dans les articulations d'un bras mécanique auquel on impose le suivi d'une certaine trajectoire. L'équation (1.2) peut alors être écrite sous une forme plus compacte :

$$Q(q, \dot{q}) + J^T \lambda = \phi(q, \dot{q}, \ddot{q}, \delta, frc, trq, g) \quad (1.6)$$

1.1.3 Résolutions des équations du mouvement : MBsysC

Enfin, le dernier module intègre les équations du mouvement. C'est également ici que l'utilisateur va pouvoir écrire, compiler et exécuter différentes fonctions déjà citées ci-dessus :

- les forces de liens (ex : les ressorts-amortisseurs)
- les forces extérieures (ex : contact pneu-sol)
- les contraintes algébriques (ex : pignon-crémaillère)
- le contrôle d'articulation (ex : un volant de voiture)
- ...

Ce dernier module propose 3 langages de programmation (*C*, *Python* et *Matlab*). Cependant, dans le cadre de ce mémoire, nous ne nous intéresserons qu'au langage *C*. En effet, c'est le seul permettant de réaliser des simulations en temps-réel (dont l'utilité sera démontrée par la suite). C'est également avec ce langage que la liaison avec le logiciel ROS a pu être établie¹.

Afin d'obtenir le mouvement du système multicorps, le module MBsysC passe par plusieurs étapes :

Chargement du modèle Il doit tout d'abord charger le modèle dessiné dans MBsysPAD dans une structure appelée MBSData.

1. L'utilisation du langage Python semble également être une solution envisageable mais elle n'a pas été vérifiée dans ce travail.

Partitionnement des coordonnées Il procède ensuite à un partitionnement des coordonnées généralisées q en un sous-ensemble de coordonnées indépendantes q_u de taille $(n - m)$ et un sous-ensemble de coordonnées dépendantes q_v de taille m qui donne lieu à une matrice Jacobienne régulière (ou inversible) J_v pour les coordonnées dépendantes.

Réduction en équations différentielles ordinaires et résolutions des boucles cinématiques Par après, il procède à une réduction des équations différentielles algébriques du mouvement (1.2) maintenant partitionnées sous la forme :

$$\begin{pmatrix} M_{uu} & M_{uv} \\ M_{uv} & M_{vv} \end{pmatrix} \begin{pmatrix} \ddot{q}_u \\ \ddot{q}_v \end{pmatrix} + \begin{pmatrix} c_u \\ c_v \end{pmatrix} = \begin{pmatrix} Q_u \\ Q_v \end{pmatrix} + \begin{pmatrix} J_u^t \\ J_v^t \end{pmatrix} \lambda \quad (1.7)$$

Cette réduction permet d'obtenir un système d'équations différentielles ordinaires :

$$M_r(q_u)\ddot{q}_u + F_r(\dot{q}_u, q_u) = 0 \quad (1.8)$$

Ce système d'équations simplifiées, qui est fonction des coordonnées indépendantes des articulations en position q_u et vitesse $\dot{q}_u$, permet de trouver facilement les accélérations $\ddot{q}_u$.

Comme on peut le voir, la réduction des équations passe par une élimination des coordonnées dépendantes q_v . Pour se faire, il faut résoudre les contraintes algébriques en position (1.3) (ou autrement dit fermer la boucle cinématique du modèle). Celles-ci étant généralement non-linéaires, elles sont résolues par la méthode itérative de Newton-Raphson avec comme conditions initiales les coordonnées indépendantes en position q_u ². Les contraintes dépendantes en vitesse, qui elles sont linéaires, sont résolues de manière classique avec comme conditions initiales les coordonnées indépendantes en vitesse $\dot{q}_u$. Enfin, les contraintes dépendantes en accélération, qui sont également linéaires, sont exprimées en fonction des coordonnées indépendantes en accélération $\ddot{q}_u$ qui, elles, sont inconnues.

Intégration temporelle Tout d'abord l'accélération $\ddot{q}_u(t)$ est obtenue à partir des conditions initiales ($q_u(t=0), \dot{q}_u(t=0)$) et l'équation (1.8). Il est maintenant possible de l'intégrer temporellement en utilisant une variable de substitution $\dot{y} = (\dot{q}_u \quad \ddot{q}_u)$ afin d'obtenir les nouvelles positions et vitesses ($q_u(t), \dot{q}_u(t)$). La variable de substitution permet de passer d'un système d'ordre 2 à un système d'ordre 1 résoluble par divers intégrateurs temporels classiques fournis par Robotran (Runge-Kutta 4, Dormand-Prince 5, Rosenbrock, ...).

Le schéma ci-dessous résume les différentes étapes à réaliser à chaque pas de temps pour une simulation, à savoir la résolution des contraintes, la réduction en équations différentielles ordinaires (ODE), et l'intégration temporelle de l'accélération.[4]

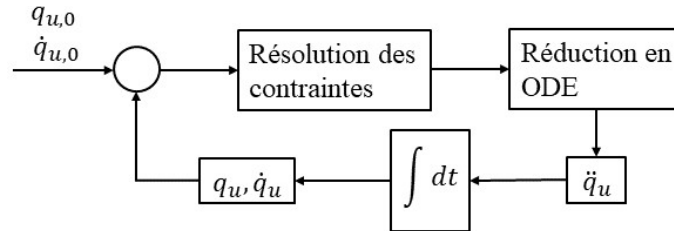


FIGURE 1.2 – Boucle temporelle de Robotran pour la résolution d'un système multicorps avec contraintes

2. Les conditions initiales sont donc très importantes afin de garantir une convergence de la méthode de Newton-Raphson.

1.2 Architecture d'un projet Robotran

Un projet Robotran possède l'arborescence suivante :

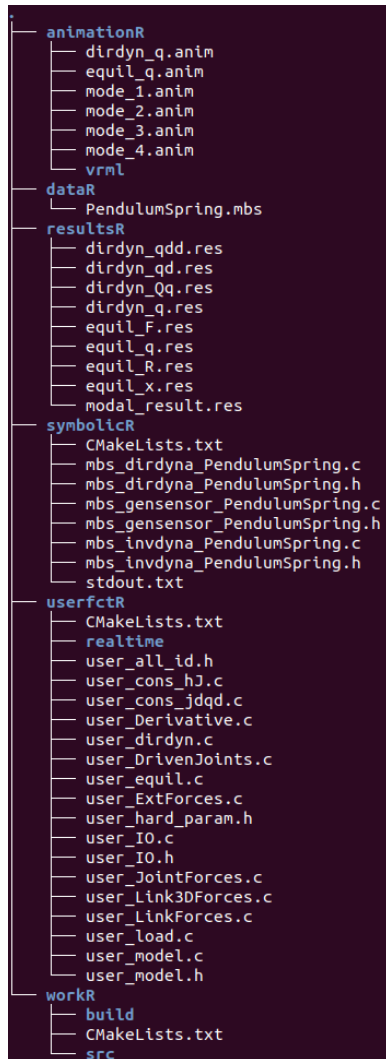


FIGURE 1.3 – Arborescence d'un projet Robotran

- Le dossier **animationR** : contient les fichiers des animations 3D des simulations réalisées.
- Le dossier **dataR** : contient le fichier .mbs qui comporte les informations introduites dans MBsysPAD.
- Le dossier **resultsR** : contient les fichiers des résultats des simulations (position, vitesse et accélération des articulations en dynamique directe, position d'équilibre, ...) pouvant être exploités pour des analyses modales, afficher la position d'une articulation en fonctions du temps, analyser les valeurs d'équilibres, ect.
- Le dossier **symbolicR** : contient les fichiers symboliques générés par MBsysTran qui comportent les équations du mouvement du modèle dessiné dans MBsysPad.
- Le dossier **userfctR** : contient les fichiers utilisateurs qui peuvent être modifiés pour y coder les modèles de forces extérieures, de forces de liens, de contraintes algébriques, ect. Il contient également le dossier **realtime** contenant les fonctions d'utilisateur pour les simulations en temps réel.
- Le dossier **workR** : contient la *main* dans le dossier **src** dans laquelle les différents modules de MBsysC sont appelés. Il comporte également un fichier *CMakeList* qui liste les instructions pour compiler et exécuter le projet. Le dossier **build** contient les fichiers de compilation et d'exécution générés par le compilateur *CMake* et la lecture du fichier *CMakeLists*.

1.3 Modèles multicorps utilisés

Afin de valider l'utilisation de Robotran dans ce projet, différents modèles multicorps ont été utilisés. Il s'agit d'un kart et d'une voiture car l'utilisation de ROS s'est focalisée sur l'intégration de capteurs et actionneurs dans un véhicule. Ces projets ROS-Robotran sont aux nombres de trois :

- Piloter la voiture à l'aide d'un clavier branché sur un dispositif autre que l'ordinateur hébergeant la session Robotran.
- Simuler un véhicule autonome à l'aide de sonars connecté au modèle Robotran de la voiture.
- Piloter la voiture et le kart à l'aide d'un simulateur à retour haptique composé d'un volant physique permettant de diriger le véhicule et de ressentir le retour de force causé par la dynamique du véhicule.

Les sections suivantes présentent ces deux véhicules ainsi que leur implémentation dans Robotran.

1.3.1 Voiture

La voiture dessinée dans le *MBsysPAD* de Robotran est un modèle coupé de chez *BMW*, la *135i*. Ce véhicule déjà implémenté avait été fourni par *Quentin Docquier* dans le cadre d'un projet du cours de Dynamique des véhicules à l'UCLouvain [5][6]. Les sections suivantes présentent les différentes caractéristiques de ce véhicule.

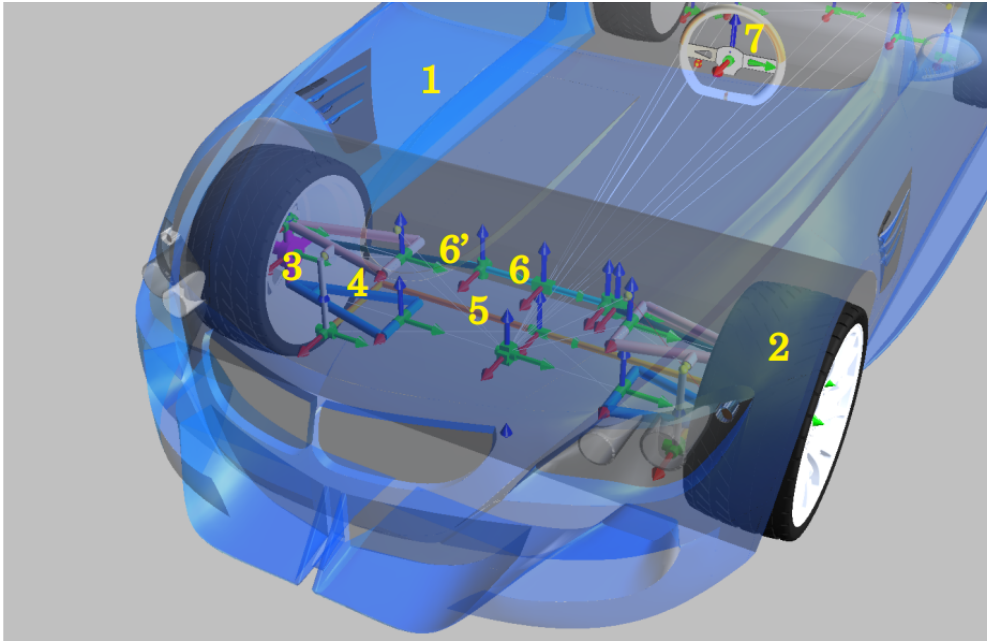


FIGURE 1.4 – Vue 3D de la voiture dans MBsysPAD

Caractéristiques générales

Cette section reprend les aspects généraux du véhicules ainsi que la manière dont ils ont été implémentés dans Robotran en détaillant chaque composant du véhicule.

Châssis Le châssis ($n^{\circ} 1$ sur la figure 1.4) est la pièce principale du véhicule. Celui-ci possède 6 degrés de liberté par rapport au sol (3 en translations et 3 en rotations). Il possède des points d'ancrage auxquels sont fixés les suspensions et la colonne de direction. Les distances entre les points d'ancrages et la géométrie des suspensions déterminent alors :

- la voie (distance entre la roue gauche et droite) : 1.50 m
- l'empattement (distance entre la roue avant et arrière) : 2.76 m

Roue Les 4 roues ($n^{\circ} 2$) sont représentées par des corps cylindriques possédant un degré de liberté en rotation par rapport à la fusée ($n^{\circ} 3$) qui est l'axe de maintien de la roue. Le contact pneu/sol est implémenté grâce à une force extérieure liée au corps de la roue. Cette force de contact est présentée plus en détails à la section 1.3.1 ci-dessous. Enfin, il est possible de faire accélérer le véhicule en appliquant un couple aux roues arrières (voiture à propulsion) et de le faire freiner en appliquant un couple de freinage aux roues avant et arrières (développé en détails à la section 4.2.4).

Il est important de détailler la géométrie des roues afin de notamment comprendre son impact et son utilisation dans un simulateur à retour haptique. En effet, les différents angles spécifiques aux roues ont une influence sur la direction et particulièrement sur le retour de force ressenti à travers elle. Les paragraphes suivants présentent ces angles et la section d'après portera sur leurs influences dans la colonne de direction.

L'**angle de chasse** (en anglais "caster angle") est l'angle formé entre la verticale et l'axe de pivot de la roue lorsque l'on regarde le véhicule de côté. Sa valeur est positive s'il est situé vers l'arrière du véhicule par rapport à l'axe de la roue. Cet angle a un impact direct sur la **traînée mécanique** (sur la figure 1.5 : "caster trail") qui est la distance entre la projection de l'axe de direction sur le sol et le point de contact de la roue sur le sol. Elle est positive si la projection est située devant le point de contact pneu sol. Le **déport de pivot** est la distance entre l'intersection de l'axe de rotation et le plan horizontal passant par le centre de la roue et le centre de la roue dans une vue de côté.

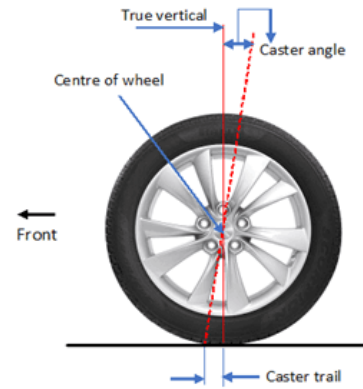
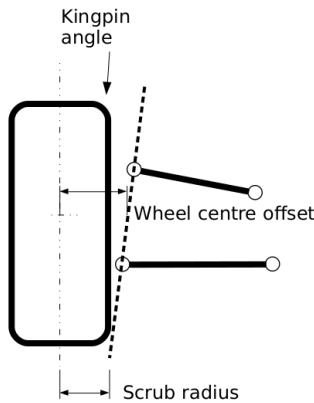


FIGURE 1.5 – Angle de chasse et traînée mécanique [7]



L'**angle de pivot** (en anglais "kingpin angle") est l'angle entre l'axe de direction et la direction perpendiculaire au sol située au point de contact lorsque le véhicule est vu de face. Il est considéré positif lorsque l'angle est situé vers l'intérieur du véhicule.

Le **rayon de frottement** (ou "scrub radius") est la distance entre l'intersection de l'axe de rotation avec le sol et le point de la roue au sol dans une vue de face. Sa valeur est positive s'il se trouve à l'intérieur du véhicule par rapport au point de contact pneu/sol.

FIGURE 1.6 – Angle de pivot et rayon de frottement [8]

Le tableau suivant reprend une gamme de valeur utilisée dans des véhicules routiers pour la géométrie des roues. Ces valeurs ont été trouvées lors d'un travail de recherche dans la littérature pour le projet du cours de Dynamique des véhicules [6] :

Géométrie de la roue	Angle de chasse	Traînée mécanique	Angle de pivot	Déport de pivot	Rayon de frottement
Valeur	3-7.5°	15-30 mm	6-14°	20-30 mm	10-15 mm

TABLE 1.1 – Valeurs usuelles pour la géométrie des roues

Enfin, il est possible de régler ces valeurs dans le modèle Robotran en modifiant la position des fusées et des roues. Avec des senseurs virtuels fournis par Robotran placés sur les joints à rotules délimitant l'axe de rotation des roues pour mesurer leurs positions et un peu de trigonométrie, les valeurs suivantes sont obtenues pour la voiture :

Géométrie de la roue	Angle de chasse	Traînée mécanique	Angle de pivot	Déport de pivot	Rayon de frottement
Valeur	5.54°	37.02 mm	$-5.44 * 10^{-3} \circ$	9.93 mm	-0.98 mm

TABLE 1.2 – Géométrie des roues de la voiture

Les valeurs obtenues pour l'angle de chasse correspondent aux données du tableau 1.1 avec une traînée mécanique légèrement plus grande. Le déport de pivot, quant à lui, est légèrement plus faible. L'angle de pivot et la rayon de frottement peuvent être supposés nuls sur ce véhicule.

Suspensions Il s'agit ici de suspensions à bras superposés (*n° 4*). Celles-ci offrent un excellent guidage des roues ainsi qu'un réglage aisé des épures³. Cela se fait au dépend d'un prix et d'un encombrement transversal élevés. L'amortisseur de la suspension est modélisé par un *lien* entre le bras inférieur de la suspension et le châssis que l'on caractérise par :

- une raideur : 33.3 kN/m
- un coefficient d'amortissement : 9.16 kN * s/m
- une longueur au repos : 0.66 m

Deux barres anti-roulis (*n° 5*) relient également les bras supérieurs des suspensions de gauche aux suspensions de droite. Ces barres, implémentées avec une certaine raideur en torsion, permettent d'atténuer le transfert de masse du véhicule lors d'un virage qui résulte en un effet de roulis autour de l'axe longitudinale. Ce roulis détériore le confort de conduite mais aussi le contact pneu/sol, ce qui peut s'avérer dangereux dans un virage. L'effet de pompage des suspensions n'est pas impactée par ces barres car lorsque les deux roues avant ou arrière vont monter ou descendre en même temps la barre n'appliquera aucune résistance.

Système de direction L'objectif du projet du cours de Dynamique des véhicules [6] était d'implémenter un système de direction et d'en étudier le comportement. En effet, le véhicule possédait tout d'abord une direction composée d'un système trois barres (une crémaillère (*n° 6*) et deux bielles (*n° 6'*)), on pouvait alors diriger le véhicule en imposant une position et vitesse à la crémaillère de direction. L'amélioration du système de direction s'est alors basée sur le schéma de la figure 1.7.

En appliquant des contraintes entre les différents corps, il a été possible de modéliser :

- un double joint de Cardan : permettant la modification de l'angle d'inclinaison du volant.
- un système pignon crémaillère : transformant le mouvement de rotation du volant en un mouvement de translation de la crémaillère. En simulant une crémaillère à rapport variable (c'est à dire avec un espacement des dents plus serrés au centre que sur les côtés), il est possible d'implémenter un ratio variable du système de direction. Ce ratio, définit comme le rapport entre l'angle de rotation du volant (*n° 7*) et celui des roues varie effectivement selon l'angle

3. "Représentation graphique (« f(x) ») de la variation des paramètres géométriques caractéristiques d'une suspension rigide « équivalente » en fonction du débattement de la roue"[5]. Ces paramètres sont notamment la demie-voie, l'empattement, le carrossage (angle entre le plan de roulement de la roue et la verticale en vue de face) et le parallélisme des roues (on parle de pincement si les plans de roulement des roues se croise devant l'essieu et on parle d'ouverture dans le cas contraire).

de braquage et vaut en moyenne 18 : 1. Il permet notamment de réduire les efforts dans le volant lors de la prise d'un virage et d'augmenter la précision de la direction.

Dans une voiture classique celui-ci sera plus élevé pour de faibles angles de braquage afin de limiter la sensibilité de la direction à de possibles mouvements parasites provenant du conducteur ou de déformations de la route. Ce ratio diminue pour de grands angles de braquage afin de limiter le nombre de tours de volant à faire lorsqu'on souhaite effectuer une manoeuvre de parking par exemple. La figure 1.8 affiche le ratio variable pour ce véhicule basé sur des valeurs et des allures de courbes trouvées dans la littérature [9]⁴.

Il est maintenant possible de diriger le véhicule en imposant une position et une vitesse à l'articulation du volant.

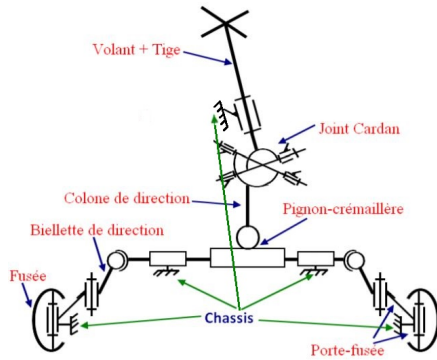


FIGURE 1.7 – Schéma cinématique du système de direction [10]

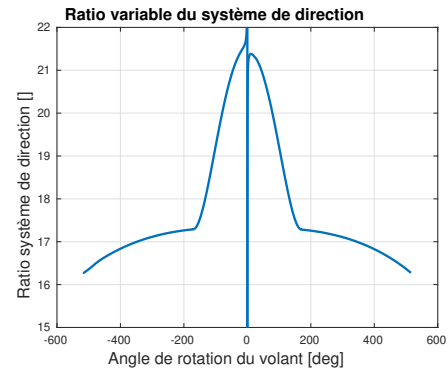


FIGURE 1.8 – Ratio variable du système de direction

Enfin, la tableau suivant reprend les masses et matrices d'inertie de chaque composants du véhicule.

Composants	Masse [kg]	Matrice d'inertie [kg.m ²]
Châssis	1480	$\begin{pmatrix} 389.68 & 0 & 0 \\ 0 & 2436.4 & 0 \\ 0 & 0 & 2641.5 \end{pmatrix}$
Roue	4x15	$\begin{pmatrix} 0.1 & 0 & 0 \\ 0 & 0.8 & 0 \\ 0 & 0 & 0.1 \end{pmatrix}$
Fusée et bras de suspensions	4x1	Matrice d'inertie négligeable
Conducteur & chargement	100	Matrice d'inertie négligeable
Crémaillère et biellette de direction	3	Matrice d'inertie négligeable
Volant	2	$\begin{pmatrix} 0.014 & 0 & 0 \\ 0 & 0.014 & 0 \\ 0 & 0 & 0.027 \end{pmatrix}$
Total	1649	

TABLE 1.3 – Répartition de la masse du véhicule

4. Le graphe présente des valeurs erronées tendant vers l'infini pour des angles de rotation du volant proche de zéro car l'angle de rotation des roues (au dénominateur) va également passer par zéro.

Couple de retour dans le volant et auto-alignement

Lorsqu'on relâche le volant de notre véhicule en plein tournant, celui-ci a tendance à revenir s'aligner sur sa position initiale. Il s'agit du phénomène d'auto-alignement qui est d'une grande importance pour la stabilité du véhicule. En effet, par les sensations qu'il transmet à travers le volant, il aide le conducteur à maintenir un virage en courbe constante et rétablir ensuite le véhicule en ligne droite.

La première cause est l'angle de chasse et la traînée mécanique qu'il induit. En effet, lors d'une prise de virage, la force centripète qui s'applique latéralement (F_{yf} sur la figure 1.9) au point de contact pneu/sol et qui est dirigée vers l'intérieur du tournant génère un effet de levier autour de l'axe de pivot des roues situé devant le point de contact pneu-sol pour une traînée mécanique positive (t_m). Cette effet de levier produit alors un couple autour de l'axe de pivot des roues qui se répercute dans le volant du véhicule. A la traînée mécanique, s'ajoute la traînée du pneu (t_p). En effet le point où s'appliquent les forces de contact entre le pneu et la route se trouve légèrement en arrière du centre de la roue. Le couple d'auto-alignement dû à la traînée mécanique et pneumatique vaut donc :

$$T_{trail} = F_{yf} * (t_m + t_p) \quad (1.9)$$

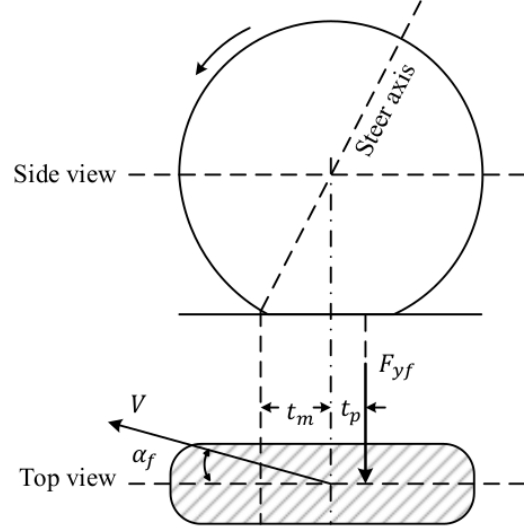


FIGURE 1.9 – Couple d'auto-alignement des roues en virage [11]

Le deuxième paramètre qui influence le moment d'auto-alignement est l'angle de pivot. Lorsqu'on regarde le véhicule de face, cet angle fait que l'axe de pivot des roues se trouve à l'intérieur du véhicule par rapport au point de contact pneu/sol. Lorsque le véhicule prend un virage, cet angle et la géométrie du système de direction font que l'avant du véhicule se soulève légèrement. A cause de la gravité, l'avant du véhicule va avoir tendance à redescendre et ainsi remettre les roues dans leur alignement initial. Cette effet est principalement observé à basse vitesse car à haute vitesse l'auto-alignement dû à la chasse est prépondérant à cause de la force centripète qui augmente quadratiquement avec la vitesse.

Contact pneu/sol et simulation d'un sol glissant

Afin d'implémenter un sol glissant, qui sera utilisé dans les applications suivantes de ROS-Robotran, il faut s'intéresser au contact pneu-sol.

Tout d'abord la zone de contact pneu-sol forme une ellipse qui sera simplifiée par un point dans Robotran. En ce point vont s'appliquer trois forces et trois moments de force (ces derniers permettront entre autres de prendre en compte la forme de contact simplifiée) :

- une force verticale F_z (selon l'axe de lacet) modélisée par un ressort-amortisseur qui simule le caoutchouc du pneu.

- une force longitudinale (ou force de traction) F_x (selon l'axe de roulis) et une force latérale F_y (selon l'axe de tangage) qui transmettent les efforts de freinage et d'accélération et qui permettent le maintien de la trajectoire lors de la prise d'un virage.
- les moments de force de renversement M_x et de résistance au roulement M_y ici supposé nuls.
- le moment de force d'auto-alignement M_z . Celui-ci va contribuer au moment d'auto-alignement des roues qui lui-même contribue au couple de retour ressenti dans le volant du véhicule.

En calculant la force verticale F_z à l'aide d'un modèle de ressort-amortisseur et en connaissant la cinématique des roues du véhicule (vitesse de rotation des roues, angle de carrossage⁵, angle de glissement⁶, ...), il est possible de calculer F_x , F_y et M_z à l'aide du modèle de contact pneu-sol de *Bakker*.

Le modèle de Bakker se base sur des modèles de pneus prédéfinis pour réaliser les calculs de forces et de couples. Ces modèles de pneu sont obtenus en réalisant des mesures expérimentales. Une "formule magique" écrite par *Bakker*, *Pacejka* et *Nyborg* :

$$y = D \sin(C \arctan(Bx - E(Bx - \arctan Bx))) \quad (1.10)$$

permet de générer une courbe à partir des données expérimentales en adaptant les paramètres B, C, D et E. Ce sont ces courbes qui permettent de déterminer les forces et moments de force du contact pneu/-sol.

Il existe d'autres manières d'établir le modèle d'un pneu en utilisant par exemple des méthodes de modélisation géométrique avancées ou la méthode des éléments finis. Néanmoins, après expérimentation dans Robotran, ces deux méthodes ne sont pas montrées aussi bonnes que la méthode de Bakker pour modéliser le contact pneu/sol. De plus elles présentaient l'inconvénient d'être beaucoup plus gourmandes en temps de calcul.

Enfin, pour citer un défaut de la méthode de Bakker, ces modèles de pneu sont basés sur du quasistatique. Ils ne conviennent donc pas à des simulations hautement dynamiques impliquant par exemple de grandes accélérations et freinages du véhicule. On peut néanmoins palier ce problème en utilisant des modèles de pneu dynamiques. Ces modèles prennent en compte la longueur de relaxation des pneus car en réalité les forces latérales de contact ne s'établissent pas immédiatement. Simuler la longueur de relaxation du pneu permet de prendre en considération ce temps d'apparition des forces. [5]

Le modèle de contact pneu/sol de Bakker utilise un coefficient d'adhérence entre le pneu et le sol de 0.8 (ce qui correspond à une route sèche, un contact sur une route mouillée aura tendance à avoir un coefficient proche de 0.5). Ici, afin de simuler une route glissante où l'adhérence est moindre, il faut donc diminuer les forces et moments de contact. Ainsi F_x , F_y et M_z seront multipliés par un facteur compris entre 0.5 et 1 pour représenter cette diminution de l'adhérence.

5. Angle entre le plan de roulement de la roue et la verticale en vue de face.

6. Angle entre l'orientation de la roue et la réelle direction du mouvement de la roue.

1.3.2 Kart

Le second véhicule présenté ici est un kart sans suspensions réalisé par *Romain Poletti* également dans le cadre du cours de *Dynamiques des véhicules* [5]. Ce véhicule n'est utilisé que dans la validation du simulateur à retour haptique 5.5.1 pour comparer son système de direction à celui de la voiture. Il est donc présenté de manière succincte et le lecteur est amené à se référencer au mémoire de *Romain Poletti* portant sur le sujet pour de plus amples informations sur l'implémentation de ce modèle [3].

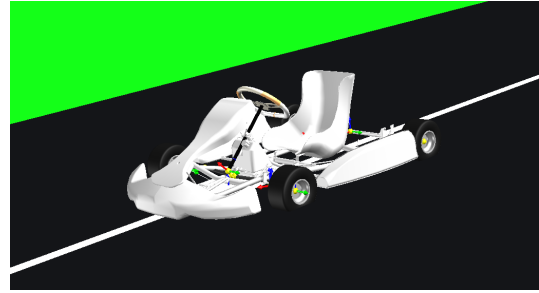


FIGURE 1.10 – Vue 3D du kart dans MBsys-PAD

Caractéristiques générales

Le **châssis**, logiquement plus petit que celui de la voiture, possède les caractéristiques suivantes :

- Voie avant : 1.06 m
- Voie arrière : 1.15 m
- Empattement : 2.76 m

Au niveau des **roues**, il est également possible de modifier la vitesse du véhicule en appliquant un couple aux roues arrières. Un régulateur PID sur la consigne de couple est implémenté afin de garantir une vitesse constante lors des simulations. Un tel régulateur est également utilisé pour la voiture, les deux véhicules ne possédant pas la possibilité d'utiliser des actionneurs/capteurs afin de permettre à l'utilisateur d'interagir avec la vitesse de son véhicule. Le modèle de Bakker est également utilisé pour implémenter le contact pneu/sol. Celui-ci nécessite tout de même une légère adaptation car il est initialement prévu pour des modèles de berlines d'environ 1500 kg alors que le kart (pilote compris) affiche une masse de 150 kg.

La **géométrie des roues** permet également une modification de l'axe de rotation des roues donnant les angles suivants :

- Angle de chasse : 10°
- Angle de pivot : 10°

Système de direction

Le système pour piloter le kart est basé sur le principe d'une "direction à manivelle" (*beel-crank steering*) différente de la direction pignon-crémaillère de la voiture. Un système de levier ($n^\circ 1$ sur le figure 1.11) permet de transmettre le mouvement de rotation du volant ($n^\circ 2$) aux deux bielles de direction ($n^\circ 3$) et fusées ($n^\circ 4$) des roues avant pour les faire tourner. C'est un système qu'on retrouve sur les karts et go-karts car il est facile et peu onéreux à installer. Néanmoins le ratio entre l'angle de rotation du volant et celui des roues est proche de l'unité (suivant la géométrie du système) et non-variable. L'angle de braquage maximal du volant est alors limité à $+/- 1$ radian contre $+/- 9$ radians pour la voiture. La sensibilité de direction d'un tel système est observée à la section 5.5.1.

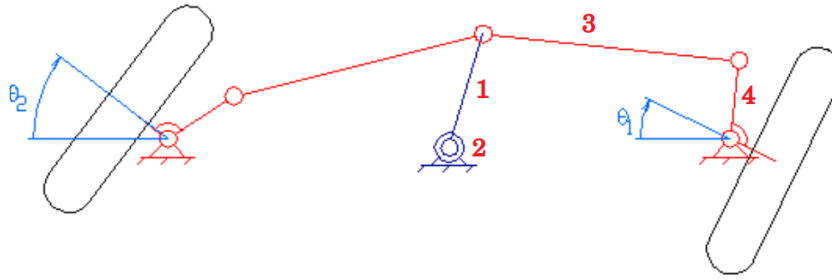


FIGURE 1.11 – Système de direction "à manivelle" [12]

Chapitre 2

Robot Operating System

ROS, pour Robot Operating System, est un logiciel open source offrant de nombreux outils informatiques utiles à la création de logiciels en robotique. Développé en 2007 par la société américaine *Willow Garage*, il est le premier projet collaboratif robotique à grande échelle. Ce principe de collaboration internationale se base sur le même que celui de *Linux*. En effet, son caractère open source permet à des individus de mettre à disposition de nouveaux *paquets* ROS utilisables par toute la communauté ROS qui peut alors les exploiter, les modifier, les améliorer, etc . Dans ROS, un *paquet* désigne un dossier comprenant les exécutables, un fichier CMakeLists pour la compilation de ceux-ci ainsi que d'autres fichiers utiles à ROS qui seront présentés ci-dessous. Il y a également de nombreuses sociétés qui investissent dans le développement de cette plateforme. De fait, cela leur permet de créer de nouvelles fonctionnalités propres à leurs robots sans pour autant devoir investir dans un nouveau système d'exploitation qui peut se montrer très coûteux.



Ce logiciel a vu le jour à cause de la complexité des systèmes robotiques. En effet, ceux-ci font intervenir de nombreuses compétences du domaine de l'ingénierie comme la mécanique, l'électromécanique, l'automatisation, les senseurs, la visualisation, la communication en temps-réel, ect. La conception d'un système robotique est donc freinée par l'avancement technologique dans chacune de ces branches auxquelles il doit s'adapter. Il est également ralenti par le grand nombre de personnes devant travailler en parallèle sur ce projet et qui doivent donc synchroniser leur travail pour créer un système robotique complet. ROS permet d'homogénéiser ces différentes compétences afin de rendre la conception d'un logiciel robotique plus aisée en réutilisant un maximum des outils déjà existants. Il est également possible de créer de nouveaux outils polyvalents qui permettent une interaction entre les différents composants, que ce soit au niveau du software ou de l'hardware. [13]

2.1 Fonctionnement

Afin de générer une application robotique, ROS se base sur quatre grands concepts : la *Plomberie*, les *Outils*, les *Capacités* et l'*Ecosystème*. [13]



FIGURE 2.1 – Concepts généraux de ROS [14]

2.1.1 La Plomberie

La Plomberie de ROS se base sur l'implémentation d'un middleware. Il s'agit d'un logiciel tiers qui permet la création d'un réseau de transfert de données entre différents composants informatiques. Afin de comprendre cette implémentation, il faut définir ici plusieurs éléments de la plomberie de ROS :

- les **topics** : c'est à travers eux que se fait l'interaction entre les composants informatiques. Il s'agit de bus informatiques¹ qui véhiculent des données.
- les **messages** : les données sont transportées sur les topics sous une forme normalisée appelée *message*². La normalisation des données, se fait via le langage IDL (Interface Description Language). ROS génère automatiquement une transcription entre ce langage et des langages de programmation courants tels que *C++*, *Python* ou encore *Java*.
- les **noeuds** : il s'agit des exécutables de ROS. Ceux-ci s'occupent d'effectuer des calculs, traiter les données, communiquer avec d'autres éléments hardware extérieurs, etc. Il est également possible de définir des objets ROS à l'intérieur de ces noeuds pour créer une interaction avec les topics, ils sont au nombre de deux et codable en *C++* ou *Python* :
 - l'objet ROS **Publisher** : cet objet dispose des fonctions nécessaires pour publier les données produites par le noeud sur un topic.
 - l'objet ROS **Suscriber** : cet objet permet de souscrire à un topic afin d'y récolter les données publiées dessus.

La polyvalence de ROS permet à un noeud d'interagir avec un topic sans se soucier du nombre de noeuds souscrivant et publiant déjà sur ce topic, ni de savoir sur quel composant hardware le noeud est exécuté. Il est donc possible de créer des topics liant plusieurs dispositifs hardware différents.

La figure à droite montre un exemple reprenant les différentes définitions introduites ci-dessus. Le *noeud_1* va par exemple produire une donnée valant 1. En définissant un objet *Publisher*, il peut stocker cette donnée dans un *message* normalisé (ici un entier de 8 bits, *int8*) et la publier sur le *topic*. Le *noeud_2* qui souscrit à ce topic grâce à l'objet *Suscriber* reçoit alors le message et peut exploiter la donnée contenue dans le message.

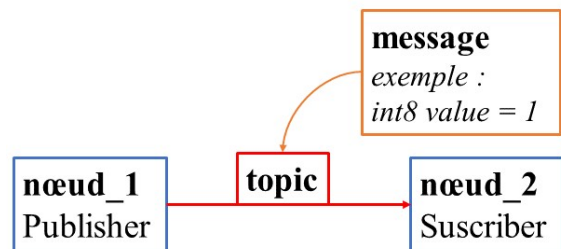


FIGURE 2.2 – Plomberie de ROS

1. "Un bus informatique est un dispositif de transmission de données partagé entre plusieurs composants d'un système numérique. Le terme dérive du latin omnibus (à tous) ; c'est le sens, d'un usage plus ancien, du terme bus en électronique. Le bus informatique est la réunion des parties matérielles et immatérielles qui permet la transmission de données entre les composants participants.", définition tirée de [15].

2. Les types de message autorisés par ROS sont : bool, int8, uint8, int16, uint16, int32, uint32, int64, uint64, float32, float64, string, time, duration et des tableaux à taille fixe et variable.

Mais encore, un noeud peut également demander un **service** à un autre noeud afin que celui exécute une tâche pour lui. Un service est donc composé de deux messages : le noeud client envoie un message de demande à un noeud offrant des services et attend ensuite le message de réponse.

Enfin, il est également possible de créer des **paramètres**. Ce sont des valeurs qui ne varient pas (ou peu) et auxquelles on peut accéder et/ou apporter des modifications dans n'importe quel noeud.

La figure 2.3 présente un exemple plus complet de l'utilisation de la Plomberie ROS d'un projet qui sera expliqué plus en détails à la section 4.1. On retrouve ici trois noeuds : *Rosbotran*, *listener* et *keyboard*. Les "chemins d'accès" */laptop* et */raspberry* permettent de spécifier dans quel groupe les noeuds sont exécutés afin de structurer son projet ROS. Les groupes représentent ici deux appareils différents, à savoir un ordinateur "classique" et un nano-ordinateur monocarte, un Raspberry Pi2. Le noeud *Rosbotran* publie des données sur le topic */raspberry/keyboard/chatter_torque* et les deux autres noeuds souscrivent à ce même topic afin d'y lire les données. Le noeud *keyboard* publie également des données pour le noeud *Rosbotran*. Il est important de préciser qu'un noeud peut donc à la fois publier et recevoir (en créant deux objet ROS, un *publisher* et *suscriber*) et ce, sur autant de topics désirés. Enfin, on peut observer que ROS fait abstraction de l'hardware sur lequel sont exécutés les noeuds.

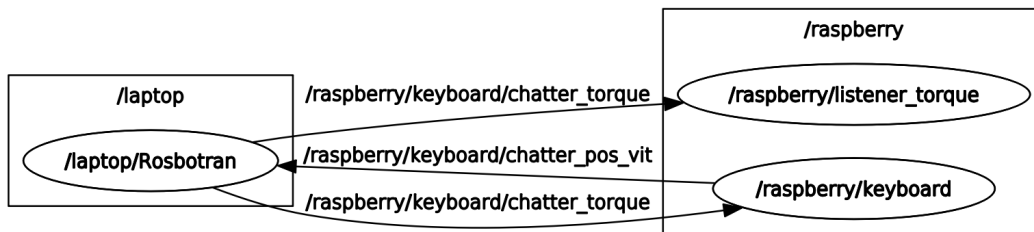


FIGURE 2.3 – Exemple de la plomberie ROS affichée par `rqt_graph`

2.1.2 Les Outils

Les Outils sont une des force de ROS. Il s'agit d'un ensemble de logiciels permettant de réaliser de nombreuses tâches. Nous trouvons notamment :

- *rqt_graph* : un interface graphique qui permet de visualiser les noeuds, topics et services utilisés (voir figure 2.3).
- *rosvbag* : un programme qui permet d'enregistrer les données publiées sur les topics afin de pouvoir les réutiliser pour des analyses postérieures.
- *rqt_plot* : un interface graphique qui permet d'afficher en temps réel sur des graphes les données publiées sur les topics.
- *catkin* : successeur de *rosvbuild*, il est le nouveau compilateur officiel de ROS. *Catkin* utilise le même flux opérationnel que CMake en y ajoutant des fonctionnalités permettant la gestion des paquets ROS ou encore la compilation de plusieurs projets indépendants en même temps.

2.1.3 Les Capacités

Les Capacités sont les bibliothèques qui implémentent les fonctionnalités d'un robot telles que la planification de tâches, de navigation, de visualisation de l'environnement, etc. Ainsi, si une fonctionnalité d'un robot est implémentée en respectant les normes d'une de ces bibliothèques, il est alors possible de

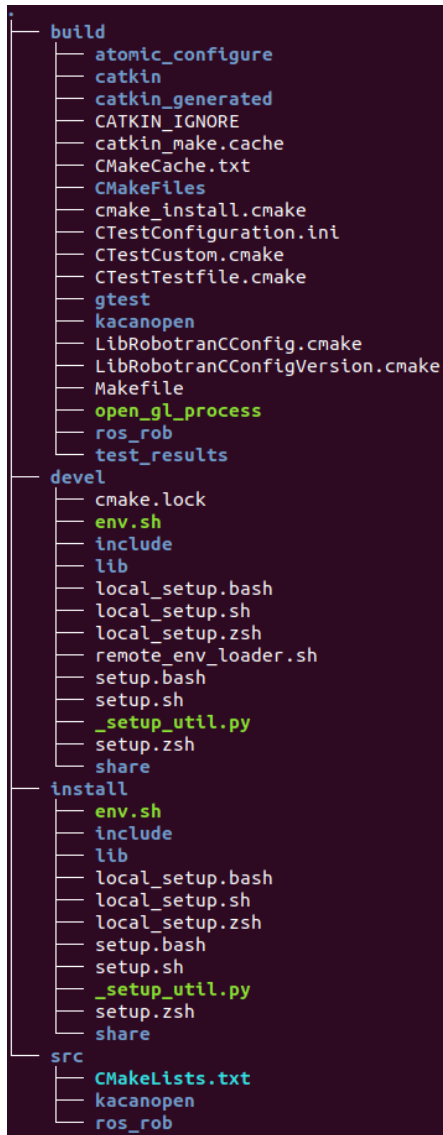


FIGURE 2.5 – Arborescence de l'espace de travail

- Le dossier **src** : contient les *paquets* ROS qui peuvent être créés ou alors téléchargés parmi les nombreux paquets que propose la communauté ROS. Sachant que ROS fait abstraction de savoir dans quel paquet un noeud est exécuté, il est de bon usage de regrouper ces noeuds par cohérence fonctionnelle dans des paquets différents. Ainsi nous retrouvons ici un paquet *ros_rob* (voir section 3.3.1) créé par l'utilisateur permettant l'intégration de Robotran dans ROS et un paquet *KaCanOpen* mis en ligne par *KITmedical* sur le site de ROS et qui, lui, permet de gérer des communications CANOpen ^a (voir section 5.3.1).
- Le dossier **build** : contient tout les fichiers de compilation générés par *catkin*. La *build* est composée de sous-dossier contenant les fichiers de compilation spécifiques à chaque paquet (par exemple ici *kacanopen* et *ros_rob*)
- Le dossier **devel** : contient les bibliothèques provenant de la compilation des paquets ainsi que les exécutables créés par ROS.
- le dossier **install** : contient les fichiers générés par la commande *install()* dans la CMakeList d'un paquet.

^a. Pour éviter toute confusion : ce paquet n'est utilisé à aucun moment sur l'ordinateur dans le cadre de ce travail, il sert ici simplement d'exemple.

Chapitre 3

Intégration de Robotran dans ROS

Ce chapitre présente l'objectif principal de ce mémoire, à savoir l'intégration de Robotran dans ROS. Il s'agit donc d'utiliser les bibliothèques et fonctions de Robotran dans un projet ROS et non pas l'inverse (par exemple créer un noeud ROS dans un projet Robotran). En effet, les fonctions de ROS ont besoin d'un compilateur particulier, *catkin*, afin de pouvoir générer des exécutables. Ce compilateur ne peut donc être utilisé que dans un espace de travail ROS (le *catkin_workspace*).



De plus, d'autres pistes de solutions ont été explorées mais ce sont toutes soldées par de grandes difficultés d'implémentation ou des résultats trop lent que pour pouvoir être utilisée dans une application en temps-réel :

- L'intégration de fonctions ROS dans un espace de travail Robotran qui n'était pas possible pour la raison expliquée ci-dessus.
- L'utilisation de l'outil *rosbridge* qui permettrait de publier des données sur des topics à l'aide de fonctions C mais qui s'est montré fastidieux à implémenter du aux faibles nombres d'informations sur le sujet.
- L'écriture des données de Robotran dans un fichier .txt en temps réel avec un noeud ROS lisant ce fichier .txt et publiant les données sur un topic. Cette méthode fonctionnait mais était peu rigoureuse et ralentissait fortement le débit d'information.

3.1 Objectif et contraintes

L'objectif principal vient d'être expliqué ci-dessus mais il nécessite quelques clarifications et contraintes. L'intérêt d'une fusion ROS-Robotran est de créer un outils de travail polyvalent utilisable par tout les utilisateurs de ces deux logiciels. Le but est donc de permettre une inclusion d'un projet Robotran aisée à l'utilisation et facilement reproductible.

Le nombre de modifications et d'ajouts au projet Robotran doit donc être le plus faible possible. Cela permet à l'utilisateur de créer son projet Robotran indépendamment et de l'insérer dans

l’environnement ROS en quelques manipulations. Il est par la suite toujours possible d’apporter des modifications au projet Robotran. De plus, avoir un faible nombre de modifications permet à l’utilisateur de facilement manipuler différents modèles multicorps pour ses simulations avec ROS. Enfin, ces modifications doivent minimiser leur impact sur le temps d’exécution de la boucle temporelle de Robotran afin de garder la possibilité de réaliser des simulations en temps-réel à haute fréquence. Une dernière exigence pour Robotran est d’éviter une modification de ses fichiers sources par l’utilisateur. De fait, ils sont propriétés de l’UCLouvain-CEREM et une modification majeure de ceux-ci pourraient être une entrave au bon fonctionnement du programme.

Les contraintes sur la complexité du paquet ROS utilisé pour l’intégration sont moindres (paquet qui sera appelé *ros_rob* dans la suite de ce travail). En effet, ce paquet est déjà créé et disponible en open-source sur le site de Robotran à partir de fin juin 2020 ainsi que sur le site GitHub [19]. Il doit cependant rester facile d’utilisation et de compréhension.

Enfin, une dernière obligation majeure est le système d’exploitation de l’ordinateur. En effet ROS ne supporte officiellement que les distributions Linux et Debian et depuis peu Windows 10. De fait, Microsoft a annoncé lors de sa *Build Conference* à Seattle en mai 2019 que la version expérimentale de ROS sur Windows 10 était maintenant officielle. ROS propose également des versions expérimentales pour Arch Linux, Gentoo, OS X (Homebrew) et OpenEmbedded/Yocto. Cependant les différents paquets et l’ensemble du projet ont été réalisés sous Linux. Au moment du choix du système d’exploitation pour ce projet en octobre 2019, la version officielle Windows n’était pas encore disponible sur le site de ROS. De plus, l’Ecosystème de ROS, qui est une de ses forces majeures, a été fortement développé pour Linux. Le choix s’est alors porté vers ce système d’exploitation qui était plus sûr tant aux niveaux des fonctionnalités offertes par ROS que par l’aide et les explications fournies par la communauté ROS. Bien que la plupart des paquets soient compatibles avec la nouvelle version de Microsoft, ceux créés dans le cadre de ce projet n’ont pas été testés sur ce système d’exploitation. Pour une réutilisation des données de ce projet, il est donc conseillé d’utiliser les deux logiciels ROS et Robotran sous Linux.

3.2 Intérêts de ROS

Bien que l’intégration de Robotran dans ROS ne soit pas axée sur la robotique en tant que telle, ROS présente de nombreux avantages pour celle-ci. C’est principalement son architecture de communication ainsi que sa polyvalence à pouvoir être utilisée sur différents software et hardware qui ont été exploités ici.

3.2.1 Utilisation d’autres composants

Robotran est, pour rappel, un logiciel permettant la simulation en temps réel ou non de systèmes multicorps. Il peut donc être intéressant d’utiliser les valeurs retournées par Robotran (par exemple les positions et les forces dans les articulations) afin de les utiliser en temps réel sur de vrais systèmes multicorps (robot, voiture, train, ...). Dans l’autre sens, utiliser des dispositifs tels un volant, un capteur de mouvement, un capteur sensoriel, etc afin de modifier la dynamique du système multicorps en temps réel pourrait apporter un avantage à l’utilisation du logiciel Robotran.

Toutefois, les actionneurs et capteurs utilisés sur de vrais modèles possèdent leur propre type de communication (généralement standardisée) et/ou leur propre langage de codage. Utiliser ceux-ci directement dans un projet Robotran demande donc une modification conséquente de ce dernier.

Cela peut donc diminuer les performances de la simulation en augmentant la charge des processeurs et de la mémoire vive. De plus, le grand nombre de modifications à apporter au projet limite la diversité des expériences possibles. Par exemple passer d'un modèle de voiture à un autre nécessite de rapporter toutes les modifications faites à un nouveau projet Robotran.

C'est ici que la combinaison des logiciels Robotran et ROS est intéressante car elle permet d'utiliser des données de capteurs directement dans Robotran en passant par ROS. L'avantage de ROS est qu'il fournit de nombreux paquets permettant une manipulation aisée de différents types d'actionneurs et capteurs et une utilisation des données qu'ils reçoivent/envoient sur des topics. Lorsque ces données se trouvent sur un topic elles deviennent accessibles en temps réel et quasi instantanément par tous les noeuds ROS se trouvant sur le réseau et souscrivant à ce topic. Un noeud qui exécute la simulation du modèle multicorps peut alors utiliser ces données à de très hautes fréquences.

Par ailleurs, Robotran est la plupart du temps exécuté sur un ordinateur "classique" et ce dernier offre une connexion limitée avec des actionneurs et capteurs que ce soit au niveau du hardware ou des protocoles de communication. Cependant ROS peut être installé sur des ordinateurs mono-cartes tels les Raspberry's qui, eux, offrent un plus large panel de connexions avec d'autres composants grâce à leurs nombreuses lignes d'entrée/sortie. L'annexe A illustre les différents capteurs et actionneurs pouvant être liés à un Raspberry. Enfin, le Raspberry (à l'aide d'une extension), est capable de communication CANopen qui peut être utilisée dans de nombreuses applications en automatisation industrielle (voir section 5.3.1).

3.2.2 Distribution des tâches sur différents composants

La simulation d'un modèle multicorps, en fonction du nombre de corps, du type et de la complexité de leurs liaisons ou encore de la dynamique de la simulation, peut avoir un impact conséquent sur l'utilisation de la mémoire vive et des processeurs de l'appareil sur lequel elle est exécutée. A celle-ci peut s'ajouter d'autres fonctions pour communiquer avec d'autres dispositifs comme expliqué ci-dessus, des fonctions réalisant des calculs intermédiaires, un interface graphique permettant la visualisation du modèle en temps réel, etc. ROS pourrait permettre d'exécuter ces fonctions sous forme de noeud sur d'autres composants hardware qui peuvent communiquer/recevoir des informations à/de l'appareil exécutant Robotran à travers les topics. Ainsi il est possible de créer des projets plus complexes (nécessitant des fonctions extérieures à Robotran) sans impacter sur la vitesse de calcul de la boucle de dynamique directe de Robotran qui reste totalement désolidarisée du projet complet.

3.2.3 Réseau ROS universel et flexible

Comme déjà explicité ci-dessus, ROS peut établir un réseau de communication entre divers dispositifs. Les conditions sont les suivantes, il doit y avoir une connexion complète et bi-directionnelle entre chaque appareil et que chaque hardware doit posséder un nom que les autres appareils peuvent détecter (par exemple une adresse IP fixe pour les connexions SSH). Lorsque le réseau est établi, un hardware est désigné comme *master* et les autres comme *slave(s)*. Le *master* gère alors les noms des noeuds, topics, services et paramètres ROS lors de leur création. Cela permet aux noeuds de se trouver l'un l'autre, de s'envoyer des données via les topics et d'invoquer des services et ce, sans se soucier de savoir sur quel hardware le noeud est exécuté. L'intérêt est qu'on peut alors changer un des dispositifs du réseau (qui est défectueux, dont on veut augmenter les performances, ...) et exécuter sur celui-ci les noeuds du précédent appareil sans devoir y apporter de modifications majeures.

3.2.4 Gestion et automatisation du procédé

ROS permet la gestion de projet complexe avec de nombreux noeuds. L'outil *roslaunch* permet notamment de lancer plusieurs noeuds simultanément sur différents dispositifs hardware. Cela offre le possibilité de lancer un projet complet avec une simulation sur Robotran et des noeuds gérant les communications extérieures avec divers capteurs et actionneurs en une seule ligne de commande. L'utilisation des *paramètres* ROS se montre également très utile. Pour rappel, il s'agit de variables pouvant être lues et modifiées sur tout les noeuds du réseau. Elles permettent donc une communication rapide entre ces derniers pour autoriser telles ou telles actions (avec notamment l'utilisation de *booléen*) en fonction de l'avancement du processus.

3.3 Fonctionnement

Cette section présente le procédé utilisé afin d'exécuter et exploiter un projet Robotran à partir de ROS. Le rendu final consiste en un paquet ROS (nommé *ros_rob*) téléchargeable sur le site de Robotran ou Github où se trouve également un tutoriel expliquant l'installation, la compilation et l'utilisation de ce paquet. L'objectif ici est de présenter de manière plus théorique la fusion de ces deux logiciels en y énumérant les modifications apportées.

3.3.1 Paquet *ros_rob*

Le paquet suit l'architecture d'un projet ROS présenté à la section 2.2 et dont l'arborescence correspond à la figure 2.4. En voici la composition plus détaillée pour les parties concernées par Robotran :

le dossier msg	On y déclare des fichiers <i>.msg</i> permettant de créer des structures de messages personnalisées pour les topics ROS afin d'y transporter nos données (dans le cas de Robotran il s'agira principalement des positions, vitesses, forces des articulations). ROS possède également des librairies de messages classiques pour véhiculer des structures d'informations pour des senseurs, de l'odométrie, ect.
le dossier src	Celui-ci contient le ou les projet(s) Robotran en entier (sans dossier <i>build</i>).
le fichier CMakeLists.txt	Celui-ci liste les instructions de compilation du paquet <i>ros_rob</i> et comporte également un lien vers la <i>CMakeLists</i> d'un projet Robotran utilisé. Cette dernière va alors être lue par le compilateur de ROS <i>catkin</i> qui va générer un exécutable ROS à partir de la <i>main</i> de Robotran.

3.3.2 Projet Robotran

Pour rappel, l'objectif était de permettre une inclusion d'un projet Robotran en y apportant le moins de modifications possible. Ces différents changements ainsi que leur raison sont présentés ci-dessous :

Le fichier CMakeLists Il comporte deux modifications. Tout d'abord le projet Robotran nécessite une version modifiée de *MBsysC*. Cette version de *MBsysC* permet la modification de la *main* en langage *C* de Robotran en *C++* compilable par ROS. Elle spécifie également un chemin d'accès différent vers l'exécutable *OpenGL* qui est un interface de programmation utilisé pour l'animation

3D d'une simulation Robotran. En effet, son emplacement est différent d'un projet Robotran "classique". C'est donc dans la CMakeLists de Robotran que le chemin d'accès vers cette version de *MBsysC* est spécifié.

Il faut également y indiquer l'utilisation de nouvelles librairies :

- Les librairies **ROS** pour l'utilisation des fonctions ROS afin de publier et recevoir des données sur des topics.
- Les librairies **Pthread** nécessaires pour le multithreading présenté ci-dessous.

Le fait que tous les fichiers générés par la compilation de la CMakeLists se retrouvent dans le même dossier *ros_rob* dans la *build* de ROS (voir figure 2.5) est un désavantage. En effet, si l'on désire intégrer plusieurs projet Robotran dans ROS, il n'est pas possible de tous les compiler en même temps car certains fichiers déjà présents dans le dossier *build/ros_rob* correspondants à la compilation d'un projet Robotran en particulier vont empêcher la compilation d'un nouveau projet. Pour remédier à ce problème, il suffit de, à chaque compilation d'un nouveau projet Robotran (ou après la modification d'un autre projet que celui actuellement compilé), spécifier un seul lien vers une CMakeLists Robotran dans la CMakeLists de *ros_rob* et supprimer le contenu du dossier *build/ros_rob*. La suppression du dossier *build/ros_rob* n'entraîne pas la perte des exécutables qui, eux, sont stockés dans le dossier *devel* de ROS. Il est donc toujours possible d'exécuter les autres projets Robotran préalablement compilés.

La main Comme déjà dis, ROS génère des exécutables à partir de fichier *C++* ou *Python*. C'était donc une des raisons de l'utilisation du module *MBsysC* car il est aisé de modifier la *main C* en *main C++*. A partir de là, il est possible d'initialiser un noeud ROS à l'intérieur de la *main* et d'en utiliser toutes les fonctionnalités.

Le multithreading Tout d'abord voici la définition de ce qu'est un thread en programmation et de l'intérêt du multithreading selon l'école informatique Supinfo [20] :

- **Qu'est-ce qu'un thread ?** : "Un thread est une file d'exécution, c'est-à-dire, que l'on ne peut accéder à l'instruction suivante tant que l'instruction actuelle n'est pas terminée."
- **A quoi cela sert le multithreading ?** : "Le multithreading sert à s'affranchir de la simple file d'exécution. Cela permet ainsi la création d'une nouvelle file d'exécution concourante de la première. De ce fait, on peut réduire les temps d'exécution de chaque instruction car l'on peut séparer les différents travaux entre différents threads."

La *main* est décomposée en plusieurs threads :

- Un thread qui exécute la *main* "classique" de **Robotran** à savoir : le chargement du modèle multicorps, les équilibres statiques et quasi-statiques, les analyses modales et la dynamique directe (ou inverse).
- Un thread qui crée un objet ROS **Publisher** : cet objet offre les fonctions nécessaires pour publier les données générées par Robotran sur un topic.

- Un thread qui crée un objet ROS **Subscriber** afin de récolter les données d'un topic et de les exploiter dans Robotran.

L'avantage du multithreading est exploité dans les simulations en temps-réel. En effet, la boucle temporelle de Robotran qui résout les équations du mouvement tourne en parallèle des fonctions ROS et sa fréquence d'exécution n'est presque pas impactée. La raison de l'impact de ROS grâce au multithreading sur la fréquence d'exécution de Robotran est expliquée ci-dessous.

Robotran travaille avec une structure appelée *msb_data*. Celle-ci contient toute les informations sur le modèle généré dans *MBsysPAD* (les positions, vitesses et accélérations des articulations, les masses des corps, etc). Lors de la résolution des équations du mouvement, Robotran a donc besoin constamment d'accès à cette structure afin d'y exploiter les informations et/ou d'y modifier les données. Lors de la simulation, les fonctions ROS ont également besoin d'accès à des données de *mbs_data* pour les mêmes raisons que Robotran. Cela conduit à une erreur de segmentation qui se déclenche lorsque plusieurs applications tentent d'avoir accès à un même emplacement mémoire en même temps. Le multithreading offre une solution à ce problème : les *mutexs*. Il s'agit d'un système de verrou donnant une garantie sur la viabilité des données manipulées par les threads¹. Des fonctions permettant de verrouiller et déverrouiller ce verrou pour chaque thread sont stockées dans les *user model* de Robotran². Ainsi, lors de l'exécution de la boucle temporelle du module dynamique de Robotran, il est possible de donner accès à ROS à *mbs_data* en appelant ces fonctions. Pendant de très courts instants, le boucle de Robotran est alors verrouillée. Cependant, l'exemple de la section 4.1 démontre que cet impact sur la vitesse de résolution des équations du mouvement peut être considéré comme négligeable. Enfin, notons que l'utilisateur pourrait également augmenter le nombre de threads et d'objets ROS s'il souhaite par exemple publier des informations sur des topics à plusieurs moments différents.

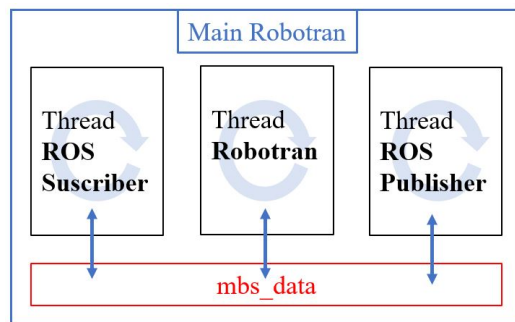


FIGURE 3.1 – Multithreading dans la main Robotran

1. Définition tirée du site *Developpez.com* [21]

2. Les *user model* sont des structures permettant à l'utilisateur de stocker différents types de variable (scalaire, vecteur, etc) ou des structures personnalisées qui dans ce cas ci stockent des pointeurs vers des fonctions.

Chapitre 4

Validation sur des cas unitaires

4.1 Commande d'un modèle Robotran à l'aide d'un actionneur extérieur

Cette section présente un exemple de l'utilisation du paquet *ros_rob* ainsi que son impact sur les performances d'une simulation en temps-réel de Robotran. Le multicorps utilisé ici est la voiture présentée à la section 1.3.1. Cet exemple implémente une boucle de contrôle permettant de piloter le véhicule en temps-réel simulé sur un ordinateur à l'aide d'un clavier physique sans fil connecté à un Raspberry et de recevoir en retour le couple présent dans le volant.

4.1.1 Architecture de l'hardware

Cet exemple nécessite l'utilisation de deux ordinateurs :

Un **Raspberry Pi2 Modèle B** : il s'agit d'un nano-ordinateur monocarte à processeur ARM pouvant héberger différents systèmes d'exploitation. Le système choisi ici est Ubuntu Mate. Il s'agit d'une distribution officielle de Linux utilisant l'environnement bureau Mate légèrement différent de GNOME qui est celui d'Ubuntu par défaut. Les fonctionnalités sont elles aussi sensiblement pareilles et ce système d'exploitation présente l'avantage d'être orienté pour les ordinateurs plus anciens ou moins puissants tels que les Raspberrys. Ce système d'exploitation permet également une installation et utilisation de ROS similaire à celle sur un ordinateur classique sous Ubuntu.

En plus de son faible prix, le Raspberry a l'avantage d'être équipé de 40 broches GPIO. Les connecteurs GPIO, pour *General Purpose Input/Output*, sont des ports fortement utilisés dans le domaine des microcontrôleurs. En jouant le rôle d'entrée ou de sortie, ils permettent le branchement et l'utilisation d'électronique embarquée sur le Raspberry (voir également l'annexe A). Cet avantage peut être exploité par ROS afin d'utiliser différents types de capteurs et actionneurs.



FIGURE 4.1 – Raspberry Pi 2 Model B [22]

Modèle	Processeur	Cadence Processeur	Nombre de coeurs	RAM
Pi2 Modèle B	ARM Cortex-A7 CPU	900 MHz	4	1 Gb

TABLE 4.1 – Spécifications du Raspberry

Un **ordinateur "classique"** sous Linux Ubuntu. Celui héberge également une session ROS et possède la puissance nécessaire pour réaliser des simulations de modèles multicorps avec Robotran.

Modèle	Processeur	Cadence Processeur	Nombre de coeurs	RAM
Asus Zenbook UX530X	Intel(R) Core(TM) i7-7500U CPU	2.70 GHz	2, 4 processeurs logiques	8 Gb

TABLE 4.2 – Spécifications de l'ordinateur

ROS impose tout d'abord une connexion complète et bi-directionnelle entre les deux appareils. Le protocole de communication SSH¹ a donc été utilisé pour établir une connexion entre l'ordinateur et le Raspberry. Celui-ci peut se faire à distance à travers un réseau *Wifi* mais par un soucis de rapidité et de stabilité de communication, un câble Ethernet Cat-5e² a été employé pour connecter l'ordinateur au Raspberry (qui de plus, ne possède pas de module *Wifi* de base). En spécifiant une adresse IP fixe pour les deux appareils, les contraintes pour une réseau de communication ROS sont bien respectées. Il faut signaler que l'ordinateur utilisé pour les tests ne possédait pas de port Ethernet, un adaptateur USB-Ethernet TP-LINK UE300 a donc été utilisé. Cet adaptateur permet un taux de transfert de données Ethernet maximal de 1 *Gbit/s* ce qui est démontré comme n'étant pas une entrave à la vitesse de communication pour ROS-Robotran.

Le schéma suivant reprend les différents composants hardware, les connexions physiques entre eux ainsi que leur alimentation nécessaire :

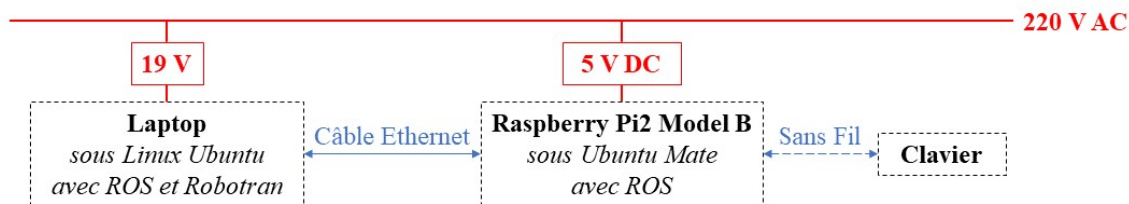


FIGURE 4.2 – Architecture physique globale

4.1.2 Architecture du software

L'architecture de l'environnement ROS avait déjà été introduite comme exemple sur la figure 2.3. En y représentant les différents threads, elle prend la forme suivante :

1. SSH, pour *Secure Shell*, est un protocole de communication sécurisé couramment utilisé pour prendre le contrôle d'un ordinateur à distance. Il permet un transfert de données chiffrées à travers un réseau et se base sur le principe client/serveur.

2. La catégorie 5e des câbles Ethernet spécifie que ces derniers peuvent supporter une vitesse de transmission de données maximale de 1 *Gbit/s*.

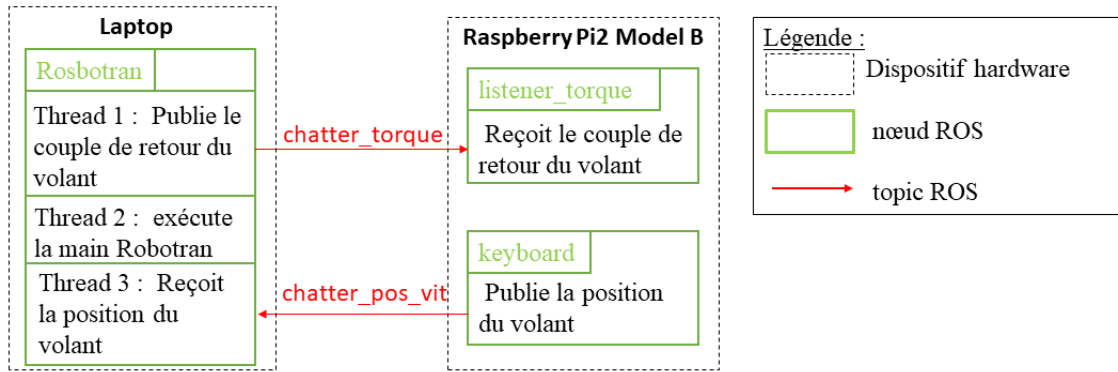


FIGURE 4.3 – Architecture globale de ROS

Le noeud *Rosbotran* est celui généré par le paquet *ros_rob* dont les fonctionnalités ont déjà présentées.

Le noeud *Keyboard* gère l'utilisation du clavier physique et modifie la position du volant à l'aide des touches directionnelles gauche et droite et d'un envoi des informations sur le topic *chatter_pos_vit*³. Ce noeud est adapté du paquet *ros-keyboard* fournit par l'*LRSE* (Laboratoire de robotique et systèmes embarqués de Buenos Aires)[23].

Le noeud *listener_torque* reçoit le couple de retour dans le volant calculé par Robotran et publié sur le topic *chatter_torque*. Il n'apporte aucune plus-value à cet exemple mais a également été implémenté pour garder un protocole de communication semblable à celui qui sera utilisé pour le simulateur à retour haptique (voir chapitre 5) afin de pouvoir en mesurer les performances.

4.1.3 Architecture de contrôle

Avant de présenter le protocole de communication il est important de définir les fonctions *CallBack* de ROS. Il s'agit de fonctions ne renvoyant rien (donc de type *void*) écrites par l'utilisateur et qui sont mises en argument lors de la création d'un objet ROS *Listener*. Ces fonctions prennent en argument le message ROS correspondant au topic auquel souscrit le *Listener*. Elles permettent donc de manipuler les données reçues sur le topic. Il est également possible de mettre une structure en argument afin de pouvoir apporter des modifications à d'autres variables à l'intérieur de ces fonctions *CallBack*.

Ces fonctions sont appelées par la fonction *ros : :spin()* (qui les appelle en boucle à la fréquence maximale que permet l'appareil) ou par *ros : :spinOnce()* (qui les appelle une seule fois mais qui peut être utilisée dans un boucle *for* ou *while* afin de contrôler la fréquence d'appel soi-même). Il faut préciser que ces deux fonctions appellent seulement les fonctions *CallBack* initialisées dans le noeud ROS où *ros : :spin()* est appelé. Enfin, lorsque ces fonctions sont appelées, elles s'exécutent seulement si un nouveau message a été publié sur leur topic (cela correspond aux actions désignées par une flèche verte sur la figure 4.4) sinon elles sont ignorées (c'est à dire les flèches rouges sur la figure 4.4).

L'architecture du protocole de communication est basée sur trois boucles tournant indépendamment l'une de l'autre :

3. Le topic *chatter_pos_vit* permet également de transporter la vitesse du volant mais celle-ci n'est pas utilisée dans cet exemple.

- La boucle du noeud *listener_torque* qui lit chaque valeur du couple publiée sur le topic *chatter_torque*. La fréquence de la boucle dépend donc de la fréquence à laquelle les données sont publiées.
- La boucle du noeud *Rosbotran* qui correspond à la boucle temporelle qui résout les équations du mouvement du système multicorps (voir figure 1.2). Pour une simulation en temps-réel, sa fréquence correspond au pas de temps choisi pour l'intégration temporelle. Ce pas de temps correspond alors au temps disponible afin de résoudre les équations du mouvement à chaque itération. On observera donc une baisse de la fréquence s'il n'est pas possible de respecter cette vitesse de résolution des équations et d'affichage de la simulation. A chaque pas de temps, la boucle vérifie, à l'aide de la fonction callback (*Cb_2* sur l'image), si une nouvelle position du volant q a été envoyée depuis le clavier du Raspberry. Si oui, elle met à jour la position du joint correspondant au volant du multicorps. Si non, la valeur précédente est gardée. Elle publie ensuite, à chaque pas de temps, la valeur du couple de retour T dans le volant.
- La boucle du noeud *Keyboard* qui elle, tourne à la plus haute fréquence que le permet le Raspberry afin de mettre à jour la valeur q pour la position du volant à partir de l'interaction avec les touches directionnelles du clavier. Un appui sur la flèche de gauche diminue la valeur de q et un appui sur la flèche de droite l'augmente. Elle publiera cette valeur sur le topic *chatter_pos_vit* seulement si un nouveau couple T a été publié (c'est la fonction *Callback Cb_1* sur l'image). La fréquence de publication des données sur le topic *chatter_pos_vit* est tributaire de deux éléments :
 1. La fréquence du topic *chatter_torque* : le fréquence de *chatter_pos_vit* ne peut pas être plus élevée que celle de *chatter_torque* étant donné que les valeurs sont publiées sur ce topic seulement si une nouvelle valeur a été publiée sur *chatter_torque*.
 2. Les temps d'exécution à l'intérieur de la boucle : dans cet exemple, ceux-ci se limitent à l'appel des fonctions pour l'interaction avec le clavier à chaque itération de la boucle. Les temps d'exécutions de ces fonctions ralentissent la fréquence de publication sur le topic *chatter_pos_vit*.

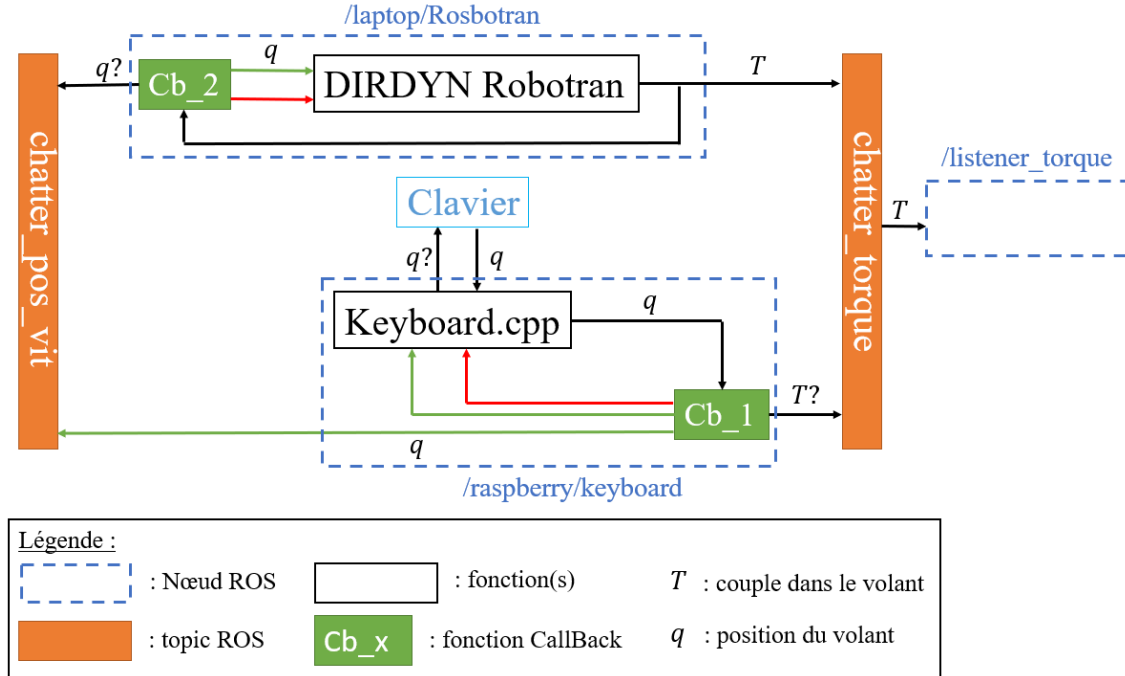


FIGURE 4.4 – Architecture du protocole de communication

Ces mesures de fréquences permettent de tirer deux conclusions. Tout d’abord, le fait d’avoir la boucle de Robotran tournant indépendamment des autres est un avantage important. En effet, la fréquence de la boucle respecte toujours le pas de temps fixé pour l’intégration temporelle⁴ et ce, peu importe si la fréquence des autres boucles augmente, diminue ou si les nœuds sont tout simplement stoppés. Il faut cependant prouver que les fonctions ROS qui publient et souscrivent à des topics à l’intérieur de la boucle de Robotran ne ralentissent pas cette dernière, ce qui sera fait à la section suivante en mesurant la fréquence du topic `chatter_torque`.

Enfin, la fréquence de rafraîchissement de la position du volant est égale à celle du topic `chatter_pos_vit`. Celle-ci permet de mesurer la fluidité et la rapidité de l’interaction entre l’utilisateur et le volant virtuel de la simulation Robotran. Il faut donc la maximiser afin d’obtenir la simulation la plus réaliste possible.

4.1.4 Résultats

Les résultats suivants sont principalement basés sur l’analyse des fréquences expliquées ci-dessus. En effet, il s’agit de prouver que l’intégration de fonctions ROS dans un projet Robotran n’a pas d’impact négatif sur ce dernier.

Simulation en temps-réel de Robotran

Tout d’abord, il a fallu déterminer le pas de temps minimal $\Delta_{t,sim}$ qui respectait une simulation en temps-réel du projet Robotran sans ROS. Le fait d’avoir le plus petit pas de temps possible se traduit par une intégration des positions et vitesses des articulations quasi-continue. Cela offre la plus haute fréquence de rafraîchissement de la position du volant et donc l’interaction la plus réaliste avec le modèle multicorps.

4. A condition que celui-ci soit assez grand pour permettre la résolution des équations en temps-réel.

Afin de garder une expérience comparable avec celle où ROS est intégré, le modèle Robotran est ici piloté à l'aide du clavier de l'ordinateur et des fonctions *real-time* intégrées dans Robotran sans utilisation de ROS.

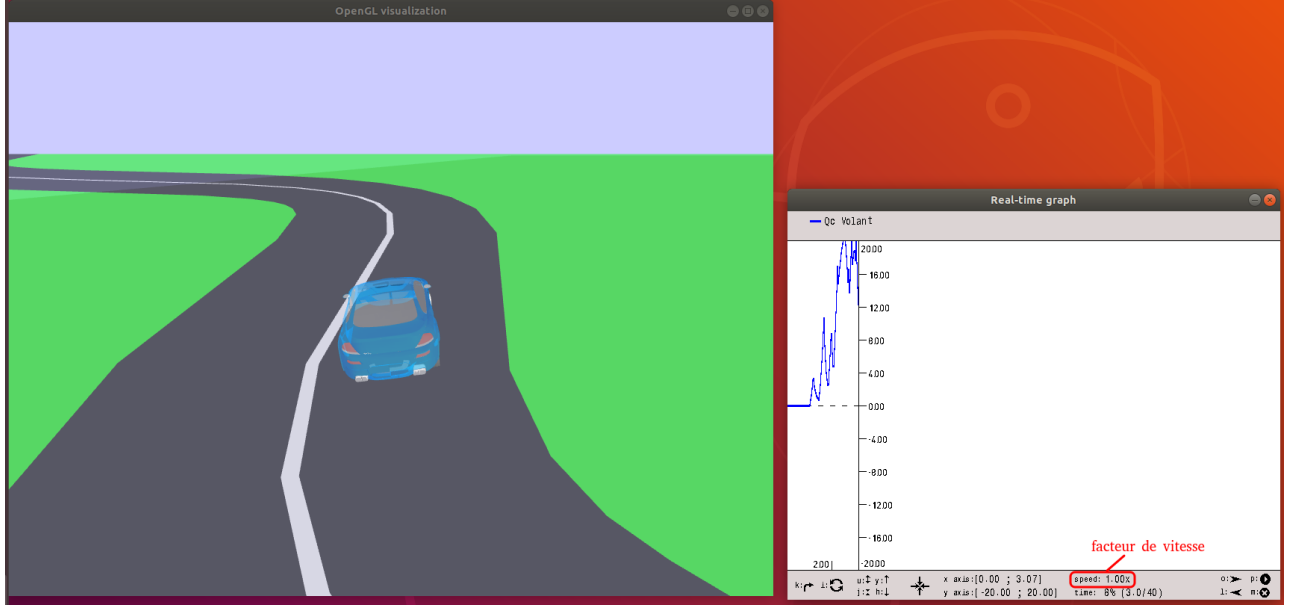


FIGURE 4.5 – Interface graphique d'une simulation temps-réel Robotran

Afin d'analyser le respect d'une simulation en temps réel, il faut introduire le facteur de vitesse réelle f_{real} défini comme suit :

$$f_{real} = \frac{\Delta_{t,sim}}{\Delta_{t,res} + \Delta_{t,wait}} \quad (4.1)$$

où :

- $\Delta_{t,res}$: temps de calcul pour la résolution des équations du mouvement (comprend le temps de la résolution des contraintes et de l'intégration des positions et vitesses des articulations). Ce temps de calcul est le plus petit possible que le permet le CPU de l'ordinateur.
- $\Delta_{t,wait}$: temps d'attente pour respecter le temps-réel ($f_{real} = 1$). C'est le temps pendant lequel le processus de Robotran se met en attente si la résolution des équations du mouvement s'est faite dans un temps inférieur au pas de temps imposé.

Lors d'une simulation en temps-réel, un interface graphique affiche (voir figure 4.5) la valeur de ce facteur de vitesse de la simulation⁵ dont la signification est reprise dans le tableau suivant. :

Valeur	Signification
100%	Respect d'une simulation en temps réel $\Delta_{t,res} = \Delta_{t,sim}$ et $\Delta_{t,wait} = 0$
<100%	Simulation trop lente, le pas de temps ne permet pas de résoudre les équations à temps $\Delta_{t,res} > \Delta_{t,sim}$ et $\Delta_{t,wait} = 0$
>100%	Simulation trop rapide, il est possible diminuer le pas de temps $\Delta_{t,res} < \Delta_{t,sim}$ et $\Delta_{t,wait} > 0$

TABLE 4.3 – Facteur de vitesse

5. Il est également possible d'enregistrer cette valeur dans un fichier afin de l'analyser après la simulation.

En commençant avec un grand pas de temps pour ce modèle (ici $\Delta_{t,sim} = 1.5ms$), et en diminuant celui-ci petit à petit, il est possible de déterminer expérimentalement (en réalisant des simulations assez dynamiques avec le véhicule) le pas de temps minimal qui garde un facteur de vitesse supérieur ou égal à 100%. La valeur trouvée et la fréquence correspondante sont :

$\Delta_{t,sim}$ [ms]	$Freq_{Rob}$ [Hz]
0.82	1219.51

TABLE 4.4 – Pas de temps minimal et fréquence maximale d’une boucle Robotran seule

Simulation en temps-réel de ROS-Robotran

Vérification des performances Il s’agit maintenant de vérifier si le projet complet ROS-Robotran avec le clavier du Raspberry permet de réaliser une simulation avec un pas de temps $dt = 0.82ms$. Cela va se faire en mesurant la fréquence du topic *chatter_torque* afin de vérifier si les fonctions ROS à l’intérieur du projet Robotran ne ralentissent pas celui-ci. De fait, un outil ROS permet de mesurer la fréquence de publication moyenne sur un topic. Comme la boucle de Robotran publie à chaque pas de temps sur le topic *chatter_torque*, il suffit alors de mesurer la fréquence sur celui-ci afin de déterminer celle de Robotran, l’objectif étant d’atteindre une fréquence de publication $F = 1219.51Hz$.

Afin de garder les mêmes conditions dynamiques de simulation, les mesures de fréquences ont été réalisées lorsque le véhicule était piloté de manière à suivre la ligne centrale de la section suivante du circuit réalisé par Nicola De Lellis et Quentin Materne dans le cadre de leur mémoire [2].

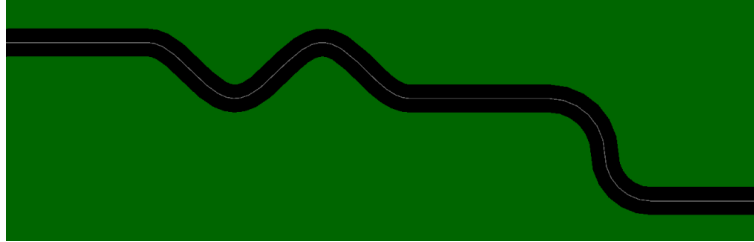


FIGURE 4.6 – Section de circuit pour les mesures de fréquence [2]

Un premier test à été réalisé avec 3 pas de temps différents pour Robotran : un proche de celui souhaité ($\Delta_{t,sim} = 0.8ms$), un plus faible ($\Delta_{t,sim} = 0.775$) (afin de mettre en exergue les raisons d’un non respect du temps-réel qui seront expliquées ci-dessous) et un plus élevé ($\Delta_{t,sim} = 0.9ms$). Ce résultat est exprimé en pourcentage de la fréquence moyenne de publication sur le topic ($Freq_{ROS}$) par rapport à la fréquence de Robotran nécessaire pour une simulation en temps-réel ($Freq_{Rob}$)⁶. Il faut faire attention au fait que ROS mesure bien la fréquence moyenne de publication sur le topic, c’est-à-dire le nombre de données publiées depuis le début sur ce topic divisé par le temps écoulé. Les premières valeurs du graphique sont donc peu représentatives de la fréquence de la boucle Robotran. En effet, les données ne sont pas publiées exactement au même moment lors de chaque itération. Il se peut donc que les deux premières publications soient très proches l’une de l’autre ou très éloignées et cela donne lieu à une première fréquence de publication plus haute ou plus basse que la moyenne. Le fait de prendre la moyenne permet de corriger cette erreur au fur et à mesure de la simulation.

6. Cela permet de comparer plusieurs pas de temps et donc fréquences différentes sur le même graphique.

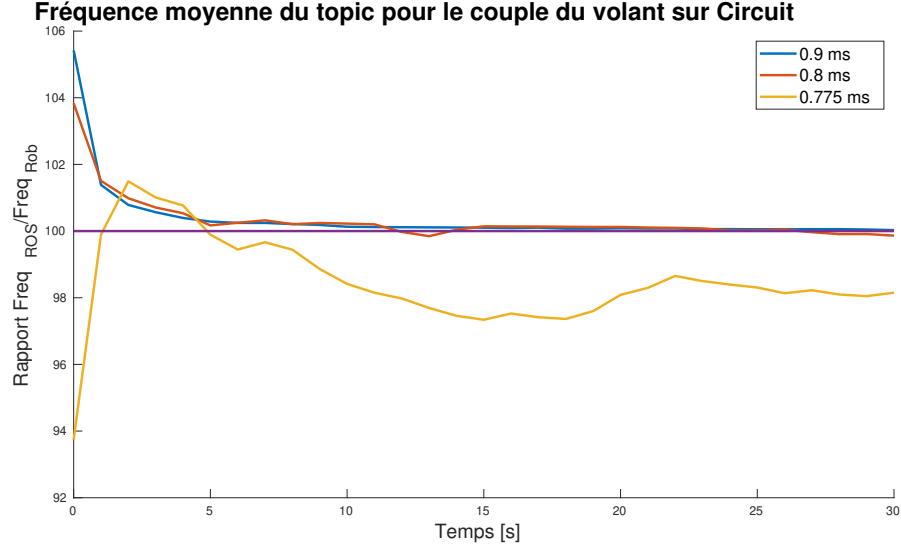


FIGURE 4.7 – Variation du pas de temps pour l'analyse de différentes fréquences

On observe qu'un pas de temps de $0.9ms$ est suffisant pour respecter une simulation temps-réel à 100%. Un pas de temps de $0.8ms$ montre quelques faibles baisses de fréquence en-dessous de celle demandée. Et enfin un pas de temps de $0.775ms$ montre une certaine insuffisance pour respecter le temps-réel.

Finalement, le même pas de temps utilisé pour l'exécution de Robotran sans ROS ($\Delta_{t,sim} = 0.82ms$) permet de garder une fréquence supérieure à $1219.51Hz$ pour la quasi totalité de la simulation. De fait, la valeur moyenne finale va s'établir légèrement en dessous de $1219.51Hz$ (avec des valeurs restant supérieures à $1218.5Hz$) mais permet tout de même de garder un facteur de vitesse réelle supérieur ou égal à 100%.

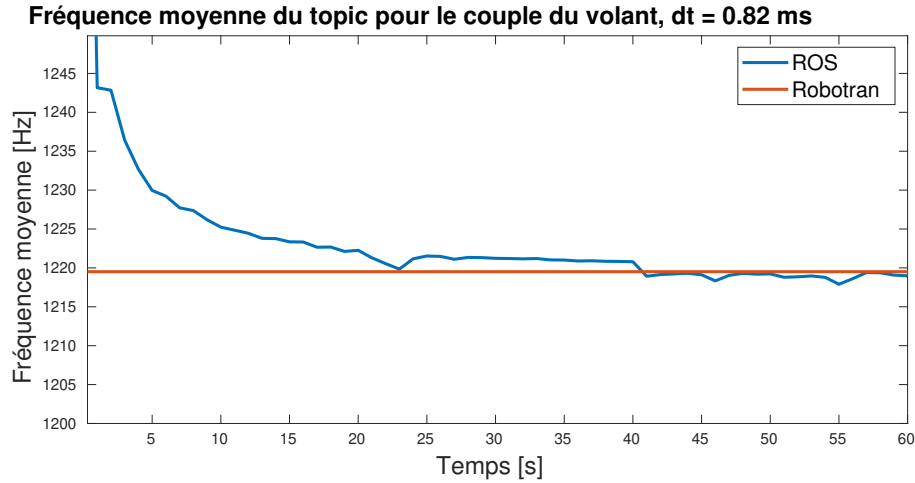


FIGURE 4.8 – Pas de temps optimal pour la simulation ROS-Robotran

Ce résultat permet d'affirmer que les fonctions ROS (qui publient et soucrivent à des topics) n'ont pas d'influence sur la vitesse d'exécution de Robotran.

Analyse des baisses de fréquence Tout d'abord, un premier test a permis de déterminer l'origine de ces baisses de fréquence, à savoir la dynamique de la simulation. En effet, pour une simulation en ligne droite, il est possible d'utiliser un pas de temps $\Delta_{t,sim} = 0.75ms$ alors que lorsqu'on suit la trajectoire du circuit celui-ci est clairement insuffisant comme le montre le graphe suivant :

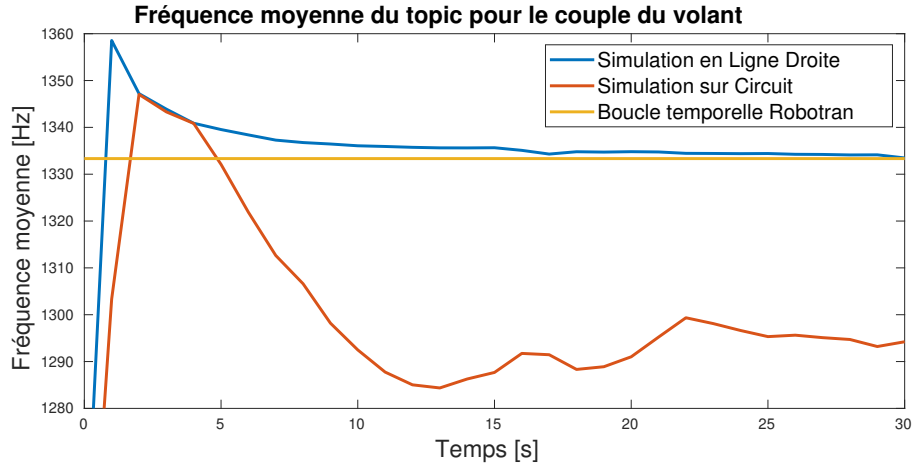


FIGURE 4.9 – Comparaison d'une simulation en ligne droite ou sur circuit pour $\Delta_{t,sim}=0.75ms$

On observe tout de même que la fréquence pour la simulation sur circuit présente la même moyenne que celle en ligne droite avant de chuter brusquement. Cette chute correspond à la prise du premier virage. De ce fait, comme le montre le graphe suivant, la fréquence diminue lors des changements de direction mais réaugmente lors des lignes droites et des tournants constants.

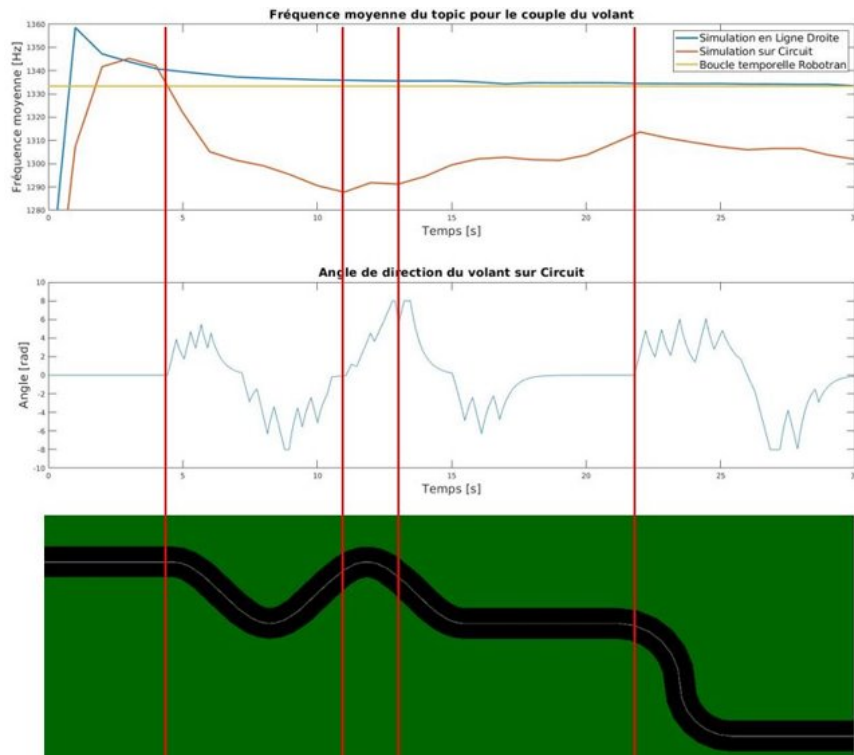


FIGURE 4.10 – Lien entre la baisse de fréquence et la dynamique du circuit pour $dt=0.75ms$

Les changements de direction sont les moments où les suspensions du véhicule sont le plus sollicitées du au transfert de masse du véhicule. Dans le modèle multicorps, ces suspensions à bras superposés sont modélisées par une boucle fermée de plusieurs corps qui implique une résolution de ses contraintes. Pour rappel, ces contraintes généralement non-linéaires sont résolues à chaque pas de temps par un algorithme itératif de Newton-Raphson avec comme conditions initiales les positions des articulations indépendantes. Lors de grands mouvements des suspensions, le nombre d'itérations augmente afin de trouver les nouvelles positions qui sont donc plus éloignées de celles initiales.

Le graphe suivant démontre un lien entre la baisse de fréquence et le nombre d'itérations⁷ nécessaires afin de résoudre les contraintes. On y observe que c'est la 4ème itération nécessaire qui cause cette diminution de la fréquence⁸. Cette 4ème itération fait augmenter le temps de résolution des équations du mouvement $\Delta_{t,res}$ de sorte que le facteur de vitesse réelle devient inférieur à l'unité.

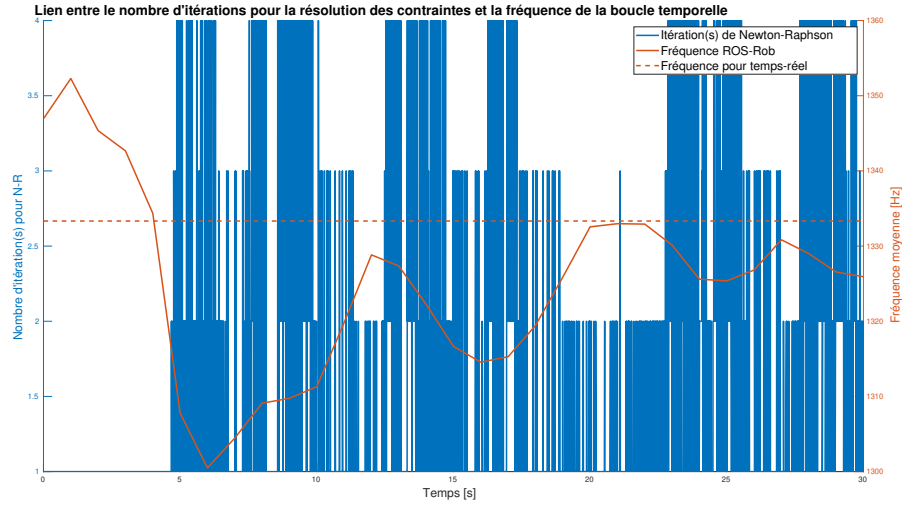


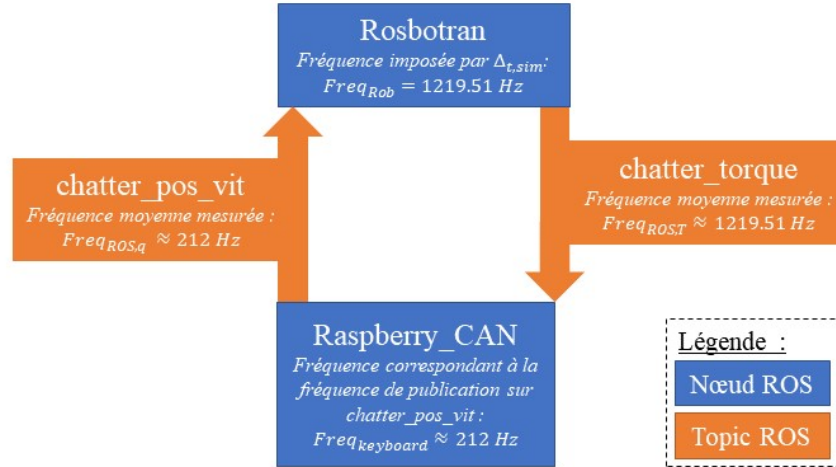
FIGURE 4.11 – Lien entre la baisse de fréquence et la résolution des contraintes pour $\Delta_{t,sim}=0.75ms$

Mesure de l'interaction de l'utilisateur avec le modèle Robotran

Comme déjà expliqué à la section 4.1.3, le niveau d'interaction avec le modèle Robotran se mesure avec la fréquence du topic *chatter_pos_vit* qui permet de mettre à jour la position du volant. Pour un pas de temps $\Delta_{t,sim} = 0.82ms$, Robotran prend la nouvelle position du volant sur le topic à une fréquence $Freq_{Rob} = 1219.51Hz$. Cependant, la fréquence de publication des données sur le topic *chatter_pos_vit* a été mesurée sur plusieurs simulations et donne en moyenne $Freq_{ROS,q} \approx 1209Hz$ car la boucle de *Keyboard* est légèrement ralentie par les fonctions pour l'interaction avec le clavier.

7. L'intégrateur temporel utilisé pour intégrer les équations du mouvement est celui de Runge-Kutta 4. Ce schéma nécessite de résoudre 4 fois les contraintes par pas de temps. Le nombre d'itérations présentée ici est celui de la 4ème résolution.

8. L'outil ROS qui mesure la fréquence sur les topics ne le fait que toute les une seconde et pour rappel il mesure la fréquence moyenne. Ce manque de précision explique pourquoi la fréquence peut parfois recommencer à chuter alors que le nombre d'itération n'a pas encore recommencé à augmenter.

FIGURE 4.12 – Schéma des différentes fréquences mesurées pour $\Delta_{t,sim}=0.82\text{ms}$

On peut alors mesurer le taux de rafraîchissement de la position du volant :

$$f_{update,q} = \frac{Freq_{ROS,q}}{Freq_{Rob}} * 100 = 99.14\% \quad (4.2)$$

Cela signifie que 99.14% des valeurs de la position du volant utilisées par Robotran auront été mises à jour par l'utilisateur à l'aide du clavier à une fréquence de 1209 Hz. Cela témoigne d'une bonne interaction entre un actionneur physique connecté à un appareil extérieur et une simulation Robotran, le tout à l'aide de ROS.

4.2 Commande d'un modèle Robotran à l'aide de capteurs extérieurs et simulation anticipée

La deuxième application a été établie afin de prouver la polyvalence d'un projet ROS-Robotran et les avantages qu'on peut en tirer. En effet, toujours en gardant la même structure de la *main* de Robotran, il est maintenant possible de piloter le même modèle multicorps qu'à la section précédente 4.1 à l'aide de sonars. Toutefois, certaines modifications ont été apportées aux fonctions d'utilisateur de Robotran car l'objectif au niveau de la conduite diffère de celui du pilotage à l'aide du clavier. Le but est de créer ici un système d'aide à la conduite utilisant les avantages de Robotran. Lorsque les simulations sur Robotran ne sont pas réalisées en temps-réel avec un affichage de la simulation, il est possible de résoudre les équations du mouvement en un temps très court (le temps de calcul peut être inférieur à la seconde pour une simulation de 5 secondes). Comme déjà présenté lors de l'introduction, cela pourrait être un avantage crucial, surtout lors de manoeuvre d'évitement d'urgence, de pouvoir prédire la trajectoire d'un véhicule à l'avance en se basant sur les résultats produits par Robotran. C'est cet aspect qui a été mis en avant dans cette application.

Le contexte est le suivant (voir figure 4.13), un véhicule roule à une certaine vitesse sur une large route, sa bande de roulement ainsi que les bandes de roulement sur sa droite sont sèches tandis que les bandes de roulement sur sa gauche présentent une certaine humidité et donc une moins bonne adhérence. Soudainement, un obstacle est détecté sur la route à l'aide de sonars placés à l'avant du véhicule. L'obstacle étant plus large du côté droit que du côté gauche, quelle manoeuvre offre la meilleure trajectoire d'évitement : une faible décalage latéral à réaliser sur un sol mouillé ou un plus large décalage sur sol sec ? Le temps moyen de réaction d'un humain face à un obstacle inattendu est de une seconde. De plus, celui-ci doit se baser sur son expérience de conduite pour décider au plus vite de sa manoeuvre d'évitement. C'est ici que ROS, pouvant gérer l'interaction avec les sonars, et que Robotran, pouvant réaliser différents scénarios d'évitement en moins d'une seconde, vont intervenir.

Le protocole d'évitement consiste donc à détecter l'obstacle et en fonction de la distance de celui-ci, produire les résultats suivants :

- Si l'obstacle ne présente pas encore un véritable danger : afficher un message d'alerte .
- Si l'obstacle est proche de la distance de freinage minimale nécessaire pour arrêter le véhicule : réaliser un freinage en ligne droite .
- Si l'obstacle est plus proche que la distance de freinage : simuler à l'aide de Robotran deux scénarios d'esquive (sur la gauche et sur le droite) qui déportent le véhicule sur le côté en freinant. Ensuite choisir le scénario qui s'éloigne le plus de l'obstacle et l'appliquer en temps réel au véhicule.

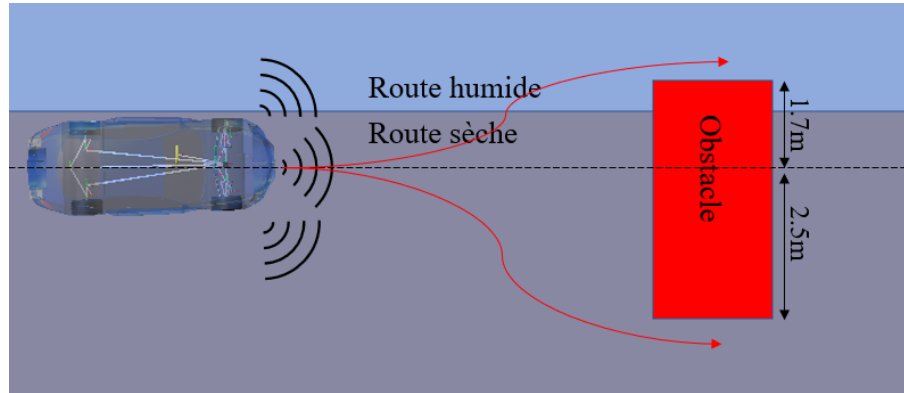


FIGURE 4.13 – Mise en contexte d'un scénario d'évitement

La suite du protocole d'évitement présenté dans cette section a été simplifié sur plusieurs points qui seront explicités à chaque fois. En effet, un protocole d'évitement sécurisé et performant nécessite de prendre en compte de nombreux paramètres ainsi que de nombreuses informations sur l'environnement dans lequel évolue le véhicule (si les trajectoires d'évitement ne présentent pas d'autres obstacles, la présence d'usagers faibles, la prise en compte des autres véhicules, etc). Ces raisons font que l'implémentation d'une méthode d'esquive efficace dépasse la portée de ce travail. La méthode utilisée ici, bien que simplifiée, reste toutefois réaliste afin de permettre la validation de cette application ROS-Robotran.

Enfin, le principe permettant de piloter le modèle à l'aide du clavier du Raspberry (voir section 4.1) reste d'application afin de permettre une simulation interactive et de montrer la possibilité de coupler plusieurs applications ROS-Robotran ensemble.

4.2.1 Architecture de l'hardware

L'architecture de l'hardware de base reste la même qu'à la section précédente mise à part le branchement de capteurs ultrason HC-SR04 sur le Raspberry.

Capteur ultrason HC-SR04

Ces capteurs se basent sur le principe des sonars pour mesurer la distance d'un objet (voir figure 4.15). Lorsque l'émetteur du sonar reçoit un signal d'entrée (appelé *Echo*) provenant du Raspberry, il envoie un signal ultrason dans la direction de l'objet. Le signal se réfléchit sur l'objet et est envoyé au capteur du sonar. Lorsque ce dernier réceptionne le signal sonore, il envoie un signal de sortie (*Trigger*) au Raspberry. Un code Python implémenté dans un noeud ROS va mesurer le temps nécessaire au signal ultrason pour faire l'aller-retour avec l'objet et le multiplier par la vitesse du son pour obtenir la distance⁹. Le sonar possède une portée allant de 2 à 400 cm ainsi qu'une précision de ± 0.5 cm. Les mesures sont donc multipliées par un facteur d'échelle afin de correspondre à l'échelle du véhicule Robotran.

9. En notant que le temps devra être divisé par deux car celui-ci prend en compte la distance du capteur à l'objet (aller) et de l'objet au capteur (retour).

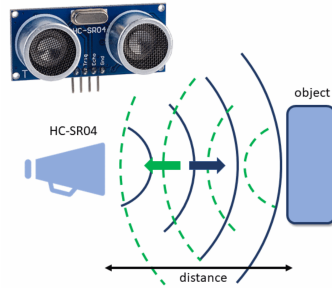


FIGURE 4.14 – Capteur ultrason HC-SR04 [24]

Au niveau de la mise en place des sonars, un câble "T-cobbler" permet de disposer des broches GPIO du Raspberry sur une breadboard pour la réalisation de circuit électronique. En effet, le sonar HC-SR04 nécessite 4 branchements :

- Une alimentation de 5V fournie par une des broches du Raspberry.
- Un branchement pour le signal d'entrée *Echo* d'une tension de 3.3V. La tension de base de ce signal est normalement de 5V. Cependant, afin de ne pas endommager les broches GPIO du Raspberry (fournie pour 3.3V) un diviseur de tension composé de deux résistances de 1kOhm et 2kOhm a été utilisé pour réduire la tension du signal.
- Un branchement pour le signal de sortie *Trigger*.
- Une branchement de la borne négative du sonar à une borne négative du Raspberry.[24]

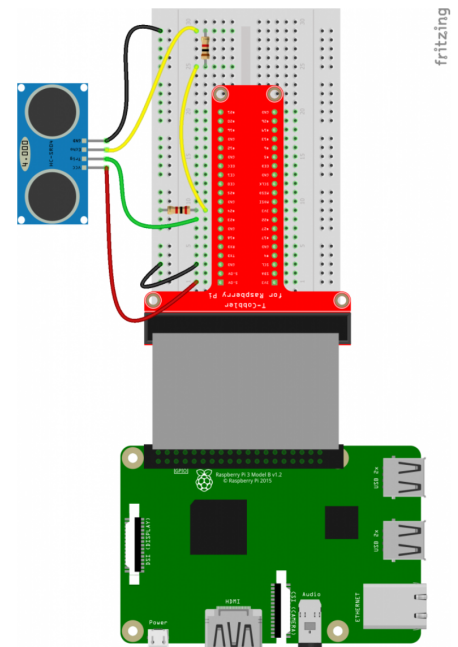


FIGURE 4.15 – Circuit électronique pour le branchement du sonar au Raspberry [24]

Le même branchement a été réalisé pour deux autres sonars qui peuvent servir à la détection d'obstacle sur les côtés du véhicules¹⁰.

Les 3 capteurs HC-SR04 ainsi que les différents composants nécessaires (un câble GPIO 40 broches, un "T-cobbler", les résistances et la breadboard) ont été achetés en kit auprès de la société *Freva* [25].

¹⁰. Ils ne sont pas utilisés dans cet exemple d'application.

Architecture physique globale

Finalement, le schéma suivant reprend les différents composants hardware, les connexions physiques entre elles ainsi que leur alimentation nécessaire :

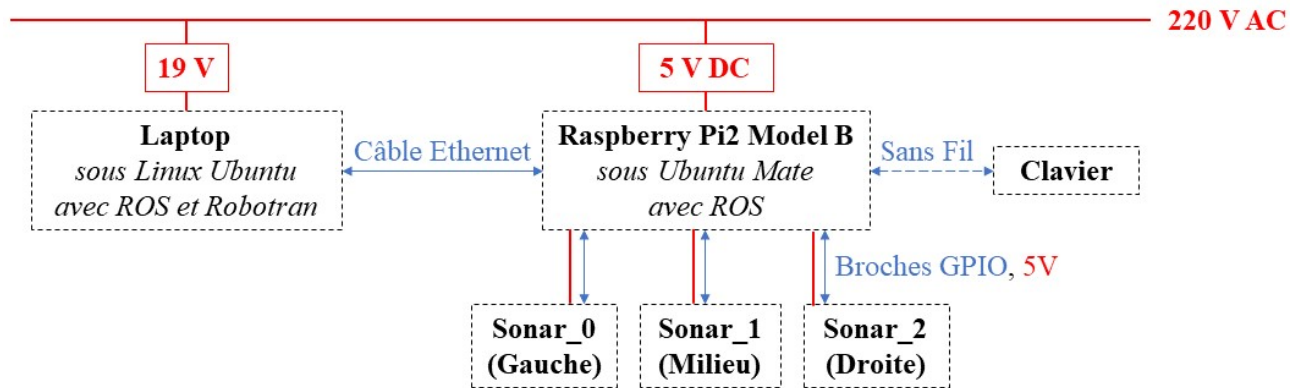


FIGURE 4.16 – Architecture physique globale

4.2.2 Architecture du software

L'architecture de l'environnement ROS présente une complexité supérieure pour cette application. En effet, le nombre de noeuds et de topics augmente fortement, en voici la composition :

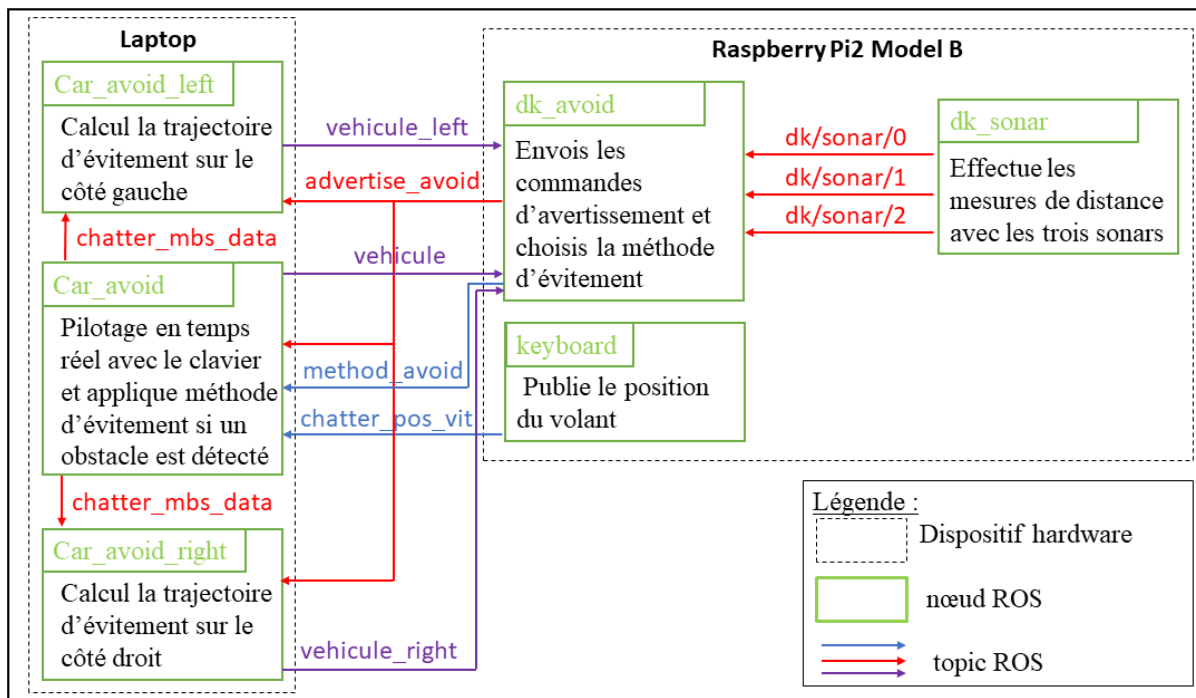


FIGURE 4.17 – Architecture globale de ROS

Les tableaux suivant résument la fonctionnalité de chaque noeud ainsi que les informations transportées sur les topics :

Hardware	Noeud(s)	Fonction
Ordinateur	<i>Car_avoid</i>	Ce noeud réalise une simulation en temps-réel d'une voiture sous Robotran. Cette voiture est pilotée par un clavier comme à l'exemple de la section 4.1 et effectuera une manoeuvre d'évitement lorsque les sonars détecteront un obstacle.
	<i>Car_avoid_left</i> et <i>Car_avoid_right</i>	Ces noeuds réalisent une simulation sous Robotran du même véhicule mais cette fois-ci pas en temps-réel. Ils débutent la simulation lorsqu'un obstacle est détecté et effectuent une manoeuvre d'évitement à gauche et à droite. La configuration dynamique initiale du véhicule correspond à celle du véhicule de <i>Car_avoid</i> lors de la détection de l'obstacle. En effet, ce dernier va envoyer la position et vitesse des articulations de son multicorps via le topic <i>chatter_mbs_data</i> pour faire démarrer les simulations de <i>Car_avoid_left</i> et <i>Car_avoid_right</i> dans la même position et vitesse.
Raspberry	<i>keyboard</i>	Ce noeud permet de piloter le modèle Robotran en temps-réel via les touches directionnelles du clavier.
	<i>dk_sonar</i> (codé en <i>Python</i>)	Ce noeud mesure les distances entre les objets et les trois sonars.
	<i>dk_avoid</i> (codé en <i>Python</i>)	Ce noeud gère l'interaction entre les sonars et les modèles Robotran. Il enverra des avertissements composés de la distance de l'obstacle, du type de freinage et du type de déviation à faire aux noeuds ROS-Robotran. C'est également lui qui choisira la méthode d'évitement la plus efficace en fonction des données reçues par les noeuds ROS-Robotran.

TABLE 4.5 – Fonctionnalité des noeuds ROS pour l'application avec les sonars

Les noeuds réalisant des simulations Robotran utilisent la même architecture multithread que présentée à la section 3.3.2 afin de pouvoir publier et recevoir des données via des topics. Cependant, les threads n'ont pas été représentés sur le schéma ci-dessus comme à la figure 4.3 par soucis de clarté.

Les noeuds *dk_avoid* et *dk_sonar* ont été adaptés à partir d'un projet/tutoriel ROS réalisé par Tiziano Fiorenzani pour la conception d'une *Donkey Car*¹¹ autonome [27]. Bien que la méthode d'esquive soit toute à fait différente, ces noeuds ont été utilisés pour la gestion des données reçues par les sonars et pour montrer le caractère opensource de ROS qui permet d'adapter facilement à son projet des noeuds ROS déjà conçus.

Cette application utilise des noeuds codés en C++ et en Python. ROS va alors gérer nativement

11. Il s'agit d'une plateforme opensource pour réaliser une voiture téléguidée utilisant un Raspberry et une implémentation en code Python [26].

la normalisation des données publiées sur les topics afin qu'elles puissent être utilisées par les deux langages. Les topics ainsi que les informations qu'ils transportent sont repris ci-dessous.

Topic(s)	Informations véhiculées
<i>dk/sonar/i</i>	Mesure de la distance de l'obstacle
<i>advertise_avoid</i>	Distance de l'obstacle (lorsque la valeur est publiée, les trois topics utilisant Robotran sont alerté de la présence d'un obstacle et la fonction <i>Callback</i> correspondante permet d'activer le protocole d'évitement).
<i>vehicule,vehicule_left et vehicule_right</i>	- Vitesse du véhicule - Déplacement latéral - Déplacement longitudinal
<i>chatter_pos_vit</i>	Position du volant
<i>method_avoid</i>	- Distance de l'obstacle - Choix du freinage - Choix de la direction d'évitement
<i>chatter_mbs_data</i>	Position et vitesse des articulations du multi-corps Robotran

TABLE 4.6 – Fonctionnalité des topics ROS pour l'application avec les sonars

4.2.3 Architecture de contrôle

Le schéma suivant présente les différentes étapes de calculs et de communication qu'effectuent les noeuds ROS (indiqués en bleu) lorsqu'un obstacle est détecté :

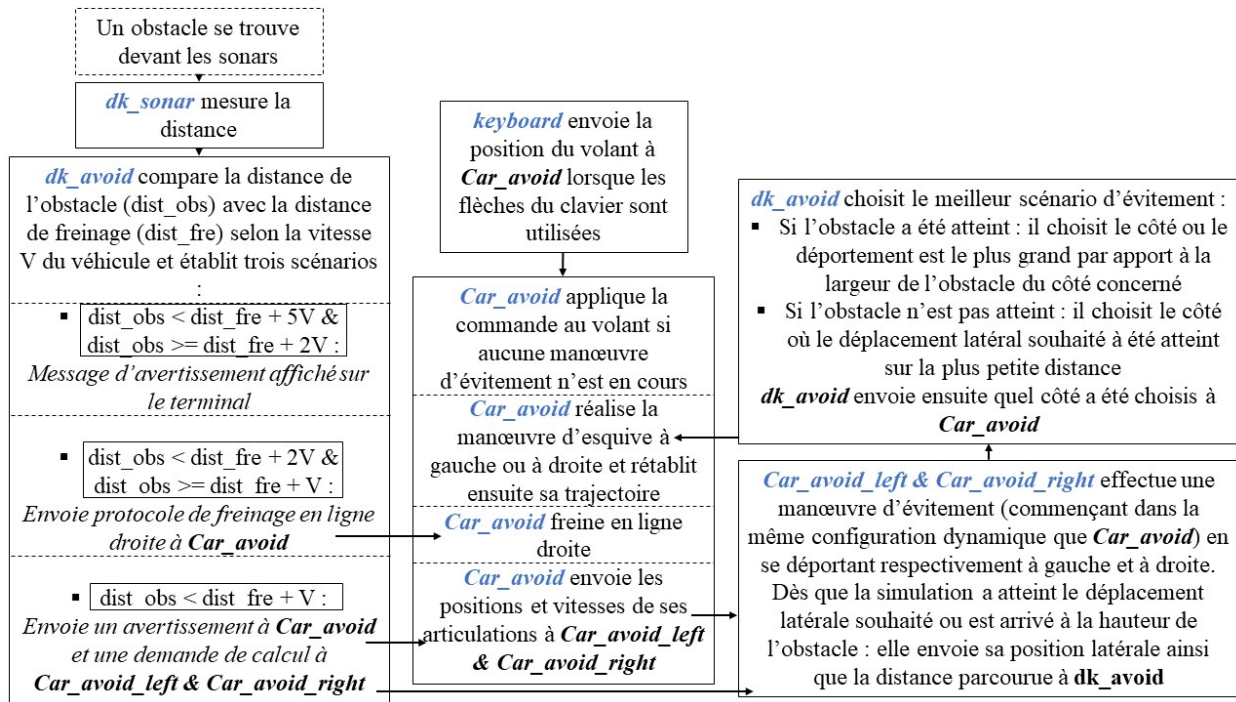


FIGURE 4.18 – Protocole de communication pour un évitement

Le protocole de communication se base sur le même principe que celui de la section 4.1.3 avec la publication d'informations sur des topics et l'utilisation de fonctions *Callback* qui exécutent leurs actions lorsque des données sont publiées.

La boucle Robotran de *Car_avoid*, qui réalise la simulation en temps-réel, utilise un pas de temps $dt = 1.2ms$. Celui-ci reste assez petit que pour avoir une interaction fluide avec le véhicule et assez grand pour permettre la publication et réception de données sans diminuer le facteur de temps réel. En effet, la publication de données sur le topic pour faire démarrer les deux autres simulations de Robotran dans la même configuration dynamique est constituée de 100 valeurs (la position et la vitesse des 50 articulations du véhicule) qui doivent être placées dans un vecteur. De plus, les processeurs de l'ordinateur sont plus largement sollicités à cause des deux autres simulations Robotran qui doivent tourner en parallèle et donc diminuer la vitesse de calcul. C'est donc ces raisons qui justifient ce plus grand pas de temps. Cette boucle publie également la vitesse du véhicule en temps-réel à *dk_avoid* qui l'utilise pour déterminer la distance de freinage.

Les simulations Robotran de *Car_avoid_right* et *Car_avoid_left* utilisent le plus grand pas temps permettant la résolution des équations du mouvement qui est dans ce cas-ci $dt = 3ms$. Le pas de temps ne peut pas être augmenté indéfiniment car il faut garantir la fermeture des boucles pour les contraintes et la stabilité de l'intégrateur. L'objectif de ces noeuds est d'effectuer les manoeuvres d'évitement à gauche et à droite le plus rapidement possible pour communiquer les déplacements obtenus et choisir la véritable manoeuvre d'évitement. Un grand pas de temps permet d'atteindre la position finale avec un plus faible nombre d'itérations et donc un temps de calcul plus rapide.

La boucle *keyboard* qui publie la position du volant pour un contrôle de celui-ci via les touches du clavier tourne à une fréquence fixe (contrairement à l'exemple précédent où elle était tributaire de la fréquence de Robotran). En choisissant une fréquence plus élevée que Robotran, on garantit de toujours avoir une information sur la position du volant disponible pour Robotran. De plus, ne pas avoir de boucle de contrôle simplifie l'architecture et le protocole de communication.

La fréquence de la boucle *dk_sonar* qui mesure à chaque fois la distance de l'obstacle est de $20Hz$ et celle de *dk_avoid* qui analyse les scénarios d'évitement est de $10Hz$. La boucle *dk_sonar* tourne à une fréquence plus élevée afin d'assurer la disponibilité d'une information sur la distance de l'obstacle pour *dk_avoid* à tout moment. La boucle *dk_avoid*, quand à elle, utilise une fréquence assez faible car utiliser une fréquence plus élevée n'aura pas d'impact significatif sur le temps de réaction de la méthode d'esquive. En effet, comme cela sera présenté à la section des résultats 4.2.5, ce sont les simulations d'esquive Robotran appelées dans cette boucle qui sont les plus chronophages et ralentissent cette boucle.

Finalement, les choix de la valeur de la distance de l'obstacle entraînant tel ou tel scénarios ont été fait de la manière suivante :

$Distance_{obstacle} < Distance_{freinage} + 5V$ & $Distance_{obstacle} \geq Distance_{freinage} + 2V$	Le $+xV$ avec V en $[m/s]$ implique que la distance de freinage est augmentée d'une distance correspondante au véhicule roulant à cette vitesse V pendant x secondes. Cela signifie que le conducteur aura entre 2 et 5 secondes pour réagir face à l'obstacle arrivant sur sa trajectoire afin de ralentir ou de changer de trajectoire.
$Distance_{obstacle} < Distance_{freinage} + 2V$ & $Distance_{obstacle} \geq Distance_{freinage} + V$	Si l'obstacle surgit brusquement ou que le conducteur n'a toujours pas modifié sa trajectoire le véhicule va alors automatiquement freiner en ligne droite. Le facteur de sécurité pour la distance de freinage et le fait de commencer la manoeuvre entre 1 et 2 secondes plus tôt permettent de prendre en compte le temps d'envoi de l'information pour faire freiner le véhicule et de garder une certaine marge de sécurité avec l'obstacle.
$Distance_{obstacle} < Distance_{freinage} + V$	Dans ce cas-ci, l'obstacle devient trop proche et le véhicule est obligé de l'éviter sur un des côtés.

4.2.4 Dynamique de l'évitement

Afin d'implémenter une méthode d'évitement qui, pour rappel, a été simplifiée par apport à une véritable méthode d'esquive de véhicule autonome, il faut mesurer et/ou calculer plusieurs grandeurs. En effet, la méthode d'évitement se base sur la distance de freinage nécessaire calculée en temps-réel ainsi que sur l'angle de braquage du véhicule qui est fonction de sa vitesse. Les hypothèses sur le sol glissant pour l'esquive de gauche ainsi que la fin de la méthode d'esquive sont également présentés.

Distance de freinage

Afin de connaître la distance de freinage nécessaire pour que le véhicule s'arrête, il faut d'abord mesurer la décélération maximale qu'il peut effectuer.

Cette décélération dépend tout d'abord du type de frein ainsi que de leur qualité. Cependant, ceux-ci ne sont pas implémentés dans le modèle Robotran utilisé. Leur effet est alors modélisé par un couple négatif appliqué aux articulations des roues. On suppose des freins parfaits capables de produire n'importe quelle valeur de couple de freinage et ce, en toute situation.

La décélération maximale dépend également du contact pneu-sol. En effet, la décélération d'un véhicule en ligne droite est due à la force longitudinale qui s'applique entre le sol et la roue. Cette dernière est donc négative pour une décélération.

Cependant, l'adhérence d'un pneu n'est pas parfaite et cette force n'est donc pas constante. On introduit ici le phénomène de glissement qui se définit comme étant la différence entre la vitesse linéaire de la roue et la vitesse au point de contact pneu-sol. Le graphique suivant présente cette force longitudinale en fonction du glissement longitudinal (composante longitudinale de la vitesse du point de contact, 0% correspond à un roulement sans glissement et 100% correspond au moment où la roue glisse complètement sans rouler) :

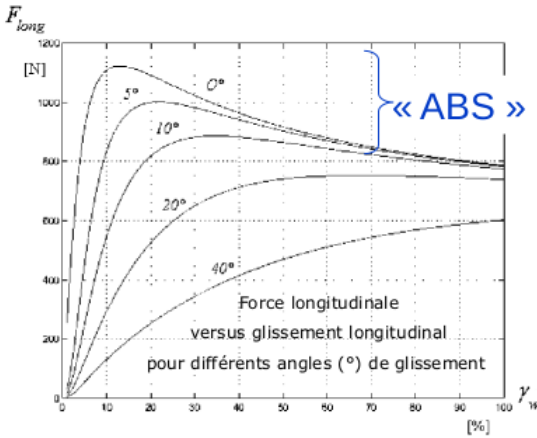


FIGURE 4.19 – Force longitudinale en fonction du glissement longitudinal [5]

Cette force présente d'abord une loi quasi linéaire pour de faible glissement. Elle atteint ensuite un maximum, qui offre donc une décélération maximale. C'est cette zone là qui est exploitée par l'ABS pour offrir un freinage optimal. Enfin, pour un glissement élevé on atteint la saturation et la force longitudinale diminue. Cela se traduit par une décélération plus faible et la zone d'adhésion du pneu qui diminue pour laisser place à la zone de glissement, le pneu commence à déraiper.

La voiture utilisée dans la simulation ne possédant pas d'ABS, il faut trouver le couple de freinage permettant de maximiser la force longitudinale tout en restant dans le régime linéaire afin d'éviter la saturation.

Introduisons d'abord la notion de couple de freinage total. Ce dernier consiste en la somme des couples appliqués à chaque roue. Cependant le couple appliqué aux roues avant et arrières est différent. De fait, lors du freinage il y a un transfert de masse vers l'avant du véhicule. La force verticale des roues avant sur le sol est plus élevée que celle des roues arrières et donc leur adhérence aussi. Il est donc possible d'atteindre une force longitudinale plus élevée à l'avant avant d'atteindre un régime de glissement total.

Ainsi, le couple de freinage appliqué au deux roues avant peut être plus élevé et dans le cas de ce véhicule, il a été choisi à 65 % du couple total et donc 35 % pour les deux roues arrières (la valeur doit à chaque fois être divisée en deux afin d'obtenir le couple de freinage pour chaque roue). Ces valeurs permettent d'atteindre le régime de glissement des pneus avant et arrières approximativement au même moment donc garantir une maximisation de la force longitudinale totale.

Une simulation a été effectuée sur sol sec en lançant le véhicule à haute vitesse et en augmentant petit à petit le couple de freinage total (en valeur absolue). La force longitudinale à chaque roue a été mesurée :

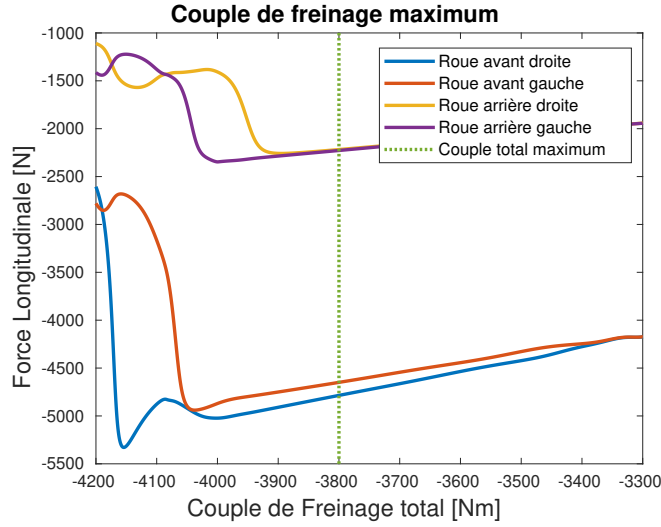


FIGURE 4.20 – Couple de freinage maximum

On observe que la force longitudinale atteint un maximum avant de décroître (en valeur absolue). De plus, cette proportion de couple avant/arrière permet d'atteindre une force longitudinale deux fois plus élevée pour les roues avant et donc un meilleur freinage. Enfin, c'est la roue arrière droite qui atteint en première le régime de saturation. Cela traduit sans doute une légère asymétrie du véhicule. La valeur du couple de freinage maximum a alors été prise lorsque la force longitudinale commence à dévier de la loi linéaire et nous donne $-3800 Nm$. Ceci donne donc $-1235 Nm$ pour chaque roue avant et $-665 Nm$ pour chaque roue arrière.

En connaissant ceci, il est possible de réaliser un calcul préliminaire afin d'estimer la décélération que ce couple de freinage produit. Tout d'abord, on peut estimer la force de freinage totale en divisant le couple de freinage total par le rayon des roues ($\approx 0.28m$) :

$$F_{freinage} = \frac{Q_{freinage}}{R_{roue}} = -13571.42N \quad (4.3)$$

En divisant la force de freinage par la masse du véhicule ($= 1649kg$), on obtient la décélération :

$$Dec = \frac{F_{freinage}}{M_{vehicule}} = -8.23m/s^2 \quad (4.4)$$

Finalement, la valeur de la décélération mesurée avec une simulation sur Robotran donne $Dec = -8.431m/s^2$, ce qui valide l'estimation. Les valeurs de décélération utilisés habituellement dans les mesures de distance de freinage sont comprises entre $-11m/s^2$ et $-8.5m/s^2$. Bien que la valeur de notre véhicule soit légèrement inférieure, elle respecte toutefois la norme imposée par l'arrêté royal sur les conditions techniques auxquelles doivent répondre les véhicules automobiles qui stipule :

"L'efficacité du dispositif de freinage de service des véhicules automobiles doit être telle que, sur une route quasi horizontale et sèche, la décélération moyenne de freinage en régime obtenue, les freins étant à froid et le moteur débrayé, ne soit jamais inférieure, quelles que soient les conditions de charge ou de vitesse, à :

a) $5 m/sec^2$, lorsqu'il s'agit d'autobus et d'autocars ;

b) $5,8 m/sec^2$, lorsqu'il s'agit de voitures, voitures mixtes et minibus ;" [28]

En connaissant maintenant la décélération maximum que le véhicule peut réaliser mesurée par Robotran, on peut directement déterminer la distance de freinage à l'aide des équations du mouvement :

$$Distance_{freinage} = F_{secu} \frac{-V^2}{2 * Dec} \quad (4.5)$$

avec V la vitesse du véhicule en m/s et F_{secu} un facteur de sécurité égal à 1.1. Le graphique à droite présente la distance de freinage sur sol sec nécessaire pour l'arrêt complet du véhicule en fonction de sa vitesse. La distance de freinage sur sol mouillé n'a pas été calculée car on supposera que le véhicule freine toujours sur une bande de roulement sec.

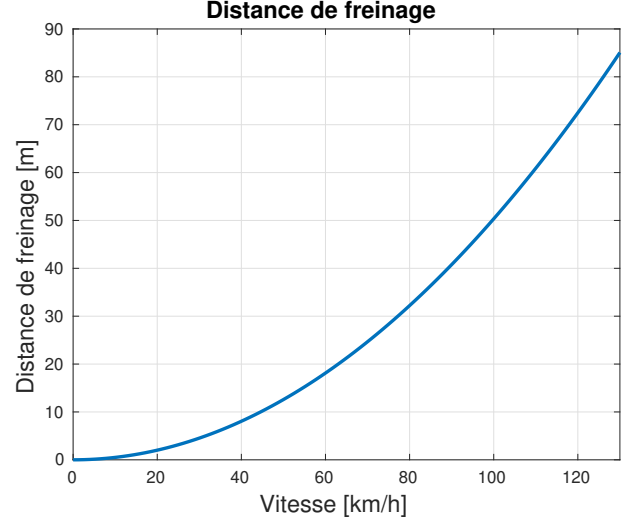


FIGURE 4.21 – Distance de freinage sur sol sec

Angle de braquage du volant

Pour connaître l'angle de braquage du volant à effectuer pour la manoeuvre, une équation a été implémentée. Celle-ci est fonction de la vitesse du véhicule. La vitesse du véhicule diminue cette valeur afin d'éviter un dérapage du véhicule avec un tournant trop serré (cela correspondrait au cas où les forces latérales du contact pneu-sol arriveraient à saturation).

L'équation établie pour l'angle de braquage en radian est la suivante :

$$Angle_{braquage} = \begin{cases} 8 & \text{si } V \in [0 \ 50[\text{ km/h} \\ (4.69 * 10^{-4})V^2 - 0.1656V + 15.1094 & \text{si } V \in [50 \ 130[\text{ km/h} \\ 1.5 & \text{si } V \in [130 \ \infty[\text{ km/h} \end{cases}$$

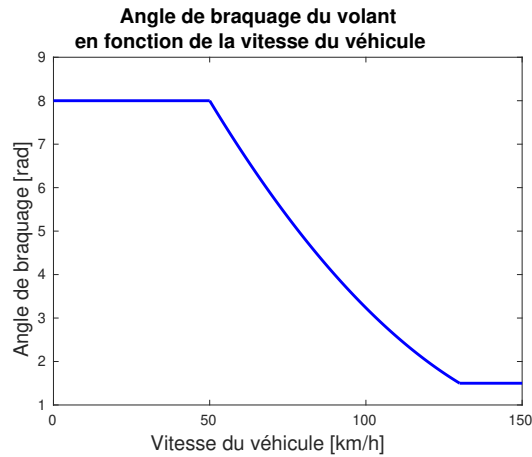


FIGURE 4.22 – Angle de braquage du volant en fonction de la vitesse du véhicule

Cette valeur correspond à un tournant sur la droite, la même valeur en négatif sera utilisée pour un tournant sur la gauche.

Simulation sur sol glissant

Comme déjà expliqué, la simulation de l'esquive à gauche se fait sur un sol glissant avec une multiplication des forces et moments de force du contact pneu-sol par 0.75 (voir section 1.3.1 pour plus d'explications). Afin de simplifier l'implémentation de la méthode, l'esquive à gauche sera supposée sur sol glissant tout le long de la simulation (et non pas juste sur la bande de gauche). Pour la simulation en temps-réel, celle-ci effectuera également toute l'esquive sur sol glissant si le côté gauche est choisi, elle retournera ensuite sur le sol de base à la fin de l'esquive.

Fin de la méthode d'évitement

Lorsque le véhicule a atteint le déplacement latéral souhaité, il rétablit sa trajectoire pour retrouver l'orientation de base. Il applique donc un angle de braquage du volant opposé à celui utilisé pour l'esquive jusqu'à ce que le véhicule atteigne l'angle de lacet initial. Le fait de commencer la manoeuvre de rétablissement seulement lorsque le déplacement latéral a été atteint permet de s'assurer que l'obstacle sera bien évité avec une marge de sécurité. Lorsque 5 secondes se sont écoulées à partir du début de la méthode d'évitement (temps maximal nécessaire pour la réaliser) il redevient possible de piloter le modèle à l'aide des touches directionnelles du clavier du Raspberry.

4.2.5 Résultats

Cette section présente les résultats obtenus avec cette application ROS-Robotran. Elle présente principalement les vitesses de calcul et temps de réaction de la méthode utilisée mais n'aborde pas en détails les performances de l'esquive qui n'étaient pas l'objectif de l'étude.

Temps de calcul et temps de réaction

Les notions de temps de calcul et de temps de réaction utilisées dans cette section sont présentées sur l'échelle du temps ci-dessous :

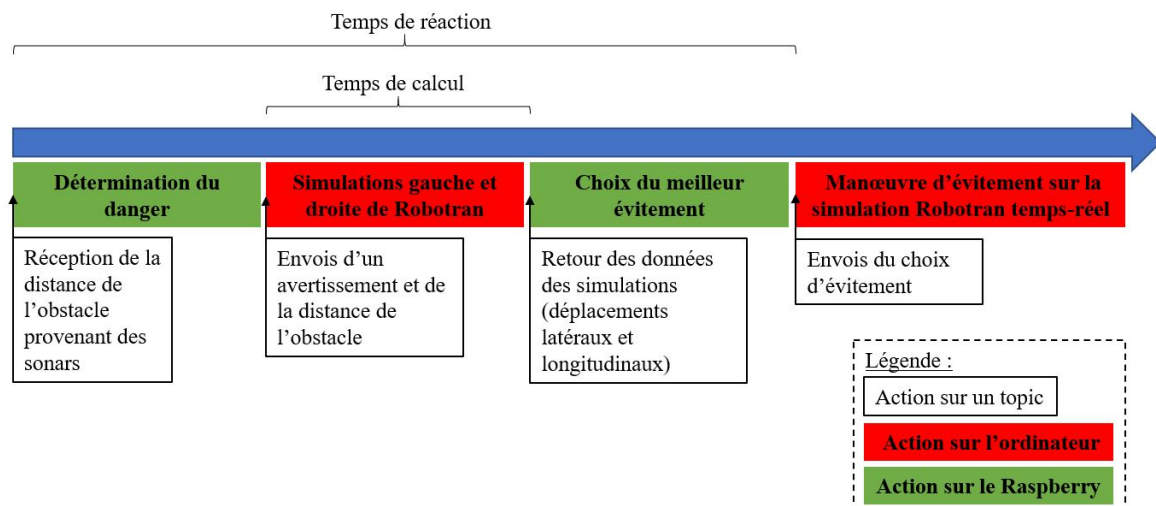


FIGURE 4.23 – Echelle du temps du protocole d'esquive sur le côté d'un obstacle

Paramètres des simulations

Les simulations sont réalisées en faisant rouler le véhicule à une vitesse constante en ligne droite et en plaçant ensuite un obstacle devant les sonars. Les paramètres suivants sont utilisés :

- Largeur de l'obstacle à gauche : 1.7m et largeur de l'obstacle à droite : 2.5m. Cette largeur d'obstacle tient compte de la largeur du véhicule pour les calculs à effectuer. C'est à dire que l'obstacle est plus petit en réalité : il faut soustraire la demie-voie du véhicule ($= 0.75m$) de chaque côté.
- Route glissante pour l'esquive à gauche où les forces et moment de contact pneu-sol F_x , F_y et M_z sont multipliées par un facteur égal à 0.75.
- La distance réelle mesurée par les sonars en mètre est multipliée par un facteur d'échelle de 200 pour correspondre à l'échelle des distances de freinage sur Robotran.
- Afin de prendre en compte le temps de calcul des deux simulations gauche et droite, le véhicule de ces deux simulations continue à rouler en ligne droite pendant 0.6 seconde une fois que l'obstacle est détecté avant d'entamer l'esquive. Il s'agit en effet du temps moyen entre le début des simulations gauche et droite et le début de la simulation en temps-réel (qui dépend de la manoeuvre à effectuer et de la rapidité de calcul des processeurs de l'ordinateur) pendant lequel la simulation en temps-réel attend de connaître le méthode choisie et continue donc à rouler en ligne droite. Cela permet donc d'avoir une trajectoire en temps-réel qui, elle, entame l'esquive directement et qui correspond donc à la trajectoire de prédiction choisie.

Résultats des simulations

La figure suivante présente les trajectoires de prédiction et la trajectoire utilisée lors de la détection d'un obstacle à une vitesse de 70 km/h. Les valeurs correspondent à la deuxième ligne du tableau 4.7 ci-dessous.

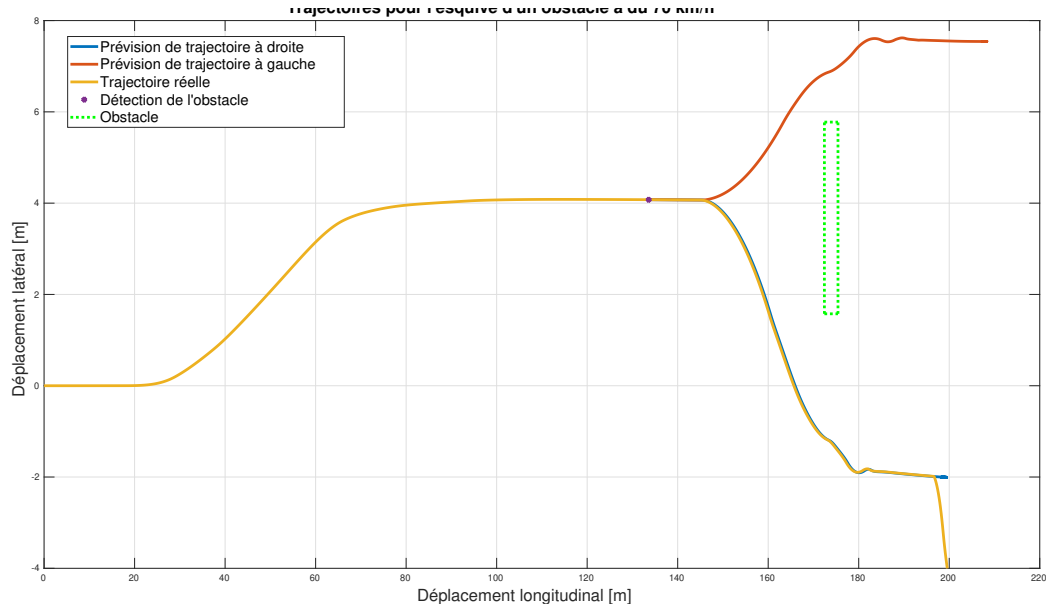


FIGURE 4.24 – Trajectoires pour l'esquive d'un obstacle à du 70 km/h

On observe tout d'abord que les deux choix de direction permettent d'éviter l'obstacle mais que le côté droit le fait sur une plus courte distance. On voit également la distance parcourue en ligne

droite après la détection de l'obstacle qui correspond à la distance réalisée par un véhicule à du 70 km/h pendant 0.6 seconde. Dans ce cas-ci, le temps entre le début des simulations de prédiction et la simulation réelle est légèrement inférieur à 0.6 seconde. La trajectoire réelle respectera donc bien la prédiction en ayant même une légère avance. Enfin, les manoeuvres effectuées sur les 120 premiers mètres et après 190 mètres correspondent à des changements de bande effectuées par l'utilisateur à l'aide des touches directionnelles du clavier.

Le tableau suivant présente divers scénarios d'esquive en fonction de la vitesse du véhicule et la distance de l'obstacle.

Vitesse [km/h]	Distance obstacle [m]	Temps de calcul [ms]	Temps de réaction [ms]	Trajectoire gauche	Trajectoire droite	Choix
50	26.42	610.70	667.42	lat : 1.7 m long : 21.99 m	lat : -2.5 m long : 18.96 m	Droite
70	38.83	542.89	617.68	lat : 1.7 m long : 29.66 m	lat : -2.5 m long : 27.01 m	Droite
90	39.44	534.15	598.33	lat : 1.7 m long : 37.27 m	lat : -2.5 m long : 34.74 m	Droite
90	76.42	/	19.42	/	/	Freinage
120	52.64	544.83	600.87	lat : 1.7 m long : 50.75 m	lat : -2.5 m long : 51.84 m	Gauche

TABLE 4.7 – Résultats des simulations ROS-Robotran avec sonars

On observe que le côté droit est plus souvent choisi sauf à haute vitesse où le glissement du sol à un impact plus faible sur le déplacement latéral car le véhicule utilise un angle de braquage plus faible.

Le temps de calcul est en moyenne de l'ordre de la demi seconde avec un temps de réaction logiquement plus long d'approximativement 60ms. De plus, lorsque les simulations de prédiction ne sont pas nécessaires pour la méthode réalisant simplement un freinage en ligne droite, on atteint un temps de réaction beaucoup plus faible de $\approx 20ms$.

Ces résultats permettent de valider la possibilité de l'utilisation d'une application ROS-Robotran dans un véritable véhicule autonome. De fait, pour rappel, le temps de réaction moyen pour un humain en bonne santé est de 1 seconde. Cette notion de temps de réaction est la même que celui utilisée ici et prend en compte : la perception et l'analyse du danger, le choix du conducteur et l'envoi de la commande à la pédale de frein et/ou au volant. Cependant ce temps de réaction peut varier entre 0.5 et 3 secondes en fonction de l'état du conducteur et de sa concentration [29]. L'utilisation de prédictions de trajectoire avec Robotran, dans ce cas-ci, peut se montrer équivalente à un humain ayant de bons réflexes et une bonne forme physique mais elle peut toutefois se montrer jusqu'à 5 fois plus rapide qu'un humain fatigué et/ou inattentif, ce qui est la cause de la plupart des accidents de la route¹².

Il faut toutefois garder en tête que cette méthode d'esquive a été simplifiée (sans prendre en compte la présence d'autres véhicules, le choix "éthique" à faire en présence d'usagers faibles, etc) et que le temps de réaction pourrait alors présenter une valeur plus élevée (qui pourrait alors être diminué avec l'utilisation de dispositifs de mesures et de calculs plus évolués).

12. Le site web *Jurifiable* reprend les données suivantes : le facteur humain apparaît dans plus de 90 % des accidents dont : entre 25 et 31% des accidents mortels dus à l'alcoolémie et la vitesse, 8 % des accidents mortels dus à la fatigue et la somnolence. Et enfin l'utilisation du téléphone au volant multipliant par 4 le risque d'avoir un accident.[30]

Chapitre 5

Simulateur à retour haptique

Ce chapitre présente l'application des logiciels ROS et Robotran à un simulateur de véhicules à retour haptique réalisé par trois étudiants de l'UCLouvain dans le cadre de leur mémoire. Il s'agit d'un simulateur de conduite permettant à un utilisateur de piloter un véhicule sur un interface graphique à l'aide d'un volant physique et d'en ressentir un retour haptique c'est-à-dire ressentir le retour de force du volant issu de la dynamique du véhicule dans ses bras et ses mains.

L'haptique se définit comme la science du toucher et de la perception du corps dans son environnement (c'est-à-dire la kinesthésie), elle est donc divisée en deux types de perception :

- La perception tactile : celle-ci se fait à l'aide de capteurs naturels présent dans la peau. Les stimuli tactiles y correspondant peuvent être perçus dans une bande passante allant de 0 à 400 Hz (mais pouvant monter jusqu'à 10kHz pour la perception de texture très fine). Elle permet donc principalement de percevoir les surfaces.
- La perception kinesthésique : elle est assurée par les muscles ainsi que l'oreille interne. Sa bande passante est plus basse et se trouve entre 20 et 30 Hz. Elle permet notamment de percevoir la forme, le poids et la résistance d'un objet lorsque nous le touchons ou tentons de le bouger. [31]

C'est principalement cette perception kinesthésique, et de manière secondaire la perception tactile, qui donnent lieu à un retour de force du volant réaliste ressenti par le conducteur.

La première section présente le simulateur tel que réalisé par *Nicola De Lellis* et *Quentin Materne* [2] et ensuite amélioré par *Romain Poletti* [3] dans le cadre de leurs mémoires respectifs¹. Elle se conclut par les problèmes persistants ainsi que les pistes d'améliorations proposées. La deuxième section introduit le nouvel hardware du simulateur ayant subi des modifications avec l'emploi du Raspberry utilisé tout le long de ce mémoire. La partie suivante développe l'architecture du software réalisée à l'aide de ROS ainsi que les protocoles de communication utilisés. Par après, la quatrième section décrit l'architecture de contrôle permettant d'automatiser les différentes communications présentes dans le simulateur. Enfin, ce chapitre se conclut par les résultats obtenus ainsi que de nouvelles pistes d'améliorations.

1. Ce simulateur sera ensuite désigné comme l'"ancien" simulateur dans la suite de ce travail. Le mémoire de *Nicola De Lellis* et *Quentin Materne* sera désigné comme le "premier" mémoire et celui de *Romain Poletti* comme le "second".

5.1 Présentation du simulateur

Cette section présente l'ancien simulateur à retour haptique dans ses grandes lignes afin de permettre la compréhension des nouveaux hardware et software implémentés. Afin de comprendre plus en détails le cheminement de la conception électromécanique et de l'ancien hardware/software utilisé, le lecteur est invité à se référer aux deux mémoires précédents portant sur ce sujet [2][3].

Le simulateur est divisé en 5 dispositifs principaux :

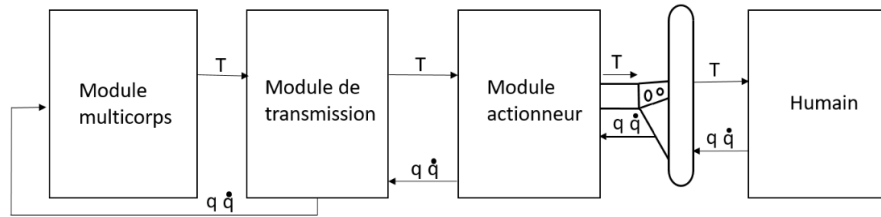


FIGURE 5.1 – Illustration des 5 composants principaux du simulateur de véhicule [3]

- **Le module multicorps** : il est représenté par le logiciel Robotran, il s'occupe de la simulation dynamique du véhicule et du calcul du couple de retour dans le volant T en fonction de la position q et vitesse $\dot{q}$ de ce dernier².
- **Le module de transmission** : il reçoit le couple calculé par le module multicorps et l'envoi au module actionneur sous forme de consigne. Il reçoit également la position et vitesse du volant provenant du module actionneur et l'envoie au module multicorps qui les utilise dans la résolution des équations du mouvement du véhicule.
- **Le module actionneur** : il permet d'appliquer un couple au volant et de mesurer la position et vitesse de celui-ci.
- **Le volant** : il est l'élément physique se trouvant entre le pilote et les trois modules précédents utilisés pour la simulation dynamique d'un véhicule. C'est à travers lui que se fait le ressenti haptique.
- **L'humain** : il permet de fermer la boucle des 5 composants. Il va interagir avec le modèle multicorps du véhicule en imposant une position et vitesse du volant et en ressentant le couple de retour dans celui-ci.

Ancien hardware et protocole de communication

Les différents modules présentés ci-dessus sont implémentés dans des dispositifs hardware reliés par des connections physiques et/ou des protocoles de communication. Le tableau suivant résume les implémentations réalisées par les 3 étudiants précédents (en explicitant les modules et les liaisons modifiés qui seront présentés dans les sections suivantes) :

2. Tout comme les deux anciens mémoires, l'accélération du volant n'a également pas été prise en compte dans ce travail.

Module & Liasion physique	Présentation
Module multicorps : <i>Ordinateur</i> (n° 8 sur la figure 5.3) → modification du software par l'utilisation de ROS-Robotran	L'ordinateur héberge une session de Robotran pour la simulation du véhicule. Lors du deuxième mémoire une attention particulière a été apportée à l'architecture de contrôle à l'intérieur de ce module où une architecture avec interruption (IRQ) a finalement été choisie. Elle s'apparente à l'architecture utilisée par le paquet <i>ROS ros_rob</i> utilisant du multithreading et des mutexs permettant d'interrompre les threads. Leurs avantages communs est une désolidarisation du processus Robotran des deux autres processus s'occupant de la réception de q et $\dot{q}$ et de l'envoi de T pour utiliser la pleine capacité d'un processeur pour la boucle temporelle de Robotran.
Liaison : <i>série RS-232 (USB)</i> → modifiée par des topics ROS	Il s'agissait d'une communication en série (où les données circulent sur une seule voie) entre l'ordinateur et le module de transmission. Les données y étaient transportées sous forme d'entier (<i>int</i> de 32bits) à une vitesse de transmission de 230 400 baud.
Module de transmission : <i>Carte de développement ICE AM3359</i> → remplacée par Raspberry Pi2 Model B (n° 7)	Le module de transmission qui servait à synchroniser les informations entrantes et sortantes du module multicorps et actionneur était une carte de développement de chez <i>Texas Instrument</i> . Elle présentait l'avantage d'être codable en C, d'être compatible avec la communication série et CAN et d'avoir une cadence de processeur de 800 MHz.
Liaison : <i>paire différentielle CAN (DB9)</i> → non modifiée	Le câble CAN avec port DB9 vers la carte de développement était utilisé pour le protocole de communication CANopen nécessaire pour communiquer avec le module actionneur. Il est présenté en détail à la section 5.3.1 car celui-ci est toujours utilisé dans le "nouveau" simulateur.
Module Actionneur : <i>Servo-moteur LXM32AD30M2</i> (n° 5), <i>Moteur BMH1401P12A2A</i> (n° 3) et <i>Codeur</i> (n° 3) → non modifiés	Le module actionneur, n'ayant également pas subi de modification, sera présenté plus en détails à la section 5.2.4. L'ensemble des dispositifs le composant ont été choisis sur base d'un cahier des charges réalisés lors du premier mémoire portant sur le sujet.
Liaison : <i>en prise directe</i> → non modifiée	Le servo-moteur qui génère le couple de retour est dit <i>en prise directe</i> , c'est-à-dire qu'il n'y a aucun élément de transmission (courroie, engrenage, etc) entre lui et le volant. Cela offre l'avantage d'un simulateur compact et ayant peu de jeu afin d'avoir une faible différence entre le couple de retour dans le volant et celui calculé par Robotran.
Volant (n° 2) (et autres structures) → non modifiés	Lors du premier mémoire, un assemblage mécanique permettant d'accueillir le servo-moteur et le codeur et de les lier à un volant physique a été dimensionnée et confectionnée (voir figure 5.2). Il était alors possible de choisir un angle d'inclinaison du volant de 15° à 30°. Dans le cadre du second mémoire, l'assemblage général a été amélioré en y ajoutant une structure en profilés carrés <i>Bosch-Rexroth</i> (réalisée avec <i>Aubain Verlé</i>) permettant d'accueillir un écran (n° 2) placé à la bonne hauteur pour une simulation de conduite ainsi qu'un siège baquet (n° 1).

TABLE 5.1 – Tableau résumant le simulateur à retour haptique réalisé par *Nicola De Lellis, Quentin Materne et Romain Poletti*

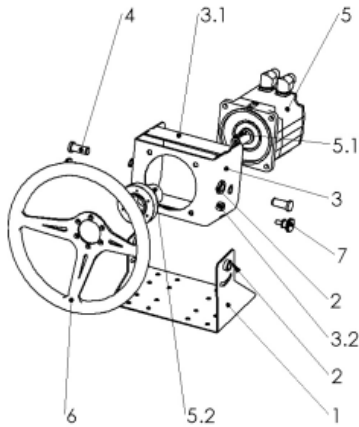


FIGURE 5.2 – Vue éclatée de l'assemblage mécanique sous *SolidWorks* [2]

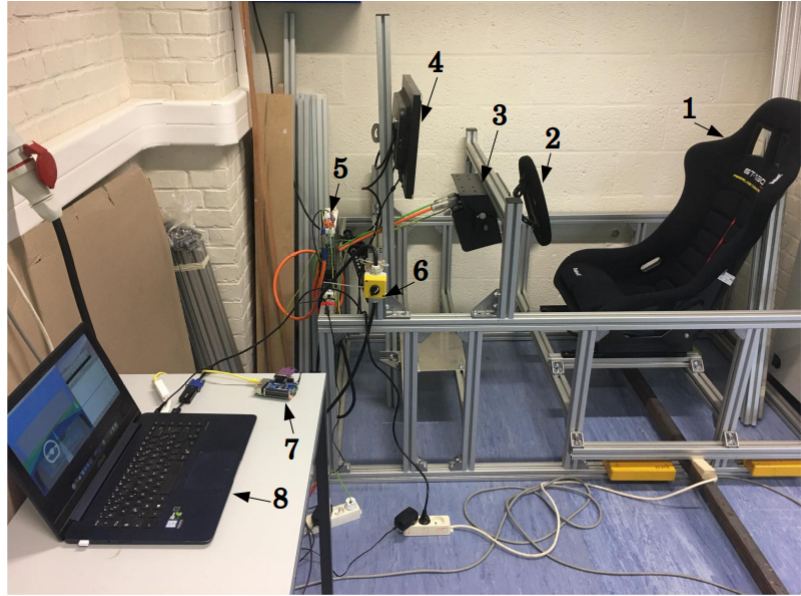


FIGURE 5.3 – Vue d'ensemble du simulateur

Problèmes persistants et pistes d'amélioration

Bien que ces deux travaux aient aboutis à un simulateur entièrement fonctionnel et présentant déjà de bonnes performances, certains problèmes persistaient.

Communication série L'utilisation d'une communication série entre le module multicorps et le module de transmission était fortement chronophage. De plus, elle n'était pas adaptée aux ordinateurs actuelles et nécessitait l'utilisation d'émulateur peu efficace (voir section 5.2.3). Sa complexité et sa lenteur était donc un frein à la communication entre ces deux modules alors qu'il est démontré au chapitre précédant que ROS offre une nette amélioration à ce niveau, ce qui justifie son utilisation.

Modèle multicorps Le modèle multicorps Robotran utilisé pour les simulations était le kart présenté à la section 1.3.2. Celui-ci a l'inconvénient de ne pas avoir un modèle de contact pneu/sol adapté nécessitant alors de modification au niveau du modèle de contact de Bakker utilisé ainsi qu'une relaxation des pneus qui causait une instabilité de l'auto-alignement du volant. Son avantage était son faible nombre d'articulations qui permettait une résolution rapide des équations de mouvement afin que celle-ci n'impacte pas sur les performances générales du simulateur. Il est démontré à la section des résultats 5.5 que Robotran (pour peu que le pas de temps utilisé pour la résolution des équations du mouvement reste suffisamment petit) n'a pas d'impact négatif sur les performances du simulateur. L'utilisation de modèle multicorps différent, la voiture, avec un modèle de contact pneu/sol adapté et système de direction plus évolué est donc une piste d'amélioration.

Adaptabilité L'ensemble dans l'ancien simulateur avait été réalisé avec deux modèles multicorps (un kart présentant de haute instabilité au niveau de la conduite qui a été abandonné et le kart présenté à la section 1.3.2). La complexité de l'implémentation de la communication en série ainsi que les nombreuses modifications à apporter au projet Robotran ne permettait pas une adaptation aisée à un nouveau modèle Robotran ou l'utilisation d'un autre module multicorps (i.e. un autre ordinateur). Les deux chapitres précédents démontrent que l'utilisation de ROS permet de palier ce problème.

Module de transmission Bien que la carte développement de chez *Texas Instrument* présentait l'avantage d'être codable en C, il était difficile d'y implémenter et exécuter les fonctions. L'utilisation du Raspberry sous Linux, permettant une modification aisée des fonctions qu'il exécute, se justifie donc comme changement.

5.2 Architecture de l'hardware

Cette partie présente tout d'abord l'architecture globale du "nouveau" simulateur avec une modification majeure au niveau du module de transmission, à savoir le remplacement de la carte de développement ICE AM3359 de chez *Texas Instrument* par un Raspberry Pi 2 et son PiCAN 2. Les différentes liaisons entre les nouveaux dispositifs sont ensuite exposées. Enfin, chaque composant sera développé en détails.

5.2.1 Architecture globale

Le schéma ci-dessous présente les différents composants hardware utilisés ainsi que leur alimentation nécessaire. Le tableau 5.2, lui, contient la description des liaisons physiques entre chaque dispositif.

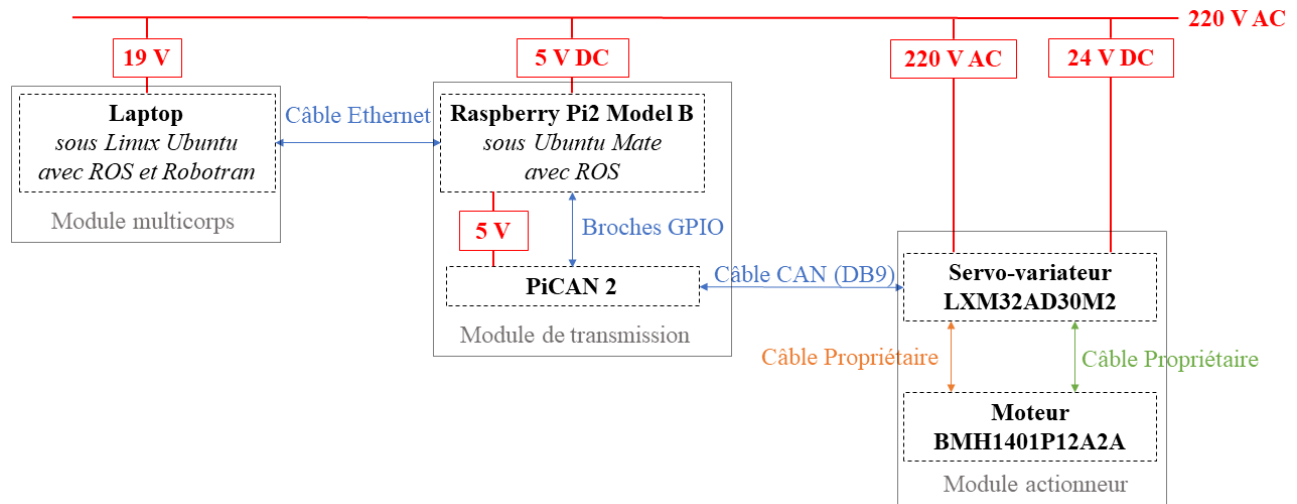


FIGURE 5.4 – Architecture physique globale

Liaison physique	Type de connexion et protocole de communication
Câble Ethernet Cat-5e	Connexion bi-directionnelle et complète pour un protocole de communication SSH
Broches GPIO	General Purpose Input/Output : envoient et reçoivent des signaux électriques, fournissent également une alimentation 5V
Câble CAN	Paire différentielle CAN (port DB9 sur le Raspberry et port RJ45 sur le servo-variateur) pour un protocole de communication CANOpen
Câble Propriétaire Orange	Alimentation triphasée de l'étage de puissance
Câble Propriétaire Vert	Liaison pour le codeur incrémental du moteur

TABLE 5.2 – Type de connexion et protocole de communication des liaisons physiques

L'alimentation des dispositifs se fait à l'aide d'adaptateurs qui convertissent le 220 V alternatif. Le servo-variateur, lui, possède deux alimentations :

- Le 220 V AC pour l'alimentation de l'étage de puissance (ou autrement dit, le moteur). Celle-ci se fait par l'intermédiaire du câble propriétaire orange. Il est ensuite possible d'activer et de désactiver l'étage de puissance à l'aide du protocole de communication CANOpen.
- Le 24 V DC pour l'alimentation de l'étage de commande (ou servo-variateur).

5.2.2 Module multicorps

Ordinateur

L'ordinateur est la pièce principale et la plus interchangeable de cette architecture. En effet, c'est à partir de ce dernier que l'utilisateur va lancer les simulations et en choisir la configuration (type de véhicule, vitesse initiale, choix des paramètres de la géométrie du véhicule dans Robotran,...). Celui-ci doit tourner sous Linux Ubuntu et posséder les logiciels ROS et Robotran.

Robotran permet de calculer le couple de retour du volant en fonction de la position et vitesse de ce dernier et ROS s'occupe du protocole de communication avec le Raspberry. ROS permet également cette interchangeabilité de l'ordinateur. En effet, l'ensemble du code pour l'intégration ROS-Robotran et la communication avec le Raspberry est disponible sous forme d'un paquet ROS accompagné de tutoriels d'installation et d'utilisation. L'utilisateur peut alors télécharger ce paquet sur son propre ordinateur, y intégrer de nouveaux projets Robotran si il le souhaite et établir une connexion complète et bi-directionnelle avec le Raspberry. A partir de là, il est possible d'utiliser la pleine capacité du simulateur à retour haptique.

5.2.3 Module de transmission

Raspberry Pi 2 Modèle B

Tout d'abord, comme expliqué ci-dessus, il a fallu opérer à un changement de l'organe de contrôle de chez *Texas Instrument*. De fait, l'objectif était d'implémenter Robotran dans un système embarqué (ici le volant haptique plus particulièrement) à l'aide de ROS. Cependant ROS ne supporte officiellement que le système d'exploitation Linux Ubuntu et Debian³ qui ne pouvait pas être installé sur la carte de développement.

De plus, cela a permis de remplacer la liaison en série RS-232 entre l'ordinateur et la carte par une liaison Ethernet entre l'ordinateur et le Raspberry. Pour la communication série, il était nécessaire d'installer un driver sur l'ordinateur afin d'émuler un port série sur ses ports USB⁴. Ce driver utilisait un buffer interne servant de zone tampon entre le port USB et l'application utilisée et ayant une taille minimum. C'était donc un timer qui gérât la fréquence de réception des données à garder et transférer. Ce timer présentait l'inconvénient d'avoir un temps de latence de $\pm 2\text{ ms}$. La communication sur des topics ROS via un câble Ethernet Cat-5e a donc l'avantage d'être plus rapide et plus stable.

Les détails sur les performances et l'utilisation d'un Raspberry dans un projet ROS-Robotran ont déjà été présentés précédemment.

3. Du moins, au moment du choix du changement de l'hardware, car ROS propose maintenant une version disponible sur Windows.

4. Les ports séries (ou port COM) n'existant plus sur la plupart des nouveaux ordinateurs.

PiCAN 2

Le Raspberry ne respectait cependant pas une des contraintes de communication, à savoir, le servo-variateur qui, lui, reste inchangé, disposant d'une fonction de communication CAN/CANopen. C'est un protocole de communication fortement utilisé dans des systèmes embarqués en temps réel qui sera présenté à la section 5.3.1.

Au niveau du software le Raspberry gère nativement l'interface CAN grâce au module kernel SocketCAN de Linux. Le kernel de Linux comprend également des drivers SocketCAN pour les puces MCP2515⁵. En revanche le Raspberry ne possède pas de port physique permettant le branchement d'un câble de communication CAN.

Il a donc fallut modifier l'hardware avec l'achat d'un PiCAN 2 de chez *SK Pang Electronics Ltd*. Il s'agit d'une carte utilisant la puce de contrôle MCP2515 avec un émetteur-récepteur CAN MCP2515. En la branchant sur les broches GPIO du Raspberry, celle-ci fournit une capacité de communication CAN au Raspberry via un port DB9 (préalablement utilisé sur la carte de développement de *Texas Instrument*). [32]

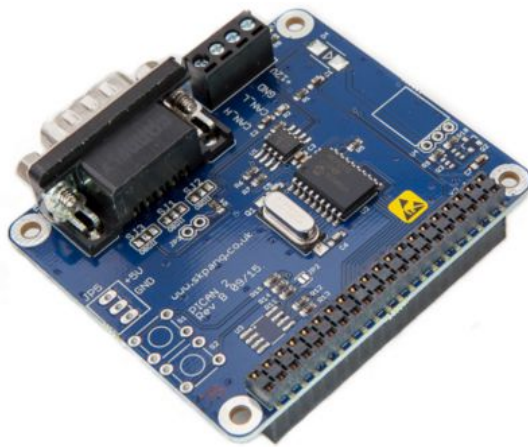


FIGURE 5.5 – Carte PiCAN CAN-Bus pour Raspberry Pi [32]

5. La puce MCP2515 est un contrôleur CAN autonome développé pour simplifier les applications nécessitant une interface avec un bus CAN.

5.2.4 Module actionneur

Le module actionneur n'a subi aucun changement par rapport aux deux mémoires précédents. Il est tout de même présenté afin de ne pas porter préjudice à la compréhension du fonctionnement complet du simulateur. Ce module est composé de trois dispositifs, un servo-variateur, un servo-moteur (ou moteur) et un codeur, qui sont détaillés ci-dessous.

Servo-variateur LXM32AD30M2

Le servovariateur (ou "étage de commande") de chez *Schneider Electric* est le dispositif responsable du contrôle du moteur et de la communication avec le Raspberry. Son utilisation est double :

- Il reçoit la position angulaire du volant mesurée par le codeur incrémental et calcul la vitesse angulaire de celui-ci. Il envoie ensuite ces deux données ^a au Raspberry.
- Il reçoit le couple de retour calculé par Robotran via le Raspberry et l'envoie au moteur via une consigne. La consigne de couple est exprimée en % par rapport au couple de maintien du servo-moteur $T_{maintient} = 1030 \text{ Ncm}$. L'envoi de la consigne se faisant par incréments de 0.1%, nous obtenons une précision du couple de 0.0103 Nm.

Les envois et réceptions des données et consignes se font à l'aide du protocole de communication CANOpen.

^a. La position et vitesse angulaire sont calculées en *unités d'utilisateur* propre au servo-moteur, il faut alors utiliser un facteur d'échelle pour les convertir en radian et radian/s qui sont les unités utilisés par Robotran.

Servo-moteur BMH1401P12A2A

Le moteur (ou "étage de puissance") contrôlé par le servo-variateur provient également de chez *Schneider Electric*. Il s'agit du modèle *Berger Lahr BMH1401P12A2A* dont les caractéristiques principales sont présentées ci-dessous. [34]

Couple de maintien	Couple maximal	Vitesse maximale	Puissance nominale	Inertie du rotor
10.3 [Nm]	30.8 [Nm]	4000 [rpm]	1.74 [kW]	16.46 [kgcm ²]

TABLE 5.3 – Caractéristiques du servo-moteur



FIGURE 5.6 – Servo-variateur LXM32AD30M2 [33]



FIGURE 5.7 – Servo-moteur BMH1401P12A2A [34]

Codeur

Afin de mesurer la position angulaire du volant, un codeur absolu est intégré dans le moteur. Le modèle correspondant au moteur utilisé est le codeur *SinCos Hiperface SKM36 Multiturn* [34].

Résolution par révolution	Nombre de tours maximal	Précision
128 période sin/cos	4096	$\pm 0.022^\circ$

TABLE 5.4 – Caractéristiques du codeur

Le codeur étant de type absolu, il doit effectuer une prise d'origine à chaque initialisation du servo-variateur. Il mesure alors la position actuelle du volant de manière incrémentale par rapport à cette position de référence.

5.3 Architecture du software et protocoles de communication

Etant donné la modification d'une pièce centrale de l'hardware (i.e. le module de transmission), l'architecture du software et les protocoles de communication utilisés ont dus être entièrement revus. Comme déjà présenté ci-dessus, la communication entre les différents dispositifs va se faire à l'aide de ROS avec :

- Un échange de données $(q, \dot{q}, T)$ sur des topics entre l'ordinateur et le Raspberry.
- Un échange de données sur un bus CAN entre le Raspberry et le servo-variateur grâce au PiCAN2 et à un paquet ROS gérant la communication CANopen.

Ces deux protocoles, et principalement la communication CANopen, sont présentés ci-dessous.

5.3.1 CANopen

CANopen est un protocole de communication exploitant un bus de terrain CAN pour permettre à des dispositifs tels que les ordinateurs, capteurs, actionneurs, ... de communiquer entre eux.

Le bus CAN (pour *Controller Area Network*) est un bus de communication en série qui connecte plusieurs appareils sur un même câble (ou bus). Chaque appareil communique et/ou récupère à son tour (suivant un ordre de priorité prédéfini) des données sur le bus. Fortement utilisée dans l'industrie automobile car il permet de connecter les nombreux capteurs et actionneurs présents dans les voitures actuelles avec une complexité de câblage réduite, il a maintenant été également adapté à l'automatisation industrielle.

Les dispositifs (également appelés noeuds⁶) communiquent entre eux à l'aide d'objets. Ces objets peuvent véhiculer des données, gérer l'établissement de la connexion et surveiller l'ensemble du réseau. Afin de lier ces objets, chaque appareil dispose d'un dictionnaire d'objets. Dans celui-ci les objets sont référencés par un index et leurs champs de données par des sous-index. Servant d'interface entre l'appareil et le bus CAN, ce dictionnaire permet à un dispositif d'envoyer une commande à un autre en se référant à son dictionnaire d'objets. Enfin, chaque dispositif connecté au bus de terrain CAN est référencé par une adresse ID (un *int* compris en 1 et 127).

6. A ne pas confondre avec les noeuds ROS.

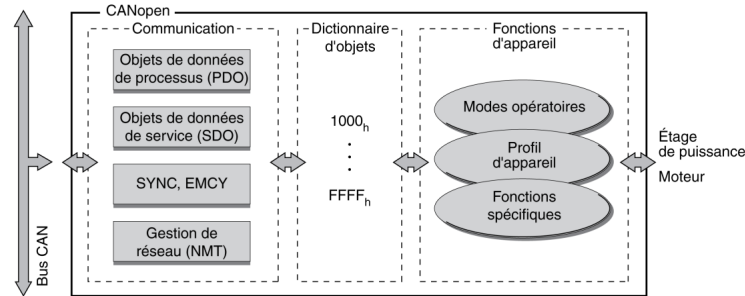


FIGURE 5.8 – Protocole de communication CANopen [35]

Comme le montre la figure 5.8, il existe plusieurs moyens d'accéder au dictionnaire d'objets d'un appareil afin de lire ou écrire dans leurs champs de données :

- Les **Objets de Données de Processus (PDO)** : ils sont fortement utilisés pour l'échange de données en temps réel (comme l'envoi d'une consigne ou la lecture d'une position actuelle) et se basent sur la relation producteur/consommateur. Leur rapidité est due au fait que le transfert de données ne nécessite aucune confirmation de réception et que celui-ci se fait sans données de gestion supplémentaires. De plus ils possèdent une longueur de message variable allant jusqu'à 8 octets, c'est-à-dire que si on envoie une consigne d'une taille de 2 octets, seuls ces 2 octets de données seront envoyés. Il faut toutefois recourir à une configuration préalable des zones de données occupées par les PDOs ainsi que de la taille de leurs messages pour qu'ils puissent accéder aux index et sous-index du dictionnaire d'un autre appareil, cela s'appelle le *mappage* (ou "*mapping*") de PDO.
- Les **Objets de Données de Services (SDO)** : ils sont principalement utilisés pour la configuration d'un appareil avec un principe de communication client/serveur. Ils accèdent directement aux zones du dictionnaire d'objets d'un autre appareil via les index et sous-index afin d'y lire ou modifier des informations. Ils n'ont donc pas besoin de mappage mais leur communication est plus lente car elle nécessite un message de confirmation de réception du destinataire.
- **SYNC, EMCY** : SYNC permet la synchronisation du réseau et EMCY détecte les erreurs présentes sur celui-ci.
- **La Gestion de réseau (NMT)** : la NMT permet de définir et modifier l'état de fonctionnement d'un appareil sur le réseau. Au moment de la mise sous tension de l'appareil, celui-ci passe par une phase d'initialisation pour se préparer à la communication avec le bus CAN et se place ensuite en état "pré-opérationnel". Cet état permet l'utilisation de SDOs pour la configuration mais pas des PDOs. Il est alors possible de changer les états de fonctionnements via des services NMT :
 1. Activer le noeud réseau (les PDOs y sont disponibles).
 2. Désactiver le noeud réseau.
 3. Passer en mode "pré-opérationnel".
 4. Réinitialiser le noeud.
 5. Réinitialiser les données de communications.[35]

Paquet ROS KaCanOpen

Afin d'établir une communication CANopen entre le Raspberry sous ROS et le servo-variateur, le paquet ROS *KaCanOpen* fourni par *KITmedical* sur GitHub a été utilisé. KaCanOpen permet

de faire le pont entre une communication CANopen et une communication ROS. Son utilisation est dite "plug and play" car en lui indiquant l'adresse ID du dispositif (ainsi que le nom et la vitesse de transmission du bus CAN) il va le détecter automatiquement et établir une connexion avec. Son architecture se divise en quatre parties [36] :

- Le module **Driver** réutilise des drivers CAN classiques comme PCAN mais surtout SocketCAN qui justifie le choix de Linux et du PiCAN2 qui utilisent ce même driver. Ce driver fournit la capacité de communication avec un bus de terrain CAN.
- Le **Core** implémente les protocoles de communication CANopen présentés ci-dessus : SDO, PDO et NMT. Cette implémentation, bien qu'elle ne soit pas conçue pour minimiser l'impact sur la mémoire vive, se prête parfaitement à l'utilisation de nano-ordinateur mono-carte sous Linux tout comme les Rapsberrys.

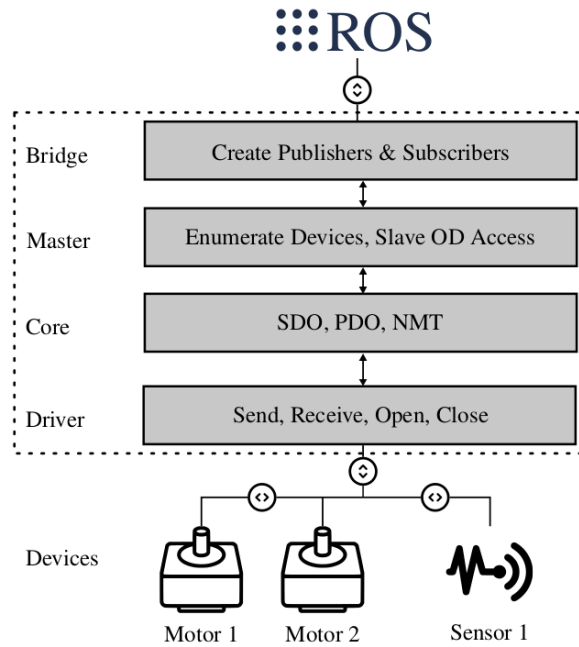


FIGURE 5.9 – Architecture de KaCanOpen [36]

- Le **Master** s'occupe d'énumérer, d'identifier et d'initialiser les dispositifs. Il gère la réception de données CANopen pour les envoyer vers un autre dispositif CANopen ou sur un topic ROS. Inversement, il peut recevoir des données d'un topic ROS et les envoyer vers un dispositif CANopen. Cet envoi/réception de données CANopen se fait via l'utilisation des index et sous-index (qui sont des valeurs hexadécimales) ou via l'utilisation d'un fichier EDS (voir ci-dessous).
- **Rosbridge** est, comme son nom l'indique, le seul module de KaCanOpen utilisant les fonctionnalités de ROS. Il utilise le Master pour directement lier les envoi/réceptions de communication CAN à des topics, services ou autres actions de ROS.

Enfin, comme la plupart de paquets ROS, KaCanOpen est disponible en opensource. Son site *Github* [37] fournit de nombreuses informations et exemples sur son utilisation. Il met également à dispositions des *todo lists* avec les nouvelles implémentations à venir, l'utilisateur peut alors fournir des pistes d'amélioration ou de propositions de code.

Fichier EDS du servovariateur

Un fichier EDS, pour Electronic Data Sheet, est un fichier texte se basant sur des profils standardisés de la communication CANopen dont les deux principaux profils sont :

- Profil de communication DS301 : pour la gestion de l'interface entre l'appareil et le bus CAN (échange de données entre les différents dispositifs du réseau CAN, initialisation de l'appareil,...)
- Profil d'appareil DSP402 : pour la gestion du positionnement, de la surveillance et des réglages d'un dispositif (changement des modes opératoires, envoi d'une consigne de couple, ...)

Ce fichier, propre à chaque dispositif CAN, recense le dictionnaire d'objets lui correspondant et associe chaque index et sous-index à un nom de paramètre. Le paquet *KaCanOpen* permet de charger ce fichier EDS dans l'exécutable ROS et de le lier au dispositif utilisé (ici le servo-variateur). Ainsi il est possible de recevoir et envoyer des communications CAN à l'aide des noms des paramètres au lieu d'utiliser les valeurs hexadécimales. Par exemple, la commande pour envoyer une consigne pour régler le couple du moteur sur 10 % qui était au préalable⁷ en hexadécimale :

→ 601/ 2B 71 60 00 64 00 00 00

devient en langage *C++* :

`myservo.set_entry("Target Torque",10);`

Cela a l'avantage de simplifier grandement l'écriture et la compréhension du code mais aussi d'offrir un code standardisé pouvant être réutilisé pour un autre dispositif sans devoir y apporter de modifications majeures. [36]

Problèmes persistants

En partie à cause de la situation exceptionnelle due au COVID-19 qui a limité la possibilité de tests en laboratoire sur le simulateur à retour haptique, deux problèmes persistent avec l'utilisation de *KaCanOpen*.

Le mappage PDO Afin d'utiliser des requêtes PDOs (plus rapide que les SDOs) il faut procéder à un mappage comme expliqué ci-dessus. Les commandes pour réceptionner la position et vitesse du volant sont configurées par défaut pour un protocole PDO et s'obtiennent donc directement à l'aide du Master *KaCanOpen* et du fichier EDS. Cependant la commande pour la consigne de couple est, elle, configurée pour un envoi SDO par défaut. Il faut alors procéder à un mappage pour référencer l'index et sous-index de la commande pour la consigne de couple à un protocole PDO. Cette manipulation n'a pas pu être réalisée par manque de temps pour maîtriser entièrement l'utilisation des fonctions de mappage et d'initialisation de nouveau PDO avec *KaCanOpen* en les testant sur le simulateur. La conséquence est une diminution de la vitesse de communication faisant passer le temps d'envoi de la commande de couple d'un présumé facteur 10^{-4} à 10^{-3} .

Rosbridge Comme expliqué ci-dessus, *Rosbridge* permet de directement lier les envois/réceptions *CANopen* à des topics ROS. Il n'a pas été possible de tester les fonctionnalités de *Rosbridge* afin de les exploiter pour les messages et topics nécessaires pour ROS-Robotran. Les données *CANopen* passent donc par un stockage dans une variable avant d'être directement envoyées ou après avoir été réceptionnées via des topics. Etant donné que le temps d'envoi/réception via des topics en passant par le stockage dans une variable est largement inférieur à l'envoi/réception de données *CANopen* (cela sera démontré à la section 5.5), cette simplification est supposée comme n'ayant pas d'impact

7. Pour un dispositif ayant comme adresse ID la valeur 1.

conséquent sur la vitesse de communication. L'utilisation de Rosbridge sera tout de même proposée dans les améliorations afin de valider/invalidier cette simplification.

5.3.2 Topic

Les topics ROS ont déjà été largement discutés et utilisés dans le chapitre précédent. L'architecture de ROS est semblable à celle utilisée pour l'exemple de l'actionneur extérieur avec l'utilisation de deux topics (voir section 4.1).

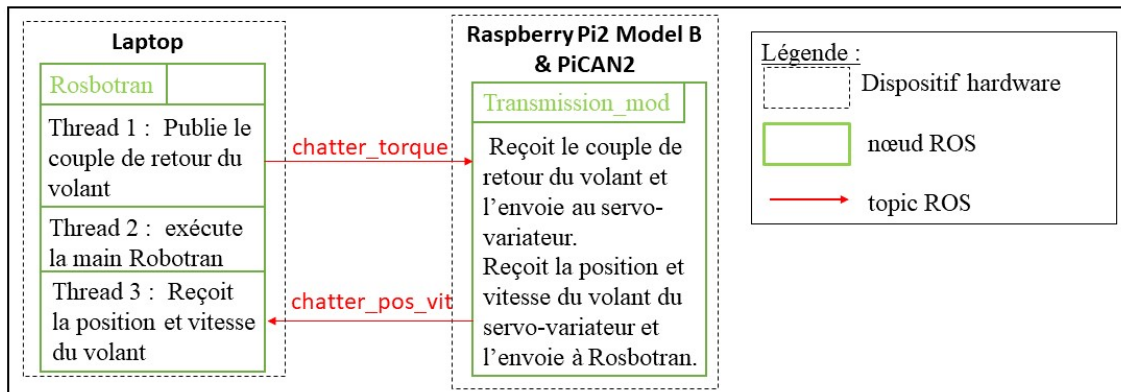


FIGURE 5.10 – Architecture globale de ROS pour le simulateur haptique

Le nœud *Rosbotran* reste inchangé, ces trois threads gérant l'envoi du couple, la résolution des équations du mouvement via Robotran et la réception de la position et vitesse du volant.

Le nœud *Transmission_mod* (ou aussi appelé *lexium32A_CANopen*), pour *Module de Transmission*, possède les fonctionnalités du paquet KaCanOpen. Il réceptionne le couple via un topic et l'envoi sous forme de consigne CANopen au servo-variateur. Il réceptionne également la position et vitesse envoyées par le servo-variateur et les publie sur un topic.

5.4 Architecture de contrôle

L'architecture de contrôle est à nouveau semblable à celle de la section sur l'utilisation d'un actionneur extérieur 4.1 avec l'utilisation de fonctions *Callback* qui permettent de rendre indépendantes les boucles temporelles des deux nœuds.

5.4.1 Fréquence des boucles des noeuds ROS

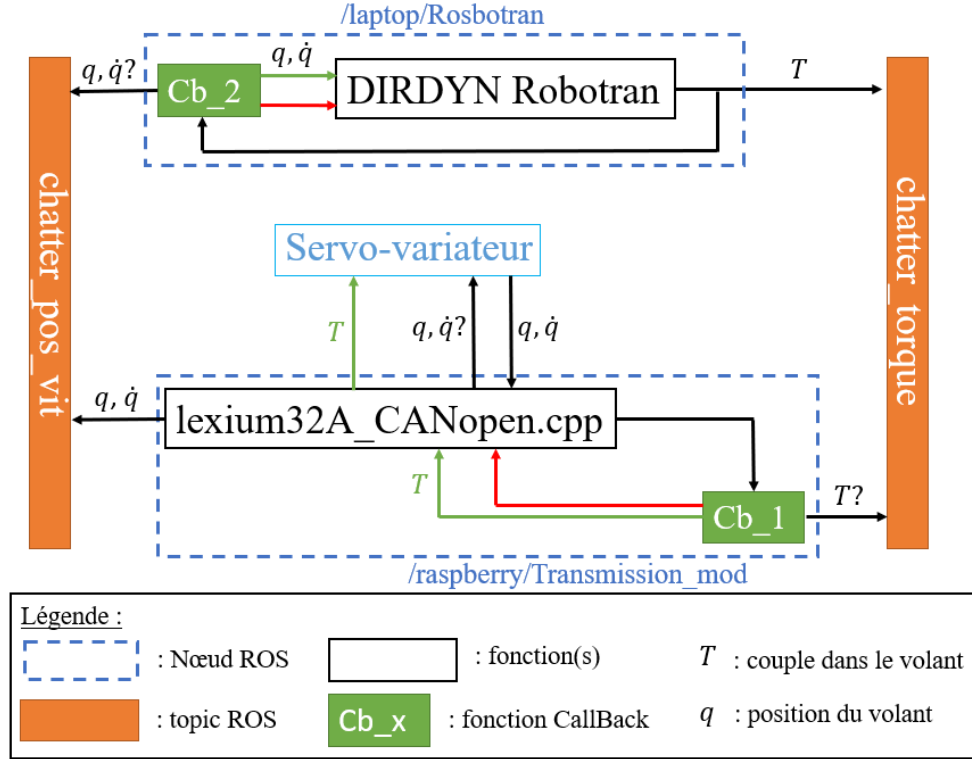


FIGURE 5.11 – Architecture du protocole de communication pour le simulateur haptique

La boucle du noeud *Rosbotran* tourne à nouveau à la fréquence imposée par le pas de temps utilisé pour la dynamique directe (DIRDYN) de Robotran. A chaque pas de temps elle publie sur le topic *chatter_torque* la consigne de couple en % dans le volant calculée par Robotran. A l'aide d'une fonction *callback* (*Cb_2* sur la figure 5.11) elle vérifie également, à chaque pas de temps, si une nouvelle information sur la position et vitesse du volant a été fournie par le noeud *Transmission_mod*. Dans le cas contraire, elle garde la dernière position et vitesse utilisée.

La boucle du noeud *Transmission_mod* tourne à la fréquence imposée par le temps d'exécution des fonctions appelées récursivement à l'intérieur de celle-ci, à savoir :

1. Deux fonctions du master KaCanOpen qui réceptionnent la position et vitesse du volant via l'envoi d'une commande PDO et les stockent dans des variables.
2. Une fonction qui publie la position et vitesse du volant sur le topic *chatter_pos_vit*.
3. L'appel de la fonction *Callback* (*Cb_1* sur la figure 5.11) qui réceptionne la consigne de couple et la stocke dans une variable.
4. Une fonction du master KaCanOpen qui envoie la consigne de couple au servo-variateur via une commande SDO.
5. Une fonction qui permet de vérifier, à l'aide des paramètres ROS si la simulation sur Robotran est finie ou non. Elle est appelée toutes les 500 itérations car elle est fortement chronophage. La valeur mesurée pour cette fonction dans le tableau 5.6 correspond à la moyenne du temps d'exécution comprenant le fait qu'elle n'est pas appelée à chaque itération.

Les temps moyens d'exécution de ces fonctions mesurés sur un échantillon de 7001 itérations sont repris dans le tableau suivant :

Fonction	Réception q et $\dot{q}$ via PDO, $\Delta t_{q\dot{q},PDO}$	Envoi q et $\dot{q}$ via topic, $\Delta t_{q\dot{q},ROS}$	Réception T via topic, $\Delta t_{T,PDO}$	Envoi T via SDO, $\Delta t_{T,SDO}$	Paramètre ROS, $\Delta t_{rotparam}$
Temps [s]	$1.693 * 10^{-3}$	$1.03 * 10^{-4}$	$7.3 * 10^{-5}$	$1.274 * 10^{-3}$	$1.2 * 10^{-5}$

TABLE 5.5 – Temps d'exécution des fonctions de *Transmission_mod*

En comparant ces temps avec les ordres de grandeurs obtenus par le deuxième mémoire travaillant sur le simulateur⁸ :

Fonction	Réception q et $\dot{q}$ via PDO	Envoi q et $\dot{q}$ via UART	Réception T via UART	Envoi T via PDO
Temps [s]	10^{-3}	10^{-4}	10^{-2}	10^{-4}

TABLE 5.6 – Ordre de grandeur des temps d'exécution des fonctions de l'ancien module de transmission

On observe que les temps d'envoi et de réception de q et $\dot{q}$ sont équivalents. L'envoi de T via un protocole SDO (dont la raison a été expliquée ci-dessus) accuse un ordre de grandeur plus élevé que l'utilisation d'un protocole PDO. Le temps pour la réception de T montre une nette amélioration grâce à l'utilisation de ROS⁹. Cependant, la fonction de réception de T via UART tournait en parallèle des 3 autres fonctions afin de limiter son impact. L'utilisation de multithreading sur le Raspberry pour séparer les fonctions d'envoi/réception via topics et via CAN est alors une piste d'amélioration qui n'a pas eu le temps d'être exploitée.

En connaissant maintenant les temps d'exécution des fonctions à l'intérieur de la boucle temporelle de *Transmission_mod*, on peut obtenir la borne supérieure de sa fréquence :

$$F_{\text{transmission_mod,max}} = \frac{1}{\Delta t_{q\dot{q},PDO} + \Delta t_{q\dot{q},ROS} + \Delta t_{T,PDO} + \Delta t_{T,SDO} + \Delta t_{rotparam}} = 316.96 \text{ Hz} \quad (5.1)$$

Tout comme c'était le cas lors de l'utilisation du clavier sur le Raspberry (voir section 4.1), la fréquence de la boucle temporelle de *Transmission_mod* est très importante. En effet, c'est elle qui dicte la fréquence de rafraîchissement du couple dans le volant physique ainsi que la mise à jour de la position et vitesse de ce dernier dans Robotran. Plus cette fréquence est élevée, plus le ressenti haptique est réaliste. La section suivante 5.5 se penche alors sur les résultats obtenus en simulation.

5.4.2 Initialisation et utilisation du servo-variateur

Le noeud *Transmission_mod*, avant de rentrer dans sa boucle itérative, va procéder à une initialisation du bus de terrain CAN et du servo-variateur avec les étapes suivantes à l'aide de KaCanOpen :

1. Activer l'interface de communication CAN en spécifiant un nom pour le bus CAN ainsi qu'une vitesse de transmission maximale de 1 Mbit/s .

8. UART est l'interface utilisé sur la carte *Texas Instrument* (ancien module de transmission) pour utiliser la communication en série avec l'ordinateur.

9. C'était aussi la réception des positions et vitesses sur l'ordinateur via le protocole en série qui était la plus chronophage, ce qui également amélioré ici grâce à ROS

2. Identifier le servo-variateur (se trouvant à l'adresse ID 32) et le placer dans une variable de KaCanOpen.
3. Charger le fichier EDS du servo-variateur dans la variable de ce dernier.
4. Régler les paramètres de sécurité : vitesse maximale du volant de $20rad/s$ et rotation maximale de 3 tours à gauche et à droite. Ces valeurs permettent l'empêchement d'un "emballement" du moteur tout en n'entravant pas les positions et vitesses usuelles d'un volant de voiture.
5. Activer le réseau du servo-variateur avec le NMT.
6. Activer l'étage de puissance (i.e. le moteur).
7. Procéder à une prise d'origine immédiate nécessaire pour mesurer la position courante du volant à l'aide du codeur incrémental.
8. Passe en mode "profil de couple" : ce mode permet d'envoyer des consignes de couple au moteur.

Lorsque l'initialisation est finie, le noeud ROS passe dans la boucle temporelle présentée ci-dessus qui permet le pilotage du modèle Robotran à l'aide du volant.

A la fin de la simulation, l'utilisateur a la possibilité d'en recommencer une nouvelle. Si une nouvelle simulation est lancée, il faut procéder à un arrêt du mode "profil de couple" qui coupe l'étage de puissance et à une nouvelle prise d'origine pour avoir à nouveau le volant en position nulle. Le noeud ROS revient alors à l'étape 5. Si non, il désactive entièrement l'étage de puissance et le réseau et clotûre son utilisation.

5.5 Expérimentations et résultats

Cette section présente les résultats obtenus lors d'essais en laboratoire sur le simulateur à retour haptique. La première partie valide l'utilisation d'un logiciel ROS-Robotran afin de rendre à nouveau fonctionnel le simulateur en se basant sur l'analyse des fréquences et vitesses de communication des topics ainsi que sur la qualité des signaux de données envoyées et reçues. La deuxième partie présente deux tests réalisés avec la voiture en faisant varier sa chasse afin de présenter une utilisation possible d'un tel simulateur.

5.5.1 Validation

Les mesures de fréquence sont réalisées avec la voiture et le kart sur des sections de circuit dynamiques avec différents types de virage afin de valider l'utilisation du simulateur en toute situation. L'analyse se base principalement sur les données obtenues avec la voiture qui s'est avérée être un modèle mieux adapté au simulateur. Certaines données sur le kart sont présentées afin de comparer le deux modèles au niveau de la complexité du système de direction. Le pas de temps utilisé pour la dynamique directe de la voiture reste inchangé par apport à l'exemple de la section 4.1 : $\Delta t_{Rob,car} = 0.82ms$. Celui pour le kart a une valeur nettement plus faible grâce à la complexité et le nombre de corps réduits du modèle : $\Delta t_{Rob,kart} = 0.25ms$. La tableau suivant reprend les valeurs moyennes des fréquences (F) des topics¹⁰ obtenues à la fin d'une simulation de 30 secondes ainsi que la valeur moyenne de leur vitesse de communication (V) pour les deux véhicules :

10. T est l'abréviation correspondant au topic *chatter_torque* et *q̇q* pour le topic *chatter_pos_vit*.

Voiture			
$F_{ROS,T,car}$ [Hz]	$V_{ROS,T,car}$ [kB/s]	$F_{ROS,q\dot{q},car}$ [Hz]	$V_{ROS,q\dot{q},car}$ [kB/s]
1218.37	9.90	204.63	3.32
Kart			
$F_{ROS,T,kart}$ [Hz]	$V_{ROS,T,kart}$ [kB/s]	$F_{ROS,q\dot{q},kart}$ [Hz]	$V_{ROS,q\dot{q},kart}$ [kB/s]
3978.58	26.38	204.76	3.32

TABLE 5.7 – Fréquence et vitesse de communication des topics

Ce tableau permet de tirer de nombreuses conclusions quant aux performances du simulateur, celles-ci sont développées dans les sections suivantes.

Au niveau de la vitesse de transmission, une hypothèse serait que celle-ci s'adapte afin de respecter la fréquence de publication et on observe alors une vitesse de transmission proportionnellement plus élevée pour le kart sur le topic *chatter_torque*. La vitesse de communication sur le topic *chatter_pos_vit* est, elle, plus faible car elle transporte plus de données.

Envoi de la position et vitesse au module multicorps

Fréquence et vitesse d'envoi Comme on peut le voir avec les données obtenues et comme il a été expliqué ci-dessus, c'est la fréquence d'envoi de la position et vitesse du volant au module multicorps qui impose la fréquence de rafraîchissement générale du simulateur. En effet, Robotran et son noeud ROS intégré ne présente quasi aucun retard quant à l'envoi du couple au module de transmission comme il est démontré ci-dessous. Le ralentissement s'observe dans le module de transmission qui accuse une fréquence moyenne pour sa boucle temporelle de $\approx 204Hz$ pour les deux modèles de véhicule. C'est donc à cette fréquence que le couple est envoyé au servo-variateur et que ce dernier renvoie la position et vitesse du volant au module de transmission qui lui même les renvoie au module multicorps. Sa valeur se trouve bien en dessous de la borne maximale mesurée avec la vitesse des d'exécution des fonctions d'envoi et de réception ($F_{transmission_{mod,max}} = 316Hz$). Une perte de temps est sans doute à déplorer lors du passage d'une itération à une autre dans la boucle *while*.

En connaissant la fréquence à laquelle tourne la boucle temporelle de la dynamique directe de Robotran pour respecter le facteur de temps réel imposé par le pas de temps $\Delta t_{Rob,car} \rightarrow F_{Rob,car} = 1219.51Hz$ ou $\Delta t_{Rob,kart} \rightarrow F_{Rob,kart} = 4000Hz$ ainsi que la fréquence à laquelle la position et vitesse du volant sont envoyées au module multicorps ($F_{ROS,q\dot{q},car}$ ou $F_{ROS,q\dot{q},kart}$), on peut connaître le taux de rafraîchissement de la position et vitesse du volant dans le module multicorps (Robotran) :

$$f_{T,mod_Robotran,car} = \frac{F_{ROS,q\dot{q},car}}{F_{Rob,car}} * 100 = 16.78\% \quad (5.2)$$

$$f_{T,mod_Robotran,kart} = \frac{F_{ROS,q\dot{q},kart}}{F_{Rob,kart}} * 100 = 5.12\% \quad (5.3)$$

Ce résultat s'illustre également avec les graphiques suivants affichant de manière ponctuelle (avec les marqueurs rouges) les données sur la position et vitesses du volant mesurées par le servo-variateur et envoyées sur le topic ROS et les données en position et vitesse utilisées par Robotran pour sa résolution des équations du mouvement pour la voiture :

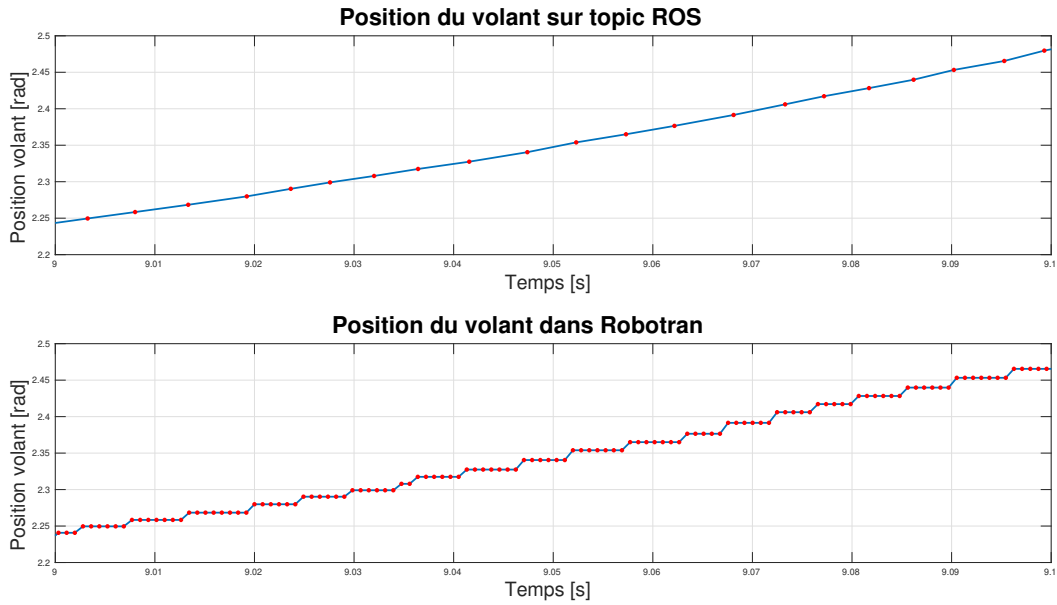


FIGURE 5.12 – Données sur la position du volant pour la voiture

Les données de Robotran prennent une allure "en escalier" car le nombre d'informations envoyées sur le topic *chatter_pos_vit* est insuffisant. Lorsque la fonction *Callback* du noeud *Rosbotran* vérifie si une nouvelle valeur a été publiée sur le topic :

- en moyenne 1 valeur sur 6 sera mise à jour pour la voiture.
- en moyenne 1 valeur sur 20 sera mise à jour pour le kart.

Robotran garde alors la dernière valeur mesurée. L'amplitude des sauts d'une valeur à une autre reste toute fois assez faible en étant inférieur à $1^\circ = 0.017rad$.

Les mêmes observations se font pour la vitesse du volant :

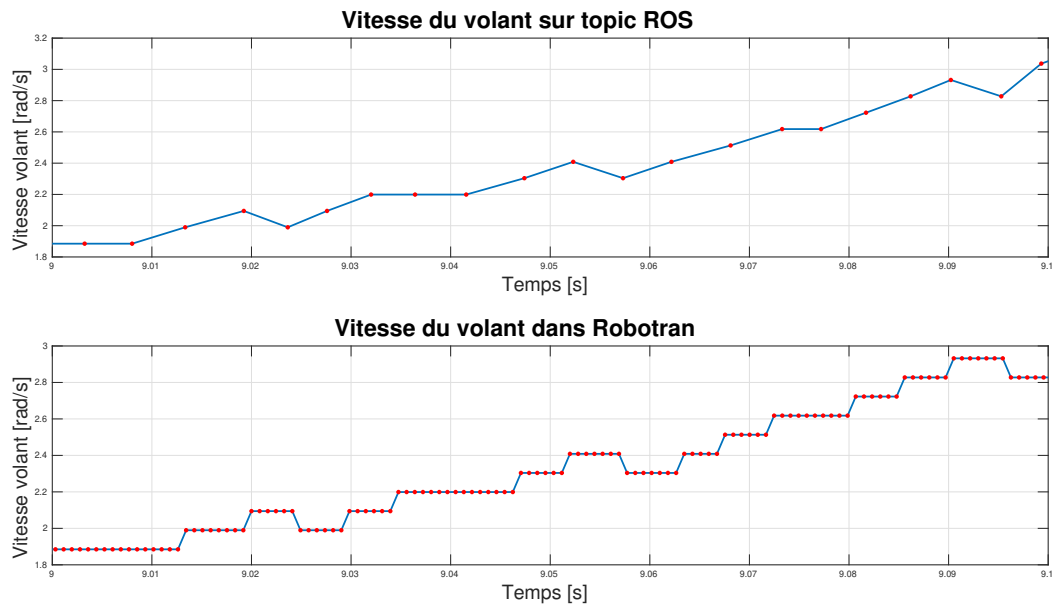


FIGURE 5.13 – Données sur la vitesse du volant pour la voiture

Les sauts d'une valeur à une autre sont ici d'une plus large amplitude avec des impulsions de $+/- \approx 5.73^\circ/s = 0.1\text{rad/s}$ comme on peut l'observer sur les figures 5.15 et 5.16.

Ressentis en simulation et observations Malgré les valeurs pessimistes obtenues ci-dessus sur la mise à jour de la position et vitesse du volant, les ressentis et observations suivantes démontrent d'assez bonnes performances pour le simulateur :

- La fréquence de rafraîchissement générale de la position, vitesse et couple du volant est de $\approx 204\text{Hz}$. Cette valeur, bien que fortement inférieure à celle obtenue par les anciens mémorants qui expliquaient avoir pu atteindre 500Hz , reste tout de même une valeur acceptable. Elle offre un ressenti haptique correcte (avec une bande passante entre 20 et 30Hz pour la perception kinesthésique et entre 0 et 400Hz pour la perception tactile) et reste dans la gamme des fréquence de simulateur de conduite commerciaux estimée entre 150 et 300Hz .
- La fréquence de rafraîchissement est indépendante de la complexité du modèle multicorps. Cette observation implique qu'aucune amélioration n'est nécessaire quant à la vitesse de résolution des équations de Robotran tant que le pas de temps reste suffisamment petit que pour avoir une fréquence de résolution des équations plus élevée que la fréquence de rafraîchissement. Toutes les améliorations possibles qui sont présentées à la section 5.6 portent sur des modifications à apportées au software du module de transmission. Cette indépendance complète de Robotran vis à vis du simulateur permettrait l'utilisation de modèles de véhicules complexes sans porter atteinte à la fréquence de rafraîchissement.
- La faible taux de mise à jour de la position et vitesse du volant ne porte pas atteinte au réalisme de la simulation. Il permet de garder un contrôle précis et réactif sur le position et vitesse du volant imposées par l'utilisateur.

Allure des courbes de données Les deux figures suivantes, qui affichent les mêmes données que les deux précédentes sur une plus grande échelle, montrent alors un quasi parfaite similitude entres les données imposées par l'utilisateur sur le topic ROS *chatter_pos_vit* et les données utilisées par Robotran.

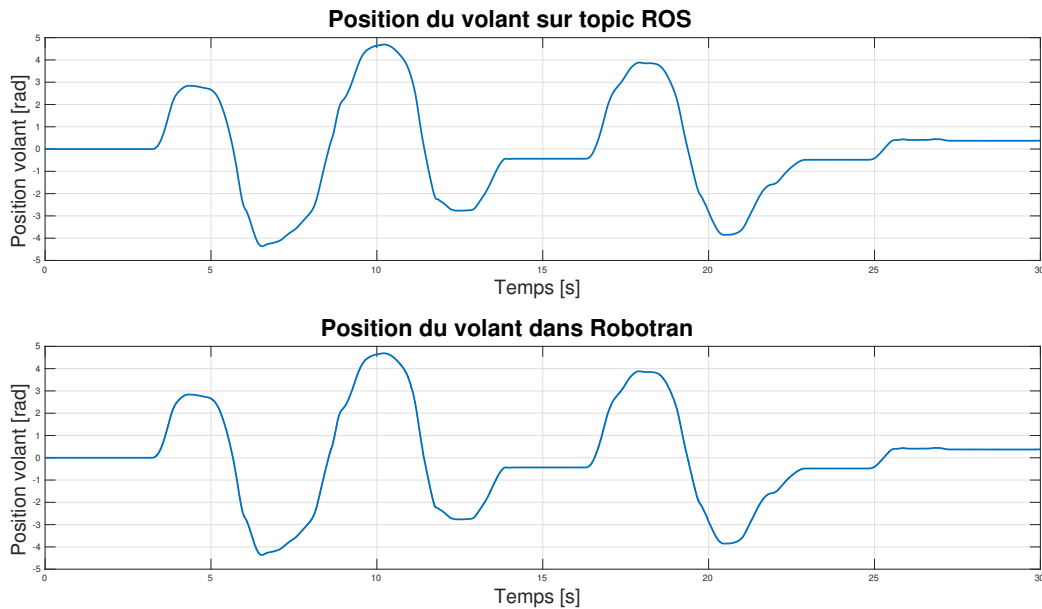


FIGURE 5.14 – Position du volant pour la voiture

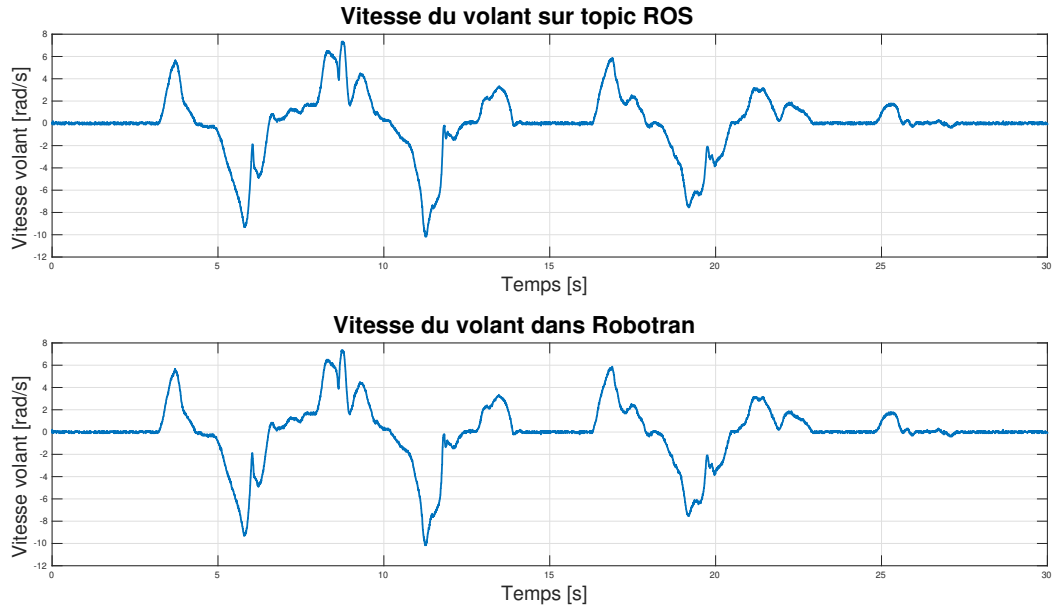


FIGURE 5.15 – Vitesse du volant pour la voiture

Les figures 5.15 et 5.16 permettent de mettre en lumière l’origine des impulsions au niveau de la vitesse mais surtout d’en observer leur impact sur la conduite du véhicule. Elles proviendraient du servo-variateur qui calcule la vitesse du volant en fonction des positions mesurées par le codeur incrémental. Il n’est donc, à priori, pas possible d’atténuer ce défaut si ce n’est qu’avec l’utilisation d’un filtre basse-bas et anti-impulsion comme cela avait été fait lors du second mémoire¹¹ et qui sont donc proposés comme améliorations possibles. Leur impact est toutefois moins important sur le voiture que le kart. En effet, ce dernier possède un ratio entre l’angle de rotation du volant et des roues proche de l’unité (contre un ratio valant en moyenne 18 : 1 pour la voiture). La direction du kart est donc beaucoup plus sensible et nécessite de plus faibles mouvements du volant à plus faible vitesse pour diriger le véhicule. Ces impulsions de $\pm 0.1 \text{ rad/s}$ sont donc proportionnellement plus fortes par rapport à la vitesse du volant pour le kart que pour la voiture. La répercussion de ces impulsions se ressent dans le couple du volant et est détaillée à la section suivante.

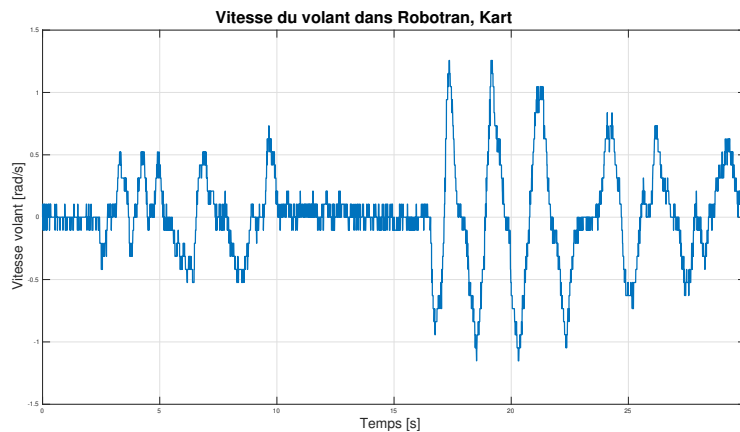


FIGURE 5.16 – Vitesse du volant pour le kart

11. Les filtres n’étaient pas actifs lors de la prise des mesures pour le kart dans les graphiques présentés ici.

Envoi du couple au module de transmission

Fréquence et vitesse d'envoi Tout comme cela a été fait avec la position et vitesse dans le module multicorps, on peut connaître le taux de rafraîchissement du couple dans le module de transmission :

$$f_{T,mod_trans,car} = \frac{F_{ROS,T,car}}{F_{Rob,T,car}} * 100 = 99.91\% \quad (5.4)$$

$$f_{T,mod_trans,kart} = \frac{F_{ROS,T,kart}}{F_{Rob,T,kart}} * 100 = 99.46\% \quad (5.5)$$

Ces valeurs proches de 100 % permettent d'affirmer que la quasi totalité des couples calculés par Robotran sont envoyés au module de transmission et ce, même pour des fréquences très élevées de 4000 Hz. Ce résultat s'illustre également avec les graphiques suivants affichant de manière ponctuelle (avec les marqueurs rouges) les données de couple calculées par Robotran et envoyées¹² par ROS :

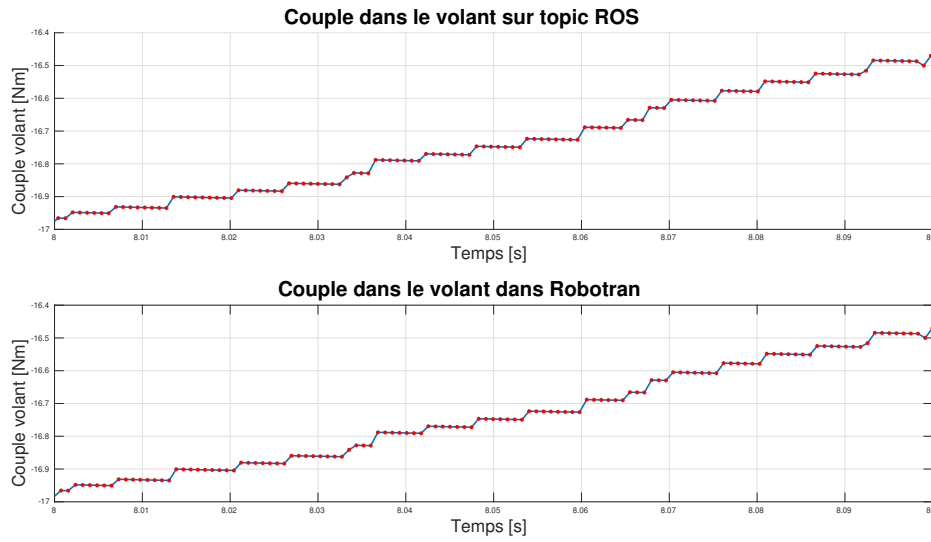


FIGURE 5.17 – Données sur le couple pour la voiture

L'effet "escalier" sur la position et vitesse du volant se répercute sur le couple calculé par Robotran qui présente alors également des valeurs augmentant et diminuant par saut maximal de $0.05Nm$.

Ressentis en simulation et observation Avec un modèle de direction complet et complexe comme pour la voiture, aucun tremblements parasites n'est ressentis. Le kart, lui, accuse un léger tremblement présent sur les graphiques (voir figure 5.19 et 5.21) mais quasi imperceptible lors de la conduite. De plus, ce tremblement reste de très faible amplitude et permet de garder un auto-alignement stable de la colonne de direction et de ne pas diminuer le confort de conduite.

Le couple de retour dans le volant de la voiture reste toutefois plus réaliste. Les variations d'angle de chasse permettent d'augmenter et diminuer sa valeur. De plus, on observe facilement une augmentation du couple lors que le véhicule roule à plus haute vitesse (phénomène physique

12. Pour rappel, le couple est envoyé sur les topics sous forme de consigne en % pour le servo-variateur. Il a toute fois été converti en Nm pour tous les graphiques suivants afin de garder des données comparables.

expliqué à la section 1.3.1) ce qui est difficilement observable avec le kart possédant une gamme de vitesse réduite afin de garder le contrôle de la direction très sensible¹³. Enfin le couple de retour généré dans la colonne de direction de la voiture permet un auto-alignement rapide et stable de celle-ci.

Enfin, il faut préciser que le couple calculé par Robotran pour le modèle du kart était à la base très élevé. De plus, avec la sensibilité de la direction, celui-ci augmentait très rapidement, rendant la conduite inconfortable voir dangereuse. Tout comme les anciens mémorants l'avaient fait, la valeur du couple est alors divisée par 3 et limitée à $\pm 10 Nm$ ¹⁴. Au niveau de la voiture, bien que les valeurs de couples soient plus élevées, celles-ci augmentent plus lentement. De plus, les valeurs obtenues correspondent à la gamme de celles d'un système de direction réelle sans direction assistée.

Allure des courbes de données A nouveau, il est possible d'observer l'ensemble des valeurs du couple de retour sur l'entièreté de la simulation :

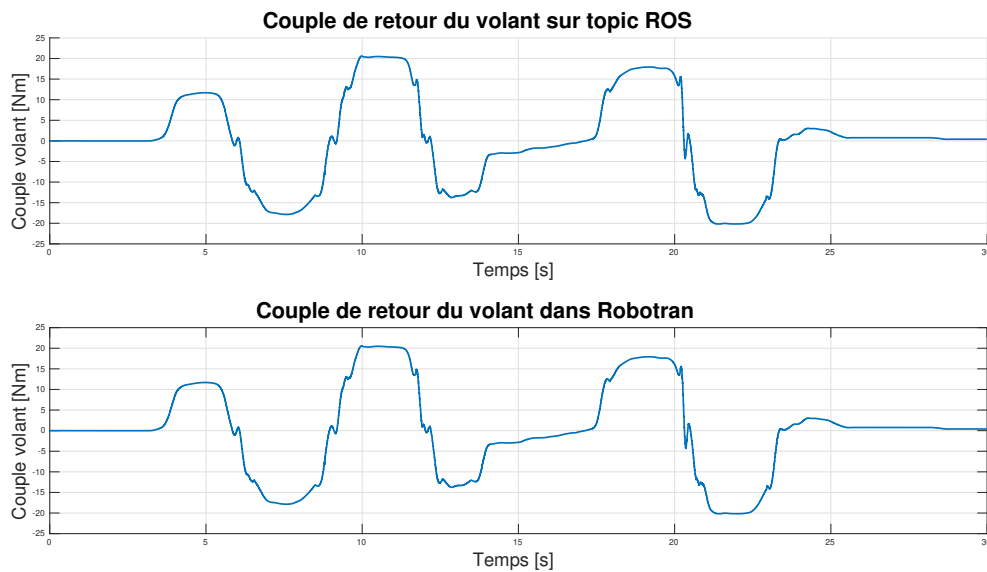


FIGURE 5.18 – Coupe dans le volant pour la voiture

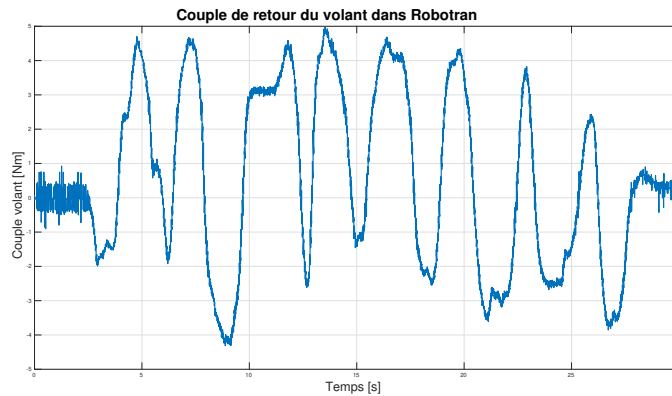


FIGURE 5.19 – Couple dans le volant pour le kart

13. La vitesse maximale conseillée est 30 km/h. La voiture, elle, est pilotable à des vitesses testées jusque 150 km/h.

14. Les valeurs présentées dans les graphes sont les valeurs ayant subies ces modifications.

En restreignant les données du couple de retour au 7.5 premières secondes de simulation, on observe bien ce tremblement parasite du volant du kart provenant lui-même des impulsions sur la vitesse du volant. Ce tremblement est également présent lorsque le volant est en position nulle (correspondant aux 2.5 premières secondes de simulation) mais ne le rend toutefois pas instable. Au niveau de la voiture qui présente la même échelle de graphique que le kart, on peut valider l'absence de toutes oscillations parasites.

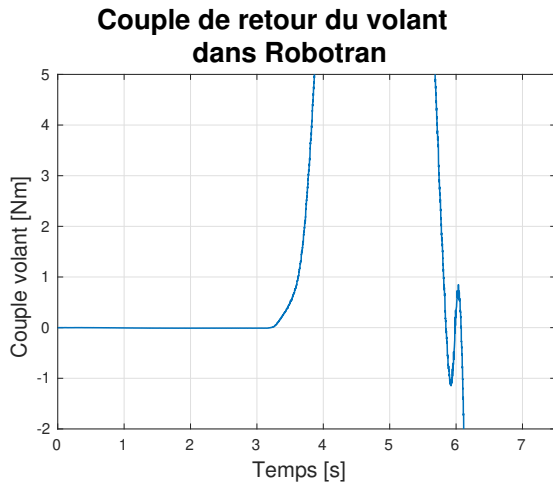


FIGURE 5.20 – Zoom du couple dans le volant pour la voiture

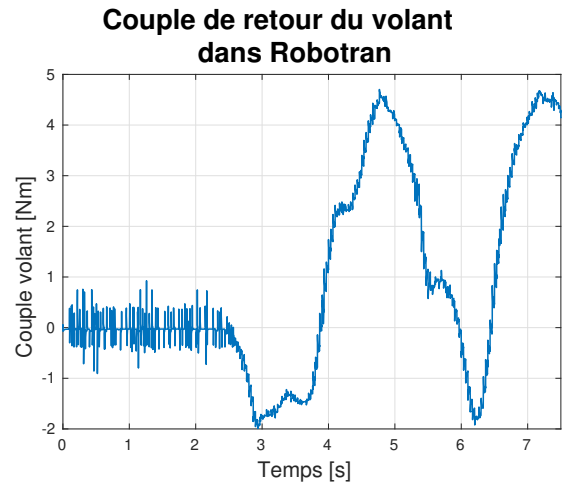


FIGURE 5.21 – Zoom du couple dans le volant pour le kart

5.5.2 Application sur des cas simples

Maintenant que l'utilisation du simulateur avec ROS-Robotran a été validée, il est possible de réaliser différentes manœuvres dynamiques afin d'en étudier l'impact sur le comportement du retour de force dans la direction et les conséquences que cela apporte au conducteur¹⁵. L'utilisateur peut facilement passer d'une simulation à une autre en ayant la possibilité de changer de véhicule (voiture ou kart). L'ensemble des démarches à réaliser sont résumées dans un tutoriel expliquant les différentes étapes pour l'installation et l'utilisation du simulateur (voir annexe B).

Les simulations disponibles sont les suivantes :

- Rouler avec le kart et la voiture sur deux circuits réalisés respectivement par le premier et second mémoire à une vitesse constante choisie au préalable.
- La prise d'un virage à pleine vitesse avec la voiture : cela permet d'observer l'augmentation du couple de retour avec la traînée mécanique et la vitesse du véhicule.
- La prise d'un rond-point avec la voiture : à un moment aléatoire la sol devient glissant pendant 2 secondes. L'utilisateur doit alors tenter de reprendre le contrôle du véhicule.
- L'esquive d'un obstacle placé à une distance aléatoire sur une longue ligne droite.

Les trois derniers tests se font avec la voiture ayant une traînée mécanique aléatoire comprise entre 34.04 et 47.02 mm. Nous nous intéressons ici aux résultats des deux derniers tests.

¹⁵. Le faible nombre de tests réalisés avec un seul conducteur compte tenu de la crise sanitaire engendrée par le COVID-19 ne permet pas une étude complète de l'interaction humain-véhicule. Les résultats et appréciations obtenus sont néanmoins présentés afin de valider une utilisation complète du simulateur.

Prise d'un rond-point

Le test est réalisé avec les paramètres suivants :

- La simulation se fait à vitesse constante. Une comparaison entre 35 km/h et 40 km/h est faite.
- Le rond-point présente un rayon de courbure de 16.5m.
- L'arrivée sur le rond point via une bande de lancement et le temps pour placer le véhicule en virage constant prend au maximum 12s. Le moment où la route devient glissante se produit entre le temps $t = 12s$ et $t = 22s$ et dure 2s.
- La route glissante est simulée en divisant par deux les forces latérales, longitudinales et le moment de force d'auto-alignement du contact pneu-sol.
- L'écran affiche une vue extérieure de l'arrière du véhicule.

Simulation à une vitesse de 35 km/h : chasse mécanique de 42.02 mm Pour cette simulation, le sol glissant débute au temps $t = 13s$. On observe une chute soudaine du couple de retour en valeur absolue (voir figure 5.22) due à la diminution des forces latérales et moments de force d'auto-alignement du contact pneu-sol. Le conducteur peut difficilement maintenir la trajectoire de son véhicule et ce dernier commence à dévier. Le conducteur augmente tout de même par réflexe l'angle de braquage du volant pour palier cette déviation. Au temps $t = 15s$, le véhicule revient sur sol normal, le couple réaugmente brusquement avant que le conducteur ne stabilise le volant. La figure 5.23 montre cette déviation du véhicule qui finit par revenir sur sa trajectoire initiale.

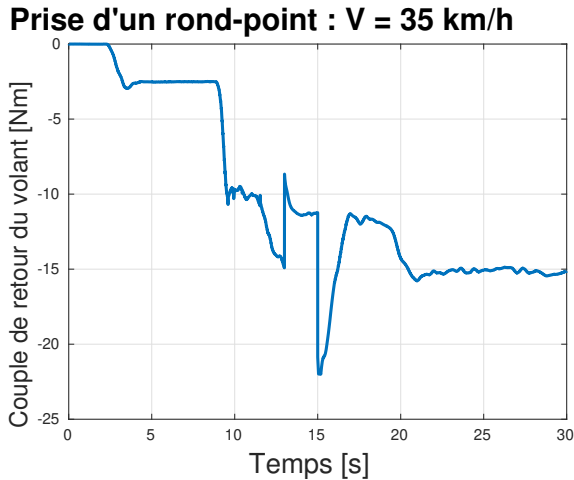


FIGURE 5.22 – Couple de retour du volant : chasse mécanique de 42.02 mm

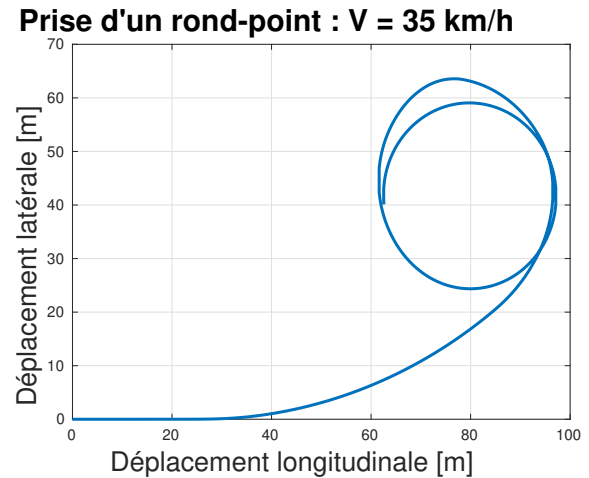


FIGURE 5.23 – Trajectoire du véhicule : chasse mécanique de 42.02 mm

Simulation à une vitesse de 40 km/h : chasse mécanique de 38.02 mm Cette simulation permet d'observer qu'une augmentation de 5 km/h de la vitesse du véhicule modifie tout à fait le comportement dynamique de celui-ci. Bien que la chasse ait diminuée, le couple présent dans le volant pour maintenir le véhicule en courbe constante (au temps $t \approx 11s$ pour les deux simulations) est multiplié par deux (voir figure 5.24). Cela montre l'importance du terme de la force centripète augmentant quadratiquement avec la vitesse dans la formule du moment d'auto-alignement du volant (voir équation 1.9).

A cette vitesse, il n'est pas possible de reprendre la trajectoire initiale du véhicule sans réaction de contre-braquage de la part du conducteur comme le montre la figure 5.25. Le contre-braquage est

une manoeuvre permettant de stabiliser le véhicule lorsque les roues arrières de celui-ci glissent vers l'extérieur du tournant. Ce qui est le cas ici sur sol glissant avec la diminution des forces latérales du contact pneu-sol permettant le maintien de la trajectoire en courbe. Le conducteur doit alors appliquer un braquage du volant dans le sens opposé à celui permettant la prise du virage. Cette manoeuvre est étudiée dans le paragraphe suivant pour deux chasses mécaniques différentes.

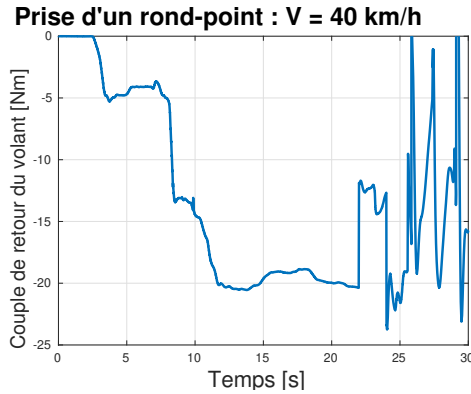


FIGURE 5.24 – Couple de retour du volant :
chasse mécanique de 38.02 mm

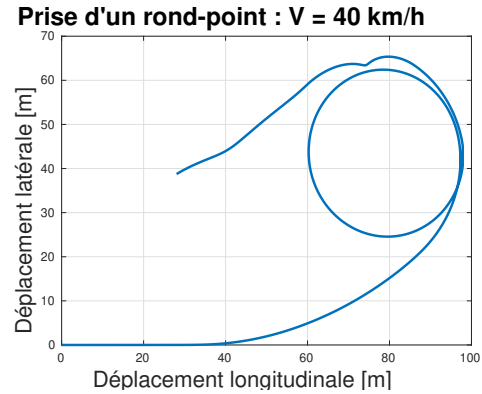


FIGURE 5.25 – Trajectoire du véhicule : chasse
mécanique de 38.02 mm

Simulation à une vitesse de 40 km/h : contre-braquage Comme expliqué ci-dessus et observé sur la figure 5.27, le fait de contre-braquer au moment de la perte d'adhérence permet de rétablir la trajectoire du véhicule. Comme le montre la figure 5.26, l'effet du contre-braquage ne se fait pas directement à cause du manque d'adhérence. Au moment où le véhicule revient sur le sol normal ($t = 17$ pour la chasse de 45.02 mm et $t = 19$ pour la chasse de 36.02 mm) le couple réaugmente brusquement (en valeur absolue) avant de changer de signe étant donné que l'angle de braquage du volant est maintenant opposé.

L'influence de la chasse montre ici que le couple de retour est globalement plus élevé pour une plus grande chasse. Le moment d'auto-alignement de la colonne de direction étant également plus fort, il aide le conducteur à reprendre la trajectoire initiale plus rapidement comme le montre la figure 5.27.

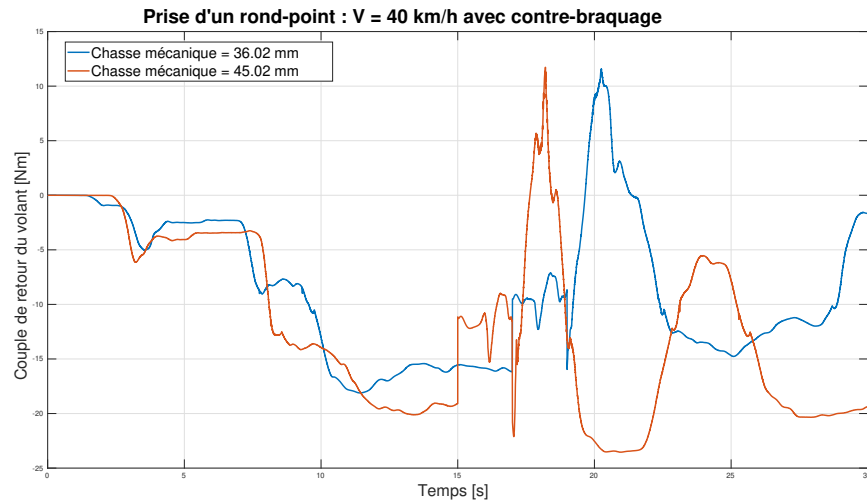


FIGURE 5.26 – Couple de retour du volant : chasse variable et contre-braquage

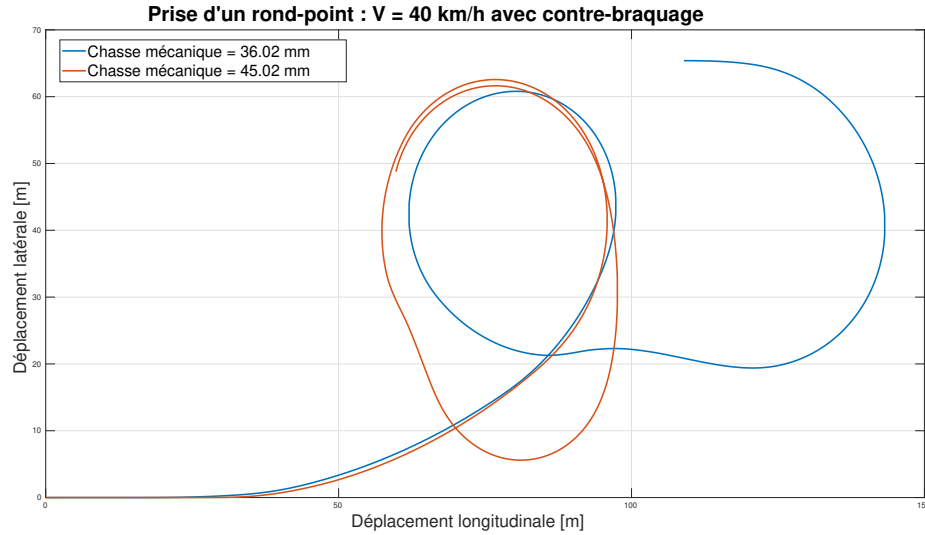


FIGURE 5.27 – Trajectoire du véhicule : chasse variable et contre-braquage

Esquive d'un obstacle

Le test est réalisé avec les paramètres suivants :

- La simulation d'esquive se déroule sur ligne droite à une vitesse constante de 75 km/h.
- L'obstacle de 2.5 mètres de large est placé devant le véhicule à une distance absolue de 150m. Le véhicule démarre la simulation à une distance absolue aléatoire entre 0 et 50 m (l'obstacle se trouve alors devant la voiture à une distance aléatoire entre 100 et 150 m).
- L'écran affiche une vue immersive à l'intérieur du véhicule.

Les trois graphiques suivants affichent la trajectoire du véhicule ainsi que le couple de retour et la position du volant lors de trois simulations sélectionnées parmi plusieurs tests réalisés avec une traînée (ou chasse) mécanique aléatoire non-connue par l'utilisateur.

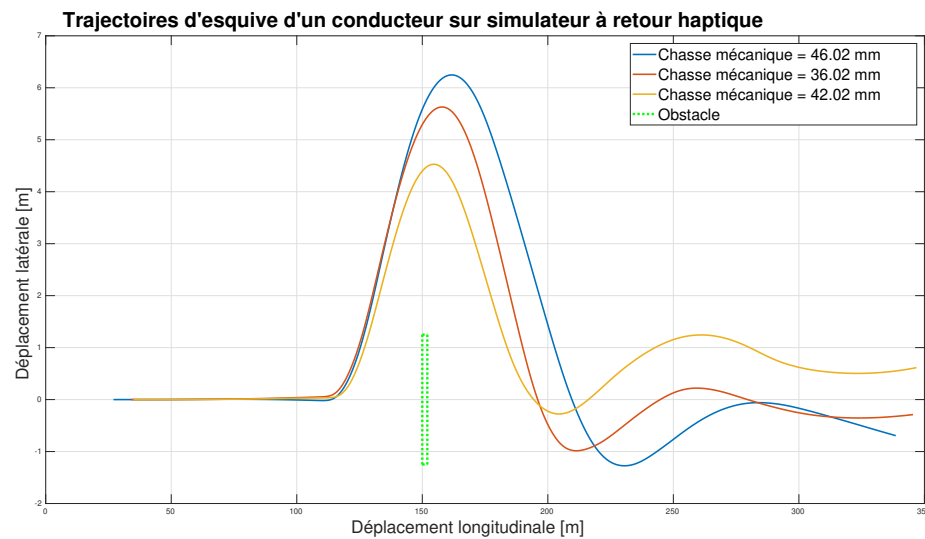


FIGURE 5.28 – Trajectoire du véhicule lors de l'esquive

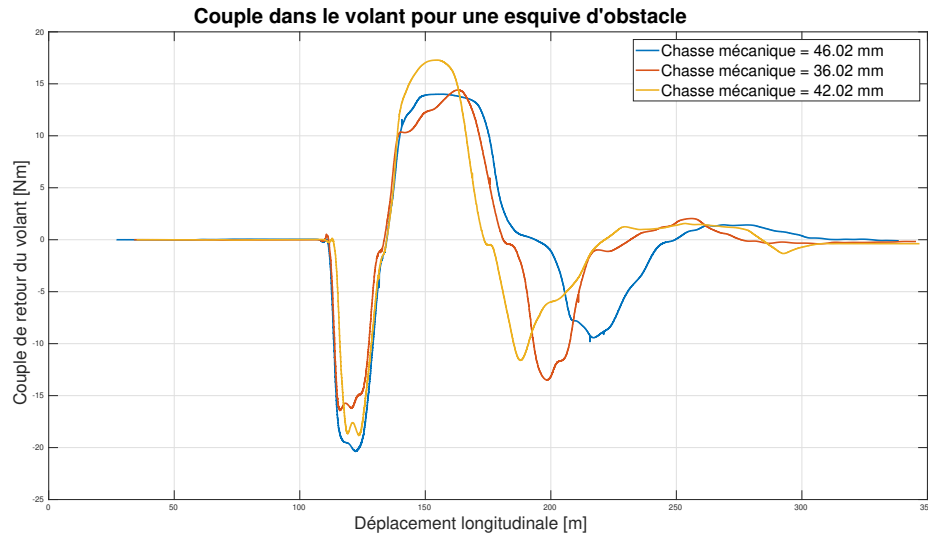


FIGURE 5.29 – Couple de retour du volant lors de l'esquive

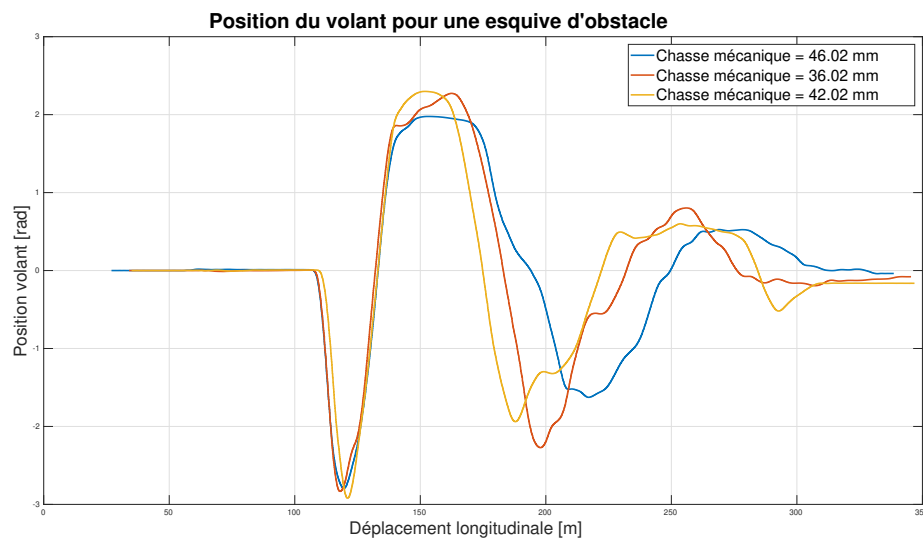


FIGURE 5.30 – Position du volant lors de l'esquive

Globalement l'utilisateur réussit toutes les esquives avec une marge de sécurité car il faut également prendre en compte la demie-largeur du véhicule ($\approx 0.75m$). La figure 5.29 montre bien que pour le premier virage ou l'angle de braquage du volant est sensiblement pareil pour chaque simulation, la chasse augmente le couple de retour. Les paragraphes suivants analysent chacune des esquives séparément.

Simulation avec une chasse mécanique de 46.02 mm Il s'agit du deuxième test mesuré. L'esquive avec une grande chasse offre un meilleur ressenti dans le volant et permet une stabilisation du volant avec des mouvements d'une plus faible amplitude après l'esquive.

Simulation avec une chasse mécanique de 36.02 mm Ce test a été mesuré juste après celui présenté ci-dessus. La figure 5.30 montre que l'utilisateur stabilise le volant plus vite mais avec des

mouvements d'une plus grande amplitude.

Simulation avec une chasse mécanique de 42.02 mm Ce test s'est déroulé après que l'utilisateur ai réalisé 5 simulations consécutives avec une chasse de 36.02 mm. L'augmentation soudaine de la chasse montrerait un lien avec le temps de réaction du conducteur qui doit fournir un effort plus élevé pour tourner le volant. En effet on constate sur la figure 5.30 que le conducteur commence la manoeuvre légèrement plus tard (et ce, même par rapport à de nombreuses autres simulations non-affichées ici). Cette hypothèse pourrait être vérifiée avec de plus nombreux tests ainsi que l'implication de plusieurs conducteurs différents.

5.6 Conclusion et améliorations

La modification de l'hardware au niveau du module de transmission et l'utilisation de ROS ont permis de rendre à nouveau fonctionnel le simulateur de véhicules. Les sensations haptiques lors de la conduite de la voiture offrent un assez bon réalisme et permettent de ressentir les variations dans la géométrie du véhicule et dans sa dynamique.

Néanmoins de nombreuses améliorations n'ont pas eu le temps d'être réalisées avec des tests en laboratoire et des pistes de solutions sont alors proposées dans les sections suivantes. Ces modifications tendent à augmenter la fréquence de rafraîchissement de la position, vitesse et couple du volant ainsi qu'à améliorer le réalisme des simulations.

Module de transmission

Le software du module de transmission est l'élément limitant la fréquence de rafraîchissement des données. Les deux améliorations suivantes devraient améliorer la vitesse et fréquence de transmission d'informations et espérer atteindre ainsi une fréquence de rafraîchissement de $500Hz$.

Rosbridge et commande PDO pour le couple avec KaCanOpen Ces deux défauts ont été présentés dans les problèmes persistants du paquet *KaCanOpen* (voir section 5.3.1). L'utilisateur est invité à se référencer aux documents et exemples du site de *KaCanOpen* [37] ainsi que le manuel de bus de terrain CAN du servo-variateur [35] pour une compréhension complète de l'utilisation de *rosbridge* et du mappage PDO.

Multithreading Tout comme cela avait été fait avec l'ancien module de transmission, l'utilisation de multithreading pour séparer les envois/réceptions via topics des envois/réceptions via CANopen sur deux *threads* différents pourrait augmenter également la vitesse et fréquence de transmissions de données. Il faut tout de même faire attention que l'utilisation de *mutexs* (ou d'autres moyens qui évitent une erreur de segmentation) pour faire passer les données de position, vitesse et couple d'un *thread* à l'autre ne compense pas le gain de vitesse d'exécution gagné.

Module multicorps

Les 2 premiers changements sur le module multicorps visent à améliorer les qualités des signaux de données (position, vitesse et couple du volant) en atténuant l'effet "en escalier" du à la fréquence de rafraîchissement plus faible que la fréquence de la boucle temporelle de Robotran. Il s'agit d'améliorations qui avaient déjà fait leur preuve lors des deux précédents mémoires mais qui n'ont pas été implémentées ici car la qualité des signaux pour la voiture offre déjà la possibilité de réaliser

des simulations sans tremblements parasites dans le volant. Néanmoins elles sont tout de même proposées ici afin d'améliorer la qualité du simulateur. Le troisième changement, lui, concerne une amélioration au niveau visuel.

Extrapolation de la position du volant Cette amélioration permet de combler le manque de données sur la position du volant lorsque *Rosbotran* vérifie si une nouvelle information a été publiée sur le topic *chatter_pos_vit*. Elle consiste à extrapoler la position du volant pour tout les temps t_{i+1} en connaissant la position et vitesse du volant au temps t_i jusqu'à ce que de nouvelles données soient publiées sur le topic.

Filtres basse-bas et anti-impulsions L'utilisation de ces deux filtres permet de filtrer les impulsions observées sur les courbes de la vitesse du volant (figure 5.15 et 5.16) afin d'obtenir une courbe plus lisse. Améliorer le signal d'entrée du module multicorps a comme conséquence directe une amélioration du signal de sortie, le couple de retour dans le volant.

Interface graphique La visualisation des simulations de Robotran est gérée par l'interface de programmation *OpenGL*. Il permet d'avoir une visualisation 3D du modèle multicorps et de son environnement (routes, arbres, autres décors, ...) à l'aide fichier 3D *.wrl*. Une simulation réalisée avec un véhicule et un décors détaillés améliore la qualité de la simulation au niveau visuel et immersif. Cependant, l'utilisation de fichiers 3D complexes risque de ralentir la vitesse du noeud *Rosbotran* en surchargeant les processeurs et la carte graphique de l'ordinateur. ROS devrait permettre de communiquer les positions des articulations du multicorps nécessaire pour *OpenGL* sur un autre dispositif hardware (en augmentant le pas de temps de Robotran comme cela avait été fait pour l'exemple d'esquive d'une voiture avec sonars (voir section 4.2.3)). Ce dispositif pourrait alors gérer de manière indépendante la visualisation 3D de la simulation. L'utilisation d'un grand écran (voir même l'ajout de deux écrans sur les côtés) augmenterait encore ce caractère immersif et réaliste.

Amélioration de l'interaction pilote-véhicule

L'implémentation de ROS dans Robotran ouvre maintenant la porte à l'utilisation de nombreux capteurs et actionneurs pour interagir avec le modèle multicorps en temps-réel. La possibilité d'utiliser un protocole de communication CANopen (qui, pour rappel, est fortement utilisé pour la communication des capteurs et actionneurs d'un véhicule) a également été prouvée. Les modifications possibles sont donc nombreuses : implémentation d'une boîte de vitesse, liaison du simulateur au concept de voiture autonome avec l'utilisation de sonars, conception d'un système inertiel pour le siège du simulateur afin de ressentir les effets de roulis et tangage du véhicule¹⁶, etc. La modification la plus concrète et la plus judicieuse est développée ci-dessous (l'idée avait également été introduite dans les deux mémoires précédents).

Pédalier La structure actuelle du simulateur possède déjà deux pédales situées aux pieds du conducteur et munies de ressorts afin de les ramener à leur position initiale après avoir exercé une pression sur ces dernières. Deux solutions sont proposées afin de faire parvenir le positionnement des pédales à Robotran qui pourra exploiter ces informations pour appliquer un couple de freinage ou d'accélération aux roues du véhicule :

- l'utilisation de capteurs lasers ou sonars placés face l'arrière des pédales pour mesurer la diminution de la distance lorsque les pédales sont enfoncées (méthode peu précise mais plus facilement implémentable).

16. Amélioration également proposée lors du second mémoire.

- l'utilisation d'un capteur de position angulaire placé dans l'articulation rotoïde des pédales pour mesurer leur pivotement autour de leur axe de rotation (méthode plus précise mais nécessitant une modification de l'assemblage mécanique).

Conclusion

Au terme de ce mémoire, l'objectif initial a été atteint, à savoir, généraliser l'intégration de Robotran via ROS dans un système embarqué. Cette intégration, sous forme de tutoriels et de paquets ROS, est maintenant disponible en *open-source* et présente une utilisation facile et rapide.

ROS-Robotran a également montré son utilité et ses performances dans la création de jumeaux numériques. Il ouvre maintenant la porte à de nombreuses applications et autres travaux basés sur ce principe.

Il a notamment été possible de simuler un véhicule autonome à l'aide de capteurs lié à Robotran lui permettant de réaliser des esquives d'obstacles placés devant lui en effectuant une prédiction des trajectoires. Cet exemple a introduit l'intérêt d'un tel système embarqué dans un véhicule réel.

Le second exemple marquant de l'utilisation de ROS-Robotran a été son intégration dans un simulateur de véhicules à retour haptique. Les résultats obtenus permettent de ressentir un retour de force dans le volant donnant une expérience réaliste de la conduite d'un véhicule et la possibilité d'en étudier l'interaction avec le conducteur. Mais surtout, ROS offre la capacité de tester facilement divers véhicules et d'améliorer le simulateur avec une possible augmentation de la vitesse de communication et l'ajout de nouveaux actionneurs et capteurs tel un pédalier.

Finalement, ce mémoire a permis l'exploitation et la compréhension de nombreux concepts mécaniques, informatiques et électroniques, ce qui entrent parfaitement dans le cadre d'un travail de fin d'étude en ingénierie civil.

Bibliographie

- [1] Wikipédia. *Jumeau numérique*. https://fr.wikipedia.org/wiki/Jumeau_numérique, mai 2020.
- [2] Nicola De Lellis and Quentin Materne. *Conception et réalisation d'un simulateur à retour haptique pour l'étude de l'interaction homme/véhicule*. Master's thesis, Ecole Polytechnique de Louvain-la-Neuve, 2015-2016.
- [3] Poletti Romain. *Simulateur à retour haptique de véhicules basé sur des modèles multicorps temps réels*. Master's thesis, Ecole Polytechnique de Louvain-la-Neuve, 2016-2017.
- [4] Robotran. *Modeling Multibody Systems with ROBOTRAN*. https://www.robotran.be/images/files/Robotran_basics.pdf, Avril 2020.
- [5] Pr. Paul Fisette. *LMECA2215 : Dynamique des véhicules*.
- [6] Martin Morel and Théo Tuerlinckx. *Etude de la direction pour un véhicule routier*. Décembre 2019.
- [7] RMS Auto Repair and Tires. *Alignement*. <https://www.rmsautorepair.com/alignment>, juin 2020.
- [8] Suspension designer. *KPI – Kingpin angle, scrub radius wheel centre offset*. <https://www.suspensiondesigner.com/kpi-kingpin-angle-scrub-radius-wheel-centre-offset/>, juin 2020.
- [9] Andrew Heathershaw. *Optimizing Variable Ratio Steering for Improved On-Centre Sensitivity and Cornering Control*. 2000.
- [10] Aflah Elouneg. *Conception et étude de fabrication du système de direction*. 2016.
- [11] Iqbal Junaid, Muhammad Zuhair Khalil, Han Changsoo, Manan Khan Abdul, and Ashfaq Ali Mian. *Adaptive Global Fast Sliding Mode Control for Steer-by-Wire System Road Vehicles*. Juillet 2017.
- [12] Wikipédia. *Steering*. <https://en.wikipedia.org/wiki/Steering>, Juin 2020.
- [13] Olivier Stasse. *Robot Operating System*. pages 9–49, 2016.
- [14] ROS. *Powering the world's robots*. <https://www.ros.org/>, Septembre 2019.
- [15] Wikipédia. *Bus informatique*. https://fr.wikipedia.org/wiki/Bus_informatique, mai 2020.
- [16] ROS. *ROS Answers*. <https://answers.ros.org/questions/>, Mars 2020.
- [17] ROS. *ROS Documentation*. <http://wiki.ros.org/>, Mars 2020.
- [18] ROS. *2019 ROS Metrics Report*. <https://discourse.ros.org/t/2019-ros-metrics-report/10938>, Mars 2020.
- [19] Tuerlinckx Théo. *ros_rob*. https://git.immc.ucl.ac.be/tuerlinckxt/ros_rob, Juin 2020.

- [20] Supinfo. *Introduction au multithreading*. <https://www.supinfo.com/articles/single/67-introduction-au-multithreading/>, Mars 2020.
- [21] Developpez.com. *Initiation à la programmation multitâche en C avec Pthreads*. <https://franckh.developpez.com/tutoriels/posix/pthreads/#LVII>, Avril 2020.
- [22] Raspberry. *Raspberry Pi 2 Model B*. <https://www.raspberrypi.org/products/raspberry-pi-2-model-b/>, Mars 2020.
- [23] LRSE. *ros-keyboard*. <https://github.com/lrse/ros-keyboard>, Mars 2020.
- [24] Freva. *Capteur ultrason HC-SR04 sur Raspberry Pi*. <https://www.freva.com/fr/2019/05/20/capteur-ultrason-hc-sr04-sur-raspberry-pi/>, Avril 2020.
- [25] Freva. *Kit de découverte GPIO pour Raspberry Pi*. <https://www.freva.com/fr/product/kit-de-decouverte-gpio-pour-raspberry-pi/>, Avril 2020.
- [26] Donkey Car. *Donkey Car*. <https://www.donkeycar.com/>, Avril 2020.
- [27] Tiziona Fiorenzani. *Git Hub, ros_tutorials*. https://github.com/tizianofiorenzani/ros_tutorials, Avril 2020.
- [28] Code de la route. *Arrêté royal portant règlement général sur les conditions techniques auxquelles doivent répondre les véhicules automobiles et leurs remorques, leurs éléments ainsi que les accessoires de sécurité*. <https://www.code-de-la-route.be/textes-legaux/sections/ar/reglement-technique-des-vehicules/328-art46-v15-328>, Avril 2020.
- [29] MMA. *Adapter sa vitesse : le temps de réaction*. <https://www.zerotracas.mma/news/4536/adapter-sa-vitesse-le-temps-de-reaction.htm>, Avril 2020.
- [30] Jurifiable. *Quelles sont les principales causes des accidents de la route ?* <https://www.jurifiable.com/conseil-juridique/droit-routier/causes-accidents-de-la-route>, Avril 2020.
- [31] Bertrand Tornil. *Le système haptique côté perception : la somesthésie*. <https://tornil.me/these/007-le-systeme-haptique-cote-perception-la-somesthesie.html>, Mai 2020.
- [32] SK Pang electronics. *PiCAN 2 datasheet*. Mars 2020.
- [33] Schneider Electric. *Lexium 32A Servo variateur Manuel*. Septembre 2017.
- [34] Schneider Electric. *BMH Servo motor Motor manual*. https://download.schneider-electric.com/files?p_Reference=0198441113749-EN&p_EnDocType=User%20guide&p_File_Id=7350052010&p_File_Name=BMH_Manual_V2.1_EN.pdf, 2016.
- [35] Schneider Electric. *LXM32A Interface CANopen Manuel bus de terrain*. Avril 2016.
- [36] Julien Mintenbeck Thomas Keh, Andreas Bihlmaier and Heinz Wörn. *KaCanOpen : An Open PlugPlay CANopen Stack for ROS in Modern C++*. In *Proceedings of the 47th International Symposium on Robotics (ISR)*, 2016.
- [37] KITmedical. *KaCanOpen*. <https://kitmedical.github.io/kacanopen/>, mars 2020.
- [38] Pinterest. *Adept RFID Starter Kit for Raspberry Pi 3,2 Model B/B+ with 40-Pin GPIO Board*. <https://www.pinterest.fr/pin/463096774165070046/?autologin=true>, Juin 2020.

Annexe A

Capteurs et actionneurs pour Raspberry

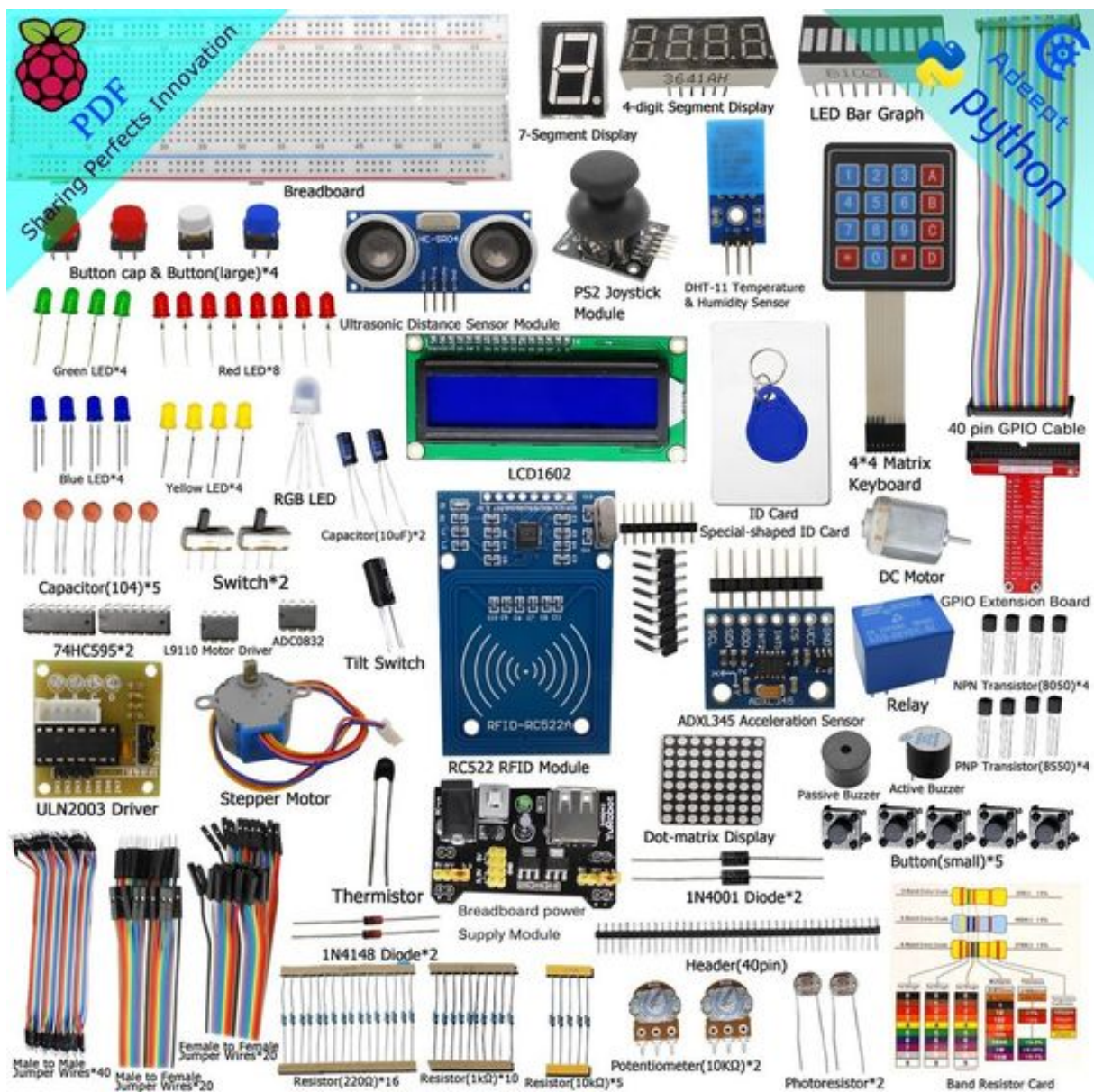


FIGURE A.1 – Capteurs et actionneurs pour Raspberry [38]

Annexe B

Tutoriel pour le simulateur à retour haptique

Ci-dessous se trouve le tutoriel pour l'utilisation du simulateur à retour haptique. Le tutoriel se trouve également sur le *gitlab* du paquet *ros_rob* afin d'avoir accès aux différents liens ainsi qu'une meilleure qualité pour la lecture : https://git.immc.ucl.ac.be/tuerlinckxt/ros_rob/-/blob/CAN-develop_v2/tutorial_simulator.md.

User manual and tutorial for the haptic feedback simulator with ROS-Robotran

This file will explain the different steps to be carried out for using the haptic feedback simulator of the UCL. The purpose of this application is to drive a vehicle (simulated on a computer) with a real steering wheel and receive the torque feedback back in the steering wheel in order to create a realistic driving-simulator.

First there is a small introduction and explication on the functioning of the simulator.

Then there will be some information on installing and using the following software :

- Robotran : this software take care about the simulation of a multi-body model (for this application it will be kart, car, ...)
- ROS : this software is used to carry on the communication between the between different devices.

Then some instructions on how to install and compile the `ros_rob` package on your computer.

After that the tutorial will show you how to connect your computer to the Raspberry used for the simulator.

The next step explains to you how to turn on the servo-drive and servo-motor of the simulator and how to launch and use the application.

Finally, the last part explains how to add its own multi-body vehicle model into the application.

Note : For this example you will need a computer runing Linux Ubuntu (the example here is done with Ubuntu 18.04.4 LTS).

Introduction

Here is a presentation of the haptic feedback simulator :



With :

1. A seat *Sabelt*.
2. A steering wheel connected to a servo-motor.
3. A screen displaying a graphical interface of the simulation of the vehicle.
4. A servo-drive used for the communication between the servo-motor and the Raspberry.
5. A switch to turn on/off the servo-motor.
6. A Raspberry with a PiCAN2 used for the communication between the servo-drive and the computer.
7. A computer running the simulation of the vehicle.

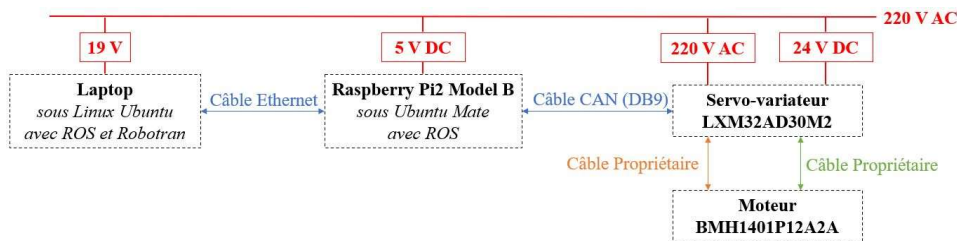
The different steps of communication are the following :

1. The driver steers the car using the steering wheel.
2. A coder include in the servo-motor measure the position and velocity of the steering wheel. The servo-drive sends those data to the Raspberry through a CANopen protocol of communication.
3. The Raspberry receive the datas and sends it to the software Robotran (runing on the computer) through a ROS protocol of communication (called *topic*).
4. Robotran, which is simulating the dynamics of a vehicle, use the datas to compute the torque feeback in the steering wheel.
5. It returns the value to the Raspberry through ROS communication. Then the Raspberry sends a CANopen command to the servo-drive which set the computed torque into the servo-motor.

The driver is now able to steer the vehicle and feel the torque feedback.

Hardware architecture

The following diagram shows the various hardware components, the physical connections between them as well as their necessary power supply :



Installation and configuration of Robotran

For using Robotran on Linux you need to install [MBSysPad](#) and [MBSysC](#) in order to use the C/C++ version of Robotran.

To understand the multibody system dynamics used here you can read the [Robotran Basics](#).

In order to have a better understanding and management of Robotran you can follow the [Getting Started Tutorial in C/C++](#). If you want to go further you can also follow the [Equilibrium and Modal\(not used here\) module of Robotran Tutorial](#).

This package also use the real-time features of Robotran so please follow [this tutorial](#).

Finally, in order to link Robotran and ROS, we need to use multithreading. Update the package index \$ `sudo apt-get update` and install the pthread library \$ `sudo apt-get install libpthread-stubs0-dev`.

Installation and configuration of ROS

For installing ROS follow [this link](#). To configure your ROS environnement follow the [Installation and Configuration](#) with one constraint, you must create your catkin workspace (`catkin_ws`) in your `/home/user_name/` folder (in order to allow Robotran to find the correct path to an executable `open_gl_exe` mandatory for the real-time simulation).

After that, in order to have a better comprehension and management of ROS, you can follow all the beginner level from [official ROS Tutorials](#) and [Writing a simple publisher/sucriber](#).

Note that this ROS-Robotran application has only be written and tested on Linux distribution.

The ROS version used here is *Melodic* and the ROS compiler used is *catkin*. The nodes are written in C++ except one in Python.

Installation and compilation of the `ros_rob` package

1. Download the [ros_rob package](#) in your `catkin_ws/src` folder.
2. Go to the CMakeLists of each Robotran project (for example : `/home/user_name/catkin_ws/src/ros_rob/src/Car/workR/CMakeLists.txt`) and set the correct way to the MBSysC folder with `/home/user_name/.robotran/mbsysc-dev/MBSysC/`.
3. Go to the `catkin_ws` folder et compile/build it with `~/catkin_ws$ catkin_make`

Setup the communication between the Raspberry and the computer

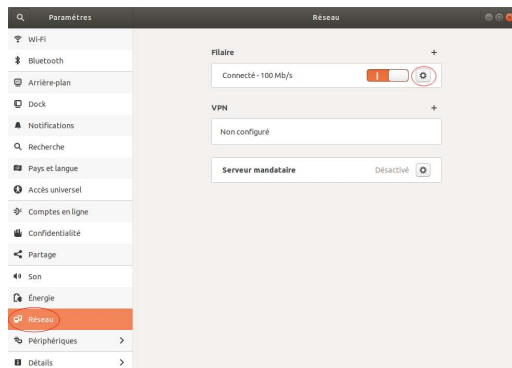
In this application, the Raspberry is used to have CANOpen communication capability (with an extension *PiCAN 2* fixed on the GPIO pins of the Raspberry). And in order to use the strength of the ROS communication, we need to create a complete, bi-directional connectivity between the Raspberry and the computer.

SSH communication

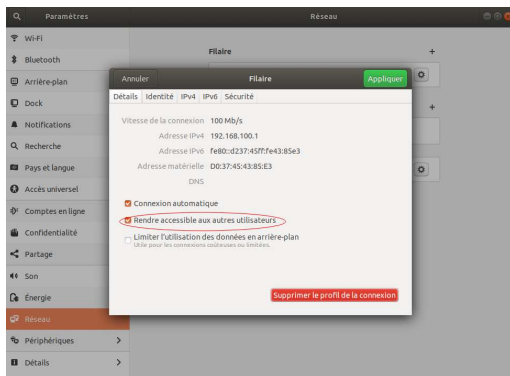
The example here show how to create an SSH communication between the computer with Ubuntu and the Raspberry Pi2 Model B with Ubuntu Mate connected by a ethernet cable. So this section may vary in function of your network configuration but you can do it by your own with some research on internet. The purpose is to obtain the complete, bi-directional connectivity between the devices. [This ROS tutorial](#) will allow you to check that and gives you some issues if you have any problems with the connection.

1. You must connect the Ethernet cable of the Raspberry to the port of the computer that you are going to use for the configuration of the SSH communication. If you don't have ethernet port on your computer you can use and adaptater ethernet to USB.
2. Install SSH client and server on the two devices : \$ `sudo apt-get install openssh-server openssh-client`
3. Create a fixe IP address for ethernet connection of the laptot.

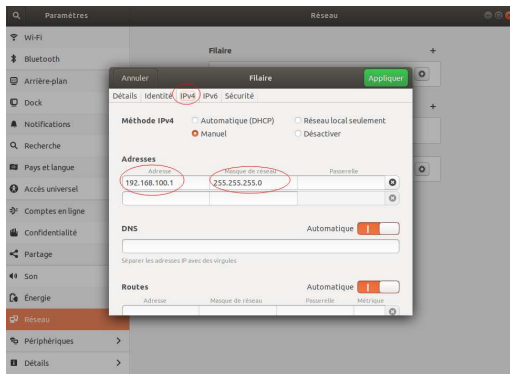
- Go to Parameters->Network
- Go the parameters of the wired used.



- Select *Make it accessible to other devices*



- Then in the IPv4 section, set the IP address to **192.168.100.1** (you can choose the value what you want but you need to keep it in mind) and set the netmask to **255.255.255.0** (this address must be the same for all the devices in the network).



4. Create a fixe IP address for ethernet connection of the Raspberry.

- You can edit the `/etc/network/interfaces` file with :

```
auto lo
iface lo inet loopback

auto eth0
iface eth0 inet static
address 192.168.100.2
network 192.169.100.0
netmask 255.255.255.0
broadcast 192.168.200.255

allow-hotplug wlan0
iface wlan0 inet manual
wpa-conf /etc/wpa_supplicant/wpa_supplicant.conf
```

- Note that the netmask address is the same and keep in mind the IP address **192.168.100.2**
 - If you want to return to a classical ethernet communication just change *static* by *dhcp* and put all the addresses in comment.
5. If not done, enable SSH on the Raspberry. By typing `$ sudo raspi-config` and then go the *Interfacing Options* and enable SSH or with the command line `$ sudo update-rc.d ssh enable && invoke-rc.d ssh start`
 6. You need to put a name on the IP address in order to use it in a more simple way.
- Just add lines with the IP address and a name (you need to use the same names used here or you will have to make some changements in the `launch_haptic_simulator.py` file) in the `/etc/hosts` file of the two devices (you need to have admin rights) :

```
192.168.100.2 raspberry_ethernet 192.168.100.1 laptop_ethernet
```

7. Reboot the laptop and raspberry to take the changes in account.
8. Reconfigure the SSH of the Raspberry : `$ sudo dpkg-reconfigure openssh-server`

9. Sometimes the fire-wall of the computer can make obstruction in the communication between the two devices. You can desable the fire-wall with (need to be done each time the computer is turned on):

```
sudo iptables -F
sudo iptables -X
sudo iptables -t nat -F
sudo iptables -t nat -X
sudo iptables -t mangle -F
sudo iptables -t mangle -X
```

Now the connection is established, you can verify that there is a full bi-directional connectivity between the two devices. [This ROS tutorial](#) will allow you to do that and gives you some issues if you have any problems with the connection.

(Optionnal) We can also increase the robustness of the connection and create an authorized-key on the laptop :

1. Create an ssh-key on the laptop :

```
ssh-keygen -t rsa
```

2. Enter the file in which save the key and a passphrase if you want.

3. Copy the public key into the new machine's authorized_keys file, for our example it will be :

```
ssh-copy-id -i <path-to-the-file-created-at-point-2> tuerlinckxt@192.168.100.2
```

ROS across multiple machines

Now that you have a SSH connection between your two devices (here a laptop and a Raspberry), we are going to setup the ROS environment. When you run ROS across multiple machines you only need one master, all the other devices are slaves. But the nodes running on slave device must be configured to use the same master. In order to do that we are going to modify the `.bashrc` file of our device.

For this example : * one master device, here the laptop (IP address = **192.168.100.1** named *laptop_ethernet*) * one slave device, here the Raspberry (IP address = **192.168.100.2** named *raspberrypi_ethernet*)

- Modify the `.bashrc` file of the master by adding this lines at the very beginning of the file :

```
source /opt/ros/melodic/setup.bash source $HOME/catkin_ws/devel/setup.bash

export ROS_IP=laptop_ethernet export ROS_MASTER_URI=http://laptop_ethernet:11311

echo "ROS_IP :"$ROS_IP echo "ROS_MASTER_URI :"$ROS_MASTER_URI
```

`echo` print the addresses and allows you to verify the addresses when you open a new terminal. `source` allows to take in account in the ROS environment the modification of the `.bashrc` file when you open a new terminal.

- Modify the `.bashrc` file of the slave by adding this lines at the very beginning of the file :

```
source /opt/ros/melodic/setup.bash source $HOME/catkin_ws/devel/setup.bash

export ROS_IP=raspberrypi_ethernet export ROS_MASTER_URI=http://laptop_ethernet:11311

echo "ROS_IP :"$ROS_IP echo "ROS_MASTER_URI :"$ROS_MASTER_URI
```

Note that the `ROS_MASTER_URI` address is the same for the two devices.

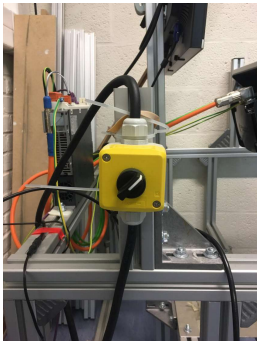
There is a `roslaunch` function of ROS which allows to launch nodes on different devices at the same time is not used in this application because we need interaction and displaying with the Linux terminal of the two devices which is not possible with `roslaunch`. However you can follow this [tutorial](#) if you want to use it in other project.

Setup the hardware and use of the haptic feedback simulator

This section show you how to connect your computer to the simulator and turn on the different devices. Next we will see how to launch a simulation.

Setup the devices

1. As explain at the **Setup the communication between the Raspberry and the computer** section. You must connect the Ethernet cable of the Raspberry to the port of the computer that you used for the configuration of the SSH communication. If you don't have ethernet port on your computer you can use and adaptater ethernet to USB.
2. Turn on the Raspberry by plugging it. You will have to wait a few seconds before it is operational.
3. Check that the cable for the CAN communication (the purple one) is connected to PiCAN2 of the Raspberry.
4. Turn on the screen and connect it to the computer.
5. Plug the cable for the ground.
6. Turn on the servo drive by plugging it.
7. Plug the servo-motor power and then turn on the rotary switch. At this time the power stage (= the servo-motor) is not yet enable, it will be done by CAN communication during the running of the code. The message "rdy" (for "ready") is displayed on the servo-drive screen.



Launch the simulation

First, open a terminal on the computer and enable the ROS environment with :

```
roscore
```

After that, open a terminal on the computer and connect it to the Raspberry with :

```
ssh raspberrypi_ethernet
```

Then run the node to enable the servo-drive with :

```
roslaunch ros_rasp lexium32A_CANopen
```

You will have to enter the password of the Raspberry, you can find it right next to him !

Finally open a new terminal again and launch the application with :

```
roslaunch ros_rob haptic_simulator.launch
```

1. So you have two main terminals :
 - One for the ROS node on the Raspberry : it take care about the CAN communication with the servo-drive and display all the informations off the servo-drive setting.
 - One for the two ROS nodes on the Laptop : one node runs the Robotran simulation and the second creates some interaction with the user for the simulation.
2. On the laptop terminal, select the vehicle you want by typing the corresponding number in terminal and pressing "enter" :

```
/home/tuerlinckx/catkin_ws/src/ros_rob/launch/haptic_simulator.launch http://localhost:...
Fichier Edition Affichage Rechercheur Terminal Aide
setting /run_id to ee22b602-a7f9-11ea-a02b-14abc5a8a313
process[rosout-1]: started with pid [25730]
started core service [/rosout]
process[Haptic_Simu-2]: started with pid [25737]

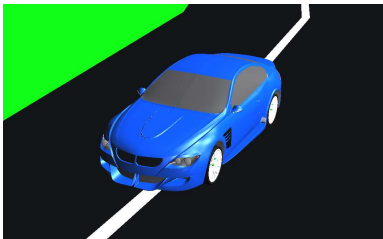
|||||
||
|| WELCOME TO THE VEHICLE HAPTIC FEEDBACK SIMULATOR ||
||
|||||

-----
Launching the process on the Raspberry ...
-----

Choose your vehicle:
1. Car
2. Kart

Enter the number of the vehicle you want : █
```

- The car



- The kart



3. Next, select the simulation you want :

```
/home/tuerlinckx/catkin_ws/src/ros_rob/launch/haptic_simulator.launch http://localhost:...
Fichier Edition Affichage Rechercheur Terminal Aide

1. Car
2. Kart

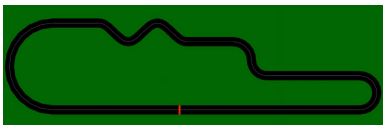
Enter the number of the vehicle you want : 1
The "Car" has been selected !

-----

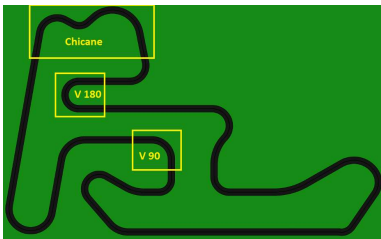
Choose your type of simulation:
1. Circuit 1 (Advised speed = 60 km/h)
2. Circuit 2 (Advised speed = 60 km/h)
3. Full speed turn (Advised speed = 150 km/h)
4. Traffic circle (Advised speed = 35 km/h)
5. Obstacle avoid (Advised speed = 75 km/h)

Enter the number of the simulation you want : █
```

- Circuit 1



- Circuit 2



- **Full speed turn** *not available with the kart* : drive at high speed then turn and feel the augmentation of the torque in the steeringwheel. Simulation done with random mechanic trail.
- **Traffic circle** *not available with the kart* : take a traffic circle at constant speed. At a random time, the ground will be sliding during 2.5 seconds, try to stay on the traffic circle ! Simulation done with a random mechanic trail.
- **Obstacle avoid** *not available with the kart* : drive on a straigh road a constant speed. An obstacle is placed at a random distance, try to avoid it ! Simulation done with random mechanic trail. The "inside wiew" is advised.

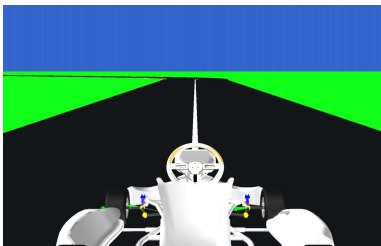
4. Choose the view by typing the correct number *not avaible with the kart* :

```

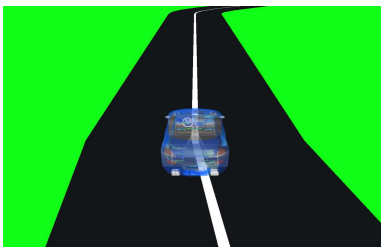
/home/tuerlinckx/catkin_ws/src/ros_rob/launch/haptic_simulator.launch http://localhost:...
Fichier Edition Affichage Rechercher Terminal Aide
-----
Choose your type of simulation:
1. Circuit 1 (Advised speed = 60 km/h)
2. Circuit 2 (Advised speed = 60 km/h)
3. Full speed turn (Advised speed = 150 km/h)
4. Traffic circle (Advised speed = 35 km/h)
5. Obstacle avoid (Advised speed = 75 km/h)
Enter the number of the simulation you want : 1
-----
Choose your simulation view:
0. Outside the car
5. Inside the car
Enter the number of the simulation view you want :

```

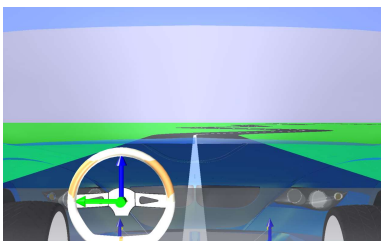
- **The kart view**



- **The car outside vue**



- **The car inside vue**



5. Then, select the initial speed of your vehicle in km/h and press "enter", the simulation of the vehicle will launch : ride it with the steerwheel !

```

/home/tuerlinckx/catkin_ws/src/ros_robot/launch/haptic_simulator.launch http://localhost:3000
Fichier Edition Affichage Rechercheur Terminal Aide
1. Circuit 1 (Advised speed = 60 km/h)
2. Circuit 2 (Advised speed = 60 km/h)
3. Full speed turn (Advised speed = 150 km/h)
4. Traffic circle (Advised speed = 35 km/h)
5. Obstacle avoid (Advised speed = 75 km/h)
Enter the number of the simulation you want : 1
-----
Choose your simulation view:
0. Outside the car
5. Inside the car
Enter the number of the simulation view you want : 0
-----
Enter the initial speed of your vehicle in km/h :

```

4. At the end of the simulation, you can choose if you want to restart a new one.
 - If yes you will return to the point 2.
 - If no the ROS node will shutdown. Then wait that the closing procedure is done on the terminal of the Raspberry and that the message "Closing procedure successful !" is displayed. Turn off the switch of the motor. Turn off the Raspberry with `sudo shutdown` and then unplug everything.

Include your own Robotran project

The examples here is made with a Robotran projects wich simulate the dynamics of a car and a kart. But you can also add your own Robotran project (other type of vehicles, other multibody system, ...).

This tutorial assumes you followed the **Installation and configuration of ROS** and **Installation and configuration of Robotran** sections.

1. Download this package in your `catkin_ws/src` folder.
2. Copy all the files and folders of your Robotran project (userfetR, workR, ect) without *build* folder in `ros_robot/src/name_of_your_project`.
3. Then add the CMakeList of Robotran to the one of ROS by modifying this line in the ROS CMakeList


```
add_subdirectory(${CMAKE_CURRENT_SOURCE_DIR}/src/name_of_your_project/workR)
```
4. Include the ROS and Pthread libraries in the Robotran CMakeList


```
include_directories(/opt/ros/melodic/include /opt/ros/melodic/lib)

target_link_libraries(${Executable} -I/opt/ros/melodic/include -L/opt/ros/melodic/lib -lroscpp -lrostime -lroscpp_serialization)

target_link_libraries(${Executable} -lpthread)
```
5. Change the extension of the Robotran main.c by main.cpp. Change the content of the main with the template [main_robotran_template.cpp](#) and follow the change instructions inside the template.
6. As you can see at point 5) and 6) we need to use a user model structure from Robotran. Follow [the user model tutorial for structures](#) in order to initialize it in *MBSysPad* and create an header file (also see [thread_struct.h](#) for an example of declaration of the structure). This structure contains pointeur to function like *give_torque_access* wich allows ROS publisher and suscriber to have access to *mbs_data* without segmentation fault cause of the multithreading. After that you can give the access to the ROS publisher and suscriber in all the user function with :


```
(*mbs_data->user_model->thread.thread_struct->pointeur_<name_of_access_function>());
```
7. **Note** : the main of Robotran is now in C++ but the user functions that you way write are usually in C and call by this C++ function. You can have a look on [Name Mangling and extern "C" in C++](#) to ensure that the C++ compiler behaves like a C compiler for this functions.
8. Go to the `catkin_ws` folder et compile/build it with `~/catkin_ws$ catkin_make`

Deal with multiple Robotran projects

If you need to deal with several Robotran projects, just follow the above instructions one more time with two exeptions :

- At the point **3** you can have only one link with a Robotran CMakeLists (only one *add_sudirectory*).
- At the point **8**, before compile/build your workspace you need to delete the `ros_robot` folder in the *build* folder.

This means that you can compile and build only one Robotran project at a time. But once the new ROS executbale is created (here the *exe_your_project_name*) you can run wich you want.

Use your own ROS message

The ROS messages here are used to vehicle data for the torque, position and velocity of a steerwheel wich are floating point values. It could be usefull to create your own ROS messages in order to deal with the values provided and needed by your Robotran project. You can consult this [tutorial](#) and this [ROS wiki](#) for more informations.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | www.uclouvain.be/epl