Intégration D'un Robot Mobile Roomba Dans Un Réseau De Capteurs

Lucile COSSOU
2ème année en Signal, Image, Communication et Multimédia,
Année universitaire 2013/2014
Stage réalisé du 13 Mai 2013 au 02 Août 2013
Sous le tutorat de :

M Roger PISSARD GIBOLLET, ingénieur au SED
A Inria Grenoble,
Service d'Expérimentations et de Développement
655 avenue de l'Europe,
Montbonnot
38 334 Saint Ismier Cedex France

Remerciements

Je remercie Roger PISSARD-GIBOLLET, mon tuteur, pour le temps qu'il m'a consacré et ses conseils ainsi que pour m'avoir fait confiance pour ce projet.

Un merci également tout particulier à l'ensemble de l'équipe du Service d'Expérimentation et de Développement d'Inria Grenoble pour m'avoir si bien accueillie et pour leur aide précieuse, en particulier à :

- Soraya ARIAS pour m'avoir orientée vers ce stage
- Nicolas et Gaetan pour avoir partagé avec moi leurs connaissances en informatique
- Jean-François pour les modifications qu'il a apporté au robot
- Sandrine, Anthony, Fabien et Frédéric pour leur bonne humeur et leur sympathie
- Anthony, pour son humour et son soutien tout au long du stage

Je tiens également à remercier Claudie pour sa disponibilité et l'efficacité avec laquelle elle a pu répondre à mes demandes administratives.

Un dernier merci revient à la communauté de développeurs de ROS, pour avoir répondu avec patience à mes questions sur leur forum.

Introduction

Ce rapport expose les manipulations effectuées et résultats obtenus lors de mon stage au sein du Service d'Expérimentation et Développement d'Inria Grenoble.

Vous trouverez dans une première partie une présentation de l'entreprise, du sujet du stage ainsi que des outils utilisés. Le second chapitre sera consacré à la prise en main des dits outils, ainsi qu'à l'évaluation de leur qualité. La suite de ce rapport sera consacrée à l'exposition du travail effectué ainsi qu'à l'explication de sa mise en œuvre.

Enfin, dans une dernière partie, ce rapport abordera les perspectives d'évolution de ce projet et les améliorations que l'on peut encore lui apporter, puis s'achèvera sur une conclusion personnelle sur l'ensemble du stage.

Sommaire

Contexte

Inria

Illustration 1: Vue d'Inria Grenoble

Le centre de recherche Inria Grenoble est l'un des huit centres de recherche de l'Institut National de Recherche en Informatique et Automatique. Créé en 1992, Inria Grenoble est implanté sur les villes de Grenoble et de Lyon. Il accueille près de 680 personnes réparties dans 9 services et 33 équipes de recherche différentes. Celles-ci ont pour coeur de recherche les domaines de l'interaction homme-machine, la modélisation et la simulation, la conception de logiciels fiables et optimisés.

Chaque centre Inria possède un Service d'Expérimentation et de Développement (SED) qui gèrent les plateformes expérimentales de l'entreprise et participent au développement logiciel au sein des équipes-projets. Le SED d'Inria Grenoble gère notamment les plateformes de Grille et Grappe, de Réalité Virtuelle, de Réseaux de Capteurs et de Robotique. Cette dernière est hébergée dans la halle robotique d'Inria Grenoble, où j'ai effectué mon stage.

Sujet du stage

Dans le cadre d'un financement d'Équipement d'Excellence (Equipex), Inria contribue à la plateforme FIT pour expérimenter les futurs besoins de l'internet des objets. FIT fédère plusieurs systèmes dont un réseau de capteurs à grande échelle (IOT-LAB) réparti sur les sites de Lille, Strasbourg, Rennes et Grenoble.

Le réseau est constitué essentiellement de nœuds capteurs fixes répartis dans des salles ou des couloirs des bâtiments de chaque site. Il existe également des nœuds capteurs mobiles qui se déplacent à l'intérieur des locaux des Inria. Pour cela le Service d'Expérimentation et de Développements de Grenoble étudie la possibilité d'utiliser des robots pour créer des nœuds mobiles. Celui choisi pour cette tâche est le Turtlebot, un robot se déplaçant au sol et dont la spécialité est la navigation.

Ce stage fait suite à un premier stage d'exploration des possibilités du Turtlebot, effectué par un élève de la filière CSE en 2012. Au cours de ce dernier, un algorithme avait été implémenté sur le Turtlebot1 afin de lui permettre d'effectuer des allers-retours de manière autonome entre un point A et un point B de Inria, tout en évitant les obstacles et en allant se recharger sur sa station d'accueil lorsque ses batteries sont faibles.

L'objectif du stage de cette année est de prendre en main la nouvelle version du robot, la nouvelle version de ses logiciels et d'implémenter sur le nouveau robot les mêmes fonctionnalités que sur le robot précédent, tout en y apportant des améliorations.

Le Turtlebot

Le Turtlebot est un robot destiné à la recherche et à l'éducation. Au départ développé par une société américaine spécialisée dans la robotique nommée Willow Garage à partir d'un robot aspirateur modifié en guise de base mobile, le Turtlebot est à présent produit par la société coréenne Yujin. Les fonctionnalités disponibles sur le Turtlebot1 et le Turtlebot 2 ne sont donc pas exactement les mêmes. Le Turtlebot 2 est constitué des composants suivants :

Illustration 2: Turtlebot 2

La base mobile iClebo Kobuki

Illustration 3: Base mobile Kobuki

Le iClebo Kobuki est un robot low-cost destiné à la recherche et à l'éducation. Il possède les caractéristiques suivantes :

- vitesse de déplacement maximale : 70 cm/s
- vitesse de rotation maximale : 180°/s
- charge utile : 10 kg

et embarque les équipements suivants :

- une batterie Lithium-Ion 14.8V, 2200 mAh permetttant une autonomie de 3h
- deux LEDs programmables
- un gyromètre (110°/s)
- deux capteurs de dérapage
- un odomètre (2578.33 tops par tour de roue)
- trois bumpers
- trois capteurs infrarouges pour détecter le vide sous le robot
- 3 récepteurs infra-rouges

La Kinect pour Xbox 360

Illustration 4: Kinect pour Xbox 360

La kinect pour Xbox 360 embarquée par le Turtlebot permet au robot d'avoir une vision de son environnement et remplace également efficacement un émetteur/récepteur laser. Elle possède notamment les caractéristiques suivantes :

- champ de vision étendu entre 0.6 et 8 mètres (pas de vision de près)
- angle de vue : 57° horizontalement, 43° verticalement
- résolution : 640*480 pixels

Le netbook

Le Turtlebot est livré avec un PC portable de petite taille et léger, facilement transportable par le robot et ne qui ne constitue pas une surcharge trop importante. Le PC choisi pour la version du Turtlebot2 est le Asus EeePC 1025C.

Il embarque la version Precise d'Ubuntu, sur une architecture 32 bits, dispose de 3 ports USB et d'une carte wifi.

Les autres composants

Le Turlebot est également composé d'un châssis en bois recomposé et plastiques légers, de câbles adaptés pour relier le Turtlebot au PC via ses ports USB et pour pouvoir alimenter la kinect à travers la base mobile. Il est également livré avec une station de rechargement.

Illustration 5: Station de rechargement

ROS

Illustration 6: logo de ROS Groovy

Développé par le laboratoire de recherche en robotique américain Willow Garage, ROS (Robot Operating System) est le middleware qui fait fonctionner le Turtlebot. Il fournit les outils nécessaires à la création d'applications robotiques tout en permettant de garder un certain niveau d'abstraction. Cela permet de réaliser des projets portables d'un robot à un autre. Notamment du Turlebot 1 au Turtlebot 2.

ROS est également une plateforme de développement très modulable à laquelle on peut ajouter des packages pour en augmenter les possibilités et pour y intégrer des logiciels tiers tels que des simulateurs comme Gazebo ou Stage.

Prise en main

Prise en main de ROS

Chaque programme de ROS est appelé *"node"*. Il peut communiquer avec d'autres programmes de ROS de trois façons différentes :

- avec des *"topics"* dans lesquels un (éventuellement plusieurs) node viendra publier tandis qu'un autre (ou plusieurs autres) pourra venir y lire les données écrites.
- avec des services, avec lesquels un *node* peut envoyer une requête à un autre *node* en particulier.
- ou avec des actions grâce auxquelles un *node* peut envoyer un but à un *node* récepteur qui va alors essayer de l'accomplir et publier régulièrement dans un *topic* des informations concernant l'évolution de la tâche demandée.

Un *node* nommé "master" permet d'interfacer tous ces *nodes* entre eux.

Une fois les *nodes* programmés, il faut les *« packager »*. Pour cela, ROS Groovy utilise un outil nommé *catkin* qui permet de compiler les fichiers d'un dossier sous la forme d'un *package*. Celui-ci peut contenir des *nodes*, des librairies indépendantes de ROS, des fichiers de configuration, des bases de données, des fichiers descriptifs des messages, services, *topics*, actions créées. Le but du *package* étant de rendre le *node* ainsi programmé facilement réutilisable. Chaque *package* contient également un fichier descriptif qui liste des dépendances relativement à d'autres *packages* de ROS.

Installation du Turtlebot

Le Turtlebot 2 est livré tout assemblé (base mobile, kinect et laptop sont fournis ensemble) et sur le laptop, ROS, ainsi que les packages nécessaires au fonctionnement de la base mobile, est préinstallé. Il est donc possible d'utiliser certaines fonctionnalités du robot dès sa sortie de la boîte.

En pratique, pour que le Turtlebot soit parfaitement opérationnel et que l'on puisse utiliser la kinect, il faut également installer les modules du Turtlebot (fournis sur le site de ROS) sur le laptop du robot. Ainsi, la liste des modules supplémentaires ajoutés au laptop est la suivante :

- ros-groovy-turtlebot (intégration de la kinect)
- ros-groovy-turtlebot-apps (intégration des fonctionnalités de base du turtlebot)
- ros-groovy-turtlebot-viz (intégration des modules de visualisation du turtlebot)
- ros-groovy-audio-common (intégration des modules sonores du laptop)

Ainsi configuré, le Turtlebot peut d'ores et déjà exécuter certaines actions complexes sur commande :

- se docker tout seul
- naviguer jusqu'à un point donné en utilisant la kinect pour se repérer
- se déplacer dans une direction donnée à une vitesse donnée
- faire de la synthèse vocale
- visualiser son environnement en 3D
- estimer la distance parcourue et son orientation grâce à ses capteurs

Tests

Afin de connaître les différences entre le nouveau et l'ancien Turtlebot et d'avoir une idée de la fiabilité de ses mesures et des fonctions pré-implémentées sur le robot, on se propose de tester les fonctionnalités du robot qui nous seront à priori utiles.

Odométrie

A l'aide d'un script python on fait effectuer au robot des trajets d'une longueur approximative de 4,80 m. On estime alors la distance parcourue par le robot à l'aide des données provenant des capteurs d'odométrie et du gyroscope et en la mesurant à l'aide d'un mètre dérouleur d'une longueur de 5m.

On calcule ensuite l'erreur entre la distance parcourue réellement (et mesurée au mètre) et la distance estimée par les capteurs du Turtlebot. On obtient les résultats suivants :

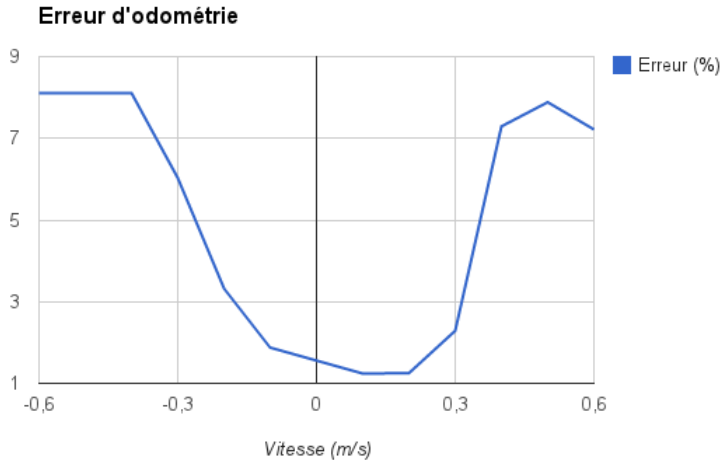


Illustration 7: Erreur d'odométrie (statistiques)

L'erreur moyenne est de 5%, mais reste inférieure à cette valeur pour des vitesses réduites en marche avant. L'odométrie va donc fournir des mesures relativement fiables pour l'usage que l'on va faire du Turtlebot (vitesse modérée, marche avant privilégiée pour pouvoir tirer profit des capteurs pendant le déplacement), mais ne suffira pas à elle seule à localiser le robot. C'est pourquoi on utilisera également la kinect pour localiser avec précision le robot.

Auto-docking

La base de rechargement émet trois champs infrarouges différents qui permettent au robot d'estimer sa position par rapport à la base.

Chacun de ces champs est alors divisé en deux sous-champs : les champs *NEAR* et *FAR* qui permettent au robot de calculer son éloignement par rapport à la base. Ainsi, avec ses récepteurs infra-rouges, le robot peut estimer à tout moment sa position par rapport à la station.

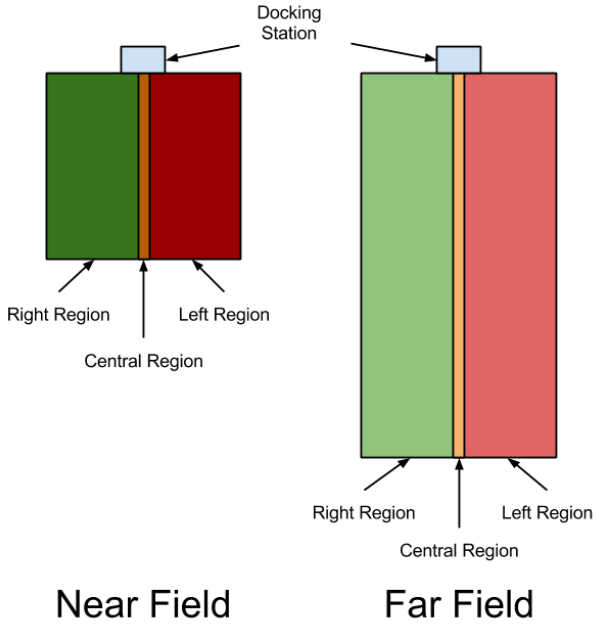


Illustration 8: Champs IR de la base de rechargement

Afin que le robot puisse également connaître son orientation par rapport à la base de rechargement, il est équipé de trois capteurs infrarouges :

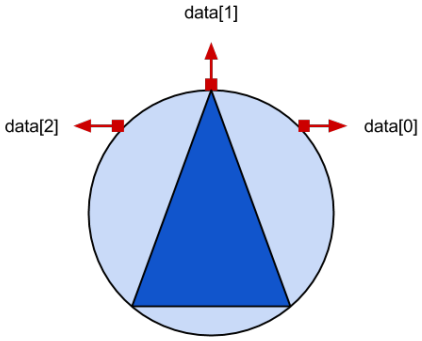


Illustration 9: Capteurs IR de la base mobile du Turtlebot

L'algorithme d'auto-docking utilisé par le kobuki exploite tout ce matériel selon le mode opératoire suivant :

- si le robot est dans la région centrale il avance (relativement vite si il est loin et relativement lentement si il est près de la station). Ensuite, lorsqu'il détecte qu'il commence à se charger, c'est qu'il est arrivé sur la station de rechargement et il peut donc s'arrêter.
- si il est dans la région de gauche, il tourne dans le sens inverse des aiguilles d'une montre, jusqu'à ce que data[0] détecte le signal. Alors le robot est orienté perpendiculairement à la région centrale et n'a qu'à se déplacer tout droit, jusqu'à capter le signal de la région "centre".
- si il est dans la région droite, on fait la même chose que dans la région gauche, mais en tournant dans le sens des aiguilles d'une montre et en attendant l'activation de data[2]

Lors des tests (plusieurs demandes de docking), on s'est aperçu que les résultats étaient globalement plutôt insatisfaisants :

- tout d'abord, les "sous-champs" near et far ne semblent pas être bien détectés par le robot qui croit être dans le sous-champ "far" alors qu'il est déjà passé par le sous-champ infrarouge "near", ce qui cause l'accélération du robot à une distance très proche de la station de rechargement.

- le comportement en cas de détection d'obstacle ne permet pas au robot de se repositionner correctement dans le cas où il bute contre la station de rechargement.
- la partie de l'algorithme traitant les cas où le robot se trouve dans la zone gauche ou droite du champ infra-rouge ne prend pas en compte toutes les positions de départ possible. En effet, selon la position du robot, celui-ci ne se tournera pas face à la perpendiculaire au champ central et aura même tendance à s'éloigner de la station d'accueil.

Tout ceci combiné ne permet pas d'avoir un taux de réussite du docking supérieur à 50%. Pour cette raison, l'algorithme de docking a donc été modifié, comme cela est expliqué plus loin dans ce rapport.

Navigation

Le Turtlebot est également fourni avec un module nommé "Navigation". Celui-ci permet de faire naviguer le robot d'un point à un autre en lui donnant des coordonnées. Pour cela, il utilise ses capteurs d'odométrie, la kinect ainsi qu'un algorithme de navigation basé sur celui de Monte Carlo. Ce dernier permet au robot d'estimer sa position à l'aide d'un nuage de particules et de probabilité. Ce module est paramétrable, et on peut (entre autres paramètres) l'utiliser avec une carte de l'environnement du robot. Cette dernière permet au robot de se repérer et facilite ainsi l'opération de navigation.

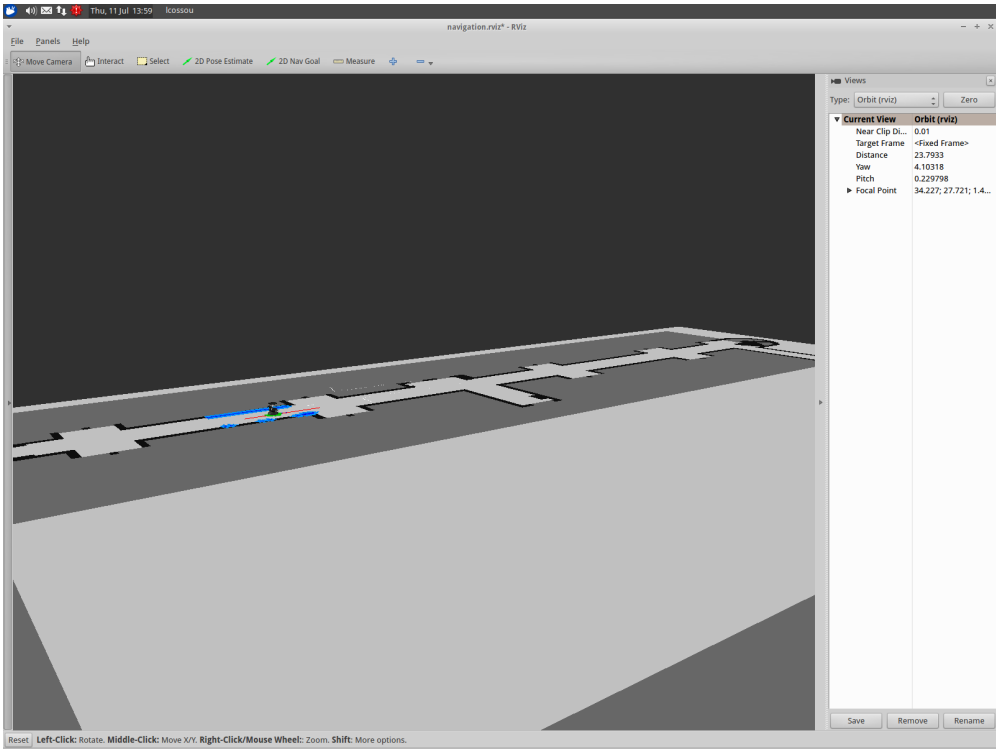


Illustration 10: Interface graphique Rviz

A l'aide d'une carte des locaux réalisée l'année précédente, et du logiciel rviz (intégré à ROS), on peut alors positionner le robot sur la carte et lui donner des objectifs à atteindre sans avoir à faire de programmation.

Le robot essaie alors de se diriger, à partir des données qu'on lui a indiquées, jusqu'au point d'arrivée. Au cours de son parcours, il ajoute à une carte nommée costmap les obstacles qu'il détecte à l'aide de la kinect. Une fois ajoutés à la costmap, les obstacles sont pris en compte par le module de navigation qui calcule le meilleur chemin à prendre pour les éviter, sans pour autant dévier de l'objectif principal.

Si le module de localisation semble être assez précis et robuste et que le robot arrive le plus souvent à destination, on remarque tout de même quelques petits problèmes :

- la détection de chocs n’est pas activée : le robot peut se cogner, cela n’engendre aucune réaction de sa part
- le robot a du mal à détecter les angles droits dans le couloir.
- du fait de l’angle de vue de la kinect et que celle-ci soit utilisée en 2D (comme un laser) plutôt qu’en 3D, les obstacles au sol ne sont pas détectés

Implémentation du scenario

Afin de pouvoir implémenter le scénario proposé, il a d'abord fallu modifier et paramétrer les fonctions de base du Turtlebot pour qu'elles fonctionnent selon nos souhaits.

Auto-docking

Le code du driver du Turtlebot 2, écrit en C++, est publié sous licence BSD par la société Yujin et est disponible sur github. Aussi il est possible de le consulter, de le modifier et de le réutiliser.

Pour que le Turtlebot puisse se docker avec un taux de réussite proche des 100% (un seul échec constaté au cours du stage), l'algorithme original a été amélioré. L'édition du code s'est faite en gardant en tête qu'en pratique, le Turtlebot aurait à se docker dans des espaces plutôt restreints et qu'on ne pourrait donc pas se permettre une grande amplitude de mouvements.

Tout d'abord, de manière générale, la vitesse de translation du Turtlebot a été réduite et ne varie plus en fonction des sous-champs infra-rouges *FAR* et *NEAR*, ce qui a permis de grandement augmenter le taux de succès de la manoeuvre.

Ensuite, le comportement du Turtlebot dans le cas où il ne se trouvait pas dans la zone "center" du champ infrarouge, mais dans la zone "left" ou la zone "right" a été corrigé :

- Si le robot est dans la zone de gauche, il tourne sur lui-même dans le sens trigonométrique jusqu'à ce que son capteur data[0] reçoive le signal. A partir de là, il se déplace tout droit tant que le capteur data[0] reçoit du signal. Si le capteur perd le signal, le robot s'arrête, tourne sur lui-même dans le sens trigonométrique encore une fois jusqu'à ce que le capteur data[0] reçoive le signal... On répète la manœuvre jusqu'à capter le signal de la zone centrale auquel cas le problème devient trivial.
- même chose si le robot est dans la zone "right" mais avec le capteur data[2] et des rotations dans le sens anti-trigonométrique.

On notera qu'il aurait été possible pour plus de fluidité (au détriment d'une amplitude de mouvements restreinte), de ne pas arrêter le robot lorsque le signal est perdu, mais de le faire pivoter légèrement dans le sens anti-trigonométrique (si le robot était dans la zone gauche) à chaque fois que le signal est perdu.

Le traitement de cas d'erreurs (station d'accueil débranchée en cours de manœuvre) a également été ajouté à l'algorithme, de même que de légères optimisations des mouvements du robot.

De manière plus globale, le code a également subi une modification cosmétique, afin de le rendre plus facile à comprendre et donc à modifier.

Dans une logique Open-Source, le code ainsi obtenu sera, après une phase de tests complémentaires sur le site du laboratoire Icube de l'Université de Strasbourg, proposé à la communauté de développeurs ROS via la plateforme github.

Navigation

La carte

Comme expliqué précédemment, pour naviguer dans un environnement de manière autonome, le module de navigation du Turtlebot a besoin d'en connaître la carte. On réutilise cette année celle réalisée par le stagiaire précédent à partir d'un plan d'Inria après avoir vérifié qu'elle était toujours à jour.

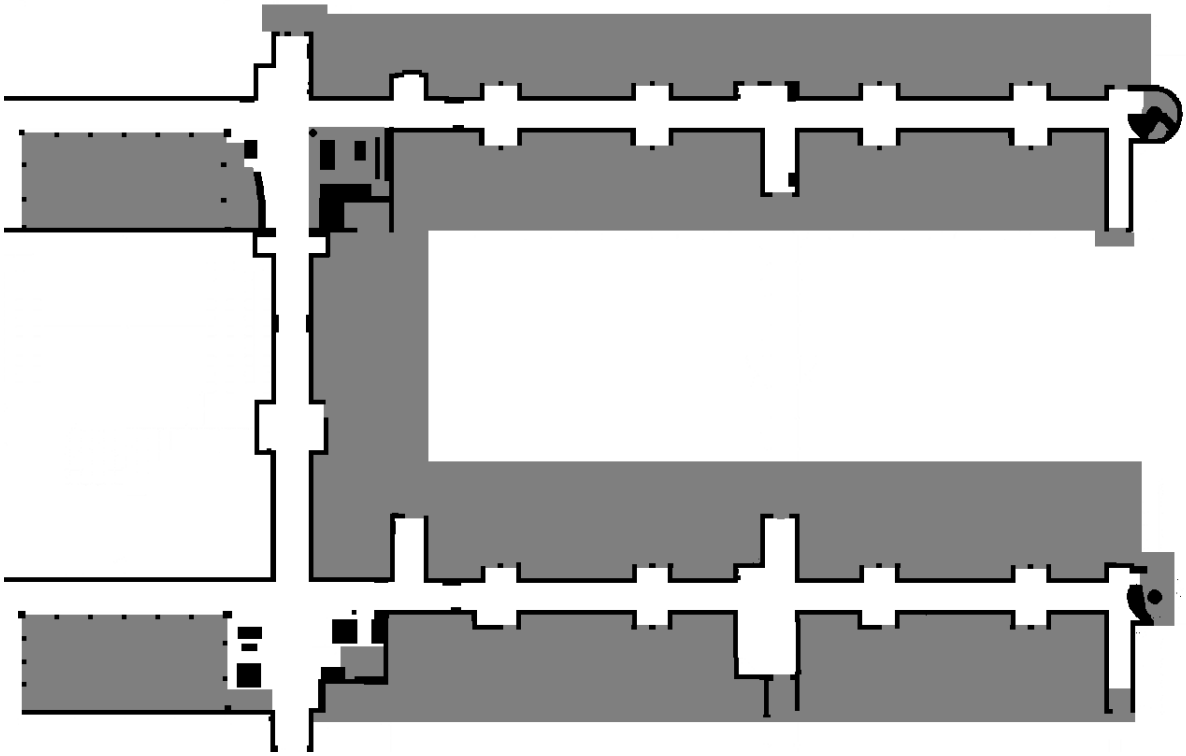


Illustration 11: Plan d'Inria utilisé par le Turtlebot

Celle-ci contient plusieurs informations que le Turtlebot va pouvoir exploiter : la position des obstacles, les zones connues et les zones inconnues mais explorables. Pour définir ces zones, on utilise un code couleur :

- si la couleur de la zone est supérieure (en RGB) à la valeur seuil haute fixée (détails concernant cette valeur plus loin), alors on a affaire à une zone connue
- si la couleur de la zone est inférieure (en RGB) à la valeur seuil basse fixée (voir plus loin également), alors on a affaire à un obstacle
- si la couleur de la zone est comprise entre ces deux seuils, alors on a affaire à une zone inconnue.

Ainsi, les zones blanches sur la carte représentent les zones connues (et donc à priori "libres"), les zones noires des obstacles et les zones grises des zones non cartographiées. Le fichier image en lui-même ne constitue pas un ensemble de données exploitables suffisant pour le Turtlebot. On va donc lui fournir également un fichier .yaml (voir l'Annexe 3) qui contient les données sur la carte nécessaires à sa correcte exploitation, notamment le nom de l'image, sa résolution en mètres par pixel, le point à l'origine de la carte, le seuil haut de couleur, le seuil bas ainsi que la possibilité de considérer les couleurs comme inversées ou non.

Ces fichiers sont ensuite chargés en mémoire par le *node map_server* du *package* de navigation.

Déplacements

Pour contrôler les déplacements du Turtlebot, on va utiliser le package de navigation autonome fourni avec le Turtlebot qui a été présenté dans la section tests.

Le *node* qui contrôle la navigation est nommé *"move_base"*. Celui-ci réalise l'interface entre les *nodes* du *package* de navigation et les autres *nodes* du Turtlebot.

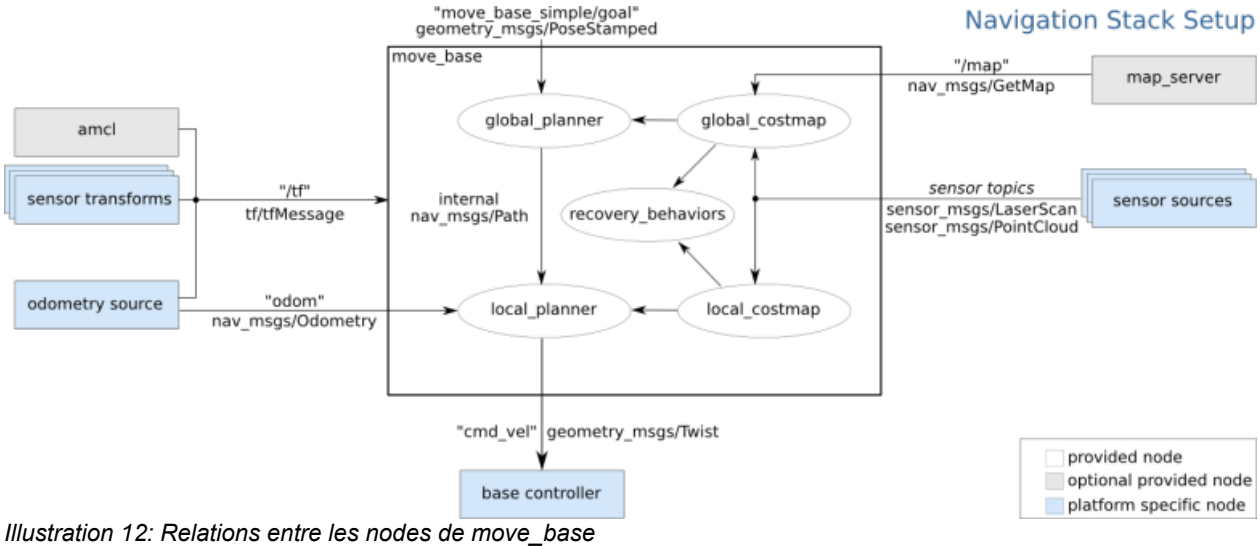


Illustration 12: Relations entre les nodes de move_base

On peut le paramétrer grâce à des fichiers .yaml et lui envoyer des requêtes grâce à des actions. En tant que serveur d'actions, *move_base* renvoie quant à lui de nombreuses informations intéressantes sur l'évolution du robot, et notamment sa position sur la carte.

Pour palier à certains des problèmes cités précédemment, on va régler les paramètres de ce module.

Ainsi, la vitesse maximale de translation du robot est limitée à 0.3 m/s au lieu de 0.6m/s pour laisser au robot le temps de réagir et d'éviter les obstacles, les *bumpers* sont ajoutés à la liste des capteurs à prendre en compte pour la mise à jour des obstacles sur la carte. Cela permet de voir ajoutés sur la *costmap* les obstacles au sol sur lesquels le robot serait venu buter (mais pas encore de faire réagir le robot). On augmente également légèrement le "taux d'inflation" des obstacles, afin que les obstacles et donc les angles prennent virtuellement plus de place sur la *costmap* et que le robot les contourne avec un peu plus de distance.

Implémentation finale

Etat de l'art

Puisque toutes les fonctionnalités nécessaires à l'exécution du scénario proposé sont prêtes, on peut coder le *node* qui contrôlera le Turtlebot.

On reprend le code écrit en python l'année précédente qui implémentait le comportement suivant : le robot va d'un point A à un point B des couloirs d'Inria un nombre indéterminé de fois, pendant 60 min ; si les batteries sont trop faibles, le robot va se recharger et ne recommence ses cycles que lorsqu'il est chargé à nouveau ; si le robot se cogne, il émet un son et prend une photo (celle-ci peut être utile pour retrouver le robot lorsqu'il est perdu).

Si l'algorithme principal ne change pas, et qu'il peut presque être utilisé tel quel, les spécificités du Turtlebot 2 et de ses programmes demandent tout de même de nombreuses modifications.

Tout d'abord, les *nodes* de base du Turtlebot 2 ne sont pas les mêmes que ceux du Turtlebot 1, et n'utilisent donc pas les mêmes *topics*, services ou actions ni n'échangent le même type de données.

Ensuite, les composants n'ont pas les mêmes valeurs critiques et celles-ci doivent donc être modifiées.

De plus, le Turtlebot 1 gérait tout seul ses capteurs de choc et adoptait un comportement par défaut lorsque ceux-ci étaient activés. Dans le cas du Turtlebot 2, aucun comportement n'a été implémenté, cela doit donc être ajouté à l'algorithme final. On obtient alors la machine à états suivantes :

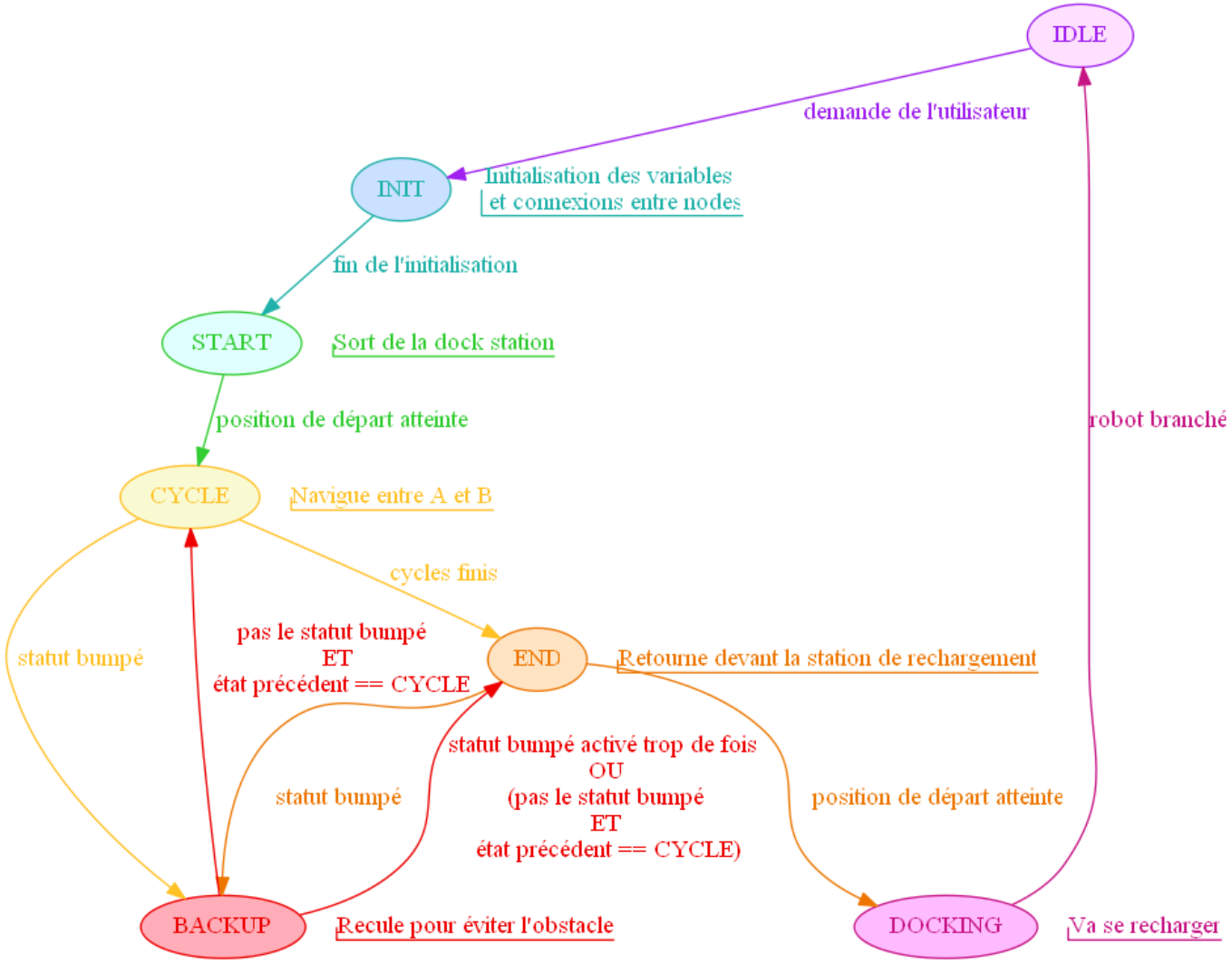


Illustration 13: Scénario implémenté : automate

Enfin, le nombre d'allers-retours effectués par le Turtlebot, ainsi que la durée maximale de ces déplacements doit être paramétrable.

Gestion des échanges entre nodes

Le *node* principal doit communiquer avec différents *nodes* de ROS via différents *topics* afin d'implémenter le scénario. On initialise ces connexions grâce à la méthode _init_pub_sub_action (voir Annexe) :

- publication dans le topic mobile_base/commands/velocity afin de commander la vitesse de rotation des roues
- publication dans le topic mobile_base/commands/motor_power afin d'activer ou désactiver les moteurs du Turtlebot
- souscription au topic mobile_base/sensors/core afin d'observer les évènements sur les capteurs du robot
- souscription au topic laptop_charge afin de pouvoir surveiller le niveau de la batterie du laptop
- création d'un client d'actions en communication avec le node serveur move_base (qui gère la navigation)
- création d'un client d'actions en communication avec le node kobuki_auto_docking (qui gère le docking du Turtlebot)
  L'ensemble de ces interactions est détaillée dans le graphe d'interactions visible à l'Annexe 1.

Valeurs critiques

Le Turtlebot doit pouvoir gérer ses batteries de manière autonome, en conséquence, on se sert de la documentation fournie par Yujin pour déterminer à partir de quel seuil les batteries du robot doivent être considérées comme basses.

A l'usage, on peut remarquer que la batterie du laptop a une autonomie plus faible que la "petite" batterie du robot kobuki (celui-ci est fourni avec deux batteries de capacités différentes) et que c'est donc ce dernier qui va constituer l'élément limitant de l'autonomie du Turtlebot. On ne verra donc pas d'intérêt à utiliser la seconde batterie de la base mobile, qui possède de une capacité encore plus grande. Ses caractéristiques de décharge et de recharge sont fournies en Annexe. A partir de celles-ci, on détermine un seuil suffisamment haut pour que le Turtlebot aie le temps de se déplacer jusqu'à la station de rechargement lorsqu'il détecte que ses batteries sont basses. Ainsi, on choisit un seuil de 10% pour la batterie du laptop et un seuil de 13,5 V.

Gestion des bumpers

Le comportement que le robot doit adopter lorsque l'un de ses *bumpers* est activé est le suivant :

- il doit s'arrêter
- signaler le problème via un signal sonore
- prendre une photo
- reculer de quelques centimères
- pivoter à 180°
- avancer d'environ 50 cm
- reprendre la navigation

Par extension, on appliquera le même comportement lorsque le détecteur de vide ou le capteur de dérapages seront activés. Si l'un de ces trois problèmes est détecté au cours d'une manœuvre, alors celle-ci est interrompue et le Turtlebot adopte le comportement décrit précédemment, y compris si il était déjà en train de reculer suite à l'un de ces événements.

Pour implémenter ce comportement, on utilisera donc une interruption ré-entrante.

Paramétrisation du programme

Les paramètres du programme sont des arguments systèmes, entrés lorsque l'on lance le programme. Le premier est un entier qui indique le nombre d'allers-retours du robot, si il est choisi négatif, on considère alors que le nombre d'allers-retours demandé est indéterminé et que seule la durée du parcours (second paramètre de type flottant) doit être prise en compte. Cela permet un lancement facile depuis le terminal, éventuellement via un *script shell*, sans qu'il y ait besoin d'une intervention humaine. Ce qui peut être utile pour intégrer le robot dans le réseau de capteurs.

Conclusion

Le scénario proposé a été correctement implémenté sur le Turtlebot 2. On peut à présent le lancer via une série de commandes *shell* (éventuellement à distance par ssh) et le robot navigue de façon autonome dans les couloirs d'Inria, tout en ayant un comportement cohérent en cas d'événement inattendu.

Perspectives d'évolution future

Si le Turtlebot 2 est à présent capable de réaliser le scénario choisi, il reste encore à améliorer son fonctionnement. Pour l'instant, il souffre encore de certains défauts : il ne peut pas détecter les obstacles de trop petite taille sans buter dessus et il perd parfois l'accroche au réseau wifi. Bien que ce dernier problème ne soit pas gênant pendant le fonctionnement du turtlebot, puisqu'il est autonome.

Outre ces problèmes à régler, le Turtlebot 2 est également sujets à des bugs inhérents aux *packages* fournis par ROS qui gagneraient à être identifiés et corrigés. Il serait également intéressant d'effectuer le *"packaging"* des programmes utilisés par le Turtlebot pour naviguer afin qu'il suffisse d'une seule commande pour lancer la navigation du robot contre 4 actuellement (pour lancer tous les *nodes* dont le Turtlebot a besoin).

Dans un futur (très) proche, la prochaine version de ROS devrait sortir, et apporter peut-être un peu plus de stabilité au Turtlebot. Le portage du travail réalisé ici devrait être faisable assez facilement étant donné que le matériel reste le même et que les fonctions de ROS utilisées par le code du *node* principal gardent le même principe de fonctionnement d'une version à une autre de ROS. Cela sera d'autant plus aisé qu'au cours du stage, j'ai dû réaliser un guide de prise en main du Turtlebot qui explique les manipulations à faire pour qu'il soit prêt à exécuter le scénario proposé et rédigé un récapitulatif des recherches et travaux effectués sur le wiki de l'entreprise.

Conclusion

Ce stage a été une première occasion pour moi de me consacrer à temps plein à un seul et unique projet, dans un domaine qui m'intéresse beaucoup -à savoir la robotique. La tâche à accomplir en elle-même autant que le cadre de travail ou bien que la communauté de développeurs qui travaillent sur ROS et le Turtlebot ont fait de ce stage une expérience très enrichissante.

Cela m'a permis de découvrir de nombreux outils de programmation spécifiques à la robotique, de m'initier au monde de l'Open Source et de renforcer mes connaissances sur les systèmes Linux.

J'ai été surprise de me rendre compte que les plus grosses difficultés que j'ai eu à surmonter n'étaient pas tant liées à des problèmes d'algorithmique ou de syntaxe en général, mais plutôt à la prise en compte en même temps de plusieurs paramètres du robot et à la gestion d'interruptions entre les différents modules.

Bibliographie

ros.org

answers.ros.org

inria.fr

http://kobuki.yujinrobot.com/

http://www.senslab.info/

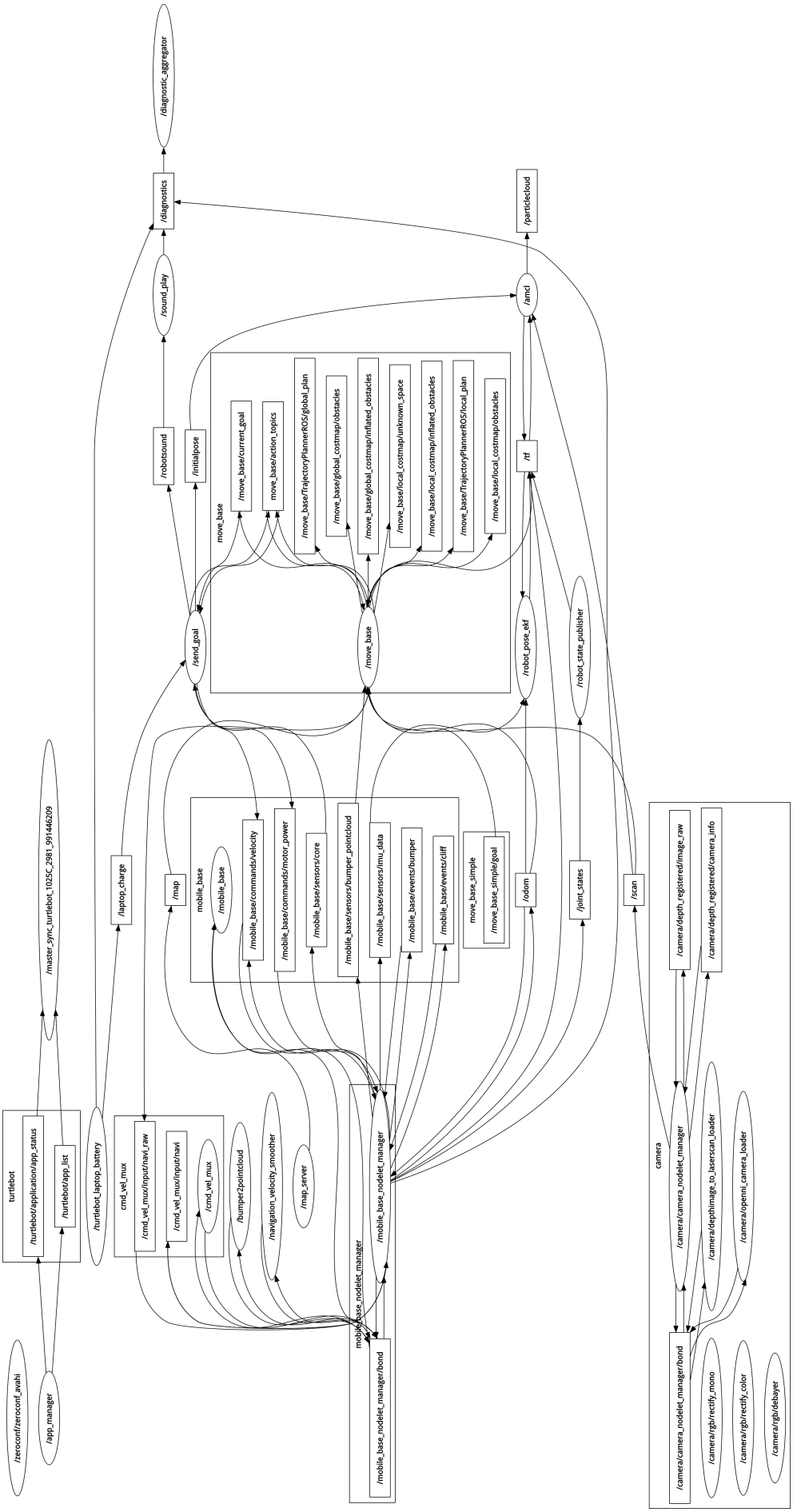
http://www.iheartrobotics.com/2010/12/limitations-of-kinect.html

spectrum.ieee.org

Index des illustrations

Annexes

Annexe 1 : Graphe d'interactions entre nodes



Annexe 2 : Code du programme principal : extraits du fichier .py

```
class NavGoal(object):

    def __init__(self, time_limit): #Initialises nav's attributes

    def _init_pub_sub_action(self): #Initialises publishers and subscribers

    def _def_position_and_velocity(self): #Initialises speeds, point A and B's
positions, and dock station position

    def play_sound(self, sound_num): #Text to Speech module

    def start_robot(self):
        print 'Moving backwards'
        if self.check_status() == True :
            for j in range(50):
                self.force_move.publish(self.command_back)
                rospy.sleep(0.1)
            print 'Turtlebot has moved backwards'
            print 'Turning'
            for j in range(10):
                self.force_move.publish(self.command_spin)
                rospy.sleep(0.1)
            self.docking = False
            print 'Starting position reached'
        else:
            print 'Sorry, Turtlebot has to dock : battery is weak'


    def cycles(self, nbr_cycles_to_do):
        if nbr_cycles_to_do > 0 : #user can select an infinite number of cycles by
giving the parameter -1
            rospy.loginfo("Turtlebot starting %s cycles. Time limit = %s",
nbr_cycles_to_do, self.stop_time - self.start_time)
            nb_cycles_limit = nbr_cycles_to_do
        else :
            rospy.loginfo("Turtlebot starting an infinite number of cycles. Time
limit = %s", self.stop_time - self.start_time)
            nb_cycles_limit = 1

        #start cycles
        while self.nbr_cycles_done < nb_cycles_limit :


            rospy.loginfo("Turtlebot moving to A")

            self.client.send_goal(self.obj)
            check_time = self.start_time
            while self.client.get_state() != GoalStatus.SUCCEEDED :
                current_time = rospy.get_time()
                if current_time - check_time > 10 :
                    check_time = current_time
                    self.dist()
                ok = self.check_status()
                if not ok :
                    break
            if not ok :
                break
            self.arrival_state('Turtlebot arrived at goal A in ' +
str( rospy.get_time() - self.start_time ) + 's' )


            rospy.loginfo("Turtlebot is going to goal B")
```

```
            self.client.send_goal(self.obj2)

            while self.client.get_state() != GoalStatus.SUCCEEDED :
                current_time = rospy.get_time()
                if current_time - check_time > 10 :
                    check_time = current_time
                    self.dist()
                ok = self.check_status()
                if not ok :
                    break
            if not ok :
                break
            self.arrival_state('Turtlebot arrived at goal B in ' +
str( rospy.get_time() - self.start_time ) + 's' )


            self.nbr_cycles_done+=1
            rospy.loginfo("Nombre d'allers effectues : %s", self.nbr_cycles_done)
            if nbr_cycles_to_do == - 1 : #if the user selected an undetermined number
of turns
                nb_cycles_limit = nb_cycles_limit +1 #put the cycle limit higher for
each cycle achieved



    def end(self):
        if not self.bumper_status_ok() :
            rospy.loginfo("exiting : bumpers have been hit too many times")
            self.en_dis_able_motors.publish(0)
            while not rospy.is_shutdown() :
                self.play_sound(RESCUE)
                rospy.sleep(12.0)
        else:
            rospy.loginfo("Robot going near dock")
            rospy.loginfo("Laptop battery charge : %s %%",
self.pc_battery_info.percentage)
            rospy.loginfo("Turtlebot battery charge : %s V",
float(self.sensor_msg.battery)/10.0)
            rospy.loginfo("Turtlebot was bumped : %s times", self.times_hit_bumpers)
            self.client.send_goal(self.start)
            while self.client.get_state() != GoalStatus.SUCCEEDED:
                self.dist()
                rospy.sleep(5.0)
            rospy.loginfo("Turtlebot arrived to start position.")


    def backup(self, situation):
        global BUMPED, DROPPED, CLIFF
        debug_message = {BUMPED : "Turtlebot hit something", DROPPED : "Turtlebot's
wheel dropped", CLIFF : "Cliff detected"}
        rospy.loginfo(debug_message[situation])
        self.en_dis_able_motors.publish(0)
        self.save_goal = self.current_goal
        self.take_picture()
        self.play_sound(situation)
        rospy.sleep(1.0)
        self.client.cancel_all_goals()
        rospy.sleep(1.0)
        self.ok = True
        self.times_hit_bumpers +=1
        self.en_dis_able_motors.publish(1)
        for h in range(5) :
            self.force_move.publish(self.command_back)
        for i in range(31) :
```

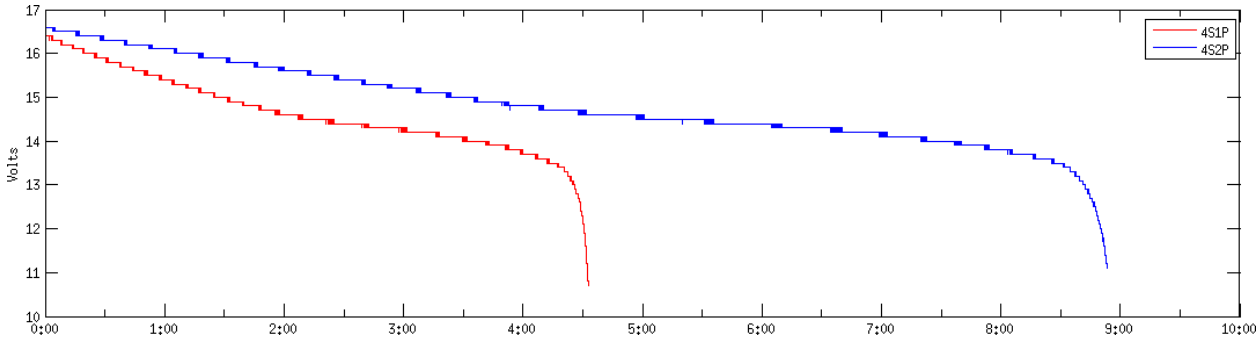
```
            if self.ok == True :
                self.force_move.publish(self.command_spin)
                rospy.sleep(0.1)
            else :
                break
        for k in range(50):
            if self.ok == True :
                self.force_move.publish( self.command_forward)
                rospy.sleep(0.1)
            else :
                break
        if self.ok == True : rospy.loginfo("Backup done. Pursuing goal.
Times_hit_bumpers = %d", self.times_hit_bumpers)
        else : rospy.loginfo("Interrupted by SensorCallback")
        if self.save_goal.pose.position.x == self.start.target_pose.pose.position.x :
            self.client.send_goal(self.start)
            rospy.sleep(1.0)
        elif self.save_goal.pose.position.x == self.obj.target_pose.pose.position.x:
            self.client.send_goal(self.obj)
            rospy.sleep(1.0)
        elif self.save_goal.pose.position.x == self.obj2.target_pose.pose.position.x:
            self.client.send_goal(self.obj2)
            rospy.sleep(1.0)


    def update_sensorsCallback(self, data_received):
        global BUMPED, DROPPED, CLIFF
        self.sensor_msg.bumper = data_received.bumper
        self.sensor_msg.wheel_drop = data_received.wheel_drop
        self.sensor_msg.cliff = data_received.cliff
        self.sensor_msg.charger = data_received.charger
        self.sensor_msg.battery = data_received.battery
        if self.sensor_msg.battery<135 :
            if self.info_printing_allowed:
                rospy.loginfo("Warning ! Turtlebot : low battery level : %s V",
self.sensor_msg.battery/10.0)
                self.info_printing_allowed = False
                self.batter_weak = True
        if self.docking == False:
            if self.sensor_msg.bumper != 0 :
                self.situation = BUMPED
                self.ok = False
            if self.sensor_msg.wheel_drop !=0 :
                self.situation = DROPPED
                self.ok = False
            if self.sensor_msg.cliff !=0 :
                self.situation = CLIFF
                self.ok = False


    def update_pc_battCallback(self, data_received): #Receives PC battery's status

    def charge_batteries(self): #Check batteries' status and waits for them to be
full

    def arrival_state(self, goal): #Displays batteries' data and number of time the
robot has been bumped

    def check_status(self): #Checks if robot is in safe situation (true or false)
        if self.battery_weak :
            self.client.cancel_all_goals()
            return False
        if not self.bumper_status_ok():
            return False
```

```
        if rospy.get_time() > self.stop_time :
            rospy.loginfo("TIME LIMIT EXCEEDED : %f > %f", rospy.get_time(),
self.stop_time)
            self.client.cancel_all_goals()
            return False
        if  self.ok == False :
            self.backup(self.situation)
            return True
        else : return True


    def update_feedbackCallback(self, data_received): #Receives robot's position
coordinates


    def update_current_goalCallback(self,data_received): #Receives current
distation's position

    def dist(self): #Displays current distance (euclidian) to the goal

    def bumper_status_ok(self): #Checks that the robot hasn't been bumped more than
11 times

    def update_picCallback(self, data): #Receives picture from kinect

    def take_picture(self): # Save picture

    def godock(self):
        self.docking = True
        rospy.loginfo("Turtlebot docking")
        docking = actionlib.SimpleActionClient('dock_drive_action',
AutoDockingAction)
        while not docking.wait_for_server(rospy.Duration(5.0)):
            if rospy.is_shutdown(): return
            print 'Action server is not connected yet. still waiting...'
        print 'Action server is connecting. Processing...'
        docking_goal = AutoDockingGoal();
        docking.send_goal(docking_goal)
        print 'Goal: Sent.'
        docking.wait_for_result()
        print docking.get_state() #ajouter un dictionnaire
        return True


def main(nb_turn, duration):
    u = NavGoal(float(duration))
    try:
        'Turtlebot will start in...'
        for i in range(5):
            print 5-i
            rospy.sleep(1.0)
        #back away from dock
        print 'STARTING'
        u.start_robot()
        print 'robot has started'
        #number of cycles and time limit
        print 'starting cycles'
        u.cycles(int(nb_turn))
        #go in front of dock
        u.end()
        #dock
        if u.bumper_status_ok() :
            if u.godock() :
                rospy.loginfo("Robot docked after %s cycles", u.nbr_cycles_done)
```

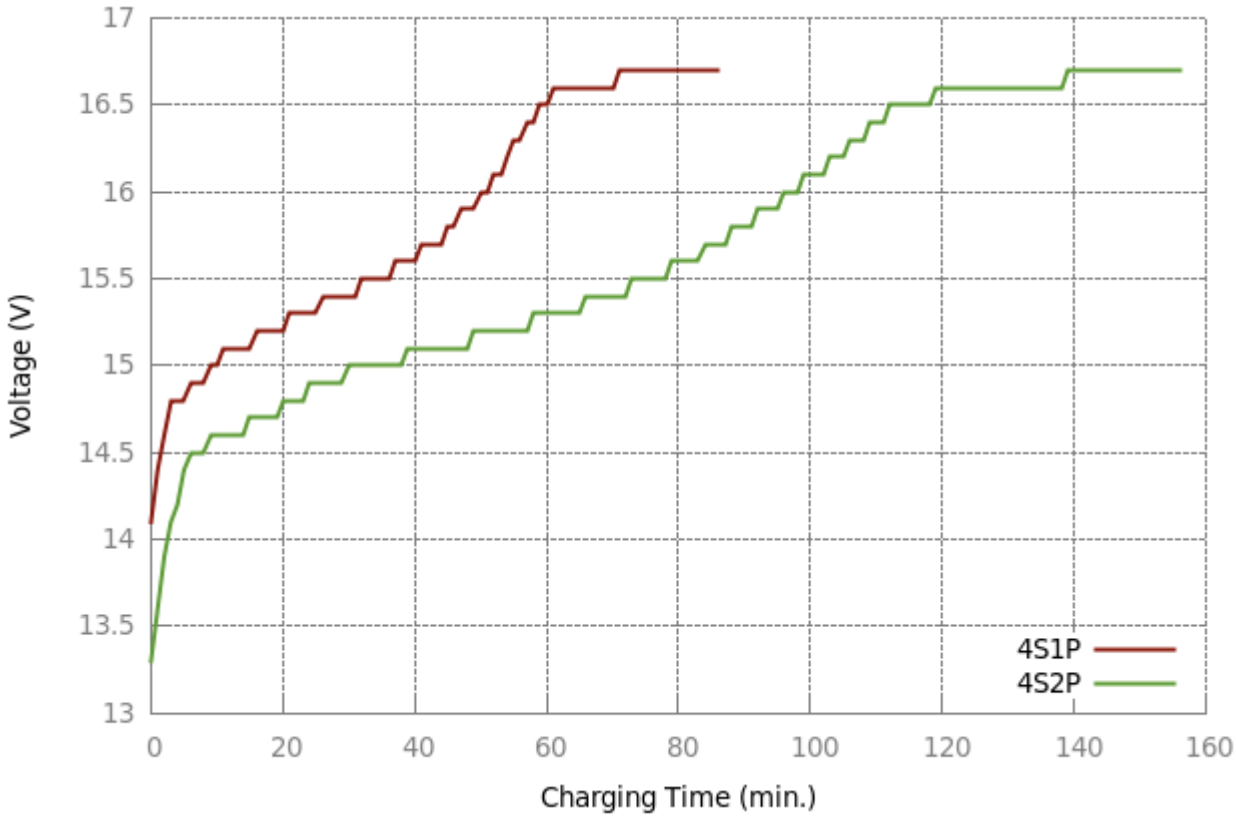
```
                #if batteries are really weak, wait for full charge
                if u.battery_weak :
                    rospy.loginfo("Batteries are week.")
                    u.charge_batteries()
                    u.nbr_cycles_done = 0
                rospy.sleep(2.0)
        u.en_dis_able_motors.publish(1)
        print 'Circuit done. Exiting'
    except KeyboardInterrupt :
        print "Shutting down"

if __name__ == '__main__' :
        if len(sys.argv) < 3 :
            print 'fsm.py nb_turns(integer) max_duration(seconds)'
            print 'hint : if you want to do an undetermined number of loops with a
defined duration use nb_turns = -1 [NOT IMPLEMENTED YET]'
        else :
            main(sys.argv[1], sys.argv[2])
```

Annexe 3 : fichier .yaml pour la configuration de la carte

```
image: only_SED_offices.png
resolution: 0.05
origin: [0.0, 0.0, 0.0]
negate: 0
occupied_thresh: 0.65
free_thresh: 0.196
```



Annexe 4: Caractéristique de décharge de la batterie (en rouge)



Annexe 5: Profil de charge de la batterie (en rouge)

Résumé/ Sum up

Dans le cadre des Financements d'Equipement d'Excellence (Equipex), le Service d'Expérimentation et de Développement de l'Inria Grenoble étudie la possibilité d'ajouter un noeud mobile à un réseau de capteurs. La solution retenue pour cela est l'utilisation d'un robot Turtlebot qui naviguera dans les couloirs d'Inria où seront, très prochainement, installés 700 capteurs supplémentaires.

Au cours de ce stage, j'ai utilisé les outils disponibles sur le Turtlebot, la documentation sur son système d'exploitation nommé ROS, le langage python et c++ ainsi que les travaux précédemment effectués sur le sujet pour permettre à un robot Turtlebot d'éxécuter le scenario suivant : se déplacer d'un point B à un point A un nombre de fois paramétrables tout en évitant les obstacles et en gérant sa puissance.

A ce stade du stage, des améliorations sont encore nécessaires sur le packaging du code réalisé ainsi que sur les applications fournies de base avec le Turtlebot.

*****

In the context of Senslab project, the Experimentation and Development Service of Inria Grenoble investigates the possibility to integrate a mobile node inside a sensors network. The selected solution to achieve it is to use a Turtlebot robot which will navigates into Inria's corridor where more sensors will, really soon, be installed.

During this internship, I used Turtlebot's available tool, Robot Operating System's documentation, c++ and python programming languages, and previous work on the topic to bring up a Turtlebot to be able to move from a point A to a point B a parametrised number of times with obstacles avoidance and power management.

At this stage of the internship, some enhancements are still required on code packaging as some debugging on provided ROS softwares.