Université Paul Sabatier Toulouse III

Master 2 Intelligence Artificielle et
Reconnaissance des Formes
Master 2 Robotique : Décision et Commande

Navigation Autonome de Robot Mobile

Manuel Développeur

Auteurs :
Pierre Beauhaire
Hugo Brefel
Salah Eddine Ghamri
Sylvain Guillaume
Luc Rubio

Tuteurs :
Michaël LAUER
Frédéric LERASLE
Michel TAIX

Mars 2018

Suivi du document

Nom du document	Version	Date de création
Navigation Autonome de Robot Mobile - Manuel développeur	1	22/03/2018

Auteurs du document

Rédaction	Relecture
Pierre Beauhaire Luc Rubio	Pierre Beauhaire Luc Rubio Sylvain Guillaume Hugo Brefel Salah Eddine Ghamri

Validation	Nom	Date	Visa

Liste de diffusion

Le manuel développeur est diffusé à l'ensemble des clients.

Table des matières

1 Introduction

Ce manuel donne les informations nécessaires à l'utilisation et à l'évolution du projet "Navigation autonome d'un robot mobile" du Master 2 IARF et RODECO. Celui-ci permet, à ce jour, d'effectuer une navigation autonome d'un Turtlebot dans un environnement connu, contenant des amers dont leur position et leur orientation sont connues. Ces indices visuels, visualisés par la kinect et le capteur RGB-D, permettent de supprimer les erreurs systématiques de localisation effectuée par les capteurs proprioceptifs embarqués – de l'odométrie.

La navigation s'effectue entre une position initiale et une position finale :

Scénario	A	B	C	D
Départ	Position initiale inconnue	Position initiale inconnue	Position initiale inconnue	Position initiale inconnue
Localisation	Sans	Avec	Sans	Avec
Obstacles	Sans	Sans	Avec	Avec
Arrivée	Position finale	Position finale	Position finale	Position finale

Ces scénarios sont :

A
Init
    Traitements sur la carte
    Génération du graphe connexe représentant les lieux accessibles
    Calcul d'une trajectoire
Suivi de trajectoire jusqu'à la position finale

B
Init
    Traitements sur la carte
    Génération du graphe connexe représentant les lieux accessibles
    Calcul d'une trajectoire
Recherche d'un amer en tournant sur lui-même
Si aucun amer en vue
    Message d'erreur et arrêt du programme
Si un amer est trouvé
    Asservissement sur l'amer
Suivi de trajectoire jusqu'à la position finale avec relocalisation sur les amers rencontrés

C
Init
    Traitements sur la carte
    Génération du graphe connexe représentant les lieux accessibles
    Calcul d'une trajectoire
Recherche d'un amer en tournant sur lui-même
Si aucun amer en vue
    Message d'erreur et arrêt du programme
Si un amer est trouvé
    Asservissement sur l'amer
    Suivi de trajectoire jusqu'à la position finale
Si un obstacle est rencontré
    Le robot s'arrête
    L'obstacle est ajouté sur la carte
    Le robot tourne de 180°
    Une nouvelle trajectoire est calculée
Suivi de trajectoire jusqu'à la position finale

D Init
    Traitements sur la carte
    Génération du graphe connexe représentant les lieux accessibles
    Calcul d'une trajectoire
Recherche d'un amer en tournant sur lui-même
Si aucun amer en vue
    Message d'erreur et arrêt du programme
Si un amer est trouvé
    Asservissement sur l'amer
    Suivi de trajectoire jusqu'à la position finale avec relocalisation sur les amers rencontrés
Si un obstacle est rencontré
    Le robot s'arrête
    L'obstacle est ajouté sur la carte
    Le robot tourne de 180°
    Une nouvelle trajectoire est calculée
Suivi de trajectoire jusqu'à la position finale avec relocalisation sur les amers rencontrés

Étant donné que le robot doit voir des AR-Codes pour pouvoir se relocaliser, on considère que suffisamment d'amers auront été placés dans l'environnement de navigation, afin que quelque soit l'endroit où se trouve le robot, il puisse visualiser au moins un amer en tournant sur lui-même.

Nous avons remarqué lors de nos expérimentations sur les TurtleBot qu'un des TurtleBot (le TurtleBot 3) avait des roues folles légèrement plus petites que celles des autres robots. Cette différence permet au robot d'éviter de frotter au sol lors de ces mouvements rotatifs, et lui permet de se déplacer de manière beaucoup plus fluide et plus stable. Pour les tests qualitatifs et les démonstrations, il sera donc plus intéressant d'utiliser ce TurtleBot, ou au moins de démonter ses roues et de les intégrer sur le TurtleBot utilisé.

De plus, les conditions d'illumination de l'environnement peuvent gêner la navigation du TurtleBot. En effet, une lumière trop importante peut entraîner une mauvaise visualisation voire une incapacité à visualiser les AR Codes par la Kinect.

2 Navigation avec amers 2D dans un environnement connu

La tâche de navigation autonome se découpe en sept briques :

Traitements sur la carte de l'environnement qui effectue des traitements sur la carte.

Recherche d'amers et Asservissement qui permet de rechercher un amer en faisant tourner le robot sur lui-même, et de s'asservir dessus, afin d'assurer une relocalisation précise du robot dans l'environnement.

Génération de trajectoire qui permet de générer et afficher le graphe de connexité pour se déplacer dans l'environnement ainsi que la trajectoire du robot.

Déplacement qui réalise un suivi de trajectoire d'un point Start à un point Goal.

Relocalisation qui permet au robot de connaître sa position de manière précise dans l'environnement, notamment durant le déplacement.

Fonctions Annexes qui regroupe des fonctions utiles pour le développement.

Main qui décrit la machine à états utilisé par le robot et appelle tous les autres blocs.

Le fichier *navigation.launch* regroupe toutes les informations essentielles au projet. Il permet de lancer plusieurs noeuds (ici des fonctionnalités) nécessaires à la navigation du robot avec les bons paramètres.

2.1 Travail réalisé

L'ensemble des fonctionnalités (Traitements sur la carte, recherche d'amers, asservissement, génération de trajectoire, déplacement) fonctionnent, ont été testées et intégrées dans notre main final. Le robot peut donc se déplacer dans son environnement grâce à l'odométrie, mais aucune relocalisation n'est effectuée durant son déplacement.

Malheureusement, le déplacement avec relocalisation (filtre de Kalman) ne fonctionne pas. Deux solutions peuvent être envisagées :

- Il manque le passage des coordonnées des points générés pour la trajectoire dans le nouveau repère donné par Kalman lors de la relocalisation. Il faudrait donc faire ce calcul pour faire fonctionner la relocalisation.
- Il faudrait sinon recréer un script Kalman pour pouvoir rediriger l'entrée d'odométrie, et pour que le topic */odom* n'influe plus sur le repère de la carte (lié à l'environnement).

2.2 Traitements sur la carte et génération de trajectoire

Pour pouvoir visualiser la carte de l'AIP sur RViz, nous avons d'abord construit une carte en faisant se déplacer le robot. La carte est sauvegardée dans le dossier *map* de notre projet. Elle est lancée par le fichier *navigation.launch*.

Pour visualiser de manière précise le déplacement du TurtleBot sur RViz, il faut tout d'abord effectuer quelques traitements d'images : dilatation des obstacles de la taille du robot pour éviter des collisions, association du robot à un point. Ces traitements sont effectués dans le fichier $Image\_Processing$, ce qui met à jour la carte sur RViz.

Le fichier Map_Gui permet d'afficher toutes les informations liées aux marqueurs : repère lié au marqueur, meilleur point de visu d'un marqueur (points rouges sur RViz), cônes de visibilité..., et la classe Set_Marker permet de les récupérer depuis le fichier *navigation.launch*.
Toutes les informations liées à la carte comme sa taille, sa résolution, ou la structure de la carte peuvent être récupérées dans le fichier Map_node.

2.3 Recherche d'amer et Asservissement

Avant tout déplacement jusqu'au point Goal, le robot doit d'abord se localiser précisément dans l'environnement. Pour cela, on va utiliser la détection d'amers.

2.3.1 Fonctionnement

Afin de détecter les amers dans la scène, nous utilisons la brique ROS : ar_track_alvar de la librairie open source Alvar pour la détection de marqueurs AR. Cette brique utilise le nuage de points 3D associé à la couleur pour identifier un marqueur. L'information de profondeur permet à l'algorithme de mieux repérer les plans des marqueurs dans la scène. Elle retourne l'identifiant, la position et l'orientation du marqueur dans le repère de la $/camera\_rgb\_optical\_frame$. La librairie fournie 55 marqueurs AR et la possibilité d'étendre cette liste facilement. Dans notre cas, nous utilisons uniquement le nœud qui permet de lire les marqueurs un par un (individual tags). En effet il existe un autre nœud qui permet d'estimer une position en observant un ensemble de marqueurs (multi-tag bundles).

2.3.2 Mise en place physique

Lors de notre projet nous avons utilisé des marqueurs de taille 10×10 cm. On place leur milieu à 31 cm du sol afin que l'axe caméra-marqueur soit le plus parallèle au sol possible.

2.3.3 Sorties

Le nœud publie les informations de détection dans des messages de type $ar\_track\_alvar\_msgs :: AlvarMarkers$ sur le topic $/ar\_marker\_pose$. Il publie aussi la TF associée au marqueur vu dans le repère de la caméra.

2.3.4 Performances et évolutions

L'orientation du marqueur trouvée par ar_track_alvar doit être : $\vec{z}$ la normale et $\vec{x}$ vers le haut.

La détection des marqueurs de 10×10 cm s'effectue jusqu'à 3 m avec une précision de 3 cm et jusqu'à 60 degrés d'angle à 3 degrés près. Un filtre de Kalman a été intégré pour fusionner les données odométriques et les observations, dans le but de réduire l'erreur.

2.3.5 Comportement du robot

Le robot va donc tourner sur lui-même jusqu'à détecter un amer dans son champ de vision. Cette fonctionnalité est implémentée dans les fichiers *Recherche* et $Recherche\_Main$. La classe *Recherche* n'a pas été appelée directement dans le *main* car les publications et écoutes de topic ne fonctionnaient pas. Une classe $Recherche\_Main$ a donc été créée pour pallier à ce problème.

2.4 Génération de trajectoire

Pour pouvoir suivre une trajectoire, le robot doit d'abord générer cette dernière. Pour cela, on utilise la classe $Point\_Cloud$ qui va nous permettre de générer les nœuds du graphe, représentant les points de passage possibles pour le robot. Ces nœuds sont des structures de données représentées dans la classe $Target$. La classe *Line* est utilisé par la classe *Graph* pour relier les nœuds entre eux (en prenant en compte l'environnement de travail, et donc les obstacles). La classe *Graph* effectue ensuite un algorithme A* sur le graphe ainsi généré pour déterminer la trajectoire optimale du robot pour atteindre le point Goal. On utilise également la classe Set_Start_Point afin d'initialiser le point de départ de la trajectoire en fonction de la position du robot, et la classe *Goal* pour récupérer la position et l'orientation rentrée par l'utilisateur pour sélectionner un point d'arrivée.

2.5 Déplacement

Maintenant que la trajectoire du robot est déterminée, on va envoyer la commande de mouvement à la base mobile du robot. Pour cela, on utilise une loi de commande décrite dans le fichier $Follow\_Path$. Ce fichier intègre également une fonctionnalité de détection d'obstacles.
Lors du déplacement, il est également possible que le robot entre en collision avec un obstacle. Un arrêt d'urgence a donc été implémenté dans la classe $Bumper$.

2.6 Relocalisation

Les amers rencontrés pendant le déplacement du robot sont utilisés afin de permettre au robot de se relocaliser précisément dans l'environnement. Pour cela, nous utilisons le module *alvar* (cf. Manuel Utilisateur) qui permet de connaître la position du robot par rapport aux amers, ainsi que l'identifiant de l'amer observé. On doit ensuite passer les position et orientation du robot dans le repère monde. On utilise ainsi la classe $Vo\_Broadcast$ qui publie ensuite cette situation au module $Kalman$ (cf. Manuel Utilisateur).

2.7 Fonctions annexes

La classe $Toolbox$ fournit des outils mathématiques utiles, comme la distance entre 2 points du graphes, le produit scalaire entre 2 vecteurs ou encore l'angle orienté formé par 2 vecteurs.

2.8 Main

Le *main* de notre projet est implémenté sous forme de machine à états. Il est décrit dans le fichier *Main*.