Université Paul Sabatier

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

Manuel développeur

Navigation Autonome de Robot Mobile

Auteurs:
Thibaut Aghnatios
Marine Bouchet
Bruno Dato
Tristan Klempka
Thibault Lagoute

Tuteurs:
Frédéric Lerasle
Michaël Lauer
Michel Taix

13 mars 2017

Suivi du document

Nom du document	Version Majeure	Version Mineure	Date de création	Dernière version
Manuel développeur	A	6	13/03/2017	1/04/2017

Auteurs du document

Rédaction	Intégration	Relecture	Validation Interne
Equipe	??	??	??

Validation du document

Validation	Nom	Date	Visa

Liste de diffusion

Le rapport du projet est diffusé à l'ensemble des clients et des intervenants externes aux projets.

Historiques de révision

Version	Modification apportée	Auteur	Date
A.0	Création du document	Bruno Dato	13/03/2017
A.1	Sections 2.1, **??** et 1	Marine Bouchet	27/03/2017
A.2	Section **??**	Tristan Klempka	28/03/2017
A.3	Section 2.5	Thibaut Aghnatios	30/03/2017
A.4	Sections **??** et 2.6	Bruno Dato	30/03/2017
A.5	Section 2.6	Bruno Dato	31/03/2017
A.6	Section 1	Bruno Dato et Marine Bouchet	1/04/2017

Contents

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 connus. Ces indices visuels, visualisés par la kinect, capteur RGB-D, permettent de supprimer les erreurs systématiques de la localisation des capteurs proprioceptifs embarqués – de l'odométrie.

La navigation s'effectue entre la position initiale, dont la position est connue ou dans la zone de visibilité d'un amer, et une position finale :

Scénario	A	B
Départ	Position initiale absolue	Position initiale inconnue mais dans une zone de visibilité d'un marqueur
Arrivée	Position finale	Position finale

Ses scenarii sont (B étant une extension de A) :

B Si pas de position en entrée
  Le robot effectue une recherche de marqueur dans son champs visuel
  Tant que pas de marqueur en vue
    Le robot tourne sur lui même
  Il effectue une autre recherche de marqueur dans son champs visuel

A Tant que le but est plus loin que le prochain marqueur dans sa direction
  Le robot se déplace jusqu'à la zone de visibilité du marqueur le plus
  ... proche dans la direction du but
  Le robot effectue une recherche de marqueur dans son champs visuel
  Tant que pas de marqueur en vue
    Le robot tourne de sur lui même
    Il effectue un autre recherche de marqueur dans son champs visuel
  Le robot se déplace jusqu'au but

Étant donné que la connaissance sur la visibilité d'un marqueur n'es pas encore opérationnelle, on considérera que le robot se trouve toujours dans une zone visibilité. On placera alors le robot dans une zone de visibilité à chaque début de navigation en veillant aussi à donner une position initiale correcte.

2 Navigation avec amers 2D dans un environnement connu

La tâche de navigation autonome se découpe en cinq briques :

Détection qui permet de repérer les amers dans le champs de vision quand ils se présentent

Localisation qui permet de savoir où se trouve le robot, en utilisant soit les données d'odométrie interne, soit les données d'observation

Trajectoire qui permet de se déplacer en boucle fermée d'un point A à un point B

Commande qui permet de suivre la trajectoire

Visibilité qui permet de générer une carte contenant les positions des amers et leurs orientations ; et de renvoyer à partir de cette carte le nombre d'amers visibles

Supervision qui gère le bon déroulement de la tâche, le déplacement du point de départ au point d'arrivée

2.1 Solution mise en place

L'architecture globale de notre produit est décrite par le schéma suivant :

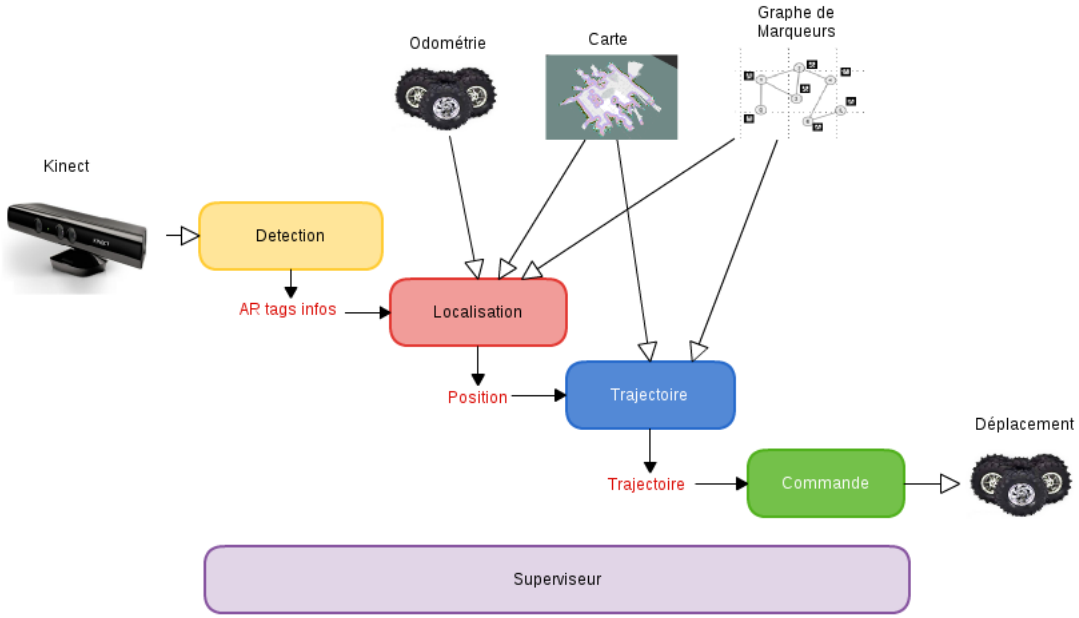

Figure 1: Architecture actuelle

2.2 Détection

Fonctionnement

Afin de détecter les amers dans la scène, nous avons utilisons à la brique ROS : *ar_track_alvar* qui utilisant 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 noeud qui permet de lire les marqueur un par un (individual tags). En effet il existe un autre noeud qui permet d'estimer une position en observant un ensemble de marqueurs (multi-tag bundles).

Mise en place physique

Lors de notre projet nous avons utilisé des marqueurs de taille 16×16 cm. On place leur milieu à 31 cm du sol afin que l'axe caméra-marqueur soit le plus parallèle au sol possible.

Utilisation

Le noeud de détection est lancé indépendamment des autres noeuds du système. Il s'initialise dans *localisation.launch* avec les paramètres suivant :

- *marker_size* : largeur (cm) des marqueurs AR utilisés
- *cam_image_topic* : topic ROS du flux d'images de la caméra

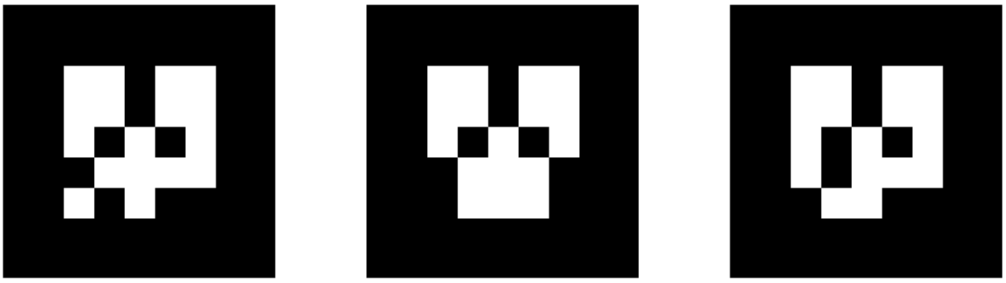

Figure 2: Exemple de marqueurs AR. Les valeurs 3, 4 et 5 sont codés avec ces marqueurs.

- cam_info_topic : topic ROS des infos propres à la caméra – c'est ici que les données de calibrage sont récupérées
- $output\_frame$: repère dans lequel sera exprimé le résultat de la détection – $/camera\_rgb\_optical\_frame$ dans notre cas

Outputs

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

Performances et évolutions

L'orientation du marqueur trouvé par ar_track_alvar doit être : $\vec{z}$ la normale et $\vec{x}$ vers le haut. Il arrive, pour des problèmes inconnus (le réseau ?) que ce repère ne soit pas correctement capturé, avec par exemple, $\vec{y}$ vers le haut. Le robot, à la fin de la localisation se retrouve alors couché. Il faudrait alors comprendre d'où vient le problème ou sinon, ne pas prendre en compte ces orientations aberrantes.

La détection des marqueurs de 16×16 cm s'effectue jusqu'à 2 m 25 avec une précision de 2 cm et jusqu'à 45 degrès d'angle à 5 degrès près. Il serait intéressant, à l'avenir, d'approfondir les résultats obtenus avec d'autres tailles si l'erreur est trop aléatoire, ou alors, intégrer un filtre de Kalman pour fusionner les données odométriques et les observations.

2.3 Localisation

Utilisation

Le noeud de localisation est lancé grâce à un ficher ROS de launch. Cependant, contrairement au noeud de détection, ce noeud ne se lance pas indépendamment des autres. Il est lancé avec l'ensemble du système de navigation avec la commande suivante : *roslaunch navigation.launch*

Ce fichier configure et lance le système de navigation qui comprend le noeud de localisation. On trouve dans ce fichier des paramètres propre à la localisation pouvant être configurés.

- map_file : Chemin de la carte utilisée pour la navigation. Notre système utilise le noeud ROS map_server pour le chargement de la carte.
- $marker\_tf\_publisher\_X$: L'utilisateur renseigne ici, pour chaque marqueur, la transformée qui sera publiée pour le repérer sur la carte. L'orientation est renseignée avec un quaternion.

Rappels

Avant toute chose, il est important de rappeler que quelque soit la situation :

- */map → /odom → /base_link* (`cf.http://www.ros.org/reps/rep-0105.html`)
- Le repère */map* doit être correctement placé en position et en orientation puisque toute les transformations vont se faire à partir de celui-ci.
- /odom est le repère relatif de l'odométrie, assimilé certain, du robot et est placé au départ sur /map. Il est en quelque sorte le point de départ d'un déplacement. Etant donné que ce repère drift au cours du déplacement du robot, il peut être éronné. La localisation via un amer connu permet de placer le repère /odom au bon endroit, et de réinitialiser l'odométrie, ce qui place /base_link au même endroit. C'est une localisation absolue.
- chaque frame peut avoir plusieurs fils mais qu'un seul parent
- la transformée entre deux frames est décrite par deux attributs :
  - *m_origin* : Vector3 de translation
  - *m_basis* : Matrix3x3 de rotation

2.3.1 *localisation_node.cpp*

Ce nœud permet d'envoyer la nouvelle localisation du robot s'il détecte un marqueur.

Traitement

La recherche se déclenche uniquement et une seule fois quand la commande de haut niveau publie un <std_msgs::Empty> sur le topic */nav/HLC/askForMarker*.
Dans le cas ou un marqueur est visible, voici le processus :

- On sait à priori : /map → /odom → /baselink → /camera_rgb_frame → /camera_rgb_optical_frame
- A l'arrivée d'un marqueur, on a : /camera_rgb_optical_frame → /ar_marker_ID
- On remonte l'information : /baselink → /camera_rgb_frame → /camera_rgb_optical_frame → /ar_marker_ID
- On sait donc : /ar_marker_0 → /baselink, qui est égal à /marker_0 → /baselink
- Sachant /map → /marker_0, on sait /map → /baselink = /map → /odom

Cette transformation est envoyée sur */new_odom* que l'on publie avec le publisher *odom_pub*.
On publie également l'identifiant du marqueur vu sur */nav/loca/markerSeen* sous un <std_msgs::Int16>.
Dans le cas où on ne voit rien on publie : -1

Initialisation

Différents champs peuvent être paramétiser :

- *DEBUG* : Active ou non les sorties de debug. Si cette option est activée, les transformations intermédiaires sont publiées.
- *TIMEOUT_AR_DETEC* : Temps d'attente maximum pour la réception d'un marqueur. Ce paramètre est utile lors de l'utilisation en réseau. La latence provoque des retards dans la publication des transformées. Il est donc nécessaire de donner une marge de temps au système. Lors de l'utilisation en mode local cette valeur peut être faible.
- *NB_MARKER* : Nombre de marqueurs total dans la scène.

Une fois le lancement du noeud effectué, l'utilisateur peut observer son comportement avec des sorties console :

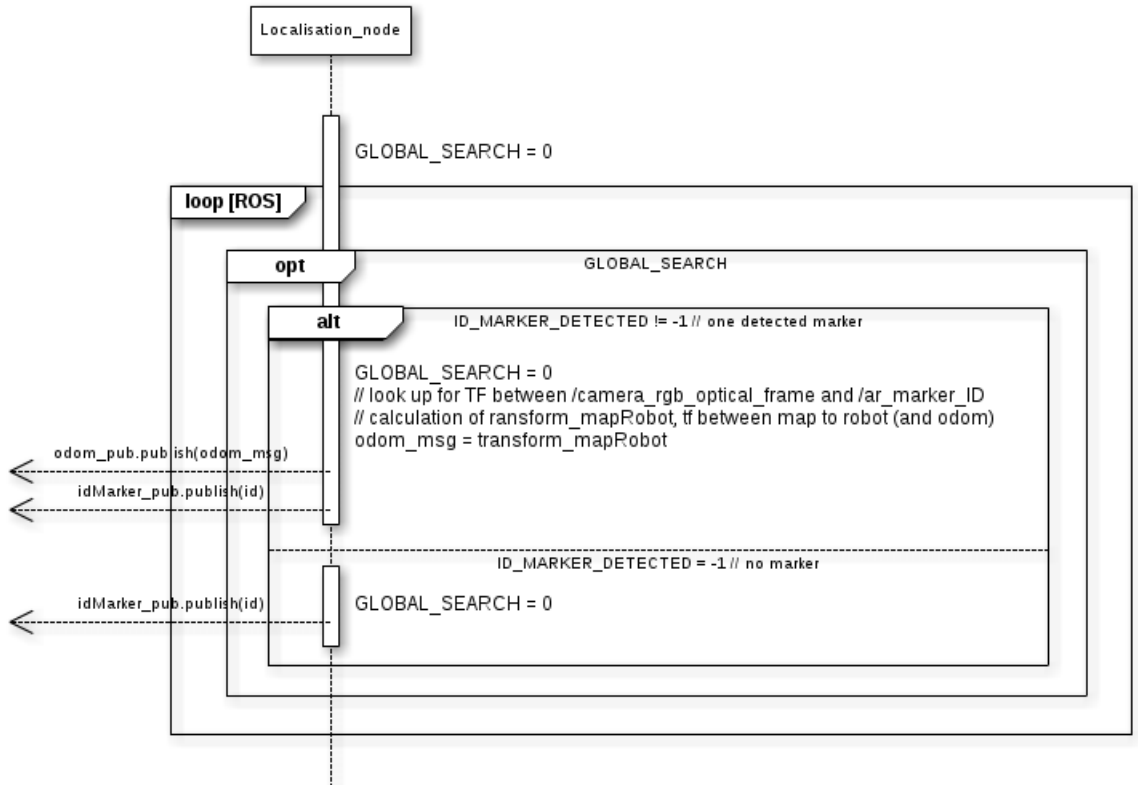

Figure 3: Traitement général lors de la localisation

- $GLOBAL\_SEARCH$: La recherche de marqueur a été demandée.
- $MARKERDETECTED : ID\_MARKER\_DETECTED$: Identifiant du marqueur détecté.
- $LOOKINGFORTF : tf$: Nom de la transformée attendue.

Améliorations

Il serait préférable de récupérer les TFs /baselink → /camera_rgb_frame et /camera_rgb_frame → /camera_rgb_optical_frame en dynamique plutôt qu'en statique, de même pour le nombre de marqueurs dans la scène.

2.3.2 *localisation_broadcaster_node.cpp*

Ce nœud permet de tout le temps broadcaster la transformée entre */map* et */odom*. Il souscrit à la $< geometry\_msgs :: Transform >$ /new_odom, qui contient la "nouvelle" position certaine du robot. Dès que celle-ci est publiée, */odom* actualise sa position et on réinitialise le $nav\_msgs/Odometry$: il devient notre nouveau référentiel.

2.4 Commande

Utilisation

Pour la version actuelle du projet, la plupart de la commande est effectué par une brique ROS. La brique *move_base* réalise la commande afin de suivre la trajectoire générée par la brique trajectoire. Cette brique est lancé dans un fichier ROS de launch.
roslaunch turtlebot_proj_nav navigation.launch

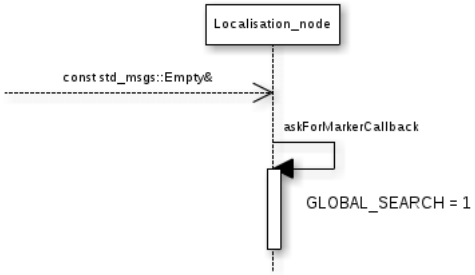

Figure 4: Callback de la demande de localision

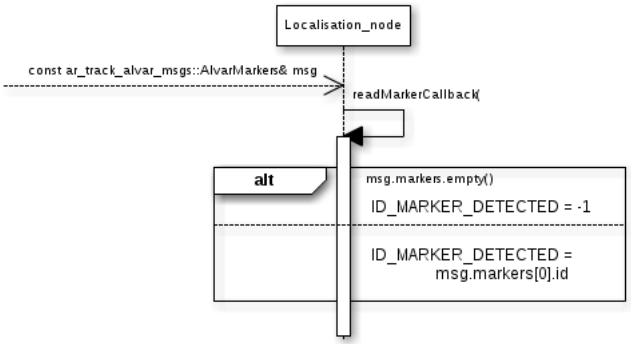

Figure 5: Callback du résulat de la détection

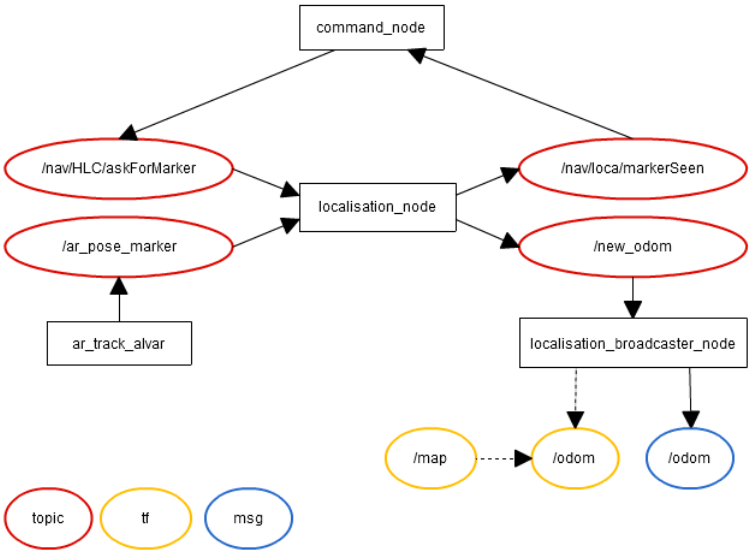

Figure 6: texte de la légende

La commande peut être configurée à l'aide du fichier XML : *move_base.launch.xml*. Ce fichier XML référence un ensemble de fichiers XML qui configure la commande. Une description exhaustive des paramètres de cette brique peut être trouvée à cette adresse :
http://wiki.ros.org/move_base
Lors de la recherche d'un marqueur, nous reprenons la main et c'est la node *commande_node* qui prend le relais. Celle-ci est également lancée dans *navigation.launch*.

Fonctionnement

Au début du projet une ébauche de commande a été réalisée. Celle-ci est simple, elle permet un déplacement en ligne droite et une rotation sur place. Cette commande est en boucle ouverte. Il peut être intéréssant de continuer ce travail afin de prendre en compte une correction odométrique notamment lors d'un évitement d'obstacle lorsque le système reprend la main sur le *navigation_stack*.
Cette ébauche ce trouve dans le fichier *commande_node.cpp*
Cette node est lancée dans le fichier ROS de launch *navigation.launch*

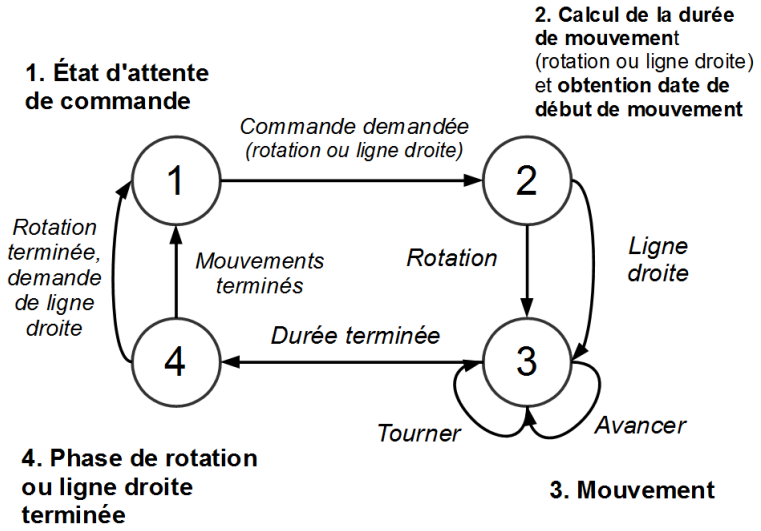

Figure 7: Schéma de la machine à états de la commande utilisée pour la recherche d'une balle et pour la recherche d'un amer

2.5 Visibilité

Le but de cette section est de créer et d'utiliser une carte de visibilité, une carte contenant l'ensemble des positions où le robot voit au moins un des amers, afin de retourner le nombre d'amers visibles pour une position donnée. L'intérêt de cette carte est de pouvoir suivre notre navigation d'amers en amers et de savoir si l'on se trouve dans un champ de visibilité pour lancer la détection et la localisation avec notre superviseur.

Nous expliquons ici comment la carte de visibilité est générée, à l'aide de *visib_pgmwriter_node.cpp*, et le mécanisme de la node *visib_pgmreader_node.cpp*, retournant le nombre d'amers visibles.

2.5.1 Génération de la carte de visibilité

Tout d'abord pour générer la carte de visibilité, il faut préalablement avoir une carte de l'environnement. Cette carte est créée à l'aide d'un logiciel de dessin ou en utilisant la cartographie disponible sur le Turtlebot *gmapping*.

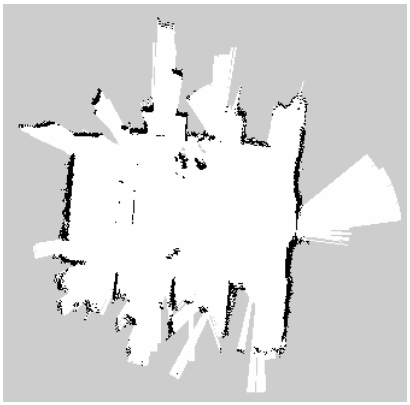

Figure 8: Cartographie de l'environnement avec le Turtlebot

Le node *visib_pgmwriter_node.cpp* ne doit pas être modifié, toutes les configurations se font directement dans le fichier *visib_init.cpp*. En effet, le node lance la fonction $Ecriture\_carte\_visib()$ qui crée dans un fichier au format PGM (Plain PGM : $http://netpbm.sourceforge.net/doc/pgm.html\#plainpgm$) l'ensemble des amers définis dans le *graph.xml* se trouvant dans le dossier */rsc*. Donc pour une taille de carte donnée et pour les configurations effectuées correctement dans *visib_init.cpp* expliquées par la suite, il suffit de modifier la position et l'orientation de nos amers dans le *graph.xml* (voir partie 2.6.2) pour que la nouvelle carte de visibilité soit générée automatiquement en lançant à nouveau notre node *visib_pgmwriter_node.cpp*.

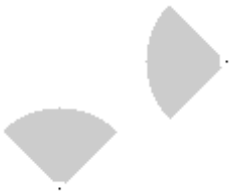

Figure 9: Exemple d'une carte de visibilité

Une fois notre carte générée, il faut pouvoir la visualiser sur Rviz et la superposer avec la carte de notre environnement. En lançant le *navigation.launch* notre carte de visibilité est publiée sur le topic *markers_visibility_map*. Il suffit de l'ajouter directement sur Rviz en ajoutant celle-ci et en se plaçant sur le topic en question. Le résultat obtenu sur Rviz est le suivant :

Au niveau de la configuration de *visib_init.cpp* dans la fonction $Ecriture\_carte\_visib()$, il faut modifier *int largeur=512; int hauteur=640;* pour que la carte ait toujours les mêmes dimensions que la carte de notre environnement. $intnbr\_amers = 2;$ pour prendre en compte le nombre d'amers contenus dans *graph.xml*. Ensuite dans la boucle for faisant la conversion de la position de mètre en pixels :

$for(a = 0; a < nbr\_amers; a++)$
{
. $x[a] = (x1[a] + 12.2)/0.05;$
. $y[a] = -(y1[a] + 13.8 - 32)/0.05;$
. $alphamax[a] = pi/4;$
}

Avant tout, il faut savoir qu'on a 3 repères différents.
- On a le repère du robot que l'on nomme R et possède la position en x et y du robot donc R

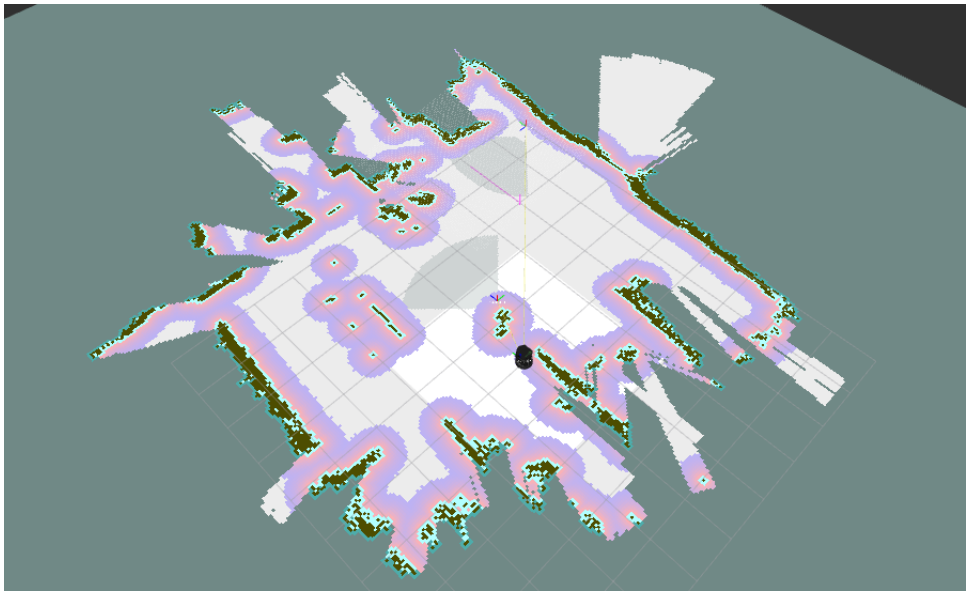

Figure 10: Superposition de nos cartes sur Rviz

(X,Y).
- Le repère de la carte de l'environnement note R' qui a pour origine (-12.2 ; -13.8) (dans notre cas pour une carte 512 x 640) donc R' (X+12.2, Y+13.8).
- Ensuite on a le repère de notre carte de visibilité noté Rp qui est positionné en haut a gauche de la carte (vecteur u dans le même sens que x et vecteur v opposé au vecteur y) c'est pourquoi on peut écrire Rp (X+12.2, -(Y+13.8-32)).

Explicitons maintenant les différentes valeurs utilisées : 0.05 correspond à la résolution qui se trouve dans le fichier de configuration .yaml dans */map*. Ici 12.2 et 13.8 correspond à l'origine de la carte pour que la superposition des cartes soit correctes. En effet, si on laisse 0.0 et 0.0, la carte se générera à la base du robot sur Rviz donc on aura un décalage lors de la superposition. C'est pourquoi dans le fichier *visib.yaml*, il faut aussi le configurer de cette manière pour notre cas :
$resolution$: 0.050000
$origin$: $[-12.200000, -13.800000, 0.000000]$

Donc, cela permet de passer du repère du robot au repère de la carte pour placer nos marqueurs et créer notre carte de visibilité. Pour le moment les modifications se font manuellement, mais afin d'améliorer notre programme, il faudrait récupérer nos différentes valeurs (résolution, origine du repère, etc...) directement dans le fichier *.yaml*.

Enfin, la valeur 32 correspond à la conversion pixel/mètres de la hauteur de notre fenêtre en faisant :
$640 * 0.05 = 32$.

Ainsi, on peut écrire :

$for(a = 0; a < nbr\_amers; a++)$
{
. $x[a] = (x1[a] + 12.2)/0.05;$
. $y[a] = -(y1[a] + 13.8 - 32)/0.05;$
. $alphamax[a] = pi/4;$
}

2.5.2 Noeud retournant le nombre d'amers visibles

Maintenant que nous possédons une carte de visibilité, le nombre d'amers visibles de la position du robot est renvoyé à l'aide de la node *visib_pgmreader_node.cpp*. Il lit le fichier PGM créé et renvoie la valeur du pixel pour une position (x,y) sur cette carte (on utilise le topic MarkersVisibility). Si le pixel vaut 15 on est dans une zone blanche donc pas de marqueurs visibles. Pour un pixel de valeur 12, cela veut dire qu'on est dans le champ de visibilité d'un amer. Pour 9, 2 amers visibles. Pour 6, 3 amers visibles. Pour 3, 4 amers visibles. Ensuite pour 5 amers visibles, le pixel passe à 0, cependant on ne peut plus distinguer si le pixel est à 0 car 5 amers visibles ou bien car il s'agit du pixel correspondant à l'amer. C'est pourquoi dans notre cas, on suppose qu'au maximum 4 amers sont visibles en même temps. Si on souhaite avoir plus d'amers visibles en même temps, il suffit d'augmenter la valeur maximale lors de l'écriture de la carte de visibilité. En effet, lors de la création de l'entête PGM, on définit la valeur maximale de nos pixels (dans notre cas 15).

ATTENTION : nous n'avons pas réussi à la rendre fonctionnelle lors de notre intégration. Lors des tests avant intégration, le node suivant renvoyé bien le nombre d'amers visibles, cependant en l'intégrant, notre fonction $pgm\_imread(char * argv)$ ne stocke plus correctement les valeurs des données. Il faut donc la débugger avant qu'elle soit utilisée.

2.6 Superviseur

2.6.1 Comportement de la navigation

Le but du superviseur est de guider le robot pour qu'il atteigne un but de coordonnées (x, y) dans la carte. Afin de gérer le comportement de la navigation à l'aide d'amers, pour atteindre ce but précis, nous avons mis en place un nœud ROS de supervision : *hightLevelCommand_node*. Ce nœud permet d'utiliser des commandes de haut niveau telles que *chercher un amer* ou bien *se déplacer vers un but*.

(TODO : but absolu)

On suppose alors qu'initialement, le robot est dans une position connue ou dans une des zones de visibilité. Si sa position est assez proche du but final suivant un seuil que l'on définira par la suite, on commande alors au robot de se déplacer directement vers le but. Si la position initiale est trop éloignée, on se déplace d'amer en amer pour atteindre l'amer qui est le plus proche du but final pour enfin se déplacer vers le but final. Tous les amers et les distances entre ces derniers sont représentés par un graphe, ainsi on utilise l'algorithme du plus court chemin Dijkstra pour déterminer vers quel amer se déplacer. Ce graphe est re-calculé à chaque fois que l'on voit un amer.

Le comportement détaillé du superviseur est décrit par une machine à états décrite dans la figure 11.

Lors de la navigation, quatre signaux sonores permettent de connaître l'état de la navigation. Ils ont lieu lorsqu'un marqueur est détecté, lorsque le robot a atteint un but intermédiaire ou final et lorsque le robot décroche de son suivi de trajectoire. Si le robot atteint sont but final avec une précision inférieure à celle demandée au lancement du superviseur, un son supplémentaire est joué.

La recherche d'une zone de visibilité n'a pas encore été implémentée. Dans le cas où l'on connait la position initiale du robot dans la carte, il faudrait alors lui commander de se déplacer vers la zone de visibilité la plus proche et alors entamer la recherche de marqueur comme dans le scénario habituel. Dans le cas où cette position est inconnue, il faudrait tout d'abord mettre en place une détection d'obstacles pour pouvoir évoluer dans un environnement inconnu jusqu'à trouver un marqueur.

2.6.2 Graphe des amers

Pour choisir vers quel amer (AR marqueur) se déplacer lorsque l'on en a détecté un et que l'on se trouve encore trop loin du but final pour se diriger directement vers celui-ci, on utilise un graphe représentant tous les marqueurs (figure 12) définis dans un fichier de format XML (figure 13). Chaque nœud du graphe possède différentes propriétés :

- Id : numéro correspondant au marqueur AR ;
- Label : nom du nœud, chaque nœud correspond à un marqueur AR ;
- PositionX : coordonnée selon l'axe x de la carte connue de l'environnement ;
- PositionY : coordonnée selon l'axe y de la carte connue de l'environnement;
- Orientation : angle entre la normale du marqueur AR et l'axe x de la carte (entre 0 et 2π).

Les numéros permettent de se situer dans le graphe lorsque l'on détecte un marqueur et ainsi trouver le marqueur suivant à atteindre pour continuer la navigation. Les coordonnées (x, y) des marqueurs permettent de connaître les positions à atteindre dans la carte. L'orientation permet de commander au robot de se déplacer en face et à une certaine distance du marqueur pour ne pas foncer dans un mur.

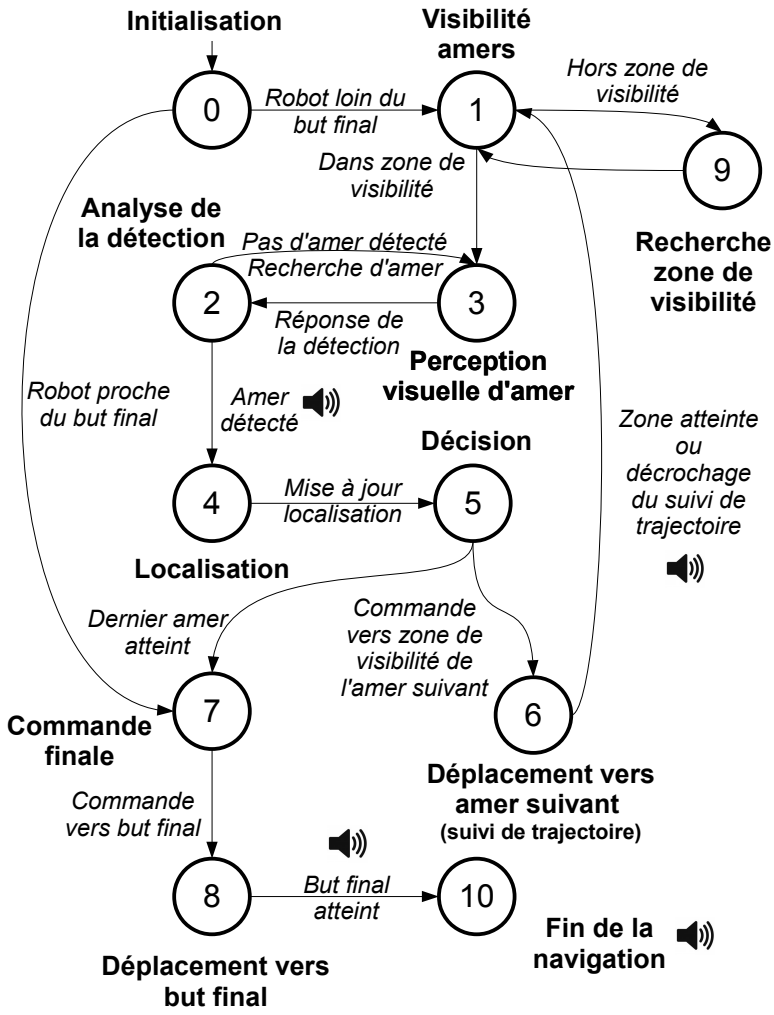

Figure 11: Schéma de la machine à états du superviseur de la navigation entre amers

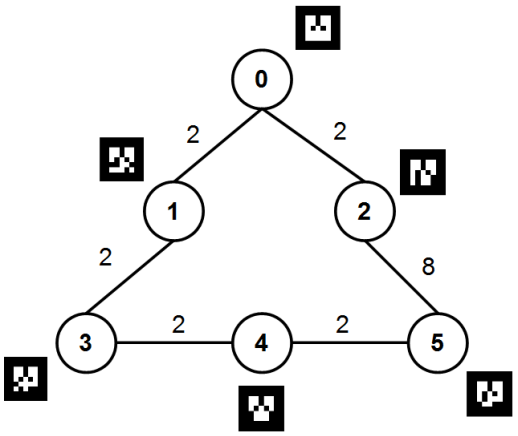

Figure 12: Exemple de graphe d'amers (marqueurs AR)

```
<DirectedGraph Title="ARMarkers">
  <Nodes>
    <Node Id="0" Label="ARMarker0" PositionX="0.0" PositionY="2.0" Orientation="0"/>
    <Node Id="1" Label="ARMarker1" PositionX="2.0" PositionY="2.0" Orientation="1.57"/>
    <Node Id="2" Label="ARMarker2" PositionX="3.0" PositionY="2.0" Orientation="3.1416"/>
    <Node Id="3" Label="ARMarker3" PositionX="4.0" PositionY="2.0" Orientation="1.57"/>
    <Node Id="4" Label="ARMarker4" PositionX="4.0" PositionY="3.0" Orientation="3.1416"/>
    <Node Id="5" Label="ARMarker5" PositionX="4.0" PositionY="4.0" Orientation="4.71"/>
  </Nodes>
  <Links>
    <Link Source="0" Target="1" Cost="2"/>
    <Link Source="1" Target="3" Cost="2"/>
    <Link Source="3" Target="4" Cost="2"/>
    <Link Source="4" Target="5" Cost="2"/>
    <Link Source="0" Target="2" Cost="2"/>
    <Link Source="2" Target="5" Cost="8"/>
  </Links>
  <Properties>
    <Property Id="PositionX" Label="Position X" DataType="float64"/>
    <Property Id="PositionY" Label="Position Y" DataType="float64"/>
    <Property Id="Orientation" Label="Orientation" DataType="float64"/>
    <Property Id="Cost" Label="Cost" DataType="int32"/>
  </Properties>
</DirectedGraph>
```

Figure 13: Exemple de définition d'un graphe d'amers (marqueurs AR) en format XML

2.6.3 Commandes de haut niveau

Pour mener à bien la navigation, le superviseur utilise la classe *HighLevelCommand* qui fournit les commandes de haut niveau suivantes :

- **init(threshold)** : L'initialisation permet de définir, suivant le seuil (**threshold**) choisi en paramètre, si le robot est trop éloigné ou non du but final pour se servir des marqueurs durant la navigation. Si le robot est trop proche, la fonction **init** renvoie -1, le superviseur commandera alors au robot de se déplacer directement vers le but final à l'aide de la fonction **sendFinalGoal()**. Dans le cas contraire, la fonction cherche le marqueur le plus proche du but final, navigue jusqu'à celui-ci, avant d'envoyer la dernière commande vers le but final.
- **seekMarker()** : Cette fonction permet de commander le comportement de la recherche d'un amer lorsque l'on se trouve dans une zone de visibilité. Tant que cette méthode est appelée et qu'aucun marqueur n'a été détecté, le robot tourne successivement sur lui-même de $\pi/4$, $-\pi/2$, $3\pi/4$, $-\pi$, $5\pi/4$, $-3\pi/2$ ou $7\pi/4$. Si aucun marqueur n'a été détecté après toutes les rotations, la recherche recommence. (TODO visib: juste avant on dit que ça marche pas)
- **sendMarkerGoal(distanceToMarker)** : Cette méthode permet de commander au robot de se déplacer en face du marqueur suivant à atteindre pour suivre le plus court chemin vers le but final. La distance à respecter par rapport au marqueur est définie par le paramètre **distanceToMarker**.
- **sendFinalGoal()** : Cette fonction commande directement le robot vers le but final, choisi à l'instanciation de la classe *HighLevelCommand*.
- **askForMarker()** : Cette méthode permet de lancer une recherche visuelle de marqueur, effectuée par le nœud de localisation.
- **finalGoal(threshold)** : Permet de jouer un signal sonore spécial lorsque la précision de la position finale du robot par rapport au but fixé est inférieure (TODO : précision inférieure ?) à un certain seuil, défini à l'aide du paramètre **threshold**.

2.6.4 Structure logicielle

Afin de communiquer avec les différents nœuds, le nœud de supervision *hightLevelCommand_node* est client d'un service, il publie sur quatre topics et il est abonné à quatre autres topics.

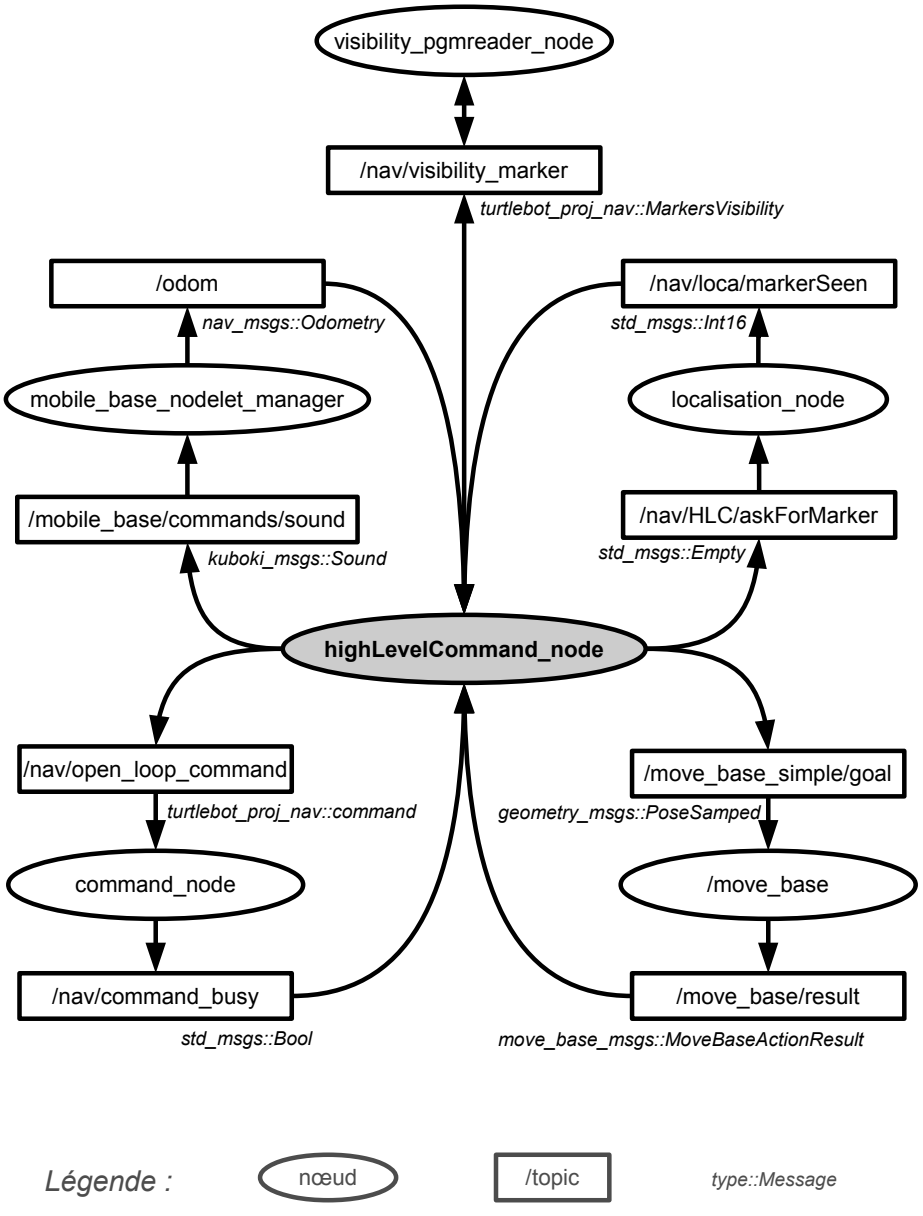

Figure 14: Schéma de la structure logicielle du nœud de supervision et de ses interactions

2.6.5 Lancement du superviseur

Une fois la configuration minimale et les nœuds de navigation lancés, on lance le superviseur dans un terminal à l'aide de la commande suivante :

rosrun turtlebot_proj_nav highLevelCommand_node X_GOAL Y_GOAL XY_THRESHOLD XY_PRECISION DISTANCE_TO_MARKERS

avec les paramètres suivant :

- **X_GOAL** et **Y_GOAL** : les coordonnées (x, y) du but final à atteindre dans la carte ;
- **XY_THRESHOLD** : le seuil à partir duquel les amers ne sont pas utilisés pour la navigation ;
- **XY_PRECISION** : la contrainte de précision demandée pour que le signal sonore final soit joué ou non ;
- **DISTANCE_TO_MARKERS** : la distance par rapport aux marqueurs que le robot doit respecter lorsqu'il se déplace d'amer en amer.

3 Conclusion

List of Figures

ANNEXE