Travaux de laboratoire

Rovio Track

02/02/2016

Amandine GOUT
Yann LIFCHITZ

Conception

Présentation de l'architecture

Dans un premier temps seront présentés les nœuds ROS réalisés pour **le contrôle en boucle ouverte**. Il sera fait état de leur structure, des lois de commande choisies dans le but de commander le robot en vitesse à l'aide de message de type Twist. Ceci comprendra la présentation des nœuds **nav_to_goal** et **nav_points**.

Puis sera présenté le travail réalisé dans le cadre de **la détection de Rovios avec les caméras Axis PTZ et de la projection des positions sur le sol**. Ceci comprendra la présentation des nœuds **detect**, **tracker**, **sampler**, **localize**, **share, positions,** ainsi que de l'apprentissage des SVM pour la correspondance positions/pan-tilts de la caméra**.**

Dans une troisième partie sera présenté le **clustering de Rovios** détectés avec le nœud **gngt**.

Enfin, la dernière partie concernera la mise en place de **la localisation et de l'identification de Rovios par filtre de Kalman** avec le nœud **Kalman**. La mise en œuvre du **contrôle en boucle fermé** sera également présentée dans le nœud **nav_kalman**.

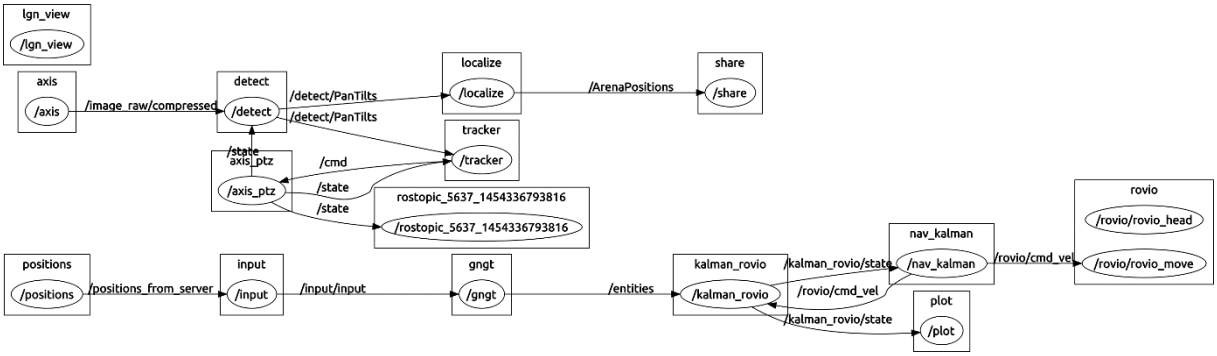


Figure 1: rqt_graph du projet complet

Objectif 1 : Contrôle en boucle ouverte

I. Nav to goal

a. Objectif

Dans cette partie, le nœud **nav_to_goal** a pour but de publier sur un topic de commande du robot un twist, afin d'amener celui-ci à un point (*goal*) défini par ses coordonnées (x,y) dans le repère « world », qui seront ensuite converties dans le repère du robot afin d'appliquer la loi de commande nécessaire au mouvement du robot vers sa cible.

b. Implémentation et méthode

Afin de réaliser cette fonction, la cible est définie en tant que paramètre avec *goal_x* et *goal_y*, que l'on peut redéfinir dans le launch file. A l'aide d'un booléen *running*, le nœud est informé de la proximité ou non du robot avec la cible. Ce booléen initialement à *true* sera rendu *false* par la méthode *exploration()* une fois celle-ci exécutée.

La méthode *exploration* étant lancée, celle-ci a recourt à la construction d'un message de type Pose2D de la cible dans le repère *world*, qu'elle convertit en message de type Pose2D contenant la position de la cible dans le repère du robot. Cette transformation se fait à l'aide de la fonction *transformPose2D.*

Une fois la cible définie dans le repère du robot, *exploration* fait appel à la méthode *moveTo*, qui construit un Twist *cmd_vel* en fonction de la loi de commande suivante :

- Dans le cas où le robot se trouve orienté d'un angle supérieur à $\frac{\pi}{9}$, une commande de rotation angulaire constante vers la cible est envoyée au robot, fonction du sens de l'angle restant.

$$\begin{cases} linear.x = 0 \\ angular.z = sgn(angle) * \max\_angular\_velocity \end{cases}$$

- Si le robot se trouve orienté d'un angle inférieur, une commande en translation d'une valeur maximale de +/- *vel_max* ou bien proportionnelle à la distante restante : *k_vel*distance* est envoyée. Concernant l'angle, une commande proportionnelle à l'angle restant est choisie : *k_alpha*angle*. Elle est également limitée par une vélocité angulaire maximale.

$$\begin{cases} linear.x = \min(k\_vel * distance, vel\_max) \\ angular.z = \max(\min(k\_alpha * angle, max\_angular\_velocity), -max\_angular\_velocity) \end{cases}$$

Une fois le robot proche d'une distance inférieure de *distance_min* par rapport à la cible, un twist nul est envoyé au robot. Le robot a alors atteint sa cible.

II. Nav points

a. Objectif

Dans cette partie, le nœud **nav_points** a pour but de publier sur le topic de commande du robot un twist, afin que celui-ci suive une trajectoire définie au préalable dans un fichier texte contenant les positions successives en (x,y) que le robot doit atteindre. Ce nœud présentera également deux services *go* et *abort*, permettant de lancer le départ du robot ou bien de l'interrompre durant son parcours.

b. Implémentation et méthode

Afin de réaliser cette fonction, les différentes cibles sont établies dans un fichier liste.txt que le nœud lira et dont il publiera la trajectoire sous la forme d'un PoseArray, qui pourra être visualisé sous RVIZ.

La loi de commande, ainsi que les méthodes permettant l'exploration, restent identiques à celles implémentées dans le nœud **nav_to_goal**. La cible à suivre est cependant extraite de la liste de cibles au fur et à mesure du parcours.

La mise en place des services *abort* et *go* diffère également au niveau de l'implémentation. Une nouvelle fois est fait usage d'un booléen *running*. Celui-ci est initialisé à false. Ainsi le nœud après initialisation ne débutera pas l'exploration tant que cette variable sera *false*. Le service *go* passe alors *running* à *true* et l'exploration a lieu tant que ceci n'est pas changé par le service *abort*. La fin de l'exécution du nœud fait également appel à la méthode *stop* qui permet l'envoi d'un Twist nul, assurant l'arrêt du robot.

c. Tests

Ces deux nœuds ont pu être testés tout d'abord sur *turtlesim*, et ont montré qu'il satisfaisait les fonctions pour lesquelles ils avaient été implémentés. Enfin un second test a été exécuté sur les rovio-mêmes prenant en compte un nœud ***fake_odometry***. Cependant deux obstacles se sont présentés :

- L'odométrie en raison du glissement des roues du Rovio sur le carrelage est faussée et ne permet pas de renvoyer des positions exactes en temps réel.
- La commande du Rovio qui se fait en vitesse à l'aide d'un message Twist ne peut se faire qu'alternativement en rotation ou en translation, ce qui ne permet pas la mise en place d'une loi de commande telle que nous l'avons définie précédemment de façon progressive.

Objectif 2 : Détection Rovios - Caméra Axis PTZ

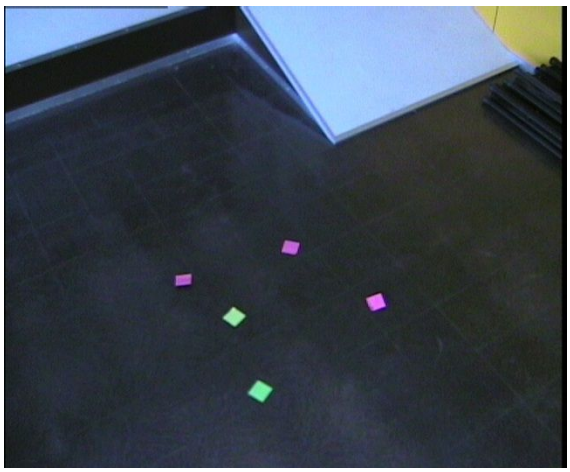
I. Objectif

Jusqu'à présent la position du robot était estimée par le nœud **fake odometry** en fonction des commandes envoyées au robot. Il s'agit ici de localiser le robot de façon plus précise en utilisant une caméra Axis PTZ. Il faut donc prévoir de localiser dans l'image un marqueur visuel qui sera fixé sur le robot, de pouvoir suivre ce marqueur, de pouvoir localiser le marqueur et enfin de partager sa position.

II. Détection

a. Analyse de l'image caméra

Le premier nœud que l'on a réalisé est le nœud **detect** chargé de détecter les robots dans l'image renvoyée par la caméra. Nous avons choisi de détecter les Post-it rose, qui seront à terme collés sur les robots. Après essais multiples, il nous a semblé que la couleur rose était la plus facile à détecter dans l'environ de travail.

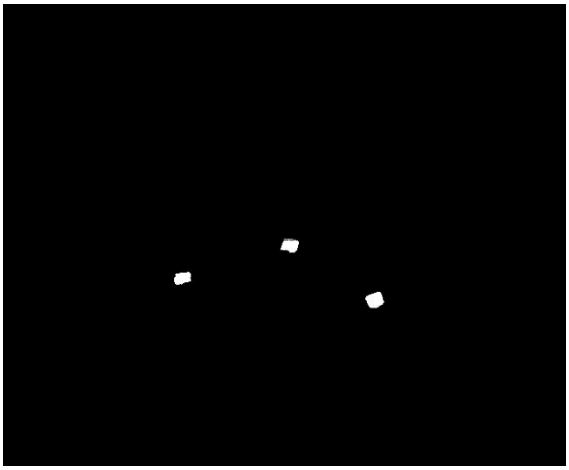


La première étape est de localiser les pixels de couleur rose dans l'image. Pour un éclairage donné, un certaine couleur peut être défini en traitant l'image en HSV. Il a donc fallu fixer les intervalles pour la teinte, la saturation et la luminosité.

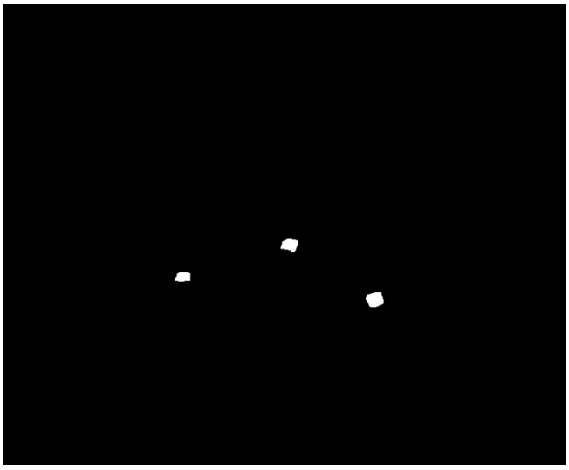
Pour notre iris (fixée à 879.0), l'intervalle HSV correspondant à la couleur rose est :

$$\begin{cases} 120 < H < 160 \\ 70 < S < 180 \\ 120 < V < 255 \end{cases}$$

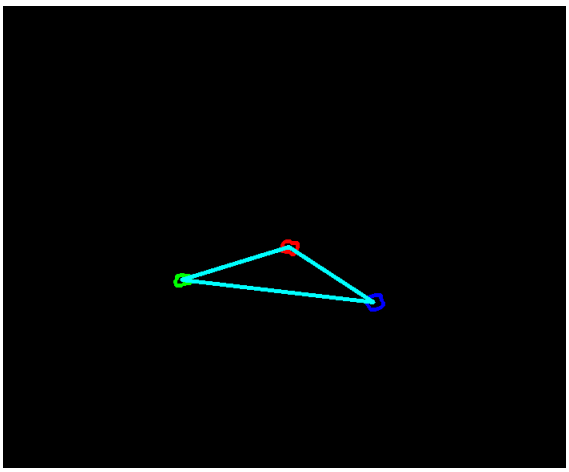
Ces conditions permettent d'obtenir une image binaire correspondant à tous les pixels détectés comme roses.



Il faut ensuite nettoyer cette image dans le cas de faux négatifs ou de faux positifs dans l'image (points noirs isolés et points blancs isolés). Pour cela nous avons appliqué successivement une ouverture morphologique (permet de supprimer les points blancs isolés) puis une fermeture morphologique (permet de supprimer les points noirs isolés). Comme éléments structurants nous avons choisi des carrés respectivement de tailles 3 et 9 pixels.



Une fois l'image nettoyée, il ne doit rester que les zones correspondantes à des postits roses. Chaque zone éclairée correspond donc à un robot. La position exacte du robot est donné par le barycentre des positions des pixels éclairés pour la région en question.



Une fois les robots localisés dans l'image, il faut transformer ces positions en pixels en coordonnées pan-tilts. Pour cela on utilise la fonction pantiltzoom qui fait cette conversion en utilisant comme référence l'état actuel de la caméra.

Ce nœud **detect** publie les positions pan-tilts sur le topic PanTilts.

b. Tracker

Le nœud **tracker** lit le topic PanTilts et publie une commande de position pour la caméra.

Trois modes de fonctionnement sont possibles :

- Mode track : la caméra se déplace pour se centrer sur un robot. Si plusieurs robots sont détectés, on choisit de se caler sur le robot qui engendre le déplacement minimal.
- Mode search : La caméra bouge pour jusqu'à trouver un robot. La loi de commande que l'on a définie est de parcourir l'espace des pan pour un tilt donné puis d'incrémenter le pan.
- Mode scan : pareil que le mode search sans l'arrêt lorsque l'un des robots est détecté.

On peut changer le mode en appelant le service *mode*.

III. Conversion en position

a. Echantillonnage

Pour pouvoir apprendre la régression qui donne les coordonnées cartésiennes d'un robot en fonction des coordonnées pantilts, il faut constituer une base de données d'apprentissage, c'est-à-dire des couples positions/pantilts qui sont le plus représentatifs de la zone de travail.

Comme repère, nous avons choisi de fixer l'origine dans un coin de la pièce et de prendre comme unité les carreaux du sol.

Nous avons choisi 12 points pour l'apprentissage (coordonnées dans le repère de la pièce) :

5 8
6 5
6 11
8 6
9 8
9 12
3 7
2 5
6 6
8 3
7 9
10 4

Pour enregistrer les coordonnées pantilts correspondantes, nous avons implémenté le nœud **sampler** qui lit le fichier de position et crée un nouveau fichier de données dans lequel il

stocke le quadruplet. On déclenche l'acquisition du point suivant en appelant le service *register*.

b. Apprentissage

Pour l'apprentissage, on utilise gaml-libsvm pour entrainer des svm sur la base d'apprentissage. L'apprentissage permet de générer deux fonctions de deux variables : celle qui donne x en fonction de pan et tilt, et celle qui donne y en fonction de pan et tilt. Pour régler les paramètres des svm, nous avons calculé pour chacun le risque empirique et avons estimé le vrai risque en calculant le risque « leave one out ».

Les paramètres choisis sont :

- RBF kernel
- Gamma = 0.002
- Epsilon = 0.1
- C = 120
- Tolérance = 0.04

Avec ces paramètres, les risques calculés sont :

- Risque empirique sur x = 0.012 (=0.11²)
- Risque empirique sur y = 0.009 (=0.10²)
- Estimation du risque réel (« leave one out ») = 2.11 (=1.45²)

Le risque réel semble acceptable étant donné le relativement faible nombre de points dans notre base de données. Nous avons jugé cet apprentissage suffisant dans un premier temps.

En utilisant gnuplot nous avons générer les graphiques correspondant aux deux prédicteurs trouvés :

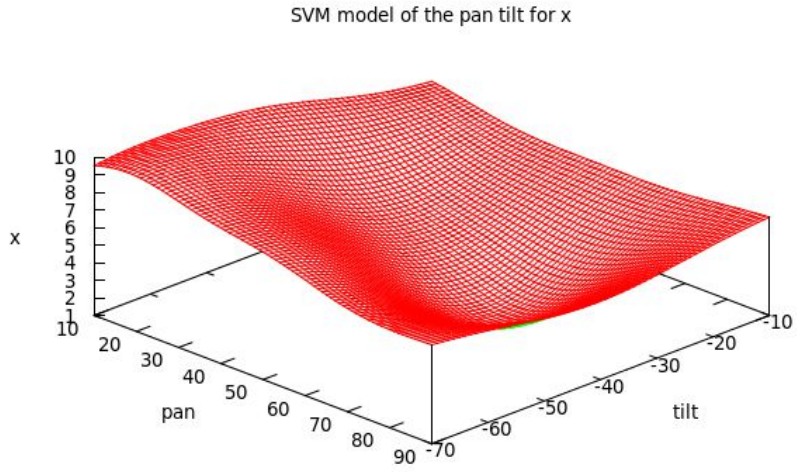


Figure 2: modèle SVM pour x en fonction de pan et tilt

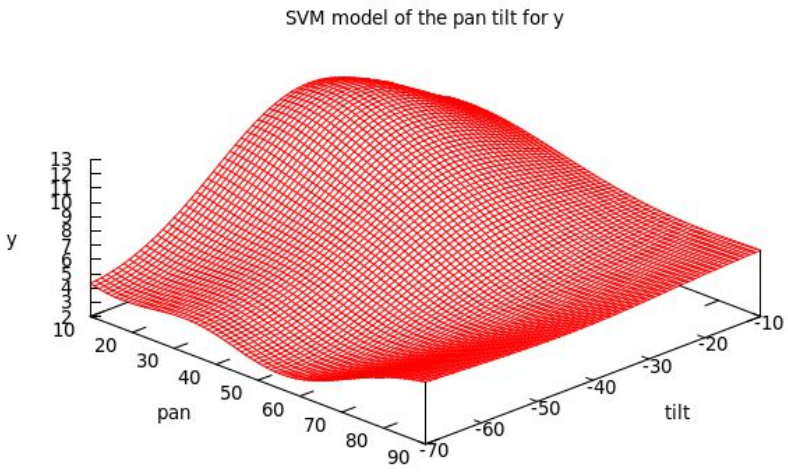


Figure 3: modèle SVM pour y en fonction de pan et tilt

c. Convertisseur

Une fois l'apprentissage terminé, on utilise les prédicteurs dans le nœud ROS **localize** qui convertit les *PanTilts* publiés par **detect** en *ArenaPositions*.

IV. Partage de positions

On veut centraliser les détections des robots effectuées par tous les groupes. Ce partage de détection permet d'appliquer e principe du « bagging », c'est-à-dire de combiner plusieurs régresseurs pour obtenir une variance faible.

Pour mettre en place le server, on utilise le code fourni dans le sujet. Nous avons choisi d'utiliser le port 10000 (choix arbitraire qui peut être changé dans les fichiers launch de **share** et **positions**).

Pour envoyer les positions vers ce serveur, nous avons écrit un nœud ROS **share**. Ce nœud s'abonne au topic *ArenaPositions* et renvoie toutes les positions au serveur.

Objectif 3 : Clustering de Rovios

I. Input

a. Objectif

Le but de ce nœud situé dans le package **gngt**, est de convertir les ArenaPositions correspondant aux robots détectés et de les convertir sous forme de point clouds afin de les soumettre à l'algorithme gngt. Il s'agit donc d'un prétraitement des positions publiées sur le topic *positions_from_server* où le nœud **positions** publie les ArenaPositions récupérées sur le serveur.

b. Implémentation et méthode

La méthode de callback *build_pointcloud* est appelée à chaque récupération d'un ArenaPositions par le subscriber. Cette méthode construit ensuite un pointcloud en partant d'une distribution de points uniforme sur la zone de la pièce concernée par l'apprentissage précédent. Le nombre d'échantillons est également fixé.

Pour chaque échantillon, la méthode *closest_point* renvoie l'ArenaPosition le plus proche. Enfin si cette position se trouve à une distance inférieure à *distance_min*, et donc si celle-ci est suffisamment proche de l'échantillon, celui-ci est conservé et inséré dans le pointcloud.

La distance considérée entre les positions est la norme L2.

Le nœud **input** publie ensuite le pointcloud résultant sur le topic *input* auquel s'abonnera le nœud **gngt**.

II. Gngt

a. Objectif

Le nœud **gngt** fourni permet de calculer à partir d'un pointcloud, une forme convexe formée de sommets et arrêtes et qui approxime la forme du pointcloud. Dans le cas présent, l'intérêt d'utiliser ce nœud est d'obtenir une forme convexe pour chaque pointcloud représentant un robot différent, puis d'en calculer le centre et de publier un marker de type cylindre centré sur ce point qui pourra être visualisé sous RVIZ. A ceci sera ajouté la publication d'un message de type Entities, constitué d'un ArenaPositions des centres, ainsi que d'un label pour chaque robot correspondant.

b. Implémentation et méthode

Le nœud **gngt** souscrit au topic *input* où le nœud **input** publie le pointcloud correspondant aux différents robots observés.

La méthode de callback correspondante est *gngt_epoch_cb.* Celle-ci a été modifiée par rapport au code fourni car elle prend désormais en paramètre deux autres publishers: un pour les cylindres et l'autre pour les *Entities*. Au sein de cette méthode est construit le graphe gngt comme prévu initialement.

La modification de *gngt_epoch_cb* inclut désormais le calcul des centres des formes convexes obtenues en réalisant un barycentre des sommets à l'aide de la méthode *getCenter*. Un marker de type cylindre est alors créé et publié, centré en chaque centre de forme convexe, à l'aide de la méthode *draw_cylinder*. Aussi, ces cylindres étant visualisés sous RVIZ, les identifiants de chaque *component* sont stockés afin d'éviter le stockage de markers cylindriques qui seraient obsolètes. Les markers cylindriques sont publiés sur le topic *cylinders.*

Concernant les messages Entities, ceux-ci sont construits à partir d'un tableau d'ArenaPositions des centres, munis du *label* de chaque *component.* Les Entities sont publiés sur le topic *entities.*

c. Résultats visualisés

Comme il peut être observé sur la photo ci-dessous, voici le résultat obtenu à l'aide des nœuds **input** et **gngt.** L'image de droite représente le Rovio détecté par la caméra, à l'aide du post-it rose collé sur sa surface. Deux autres post-it roses représentant deux autres Rovios potentiellement présents dans la pièce ont été rajoutés.

A l'aide de RVIZ sont visualisés les topics *markers* et *cylinders* de **gngt**. On peut alors constater que chaque robot est représenté par une forme convexe propre à chaque robot car de couleur différente. En chacun des centres des formes convexes est publié un cylindre.

D'autres tests en temps réel permettent également de constater le suivi de chaque robot lors de son déplacement. Il est nécessaire de faire un compromis sur le temps de persistance sur le serveur de position. Si la persistance est trop élevée, gngt prend en compte des positions anciennes du robot ce qui implique un retard sur la localisation. Si la persistance est trop

petite, il risque d'y avoir par moment aucun point sur le serveur (les positions ne sont actualisées qu'après mise à jour de l'état de la caméra qui peut parfois être longue). Nous avons donc réglé la persistance à 1.5s. Aussi la disparition des cylindres lors du retrait des robots a été vérifiée, ce qui valide le fonctionnement de la publication des markers en fonction des identifiants des *components* utilisés.

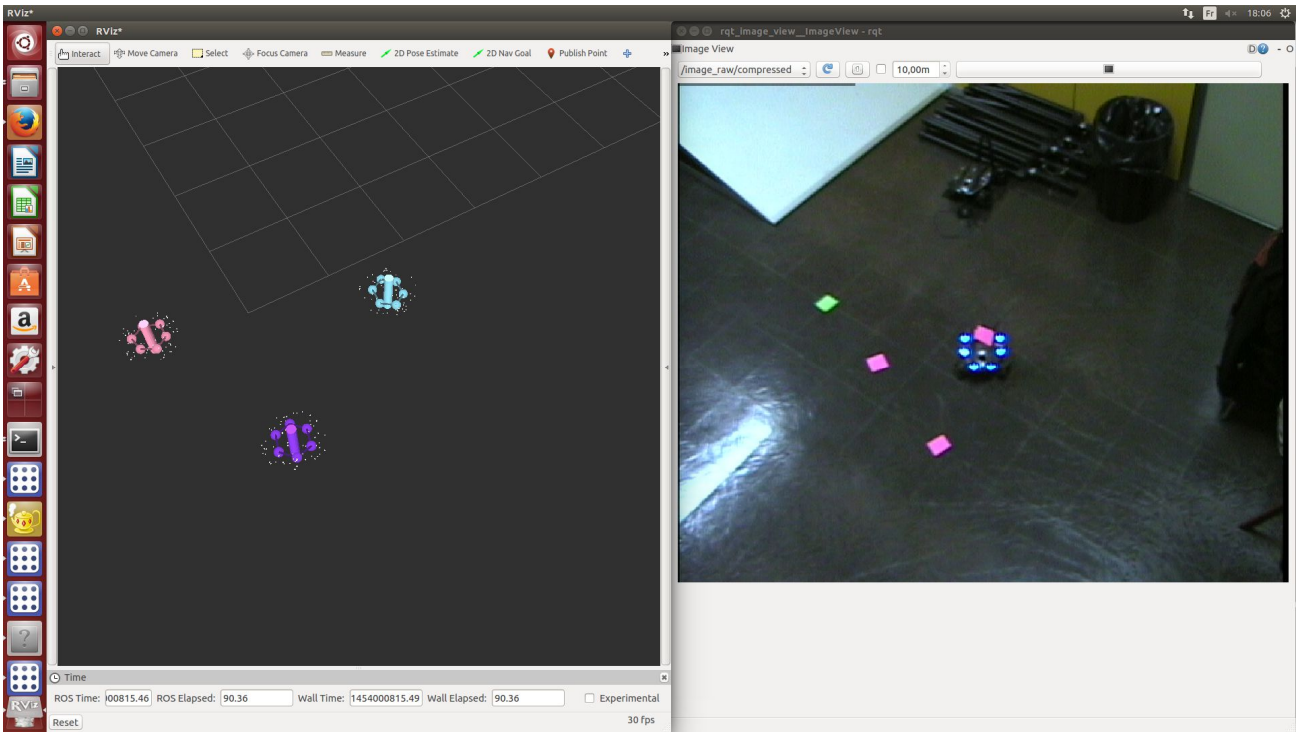


Figure 4: Observation des markers cylindriques de gngt sous RVIZ

Objectif 4 :

- Kalman : localisation et identification de Rovios
- Contrôle en boucle fermée

I. Kalman

a. Objectif

On a maintenant une mesure de la position du robot renvoyée par le nœud gngt. Cette mesure est imprécise et ne nous permet pas de connaitre l'orientation du robot, pourtant essentielle pour le contrôle. Afin d'affiner la localisation et permettre de connaitre dynamiquement l'orientation, on implémente une filtre de Kalman.

b. Modèle et paramètres

- Vecteur d'état : $X = \begin{pmatrix} x \\ y \\ theta \end{pmatrix}$
- Vecteur de commande (lecture de la commande twist *cmd*) : $U = \begin{pmatrix} v \\ \Omega \end{pmatrix}$
- Vecteur d'observation (sortie de **gngt**) : $Z = \begin{pmatrix} x_d \\ y_d \end{pmatrix}$
- Matrice de covariance du vecteur d'état initialisé à : $P = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 3.5 \end{pmatrix}$
- Matrice de covariance du bruit de mesure : $R = \begin{pmatrix} 0.1 & 0 \\ 0 & 0.1 \end{pmatrix}$
- Matrice de covariance du bruit d'intégration : $Q = \begin{pmatrix} 0.5^2 & 0 & 0 \\ 0 & 0.5^2 & 0 \\ 0 & 0 & 0.001 \end{pmatrix}$

A chaque mise à jour de la position, on utilise le filtre pour faire une prédiction et une mise à jour de l'état.

Pour définir A et B dans les équations du filtre de Kalman, on doit calculer les jacobiens de l'équation d'intégration (ici on est dans le cas non linéaire, il faut donc linéariser autour du point de travail). On trouve :

$$A = \begin{pmatrix} 1 & 0 & -v * \sin(theta) * dt \\ 0 & 1 & v * \cos(theta) * dt \\ 0 & 0 & 1 \end{pmatrix}$$

$$B = \begin{pmatrix} \cos(theta) * dt & 0 \\ \sin(theta) * dt & 0 \\ 0 & dt \end{pmatrix}$$

Alors on peut appliquer :

- Prédiction :

$$X = A * X + B * U$$
$$P = A * P * A^T + Q * dt^2$$

- Mise à jour :

$$Innov = Z - H * X \text{ avec } H = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$
$$Innov_{cov} = H * P * H^T + R$$
$$K = P * H^T * {Innov_{cov}}^{-1}$$
$$X = X + K * Innov$$

$$P = (I - K * H) * P \text{ avec } I = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

Nous avons choisi d'implémenter ce filtre de Kalman sous la forme d'un nœud ROS python. Pour les calculs vectoriels nous avons utilisé la bibliothèque numpy.

Le nœud **kalman**, s'abonne au topic *cmd* de commande du rovio ainsi qu'au topic *entities* de sortie de **gngt**. Il publie son état sous la forme d'un tableau de float afin de pouvoir facilement observer l'évolution de l'état en utilisant **rqt_plot**.

II. Nav kalman

a. Objectif

Le but de ce dernier nœud est de mettre en place un contrôle en boucle fermée et d'essayer de faire suivre au Rovio une trajectoire préalablement définie dans un fichier liste.txt comme il a été fait précédemment à l'aide de l'odométrie. A la différence de cette première tentative, nous allons utiliser la détection des Rovios à l'aide de la caméra au lieu de reposer sur l'odométrie. Le filtre de Kalman, nous permet également d'obtenir un suivi lissé de la position du Rovio.

b. Implémentation et méthode

Le nœud **nav_kalman,** semblable à **nav_points** concernant la commande du robot, diffère de celui-ci en quelques points.

La structure concernant la lecture de la trajectoire, les services *go* et *abort*, ansi que la loi de commande restent les mêmes. La principale différence est l'arrêt de l'utilisation de TF afin de convertir la position de la trajectoire depuis le repère *world* au repère du robot.

Le repère de référence est désormais celui que nous avons choisi dans la pièce. Celui-ci prend origine dans l'angle à gauche de la porte d'entrée, x décrit le sens de la largeur de la pièce orienté vers la fenêtre. La coordonnée y décrit le sens de la longueur de la pièce orienté vers le tableau. Enfin une unité correspond à un carreau du sol. La conversion des positions depuis ce repère vers celui du robot se fait à l'aide de la fonction *convertPosition*.

La position courante du robot, elle, est désormais obtenue en souscrivant au topic *kalman_rovio/state* et est enregistrée à l'intérieur de la méthode *callback.*

Lors du lancement de l'exploration par le service *go*, le nœud réalise une étape préalable d'initialisation du filtre de Kalman à l'aide d'une trajectoire périodique rectiligne. Celle-ci est définie dans la méthode *initialize.* Cette étape permet en théorie de fixer l'orientation initiale du robot. Une fois cette étape terminée, l'exploration est lancée comme précédemment décrit pour le nœud **nav_points**.

c. Résultats et problèmes rencontrés

Au cours des tests réalisés, nous avons pu vérifier le bon fonctionnement des calculs et conversion de trajectoire depuis le repère de la pièce dans le repère du robot. Voici tout d'abord les prédictions observées lors de la période d'initialisation du filtre de Kalman:

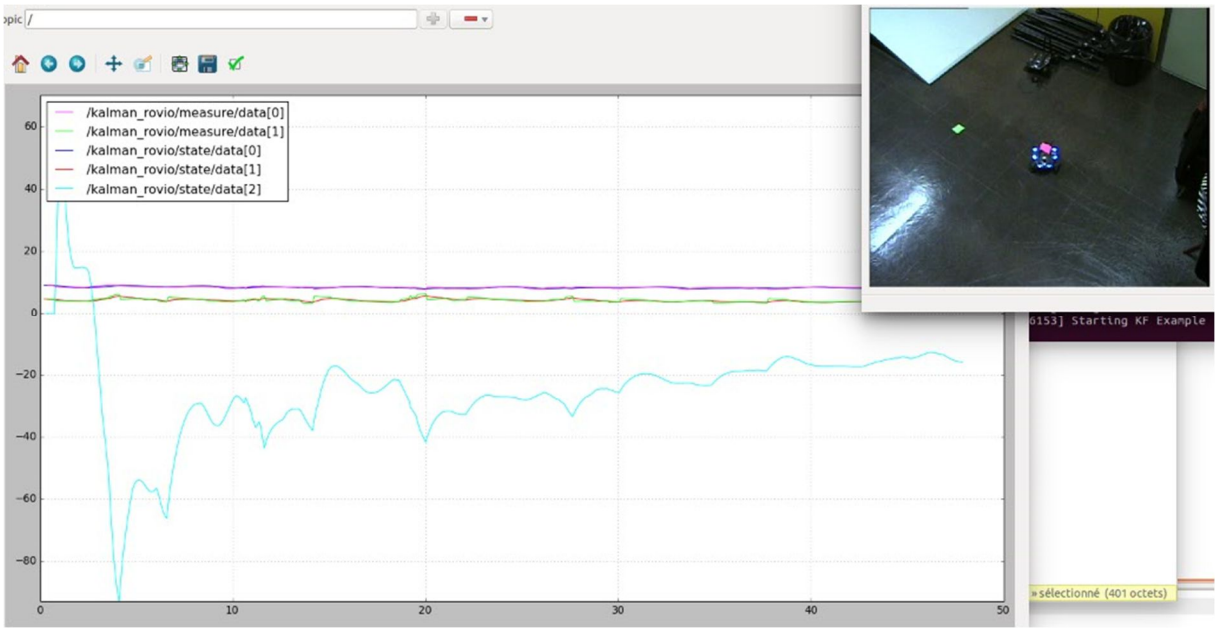


Figure 5: Prédictions des positions x,y,theta avec filtre de Kalman

On remarque que l'orientation du robot (theta, en bleu sur la figure) se stabilise pendant la période d'initialisation. La valeur obtenue à la fin de cette phase est ici d'environ -20° pour une orientation réelle d'environ -40° ce qui est malheureusement peu précis.

Cependant, nous ne sommes pas parvenus à faire fonctionner le contrôle en boucle fermé pour deux raisons :

- Le Rovio présente une loi de commande propre qui limite les vitesses publiées par les Twist de façon très restrictive. Il est normal d'observer une vitesse maximale définie en translation ou rotation. Cependant nous avons été surpris de constater qu'il existait également une vitesse minimale angulaire et de translation, qui ne permet pas l'exécution de faibles vitesses à l'approche de la cible.

- La commande du Rovio ne peut également d'effectuer que de façon alternative en rotation et translation. Il n'est pas possible d'exécuter une translation incurvée, ce qui nous a amenés à modifier la loi de commande dans **nav_Kalman** pour finalement n'effectuer plus qu'une rotation vers la cible, suivi d'une translation rectiligne.

- Le dernier point majeur a été de constater la mauvaise prédiction d'angle réalisée par le filtre de Kalman. Bien que l'on parvienne à stabiliser la prédiction de l'angle du robot, quoique déjà de façon peu précise (45° au lieu de 60° attendus), celui-ci effectue un saut lors de la prise en compte de commandes en rotation et ne suit plus les valeurs attendues. La position theta n'étant pas mesurée, celle-ci dépend donc principalement de notre modèle de prédiction ainsi que de la commande U. Or celle-ci étant mal connue étant donné la spécificité des Rovios, sa prise en compte fausse le modèle et ne permet pas une prédiction correcte de *theta*. Sur la figure ci-dessous, on peut observer le saut réalisé par theta au départ de la commande angulaire :

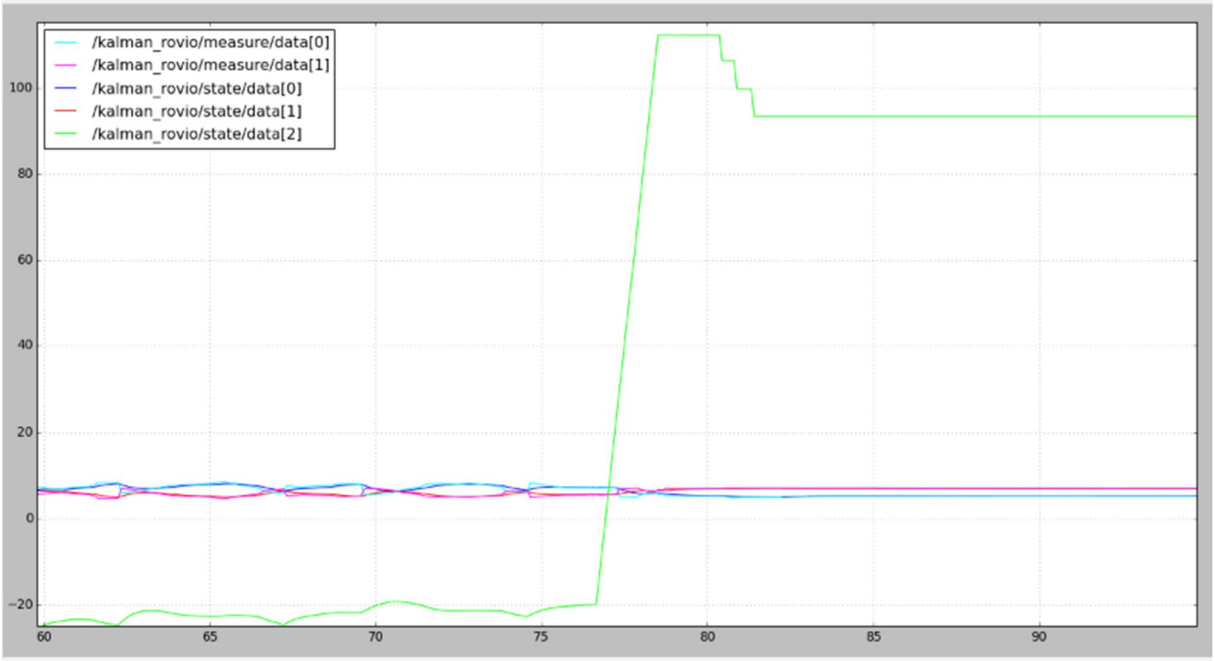


Figure 6: Pic observé sur theta lors de l'envoi d'une commande angulaire

Ainsi, ne disposant pas de la position angulaire du robot en temps réel, il nous a été impossible de paramétrer correctement le filtre de Kalman et de réaliser la commande des Rovios en boucle fermée.

D'un point de vue pratique, les Rovios ont également présenté l'inconvénient d'avoir un problème de batterie. Ils ne disposaient que d'une autonomie très limitée (quelques minutes) bien qu'ayant été chargés pendant plusieurs jours.

Malheureusement nous n'avons pas pu avoir accès à d'autres robots lors des dernières séances de travaux de laboratoire, ce qui ne nous a pas permis de conclure définitivement quant au résultat du contrôle en boucle fermée. Seul un essai a pu être réalisé la veille de notre soutenance sur un Turtlebot. Cependant nous ne sommes pas parvenus, dans le temps qui nous était imparti, à le faire fonctionner correctement en raison de la nécessité d'envoyer des commandes en continu pour l'initialisation du filtre de Kalman. Ceci nécessitait en effet un changement de l'initialisation que nous avons effectué. Malheureusement, un problème technique est survenu, entraînant la non réponse du robot aux commandes de vitesse.