SY31_P21

Rapport Projet SY31

Groupe de TP: Jeudi 13h15-16h15

Responsable de TP: Antoine Lima, Corentin Sanchez

Étudiants: HO Xuan Vinh

Table de matières

I. Introduction

Dans ce projet, j'ai choisi le scénario #1 impliquant Caméra & LiDAR. Le but de ce scénario est que le robot observe et s'approche de la cible si possible. Dans ce scénario, la cible sera un objet de couleur unique, identifié par la caméra. Le robot visera la cible et s'arrêtera s'il s'approche trop près des obstacles. Lorsque le robot s'arrêtera, nous dirigerons manuellement le robot dans une autre direction.

Parce que ce scénario implique Caméra et LiDAR, je me suis concentré sur l'utilisation de ce que j'ai appris en TP4 et TP5. D'ailleurs, l'arrêt du robot en s'approchant trop près d'un obstacle est similaire à l'exercice que j'ai pratiqué en TP1. Par conséquent, les connaissances générales que j'utilise se concentreront sur TP1 TP4 et TP5.

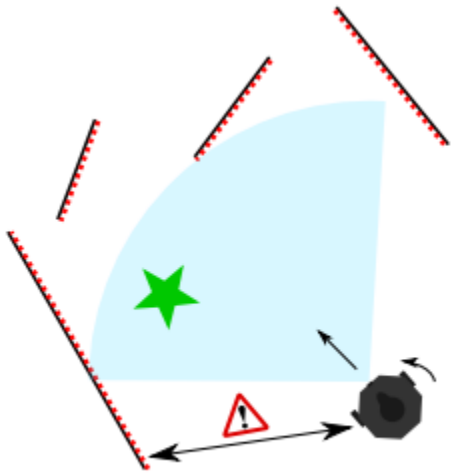


Figure 1. Image de description du projet

II. Description

1. Nœuds

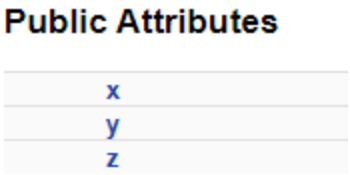
Dans ce projet, je crée 2 nœuds avec 2 objectifs importants qui, je pense, jouent un rôle important, 2 nœuds sont:

- detect(**class CameraNode**): s'appuie sur TP4 et TP5 et utilise le fichier de détection appris dans TP5. La tâche de ce nœud est de déterminer l'emplacement de la cible, la surface cible. C'est l'information nécessaire pour décider du mouvement du robot.
- moving(**class moving**): le noeud le plus important du projet. C'est le nœud qui prend la décision de déplacer et de rediriger le robot en fonction des informations déterminées par le nœud detect.

2. Topics

Bien entendu, les Nœuds ne pourra pas accomplir sa tâche sans le support des Topics, les Topics que j'utilise dans ce projet sont:

- /camera/image_rect_color: topic standard dans TP5.
- ~output: topic standard
- /scan: qui permet de récupérer les données brutes du LiDAR. Type : **LaserScan**
- /cmd_vel : qui permet de contrôler le mouvement de Turtlebot. Type : **Twist**
- /position: la détection de cible est basée sur les valeurs de (x, y, z) que je fais correspondre (left,forward,right. C'est gauche, avant, droite) dans l'image de la caméra. Par exemple x=1 si la cible est à gauche et x=0 sinon. Dans la section suivante, je montrerai que dans ce projet j'ai utilisé 4



directions principales. Type : **Point32.**

- /surface: déterminer la surface de la cible, je ne sais pas si la cible sera un rectangle ou un cercle. Mais je vais déterminer les paramètres qui suffisent pour calculer l'aire. Je m'appuie sur les fonctions apprises en TP5. Type : **Float32.**

3. Relation entre Noeuds et Topics

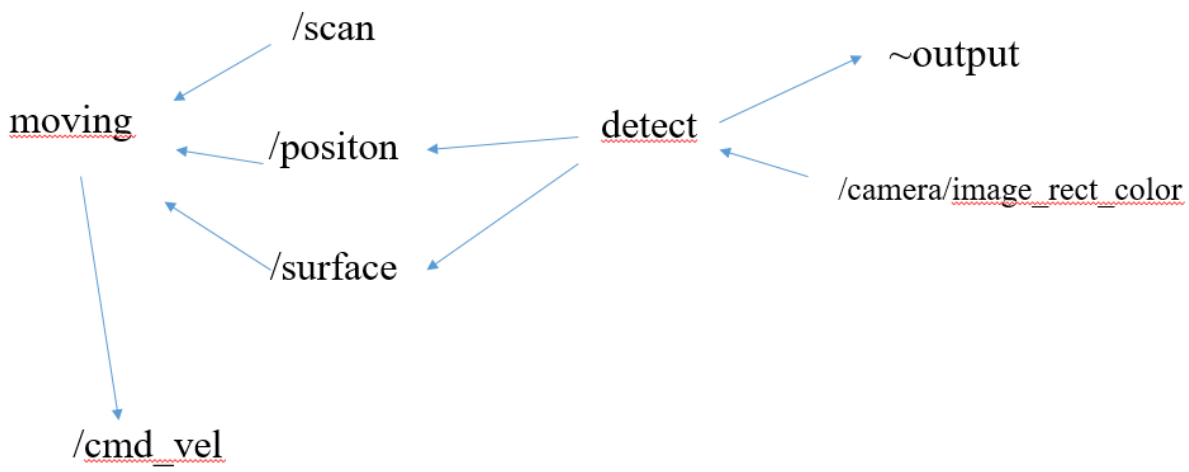


Figure 2. Relation entre Noeuds et Topics

III. Algorithmes et fonctions utilisés

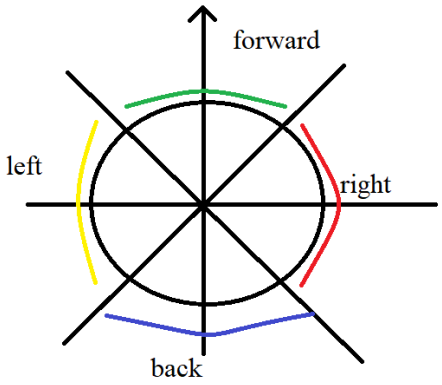


Figure 3. Direction du Robot

1. Class CameraNode

Comme décrit précédemment, dans la classe CameraNode, noeud detect Publisher dans les Topics suivantes:

- position
- surface
- ~output

et Subcriber aux Topics suivants:

- /camera/image_rect_color

Le code, les fonctions et les algorithmes que j'utilise proviennent principalement du fichier de **detect.py** dans TP5. J'ai simplement ajouté quelques éléments à mon projet.

Le premier est **self.position = Point32()** et **self.target_surface = Float32()**. Ils m'aident à déterminer l'emplacement et la surface de la cible.

La seconde est la commande est

```
contours, hierarchy = cv2.findContours(mask, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_NONE)
```

Si **longueur(contours) == 0** alors la surface de la cible sera 0 et la position de la cible prendra les paramètres comme x=y=z=0. Sinon, il calculera la surface et déterminera l'emplacement de la cible. Je peux déterminer le centre (centre) et le rayon (rayon) pour calculer si la cible est un cercle. Ou utilisez **boundingRect** pour calculer si la cible est un rectangle.

Ensuite je réutilise cette partie:

```
try:
    img_bgr = self.bridge.imgmsg_to_cv2(msg, "bgr8")
except CvBridgeError as e:
    rospy.logwarn('ROS->OpenCV %s', e)
    return

#calcul width
width = img_bgr.shape[1]
```

Et calculez la largeur, déterminant ainsi si la cible est à gauche, à droite ou devant. Par exemple **center < width/3** alors la cible est à gauche et **position.x=1.**

```
if ( center < width/3 ):  #left
    self.position.x=1
```

2. Class moving

De même, noeud **moving** Publisher dans les Topics suivantes:

- /cmd_vel

et Subcriber aux Topics suivants:

- positon
- surface
- /scan

Car la relation entre le robot et les obstacles est la distance. J'ai fixé cette limite à 10cm(0.1). Donc, si nous utilisons la distance pour établir un contact entre la

cible et le Robot, c'est une similitude. Par conséquent, le robot traitera en fonction de la surface de la cible qu'il observe.

D'abord, si le Robot est entouré d'obstacles, il restera bien sûr immobile.

```
if(distance_max <= distance_obstacle):
```

Si la surface est grande (trop petite), la distance est suffisante pour observer (trop loin). `elif(target_surface < 5000):`

Si la surface est suffisamment petite, le robot se déplacera vers la cible. `elif(target_surface <= 30000):`

Je réutilise aussi la fonction

```
for i, theta in enumerate(np.arange(msg.angle_min, msg.angle_max, msg.angle_increment)):
    # ToDo: Remove points too close
    if msg.ranges[i] < 0.1:
        continue
    # ToDo: Polar to Cartesian transformation
    coords.append([msg.ranges[i]*np.cos(theta), msg.ranges[i]*np.sin(theta)])
```

qui est appris en TP4 dans le fichier **transform.py.**

Sur la base du tableau de **coords**, je calcule la distance de tous les emplacements par rapport au Robot. Puisque je choisis Robot comme origine, la position du robot est constante avec tous les calculs de position, le calcul de distance devient plus simple comme suit:

```
distance = []
for point in coords:
    distance.append(np.sqrt(point[0]*point[0]+point[1]*point[1]))
```

Avec la direction, mon idée est de diviser l'espace sur 360 degrés dans les 4 directions que j'ai mentionnées plus tôt. Voici la division spécifique:

- de 316 à 45 est forward
- de 46 à 135 est right

- de 136 à 225 est de back
- de 226 à 315 est left

Mais en fait, lorsque j'essaie d'exécuter le programme, le nombre de distances atteintes n'atteint pas 360. Je n'obtiens qu'environ 200 à 250 valeurs de **distance** dans le tableau des distances. Par conséquent, j'ai choisi de diviser l'espace par des valeurs de 0 à 200, bien que cela puisse entraîner une légère distorsion de l'orientation du robot, cela garantit le fonctionnement du programme.

```
first = min(distance[0:25])
second = min(distance[175:200])
self.d_forward =  min(first,second)
self.d_right = min(distance[26:75])
self.d_back = min(distance[76:125])
self.d_left = min(distance[126:175])
```

IV. Conclusion

Bien que j'avais initialement prévu d'écrire 3 fichiers avec 3 Noeuds différents, je me suis rendu compte que les choses seraient plus compliquées. Mon algorithme n'est pas encore parfait, car je vais essayer de l'exécuter mercredi, mais dans l'ensemble, tout va bien.

J'ai décidé de rester aussi simple que possible, mais combiner TP1 avec TP4 et 5 n'était pas facile. Si l'association est trop simple, ce projet n'est pas différent de TP1, comme un robot qui ne va que dans un sens.

Bien que mon programme n'ait eu aucune erreur lors du test, la communication entre les Nœuds et les Topics n'était pas celle à laquelle je m'attendais. De plus, la réutilisation du code dans TP5 dans le but d'un disque bleu a limité mon programme. En conséquence, le Robot n'a pas fonctionné comme je l'avais prévu.

Enfin, ce projet m'aide à résumer ce que j'ai appris au cours des 5 sessions de TP ainsi qu'à revoir certaines connaissances de SY31. C'était intéressant et j'espère que ce que j'ai appris sera utilisé efficacement par moi à l'avenir.