ROSS2-D2

Drone : ROS2-D2

-Rapport challenge AquaBot-

Equipe Ross2D2

Membres du groupe :

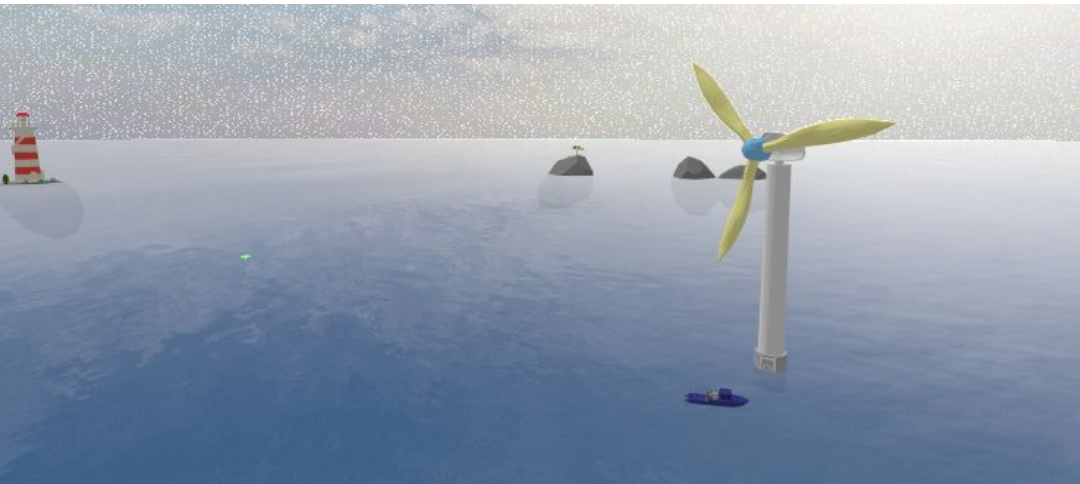
Ilyes LAMSSALAK

Manil SABEUR

Jules CUNIN

SIREHNA | AQUA.BOT

NAVAL GROUP



Sommaire

Présentation de l'équipe

Ilyes

Jules

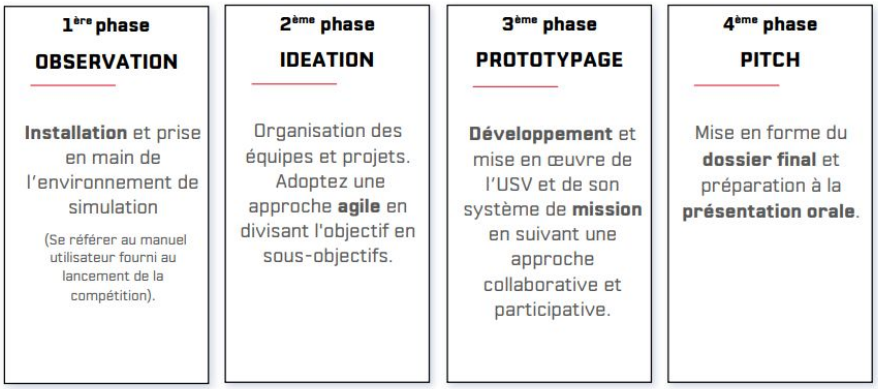
Manil

Nous sommes trois étudiants de Centrale Nantes (photo du campus plus haut) en option Données, Analyse et Traitement du Signal et de l' Image participant à ce challenge.

Centrale est une école généraliste, qui propose des options en liens étroits avec le challenge AquaBot, telles que de la Robotique, ou bien de l'Analyse et du Traitement d' Images

N'ayant initialement que très peu de connaissances sur Ros, ce challenge était un vrai défi pour nous, et nous a permis grâce à l'aide de roboticiens d'en apprendre beaucoup à travers ce projet.

Organisation de l'équipe



Observation : Jusqu'au 7 Octobre

Idéation : Du 7 au 14 Octobre

Prototypage : Du 14 Octobre au 25 Novembre

Pitch : Du 25 Novembre à la fin

Répartition des rôles

Ilyes :

- Navigation (PathPlanner et MPC)
- Manager

Jules :

- Vision (Suivi d'une éolienne)
- Partie Identification

Manil :

- PID, commande
- Navigation (PathPlanner)

Nous nous réunissons tous les lundis soirs pour mettre en commun et partager nos avancées, nous nous retrouvons souvent le samedi pour travailler ensemble sur le code.

N' étant pas familiers avec *ros2*, nous avons pris le parti d' allonger la phase d'observation pour cadrer au mieux le projet et gagner du temps par la suite

Description des algorithmes mis en place

Idée Générale

Pour compléter chaque tâche de manière efficace, en trouvant le bon rapport entre efficacité et simplicité du code, nos idées sont les suivantes :

- Inspection : Calculer le path, tout en évitant les obstacles, jusqu'à l'éolienne la plus proche pas encore visitée. Faire un tour de l'éolienne jusqu' à ce qu'on scanne le QR code, puis partir vers l' éolienne suivante. On asservi l'angle de la caméra -grâce aux informations sur la position du bateau, son cap, et la position de l'éolienne- pour qu'elle soit toujours orientée vers l'éolienne visée.

- Identification : Connaissant les valeurs d'angle de dérive et de distance envoyées par le capteur acoustique sur le bateau, mais aussi le cap du bateau et ses coordonnées dans l'environnement world, on peut calculer la position estimée de l'éolienne dont l'état est critique. On cherche à estimer à quelle éolienne (dont on connaît les coordonnées exactes) cela correspond. On s'attardera à filtrer les données avant de les envoyer dans les instructions de la partie commande du bateau.

Description des algorithmes mis en place

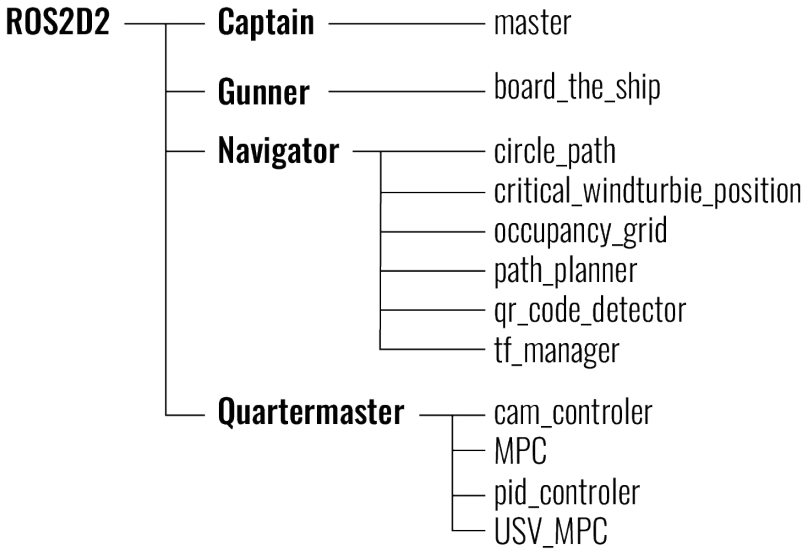
Idée Générale

- Stabilisation et maintenance : Notre idée est d' utiliser une approche similaire à l' inspection, sauf que dans le cas actuel, une fois un QR code détecté, on essayera de garder le bateau immobile. Reste encore le problème de mettre la proue, face à l' éolienne, pour le résoudre : la caméra vise l' éolienne et nous connaissons l'angle de la caméra par rapport à l'axe du bateau, on sait donc de quel angle faire pivoter le bateau.

- Round Trip : Notre idée est de travailler sur un MPC, qui pourra satisfaire les exigences de cet étape. En effet il serait beaucoup plus compliqué de maintenir le cap grâce au PID, qui suit uniquement le path sans se soucier du cap.

Description des algorithmes mis en place

Architecture du code



Approche technique

Gunner :

- *board_the_ship*

Ce fichier de lancement configure et exécute les différents noeuds nécessaires au fonctionnement de l'USV et à la réalisation des différentes tâches.

Quartermaster :

- *Cam_command*

Le nœud ajuste en continu l'orientation de la caméra pour qu'elle pointe vers une éolienne définie comme objectif, en calculant l'angle nécessaire et en publiant les commandes correspondantes.

On s' abonne ici à un topic */tf* permettant de lire les transformations entre les différents repères.

Ayant toutes les données dans la même base, on peut maintenant calculer l' angle à donner à la caméra, puis le publier sur le topic permettant de la contrôler.

- *pid_controler*

Ce noeud implémente un contrôleur PID pour ajuster la direction et la vitesse de l'USV afin de suivre un chemin planifié.

Le correcteur permet d' analyser les oscillations rapides pour réduire les comportements instables de l'USV. La vitesse est aussi gérée suivant la position du bateau, il réduit la poussée lorsque la cible approche et stoppe l'USV lorsque l' objectif est atteint.

On publie ensuite sur chaque topic gérant les propulseurs, les valeurs optimales pour avoir direction souhaitée.

- *MPC*

Ce code implémente un modèle predictive control(MPC) pour un véhicule de surface autonome (USV), permettant de suivre une trajectoire de référence en optimisant les commandes tout en respectant les contraintes dynamiques et physiques.

Pour gérer le contrôle et l'optimisation, nous avons établis des équations en annexe. Puis nous avons tout implémenté à l'aide de la librairie do_mpc en python.

Le but est donc de créer une suite de commandes pour minimiser l'erreur entre la trajectoire prévu et désirée. Pour plus de compréhension, nous avons affiché les états prévus sur rviz2.

Les équations du MPC sont disponible en annexe.

- *USV_MPC*

Ce code définit un nœud ROS2 appelé thruster_control_node_py, chargé de contrôler les propulseurs d'un véhicule de surface autonome (USV) à l'aide d'un contrôleur prédictif basé sur le modèle (MPC). Le nœud publie les commandes de poussée et d'angle pour les propulseurs gauche et droit, tout en recevant des données de capteurs (IMU, TF) et de trajectoire (Path).

Les informations de position, orientation et vitesse de l'USV sont calculées à partir des données TF et IMU. Lorsqu'un chemin de référence est reçu, le MPC génère une séquence optimale de commandes pour un horizon de prédiction de 40 étapes. Ces commandes sont publiées périodiquement pour piloter les propulseurs. En parallèle, le nœud publie une trajectoire prédit pour visualisation.

Le contrôle est exécuté toutes les 50 ms via un timer. Si toutes les données nécessaires sont disponibles, le nœud suit le chemin pré-planifié en ajustant continuellement les commandes, sinon il maintient ou ajuste les dernières valeurs. Le programme s'arrête proprement via une méthode main.

Navigator :

- *circle_path*

Ce programme implémente un nœud ROS2 nommé path_publisher qui génère et publie un chemin circulaire autour d'une éolienne cible pour un véhicule autonome de surface (USV).

Le nœud fonctionne grâce à deux abonnements : Aux transformations TF, pour déterminer la position actuelle de l'USV et à un message PoseStamped, pour obtenir la position de l'éolienne (cible).

Lorsqu'une position d'éolienne est reçue, le nœud génère des waypoints formant un cercle autour de cette position, avec un rayon défini (10 m par défaut). Les waypoints incluent une orientation dirigée vers l'éolienne.

Toutes les secondes, le nœud vérifie si l'USV a atteint le waypoint suivant, basé sur une distance seuil (1 m par défaut). Si le waypoint est atteint, il est supprimé, et le chemin mis à jour est publié sur le topic /navigation/circle_path.

Le calcul des waypoints repose sur une incrémentation angulaire uniforme pour générer des positions circulaires. Les orientations des waypoints sont exprimées en quaternions, calculées pour pointer vers l'éolienne.

Le programme démarre et gère les interactions principales via la méthode main, assurant une exécution propre avec rclpy.

- *occupancy_grid*

Ce script définit un nœud nommé *OccupencyGrid*, qui publie une carte d'occupation fixe sur le topic *navigator/occupencygrid.* La carte est chargée à partir d'un fichier .npy et représente une grille de 600x600 cellules, avec chaque cellule exprimant une probabilité d'occupation en pourcentage.

Le nœud initialise un message *OccupancyGrid* avec des métadonnées spécifiques :

- **Résolution** : 1 mètre par cellule.
- **Origine** : Coordonnée en bas à gauche, positionnée à (-300, 300).
- **Orientation** : Une rotation de -π/2 est appliquée pour aligner la grille correctement.

La carte est publiée toutes les secondes, avec les données stockées dans un tableau aplati. Le nœud reste actif jusqu'à son arrêt explicite.

- *path_planner*

Ce script crée un nœud *PathPublisherNode* qui calcule et publie un chemin pour un robot tout en évitant des obstacles. Le nœud utilise la bibliothèque *PyVisGraph* pour gérer la navigation dans un environnement avec plusieurs îles et éoliennes comme obstacles.

Le nœud commence par s'abonner à deux topics : */captain/goal_pose* pour obtenir la position cible et /tf pour récupérer la position actuelle du robot et des éoliennes via les transformations tf (avec le callback tf_listener_callback). Dans le callback *goal_pose_callback*, la position cible est récupérée, et à chaque itération du timer (fonction timer_callback), le chemin est recalculé.

Le calcul du chemin est effectué en utilisant la méthode *shortest_path* de *VisGraph*, qui prend en compte la position actuelle et la position cible dans un cadre de référence adapté (transformé via les fonctions *world_to_matrix* et *matrix_to_world*).

En plus du chemin, des polygones sont générés autour des éoliennes à l'aide de la méthode *add_polygone_around*, ce qui permet d'éviter ces obstacles. Chaque éolienne est identifiée grâce à son identifiant tf, et un polygone est créé autour d'elle pour la considérer comme un obstacle dans le calcul du chemin.

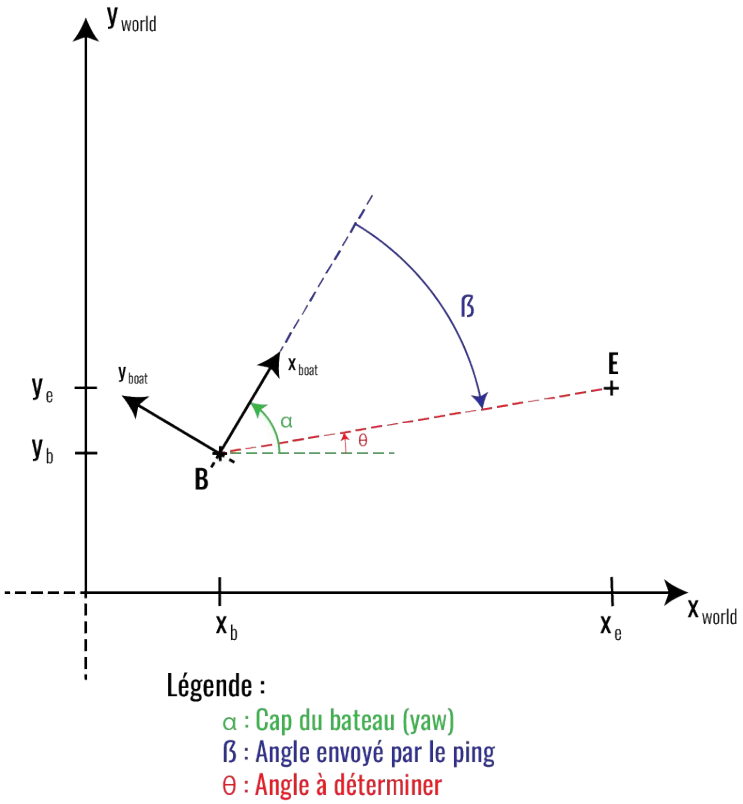
Une fois le chemin trouvé, il est enrichi de courbes (en utilisant la méthode *add_circle*) pour adapter la trajectoire si nécessaire, en respectant un rayon de giration minimal. Le chemin est ensuite publié sous forme de message *Path* sur le topic */navigator/path.* En parallèle, les obstacles sont publiés sous forme de marqueurs sur le topic */navigator/islands*, permettant de les visualiser dans RViz.

Ainsi, ce nœud gère à la fois la planification de trajectoire et la visualisation des obstacles en temps réel.

- ## *Critical_windturbine_position*

Grâce à l'angle theta, on peut passer des coordonnées polaires, données par le ping, en coordonnées cartésiennes, et donc trouver (x_e,y_e), car on connaît déjà (x_b,y_b)

Schéma du problème



L' approche mathématique étant faite, le principe du node est le suivant : on calcule la position estimée de l'éolienne grâce à la position du bateau et les informations du ping. Ensuite, on regarde dans la liste contenant les positions exactes des éoliennes, laquelle est la plus proche de notre estimation, on stocke ses coordonnées dans la liste *detected_turbines.* On regarde ensuite quelles coordonnées ont le plus d'occurrences dans les 10 derniers termes de la liste, on fait en fait une médiane glissante sur le temps pour parer le fort bruit du capteur.

On publie ensuite ces coordonnées sur un topic qui sera récupéré par le manager

- ## *qr_code_detector*

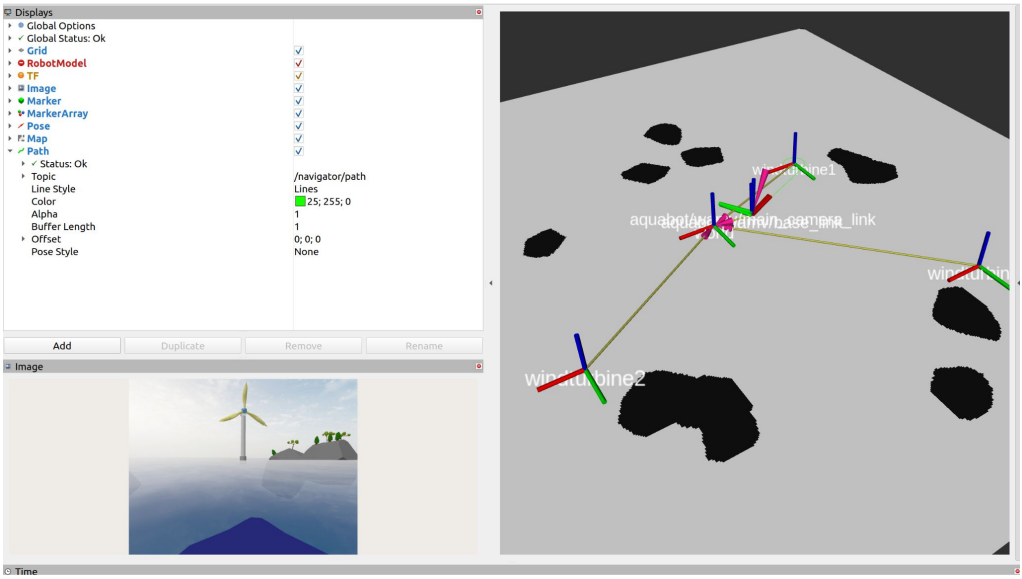
Ce script crée un nœud qui reçoit des images d'une caméra, utilise OpenCV pour détecter et décoder des QR codes, puis publie les données extraites sur un topic. Le nœud abonne à */aquabot/sensors/cameras/main_camera_sensor/image_raw* pour recevoir les images, utilise *CvBridge* pour convertir les messages ROS en images OpenCV, et utilise *QRCodeDetector* d'OpenCV pour extraire les données des QR codes. Si un QR code est détecté, les données sont publiées sur */navigator/qr_code_data.*

- ## *manager*

Ce code définit un nœud *TfPublisher* qui publie des transformations (TF) pour l'Aquabot à partir de différentes sources de données. Il s'abonne aux topics du GPS (*/aquabot/sensors/gps/gps/fix*), de l'IMU (*/aquabot/sensors/imu/imu/data*), des états des joints (*/aquabot/joint_states*) et des positions des éoliennes (*/aquabot/ais_sensor/windturbines_positions*). Les données GPS sont converties en coordonnées cartésiennes grâce à la fonction *gps2cartesian_from_lat_lon*, puis utilisées pour mettre à jour la transformation de la base du robot, via *self.base_link_transform*.

L'orientation du robot est récupérée de l'IMU et appliquée à cette transformation dans la méthode *imu_callback*. Pour la caméra, l'angle du joint est extrait des états des joints dans *joint_states_callback*, et la transformation correspondante est mise à jour dans update_camera_tf, où la rotation est calculée à partir de l'angle du joint et d'une rotation fixe. Enfin, les positions des éoliennes sont également transformées en coordonnées cartésiennes et publiées sous forme de transformations dans *windturbines_callback*. Le nœud publie ainsi en continu des informations sur la position et l'orientation de l'Aquabot ainsi que des éoliennes, permettant une localisation précise dans l'espace.

Environnement rviz



Environnement rviz avec les repères des différents objets et les obstacles

Problèmes rencontrés

Le MPC a été complexe à mettre en place, le travaille de recherche au niveau de la modélisation et des librairies de contrôle disponibles. De plus le système étant complexe, trouver d'où proviennent les erreurs n'est pas une tâche aisée. Nous pensons que dans notre cas, des problèmes de transformations induisent des erreurs de signes que nous n'avons pas résolues. De plus nous comptions sur le MPC pour la réalisation des tâches de contrôle plus complexes notaments pour l'étape bonus dont un noeuds de trajectoire était prévu : "*circle_path*".

Ensuite, l'avancement du code en groupe a nécessité l'utilisation de d'outils de github comme les forks au sein du groupe où tout le monde travaillait sur son code et faisait une *pull request* pour chaque module finit sur la branche principale.

La prise en main de l'environnement a évidemment été un facteur ralentissant dans la poursuite de notre objectif.

Guide d'installation et d'utilisation de ROS2-D2

Packages non habituels :

- `pyvisgraph`
- `collections`
- `do_mpc, casadi`

Commande de lancement :

```
ros2 launch gunner board_the_ship.launch.py
```

Nous vous fournissons la configuration de rviz2 pour suivre la position et l' orientation de tous les objets en présences.

ANNEXES

Modèle et cost function du MPC

$$\dot{\mathbf{x}} = f(\mathbf{x}, \mathbf{u}) = \begin{bmatrix} u\cos\phi - v\sin\phi \\ u\sin\phi + v\cos\phi \\ r \\ \dfrac{K\omega_1\cos\theta_1 + K\omega_2\cos\theta_2 - x_U u - x_{UU}u|u| + mvr}{m} \\ \dfrac{K\omega_1\sin\theta_1 + K\omega_2\sin\theta_2 - y_V v - y_{VV}v|v| - mur}{m} \\ \dfrac{x_L K\omega_1\sin\theta_1 - y_L K\omega_1\cos\theta_1 + x_R K\omega_2\sin\theta_2 - y_R K\omega_2\cos\theta_2 - n_R r - n_{RR}r|r|}{I_z} \end{bmatrix}$$

où

$$\mathbf{x} = \begin{bmatrix} x \\ y \\ \phi \\ u \\ v \\ r \end{bmatrix}, \quad \mathbf{u} = \begin{bmatrix} \omega_1 \\ \omega_2 \\ \theta_1 \\ \theta_2 \end{bmatrix}$$

Minimiser

$$J = \int_t^{t+N} \left[(\mathbf{x} - \mathbf{x}_{\text{ref}})^\top Q(\mathbf{x} - \mathbf{x}_{\text{ref}}) + \mathbf{u}^\top R\mathbf{u}\right] dt + (\mathbf{x}(t+N) - \mathbf{x}_{\text{ref}}(t+N))^\top Q(\mathbf{x}(t+N) - \mathbf{x}_{\text{ref}}(t+N))$$

$$\dot{\mathbf{x}} = f(\mathbf{x}, \mathbf{u})$$

$$\begin{cases} -5000 \leq \omega_1 \leq 5000 \\ -5000 \leq \omega_2 \leq 5000 \\ -\dfrac{\pi}{4} \leq \theta_1 \leq \dfrac{\pi}{4} \\ -\dfrac{\pi}{4} \leq \theta_2 \leq \dfrac{\pi}{4} \end{cases}$$