Ensta Bretagne

Promotion 2021, Robotique autonome, UE 5.4

Projet Magmap

Encadrant : Luc Jaulin

Table des matières

Remerciements

Nous remercions Nathan Fourniol et Alain Bertholom pour leur aide concernant l'AUV Riptide ainsi que Fabrice Le Bars pour son aide à la mise en œuvre des robots.
Nous remercions également Romain Schwab pour son aide à l'utilisation du magnétomètre et Gilles Le Maillot pour la préparation du câble long de celui-ci.
Enfin, nous remercions Luc Jaulin pour son aide sur l'ensemble du projet.

Introduction

Résumé

Chaque année, l'ensemble des étudiants de la spécialité robotique réalise un projet de groupe. Cette année, ce projet s'intitule Magmap, pour carte magnétique. Le but du projet est de réaliser des cartes magnétiques terrestres et sous-marines à l'aide de robots autonomes. Les cartes magnétiques permettent de détecter des objets enfouis dans le sol, des épaves, des mines... Deux robots ont été utilisés dans le cadre de ce projet : Saturne (robot terrestre - UGV) et un Riptide micro-UUV MKII (robot sous-marin - UUV/AUV).

Abstract

Each year, all the students of the autonomous robotics speciality work on a common project. This year, this project is called Magmap, for magnetic map. The aim of the project is to generate terrestrial and underwater magnetic maps using autonomous robots. Magnetic maps can be used to detect objects buried in the ground, ship wrecks, mines... Two robots have been used in this project: Saturne (Unmanned Ground Vehicle - UGV) and a Riptide micro-UUV MKII (Unmanned Underwater Vehicle - UUV/AUV).

Répartition des tâches

Afin de réaliser ce projet, celui-ci a été découpé en 12 tâches. Pour les parties liées à chacun des robots ainsi que le projet global, des architectes ont été désignés.

- ➤ Transverse :
  - → Communication : Julien et Paul
  - → Théorie et utilisation pratique d'un magnétomètre : Agathe, Alexandre, Robin
  - → Réalisation de mesures avec le robot Saturne : Bertrand, Kévin et Robin
- ➤ Robot Saturne :
  - → Prise en main et contrôle du robot : Bertrand, Maha et Mamadou
  - → Simulation : Bertrand
  - → Prévision des trajectoires : Colin et Kévin
  - → Post-processing des données : Gwendal, Jules, Paul-Antoine et Quentin B
  - → Architecte : Kévin
- ➤ Robot Riptide :
  - → Prise en main et architecture électronique : Corentin et Hamid
  - → Simulation : Mourtaza, Quentin V et Romane
  - → Contrôle du robot : Julien et Paul
  - → Architecte : Hamid
- ➤ Architecte global : Robin

Utilisation d'un magnétomètre

1.1 Présentation du magnétomètre utilisé

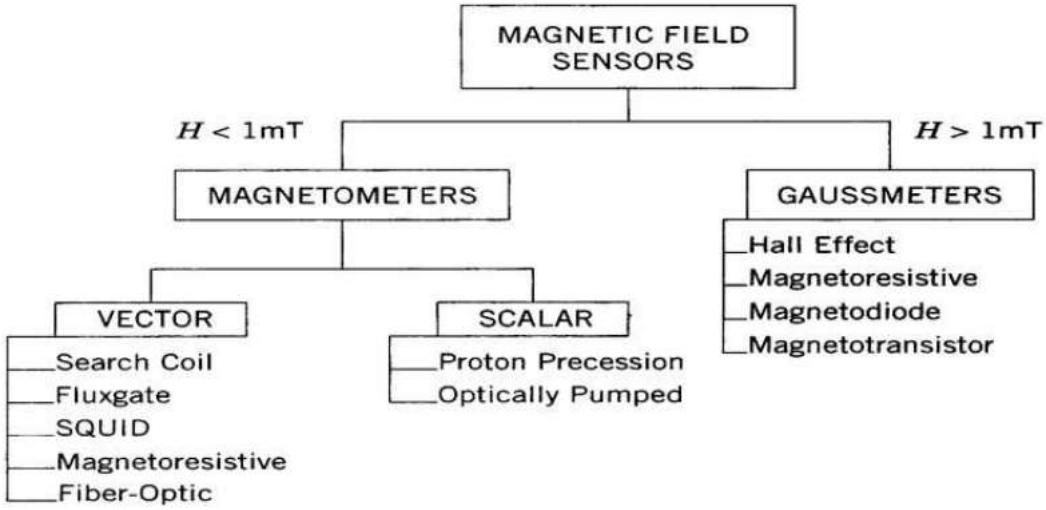

Figure 1 : Types de capteurs de champ magnétique

Les capteurs de champ magnétique permettent de mesurer l'aimantation (magnetization) de matériaux magnétiques, par exemple des objets ferreux, et la direction du champ magnétique en un point donné en unité Tesla (T). Deux grandes familles sont à distinguer : les gaussmètres adaptés aux champs supérieurs à 1 mT, et les magnétomètres pour ceux inférieurs à 1 mT. Enfin, la catégorie des magnétomètres regroupe elle-même des capteurs scalaires, capables d'acquérir une valeur absolue en un point, et les capteurs vectoriels ou axiaux fournissant la valeur du champ selon un ou plusieurs axes. [Haq16]
Le projet demande à utiliser le magnétomètre FGM3D/100 de Sensys Magnetometers, qui est un magnétomètre *fluxgate* tri-axial. En comparaison avec les autres capteurs existants, le modèle *fluxgate* a plusieurs avantages :

→ Une grande sensibilité de mesure dans la catégorie des magnétomètres axiaux mais bien qu'inférieure à celle fournie par un magnétomètre scalaire

→ Une mesure selon trois axes

→ Une faible consommation

→ Une petite taille

Les contraintes associées à ce modèle ne sont pas problématiques pour le cas d'étude : la plage limitée aux champs magnétiques faibles est recherchée et la bande passante peut éliminer des bruits parasites. En effet, des champs magnétiques de l'ordre du champ magnétique terrestre de Brest, valant 4.6 µT, devraient être observées.
Ces caractéristiques sont retrouvées au travers de celle du FGM3D/100 [Mag15] :

Consommation	3 W
Poids	120g
Plage de mesures	±150 µT
Sensibilité	70 pT ($10^{-}15$ T)
Erreur relative de mesure	±0.1 %
Déclinaison entre les axes	0.5°
Bande-passante	2kHz
Bruit	< 10 pT $/\sqrt{Hz}$ @ [0.1,10] Hz

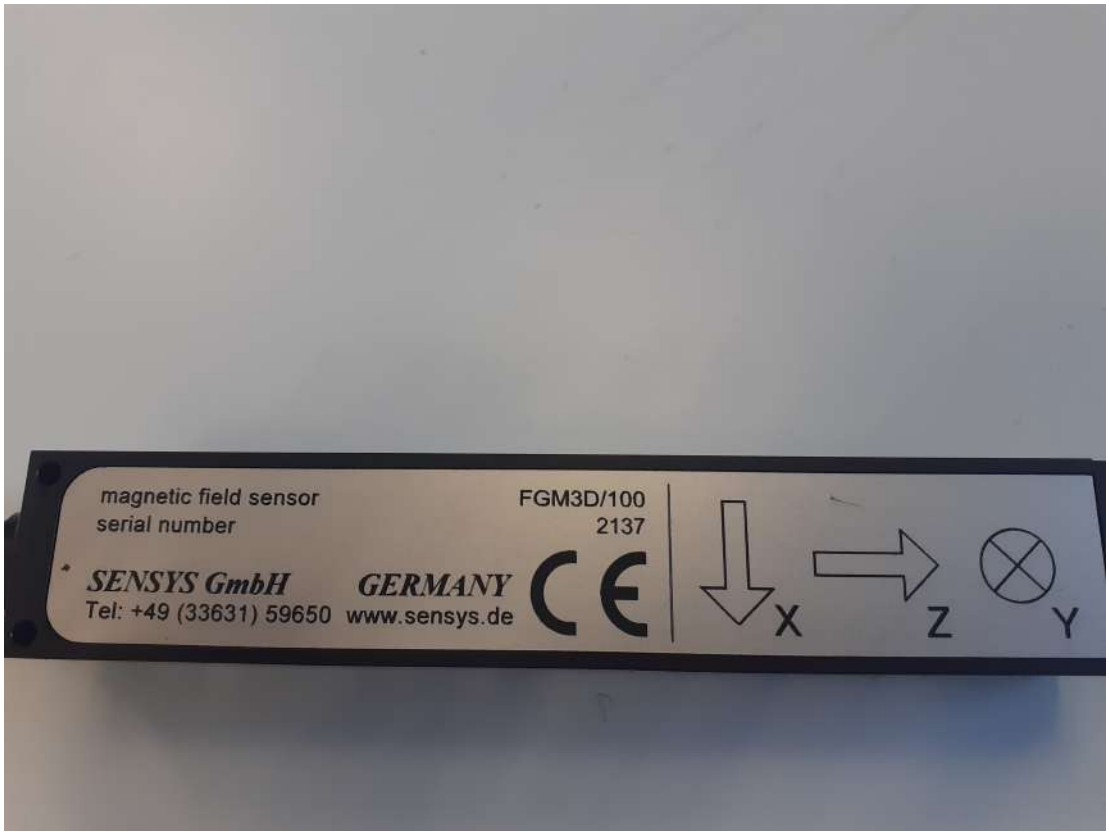

Figure 2 : Magnétomètre FGM3D/100

1.2 Théorie du magnétomètre *fluxgate*

Le magnétomètre *fluxgate* mono-axial, ou magnétomètre à saturation, est composé de trois grands éléments :

→ Un matériau magnétique facilement saturable (corps)

→ Une bobine d'excitation délivrant un courant alternatif

→ Une bobine de mesure estimant la force électromotrice induite dans le matériau

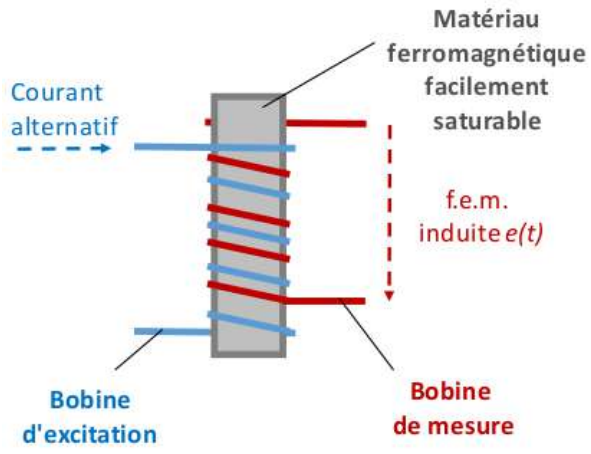

Figure 3 : Composants du magnétomètre fluxgate

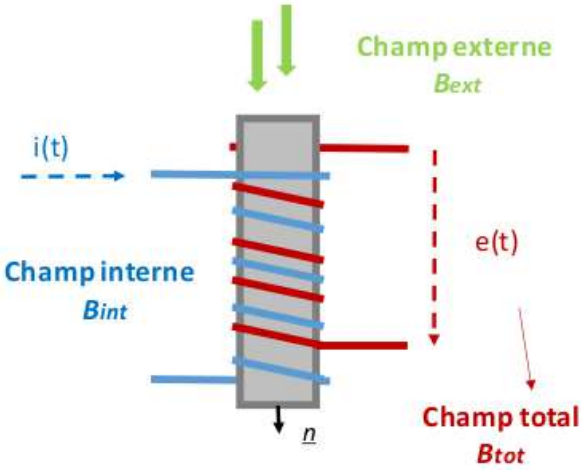

Figure 4 : Les différents acteurs lors des mesures

Soit B_{int} le champ magnétique produit par le courant alternatif $i(t)$, B_{ext} le champ magnétique extérieur (à mesurer) s'appliquant au corps du magnétomètre, B_{tot} le champ résultant total, $e(t)$ la force électromotrice induite mesurée par le magnétomètre. $\overrightarrow{n}$ désigne un vecteur perpendiculaire à la surface des spires de la bobine de mesure, avec $|\overrightarrow{n}| = n$ le nombre de spires par bobine.
La loi de Faraday donne :

$$e(t) = -\frac{\mathrm{d}\phi(t)}{\mathrm{d}t} \text{ avec } \phi \text{ le flux magnétique dans la bobine de mesure}$$

Or

$$\phi(t) = \iint_s \overrightarrow{B_{tot}}(t) \cdot \overrightarrow{n}\, dS = nSB_{tot\perp}(t) \text{ avec } S \text{ la surface d'une spire}$$

D'où :

$$e(t) = -\frac{\mathrm{dnSB_{tot\perp}(t)}}{\mathrm{d}t} = -nS\frac{\mathrm{d}B_{tot\perp}(t)}{\mathrm{d}t}$$

On en déduit :

$$B_{tot\perp} = \frac{1}{nS}\int_{t0}^{tf} e(t)dt$$

Finalement :

$$B_{ext\perp} = \frac{1}{nS}\int_{t0}^{tf} e(t)dt - B_{int\perp}$$

Pour obtenir les deux autres composantes axiales du champ magnétique externe, trois magnétomètres *fluxgate* mono-axiaux sont assemblés orthogonalement les uns par rapport aux autres. [Vid13] [SA17]

1.3 Théorie détection d'objets sous-terrain

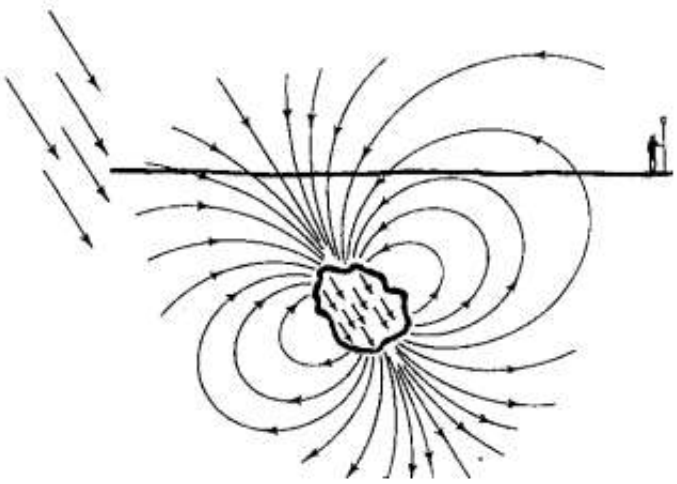

Figure 5 : Exemple de champ magnétique émis

Le projet Magmap a pour finalité la détection de l'aimantation d'objets ferromagnétiques. L'aimantation d'un objet est influencée par son aimantation propre permanente et sa capacité à s'aimanter sous l'influence du champ magnétique terrestre selon sa composition. [BR90]
Elle est définie selon :

$$\overrightarrow{M} = \overrightarrow{M}_{permanent} + K_{eff} * \overrightarrow{H}_{Terre}$$

Avec M l'aimantation (Am^2), K_{eff} la susceptibilité magnétique de l'objet (J/T2), et $H_{Terre} = 4.6 \times 10_{-6}T$ à Brest.
L'induction magnétique ainsi perçue en un point devient :

$\overrightarrow{B} = \frac{\mu_0}{4\pi}\left[3\frac{\overrightarrow{M}\cdot\overrightarrow{r}}{r^5}\overrightarrow{r} - \frac{\overrightarrow{M}}{r^3}\right]$ avec r la distance de l'objet magnétique, $\mu_0 = 4\pi * 10^7$ H/m.

La présence d'un objet magnétique se traduit par une anomalie détectée par le magnétomètre [Sch18] :

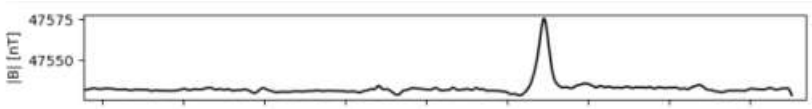

Figure 6 : Exemple d'anomalie magnétique

Des formules empiriques donnent une vérification simple pour la détectabilité d'un objet en acier :

$\left|\overrightarrow{M}\right| = K \times Poids$ avec M l'aimantation et $K = 0.1$ Am2/kg

$\left|\overrightarrow{B}\right| = 100\frac{M}{r3}$ avec B le champ magnétique en nT, r la distance à la source en m.

Pour un objet en acier de 100g à 30 cm de profondeur :
$M = 0.01\ Am^2$ et $B = 37$nT
L'objet est donc théoriquement détectable par le magnétomètre FGM3D/100.

1.4 Simulation du magnétomètre

La bibliothèque Magpylib a été utilisée sous Python pour reproduire et valider la détection de différents objets ferromagnétiques, à partir de la formule de la valeur en un point de l'induction magnétique générée par une source.
Pour imiter au mieux les manipulations futures réalisées en conditions réelles, différentes entités magnétiques de différentes formes (sphère, cylindre, plan) et de différentes aimantations sont placées à une certaine profondeur. Le magnétomètre, un cylindre magnétique, est alors translaté sur le plan du sol ou suit une trajectoire en boustrophédon sur ce même plan. La carte ou les mesures magnétiques sont issues de la norme du champ magnétique mesuré au niveau du magnétomètre.

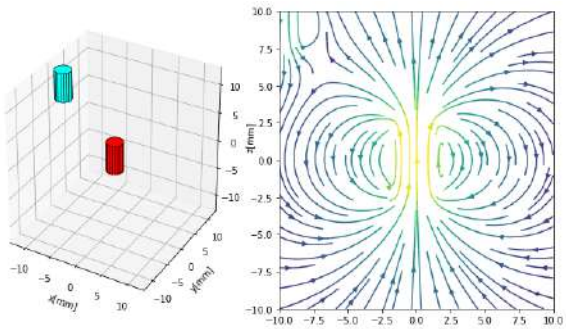

Figure 7 : Configuration de la simulation

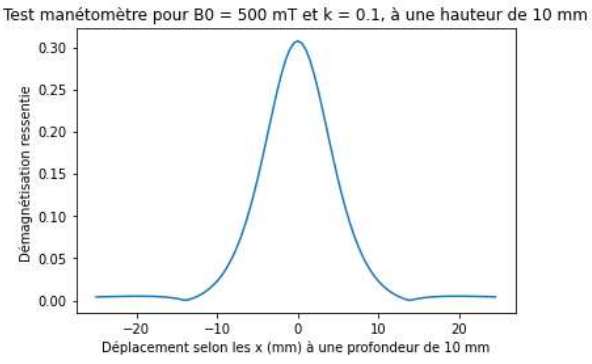

Figure 8 : Mesures dans le plan X-Z après passage d'un magnétomètre

La forme et la profondeur des objets peuvent changer légèrement l'amplitude des anomalies magnétiques perçues. Leur orientation change la forme des anomalies observées :

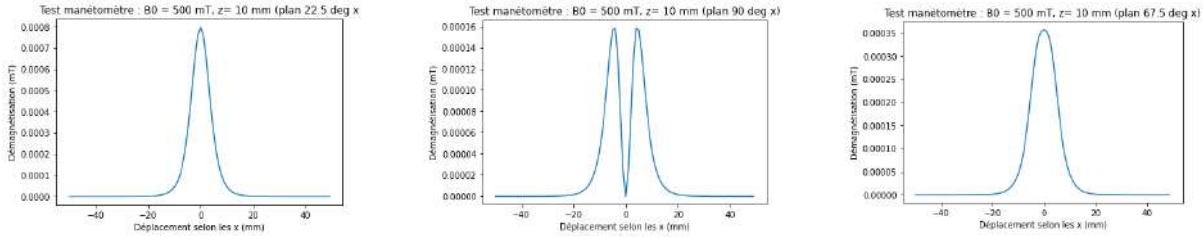

Figure 9 : Anomalies obtenues pour plusieurs orientations d'un même objet

Ces profils, ici en valeur absolue, sont en réalité caractéristiques des signatures magnétiques. En effet, toute anomalie est théoriquement combinaison linéaire des trois fonctions d'Anderson :

$y_0(x) = \frac{1}{(1+x^2)^{5/2}}$
$y_1(x) = \frac{x}{(1+x^2)^{5/2}}$
$y_2(x) = \frac{x^2}{(1+x^2)^{5/2}}$

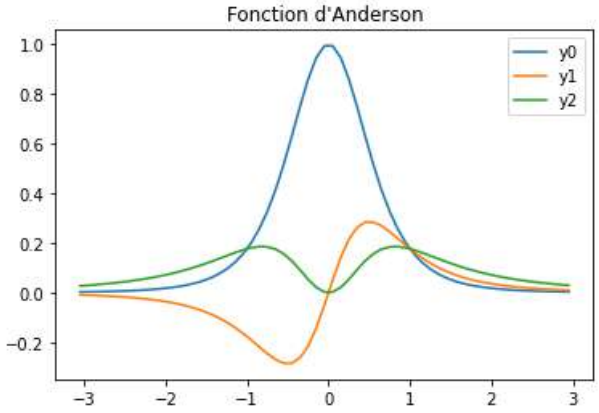

Figure 10 : Les trois courbes d'Anderson

Finalement, des cartes complètes en 2D ont été établies :

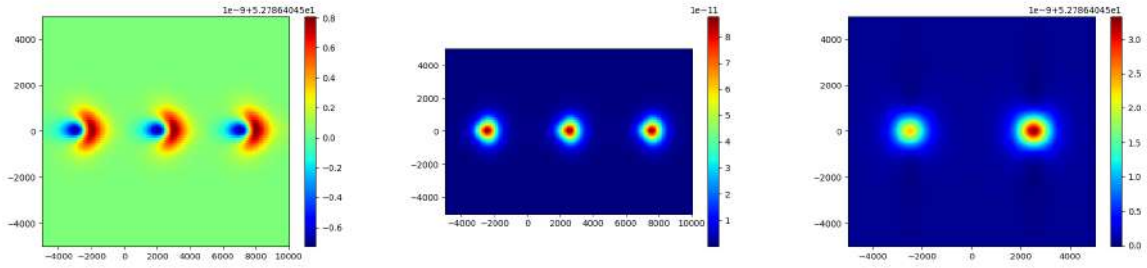

Figure 11 : Cartes magnétiques complètes obtenues en simulation

Ci-dessus, trois cylindres métalliques simulés et leur impact sur les mesures du magnétomètre. À gauche, les mesures sont affichées telles qu'elle. Les mêmes mesures sont affichées au milieu, en valeur absolue.
L'image de droite montre un test entre deux cylindres, l'un des deux bien plus proche du magnétomètre que l'autre. Le résultat est clair, un des deux cylindres est bien plus visible que l'autre. Les profils obtenus sont encore une fois caractéristiques des fonctions d'Anderson.

1.5 Mise en place d'un filtre d'Anderson

Lors de la simulation du magnétomètre, il a été vérifié que, lorsque celui-ci se déplace à une vitesse constante en ligne droite, toute anomalie magnétique est théoriquement combinaison linéaire des trois fonctions d'Anderson. [Sch18]
L'existence d'une telle décomposition peut alors servir à la construction d'un filtre pour éliminer une partie des bruits de mesure après acquisition.
En considérant un signal bruité issu des mesures du magnétomètre, ce filtre se décompose en trois étapes :

→ Corrélation croisée du signal bruité s avec chaque fonction d'Anderson y_0, y_1, y_2 :

$$corr_{s/y_{[0,1,2]}}[k] = \sum_{i=0}^{n} s[i+k] * \overline{y_{[0,1,2]}[i]}$$

avec $\overline{y}$ désignant le conjugué complexe et n le nombre de valeurs du signal s.

→ Obtention des composantes issues de la projection du signal bruité sur la base d'Anderson (produit scalaire) :

$$\forall i \in [0,1,2],\ a_i = s \cdot y_i$$

→ Reconstruction du signal à partir de sa projection sur la base (à un facteur d'échelle près) :

$$s_{theorique} = \sum_{i=0}^{2} a_i \times corr_{s/y_i}$$

Ci-dessous, un exemple de données acquises par le magnétomètre lors d'une courte mission, avant et après passage par le filtre d'Anderson. Pour ce cas donné, les anomalies sont bien conservées, et le signal paraît plus lisse.

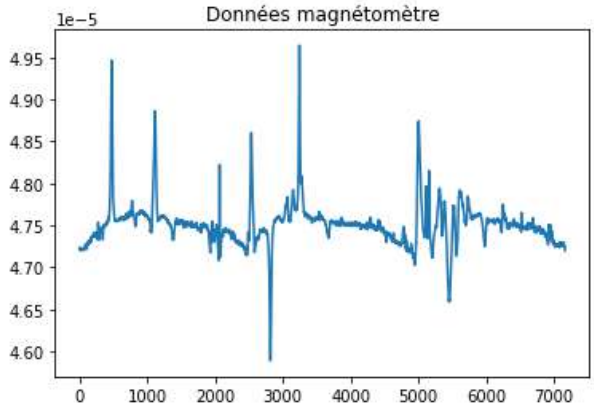

Figure 12 : Signal brut du magnétomètre

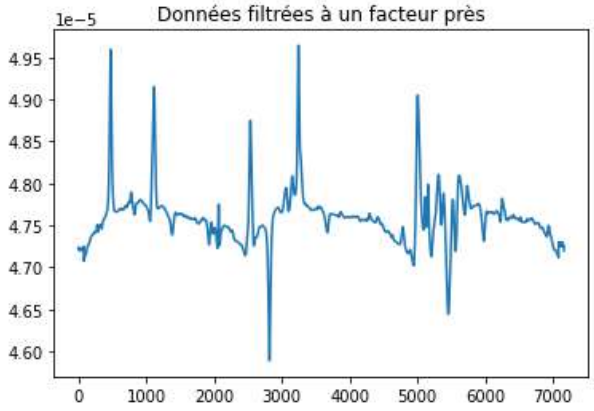

Figure 13 : Signal filtré par les fonctions d'Anderson

Il est à noter toutefois que quelques valeurs, en tout début et toute fin du signal, ne sont pas filtrées à cause d'un décalage généré par l'étape de corrélation croisée.
Pour des missions plus longues, le filtre d'Anderson se révèle inexploitable à cause de nombreux virages sur la zone à mesurer (trajectoire en boustrophédon du robot), où la vitesse varie. L'hypothèse de déplacement en ligne droite à vitesse constante du magnétomètre n'est alors plus vérifiée.

1.6 Utilisation pratique d'un magnétomètre

1.6.1 Connexion

Un digitaliseur permet de connecter le magnétomètre à un ordinateur. La récupération des données s'effectue à l'aide d'un logiciel propriétaire qui envoie une clé d'authentification au digitaliseur. Les données peuvent ensuite être lues, enregistrées au format CSV ou envoyées sur un port série (avec un baudrate maximal de 921600 baud). Cette fonctionnalité est particulièrement intéressante car elle permet de traiter les données à l'aide d'autres logiciels et de les synchroniser (dans la mesure du possible) avec d'autres capteurs comme un récepteur GNSS.

Figure 14 : Digitaliseur du magnétomètre

Dans le cas des mesures avec le robot Saturne, un récepteur GNSS était fixé sur la luge contenant le magnétomètre. En effet, il est nécessaire d'éloigner le robot du magnétomètre, ici en le tractant dans une luge derrière le robot, afin que les moteurs ne perturbent pas la mesure.
Au moyen de ports série virtuels, il est possible de transmettre les données magnétiques à une machine virtuelle Ubuntu. Cette dernière recevant aussi les données du récepteur GNSS placé dans la luge, il est possible d'obtenir la norme de champ magnétique en une position GNSS donnée.
Le transfert des données implique une baisse de la fréquence de publication des données magnétiques. Les données brutes sont fournies à une fréquence de 400 Hz. Afin de ne pas saturer la liaison, les données fournies via le port série sont issues d'une moyenne glissante effectuée par le logiciel sur 40 échantillons.

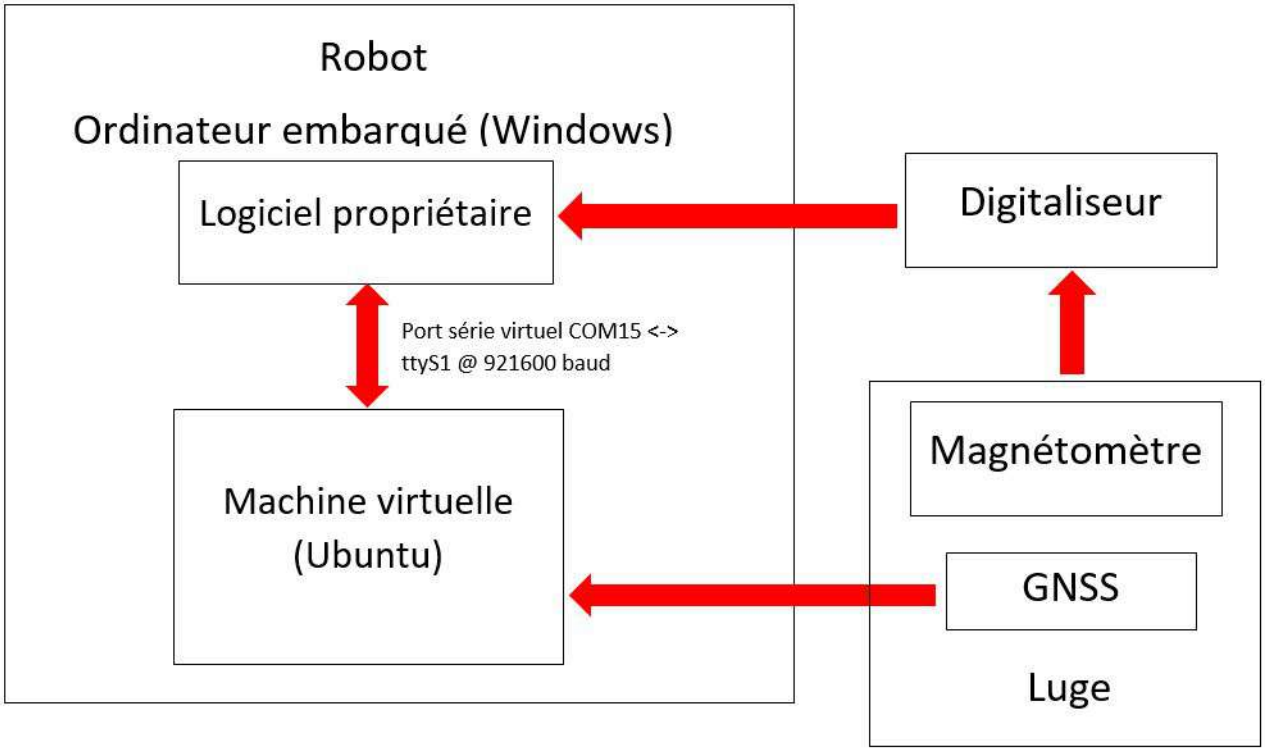

Figure 15 : Connexion du magnétomètre au PC / machine virtuelle

1.6.2 Filtrage

En utilisant les données brutes du magnétomètre (fournies à 400 Hz), on détecte une perturbation dont la fréquence est de 50 Hz. Celle-ci est probablement due au courant électrique. En utilisant un filtre de type moyenneur glissant, on réussit à la faire disparaître. Le logiciel propriétaire utilise ce même type de filtre pour diffuser les données sur un port série.

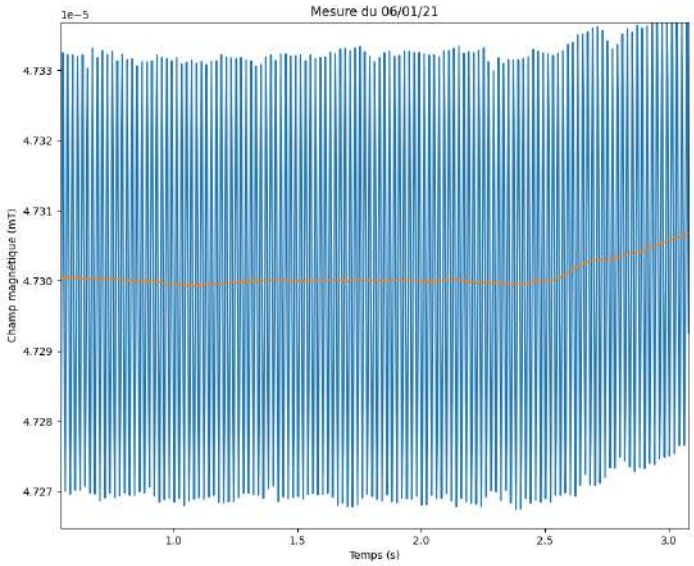

Figure 16 : Bruit à 50 Hz

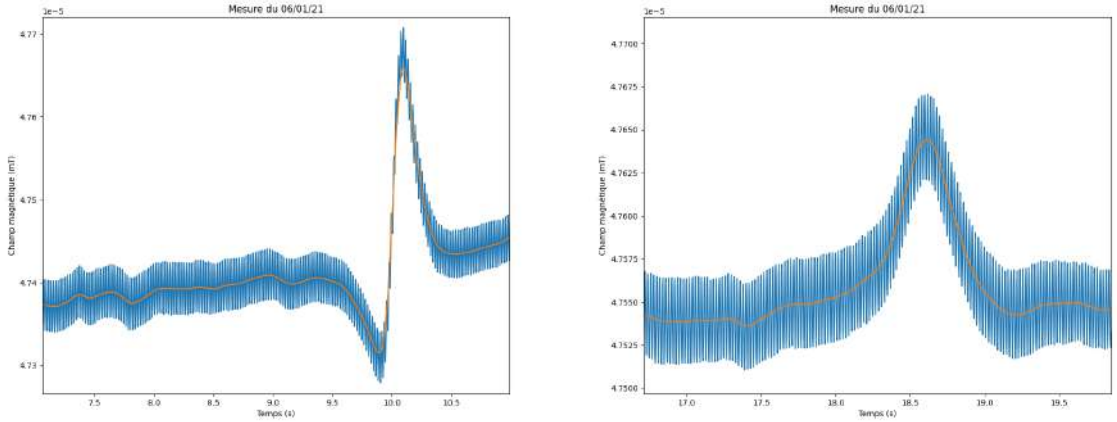

Figure 17 : Anomalies magnétiques

1.6.3 Interprétation

Afin de détecter les anomalies magnétiques, nous avons fait le choix de ne pas utiliser les fonctions d'Anderson en raison du nombre de virage assez important effectué par le robot.
La détection passe donc par la recherche des maximums et des minimums locaux. Lorsque la différence entre un minimum local et un maximum local successifs est suffisamment importante sur une durée fixée, on considère le point comme anomalie.

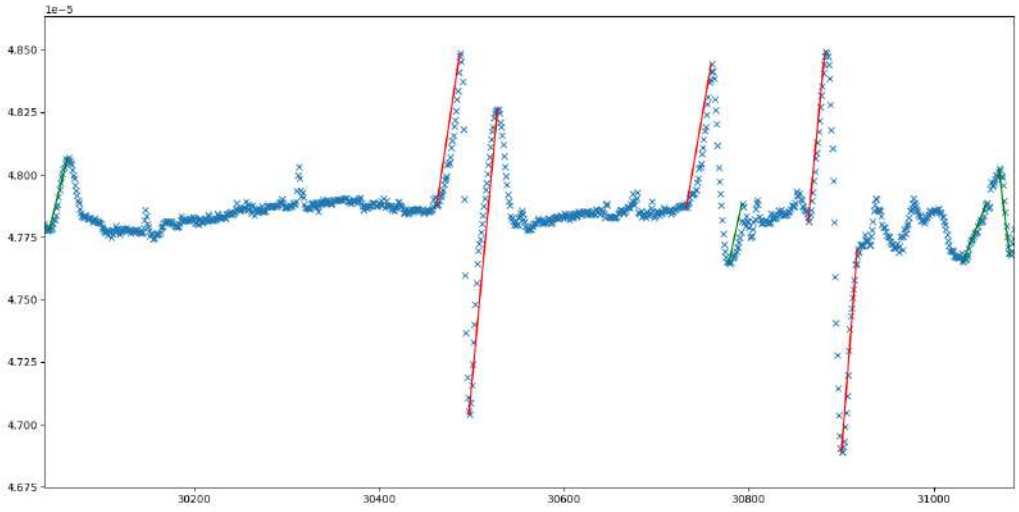

Figure 18 : Détection d'anomalies

Prise en main et contrôle des robots

2.1 Robot Saturne

Saturne est un robot à 4 roues motrices fabriqué en grande partie à l'ENSTA Bretagne. Il est composé de 2 parties indépendantes. La partie inférieure contient les 4 moteurs, les batteries et l'électronique de puissance. Elle peut être directement commandée avec une radiocommande. La partie supérieure est la charge utile, nécessaire pour rendre le robot autonome. Elle est composée d'un ordinateur de bord, d'un module wifi longue portée et des capteurs. Le robot est étanche. En pratique, il peut fonctionner sous la pluie mais n'est pas amphibie.

Figure 19 : Le robot Saturne

2.1.1 Moteurs, batteries et sécurité

Saturne est propulsé par 4 moteurs et alimenté par 3 batteries LiPo 6S (deux batteries pour la puissance, et une pour alimenter l'électronique). On estime que le robot a une autonomie de 3 heures pour une mission en continu à 50% de la vitesse maximale.
Le robot étant puissant, 3 arrêts d'urgence ont été installés (2 sur la partie inférieure et 1 sur la partie supérieure). De plus, la radiocommande est prioritaire sur les commandes envoyées par l'ordinateur de bord, donc il suffit d'allumer la radiocommande pour reprendre le robot en main.
Le robot est aussi équipé d'un gyrophare et de lampes pour prévenir qu'il est en mode autonome. Saturne n'a pas de système spécifique pour tourner comme l'aurait une voiture, mais est dirigé en utilisant un système de différentielle de commande entre les roues de droites et les roues de gauches (modèle char). Par exemple, en envoyant une commande plus grande à gauche qu'à droite, notre robot ira vers la droite. Il peut aussi tourner sur lui-même en envoyant une commande opposée de chaque côté.

2.1.2 Capteurs et électronique embarquée

Saturne embarque des capteurs nécessaires pour réaliser ses missions autonomes :

→ Un récepteur GNSS qui nous donne une estimation de la position du robot dans le repère global. Il a une fréquence de 1 Hz et, sans correction RTK, à une précision de l'ordre de 4 mètres.

→ IMU (modèle SBG Ellipse 2A) : centrale inertielle qui nous permet d'avoir une estimation du cap du robot. Cette IMU a été préalablement calibrée magnétiquement par nos soins. Elle nous permet d'avoir des mesures précises à l'ordre du degré avec une fréquence de 25 Hz.

La carte de commande des moteurs nous renvoie aussi la tension des 2 batteries de puissance et l'intensité consommée par chaque moteur.
L'ordinateur de bord est un Intel Nuc avec comme système d'exploitation Ubuntu 18.04. Il est couplé avec un module wifi Ubiquiti longue portée qui nous permet de rester connecté à distance.
Saturne embarque aussi un LiDAR et une caméra, mais ces capteurs non pas été utilisés pour le projet Magmap. On notera qu'aucune mesure de la vitesse n'est faite, notre robot avance donc à une vitesse arbitraire qui varie avec la charge des batteries.

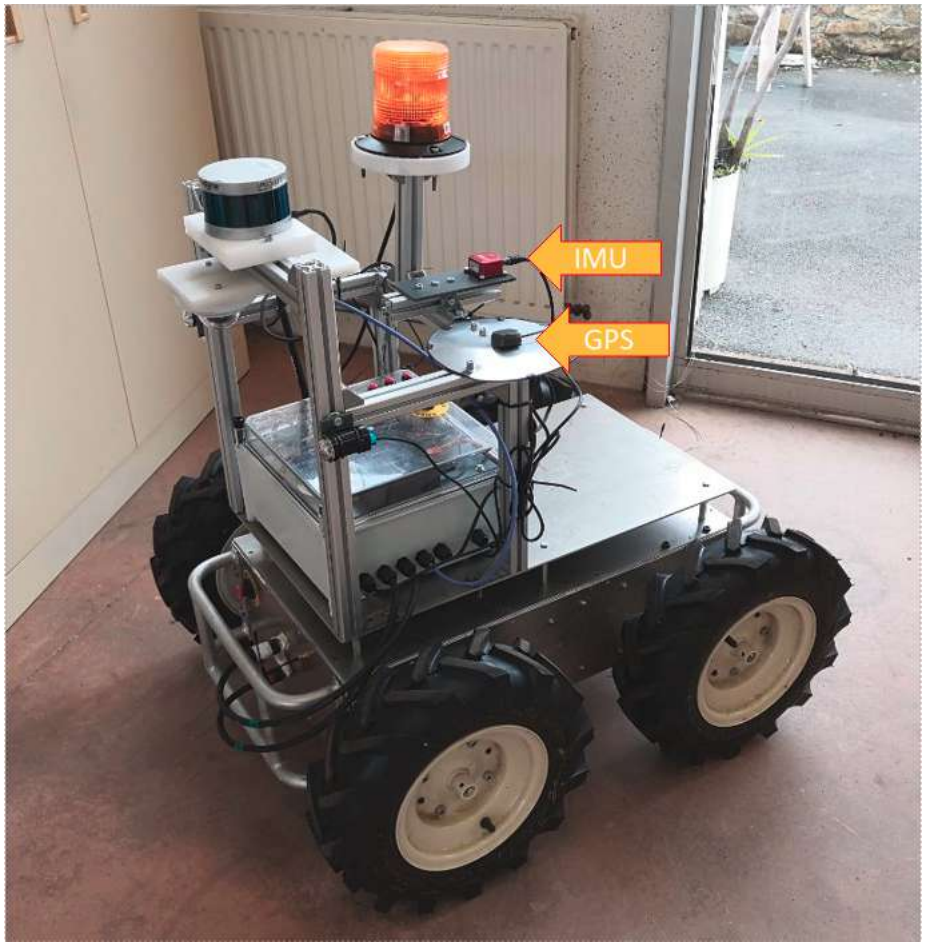

Figure 20 : Matériel embarqué sur Saturne

2.1.3 Architecture logicielle

Pour mettre en place l'architecture logicielle du robot, le *middleware* ROS a été utilisé. Ce *middleware* facilite le développement collaboratif et permet d'intégrer facilement de nouvelles fonctionnalités. Sur le robot Saturne, un travail préalable permettait de réaliser des missions composées d'objectifs élémentaires comme le suivi de ligne, de cercle et de *waypoint*. En se basant sur l'architecture existante, nous avons ajouté un nœud haut niveau qui permet de lancer une mission contenue dans un fichier texte.

Architecture Ros

→ Estimateur : ce nœud calcule une estimation de la position du robot. Il reçoit une estimation de la position du robot par le GNSS et les données de l'IMU. Entre chaque nouvelle mesure GNSS, il intègre la mesure donnée par l'IMU afin d'obtenir une fréquence plus élevée que le GPS seul.

→ Mission : ce nœud gère les objectifs de la mission. C'est ce nœud qui lit le fichier texte contenant les objectifs et qui gère les conditions de fin d'objectif et de fin de mission.

→ Objectif : Ce nœud calcule une commande en cap pour que le robot réalise son objectif courant. Le cap est calculé en utilisant la méthode des champs de potentiels.

→ Contrôleur : Ce nœud prend le cap consigne et le convertit en commande à envoyer aux cartes de contrôle des moteurs.

→ IHM : une IHM a été développée afin de surveiller le déroulement de la mission. On y affiche les objectifs et la position estimée du robot en temps réel afin de vérifier que tout fonctionne correctement.

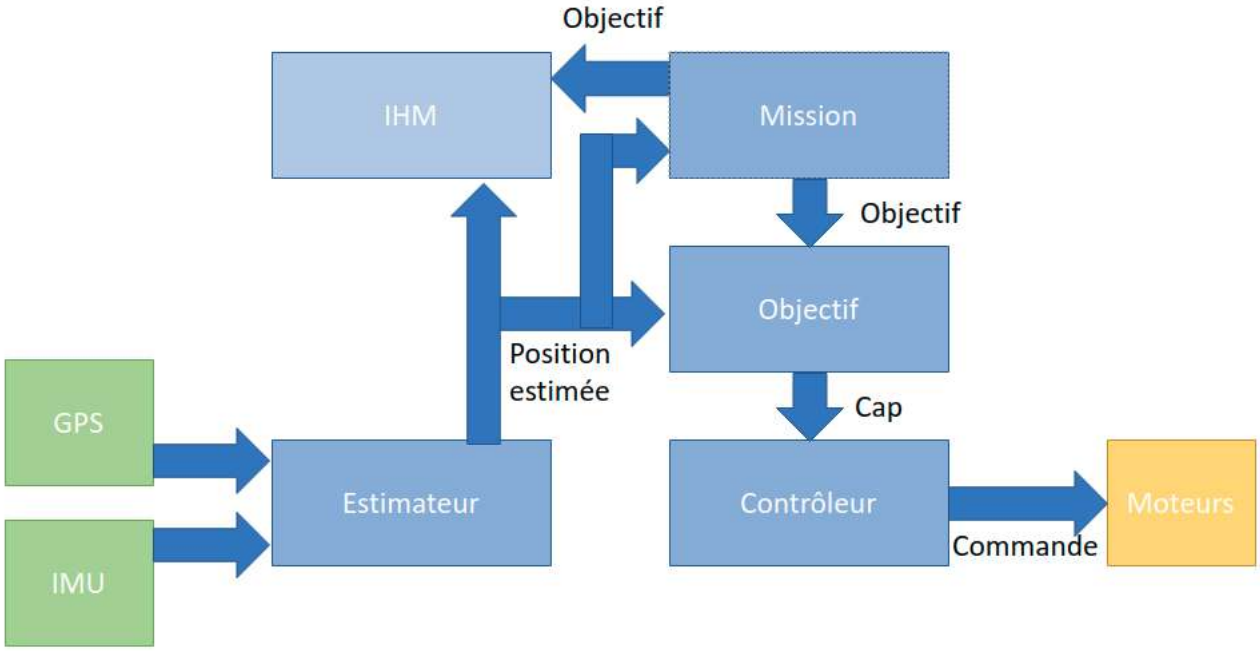

Figure 21 : Graphe des nœuds ROS

2.2 Robot Riptide

2.2.1 Description

Riptide Autonomous Solutions

Riptide Autonomous Solutions était une société américaine basée dans le Massachusetts qui se concentre sur le développement et la production de véhicules maritimes sans équipage (UMV), de véhicules sous-marins autonomes (AUV) et de véhicules de surface autonomes (ASV).
La société a été rachetée par BAE Systems en juin 2019 et ses produits sont présentés comme étant à la pointe techniquement, à un prix nettement inférieur à celui des concurrents tels qu'Edge Tech.
L'électronique, les logiciels libres et les techniques de fabrication récentes devraient permettre d'obtenir de meilleures performances. Leurs produits, à première vue simples et

flexibles, sont censés être personnalisables pour adapter la charge utile ou les paramètres opérationnels. Leur gamme d'AUV est variée et permet d'ajouter un sonar, une antenne DVL ou Iridium selon les besoins.

Le Riptide MKII microUUV

Le premier AUV utilisé est le Riptide MKII microUUV de Riptide Autonomous Solutions. Pour des soucis d'écriture, celui-ci sera appelé "Riptide" dans la suite de ce rapport. Il s'agit d'un AUV low-cost modulable. L'ENSTA Bretagne en possède quatre, sans options, non utilisés jusque-là.
Le Riptide peut être divisé en trois parties distinctes :

- la tête (navigation)
- le *payload* central (alimentation)
- la queue (actionneurs)

FIGURE 3.1 – Options de modules Riptide

Figure 22 : Différents modèles de Riptide

Le Riptide est conçu avec des composants électroniques propriétaires dont la documentation est souvent introuvable. L'ancienne architecture de l'AUV est la suivante :

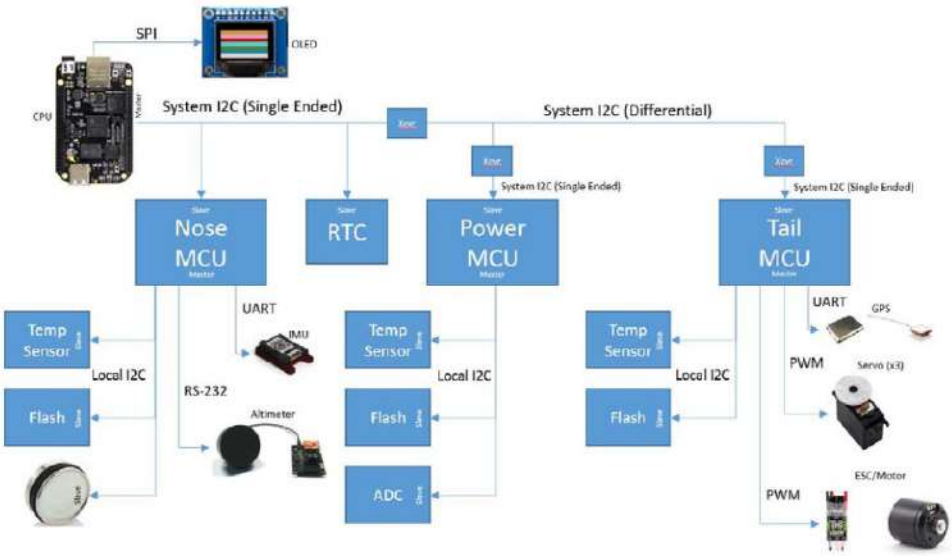

Figure 23 : Ancienne architecture du Riptide

Le protocole de communication utilisé était l'I2C différentiel. Ce dernier n'est pas détaillé dans la documentation du Riptide, ce qui complique grandement l'interfaçage ou l'ajout de nouveaux composants. Afin de palier à ce problème, une reprise totale de l'architecture du robot a été étudiée.

Nouvelle architecture

La solution retenue s'articule autour d'une Raspberry Pi 4B comme ordinateur embarqué avec ROS Melodic comme *middleware*.

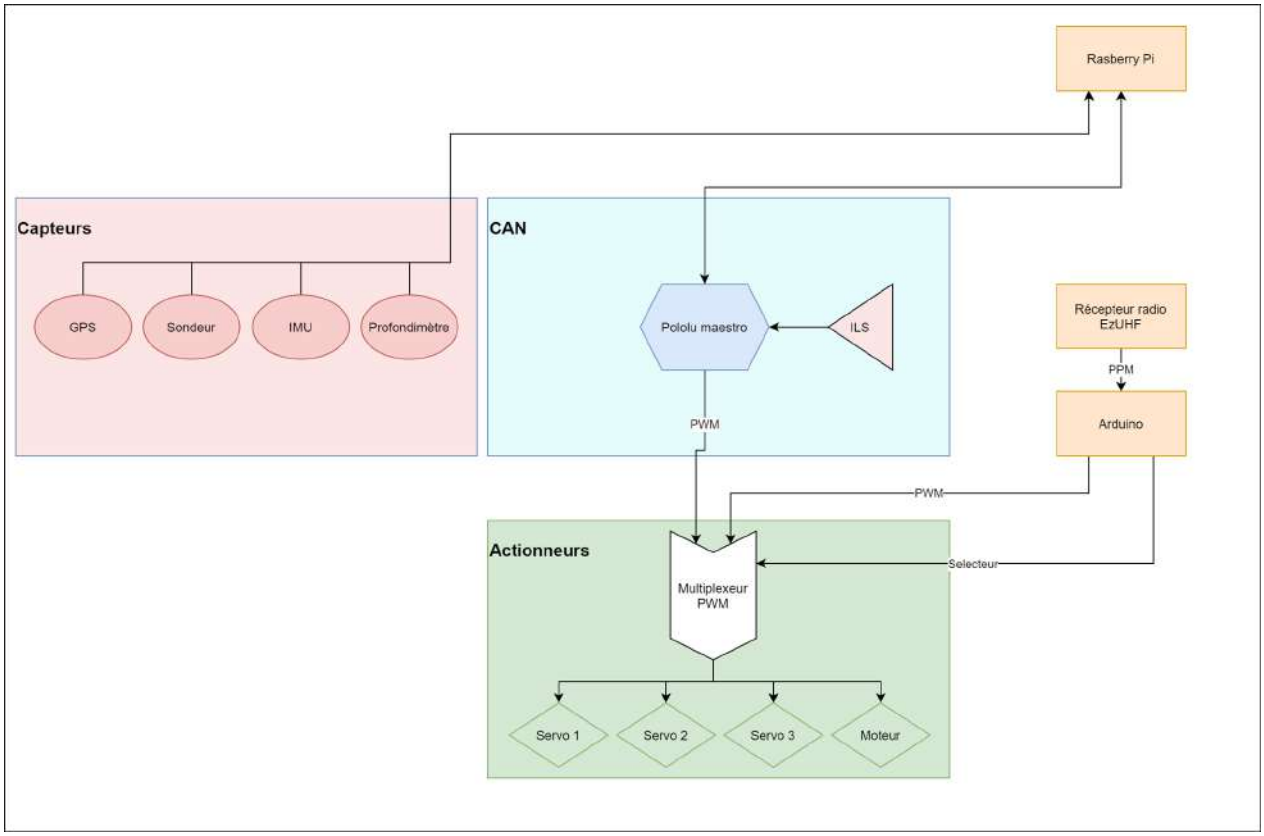

Figure 24 : Nouvelle architecture du Riptide

Ainsi, la tête contient l'IMU, le capteur de pression et de température ainsi qu'un écho-sondeur. La queue abrite le moteur principal ainsi que les trois ailerons de direction, disposés à 120 degrés les uns des autres. L'aileron supérieur contient une antenne GPS et une antenne WiFi pour le contrôle à distance de l'AUV. Le module GPS est quant à lui placé dans le creux de la queue. La charge utile centrale contient la Raspberry Pi 4B, les batteries, un système de démarrage (marche/arrêt) ainsi que les poids nécessaires à l'équilibrage.

Figure 25 : Tête du Riptide

Figure 26 : Queue du Riptide

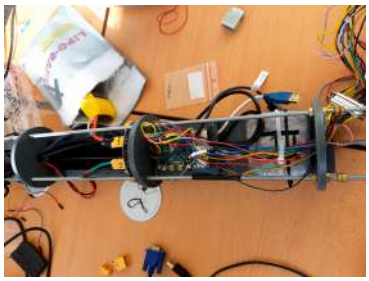

Figure 27 : Corps du Riptide

Le système de démarrage utilise un interrupteur magnétique ILS. Ce dernier est connecté à un micro-contrôleur (PIC) alimenté en 5V et qui décode le signal de l'interrupteur. Dès qu'une séquence précise est reçue (séquence retenue pour les tests : front montant

et maintien pendant 2 secondes), le PIC déclenche le système d'allumage et tous les autres composants sont alimentés. Ce système permet de préserver la batterie du Riptide pendant sa mise à l'eau. En effet, afin de le mettre à l'eau, il faut étanchéifier les différents joints, ce qui est assez long. Il est donc très utile d'avoir un système de démarrage/arrêt du circuit électrique.
Les différents drivers ont été écrits pour ROS Melodic. Ces derniers permettent d'une part de recueillir les données des capteurs mais également de publier les commandes pour les moteurs. La même architecture globale a été utilisée par l'équipe travaillant sur le contrôle et la simulation du Riptide.

2.2.2 Modélisation

Les actionneurs du Riptide sont les suivants : une hélice à l'arrière pour la propulsion, trois servomoteurs contrôlant les ailerons, placés à 120 degrés les uns des autres à l'arrière du Riptide. Pour notre application, nous ne prenons pas en compte la capacité du Riptide à tourner sur lui-même (axe $\overrightarrow{z}$).

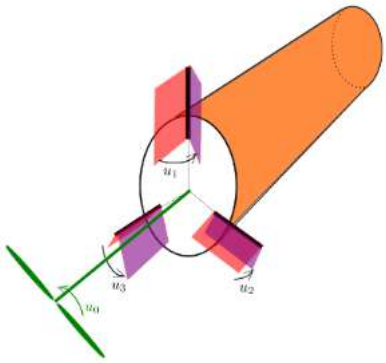

Figure 28 : Contrôle du Riptide

D'après les équations de Fossen, la force de propulsion exercée par l'hélice est proportionnelle au carré de la commande envoyée; le coefficient de proportionnalité dépendant alors notamment du diamètre de l'hélice et de la masse volumique de l'eau. On pose aussi comme hypothèse que le frottement lié à l'eau est proportionnel au carré de la vitesse. On obtient alors :
$(m + m_a).a = \alpha.u_0.|u_0| - \beta.v_x.|v_x| \Leftrightarrow a = p_1.u_0.|u_0| - p_2.v_x.|v_x|$
Avec p_1 et p_2 des constantes dépendant de la surface normale du Riptide à l'axe $\overrightarrow{x}$, des coefficients de frottement de l'eau, et de la masse ajoutée m_a de l'eau. On cherche ensuite à modéliser la force appliquée sur les ailerons (traînée et portance). La force de traînée est négligeable par rapport à celle de portance on obtient donc pour un aileron :
$\overrightarrow{F_{pa}} = \gamma.v.\sin(u).\overrightarrow{ax}$
Avec $\overrightarrow{ax}$ l'axe de l'aileron considéré, et u l'angle entre cet aileron autour de cet axe. Puisque la commande des ailerons ne permet que de faibles variations d'angle, on se place ici dans l'approximation des petits angles : on considère donc $u \equiv \sin(u)$.
En exprimant chacun des ailerons dans le référentiel associé au Riptide, on a alors :

$$\begin{bmatrix}\omega_x\\ \omega_y\\ \omega_z\end{bmatrix} = v.B.\begin{bmatrix}u_1\\ u_2\\ u_3\end{bmatrix} \text{ avec } B = \begin{bmatrix}p_3 & 0 & 0\\ 0 & p_4 & 0\\ 0 & 0 & p_4\end{bmatrix}.\begin{bmatrix}-1 & -1 & -1\\ 0 & \sin(\frac{2\pi}{3}) & -\sin(\frac{2\pi}{3})\\ 0 & \cos(\frac{2\pi}{3}) & -\cos(\frac{2\pi}{3})\end{bmatrix}$$

p_3 et p_4 dépendent ici des paramètres d'inertie du Riptide (masse, rayon, ...). On notera qu'on choisit ici d'approximer l'inertie du Riptide comme correspondant à celle d'un cylindre plein suivant l'axe $\overrightarrow{x}$, ce qui explique la forme diagonale de la matrice et la répétition de p_4. On obtient :

$$\begin{cases}\dot{p} = R(\varphi,\theta,\psi).v_r\\ \dot{v_r} = a_r - \omega_r \wedge v_r\\ \begin{bmatrix}\dot{\varphi}\\ \dot{\theta}\\ \dot{\psi}\end{bmatrix} = \begin{bmatrix}1 & \tan(\theta)\sin(\varphi) & \tan(\theta)\cos(\varphi)\\ 0 & \cos(\varphi) & -\sin(\varphi)\\ 0 & \frac{\sin(\varphi)}{\cos(\theta)} & \frac{\sin(\varphi)}{\cos(\theta)}\end{bmatrix}.\omega_r\end{cases}$$

Où φ, θ, ψ sont les angles d'Euler, ωr est le vecteur de rotation exprimé dans le cadre du robot, a_r est le vecteur d'accélération dans le cadre du robot, et p est la position du robot dans le cadre du monde. On utilise ensuite les équations de changement de référentiel [3] entre le référentiel du Riptide et le référentiel observateur, le modèle global associé au Riptide est le suivant :

$$\begin{cases}\dot{p} = R(\varphi,\theta,\psi).(v_r,0,0)^T\\ \dot{v_x} = p_1.u_0.|u_0| - p_2.v_x.|v_x|\\ \begin{bmatrix}\dot{\varphi}\\ \dot{\theta}\\ \dot{\psi}\end{bmatrix} = \begin{bmatrix}1 & \tan(\theta)\sin(\varphi) & \tan(\theta)\cos(\varphi)\\ 0 & \cos(\varphi) & -\sin(\varphi)\\ 0 & \frac{\sin(\varphi)}{\cos(\theta)} & \frac{\sin(\varphi)}{\cos(\theta)}\end{bmatrix}.v_x.B(p_3,p_4).\begin{bmatrix}u_1\\ u_2\\ u_3\end{bmatrix}\end{cases}$$

2.2.3 Algorithmes

Les équations du Riptide expliquées ci-dessus ont déjà été implémentées par la promotion CI2020 dans le cadre du projet GoZone. Cependant le simulateur déjà existant ne pouvait pas être utilisé tel quel. Trois opérations ont été nécessaires.

Passage en Python 3

Tout le code existant était en Python 2. Or depuis le premier janvier 2020, Python 2 est officiellement devenu obsolète. Ainsi, passer le code en Python 3 semble raisonnable pour permettre aux futurs étudiants de reprendre le code avec plus de facilité. Cela a posé plusieurs problèmes au niveau des affichages avec matplotlib mais surtout au niveau de l'intégration ROS car Python 3 n'est devenu la cible de ROS1 que depuis ROS Noetic. Certaines incompatibilités peuvent apparaître avec les versions précédentes (ROS Melodic dans notre cas).

Planificateur de missions

A l'image du robot Saturne, pour correctement effectuer la mission, le robot doit avoir un planificateur de mission. Nous avons donc dû adapter le contrôleur afin de per-

mettre au Riptide de suivre et valider les *waypoints*. Pour cela, nous avons utilisé un plugin sur le logiciel QGIS qui est un logiciel gratuit et open-source. Grâce à celui-là nous pouvons donner des *waypoints* à suivre dans un ordre donné et le tout est enregistré dans un fichier JSON qui est ensuite lu afin d'envoyer la liste de *waypoints* à suivre.
Pour valider le *waypoints* deux options ont été envisagées, la première était une validation par sphère. Si le robot passe dans une sphère de rayon r donné alors le *waypoint* est validé. Cependant cette méthode, qui en simulation peut fonctionner, présente des problèmes dans la réalité. En effet si un courant trop puissant empêche le robot de réellement frôler la sphère, le Riptide pourrait dépenser beaucoup de son énergie pour le valider. Nous avons donc utilisé une validation de *waypoints* par demi-plan. Si l'AUV passe le plan orthogonal formé par le *waypoint* cible alors on considère celui-ci comme validé.

Comme on peut le voir ci-dessous :

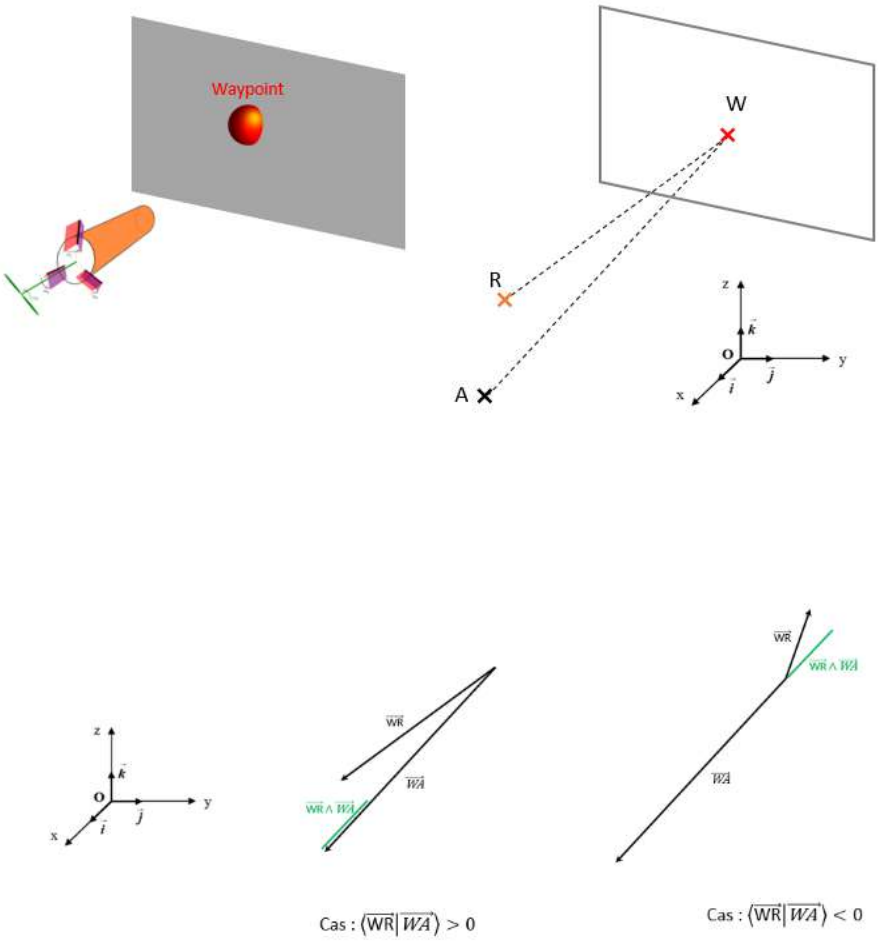

Figure 29 : Déplacement de l'AUV vers un *waypoint*

Avec R le point représentant la position du Riptide, W le point du *waypoint* et A le point de l'ancien *waypoint*.
Si l'on calcule le produit scalaire entre $\overrightarrow{WR}$ et $\overrightarrow{WA}$ on remarque que celui va changer de signe lors du franchissement du demi-plan. Ainsi en observant son signe, on peut

déterminer si la condition de validation a été atteinte.
Afin d'aller vers le bon *waypoint* le fonctionnement du contrôleur est relativement simple. On calcule le cap à suivre et on réduit l'erreur entre le cap réel et le le cap théorique (de même pour le tangage).

Passage en ROS

Cela a été la plus grosse partie du travail sur cet algorithme. En effet, l'année précédente le Riptide avait déjà été retravaillé du point de vue logiciel et un passage sous ROS avait été entamé (auparavant le Riptide fonctionnait uniquement avec MOOS). Des *drivers* ROS pour les différents capteurs et actionneurs ont été codés. Nous avons donc pu récupérer les noms des *topics* afin d'interfacer notre contrôleur avec le Riptide. Le

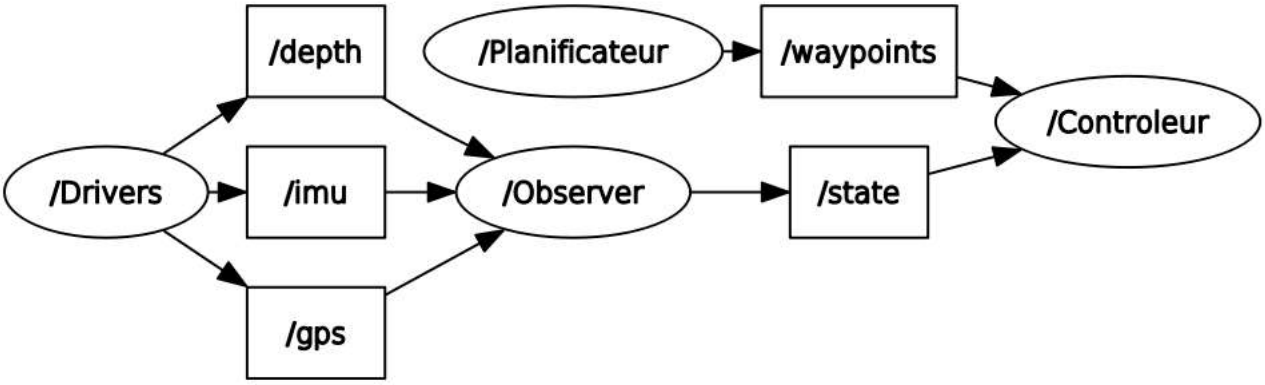

Figure 30 : *Node graph*

fonctionnement est plutôt simple. Tout d'abord on récupère les données des différents capteurs (que cela soit via Gazebo ou via un code Python qui simule des données capteurs) sur les *topics* correspondants : /depth, /imu ou /gps.

- ☞ /depth contient l'information de profondeur du Riptide sous forme d'un message de type std_msgs/float32.
- ☞ /imu contient les données de la centrale inertielle sous la forme d'un message de type sensor_msgs/Imu
- ☞ /gps contient les données GNSS sous la forme d'un message de type sensor_msgs/NavSatFix

Ensuite ces données sont envoyées à l'observateur qui estime l'état sous la forme d'un message de type geometry_msgs/Pose et qui l'envoie sur le *topic* /state. Il ne reste plus au contrôleur qu'à écouter le *topic* /state et le *topic waypoints* d'où vient la mission à effectuer et ainsi donner le bon cap à suivre au robot.
Les commandes u_1, u_2, u_3 et u_4 calculées par le contrôleur sont ensuite soit envoyées sur les différents *topics* de la carte contrôlant les actionneurs soit au Riptide simulé sur Gazebo. Il est aussi possible de visualiser sur Python l'évolution en 3D de l'AUV.

Simulation

3.1 Robot Saturne

3.1.1 Introduction

Pour pouvoir tester les algorithmes de déplacement, une simulation du robot sous Gazebo a été développée. Elle permet aussi de vérifier que la trajectoire que l'on veut faire ne rencontre pas d'obstacle et reste bien dans les limites du terrain.

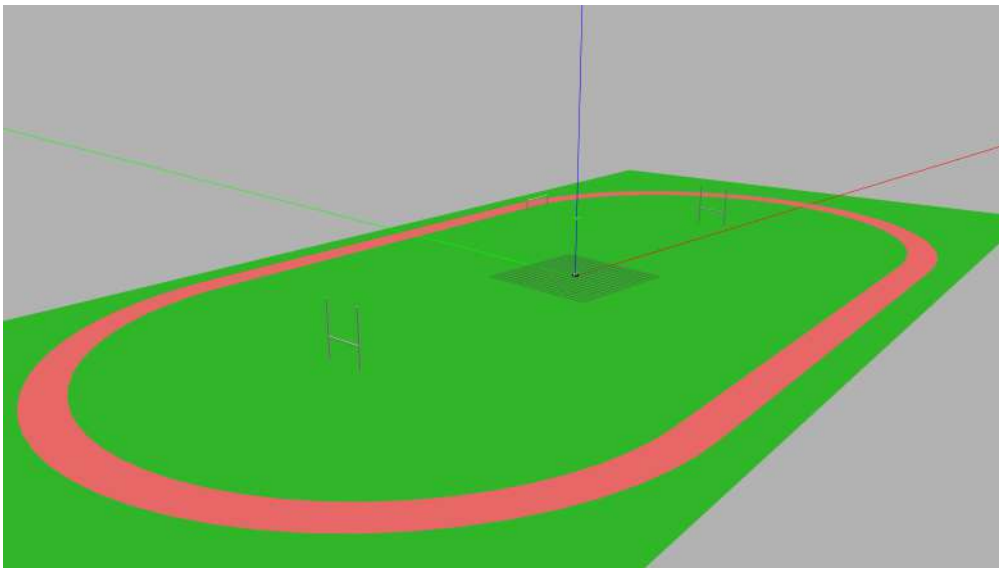

Figure 31 : L'environnement simulé

3.1.2 Modélisation du robot

Notre robot est modélisé par un modèle char : il tourne en utilisant le différentiel de commande sur les moteurs droits et gauches. Le robot est modélisé avec des formes géométriques simples pour calculer les collisions et les inerties : les roues sont des cylindres et le châssis du robot est une boite. La forme complexe de la partie supérieure est purement visuelle. Le robot simulé est équipé d'une IMU et d'un récepteur GNSS, comme le robot réel. Nous avons donné des covariances semblables aux valeurs des capteurs réels et ils communiquent les mêmes types de données sur les mêmes *topics* ROS que le robot réel. Ils sont aussi placés aux mêmes endroits.

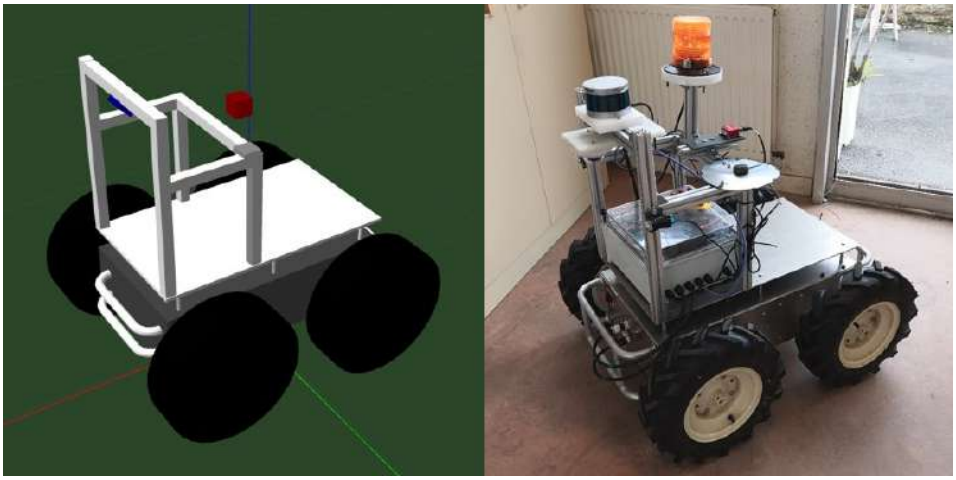

Figure 32 : Le robot simulé et le robot réel

3.1.3 Différences entre robot réel et robot simulé

L'utilisation de ROS nous permet d'utiliser globalement le même code que le robot réel. Les drivers des capteurs réels ont été supprimés car Gazebo s'occupe d'envoyer les données des capteurs simulés sur les *topics*. Un nœud ROS a été créé pour faire une interface entre la commande normalement envoyée aux moteurs (commande comprise entre -1000 et 1000) et la commande envoyée pour les moteurs simulés (commande en m/s).

3.1.4 Modélisation de l'environnement

Le stade de l'école a été reproduit (Fig. 13) afin de tester au mieux les missions. Le terrain est à l'échelle réelle et les obstacles sont placés au plus proche de leur emplacement réel. Les obstacles ne sont pas uniquement décoratifs, ils sont solides et le robot peut entrer en collision avec ceux-ci.

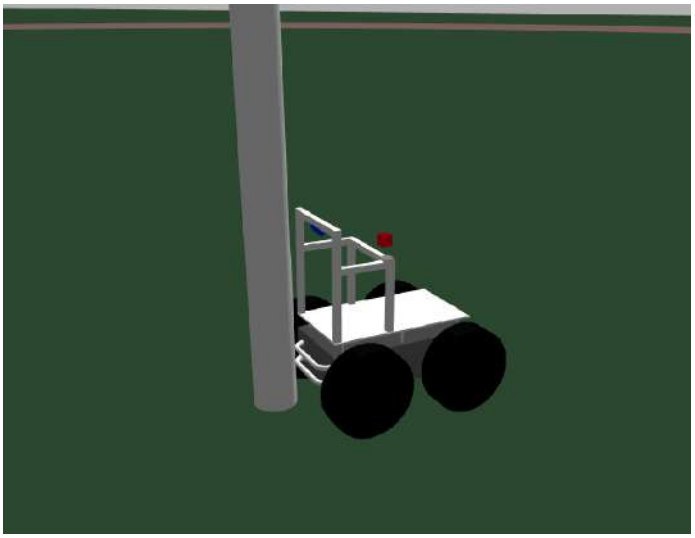

Figure 33 : Le robot Saturne simulé contre un obstacle, la trajectoire n'est pas faisable

3.2 Robot Riptide

3.2.1 Modélisation géométrique du Riptide

Le Riptide est un AUV (*Autonomous Underwater Vehicle*) de type torpille, sa forme est donc assez simple. Il comprend trois ailerons et un propulseur situé à l'arrière. Nous avons dans un premier temps mesuré ses dimensions réelles puis modélisé l'AUV en CAO à l'aide du logiciel Autodesk Inventor. La modélisation s'est faite en trois parties afin de pouvoir facilement dissocier le corps du Riptide de ses actionneurs. La première partie modélisée fut le corps, une simple forme cylindrique d'environ un mètre de long sur 15 cm de rayon. Le corps principal du Riptide a ensuite été exporté sous Blender pour l'alignement des axes et la production de deux fichiers nécessaires à la simulation, un fichier avec l'extension .dae qui permettra la visualisation et un fichier avec l'extension .stl qui permettra la gestion des collisions.
Les actionneurs ont également été modélisés sur Autodesk : les ailerons et le propulseur à deux pales. Ces fichiers ont simplement été exportés avec l'extension .dae après un passage sur le logiciel Blender pour permettre la visualisation sur Gazebo.

Figure 34 : Corps du Riptide sur Autodesk Inventor / Photo du Riptide

Ces fichiers 3D seront ensuite lus par Gazebo pour être affichés dans le simulateur ou pris en compte dans les calculs de collision. Ils constitueront aussi une importante ressource pour connaître les différents paramètres du Riptide, notamment la surface des ailerons qui sera prise en compte dans les équations de mouvements.

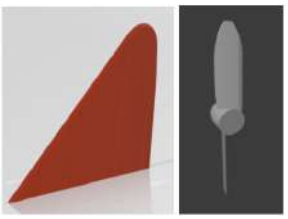

Figure 35 : Aileron et propulseur du Riptide modélisé sur Autodesk Inventor

3.2.2 Prise en main de Gazebo et du plug-in UUV Simulator

Pour ce projet, que ce soit pour Saturne ou pour le Riptide, le choix a été fait, en simulation, de fonctionner avec le logiciel Gazebo. Ce choix a été fait pour son réalisme, sa

simplicité d'utilisation et surtout son interfaçage facile avec ROS, *middleware* utilisé à la fois sur le Riptide et sur le Saturne. Ce simulateur permet donc de tester les algorithmes utilisés en réel sur les robots avec le minimum de retouches sur le code.
La simulation de robots sous-marins est assez complexe, nous avons donc recherché des plugins permettant de simuler l'environnement sous-marin et les AUV. Le plugin UUV simulator nous est apparu le plus pertinent pour plusieurs raisons. Ce plugin est assez utilisé et donc assez documenté, ce qui nous a permis de nous familiariser assez vite avec ses nombreuses fonctionnalités. De plus, de nombreux UUV (*Unmanned Underwater Vehicle*) sont déjà implémentés dans ce simulateur (LAUV, ECA A9, Sirehna Iver2, RexRov), dont certains ont la même forme que le Riptide, il nous sera donc plus facile de nous en inspirer pour la modélisation de notre robot.

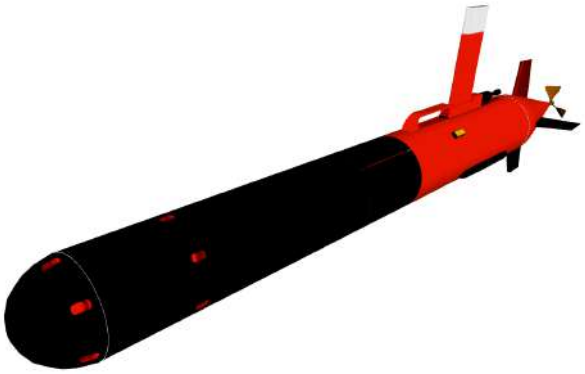

Figure 36 : Modèle 3D du LAUV implémenté dans UUV simulator

Une fois le choix du simulateur effectué, il a fallu se familiariser avec Gazebo et ce plugin. La réalisation de plusieurs tutoriels nous a permis de découvrir l'architecture du plugin. Il implémente la dynamique des AUV avec l'équation de Fossen pour laquelle une version plus ou moins simplifiée peut être choisie. Le simulateur est composé de plusieurs fichiers avec l'extension .world qui permettent de lancer des mondes sur Gazebo. Une fois le monde lancé avec ses paramètres, il convient d'y faire apparaître un ou plusieurs AUV.
La création des AUV comprend assez peu d'étapes en théorie: il faut modéliser sa forme (à la main ou via un logiciel de CAO) pour définir le visuel et les géométries de collisions, définir les inerties et l'hydrodynamisme du corps, puis ajouter et paramétrer les capteurs et les actionneurs. Gazebo peut lire les fichiers avec l'extension .urdf et .xacro, qui permettent de spécifier les paramètres d'un robot (taille, masse, inerties). C'est dans ces fichiers que l'on va définir les liens vers les modèles 3D pour la visualisation de notre robot. On va aussi pouvoir y spécifier les capteurs et les actionneurs, leurs types et leurs paramètres, grâce aux plugins de UUV simulator qui supportent l'intégration de ceux-ci.

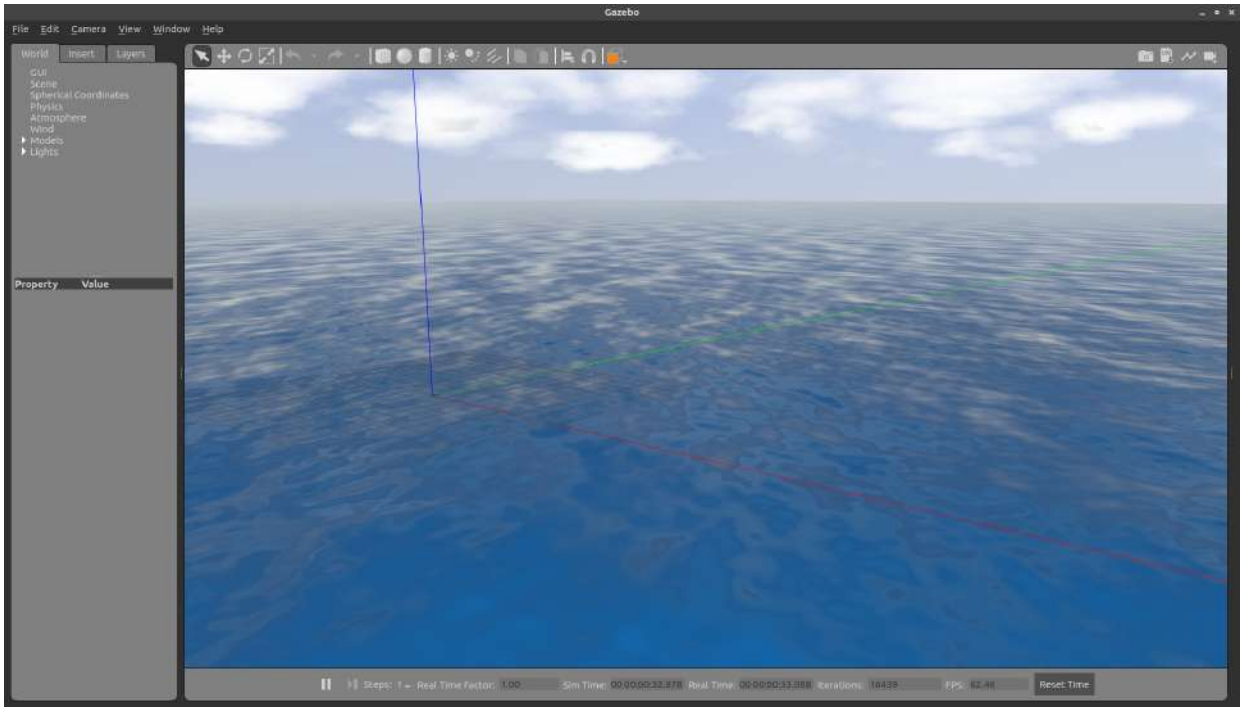

Figure 37 : Exemple du monde « ocean_waves.world » sur UUV simulator

3.2.3 Modélisation physique du Riptide sur UUV Simulator

La définition d'un AUV sous UUV Simulator passe donc par la combinaison de modèles 3D et de fichiers de description .urdf et .xacro. Le projet UUV Simulator étant un projet open-source, nous pouvons utiliser des modèles d'AUV existants pour s'en inspirer. Nous avons fait le choix d'utiliser le modèle du LAUV (Light Autonomous Underwater Vehicles) développé par le "Laboratório de Sistemas e Tecnologia Subaquática (LSTS)" de l'Université de Porto et OceanScan-MST. [MSV+16] Il s'agit d'un modèle d'AUV de type torpille à 4 ailerons disposés en "+" autour du corps principal ainsi que d'un 5ème aileron central de dimension plus importante, et avec un propulseur à l'arrière de 3 pâles. À partir de ce modèle existant, il faut adapter le visuel, les collisions, les dimensions et le positionnement des actionneurs. Les modèles 3D du Riptide et de ses actionneurs effectués en CAO sont venus remplacer les modèles du LAUV. Les fichiers .urdf et .xacro ont été modifiés pour coller à la description du Riptide. Ci-dessous un aperçu de l'arborescence des packages ROS lauv_description et riptide_description, suivi d'un descriptif détaillé.

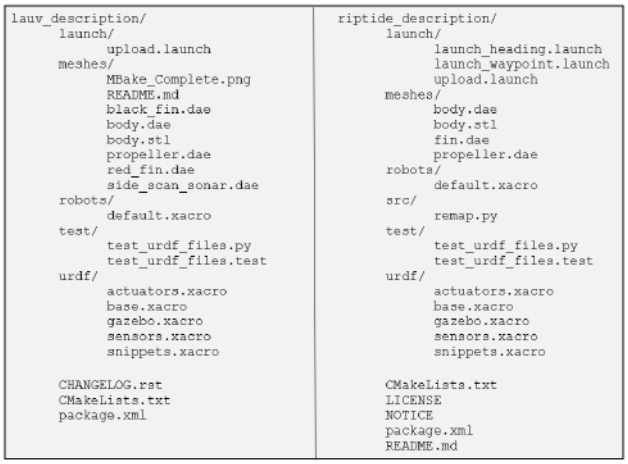

Figure 38 : Arborescence des packages lauv_description et riptide_description

Dans le package riptide_description, on peut trouver une licence, un README et des fichiers de configurations du package. Dans le dossier « launch » :

→ upload.launch : il permet de faire apparaître l'AUV sur Gazebo avec des paramètres possibles de position et d'attitude initiale. Il nécessite un remplacement des liens vers lauv_description par riptide_description.

→ Ajout des fichiers launch_heading.launch et launch_waypoint.launch pour démarrer Gazebo, faire apparaître le Riptide et lancer les contrôleurs développés pour le Riptide pour effectuer un suivi de cap et de *waypoint*.

Dans le dossier « meshes », fichiers 3D du LAUV ont été remplacés par ceux modélisés pour le Riptide (en .dae pour le visuel, et en .stl pour les géométries de collisions).
Dans le dossier « robots » : le fichier default.xacro permet de créer le robot et de définir l'environnement dans lequel il évolue. Il nécessite un remplacement des liens vers lauv_description par riptide_description.
Dans le dossier « src » : Le fichier remap.py est un nœud ROS en Python qui permet de rediriger les messages ROS entre le contrôleur et Gazebo. En effet, le contrôleur a été développé pour le robot réel et les types de message, les noms des *topics* peuvent différer de ceux utilisés dans Gazebo qui utilisent ceux définis par défaut dans le plugin UUV Simulator.
Dans le dossier « test » : Les fichiers test_urdf_files.py et test_urdf_files.test sont exécutés automatiquement, ils permettent de tester la syntaxe des fichiers en .urdf et nécessitent d'être modifiés pour un remplacement des liens vers lauv_description par riptide_description. Dans le dossier « urdf » :

→ actuators.xacro: définition du propulseur et des ailerons (visuels et positions)

→ base.xacro : définition de la masse, du centre de gravité, de la masse volumique du fluide, de la géométrie visuelle et de collision du corps du Riptide et des inerties (simplifiées pour le cas d'un cylindre plein). La masse a été définie de manière empirique à volume fixe pour rendre l'AUV stable à 1 mètre de profondeur

→ gazebo.xacro : définition des dimensions du Riptide (longueur, diamètre, rayon et hauteur mesurés sur le robot ainsi que le volume) et définition des équations de mouvements pour préciser l'hydrodynamisme du véhicule (inchangé par rapport au LAUV). Pour le calcul du volume, il convient d'utiliser celui d'un cylindre de longueur et rayon connus, auquel on applique un coefficient de 90% pour prendre en compte la réduction de section au niveau de la tête et la queue de la torpille

→ sensors.xacro : définition des capteurs qui équipent l'AUV

→ snippets.xacro : définition des liaisons et de la dynamique des actionneurs (inchangé par rapport au LAUV excepté pour la surface des ailerons qui a été calculée via le modèle sur CAO).

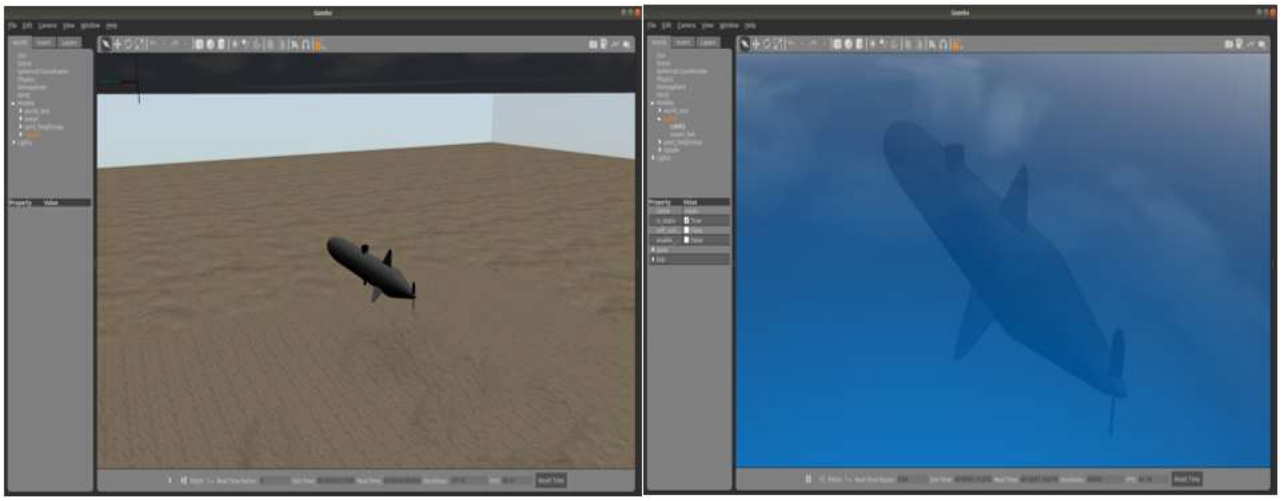

Figure 39 : Riptide dans Gazebo avec UUV Simulator

3.2.4 Simulation des capteurs et actionneurs

Le Riptide est composé de 3 capteurs pour se positionner et de 4 actionneurs pour naviguer. Ceux-ci comprennent un GPS, une IMU et un capteur de pression concernant les capteurs. On compte également un écho-sondeur qui n'a pas été simulé. Pour ses actionneurs, le Riptide utilise 1 propulseur deux pâles à l'arrière et 3 ailerons pour contrôler son cap et son assiette.
La simulation des capteurs est implémentée dans le plugin UUV Simulator et il suffit d'y faire référence dans le fichier sensors.xacro. On y détaille les différents types de capteurs ainsi que le corps auquel est relié le capteur, sa position et son orientation. Lorsque l'AUV sera lancé dans Gazebo, il sera possible de lister les *topics* utilisés pour identifier ceux des capteurs qui nous intéressent. Typiquement, RQT peut être utilisé pour identifier les *topics* et les informations inhérentes. Parmi les exemples d'AUV entièrement modélisés pour Gazebo & UUV Simulator, de nombreux exemples d'utilisation des capteurs sont présents et permettent aisément de prendre en main le sujet.
Le principe est relativement similaire pour les actionneurs définis dans le fichier actuators.xacro. Il contient les informations relatives aux types d'actionneurs, au corps auquel

sont reliés ces actionneurs et les positions relatives des actionneurs par rapport à ce corps (ainsi que les orientations). Le comportement dynamique de ces actionneurs est quant à lui décrit dans le fichier snippets.xacro (amplitudes des joints, traînée des ailerons, surfaces etc. . .). Ces paramètres ont été laissés inchangés par rapport au LAUV pris comme base, excepté pour la surface des ailerons qui intervient dans le mouvement du Riptide et qui a été mesurée via la CAO. Les *topics* utilisés pour envoyer des commandes aux actionneurs sont également visibles lorsque la simulation est lancée avec l'AUV.

3.2.5 Interface entre la simulation et la partie contrôle

Le Riptide ayant été modélisé sur simulation, il faut intégrer un éventuel contrôleur qui viendra piloter celui-ci. Le contrôleur a été implémenté avec des conventions de nommage adaptées au robot réel (nom des *topics* et types de messages associés aux drivers des capteurs et actionneurs). Ces mêmes capteurs et actionneurs sont intégrés sur la simulation via le plugin UUV Simulator qui fonctionne avec des conventions de nommage différentes. Il convient donc d'implémenter une interface entre la simulation et le contrôleur. L'architecture ROS globale intégrant l'interface est illustrée plus bas. Cette interface est un nœud ROS qui :

→ Reçoit les données du GPS depuis Gazebo dans un message de type geometry_msgs/NavSatFix sur le *topic* /riptide/gps, puis les republie dans un message du même type sur le *topic* /gps

→ Reçoit les données de l'IMU depuis Gazebo dans un message de type geometry_msgs/Imu sur le *topic* /riptide/imu, puis les republie dans un message du même type sur le *topic* /imu

→ Reçoit les données du capteur de pression en kPa depuis Gazebo dans un message de type geometry_msgs/FluidPressure sur le *topic* /riptide/pressure, puis republie la profondeur correspondante sur un message de type std_msgs/Float32 sur le *topic* /depth

→ Reçoit la commande du propulseur depuis le contrôleur dans un message de type std_msgs/Float32 sur le *topic* /pololu/esc puis la republie dans un message de type uuv_gazebo_ros_plugins_msg/FloatStamped sur le *topic* /riptide/thrusters/0/input

→ Reçoit la commande du premier aileron depuis le contrôleur dans un message de type std_msgs/Float32 sur le *topic* /pololu/servo1 puis la republie dans un message de type uuv_gazebo_ros_plugins_msg/FloatStamped sur le *topic* /riptide/fins/0/input

→ Reçoit la commande du second aileron depuis le contrôleur dans un message de type std_msgs/Float32 sur le *topic* /pololu/servo2 puis la republie dans un message de type uuv_gazebo_ros_plugins_msg/FloatStamped sur le *topic* /riptide/fins/1/input

→ Reçoit la commande du troisième aileron depuis le contrôleur dans un message de type std_msgs/Float32 sur le *topic* /pololu/servo3 puis la republie dans un

message de type uuv_gazebo_ros_plugins_msg/FloatStamped sur le *topic* /riptide/fins/2/input

Génération de trajectoires

4.1 Problématique

Le but de cette partie est de concevoir un balayage d'une zone quelconque utilisé par un des deux systèmes robot + remorque du projet Magmap, à savoir l'UGV Saturne ou l'AUV Riptide tractant le magnétomètre par un câble. On cherche ici à explorer au maximum la zone d'étude, tout en prévenant que le robot ou la charge utile tractée ne sorte de la zone d'étude ou encore percute un obstacle. De plus, si la luge contenant le magnétomètre peut embarquer un récepteur GNSS, cette situation ne se présentera pas avec le Riptide, rendant l'étude de la position de la charge utile un sujet au cœur de notre projet. Pour faciliter leur étude, le câble tractant cette dernière doit toujours être tendue, ce qui rajoute une contrainte sur la trajectoire générée. Une détente éventuelle de la corde est la plus probable lors des phases de giration, nous chercherons donc dans la suite de cette partie à étudier une stratégie de giration gardant la corde tendue, mais aussi une trajectoire globale réduisant au maximum le nombre de virages. Enfin, pour les raisons énergétiques, nous chercherons à minimiser les distances de ralliement entre deux zones d'études. La stratégie développée ici appliquera les solutions explorées par Xin Yu dans son étude. [Yu15]

4.2 Boustrophédon sur un polygone convexe

Toute surface d'étude peut être modélisée par un polygone représentant l'enveloppe extérieure de la zone, ainsi qu'un certain nombre de polygones représentant l'enveloppe intérieure – c'est-à-dire les obstacles ou les zones à éviter. Or le polygone quelconque obtenu est souvent complexe, et trouver une trajectoire incluant des boustrophédons réduisant le nombre de virages ne peut être faite simplement. L'approche que nous allons utiliser consiste à ramener le problème à l'étude de plusieurs cas simples, plutôt qu'un seul cas complexe. Nous allons donc étudier ici la génération d'un polygone convexe, puis nous verrons dans les parties suivantes comment transformer un problème quelconque vers un problème ne contenant que des polygones convexes.
En effet, dans le cas d'un polygone convexe, nous pouvons générer un boustrophédon dans n'importe quelle direction, sans pour autant entraîner une discontinuité dans la trajectoire. On peut alors chercher quelle direction minimise le nombre de virages, puis générer des lignes parallèles formant le boustrophédon.

4.2.1 Trouver la direction optimale de parcours

Les polygones convexes possèdent une autre propriété intéressante : ils possèdent tous une altitude minimum facilement calculable, comme expliqué dans la partie 3.3 de l'étude de Xin Yu. En plaçant les lignes parallèles de boustrophédon perpendiculairement à la direction correspondant à cette altitude minimum, on obtient alors la trajectoire minimisant le nombre de virages, qui est alors :

$$N_{virages} = \lceil \frac{W}{d} \rceil \tag{4.1}$$

où W est l'altitude minimum et d la largeur de fauchée du boustrophédon.

Comme démontré dans l'article de Wesley H.Huang [Hua01] la direction minimisant l'altitude du polygone convexe est forcément une direction portée par deux sommets du polygone. On peut donc, en complexité linéaire par rapport au nombre de sommets, la déterminer. On utilise pour cela l'algorithme des « rotating calipers », qui fonctionne comme suit :

Algorithm 1 Width of a Convex Polygon

Input: Vertex list of the given convex polygon in counterclockwise order
Output: Width of the polygon and direction of the corresponding parallel lines of support

Delete middle vertices of any collinear sequence of three vertices
$V_a \leftarrow$Vertex with minimum y-coordinate
$V_b \leftarrow$Vertex with maximum y-coordinate
$RotatedAngle \leftarrow 0$
$Angle \leftarrow 0$
$Width \leftarrow \infty$
$Caliper_a \leftarrow$Unit vector along positive x-axis
$Caliper_b \leftarrow$Unit vector along negative x-axis
while $RotatedAngle < \pi$ **do**
    $E_a \leftarrow$Edge from V_a to its next adjacent vertex
    $E_b \leftarrow$Edge from V_b to its next adjacent vertex
    $A_a \leftarrow$Angle between $Caliper_a$ and E_a
    $A_b \leftarrow$Angle between $Caliper_b$ and E_b
    $Altitude \leftarrow 0$
    if $A_a < A_b$ **then**
        Rotate $Caliper_a$ by A_a
        Rotate $Caliper_b$ by A_a
        $V_a \leftarrow$The next adjacent vertex of V_a
        $Altitude \leftarrow$Distance from $vertex_b$ to $Caliper_a$
        $RotatedAngle \leftarrow RotatedAngle + A_a$
    else
        Rotate $Caliper_a$ by A_b
        Rotate $Caliper_b$ by A_b
        $V_b \leftarrow$The next adjacent vertex of $vertex_b$
        $Altitude \leftarrow$Distance from $vertex_a$ to $caliper_b$
        $RotatedAngle \leftarrow RotatedAngle + A_b$
    end if
    if $Altitude < Width$ **then**
        $Width \leftarrow Altitude$
        $Angle \leftarrow RotatedAngle$
    end if
end while
return $Width$ and $Angle$

Figure 40 : Pseudo code du calcul de la direction optimale

Cet algorithme va imiter un pied à coulisse (d'où son nom) tournant autour du polygone et mesurant la distance entres chacun des sommets deux par deux, en gardant une trace de l'altitude minimale rencontrée jusque-là, ainsi de la direction qui lui est associée. Il est important de noter que l'angle envoyé par l'algorithme est directement celui correspondant à la direction optimale du boustrophédon, et non la direction portée par les deux sommets à distance minimale l'un de l'autre.

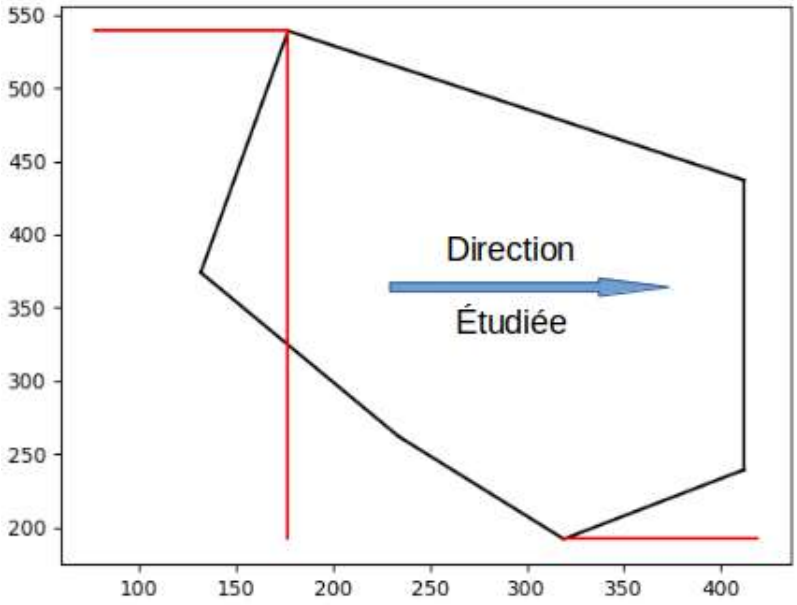

Figure 41 : Schéma de fonctionnement de l'algorithme nom image

4.2.2 Générer les lignes parallèles du boustrophédon

Une fois la direction optimale α du boustrophédon trouvée, il suffit de générer des droites parallèles suivant cette dernière pour générer tous les *waypoints* du boustrophédon.

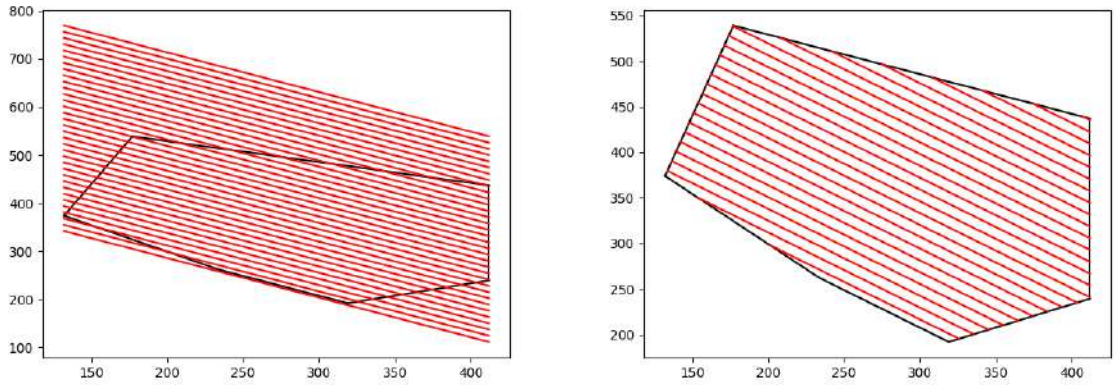

Figure 42 : Génération des lignes de boustrophédon

Pour générer ces droites, on procède comme suit :

→ On calcule les coordonnées x_{min}, x_{max}, y_{min}, y_{max} du carré englobant le polynôme

→ On calcule les coefficients de la première droite :

$$y = ax + b \text{ où } a = tan(\alpha) \text{ et } b = y_{max} + |a * x_{max}| \tag{4.2}$$

→ On calcule le pas à affecter à b pour que les droites soient espacées de la largeur de fauchée d :

$$\delta b = \frac{d}{sin(\frac{\pi}{2} - a)} \tag{4.3}$$

→ On itère jusqu'à avoir rempli le polygone, en sauvegardant à chaque étape l'intersection de la droite avec le polygone. Lorsque l'angle est trop proche de la verticale, on change l'équation de la droite par :

$$x = x_{min} + i_{iteration} * d \tag{4.4}$$

4.3 Décomposition d'un polygone quelconque en sous-parties convexes

Dans cette partie, nous envisageons une nouvelle fois le cas général. Seulement cette fois, nous cherchons non plus à résoudre le problème directement, mais à retrouver un cas plus simple. On souhaite donc ici trouver une méthode de décomposition d'un polygone quelconque en un ensemble de polygones convexes. Il existe de nombreuses solutions à ce problème, mais en prenant en compte notre objectif final, à savoir une trajectoire composée de boustrophédons, une décomposition trapézoïdale s'avère particulièrement intéressante. Cette partie ne sera qu'une traduction des explications données par Xin Yu [Yu15], permettant une analyse concise du fonctionnement de l'algorithme.

4.3.1 Principe général

La méthode de décomposition convexe proposée par Xin Yu [Yu15] est une amélioration de la décomposition trapézoïdale, conçue pour réduire les cellules inutiles. La décomposition trapézoïdale comprend des cellules qui ont la forme de trapèzes ou de triangles (trapèzes dégénérés). Cette décomposition adopte une méthode de type « sweep line » et traite chaque sommet comme un événement. Pour former la décomposition, une ligne verticale est balayée de gauche à droite à travers le champ de polygones. Lorsqu'un événement est rencontré, il étend les rayons vers le haut et vers le bas à travers l'espace libre du champ de polygones jusqu'à ce qu'un bord situé immédiatement au-dessus et au-dessous de l'événement soit atteint. Des cellules trapézoïdales sont formées, complétées ou fermées lors de l'événement en fonction du type d'événement. Une fois que la ligne de balayage a terminé l'événement le plus à droite, la décomposition est terminée.
Cependant, l'inconvénient de la décomposition trapézoïdale est qu'elle a tendance à produire des cellules convexes redondantes, c'est-à-dire des cellules adjacentes de même orientation. Certaines petites cellules convexes adjacentes issues de la décomposition peuvent donc être fusionnées en de plus grandes cellules à posteriori de la décomposition.

4.3.2 Classification de chaque sommet

Comme dit précédemment, la première étape de l'algorithme consiste à identifier pour chaque sommet l'évènement qui lui correspond. On peut classer ces évènements en deux grandes catégories : ceux qui engendrent la création ou la complétion d'une cellule, et les autres.
Dans la première catégorie, on retrouve les évènements Open, Split, Merge et Close. Un sommet S est un événement Open si ses deux sommets voisins se trouvent sur le côté droit de la ligne de balayage et si l'angle intérieur de S est inférieur à π ; sinon S est un événement Split. Un sommet est un événement Close si ses deux sommets voisins se trouvent sur le côté gauche de la ligne de balayage et si l'angle intérieur à S est inférieur à π ; sinon S est un événement Merge.
C'est dans la deuxième catégorie que Xin Yu propose une amélioration par rapport aux méthodes de décomposition trapézoïdales classiques. Alors que celles-ci ne proposent qu'un type d'évènement dans cette catégorie, Xin Yu en distingue ici quatre : les évènements Floor Convex, Ceil Convex, Floor Concave et Ceil Concave. Un sommet S est un événement Floor Convex si le sommet précédent Sprev se trouve sur le côté gauche de la ligne de balayage tandis que son voisin suivant Snext se trouve sur le côté droit de la ligne de balayage, et si l'angle intérieur de S est inférieur à π ; sinon S est un événement Floor Concave. Au contraire, un sommet S est un événement Ceil Convex si le sommet précédent Sprev se trouve sur le côté droit de la ligne de balayage tandis que son voisin suivant Snext se trouve sur le côté gauche de la ligne de balayage, et l'angle intérieur à S est inférieur à π ; sinon S est un événement Ceil Concave. Exemple de classification, image : classification.png

4.3.3 Génération des sous-parties convexes

Une fois chacun des sommets classés, on applique le balayage avec la ligne verticale : pour chaque évènement rencontré, on effectue différentes actions, répertoriées dans les sous-parties suivantes. On identifie aussi les différentes arêtes que la ligne intersecte, qui nous aidera à compléter chaque cellule. Les arêtes « actives » seront ajoutées à une liste nommée ici simplement L. Chaque cellule comportera deux listes : une pour les sommets appartenant au « bas » de la cellule, Floor list l'autre pour les sommets en « haut » de la cellule, Ceil list.

Cas « OPEN »

Les deux arrètes incidentes de cet événement sont insérés dans L. Une nouvelle cellule est ouverte et le sommet de cet événement est ajouté à la Floor list de la cellule.

Cas « SPLIT »

Les arêtes immédiatement au-dessus et au-dessous de cet événement sont recherchées dans L. Ensuite, l'intersection de la ligne de balayage et de l'arête supérieure, et l'intersection de la ligne de balayage et de l'arête inférieure sont déterminées. On trouve la cellule à laquelle appartient cet événement. On ajoute ensuite les points d'intersection au-dessus et au-dessous dans la Ceil list et la Floor list de la cellule, respectivement, et

la cellule actuelle est clôturée. Ensuite, deux nouvelles cellules sont ouvertes. On ajoute le sommet de cet événement dans la Floor list de la nouvelle cellule supérieure et dans la Ceil list de la nouvelle cellule inférieure. On ajoute le point d'intersection du dessus dans la Ceil list de la nouvelle cellule supérieure et le point d'intersection du dessous dans la Floor list de la nouvelle cellule inférieure. On insère ensuite les deux arêtes incidentes de cet événement dans L.

Cas « MERGE »

Les deux arêtes incidentes de cet événement sont supprimées de L. Les arêtes immédiatement au-dessus et au-dessous de cet événement sont recherchées dans L. Ensuite, on détermine l'intersection de la ligne de balayage et de l'arête supérieure, ainsi que l'intersection de la ligne de balayage et de l'arête inférieure. On cherche les deux cellules auxquelles appartient cet événement. On ajoute le sommet de cet événement dans la Floor list de la cellule supérieure et dans la Ceil list de la cellule inférieure respectivement. On ajoute le point d'intersection supérieur dans la Ceil list de la cellule du haut et le point d'intersection inférieur dans la Floor list de la cellule du bas respectivement. Les deux cellules sont ensuite clôturées. Ensuite, une nouvelle cellule est ouverte. On ajoute le point d'intersection ci-dessus dans la Ceil list de la nouvelle cellule et le point d'intersection ci-dessous dans la Floor list de la nouvelle cellule, respectivement.

Cas « FLOOR_CONCAVE » / « CEIL_CONCAVE »

On supprime l'arête gauche incidente de cet événement de L. On recherche l'arête qui se trouve immédiatement au-dessus de cet événement dans L, et on détermine le point d'intersection avec la ligne de balayage. On ajoute ensuite l'arête droite incidente de cet événement dans L. On trouve la cellule à laquelle appartient cet événement. On ajoute le sommet de cet événement dans la Floor list de la cellule actuelle et le point d'intersection dans la Ceil list de la cellule actuelle respectivement. La cellule courante est alors clôturée. Après cela, une nouvelle cellule est ouverte. On ajoute le sommet de cet événement dans la Floor list de la nouvelle cellule et le point d'intersection dans la Ceil list de la nouvelle cellule respectivement.

Cas « FLOOR_CONVEX » / « CEIL_CONVEX »

On trouve la cellule à laquelle appartient cet événement. On ajoute le sommet de cet événement dans la Ceil list de la cellule actuelle. On supprime l'arête gauche incidente de cet événement de la liste L et on insère l'arête droite incidente de cet événement dans la liste L.

Cas « CLOSE »

Les deux arêtes incidentes de cet événement sont supprimées de L. On trouve la cellule à laquelle appartient cet événement. Le sommet de cet événement est ajouté à la Ceil list de la cellule actuelle et la cellule actuelle est clôturée.

4.3.4 Simplification par fusion

Une fois la décomposition effectuée, on parcourt la liste ses sous-parties convexes à la cherche de cellules adjacentes de même direction optimale. Si les deux cellules possèdent une arête commune, alors elles sont fusionnées. Si la frontière commune correspond à l'arête d'une cellule, mais seulement à une partie de l'arête de la seconde, elles sont laissées telles quelles sont.

4.4 Gestion des virages : la trajectoire du robot

Nous avons maintenant obtenu un ensemble de sous-parties convexes sur lesquelles sont générés les boustrophédons qu'effectuera la charge utile. Il est maintenant question de générer la trajectoire du robot qui permettra à la charge utile d'effectuer les boustrophédons.

4.4.1 Stratégie de virage

Une première idée intuitive pour générer la trajectoire du robot est de se placer à une distance l égale à la longueur du câble entre le robot et la charge utile, suivant la tangente à la courbe de la trajectoire souhaitée pour la charge utile.

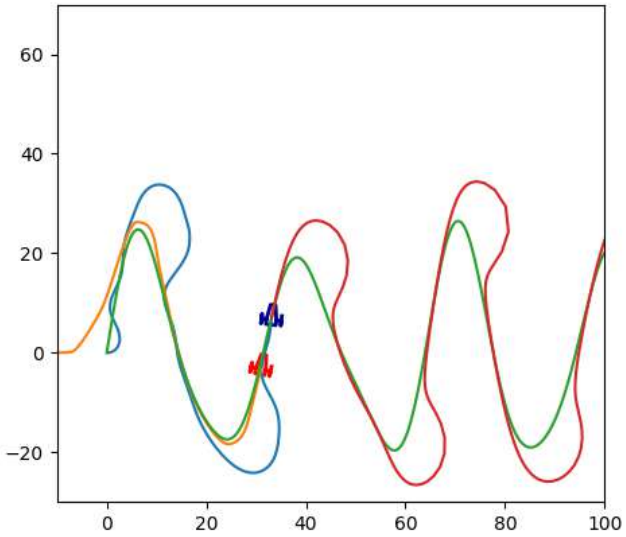

Figure 43 : Exemple de trajectoire générée

En bleu, on voit la trajectoire suivie par le robot avec en consigne la courbe rouge. En vert la trajectoire souhaitée de la luge et en orange la trajectoire réelle de la luge. On a donc pu résoudre le cas de la courbe décrite mathématiquement. Seulement, cette stratégie doit être adaptée aux trajectoires de boustrophédon présentant des angles forts avec un nombre infini de tangentes. La stratégie finale retenue est la suivante :

→ Une fois la ligne droite entre deux *waypoints* w_1 et w_2 du boustrophédon effectué, le robot continue sur une ligne droite d'une distance l.

→ Le robot décrit ensuite un arc de cercle de rayon l, avec pour centre w_2. Cela amène le robot dans l'axe de la ligne suivante, entre w_2 et le prochain *waypoint*.

4.4.2 Interface graphique

Pour pouvoir mieux avancer sur le projet, mais aussi donner une meilleure visibilité du résultat à l'utilisateur, la mise en place d'une interface graphique, réalisée avec PyQt5, s'est imposée comme un incontournable.

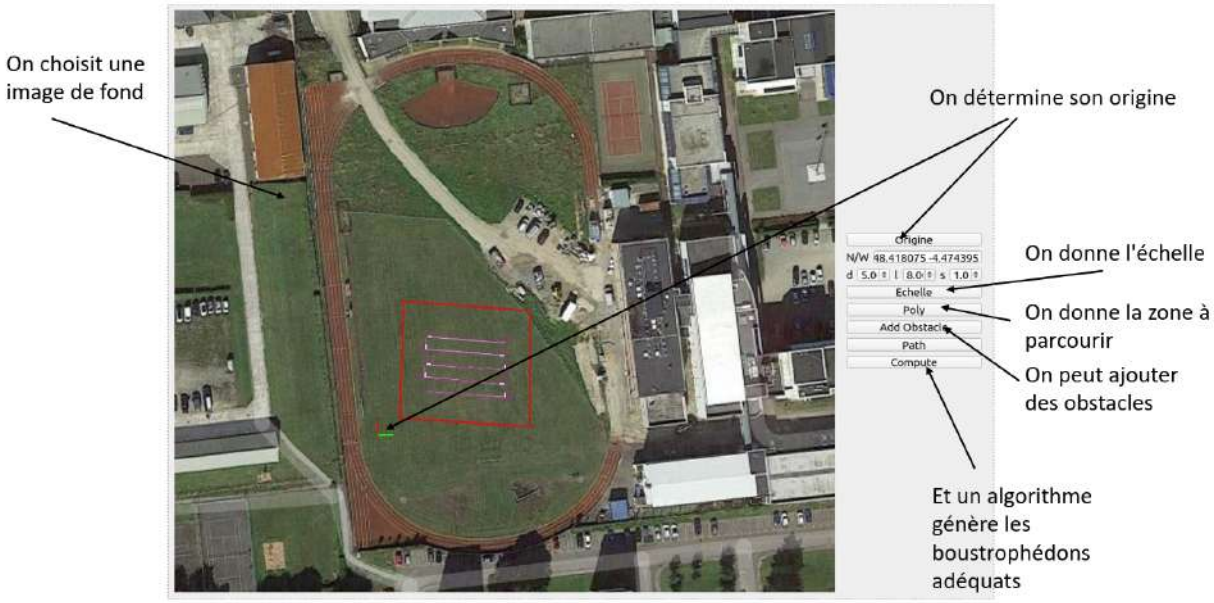

Figure 44 : Rendu de l'interface graphique de la partie planification

Elle permet à l'utilisateur de générer facilement la surface d'étude qu'il souhaite explorer, mais aussi de pouvoir changer assez facilement cette dernière si la zone choisie n'est pas compatible avec la solution de décomposition développée. Elle génère ensuite un fichier au format texte contenant toutes les informations nécessaires à la génération des commandes du robot Saturne.

4.4.3 Tests sur le robot Saturne

Le modèle de trajectoire appliqué au robot a pu être testé sur le robot Saturne. Si les résultats semblent prometteurs lorsque la largeur de fauchée entre les lignes du boustrophédon est suffisamment grande, le robot a plus de mal en pratique à maintenir la corde tendue lorsque l'écart entre la longueur l du câble tractant la charge utile et la largeur de fauchée est trop grand.

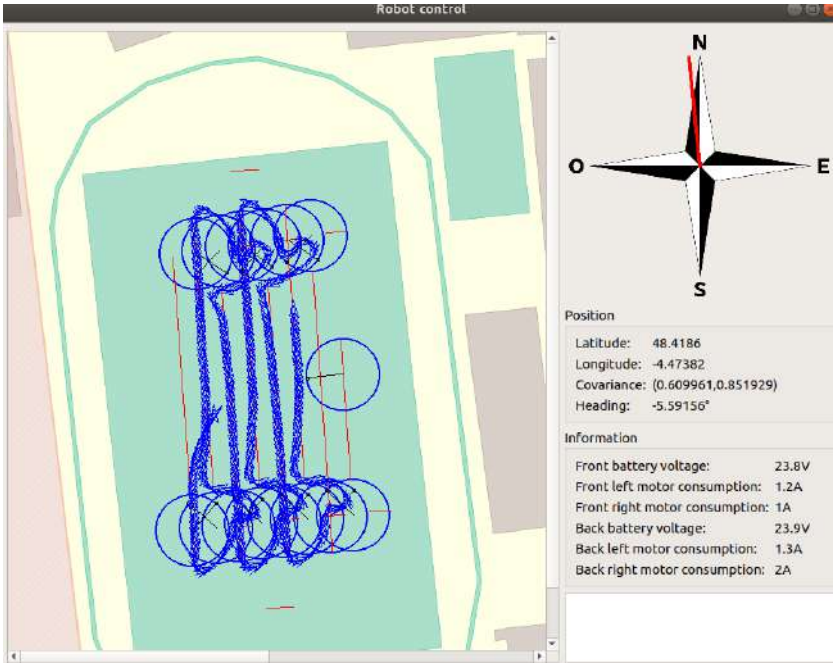

Figure 45 : Essai avec le robot Saturne

4.5 Prise en compte de la stratégie de virage dans la génération de trajectoire

Puisque nous devons garantir que ni le robot, ni la charge utile ne sortent de la zone ou ne rencontrent un obstacle, il faut prendre en compte la stratégie de virage en compte dans la génération des boustrophédons. La méthode retenue est de calculer la distance maximum R parcourue par le robot en dehors du boustrophédon pour effectuer son virage, que l'on peut en pratique ajouter à une distance de sécurité S. Le problème se ramène alors à un problème de Polygon Offset : il suffit de dilater les polygones obstacles d'un facteur R+S et d'éroder le polygone extérieur du même facteur. On souhaite appliquer cette transformation avant même la décomposition, et ainsi garantir que quelle que soit cette dernière, le robot disposera d'un espace suffisant à l'intérieur de la zone autorisée pour effectuer son virage. Toutefois, cette opération s'avère particulièrement complexe car elle peut profondément modifier la structure de la zone à explorer. Même si une librairie externe comme PyClipper a été utilisée, la solution globale de planification ne supporte pour l'instant que des changements mineurs.

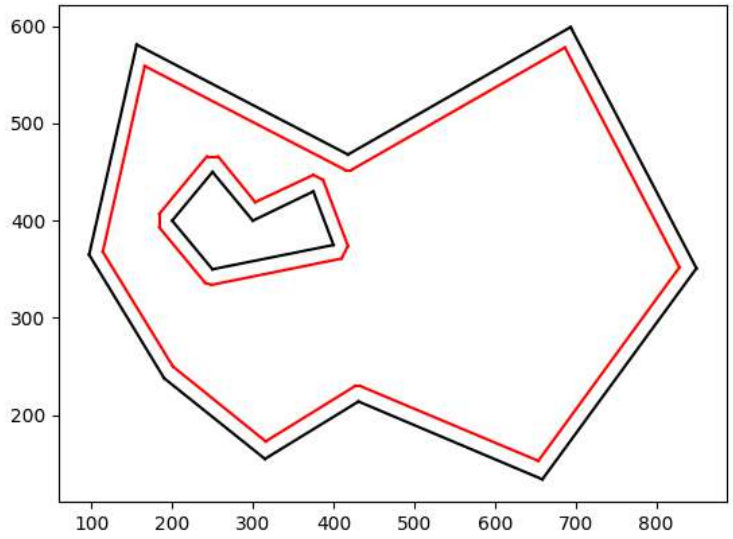

Figure 46 : Exemple d'un problème de Polygon Offset

4.6 Ordre de parcours

Une fois la décomposition effectuée, nous disposons d'un ensemble de sous parties convexes sur lesquelles nous avons généré des boustrophédons. Cependant, l'ordre dans lequel elles seront parcourues par le robot peut entraîner une large augmentation de la distance totale parcourue par le robot, sans pour autant apporter de valeur ajoutée à l'acquisition. Il convient alors de déterminer dans quel ordre effectuer les boustrophédons pour minimiser la distance non utile parcourue.
Ce problème correspond à un problème mathématique connu : le *Traveling Salesman Problem*. Les nœuds correspondront ici à chacun des points d'entrée et de sortie des différents boustrophédons. Les poids associés aux nœuds correspondront aux distances physiques entres les deux points, à l'exception du poids correspondant à la liaison entre deux points d'un même boustrophédon. Ces poids particuliers se verront affecter un poids négatif important, pour « forcer » le passage consécutif de ces deux points. Une fois le problème posé, de nombreux algorithmes proposent une solution sous-optimale acquise rapidement, dont nous nous contenteront ici.

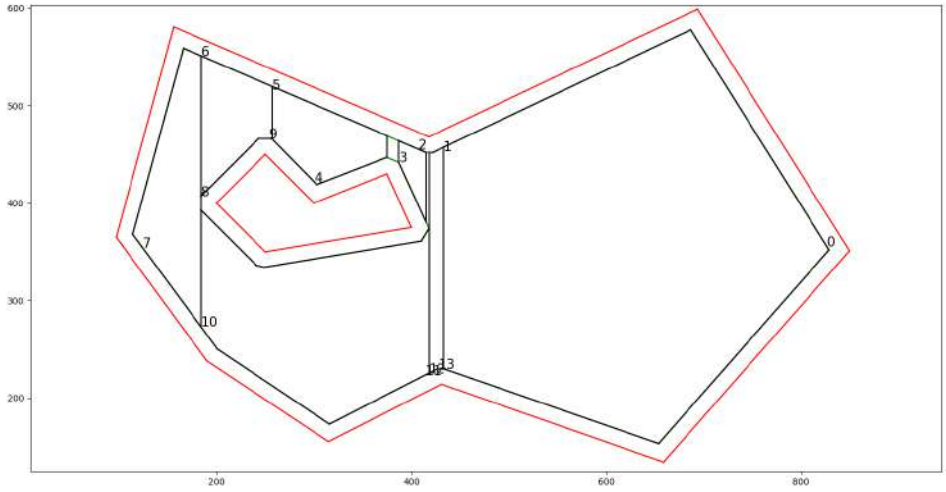

Figure 47 : Exemple d'un résultat sous-optimal d'ordonnancement

4.7 Résultats

La solution finale est fonctionnelle dans des cas relativement simples. Pour une zone convexe ou non convexe sans obstacle l'algorithme trouvera le plus souvent une bonne décomposition. L'ajout de petits obstacles parsemés converge lui aussi assez souvent vers une solution. Cependant, deux problèmes majeurs empêchent parfois la décomposition :

→ La fonction permettant de retrouver la cellule à laquelle appartient un évènement est pour l'instant relativement simple, et manque de robustesse lorsque le champ de polygone se complexifie.

→ Un changement profond du champ de polygone suite à l'érosion / dilation de ces derniers d'un facteur R+S n'est pas supporté par l'algorithme.

→ Par manque de temps, l'algorithme n'intègre pas encore d'évitement d'obstacle dans la phase de ralliement vers le prochain boustrophédon.

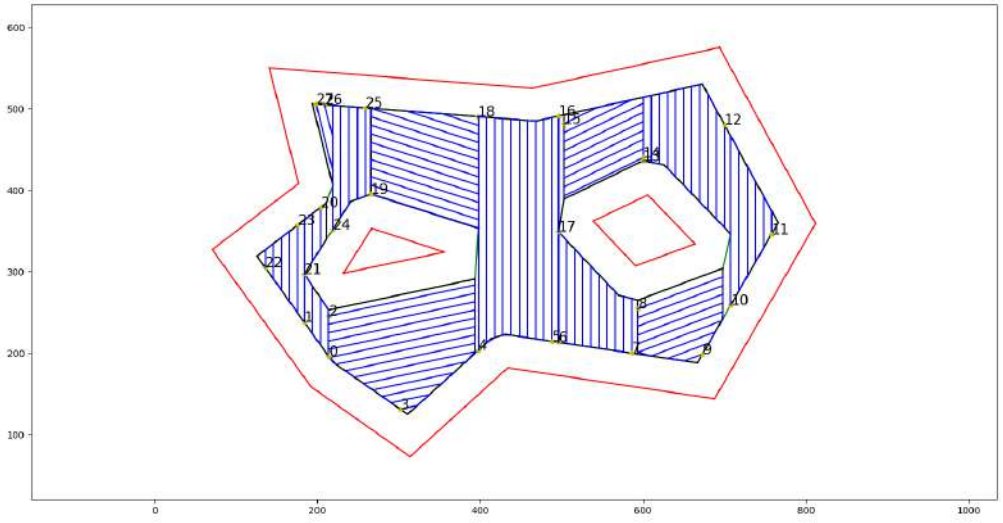

Figure 48 : Résultat d'une solution de planification

Post-processing des données du robot Saturne

5.1 Modélisation du système

Le système est composé d'un véhicule de type char tractant une luge embarquant un magnétomètre. Le véhicule de remorquage est relié à la luge par un câble. En supposant que ce câble est constamment sous tension, nous sommes en mesure de déduire l'équation décrivant la cinématique du système. En notant x, l'état du système et u ses valeurs d'entrées, la cinématique du système est décrite par :

$$f(\mathbf{x}(t), \mathbf{u}(t)) = \begin{cases} \dot{\mathbf{x}}_1 = \frac{\mathbf{u}_1+\mathbf{u}_2}{2}.cos(\mathbf{x}_3) & (5.1a) \\ \dot{\mathbf{x}}_2 = \frac{\mathbf{u}_1+\mathbf{u}_2}{2}.sin(\mathbf{x}_3) & (5.1b) \\ \dot{\mathbf{x}}_3 = \mathbf{u}_2 - \mathbf{u}_1 & (5.1c) \\ \dot{\mathbf{x}}_4 = -\frac{\mathbf{u}_1+\mathbf{u}_2}{2}.sin(\mathbf{x}_4) - \dot{\mathbf{x}}_3 & (5.1d) \end{cases}$$

Figure 49 : Équation cinématique du système

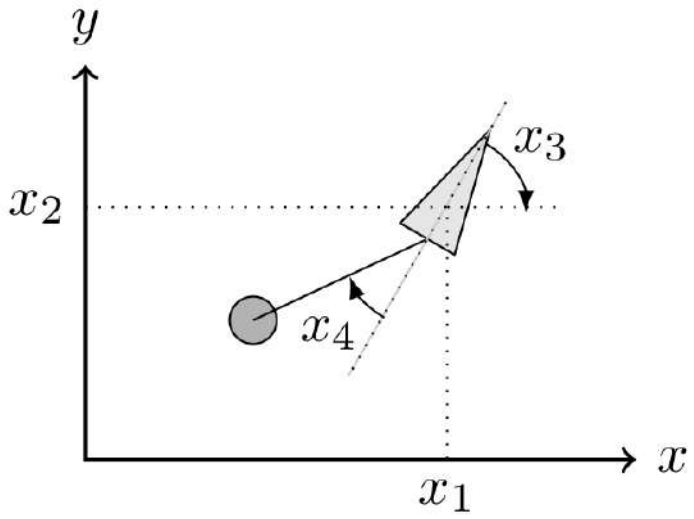

Figure 50 : Modélisation cinématique du robot tractant une luge

Ici, x_1, x_2 et x_3 sont respectivement l'abscisse, l'ordonné et l'orientation du robot de tractage, x_4 est l'angle de la luge avec le robot.

5.2 Simulateur

Une simulation Python a été réalisée en utilisant VIBes [DN14] afin de tester le comportement du système. Une classe Tank a été créée pour instancier un véhicule avec sa luge. Ensuite, le script intègre l'équation d'évolution en utilisant la méthode d'Euler afin d'obtenir l'état du système en fonction des données fournies. Nous avons pu constater que le comportement du système semble correct et que le modèle est fidèle à la réalité. De plus, le capteur GNSS et un accéléromètre sont simulés afin d'enfermer leur position dans une boîte.

5.3 Encadrement fiable de l'angle de la luge

Considérant que la corde reste continuellement sous tension pendant que le robot est en mouvement, alors l'évolution de l'angle x_4 entre le robot et la luge est pilotée par l'équation différentielle suivante :
$\dot{\mathbf{x}}_4 = -\frac{\mathbf{u}_1+\mathbf{u}_2}{2}.sin(\mathbf{x}_4) - \dot{\mathbf{x}}_3$

Nous pouvons considérer, en début d'expérience, que x_4 est compris dans l'intervalle $[\frac{-\pi}{2}, \frac{\pi}{2}]$, il s'agit de la condition initiale sur l'équation différentielle. La réalité physique veut qu'après quelques secondes de mouvements la luge suive la trajectoire du robot, cela quelque soit la condition initiale. A partir de deux méthodes : la méthode de Monte-Carlo, puis la méthode de l'analyse par intervalles nous allons prouver que l'équation différentielle décrit bien la réalité physique.

Dans la méthode de Monte-Carlo, on choisit un nombre discret de points dans l'intervalle $[\frac{-\pi}{2}, \frac{\pi}{2}]$ et on résout itérativement par une méthode d'Euler l'équation différentielle, on remarque qu'après quelques secondes les trajectoires liées à chaque condition initiale sont confondues, aucune de ces conditions initiales ne rend l'équation différentielle divergente.

La méthode de Monte-Carlo est statistique, i.e. le nombre de conditions initiales testées est discret. Autrement dit, toutes les hypothèses sur la position initiale de la luge n'ont pas été testées.

L'analyse par intervalles élimine cette incertitude. En analyse par intervalles, la brique élémentaire n'est plus le nombre comme en analyse classique (utilisée pour la méthode de Monte-Carlo) mais l'intervalle : un nombre est décrit par un intervalle, un ensemble est décrit par un intervalle. Un intervalle est donc un ensemble de réels infini compris entre deux bornes, on a donc l'ensemble des angles initiaux possibles de la luge dans $[\frac{-\pi}{2}, \frac{\pi}{2}]$.

L'intervalle initial est malheureusement trop large pour pouvoir avoir une convergence de l'angle de la luge. On découpe donc l'intervalle $[\frac{-\pi}{2}, \frac{\pi}{2}]$ en n sous-intervalles réguliers, puis on intègre ces n sous-intervalles grâce à l'équation différentielle, enfin on fait l'union des solutions afin d'obtenir l'intervalle contenant l'angle de la luge à l'instant suivant. Par itération, on construit un tube contenant l'angle de la luge au cours du temps.

Le résultat montre bien une contraction de l'intervalle de départ $[\frac{-\pi}{2}, \frac{\pi}{2}]$ (figure 3.) . La luge a un comportement physiquement réaliste. Autrement dit, peu importe la position

de la luge au départ, la luge suivra après quelques dizaines de secondes la trajectoire de l'UGV. On peut donc valider notre modélisation cinématique avec une confiance totale.

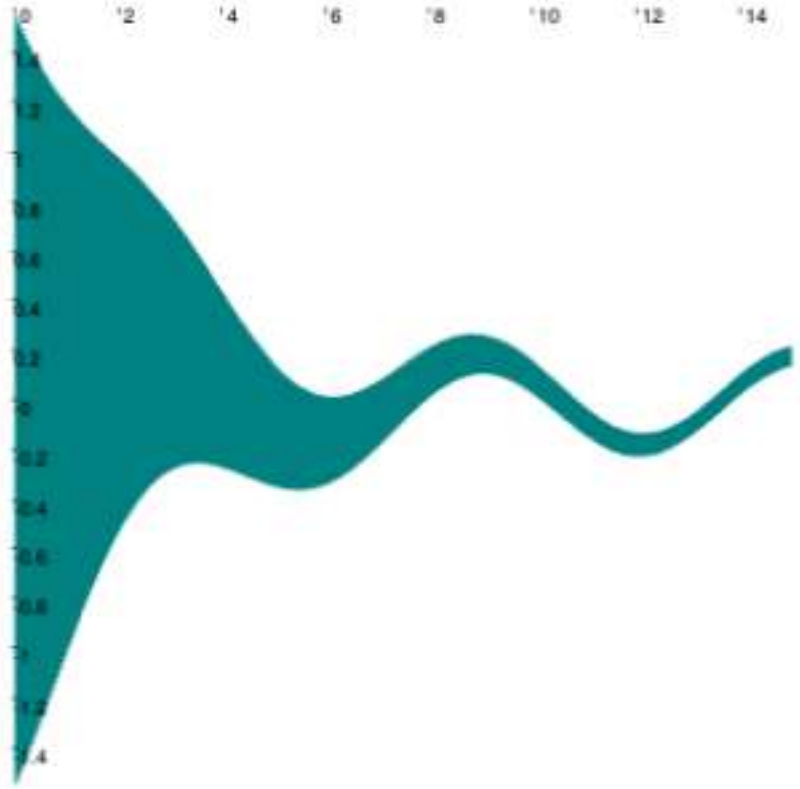

Fig. 3. Integration example using intervals

Figure 51 : Exemple d'intégration utilisant l'analyse par intervalles

5.4 Localisation de la luge

En disposant d'une boîte contenant l'UGV, et d'un intervalle encadrant l'angle x_4, nous pouvons obtenir une boîte contenant la luge, en connaissant la longueur de la corde. Cette boîte a la forme d'un secteur de diagramme circulaire. L'équation 4 présente la position du magnétomètre en fonction de l'état du système x_1, x_2 et x_4. En appliquant un contracteur polaire à ces intervalles, nous pouvons obtenir les intervalles contenant x_m et y_m.

5.5 Localisation ensembliste du système

L'objectif de nos travaux est d'obtenir des garanties quant à la surface couverte par le magnétomètre. Pour obtenir ces garanties, nous avons dû utiliser des méthodes de localisation ensemblistes pour déterminer la position de l'ensemble du système à chaque

instant. Pour réaliser cette localisation nous avons utilisé un outil permettant de caractériser un ensemble au cours du temps : les tubes.

5.5.1 Les tubes

Un tube [R+17] est un outil permettant de représenter la trajectoire du robot au cours du temps. Dans un tube, chaque position du robot est associée à un instant. La plus-value par rapport aux méthodes d'intervalles classiques et que l'on peut contraindre l'ensemble des tubes par des contraintes temporelles. Un tube $[x](\cdot)$ est défini sur un domaine temporel $t \in [t0, tf]$ comme une enveloppe de trajectoires définies sur le même domaine t. Nous parlons d'une enveloppe car il peut exister des trajectoires enfermées dans le tube qui ne sont pas des solutions à notre problème. Une trajectoire $x(\cdot)$ appartient au tube $[x](\cdot)$ si $\forall t \in [t0, tf], x(t) \in [x](t)$.

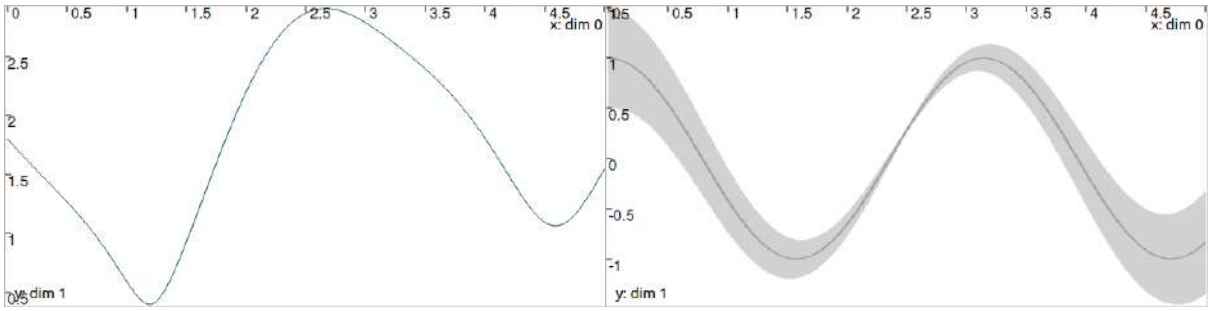

Figure 52 : Exemple de tube

On peut voir ci-dessus un exemple de tube contenant une trajectoire définie. Pour représenter la trajectoire du robot de façon fiable, il faut contraindre ce tube via un réseau de contraintes. Autrement, du fait de l'incertitude des capteurs et de l'intégration successive des équations différentielles décrivant la dynamique d'un système, le tube ne ferait qu'augmenter, tout comme l'incertitude sur l'état du système, et ne serait pas un outil fiable.

5.5.2 Réseau de contraintes

De façon à obtenir une trajectoire la plus précise possible, il nous a fallu définir les contraintes inhérentes à notre système. Pour obtenir ce système de contraintes, nous avons décomposé les équations d'état du système. Ces équations peuvent être décomposées suivant cet ensemble de contraintes élémentaires :

$$\begin{cases} (i) & \mathbf{v}(\cdot) = f(\mathbf{x}(\cdot), \mathbf{u}(\cdot)) \\ (ii) & \dot{\mathbf{x}}(\cdot) = \mathbf{v}(\cdot) \\ (iii) & \dot{\mathbf{v}}(\cdot) = \mathbf{a}(\cdot) \\ (iv) & \mathbf{d}(\cdot) = \mathbf{x}_{1,2}(\cdot) - \mathbf{x_m}(\cdot) \\ (v) & \mathbf{t} = \mathbf{x}_3 + \mathbf{x}_4 \\ (vi) & \mathcal{L}_{polar}\left(\mathbf{d}(\cdot),\ [L],\ \mathbf{t}\right) \\ (vii) & \mathcal{L}_{eval}\left(t_i,\ [\mathbf{y}_{1,2}],\ \mathbf{x}_{1,2}(t_i),\ \mathbf{v}_{1,2}(t_i)\right) \\ (viii) & \mathcal{L}_{eval}\left(t_j,\ [\mathbf{y}_{4,5}],\ \mathbf{v}_{1,2}(t_j),\ \mathbf{a}_{1,2}(t_j)\right) \end{cases} \tag{5.2}$$

Ces contraintes impliquent des variables intermédiaires introduites pour faciliter la décomposition. De plus ces contraintes contiennent les contraintes liant le robot et la luge. Ce qui permet à partir du tube contenant la trajectoire du robot d'obtenir le tube contenant la trajectoire de la luge. Ainsi cet ensemble de contraintes appliqué sur le tube englobant la trajectoire du robot, permet à partir d'un ensemble de points obtenues par les capteurs du robot, d'obtenir les tubes de la trajectoire du robot et de la luge :

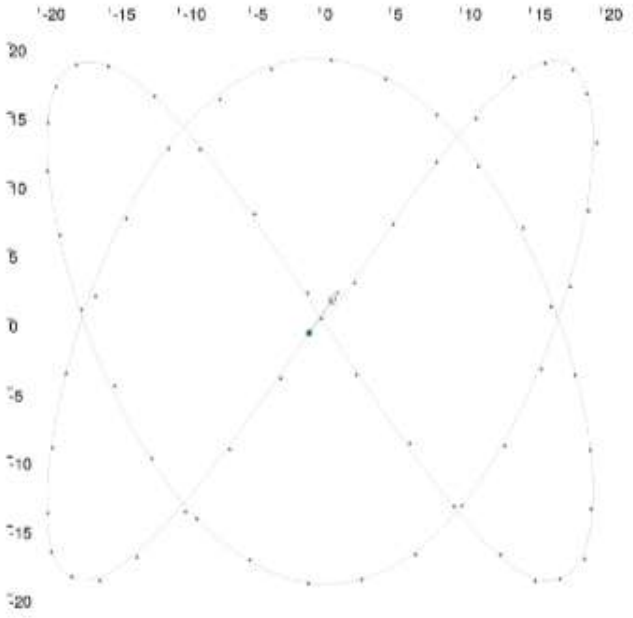

Figure 53 : Trajectoire réel du système

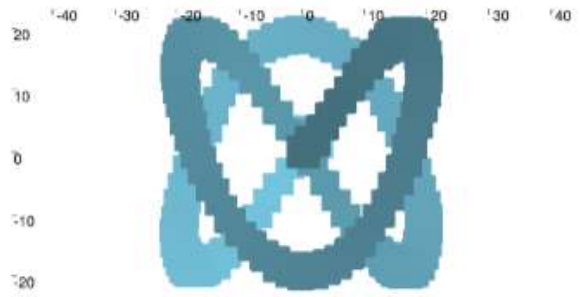

Figure 54 : Tube englobant la trajectoire du robot

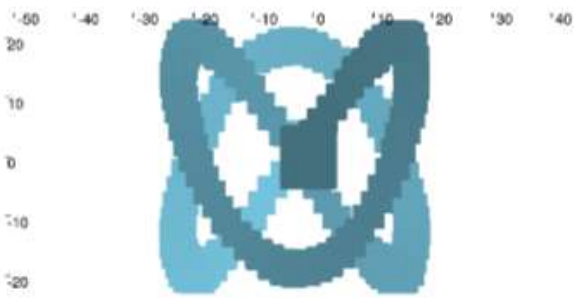

Figure 55 : Tube englobant la trajectoire de la luge

On remarque que l'incertitude liée au positionnement de la luge à l'état initial fait que le tube englobant la luge est très large au début, mais que l'intervalle encadrant l'angle entre la luge et le robot converge très rapidement, et par conséquent l'estimation de la position de la luge est vite fiable.

5.6 Couverture de la cartographie

A la fin de chaque mission il est nécessaire de déterminer la zone qui a été explorée par le magnétomètre. Cela permet d'éventuellement revenir sur les zones dont on ne possède pas encore de mesures, mais aussi de s'assurer que certaines zones n'ont pas été approchées par le magnétomètre, comme par exemple des zones dangereuses ou confidentielles.
Il est possible d'utiliser les ThickSets pour caractériser ces zones. En effet, cette bibliothèque basée sur l'arithmétique des intervalles permet de déterminer un Subset et un Supset qui sont deux ensembles permettant d'en encadrer un troisième. L'idée est de considérer que le magnétomètre possède un intervalle de mesure circulaire de rayon 5 m. Il se pose alors la question de savoir ce qu'est capable de voir ce capteur sachant que sa position est incertaine. L'idée est donc de caractériser la zone qui va être mesurée à coup sûr qui va donc être l'intersection des cercles de mesures dont le centre se trouve dans la boîte contenant le magnétomètre, et la zone peut-être mesurée par la réunion de ces intervalles. Ces zones sont respectivement représentées en rouge et en orange sur la figure ci-dessous.

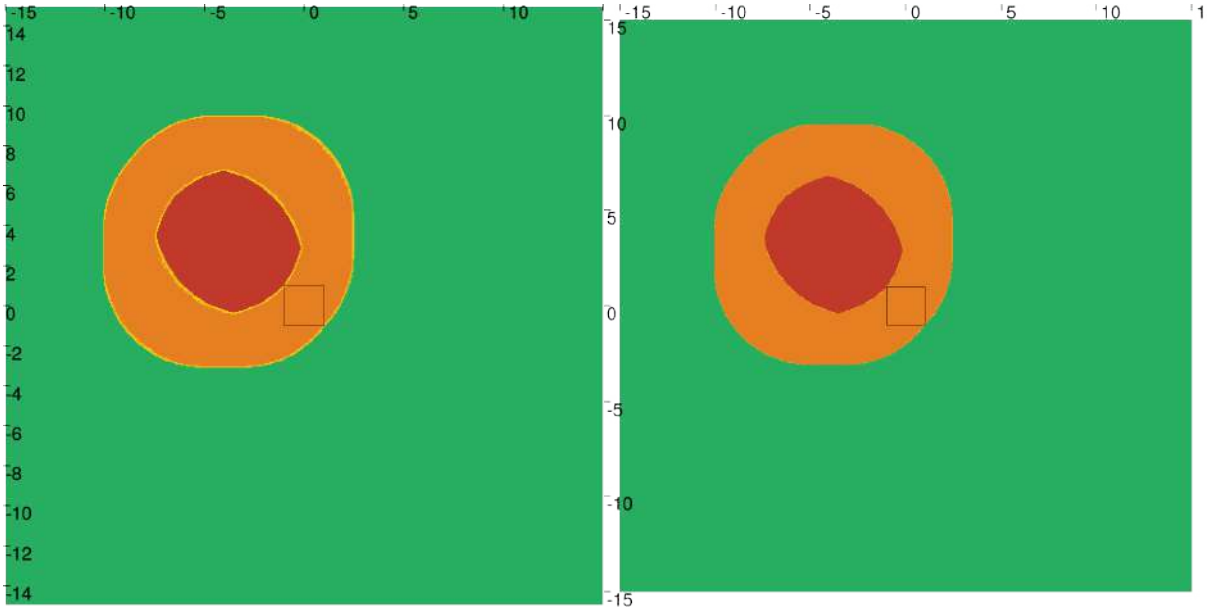

Figure 56 : Pavage avec une précision de 0.1m et précision 0.01m

Cette méthode de pavage est basée sur l'algorithme Sivia (Set Inversion Via Interval Analysis). Elle présente l'avantage d'être rapide lors de la résolution numérique de problèmes d'inversion ensembliste comme ici. Il reste cependant une zone à la frontière de chacune des autres qui est qualifiée d'incertaine et qui est représentée ici en jaune. C'est une zone pour laquelle on ne peut pas être sûr que le magnétomètre ait pu prendre des mesures. C'est en fait la condition d'arrêt de l'algorithme Sivia, si les boîtes à traiter sont plus petites qu'une certaine valeur de seuil à fixer, alors il les classe comme incertaines. En fonction de cette valeur on est en mesure de maîtriser l'aire de cette zone incertaine afin de la réduire, mais cela se fait au prix du temps de calcul qui augmente.
A partir d'une boîte noire dans lequel on est sûr que le magnétomètre se trouve, il est possible donc de trouver ces deux zones qui sont caractérisées par la fonction suivante :

$$f(x_1, x_2, c_1, c_2, r, \phi) = \sqrt{((c_1 + r.cos(\phi) - x_1)^2 + (c_2 + r.sin(\phi) - x_2)^2)}$$

Ici c_1 et c_2 représente respectivement les intervalles d'abscisse et d'ordonnée de la boîte contenant le magnétomètre, r est la longueur de la corde, et ϕ l'angle entre le robot tracteur et la luge. Cela permet de caractériser l'ensemble X d'abscisse x_1 et d'ordonnée x_2 qui va être la zone vue par le magnétomètre. Ensuite la bibliothèque ThickSets permet de réaliser l'intersection et la réunion de tous ces ensembles afin de déterminer les zones explorées.
A la suite d'une mission il est donc possible de réaliser une carte caractérisant la couverture de la cartographie. Cela se fait avec l'union de chaque ensemble trouvé à chaque instant. Quelques exemples sont disponibles ci-dessous et montrent les sorties de la couverture de la cartographie. On est ici bien en mesure de voir la zone qui a été vue à coup sûr en rouge, la zone peut-être vue en orange et la zone inexplorée en vert.

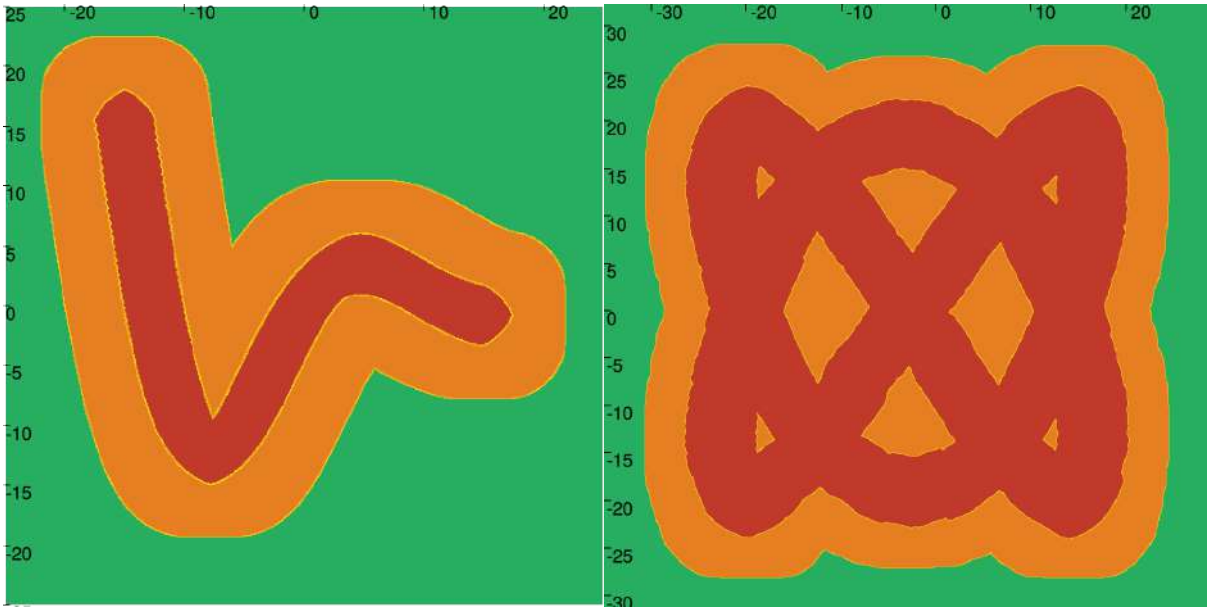

Figure 57 : Pavage représentant l'exploration de la zone

Nous n'avons pas eu la possibilité de réaliser une mission sous-marine avec le Riptide donc nous n'avons pas de données à post-traiter pour essayer cet algorithme. Pour ce qui est de la mission terrestre, nous n'avions pas assez de données notamment en provenance de l'IMU, puisque le modèle que nous avions ne fournissait pas de vitesses estimées. Cela pose un problème dans le réseau de contraintes qui ne possède pas assez de contraintes pour fournir une trajectoire réaliste du robot Saturne et de la luge.
Les résultats visibles sur la figure ci-dessus ont été réalisés en simulation et avec un système de GNSS classique présentant une précision de l'ordre de 3 mètres. Cela permet de correctement représenter les résultats qu'on aurait pu obtenir avec une expérimentation sous l'eau, puisque la précision du positionnement sous-marin aurait été de cet ordre de grandeur. Cela justifie ici le fait que nous n'avons pas utilisé de système de correction RTK sur le positionnement pour avoir un positionnement précis à 10 centimètres près.

Résultats

Les expérimentations avec le robot Saturne qui ont été effectuées sur le stade de l'ENSTA Bretagne ont permis d'établir la carte magnétique suivante (sensibilité pour les anomalies : 0.5×10^{-9} T).

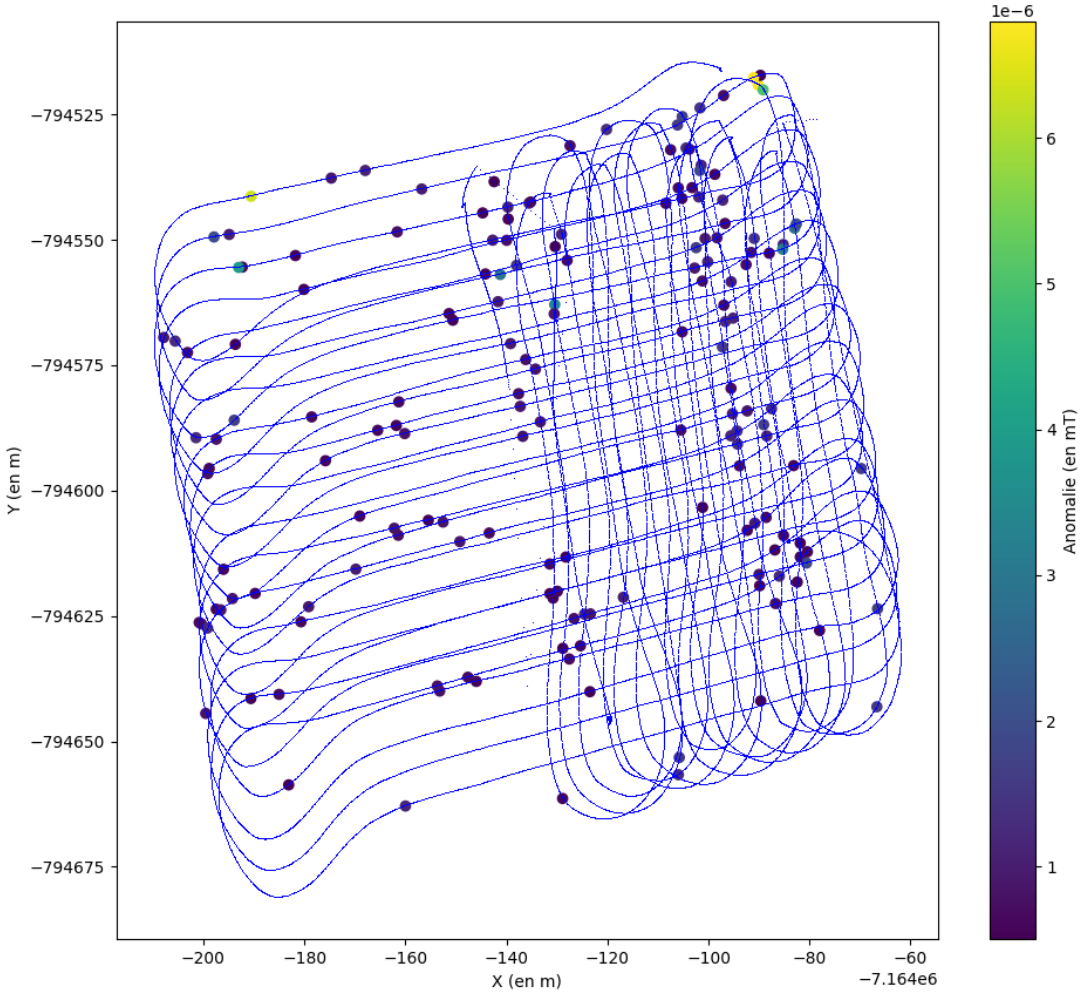

Figure 58 : Carte magnétique du stade de l'ENSTA Bretagne

La trajectoire est celle de la luge contenant le magnétomètre (reliée à Saturne avec une corde de 8 mètres). On remarque une anomalie plus forte que les autres en haut à droite : il s'agit sûrement de l'aiment caché sur le stade par Luc Jaulin.

Conclusion

L'objectif de notre projet était de réaliser des cartes magnétiques à l'aide de robots autonomes. Du point de vue terrestre, notre projet est réussi étant donné que nous avons été en mesure de produire une carte magnétique du stade de l'ENSTA Bretagne. La stratégie de virage de Saturne reste à optimiser.
Pour la partie sous-marine, le robot Riptide a été testé avec succès lors de manipulations au lac de Guerlédan la seconde semaine de février 2021. L'ajout d'un magnétomètre sur celui-ci reste à effectuer, ainsi que de trouver un moyen de le localiser précisément.
Les simulations développées pour les deux robots sont un outil très puissant pour préparer les missions de ces derniers. Elles permettront de tester des missions plus complexes que celles réalisées jusqu'à présent, ce qui représente un gain de temps important sur le terrain.

Bibliographie

[BR90] Larry Barrows and Judith E. Rocchio. Magnetic surveying for buried metallic objects. *Ground Water Monitoring Review*, 1990.

[DN14] Vincent Drevelle and Jeremy Nicola. Vibes: A visualizer for intervals and boxes. *Mathematics in Computer Science*, 8(3-4):563–572, 2014.

[Haq16] Ijaz Ul Haq. Presentation on magnetometers, September 2016.

[Hua01] W. H. Huang. Optimal line-sweep-based decompositions for coverage algorithms. In *Proceedings 2001 ICRA. IEEE International Conference on Robotics and Automation (Cat. No.01CH37164)*, volume 1, pages 27–32 vol.1, 2001.

[Mag15] Sensys Magnetometers. Fgm3d datasheet. Technical report, 2015.

[MSV+16] Musa Morena Marcusso Manhães, Sebastian A. Scherer, Martin Voss, Luiz Ricardo Douat, and Thomas Rauschenbach. UUV simulator: A gazebo-based package for underwater intervention and multi-robot simulation. In *OCEANS 2016 MTS/IEEE Monterey*. IEEE, sep 2016.

[R+17] Simon Rohou et al. The Tubex library – Constraint-programming for robotics, 2017. http://simon-rohou.fr/research/tubex-lib/.

[SA17] Metrolab Technology SA. Fluxmeters: magnetometer technology primer, 2017.

[Sch18] Romain Schwab. Recherche d'épaves par un zodiac autonome tractant un magnétomètre, 2018.

[Vid13] NASA Videos. Fluxgate magnetometers, 2013.

[Yu15] X. Yu. *Optimization Approaches for a Dubins Vehicle in Coverage Planning Problemand Traveling Salesman Problems*. PhD thesis, Auburn University, 2015.

Annexes

Dépôt Github du plugin Gazebo du Riptide :
https://github.com/MourtazaKASSAMALY/riptide_plugin
Site Magmap :
https://magmap.netlify.app/index.html

Figure 59 : La luge utilisée avec le magnétomètre, un récepteur GNSS et une pièce de bois pour alourdir la luge