Projet Scout

Localisation au sein d'une meute de robots sous-marins

Table des matières

Table des illustrations 3
1. Présentation et architecture du projet 4
1.1 Principe 4
1.2 Outils utilisés 5
1.2.1 Description des outils 5
1.2.2 Interfaçage des outils 6
1.3 Conventions de repère 6
1.4 L'AUV 7
1.4.1 Actionneurs 7
1.4.2 Les équations de mouvement 7
1.4.3 Configuration des équations de mouvement 7
2 Localisation par intervalles 8
2.1 Théorie des intervalles 8
2.2 Algorithme Sivia 9
2.3 Simplification de la bibliothèque dans MOOS 9
3 Commande par champ de vecteurs 10
3.1 Commande 10
3.2 Conversion en commandes unitaires 11
4 Démonstrations 12
4.1 Localisation par intervalles et singularités 12
4.1.1 Singularité en ligne droite 12
4.1.2 Rompre la singularité 13
5 Utilisation du Middleware MOOS 14
5.1 Architecture MOOS 14
5.2 uSimState : Implémentation des équations d'état 15
5.2.1 Rôle 15
5.2.2 Constitution 15
5.2.3 Tests 15
6 Interface Homme-Machine 17
6.1 Interface graphique du simulateur : MORSE/BLENDER 17
6.1.1 Présentation de MORSE 17
6.1.2 Implémentation des équations d'état et visualisation 17
6.1.3 Tests 18
6.2 Interaction avec le logiciel 19
Conclusion 20
Bibliographie 21

Table des illustrations

Figure 1 : Schéma de fonctionnement du bateau et des scouts 4
Figure 2 : mécanisme interne d'un scout implémenté sous MOOS 5
Figure 3 : relation entre MOOS, MORSE et Blender 6
Figure 4 : Représentation du robot, du repère absolu et de l'effet des actionneurs 6
Figure 5 : résolution de $\boldsymbol{x}^2 = \mathbf{0}$ avec le calcul par intervalles 8
Figure 6 : Exemple de contracteur 8
Figure 7 : Localisation avec l'algorithme Sivia sur de nombreux échantillons de temps 9
Figure 8 : commande des scouts (vidéo) 10
Figure 9 : Courbe de la fonction arc-tangente 11
Figure 10 : Courbe de la fonction dents de scie 11
Figure 11 : Problème de localisation en ligne droite (vidéo) 12
Figure 12: Singularité rompue (vidéo) 13
Figure 13 : Schéma de la répartition du travail en application MOOS 14
Figure 14 : Représentation avec MATLAB de l'état du robot 16
Figure 15 : Deux illustrations des usages possibles de MORSE 17
Figure 16 : Capture d'écran de la fenêtre BLENDER/MORSE lors d'une simulation 17
Figure 17 : Test d'accélération, image 1 18
Figure 18 : Test de plongée, image 1 18
Figure 19 : Test d'accélération, image 2 18
Figure 20 : Test de plongée, image 2 18
Figure 21 : Test d'accélération, image 3 18
Figure 22 : Test de plongée, image 3 18

1. Présentation et architecture du projet

Bertrand Moura

1.1 Principe

Dans le cadre de l'UV robotique mobile, la filière ROB de l'ENSTA Bretagne a développé un projet en coopération avec l'entreprise RTSys. RTsys est une entreprise basée près de Lorient en France et spécialisée dans la conception et la production d'équipements et de solutions dans les domaines des drones sous-marins, de l'acoustique sous-marine et de l'intégration de systèmes. En parallèle de notre formation, nous avons donc participé à l'élaboration du projet SCOUT, principalement en phases de recherche et de simulation.

Le projet met en œuvre deux robots sous-marins de type ROV, appelés « Scout » dans la suite de ce rapport. Ils doivent suivre une trajectoire consigne, la trajectoire du bateau qui se trouve derrière eux, en se positionnant à une certaine distance du bateau définie par l'utilisateur.

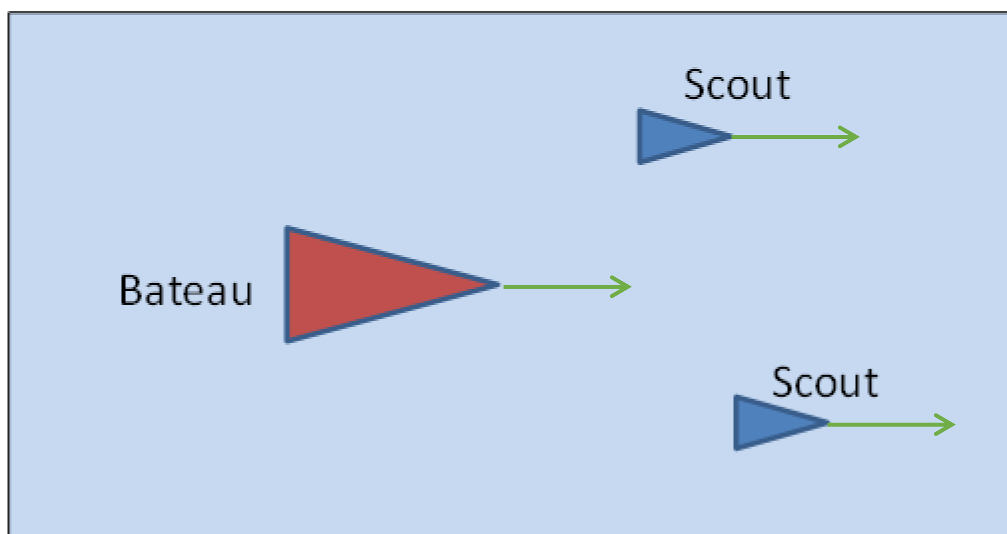

Figure 1 : Schéma de fonctionnement du bateau et des scouts

Ce projet soulève de nombreuses problématiques. Tout d'abord, étant donné que les scouts ne peuvent pas se localiser par GPS en milieu sous-marin, il faut trouver un autre moyen de localisation. Cela revêt une importance critique pour permettre aux scouts de connaître leur positionnement par rapport au bateau est ainsi de pouvoir déterminer une consigne de trajectoire. La solution que nous avons retenue est la localisation grâce à la théorie des intervalles. Concernant la trajectoire, une autre problématique sera de déterminer des lois de commandes efficaces pour pouvoir remplir notre objectif de suivi de trajectoire du bateau. Nous avons opté pour une commande par champ de vecteur. Enfin, afin de valider les solutions en simulation, il est important de créer un environnement de simulation proche de la réalité. Pour cela, nous avons utilisé de nombreux outils de modélisation permettant de simuler les différents éléments des scouts (capteurs, actionneurs), valider leur intelligence (loi de commande, localisation, contrôleur), visualiser les résultats sur une interface graphique, simuler la réalité en introduisant le contrôle du bateau par joystick. Tous ces éléments sont développés dans la suite du rapport.

1.2 Outils utilisés

1.2.1 Description des outils

Nous avons utilisé trois outils principaux pour réaliser ce projet : l'intergiciel (middleware) MOOS, le logiciel multiplateforme de développement en robotique MORSE, et le logiciel de modélisation Blender.

MOOS permet de créer un réseau d'échange d'informations entre différentes applications. Un robot scout peut se voir comme un ensemble d'applications qui communiquent et interagissent entre elles, au moyen de MOOS. On représente ci-dessous l'interfaçage des différentes applications du scout avec MOOS :

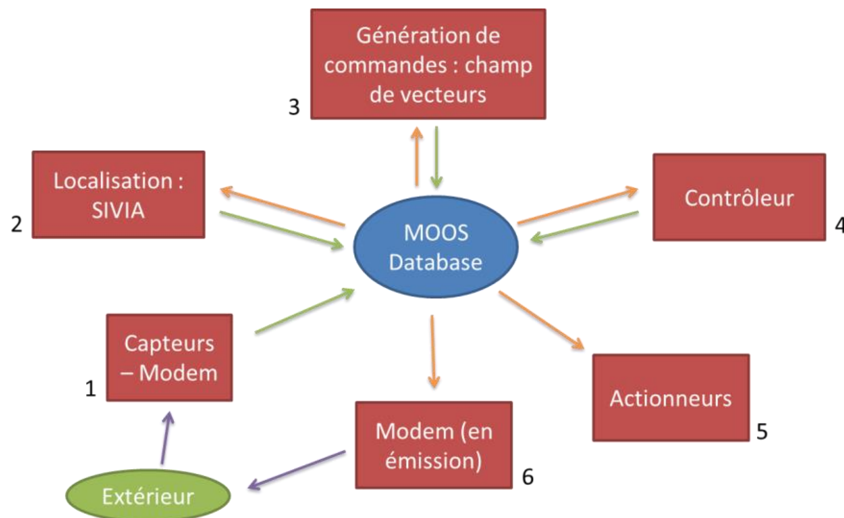

Figure 2 : mécanisme interne d'un scout implémenté sous MOOS

MORSE permet de faire de la simulation réaliste de robots dans des environnements extérieur ou intérieur. Il s'utilise facilement sur la base de scripts Python. On peut simuler les différents capteurs et actionneurs, ainsi que l'architecture du robot (forme, dynamique) de façon réaliste en s'appuyant sur des modèles déjà fournis ou en implémentant nos propres modèles.

Enfin, afin de visualiser la simulation, on utilise Blender qui permet de représenter graphiquement les robots, le bateau et l'environnement simulés en MORSE. MORSE et Blender permettent de simuler le système dans un environnement précis. Mais dans notre application, seuls les sous-ensembles définis dans l'environnement MOOS seront à la fin réellement implantés dans les vrais robots.

1.2.2 Interfaçage des outils

La relation entre les trois outils est schématisée ci-dessous :

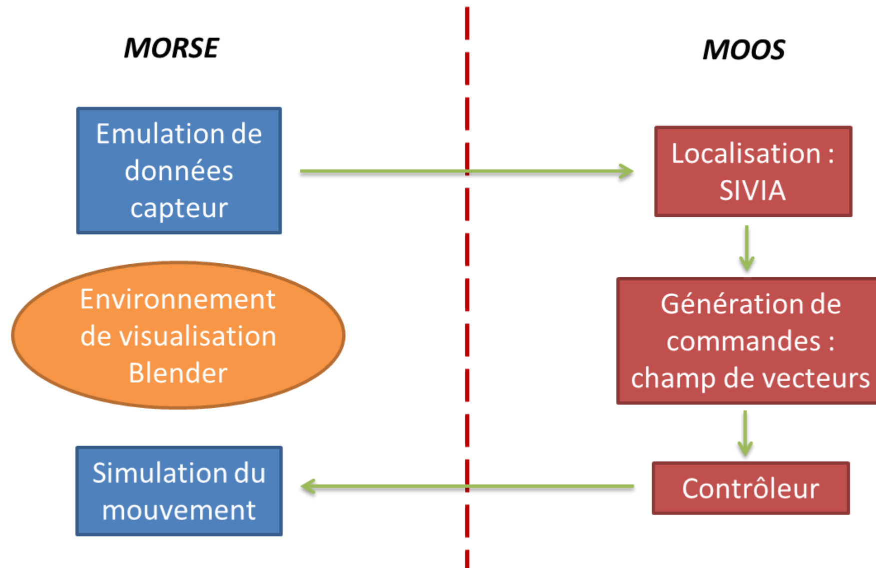

Figure 3 : relation entre MOOS, MORSE et Blender

1.3 Conventions de repère

Ségolène Pommier, Thibault Viravau

Les repères choisis pour le projet sont présentés sur « *Figure 4 : Représentation du robot, du repère absolu et de l'effet des actionneurs* », il est à noter que l'axe vertical est orienté positif vers le bas ce qui entraine une inversion des signes des angles de tangage et de cap.

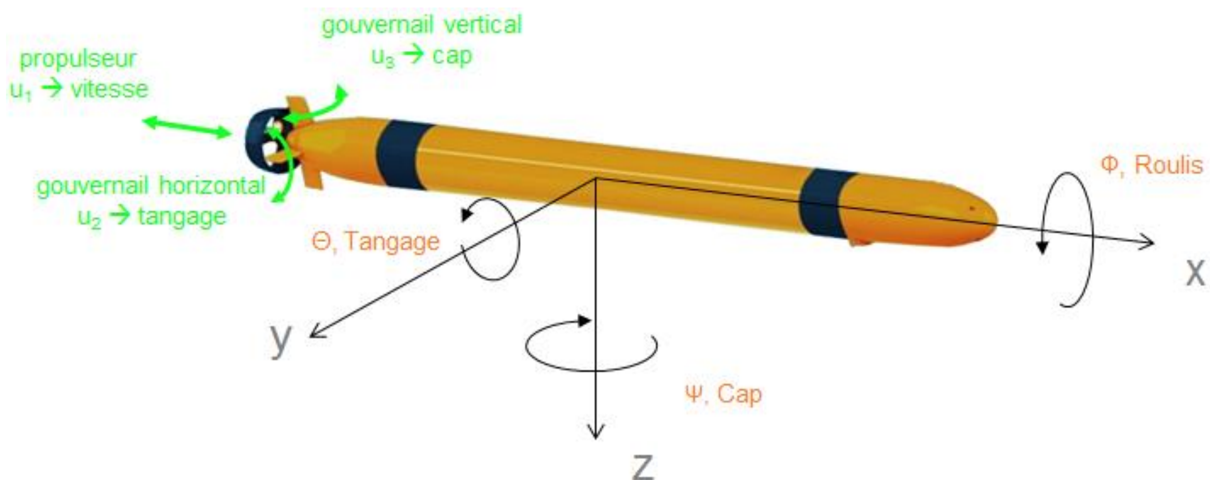

Figure 4 : Représentation du robot, du repère absolu et de l'effet des actionneurs

1.4 L'AUV

1.4.1 Actionneurs

L'AUV (Autonomous Underwater Vehicle) Scout est un robot sous-marin à forme de torpille, il dispose de trois actionneurs :

- Le propulseur placé à l'arrière du véhicule agissant sur l'accélération
- Une paire de gouvernails horizontaux agissant sur la plongée
- Une paire de gouvernails verticaux agissant sur le cap

1.4.2 Les équations de mouvement

Les équations qui régissent le lien entre l'état du robot et l'ordre donné sur ses actionneurs (u_1, u_2 et u_3) sont les suivantes :

Position

$$dX = v * cos(\Psi) * cos(\Theta) * dt$$
$$dY = v * sin(\Psi) * cos(\Theta) * dt$$
$$dZ = -v * sin(\Theta) * dt$$

Vitesse

$$v = v + (\beta_v * u_1 - \alpha_v * |v| * v) * dt$$

Attitude

$$d\Phi = -\left(\zeta_\Phi * \left(\eta_\Phi * sin(\Phi) + tan(\Theta) * v * (sin(\Phi) * u_2 + cos(\Phi) * u_3)\right)\right) * dt$$
$$d\Theta = \delta_\Theta * (cos(\Phi) * v * u_2 - sin(\Phi) * v * u_3) * dt$$
$$d\Psi = \gamma_\Psi * \left(\frac{sin(\Phi)}{cos(\Theta)} * v * u_2 + \frac{cos(\Phi)}{cos(\Theta)} * v * u_3\right) * dt$$

1.4.3 Configuration des équations de mouvement

Les paramètres $\alpha_v\ \beta_v\ \zeta_\Phi\ \eta_\Phi\ \delta_\Theta$ et γ_Ψ nous permettent d'adapter le modèle d'état au comportement réel du robot en modifiant par exemple son inertie, la poussée du propulseur, etc. Ainsi afin de régler ces paramètres, nous avons créé un programme de configuration en Python qui demande à l'utilisateur :

- Vitesse maximale
- Temps de demi-décélération
- Rayon de courbure en cap
- Rayon de courbure en plongée

A l'aide de ces constantes physiques inhérentes à la physique de l'AUV, le programme génère les paramètres constants du modèle d'état (en bleu ci-dessus). Il suffit alors d'effectuer un copier/coller pour les intégrer au programme.

2 Localisation par intervalles

Bertrand Moura

Pour résoudre le problème de localisation, on a choisi de s'affranchir d'une méthode classique probabiliste, type estimation de paramètres par filtre de Kalman, en choisissant à la place une approche de calculs par intervalles, qui garantit des solutions dans un intervalle d'incertitude donné.

2.1 Théorie des intervalles

Dans la théorie des intervalles, on remplace une valeur connue à une incertitude près par un intervalle dans lequel on est certain que cette mesure se trouve. Ainsi une valeur mesurée x connue avec une incertitude ε sera remplacée par l'intervalle $[x-\varepsilon, x+\varepsilon]$. L'objectif visé est de garantir un résultat recherché plutôt d'avoir un résultat avec un pourcentage de certitude donné. Les méthodes numériques utilisant le calcul par intervalle produisent des résultats fiables. En deux dimensions, les intervalles deviennent des boîtes. Toute une arithmétique autour calcul par intervalles est développée et permet de résoudre les problèmes linéaires. La résolution de ces problèmes se base essentiellement sur l'itération des calculs et sur le principe des contracteurs.

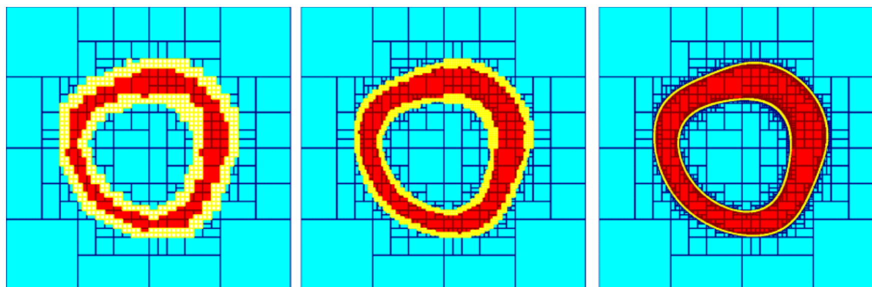

Figure 5 : résolution de $\boldsymbol{x}^2 = \boldsymbol{0}$ avec le calcul par intervalles

Si une boîte encadre une solution, on peut la couper en deux et réitérer les calculs de résolution du problème sur les deux boîtes ce qui peux permettre de discriminer une des deux boites et de réduire ainsi l'ensemble de solutions. Cependant cette opération est coûteuse (on augmente le nombres de boîtes de 2^n, donc autant d'itérations à faire ensuite sur ces boîtes). Il est donc intéressant d'affiner une boîte solution avant d'envisager de la bissecter. Cela ce fait grâce à l'opération de contraction qui permet de réduire une boîte solution donnée aux intervalles qui incluent strictement la solution.

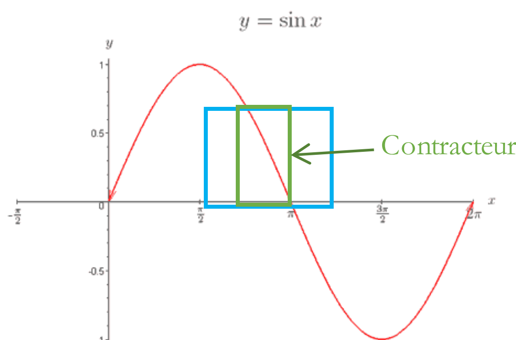

Figure 6 : Exemple de contracteur

2.2 Algorithme Sivia

Ricciardelli Gianluca, Guillemot Cyrill, Fadlane Maxime

Dans notre application, on définit un intervalle de distance entre le scout et les autres acteurs. Cet intervalle permet de déterminer un espace de localisation pour le scout par rapport au bateau. Logiquement, au temps $t = 0$ cet ensemble est un anneau car aucune information supplémentaire ne permet de déterminer dans quelle région se trouve le scout par rapport au bateau.

Le principe de l'algorithme Sivia est de localiser le scout au moyen du calcul par intervalle, en contractant les boites pour réduire l'espace des solutions. L'opération de contraction permet d'affiner la solution en prenant en compte les solutions précédentes. Les contracteurs peuvent être extérieurs ou intérieurs – si l'équation à résoudre est celle d'une surface –. C'est grâce à la connaissance de la localisation du scout à des temps antérieurs que l'on peut, en fonction de la variation des mesures, discriminer certaines positions dans l'anneau initial. On comprend donc qu'une singularité apparaît dans le cas où le bateau et les scouts avancent tous dans la même direction et à la même vitesse pendant une longue durée car l'information antérieur ne permet plus de contracter les solutions et aucune variation de mesure ne permet de discriminer des zones dans la boite solution.

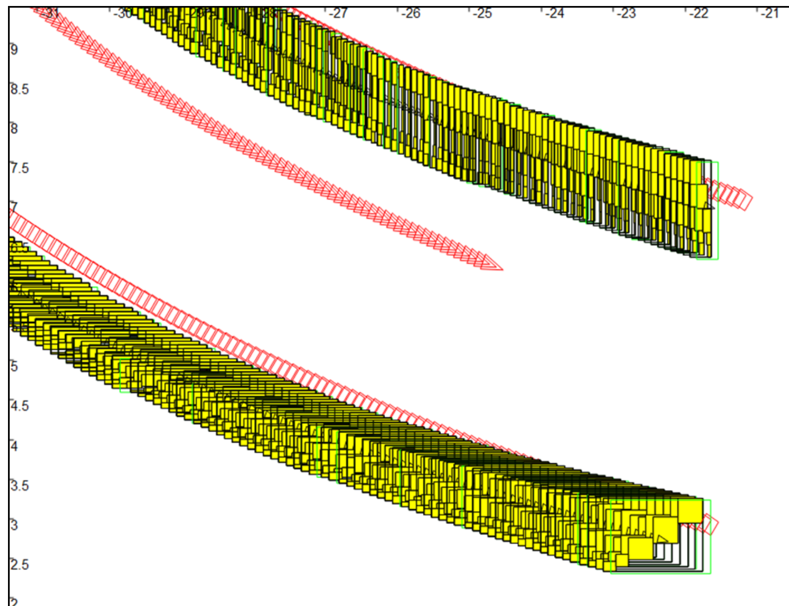

Figure 7 : Localisation avec l'algorithme Sivia sur de nombreux échantillons de temps

2.3 Simplification de la bibliothèque dans MOOS

Pour implémenter l'algorithme de localisation en C++, nous avons initialement utilisé la bibliothèque IBEX qui permet de réaliser les opérations de calcul par intervalles. Cependant, elle est très exhaustive et donc trop complète pour nos besoins ; de plus, elle est propriétaire. Nous avons donc décidé d'écrire notre propre bibliothèque simplifiée, adaptée au problème et compréhensible. Pour cela, nous n'avons gardé que les fonctions nécessaires à notre application ce qui permet d'alléger le poids de la bibliothèque et aussi de gagner en rapidité pour l'exécution sous MOOS.

3 Commande par champ de vecteurs

3.1 Commande

Bellaïche Adrien, Moura Bertrand

Pour commander les robots en fonction de la consigne, nous avons choisi une commande par champ de vecteurs.

La commande par champ de vecteur simple n'est pas suffisante. En effet cette commande n'est pas prédictive et elle n'anticipe donc pas le déplacement de la consigne lorsque le bateau change de cap. Pour améliorer la commande, on a donc dans un deuxième temps pris en compte la vitesse de rotation du bateau avec la formule de Varignon. En effet, si désormais on tient compte du fait que le bateau à une vitesse de rotation, alors on peut calculer un vecteur consigne qui prédit la position de la prochaine consigne (avec une erreur) et donc qui diminue fortement les erreurs de position. Dans la vidéo ci-dessous, on vérifie que l'estimation est efficace et que l'erreur de position est fortement réduite :

Figure 8 : commande des scouts (vidéo[1])

[1] https://www.youtube.com/watch?v=kTMAgzo6U6g

3.2 Conversion en commandes unitaires

Ségolène Pommier - Thibault Viravau

Suite à un problème d'interface entre les données en sortie du correcteur et les données d'entrées du modèle d'état, nous avons dû effectuer une conversion des données reçues (rôle du Contrôleur). Ainsi, les données DESIRED_THRUST, DESIRED_HRUDDER et DESIRED_VRUDDER issues du correcteur sont en réalité : la valeur de consigne de vitesse m/s, la consigne de cap en radian, et l'erreur de profondeur en m.

Ces valeurs nécessitent une conversion pour être comprises entre -1 et 1. Ces valeurs représentent la poussée maximale pour le moteur, ainsi que les butées pour l'angle des gouvernails. Pour cela, nous utilisons la fonction arc-tangente pour borner les valeurs de u_1 et u_2 car la fonction $\tan^{-1}$ est strictement croissante, à valeur dans [-1, 1] lorsqu'elle est normalisée (cf. « *Figure 9 : Courbe de la fonction arc-tangente* »). Pour u_3, nous utilisons la fonction dents de scie car il faut une fonction 2π-périodique selon err_{Ψ}, l'erreur de cap, et qui soit similaire à la fonction identité (cf. « *Figure 10 : Courbe de la fonction dents de scie* »).

$$u_1 = \frac{2}{\pi}\tan^{-1}(err_v)$$
$$u_2 = \frac{2}{\pi}\tan^{-1}(err_\theta)$$
$$u_3 = -\frac{2}{\pi}\tan^{-1}(tan(\frac{err_\Psi}{2}))$$

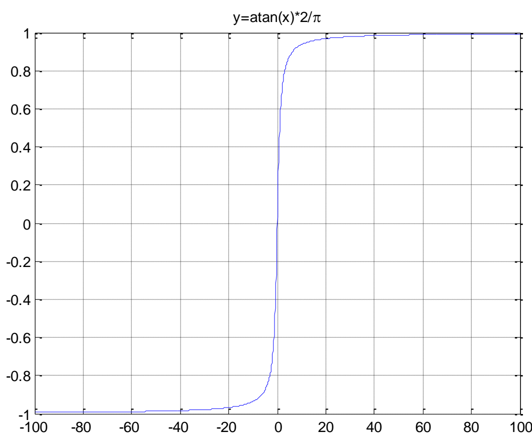

Figure 9 : Courbe de la fonction arc-tangente

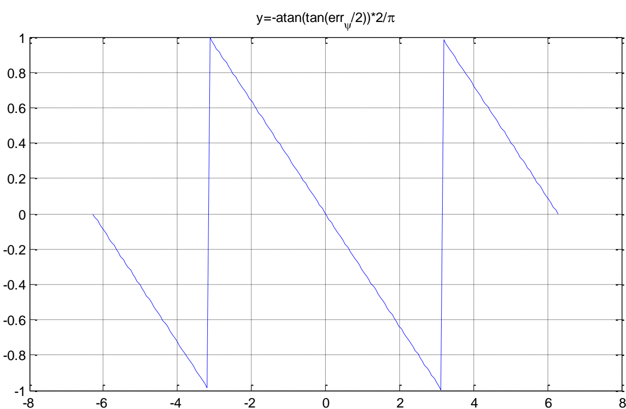

Figure 10 : Courbe de la fonction dents de scie

4 Démonstrations

Bertrand Moura

Avant de simuler les scouts avec MORSE et Blender, nous avons validé l'algorithme de localisation ainsi que les lois de commande en les simulant directement en C++ (sous QtCreator) sans prendre en compte les modèles physique des capteurs, senseurs, milieu extérieur, etc.

4.1 Localisation par intervalles et singularités

4.1.1 Singularité en ligne droite

Certains cas de fonctionnement peuvent poser problème dans la localisation des scouts. C'est le cas lorsque le bateau et les robots avances tous ensemble en ligne droite à vitesse constante. Dans cette configuration-là, une singularité se produit : le scout n'ayant aucune information de variations de distance où d'orientation par rapport au bateau, il peut se trouver à n'importe quel endroit sur un cercle autour du bateau. En réalité, comme on utilise l'information de localisation donnée aux temps antérieurs, l'incertitude n'est pas aussi grande au début. Mais si cette configuration dure dans le temps, la zone d'incertitude va s'agrandir à chaque pas de temps jusqu'à reconstituer complètement un anneau d'incertitude. C'est une configuration singulière concernant la localisation qui peut causer problème si elle n'est pas rompue.

Figure 11 : Problème de localisation en ligne droite (vidéo[2])

[2] https://www.youtube.com/watch?v=8V4U1kPslOc

4.1.2 Rompre la singularité

Afin de retrouver une situation non singulière, il faut rompre la similarité de trajectoire. Un moyen consiste à faire faire une boucle à l'un des deux scouts. Cela créera une variation de distance et de vitesse entre le scout et le bateau qui permettra de déterminer sa position de façon précise. De plus, comme les deux scouts communiquent entre eux, l'information de position plus précise du scout qui fait la boucle permettra au deuxième scout de se localiser lui aussi de façon précise. Ainsi, il suffit de modifier une trajectoire pour rétablir la précision pour les deux scouts.

Figure 12: Singularité rompue (vidéo[3])

Le fonctionnement en ligne droite est cependant un cas normal et fréquent d'utilisation. C'est d'ailleurs dans cette configuration de fonctionnement que RTSys souhaitait utiliser cette application. Faire une boucle devient donc potentiellement trop extrême par rapport à un objectif de maintenir une ligne droite.

Il existe d'autres manières de rompre cette singularité qui permettrait de limiter les ruptures de trajectoire. Par exemple, on pourrait imaginer qu'à partir d'un certain seuil d'incertitude de localisation, au lieu d'effectuer une boucle, un des deux scout accélèrerait de façon soudaine, ou bien aurait une trajectoire sinusoïdale d'amplitude faible pendant un cours instant. Le but étant toujours de pouvoir briser la similarité de direction et de vitesse entre les scouts et le bateau.

En introduisant un troisième scout, on pourrait lui attribuer comme seule fonction de réaliser des tours autour des deux premiers scouts ce qui permettrait d'avoir une information de localisation toujours précise.

[3] https://www.youtube.com/watch?v=MQhBzluYpeU

5 Utilisation du Middleware MOOS

5.1 Architecture MOOS

Louise Devigne, Thomas Le Mézo

Le logiciel MOOS (Mission Oriented Operating Suite) nous a permis de séparer les applications afin de mieux répartir le travail au sein de l'équipe selon le schéma suivant :

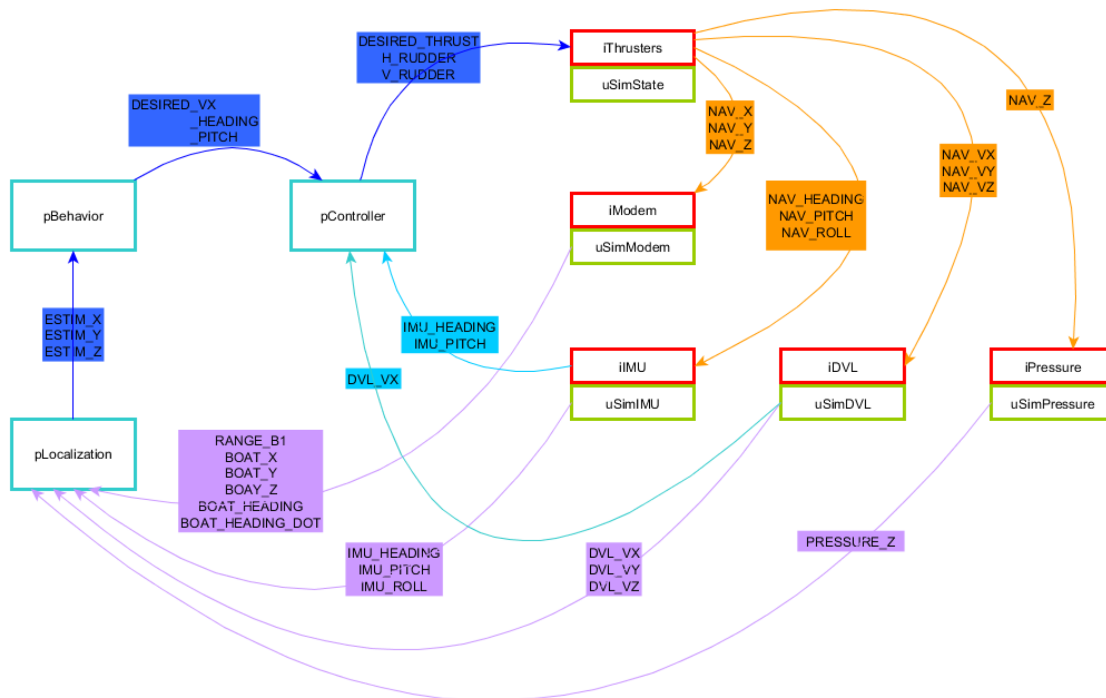

Figure 13 : Schéma de la répartition du travail en application MOOS

Chacun des blocs correspond à une application MOOS et a été codé par un élève(ou un binôme). Les blocs rouges (i = interface) font la connexion entre la communauté et les capteurs. L'idée a été de remplacer ces blocs par des blocs de simulation (blocs verts) afin de pouvoir représenter les capteurs.

Pour les blocs de simulation le travail a été de récupérer des données et de les bruiter afin de reproduire le comportement qu'auraient les capteurs.

Ensuite, les blocs bleus représentent les blocs de plus bas niveau (calculs, localisation …), ils sont au nombre de 3 :
- pLocalization : localisation des Scouts en utilisant SIVIA
- pBehavior : implémentation des champs de vecteurs
- pController : régulation en vitesse et en cap

Pour ces blocs, le travail a été d'adapter le code qui a été au préalable implémenté en C++ dans l'environnement de développement Qt en utilisant IBEX. La partie suivante illustre le concept des modules en présentant ***uSimState***.

5.2 uSimState : Implémentation des équations d'état

Ségolène Pommier - Thibault Viravau

5.2.1 Rôle

Le module ***uSimState*** de MOOS incarnait la réalité physique, au sein de la simulation virtuelle. Il est fait pour être interfacé avec les modules qui simulent les capteurs (***uSimIMU***, ***uSimDVL***, ...) afin de leur donner la « réalité » qu'ils doivent brouiller et restituer avec imprécision.

5.2.2 Constitution

Ce module est constitué comme suit :

- *En entrée :* il prend les actions données aux contrôleurs (DESIRED_HRUDDER, DESIRED_VRUDDER, DESIRED_THRUST)
- *En mémoire :* il garde l'état actuel du robot
- *En continu :* il intègre les équations d'état et met à jour l'état du robot
- *En sortie :* il donne les différentes variables de l'état : position, attitude, vitesse

5.2.3 Tests

Afin de vérifier que uSimState fonctionnait correctement – notamment en termes de signes concernant les commandes –, nous avons créé un fichier carnet de bord « log.txt » dans lequel étaient enregistrés les états successifs du robot ainsi que ses commandes, de façon horodatée. Ce fichier était importé dans MATLAB qui nous permettait de visualiser les courbes des états en fonction du temps ainsi que les constantes de temps relatives aux commandes.

Les conditions de tests étaient les suivantes :

- Test en ligne droite avec commande propulseur constante maximale ($u_1=1$)
- Test à vitesse constante avec commande gouvernail (H ou V) constante maximale (u_2, u_3 = ±1)
- Test avec commande propulseur constante non-nulle et commande gouvernails non-nulle

Malheureusement MATLAB n'est absolument pas adapté à la visualisation de solides en 3 dimensions, comme le montre « *Figure 14 : Représentation avec MATLAB de l'état du robot* », il était donc peu intuitif de débugger notre programme à ce stade d'avancement. C'est avec MORSE que nous avons pu disposer d'une visualisation efficace.

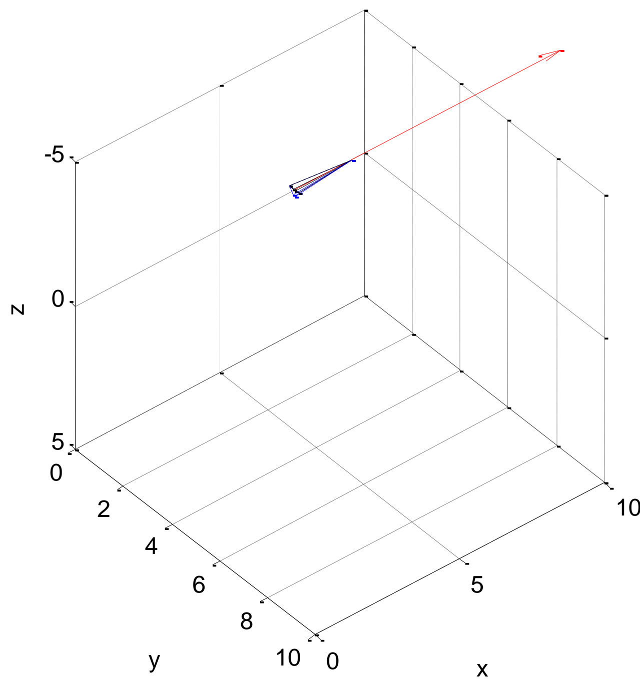

Figure 14 : Représentation avec MATLAB de l'état du robot

6 Interface Homme-Machine

6.1 Interface graphique du simulateur : MORSE/BLENDER

Ségolène Pommier – Thibault Viravau – Thomas Le Mézo

6.1.1 Présentation de MORSE

MORSE est un moteur de simulation et de visualisation 3D écrit en Python, basé sur progiciel d'infographie open-source BLENDER. Il peut être utilisé tant pour faire des jeux vidéo que des simulations physiques comme le montre « *Figure 15 : Deux illustrations des usages possibles de MORSE* ».

Figure 15 : Deux illustrations des usages possibles de MORSE

6.1.2 Implémentation des équations d'état et visualisation

MORSE intègre un moteur physique, capable de gérer les interactions entre les objets soumis à des forces diverses et il offre aussi la possibilité d'entrer nos propres équations d'état. C'est ici ce que nous avons fait : MORSE devenait la « réalité », simulant la position des objets, leurs mouvements (connaissant les ordres sur les actionneurs) et proposant d'afficher à l'écran une représentation du monde 3D simulé, comme présenté sur la « *Figure 16 : Capture d'écran de la fenêtre BLENDER/MORSE lors d'une simulation* ». Sur cette image les sous-marins (en jaune) sont en phase de plongée et le bateau (en bleu) avance lentement. Les modèles « .blend » sont ceux fournis par notre encadrant à titre de démonstration.

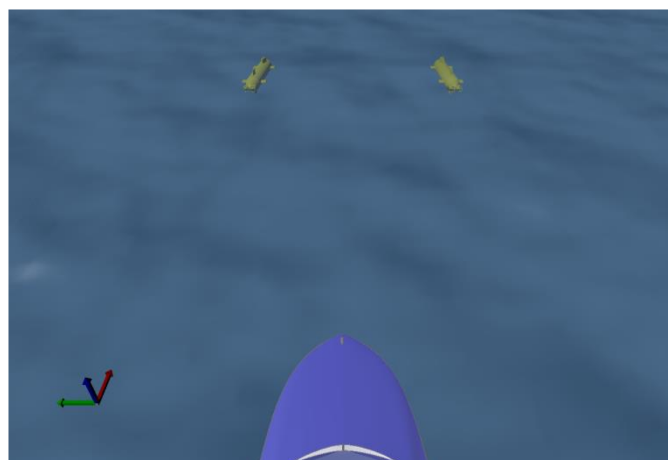

Figure 16 : Capture d'écran de la fenêtre BLENDER/MORSE lors d'une simulation

6.1.3 Tests

La partie BLENDER intégrée à MORSE en fait, par essence, un moteur de visualisation autant que de simulation. Ici et grâce aux modèles « .blend » fournis par notre encadrant, cette polyvalence nous a permis de visualiser très simplement le comportement des robots dont nous souhaitions tester le modèle d'état ; en particulier sur les scénarii évoqués précédemment : commande en propulsion maximale, commande en virage, commande en plongée, …

Le résultat visuel de deux de ces tests est visible sur les six figures suivantes.

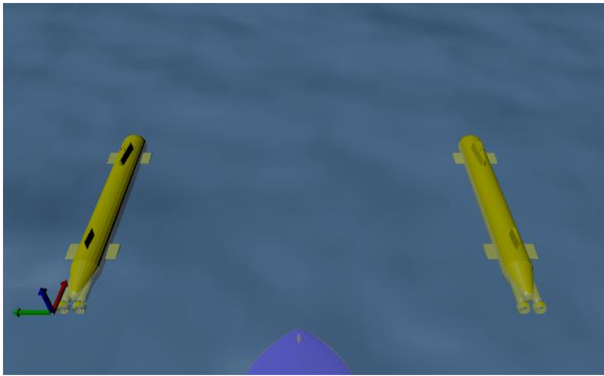

Figure 17 : Test d'accélération, image 1

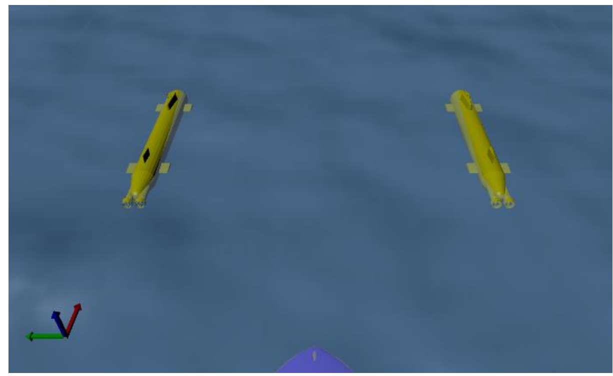

Figure 18 : Test de plongée, image 1

Figure 19 : Test d'accélération, image 2

Figure 20 : Test de plongée, image 2

Figure 21 : Test d'accélération, image 3

Figure 22 : Test de plongée, image 3

6.2 Interaction avec le logiciel

Théo Blanchard – Bastien Sultan

Afin de fournir à l'utilisateur un mode de contrôle du simulateur intuitif, nous avons choisi d'utiliser un joystick commandant le déplacement du bateau.

Ayant essayé dans un premier temps d'implémenter la commande des objets 3D *via* le joystick directement dans BLENDER, nous avons constaté que le matériel utilisé n'était pas reconnu par le logiciel. A ces causes, il nous a semblé opportun de développer un module extérieur, générique à tous les joysticks, permettant d'interagir avec la simulation sans passer par BLENDER.

Le moteur de simulation MORSE étant sous-tendu par Python, nous avons écrit un script permettant de récupérer les informations du joystick (axe sollicité / amplitude du mouvement) *via* le module *Pygame*. Un problème d'interfaçage est toutefois à signaler : MORSE s'appuyant sur la version 3.3 de Python – qui ne permettait pas, dans les conditions du test, d'importer convenablement le module *Pygame* – nous avons été contraints d'interfacer cette version de l'interpréteur avec sa version 2.7, qui ne pose aucun problème d'implémentation du module. Deux possibilités s'offraient à nous : nous pouvions mettre en place un socket TCP entre Python 3.3 et Python 2.7 – ce qui, potentiellement, pouvait être à la source d'erreurs interrompant le programme et s'avérait plus complexe que la solution alternative, que nous avons choisie. Elle consistait à écrire un script séparé des classes MORSE, interprété par Python 2.7, qui lit les données issues du joystick et les enregistre dans un fichier ; lequel est lu dans la classe MORSE régissant le mouvement du bateau. Les données ainsi récupérées sont donc ajoutées en « *offset* » aux équations d'état déjà implémentées.

Afin de laisser à l'utilisateur la possibilité d'utiliser ou non le module développé, nous avons créé un script *bash* (placé et exécutable dans le répertoire /*morse* du projet) encapsulant les opérations de lancement du module et de la simulation ; qui peut toujours être démarrée indépendamment *via* la commande classique *morse run scout*.

Conclusion

Le projet « Scout » mené à l'ENSTA Bretagne a permis de valider l'utilisation de la théorie des intervalles pour réaliser le scénario demandé par la société RTSys. Le maintien d'une configuration particulièrement instable des deux AUV devant se positionner devant le bateau n'a pu être obtenu qu'avec le calcul par intervalle. De plus, dans un contexte sous-marin, la rareté des communications et les problématiques de localisation ajoutent une difficulté supplémentaire au problème.

La réussite de ce projet a été permise par l'utilisation de plusieurs logiciels et intergiciels :

- MOOS : un intergiciel qui nous a permis de séparer chaque fonction du système en une « application » autonome communicante. Les tests unitaires sur chacune des applications a facilité une intégration rapide et modulaire du système final.
- MORSE : un logiciel de simulation permettant de simuler physiquement et de créer un rendu 3D de notre système.
- GIT : une plateforme de gestion de code qui nous a permis de travailler efficacement et de manière modulaire dans une équipe d'une dizaine de personnes.

Nous tenons finalement à remercier :

- RTSys pour nous avoir proposé de travailler sur le projet COMET,
- Luc Jaulin pour nous avoir encadrés tout au long de ce projet,
- Simon Rohou et Jeremy Nicola pour nous avoir enseigné l'utilisation de MOOS et MORSE et pour le temps passé à débugger nos codes.

Bibliographie

Pour l'exploration sous marine

V. Drevelle, L. Jaulin and B. Zerr (2013), Guaranteed Characterization of the Explored Space of a Mobile Robot by using Subpavings, NOLCOS 2013

C. Aubry, R. Desmare and L. Jaulin (2013). Loop detection of mobile robots using interval analysis. *Automatica*. vol. 49, Issue 1. pp 463-470.

Pour la localisation sous marine

M. S. Ibn Seddik, L. Jaulin and J. Grimdale (2014). Phase Based Localization for Underwater Vehicles Using Interval Analysis. *Mathematics in Computer Science, special issue on Interval methods and applications*, vol. 8, number 3,4, pages 495-502,

R. Neuland, J. Nicola, R. Maffei, L. Jaulin, E. Prestes and M. Kolberg (2014). Hybridization of Monte Carlo and Set-membership Methods for the Global Localization of Underwater Robots, IROS 2014, pages 199-204

J.Sliwka, F. Le Bars, O. Reynet, and L. Jaulin (2011). Using interval methods in the context of robust localization of underwater robots. In NAFIPS 2011, El Paso, USA.

Pour la localisation avec le calcul par intervalles

L. Jaulin, M. Kieffer, E Walter and D. Meizel (2002), Guaranteed robust nonlinear estimation, with application to robot localization, *IEEE Transactions on systems, man and cybernetics; Part C Applications and Reviews.* Volume 32, Number 4, pages 374-382

F. Le Bars, J. Sliwka, O. Reynet and L. Jaulin (2012). State estimation with fleeting data, *Automatica*, Vol. 48, number 2, pages 381-387.

L. Jaulin, E. Walter, O. Lévêque and D. Meizel (2000), Set inversion for chi-algorithms, with application to guaranteed robot localization, *Math. Comput. Simulation, 52, 197-210.*