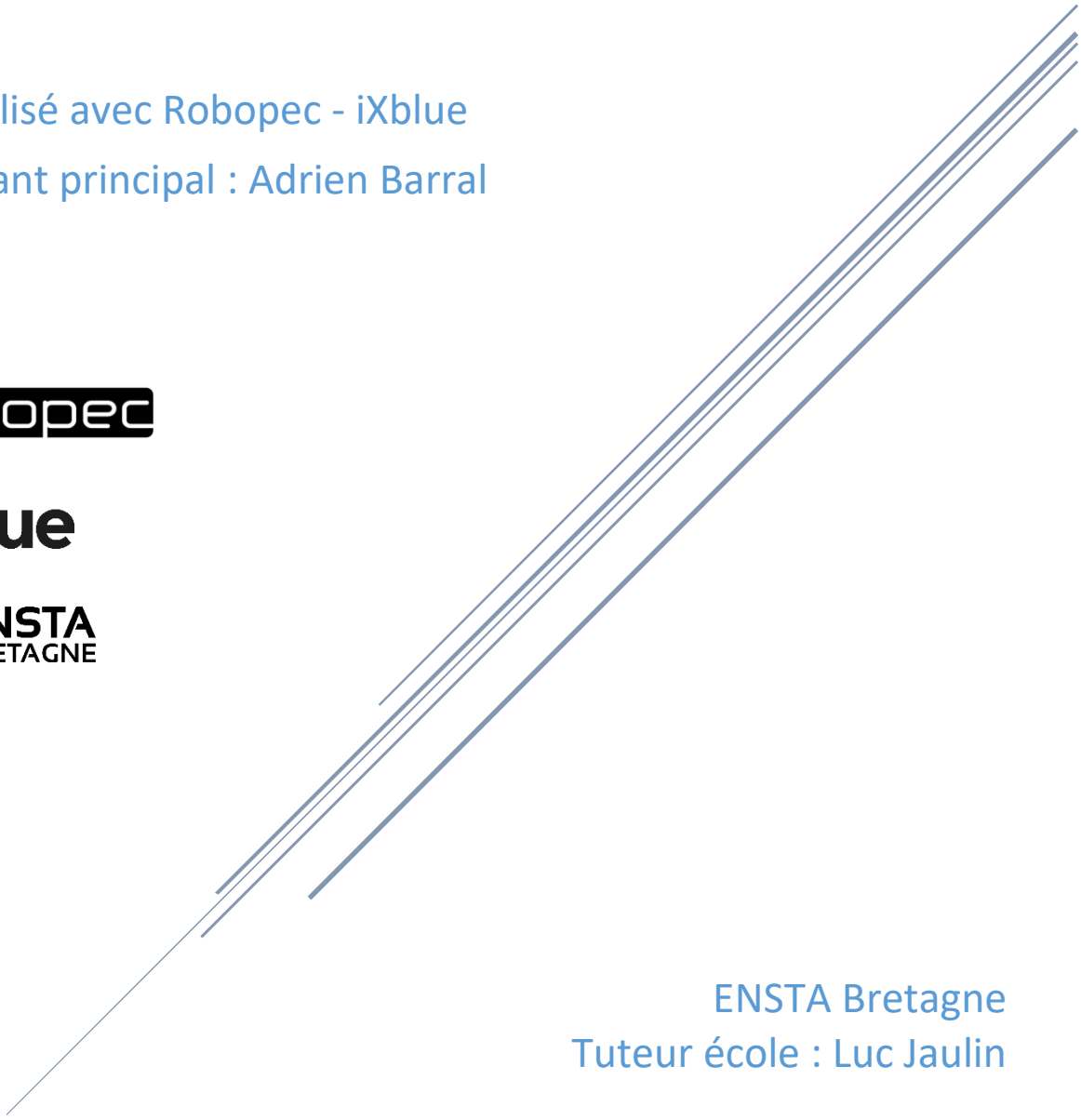
JULES BERHAULT

DEVELOPPEMENT ET OPTIMISATION D'UN SYSTEME DE DETECTION ET DE SUIVI D'OBSTACLES EN MER

PFE réalisé avec Robopec - iXblue

Encadrant principal : Adrien Barral



ENSTA Bretagne

Tuteur école : Luc Jaulin

Table des matières

Remerciements

Je tiens d'abord à remercier Adrien Barral pour avoir dirigé ce stage de fin d'étude. Son expérience et son calme m'ont toujours apporté le bon conseil au bon moment. Ses remarques éclairées ont participé à améliorer la précision de ce manuscrit. Auguste Bourgois et Simon Rohoux qui examinent mon travail, et participent à mon jury. Luc Jaulin pour son encadrement et son suivi de ce stage. Arnaud Pietrosemoli, pour sa collaboration fructueuse. Les membres de l'équipe Robopec : Maxime Faldhane, Robin Vanhouve, Bertrand Caré, Romain Reignier, Christophe Rousset, Guilhem Sals, Axel Duberc, Marine, Eric, Antoine, Leslie, Guillaume, Charlotte, Théo, Pierre et leur esprit d'équipe.

Introduction

Présentation du système

La navigation en mer nécessite une vigilance constante car la présence d'autres navires, d'obstacles fixes ou même d'OFNI (Objet Flottant Non Identifié) sur la trajectoire peut causer des dégâts importants en cas de collision. De nombreux systèmes marins d'évitement dynamique d'obstacles existent et continuent de se développer pour assurer la sécurité en mer. La plupart de ces systèmes fonctionnent sur la base de capteurs tels que les lidars, les radars, les sonars et même via l'exploitation des AIS.

Robopec, filiale d'iXblue, développe un système nommé MDT pour *Maritime Detection and Tracking* proposé sous la forme d'un boitier électronique s'interfaçant à différents capteurs cités précédemment et fusionnant les mesures de chacun afin de fournir une connaissance précise de l'environnement du navire. Le système propose par la même occasion un algorithme d'évitement d'obstacle dynamique renvoyant une trajectoire sûre générée à partir d'une mission programmée. Mais la subtilité qui démarque MDT des systèmes existant est le fait qu'il exploite une caméra infra-rouge pour détecter et localiser les obstacles en mer. L'avantage majeur de ce capteur est sa portée bien supérieure au lidar et au sonar.

Ce système de détection et de localisation via caméra infra-rouge est une nouveauté dans le domaine. Son avantage premier est sa portée longue distance qui surpasse les autres capteurs, ce qui est très utile pour un tel problème d'évitement d'obstacles. Le principal défaut est la précision de la mesure des cibles à longue distance qui semble très perturbée par les condition marines. La caméra infra-rouge étant perturbée par la houle, les images qu'il fournit varie beaucoup de l'une à l'autre et il est difficile pour le système de s'adapter. Ajoutons à cela que le système est largement dépendant de ce que la caméra observe, tout comme l'œil humain.

L'objectif du projet

Le système MDT est bien souvent confronté à un environnement imparfait et doit fonctionner au milieu d'une mer agitée avec une ligne d'horizon pas toujours visible lorsqu'elle est couverte par les côtes. Dans ces conditions, la mesure fournit par le système de localisation des cibles via caméra IR est fortement perturbé et perturbe à son tour la mesure déduite par le filtre de Kalman utilisé pour effectuer une fusion de capteur. Le calcul de projection qui est réaliser pour localiser les cibles depuis une image de la caméra IR se base sur, d'une part, l'état de l'observateur (position, orientation) et d'autre part, la ligne d'horizon qui joue un rôle de référence dans la projection des cibles. L'estimation de ces deux derniers paramètres peuvent gagner davantage en précision en utilisant les valeurs retournées par de nouveaux capteurs embarqués sur le système, ce qui impacterait logiquement la mesure de la distance des cibles observées. Reste à savoir si l'implémentation de ces paramètres supplémentaire, qui nous rapprocheraient de la réalité, impacteront positivement ou négativement la mesure des cibles.

Dans un premier temps, nous étudierons les phénomènes susceptibles d'impacter la localisation des cibles et nous analyseront leur influence sur la mesure. Ensuite nous chercherons à prendre en compte un mouvement supplémentaire dans le modèle dynamique du système afin d'estimer l'effet sur la localisation des cibles. Enfin nous trouverons un moyen de rectifier l'erreur causée par le quiproquo fait par le système de détection d'horizon avec le trait de côte dans le but d'améliorer, encore, la mesure.

Détection et localisation d'obstacle en mer

Principe du système

Pour détecter les obstacles en mer, le système MDT s'appuie en partie sur les images de la caméra infrarouge. L'image est ainsi traitée par le nœud ROS, nommé *p_detect_ir_cnn*, à travers un réseau de neurones à convolution (CNN) entrainé pour détecter et localiser des éléments caractéristiques dans l'environnement marin.

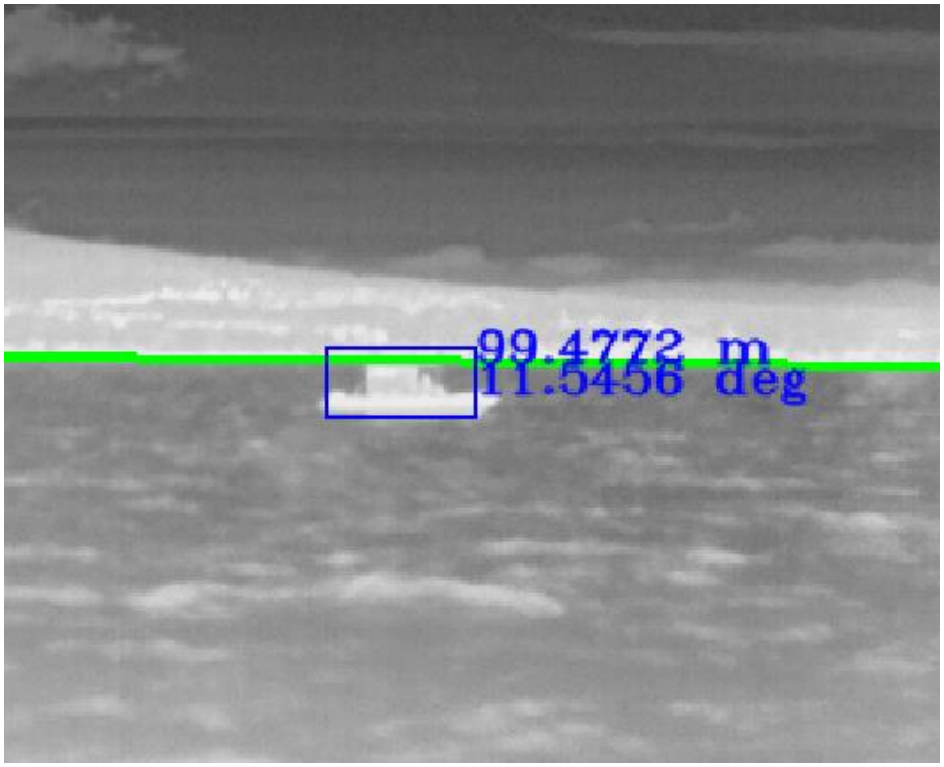


FIGURE 1 : REPRESENTATION D'UNE DETECTION OBSERVEE PAR LA CAMERA IR

Une fois ces éléments localisés sur l'image et délimités par une boite (*bounding box*), le nœud va ensuite, à partir de cette information, estimer la distance et l'azimut de ces éléments sur une carte grâce à une méthode géométrique. Cette méthode exploite l'angle formé par l'objet et l'horizon [1] avec la caméra infrarouge γ et la hauteur de celle-ci au-dessus de la surface de l'eau pour estimer la distance d'une cible sur la surface de l'eau.

Cet angle est calculé à partir du nombre de pixel entre l'horizon et la cible avec l'utilisation des paramètres caméra comme décrit dans la section suivante.

L'azimut de l'objet est estimé de la même manière et couplé avec la donnée de distance elles permettent de situer l'objet par rapport au navire. Connaissant la position et le cap du navire, le nœud *p_detect_ir_cnn* est en mesure de renvoyer la liste des positions de ces obstacles afin qu'ils soient affichés sur l'interface utilisateur.

Calcul de la distance d'un obstacle en mer avec l'aide de l'horizon

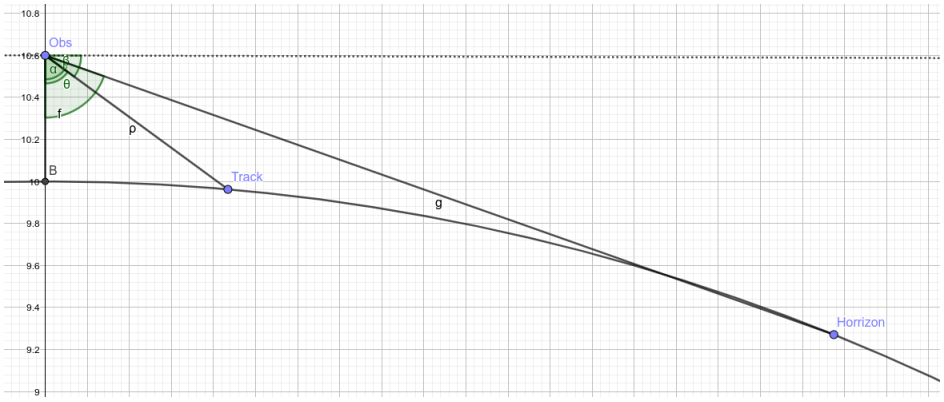


FIGURE 2 : REPRESENTATION SCHEMATIQUE D'UNE VUE EN COUPE D'UNE SITUATION DE DETECTION D'UNE CIBLE SUR L'EAU

- g : distance euclidienne de l'horizon
- ρ : distance euclidienne de la cible (track)
- θ : angle nadir - observateur - horizon
- α : angle nadir - observateur - cible
- β : élévation de la cible (track)
- f : hauteur de l'observateur

On cherche l'angle $\gamma = \theta - \alpha$, l'angle entre l'horizon et la cible (Track) avec l'observateur.

Détermination de l'élévation de l'horizon

Dans un premier temps on cherche à déterminer θ

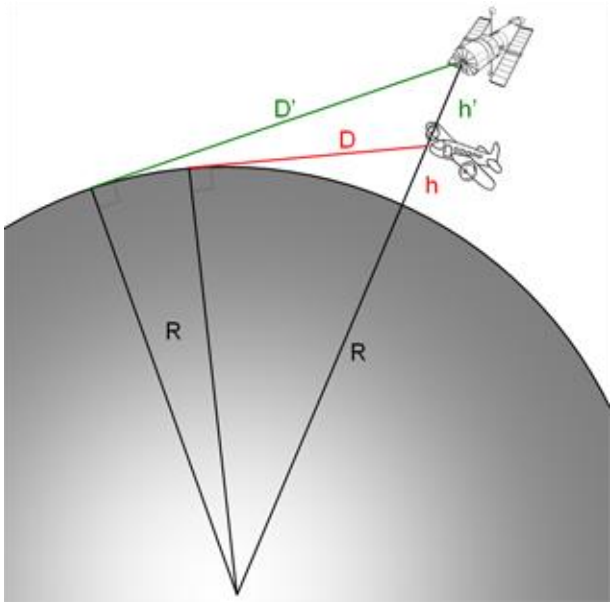


FIGURE 3 : SCHEMA DU TRIANGLE RECTANGLE FORME PAR LE SEGMENT D'HORIZON TANGENT A LA SURFACE TERRESTRE

Soit h la hauteur de la caméra, et R le rayon de la terre, si on suppose la Terre parfaitement ronde, on peut former un triangle rectangle composé de la ligne d'horizon [1] et du segment qui va du centre de la Terre à l'observateur $R+h$ en tant qu'hypoténuse. Sachant que la ligne d'horizon est une ligne tangente du cercle représentant la Terre au niveau du point d'intersection, on a un angle droit en ce sommet. Ce qui peut nous permettre de déduire la distance de l'horizon D grâce aux données du problème :

$$D^2 = (\mathrm{R} + \mathrm{h})^2 + \mathrm{R}^2$$

$$\Leftrightarrow \mathrm{D} = \sqrt{2\mathrm{Rh}}$$

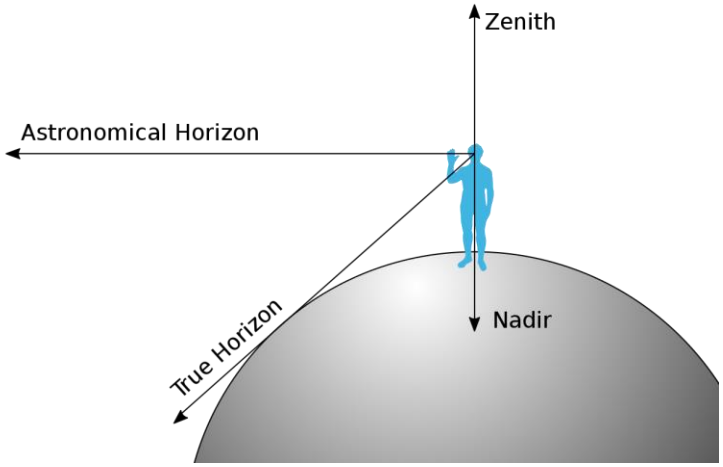


FIGURE 4 : REPRESENTATION DE L'HORIZON ASTRONOMIQUE ET L'HORIZON VRAI

On déduit simplement et par la même manière, θ, l'angle entre le nadir et l'horizon visible :

$$\tan(\theta) = \frac{R}{\sqrt{2Rh}}$$

$$\Leftrightarrow \theta = \tan^{-1}\left(\frac{R}{\sqrt{2Rh}}\right)$$

Détermination de l'élévation de la cible

Puisqu'on sait comment obtenir θ simplement à partir de la hauteur de l'observateur, on va chercher $b = -\beta$, et on en déduira α sachant que :

$$\alpha = \frac{\pi}{2} - \beta = \frac{\pi}{2} + b$$

La cible (Track) est à une distance ρ de l'observateur à une hauteur nulle (sur la surface de l'eau)

Les coordonnées de la cible si on l'exprime par rapport au centre de la terre sont :

$$x = \rho\cos(b)$$
$$y = (R + h) + \rho\sin(b)$$

avec b l'élévation de la cible.

Comme l'objet est sur la terre, on sait aussi que : $x^2 + y^2 = R^2$.

En substituant x et y, on a :

$$\rho^2 \cos^2(b) + \left((R + h) + \rho \sin(b)\right)^2 = R^2$$

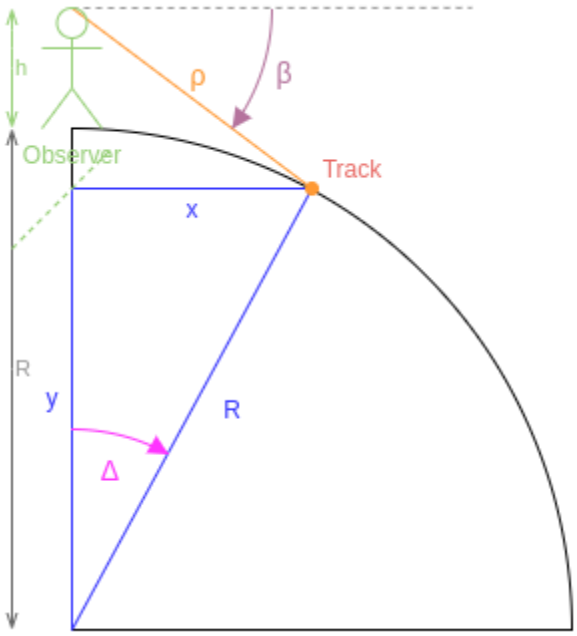


FIGURE 5 : REPRESENTATION DU TRIANGLE RECTANGLE FORME PAR UNE CIBLE SUR LA SURFACE TERRESTRE ET L'OBSERVATEUR

On nomme Δ, l'angle formé par l'observateur et la cible avec le centre de la Terre. De cet angle, on obtient un triangle rectangle d'hypoténuse R correspondant au rayon terrestre. On en déduit les égalités suivantes :

$$x = R \sin(\Delta) = \rho \cos(b)$$

$$y = R \cos(\Delta) = (R + h) + \rho \sin(b)$$

L'introduction de la variable Δ, va nous permettre de poser le problème sous la forme d'une équation du second ordre.

On pose :

$$X = \sin(\Delta)$$

On a alors :

$$x = RX$$
$$y = (R + h) + R \tan(b) X$$

En développant, on obtient une équation du second ordre avec X en tant qu'inconnue, b étant l'élévation de la cible déduite depuis l'image caméra :

$$\begin{aligned} & x^2 + y^2 = R^2 \\ \Leftrightarrow\ & (RX)^2 + ((R+h) + R\tan(b)X)^2 = R^2 \\ \Leftrightarrow\ & R^2X^2 + R^2\tan^2(b)X^2 - 2R(h+R)\,tan(b)X - R^2 + (h+R)^2 = 0 \\ \Leftrightarrow\ & (1 + tan^2(b))X^2 - 2\tfrac{h+R}{R}\tan(b)X - 1 + (\tfrac{h+R}{R})^2 = 0 \end{aligned}$$

soit,

$$\begin{aligned} & aX^2 + bX + c = 0 \\ & a = 1 + tan^2(b) \quad b = -2(\tfrac{h+R}{R})\tan(b) \quad c = -1 + (\tfrac{h+R}{R})^2 \end{aligned}$$

Avec l'élévation b de la cible, qui dépend de l'angle γ (et de l'angle θ calculé grâce à la hauteur de l'observateur) :

$$b = -\beta = \alpha - \frac{\pi}{2} = \theta - \gamma - \frac{\pi}{2}$$

Enfin, pour déduire γ, l'angle entre l'horizon et la cible, on traite les images numériquement afin de mesurer le nombre de pixels qui séparent l'horizon de la cible.

On a donc :

$$\gamma = \frac{\Delta_v V_{fov}}{V_{pix}}$$

On utilise les paramètres de la caméra avec Δ_v, l'écart en pixel sur l'image entre la cible et l'horizon, V_{fov}, les angles FOV de la caméra et V_{pix}, les dimensions en pixel de l'image.

Etude des causes de l'erreur de détection par caméra IR et quantification de leur impact sur la mesure

Pour détecter les obstacles en mer à l'aide de la caméra infrarouge, il fait dans un premier temps détecter et localiser des éléments caractéristiques sur les images de la caméra infrarouge.

Une fois ces éléments localisés sur l'image et délimités par une boite (*bounding box*), le nœud va ensuite, à partir de cette information, estimer la distance et l'azimut de ces éléments sur une carte grâce à une méthode géométrique. Cette méthode exploite l'angle formé par l'objet et l'horizon [1] avec la caméra infrarouge et la hauteur de celle-ci au-dessus de la surface de l'eau pour estimer la distance de la cible.

Cet angle est calculé à partir du nombre de pixel entre l'horizon et l'objet comme décrit dans la section suivante.

L'objectif de cette étude est de mettre en valeur l'impact des erreurs de détection sur la mesure de distance du système de détection infra-rouge MDT.

La réalisation d'un programme python traduisant les expressions mathématiques de la théorie du système MDT en fonctions simples à permit de constituer un modèle mathématique de la relation entre l'entrée et la sortie. A l'aide des outils *matplotlib* il est ainsi possible de visualiser la mesure de la distance d'une cible en fonction des paramètres d'entrée et des coordonnées de la cible observée sur l'image de la caméra IR.

Une série de graphiques expose les évolutions de la mesure de distance en fonction des variations dans la détection, notamment autour de la segmentation de l'horizon ou de l'encadrement de la cible.

Configuration du système étudié

Elément	**Paramètre**	**Valeur**	**Unité**
Environnement	Hauteur observateur	2.0	m
	Hauteur cible	0.0	m
	Rayon terrestre	6378137.0	m
Caméra	Angle de tangage caméra (assiette)	0.0	deg
	Angle de roulis caméra (inclinaison)	0.0	deg
	FOV horizontal caméra (HFOV)	50.0	deg
	FOV vertical caméra (VFOV)	40.0	deg
	Résolution caméra	640 × 512	pixel
	Echelle distance focale caméra	1.0	

Evolution de la distance mesurée par rapport à la position en pixel de la cible sur l'image

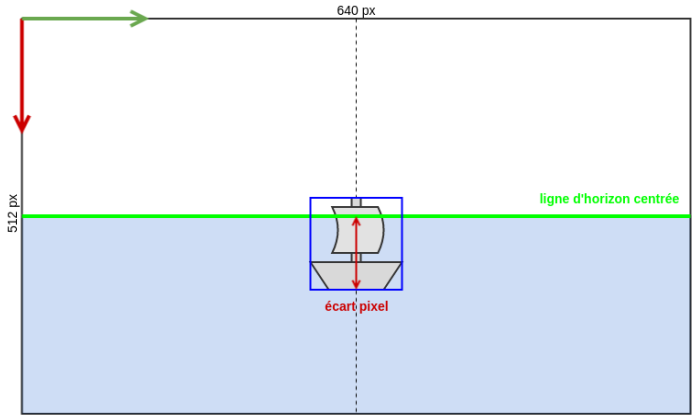


FIGURE 6 : REPRESENTATION D'UNE SITUATION SIMPLE DE DETECTION D'UNE CIBLE SANS ERREUR

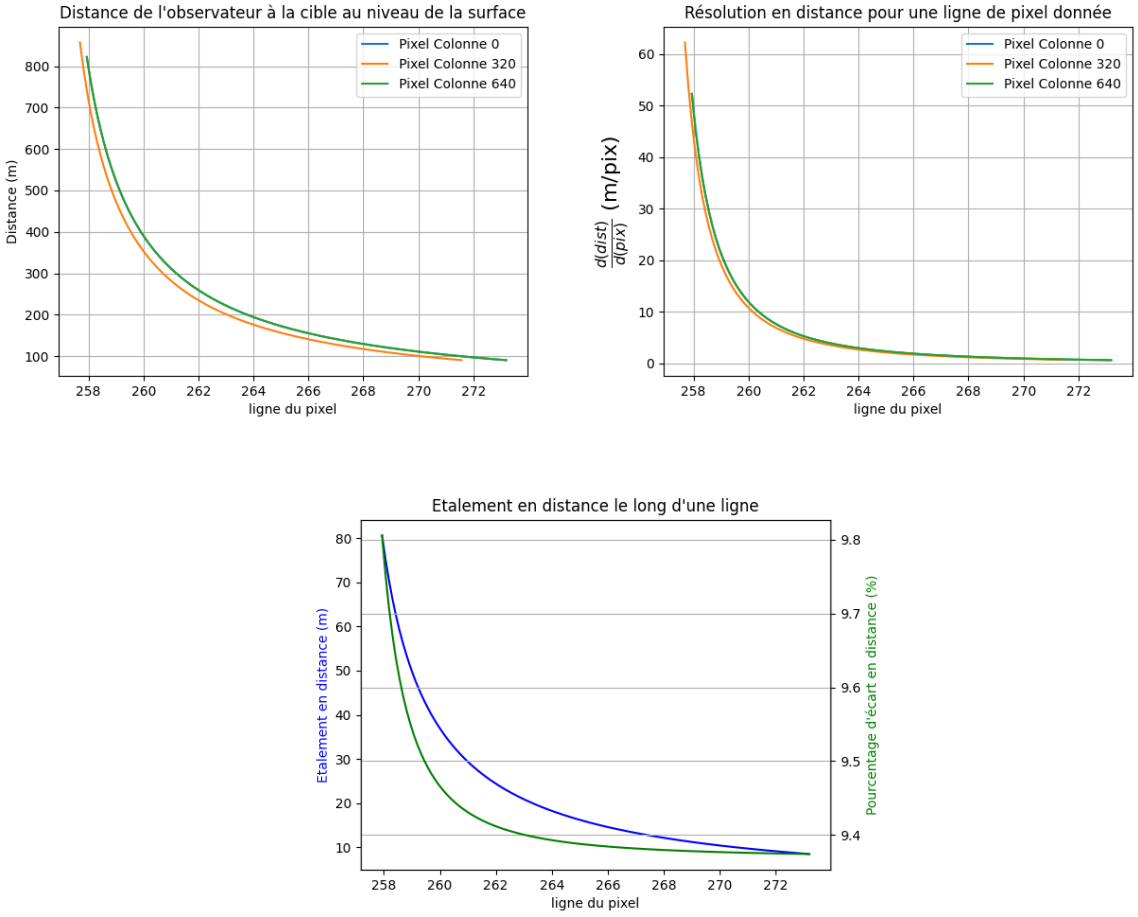


FIGURE 7 : DISTANCE, RESOLUTION ET ETALEMENT D'UNE LIGNE DE PIXEL PROJETEE DANS LE REPERE CARTESIEN DU SYSTEME

NB : La courbe représentant la colonne de pixels 0 est identique et cachée par la courbe de la colonne 640

Sur le premier graphique, on peut observer la distance évoluer selon la position du pixel sur l'axe des ordonnées en supposant que l'horizon est situé au milieu de l'image. La distance d'une cible par rapport à l'observateur évolue théoriquement de manière inversement proportionnelle à l'écart observé entre l'horizon et la limite inférieure de la cible sur l'image. On remarque que la distance décroit très rapidement sur les premiers pixels puis s'approche de 0 au fur et à mesure que l'écart en pixel entre l'horizon et la cible augmente. On remarque que la courbe d'évolution diffère selon la position en abscisse sur l'image mais reste quasiment la même.

Sur le deuxième graphique, on observe la résolution en distance pour une ligne de pixel donnée, ce qui se traduit par la distance réelle que représente un pixel lorsqu'il est plus ou moins proche de l'horizon.

Le dernier graphique montre l'étalement en distance le long d'une ligne. L'étalement correspond à la différence maximale de distance observée d'une cible le long d'axe horizontal donné (ligne).

Erreur de mesure de la distance provoquée par une erreur de détection

Qu'il s'agisse d'une erreur de segmentation de l'horizon ou d'une erreur de délimitation de la *bounding box* de la cible, seul l'écart observé entre la ligne d'horizon obtenue par segmentation et la limite inférieure de la boite compte pour l'estimation de la distance de l'objet puisqu'il définit l'angle γ qui permet de projeter la cible sur l'eau. On peut ainsi dire que ces erreurs ont des impacts équivalents sur la mesure.

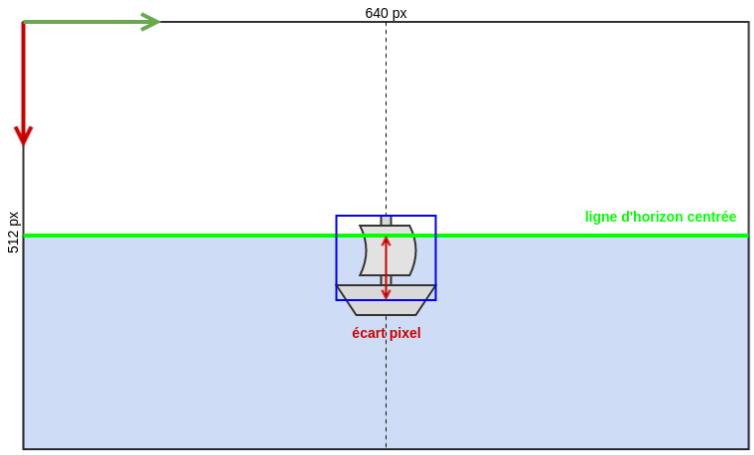


FIGURE 8 : REPRESENTATION D'UNE ERREUR DE DELIMITATION DE LA *BOUNDING BOX*

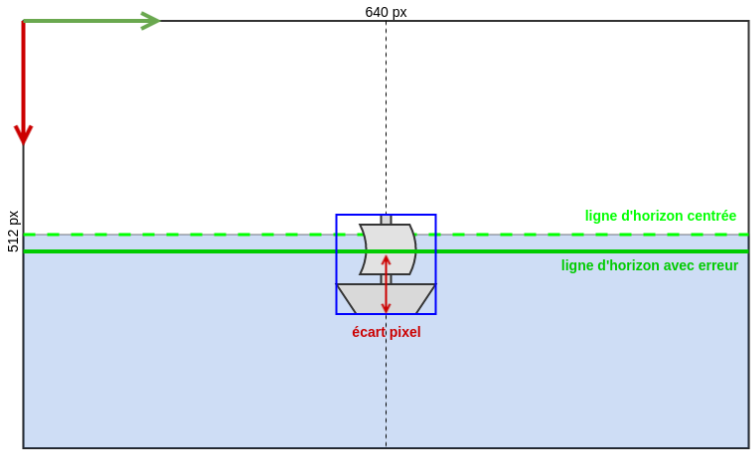


FIGURE 9 : REPRESENTATION D'UNE ERREUR DE SEGMENTATION DE L'HORIZON

En fonction de la distance de la cible

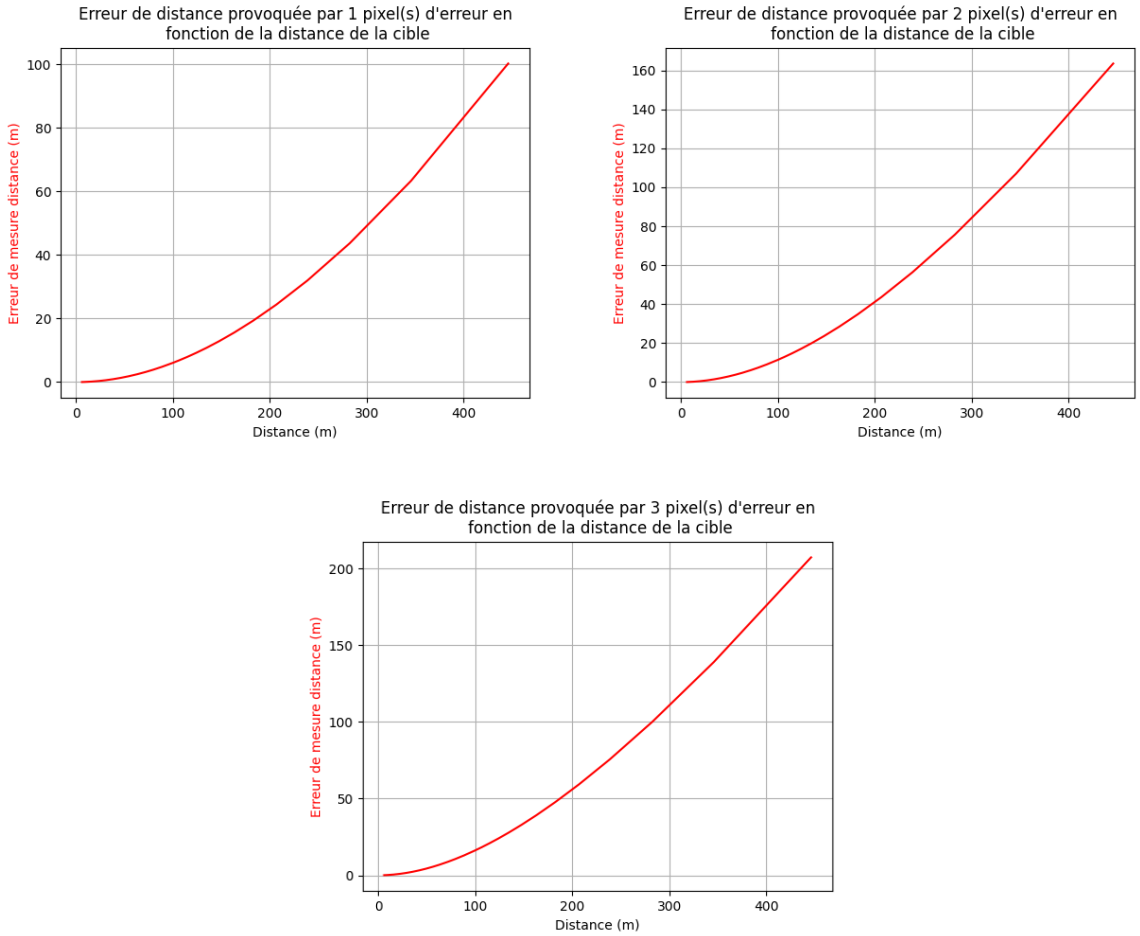


FIGURE 10 : ERREUR DE DISTANCE PROVOQUEE PAR UNE ERREUR EN PIXEL EN FONCTION DE LA DISTANCE DE LA CIBLE

Ces trois graphiques exposent l'erreur commise en distance, causée par une mauvaise détection de la part du réseau de neurones, en fonction de la distance de la cible par rapport à l'observateur. On y représente les erreurs pour 1, 2 et 3 pixels d'écart.

Aussi infime soit-elle, une erreur de segmentation d'un pixel entraine, sur une distance lointaine, une erreur de mesure de plus d'une centaine de mètres. Par exemple, pour une cible placée à 300 m, l'erreur sur la distance induite par un pixel d'écart sur la segmentation est de 50 m. Pour 2 pixels c'est de 80 m et 120 m pour 3 pixels d'écart.

La distance estimée d'une cible située à une longue distance (plus de 500 m) et dont sa position sur l'image est proche de la ligne d'horizon est très imprécise et d'autant plus sensible aux sauts de mesure.

Erreur de distance provoquée par le trait de côte perçu comme la ligne d'horizon

Il peut arriver que la ligne d'horizon soit erronée à cause de la côte visible au loin.

En effet l'horizon normalement est défini par la surface de l'eau au loin, cependant, si cet horizon est masqué par la côte, la segmentation de l'horizon donnera une ligne d'horizon qui ne correspond pas à celle recherchée mais la ligne qui sépare la côte de la mer soit le trait de côte.

La distance théorique de l'horizon est finie et peut être calculée, pour une hauteur de l'observateur de 2 mètres, la distance théorique de l'horizon est de 5051.0 mètres.

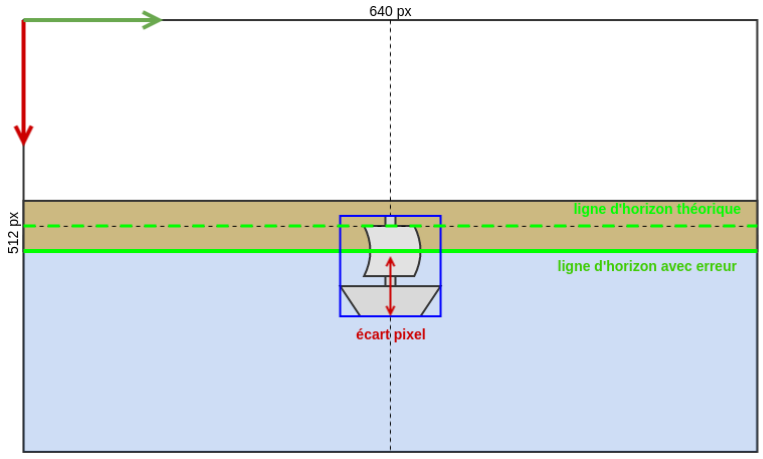
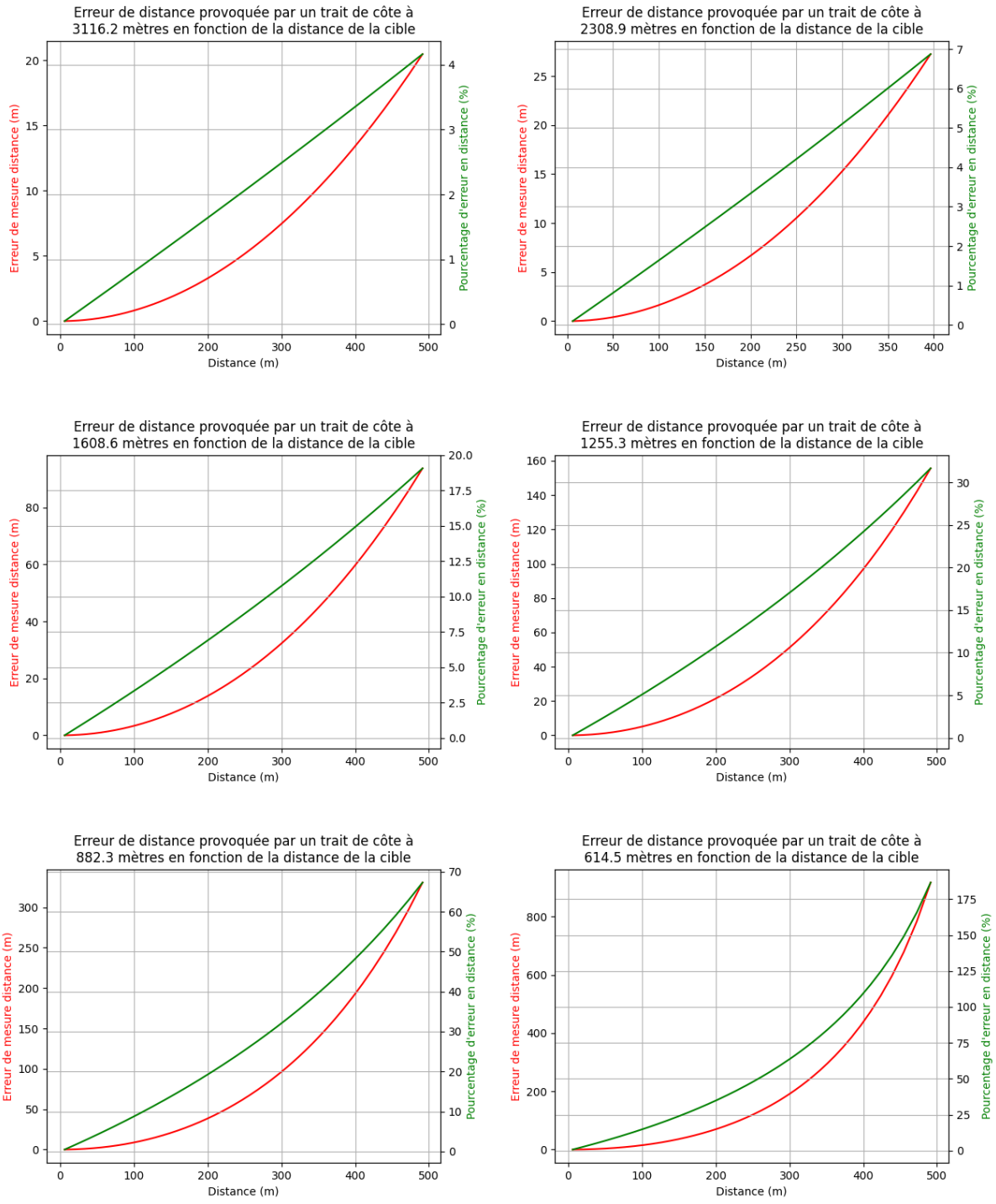


FIGURE 11 : REPRESENTATION D'UNE ERREUR DE DETECTION CAUSEE PAR LA CONFUSION AVEC LE TRAIT DE COTE

En fonction de la distance de la cible



Ces 6 autres courbes exposent l'erreur commise en distance, en fonction de la distance de la cible par rapport à l'observateur, causée par le trait de la côte qui fausse la ligne d'horizon. On y représente les erreurs pour 6 distances différentes de l'observateur à la côte.

On remarque que l'erreur d'estimation de la distance croit d'autant plus à mesure que l'observateur est éloigné de la cible et lorsque la côte est proche de l'observateur.

La côte joue elle aussi un rôle important dans la mesure de la distance d'une cible, si elle n'est pas suffisamment éloignée (moins de 1500 m) elle provoque des erreurs non négligeables. Plus la côte est proche et plus l'estimation sera faussée. Par exemple, pour une situation où la côte située à 2300 m de l'observateur et une cible observée à 500 m, l'erreur commise représentera plus de 10% de la distance mesurée. Pour la côte à 1000 m, l'erreur représente 30% et grimpe à 200% pour la côte à 600 m.

Pour évaluer l'équivalence entre l'erreur causé par le trait de côte et l'erreur en pixel causé par une mauvaise segmentation voire l'annexe : Equivalence entre l'erreur causé par le trait de côte et l'erreur en pixel causé par une mauvaise segmentation.

Ecart de distance causé par une variation de hauteur de l'observateur en fonction de la distance de la cible

Phénomène de pilonnement (heave)

Le pilonnement [2] désigne le déplacement vertical d'unbateau (de haut en bas), c'est l'un des trois déplacements possibles d'un navire avec l'embardée et le cavalement

Il est causé par la rencontre avec les vagues. Particulièrement désagréable pour les passagers et l'équipage, il peut être associé au tangage pour créer le tossage.

Pilonnement (*heave*) négatif de l'observateur

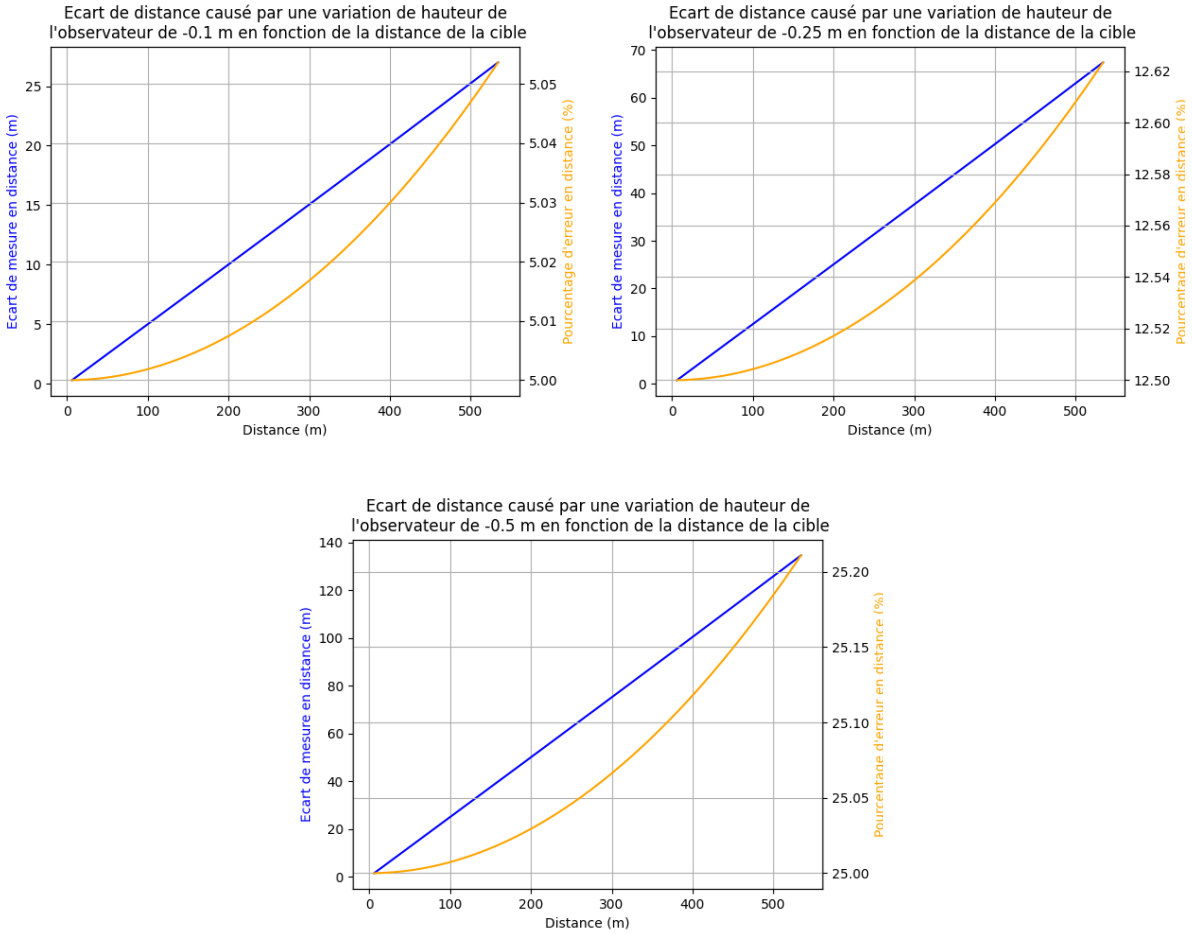


FIGURE 12 : ECART DE DISTANCE CAUSE PAR UNE VARIATION DE HAUTEUR NEGATIVE DE L'OBSERVATEUR EN FONCTION DE LA DISTANCE DE LA CIBLE

Ces courbes montrent les écarts de mesure de distance de la cible avec la mesure réelle engendrés en fonction de la distance de la cible pour un pilonnement négatif de l'observateur, c'est à dire un abaissement par rapport à sa hauteur moyenne. On y représente aussi le pourcentage de l'erreur commise sur l'estimation.

Pilonnement (*heave*) positif de l'observateur

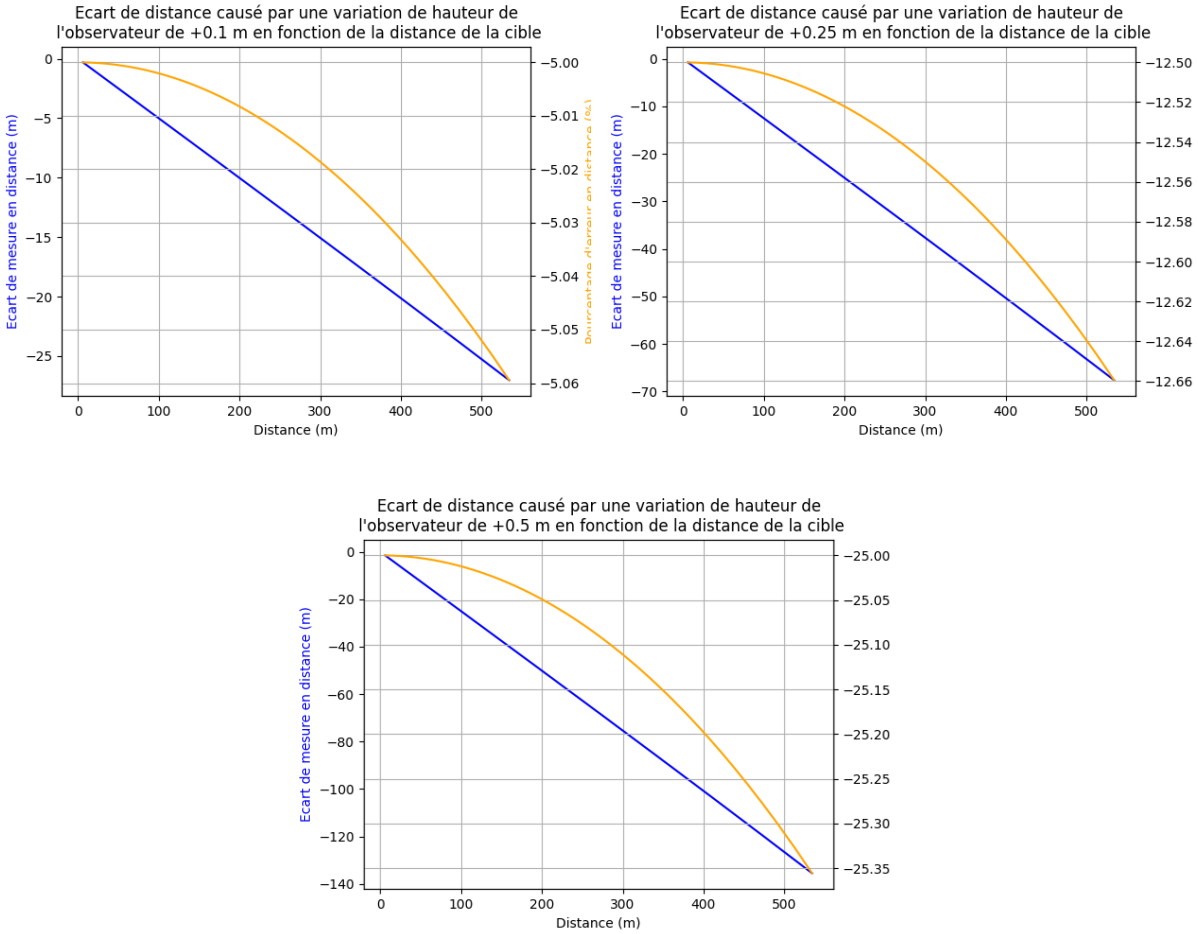


FIGURE 13 : ECART DE DISTANCE CAUSE PAR UNE VARIATION DE HAUTEUR POSITIVE DE L'OBSERVATEUR EN FONCTION DE LA DISTANCE DE LA CIBLE

Ces courbes montrent les écarts de mesure de distance de la cible avec la mesure réelle engendrés en fonction de la distance de la cible pour un pilonnement positif de l'observateur, c'est à dire une élévation par rapport à sa hauteur moyenne.

Ecart de distance causé par une variation de hauteur de la cible en fonction de la distance de celle-ci

Représentation du pilonnement de la cible

Si on observe attentivement l'écart qui sépare la cible (bateau) de l'horizon sur cette image animée, on remarque qu'il oscille : l'écart se réduit puis d'étend, et cela alors que le navire se déplace à un rythme constant. De la même manière que la hauteur de l'observateur varie par l'effet du pilonnement, la cible subit elle aussi cette variation de hauteur.

L'effet de perspective est aussi à prendre en compte. En effet si l'observateur est rabaissé, la cible parait par conséquent plus proche de l'horizon.

Pilonnement (*heave*) négatif de la cible

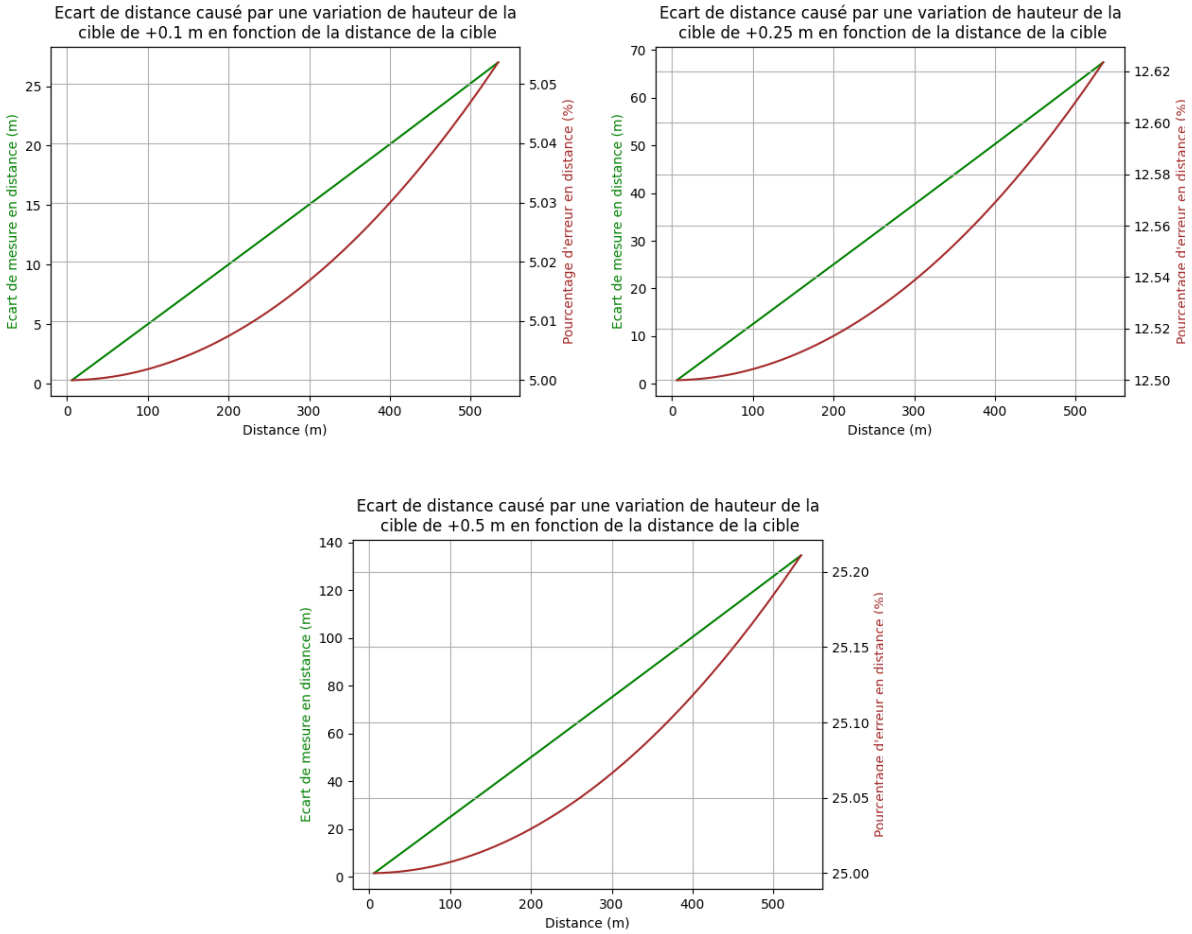


FIGURE 14 : ECART DE DISTANCE CAUSE PAR UNE VARIATION DE HAUTEUR NEGATIVE DE LA CIBLE EN FONCTION DE LA DISTANCE DE LA CIBLE

Ces courbes montrent les écarts de mesure de distance de la cible avec la mesure réelle engendrés en fonction de sa distance pour un pilonnement négatif de la cible cette fois-ci. On y représente, pareillement, le pourcentage de l'erreur commise sur l'estimation.

Pilonnement (*heave*) positif de la cible

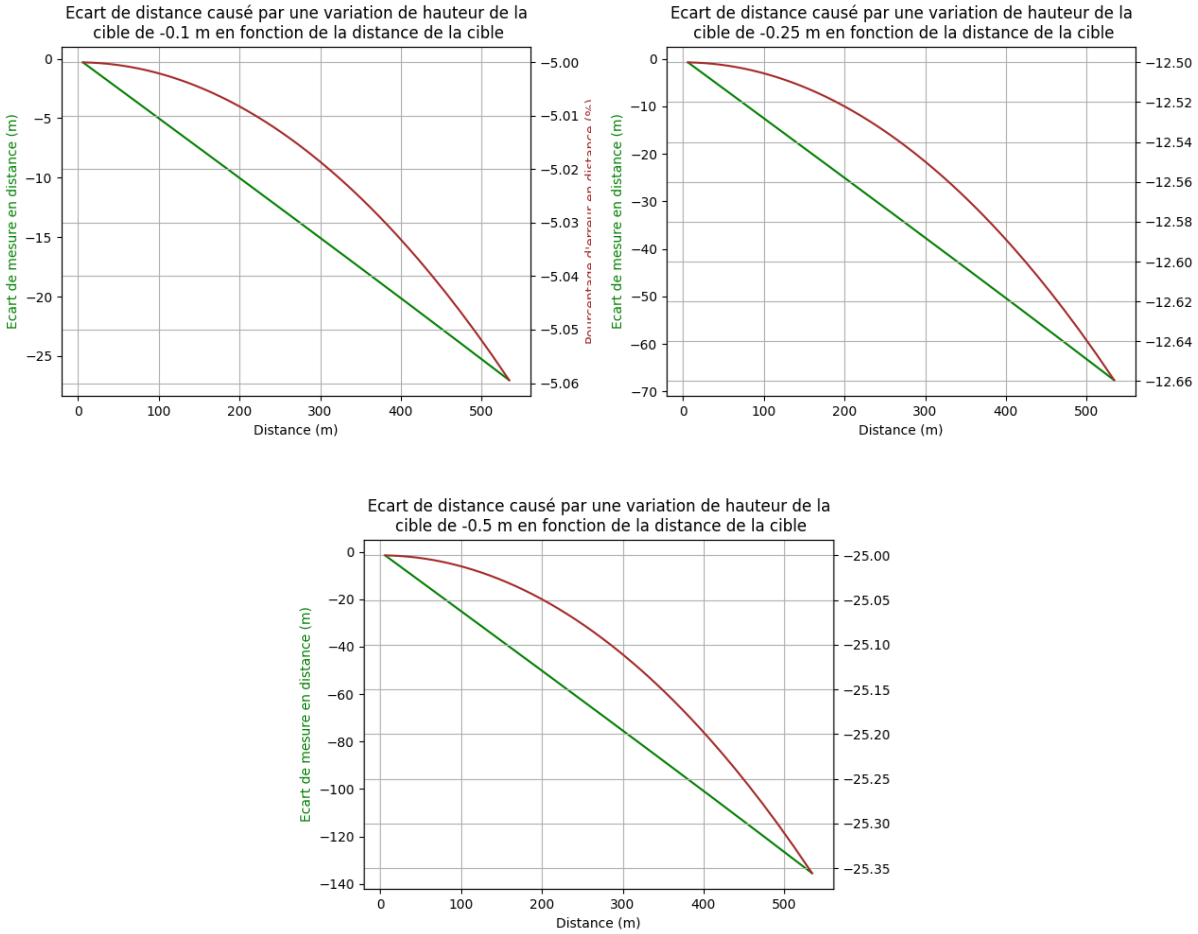


FIGURE 15 : ECART DE DISTANCE CAUSE PAR UNE VARIATION DE HAUTEUR POSITIVE DE LA CIBLE EN FONCTION DE LA DISTANCE DE LA CIBLE

Ces courbes montrent les écarts de mesure de distance de la cible avec la mesure réelle engendrés en fonction de sa distance pour un pilonnement positif de la cible.

Similarité des résultats d'écart de mesure entre le pilonnement de la cible et celui de l'observateur

La méthode de mesure de distance MDT exploite l'angle formé par l'objet et l'horizon avec la caméra infrarouge $\gamma = \theta - \alpha$ et la hauteur de celle-ci dessus de la surface de l'eau f pour estimer la distance de l'objet (*Track*), ici $x = \rho\cos(b)$.

Cet angle γ est calculé à partir du nombre de pixel entre l'horizon et l'objet grâce aux paramètres caméra :

$$\gamma = \frac{\Delta_v V_{fov}}{V_{pix}}$$

Ce même angle permet ensuite de déterminer la distance de l'observateur.

On rappel la méthode de calcul décrite précédemment :

Soit h la hauteur de la caméra, et R le rayon de la terre, on a θ représentant l'angle formé par la verticale et l'horizon avec l'observateur :

$$\theta = \tan^{-1}\left(\frac{R}{\sqrt{2Rh}}\right)$$

et α représentant l'angle formé par la verticale et la cible avec l'observateur :

$$\alpha = \frac{\pi}{2} + \sin^{-1}\left(\frac{R^2 - (R+h)^2 - \rho^2}{2(R+h)\rho}\right)$$

Par simple soustraction de ces angles on obtient l'angle γ formé par l'horizon et la cible :

$$\gamma = \theta - \alpha$$
$$\gamma = \arctan\left(\frac{R}{\sqrt{2 \cdot R \cdot h}}\right) - \frac{\pi}{2} + \arcsin\left(\frac{R^2 - (R+h)^2 - \rho^2}{2 \cdot (R+h) \cdot \rho}\right)$$

Pour observer les différences provoquées par un pilonnement sur l'observateur d'une part et la cible d'autre part, on suppose alors deux variables γ_o et γ_c représentant respectivement l'angle γ lorsque l'observateur (O) ou la cible (C) subit une variation de hauteur z causé par le pilonnement (*heave*).

$$\gamma_o = \arctan\left(\frac{R}{\sqrt{2 \cdot R \cdot (h+z)}}\right) - \frac{\pi}{2} + \arcsin\left(\frac{R^2 - (R+h+z)^2 - \rho^2}{2 \cdot (R+h+z) \cdot \rho}\right)$$
$$\gamma_c = \arctan\left(\frac{R}{\sqrt{2 \cdot R \cdot h}}\right) - \frac{\pi}{2} + \arcsin\left(\frac{(R+z)^2 - (R+h)^2 - \rho^2}{2 \cdot (R+h) \cdot \rho}\right)$$

On trace ensuite γ_o en rouge et γ_c en bleu sur un graphique en fonction du pilonnement z afin d'observer leur variation et les comparer :

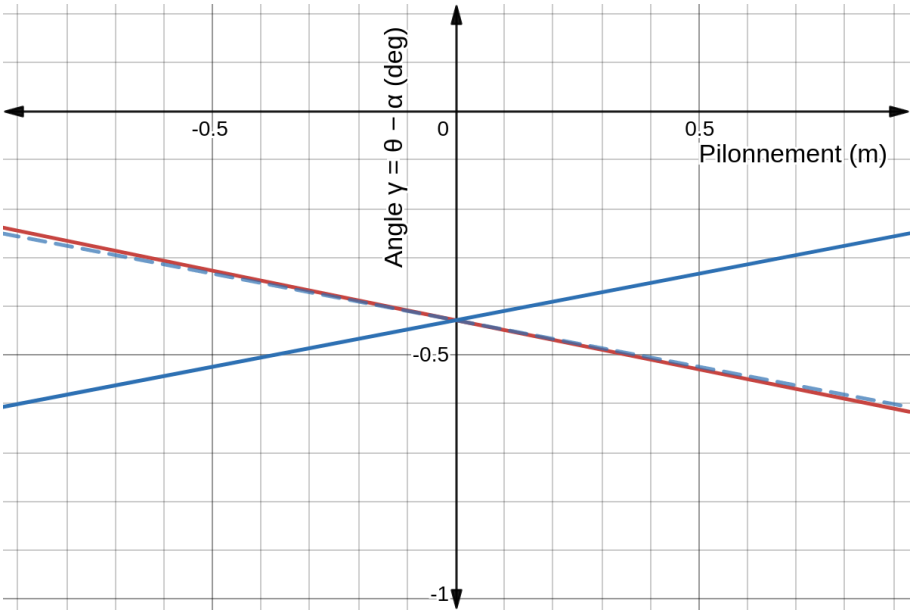


FIGURE 16 : ANGLE GAMMA EN FONCTION DE LA VALEUR DU PILONNEMENT DE L'OBSERVATEUR ET DU PILONNEMENT DE LA CIBLE

On remarque γ_o et γ_c sont quasiment symétriques selon l'axe vertical. Cela signifie que γ_o est quasiment égal à γ_c pour une valeur de pilonnement opposé.

Ce résultat justifie donc que les écarts provoqués soient par un pilonnement de l'observateur soit par un pilonnement de la cible soient presque identiques pour des signes de pilonnement opposés.

Ainsi, le pilonnement de l'observateur a le même effet sur la mesure que le pilonnement de la cible.

Conclusion de l'étude

Les paramètres qui influencent le plus la précision de localisation de la cible sont : la précision de segmentation, l'estimation de la vraie position de l'horizon et l'estimation de la hauteur du porteur sur l'eau. Nous allons voir par la suite comment nous avons adressés les deux derniers points.

Correction des erreurs de mesure liées au mouvement vertical du navire

En mer, la houle applique sur le navire des forces qui déstabilise le navire. La caméra infrarouge embarquée à bord est alors perturbée par ces mouvements.

La caméra, très sensible aux erreurs de mesure va avoir du mal à fournir des données correctes permettant de déterminer la position des objets en mer avec précision. C'est pourquoi il est important de prendre en compte les variations d'angle captées par l'unité de navigation inertielle.

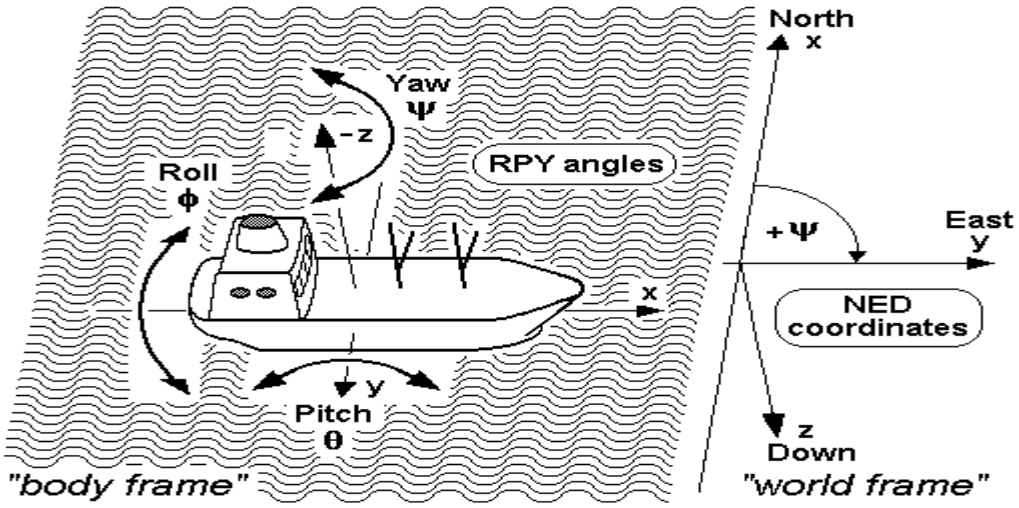


FIGURE 17 : MOUVEMENTS DE TRANSLATION ET DE ROTATION D'UN NAVIRE SUR L'EAU

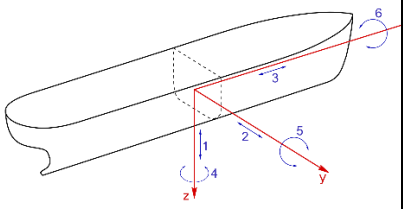
Les mouvements qui engendrent le plus d'erreurs sont le roulis (roll) et le tangage (pitch). Ils peuvent être estimés par l'INS (unité de navigation inertielle) embarquée et exploité dans le nœud de détection et de suivi des cibles par caméra infrarouge nommé p_detect_ir_cnn. Cela permettrait au système de gagner en précision par rapports aux mouvements effectués par le navire ce pourrait réduire les erreurs résultantes sur la distance et l'azimut de l'obstacle. Grâce aux tfs sur ROS, on peut reproduire les mouvements de structure du navire dans un environnement virtuel. On met ainsi à jour l'état du navire et de ses mouvements articulaires au même rythme que les capteurs est ainsi corrigé et les coordonnées de la caméra infrarouge dans l'espace peut être réestimées de manière dynamique pour offrir une meilleure précision de calcul.

Phénomène de pilonnement (*heave*)

Cependant, à l'heure actuelle, les mouvements pris en compte concernent uniquement les rotations du navire mais pas le mouvement de pilonnement [2]. Ce déplacement vertical qui se traduit par la variation en altitude du navire n'est pas considérée.

Il existe plusieurs phénomènes dus à la houle qui mettent en mouvement un navire en mer : Trois translations et trois rotations.

A l'heure actuelle, cinq de ces mouvements sont pris en compte : Le cavalement et l'embardée sont estimé directement grâce à la localisation du navire. Les mouvements de rotation tels que le roulis, le tangage et le lacet sont quant à eux estimés par l'INS embarquée.

Axe de rotation / de déplacement	**Déplacement**	**Mouvement rotationnel**		**Angle**	**Schéma**
x : Longitudinal	Cavalement (3) : Déplacement horizontal d'avant en arrière	Roulis (6) : Oscillation latérale de bâbord à tribord (vagues, vent, etc.)	Axe de rotation dans la plus grande longueur du bateau.	Gite / Angle de roulis	
y : Transversal	Embardée (2) : Déplacement horizontal latéral	Tangage (5) : Oscillation d'avant en arrière, la proue se relève et s'abaisse (vagues, etc.)	Axe de rotation dans la largeur du bateau	Assiette (pas de sens en marine)	
z : Vertical	**Pilonnement (1) : Déplacement vertical de l'altitude du bateau**	Lacet (4) : Oscillation sur le plan horizontal, modifiant le cap (mouvement de la barre, dérive lié au courant, au vent, vagues...)	Axe de rotation coaxiale au mât	Cap	

Reste le phénomène de pilonnement observé par l'INS. Une contrainte importante car elle va directement influencer la hauteur de la caméra par rapport à la surface de l'eau et dans ce sens elle provoque une erreur d'imprécision dans le calcul de distance effectué par MDT.

Ajustement du modèle dynamique MDT

Lors des précédents essais effectués en mer avec une houle plus ou moins importante, la distance estimée d'une cible au loin est fortement perturbée. En mettant de côté les erreurs de mesures liées aux imprécisions de la segmentation de l'horizon, il est possible que cette instabilité de la mesure du système de détection et de localisation d'obstacle par caméra IR soit liée au manque de justesse de la valeur de la hauteur de la caméra pouvant être influencée par phénomène de pilonnement. Ce mouvement, non prit en compte, pourrait être la cause d'une partie de ces variations brusques que l'on observe sur la mesure de la distance d'une cible en mer.

L'objectif de ce projet est donc de prendre en compte les variations de hauteur du navire le long de l'axe vertical du navire pour réestimer sa position verticale sur l'eau. On aura ainsi une connaissance un peu plus précise du positionnement de la caméra infra-rouge dans l'espace et la mesure pourrait être de ce fait, plus stable.

Implémentation de la mesure du pilonnement dans la dynamique du système

La modélisation dynamique du système MDT s'est constituée avec le temps et de nombreuses modifications de sa structure se sont succédé. Le dernier modèle exploite les données gyroscopiques de la centrale de navigation inertielle pour orienter virtuellement le navire.

Dans le système MDT, le navire est modélisé virtuellement au moyen de tfs. Dans ROS [3], les tfs représentent des référentiels de coordonnées. Les tfs maintiennent la relation entre eux dans une structure arborescente mise en mémoire tampon au cours du temps, et permettent ainsi de transformer des points, des vecteurs, etc. entre deux cadres de coordonnées à tout moment. On utilise les tfs dans MDT pour connaitre les coordonnées dans l'espace de chaque capteur par rapport à une quelconque autre position dans le repère cartésien. Cela nous permet de rapidement estimer des distances et orientions caractéristiques du système MDT.

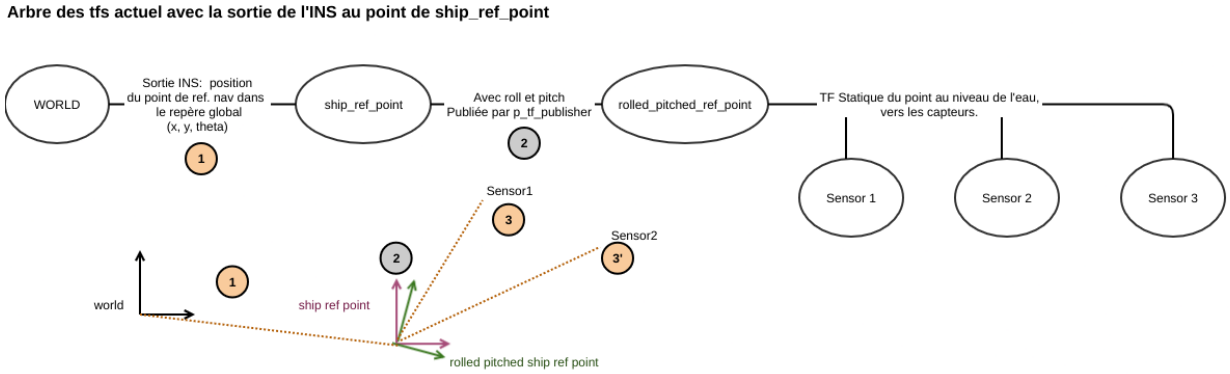


FIGURE 18 : ARBRE ACTUEL DES TFS

L'arbre ci-dessus présente l'architecture des tfs utilisées pour représenter le modèle dynamique standard d'un navire équipé d'MDT. On y retrouve un référentiel du navire à la surface de l'eau, à plat, et un autre du navire orienté par les mesures gyroscopiques de l'INS, les référentiels sous-jacents sont des tfs statiques que l'on peut représenter comme de simples bras de levier. Ce modèle ne traduit pas les variations de hauteur que peut subir le navire lorsqu'il est contraint au pilonnement.

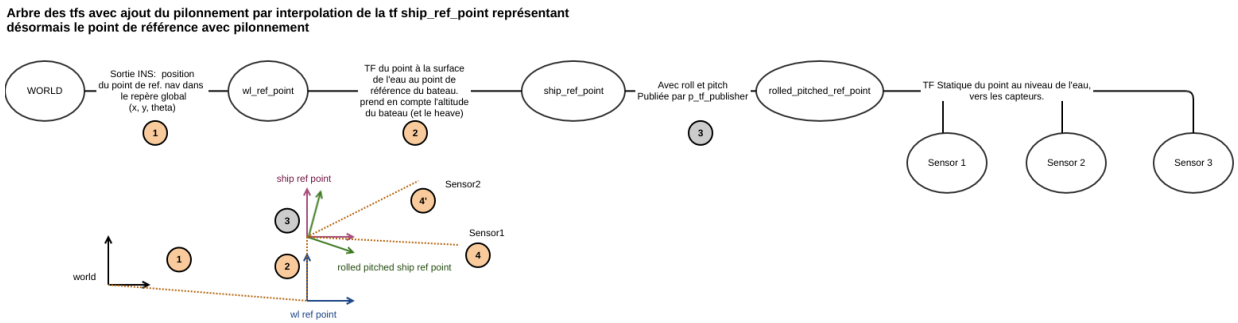
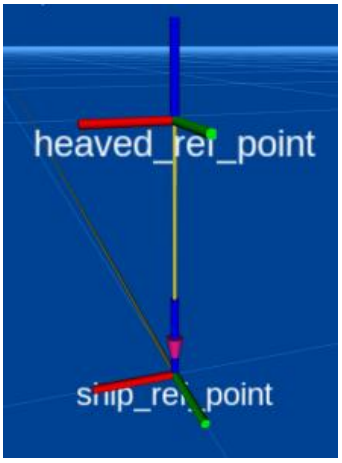


FIGURE 19 : ARBRE DES TFS AVEC AJOUT DU PILONNEMENT

Ce nouveau modèle a particularité d'interpoler une *tf* supplémentaire, intermédiaire, entre le référentiel du navire à plat et celui du navire orienté. On obtient donc un nouveau révérenciel exprimant l'état du navire pilonné puis orienté qui devrait en théorie s'approcher davantage de la réalité. Ce modèle n'était pas réalisable auparavant car la mesure de pilonnement par l'INS n'est apparue que récemment dans les nouvelles versions du capteur.



La création d'un nœud nommé *p_tf_heave_publisher* permet, à la manière du nœud actuel *p_tf_publisher*, de publier la mesure du pilonnement en fonction des mesures renvoyées par l'INS. Mis à part quelques modifications au sein du nœud de détection et de localisation des cibles, *p_detect_ir_cnn*, l'ajout d'un référentiel intermédiaire dans l'arbre des *tfs* du système ne change pas drastiquement son fonctionnement global. Le nœud *p_detect_ir_cnn* se sert, en effet, des *tfs* ROS pour, à tout moment, connaitre les coordonnées, distances et orientations caractéristiques du système dans son ensemble afin de réaliser plus facilement les calculs et projections mathématiques nécessaire à son fonctionnement.

Maintenant que le paramètre de pilonnement est pris en compte dans le système MDT, il est indispensable de l'évaluer si l'on souhaite démontrer son apport à MDT.

Résultats de la prise en compte de l'effet de pilonnement dans MDT

Pour mettre à l'épreuve cette nouvelle architecture, il faut pouvoir comparer ses performances avec la précédente.

Jeu de données utilisé

Pour étudier la validité de l'implémentation du pilonnement (*heave*) dans le système de détection et localisation d'obstacle via caméra IR (*p_detect_ir_cnn*), nous l'évaluerons sur une situation simple, celui de l'approche frontale progressive du navire équipé du système MDT vers une cible fixe.

Pour cela, on dispose d'un enregistrement ROS, nommé *rosbag*, un fichier de registres contenant l'ensemble des états du navire et des mesures publiées pas les divers capteurs lors d'un précédent essai. On dispose alors de l'essentiel des données nécessaires au système de détection par caméra IR. Ces données contiennent notamment la mesure des variations de hauteur provoquées par l'effet du pilonnement mesuré par l'INS à ce moment.

Visualisation RVIZ

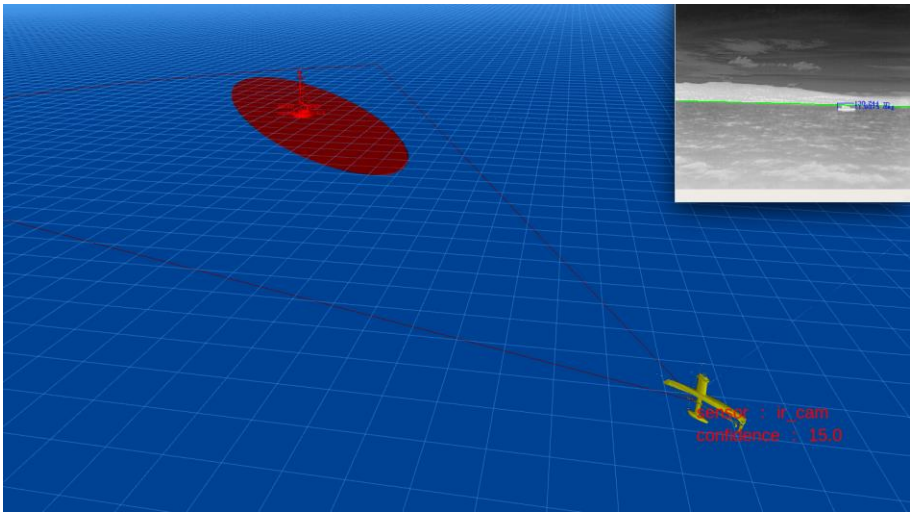


FIGURE 20 : VISUALISATION DU NAVIRE MDT DANS UN ENVIRONNEMENT 3D VIRTUEL

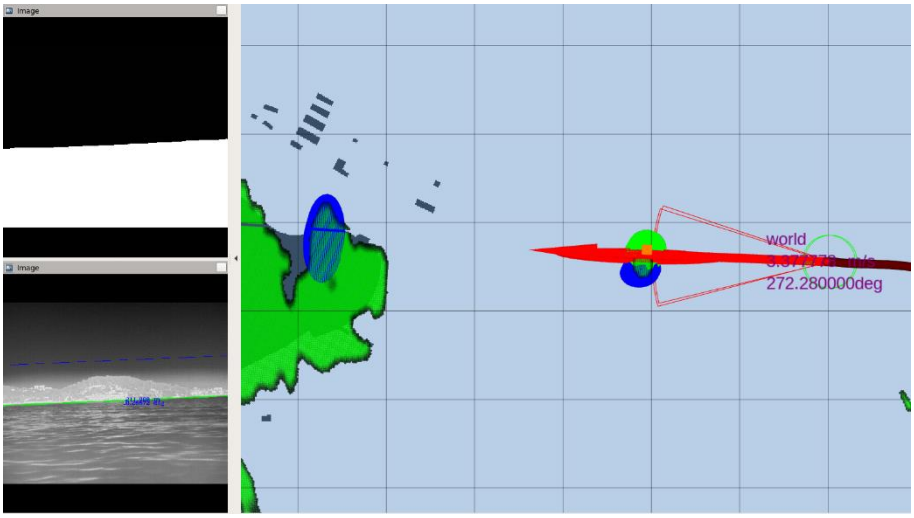


FIGURE 21 : VISUALISATION DU NAVIRE MDT SUR LA CARTE MARINE 2D

La simulation RVIZ permet de visualiser la cible observée en temps réel, ainsi que la détection et le positionnement estimés par le système et matérialisés par une zone d'incertitude. Cette représentation permet par la même occasion d'observer l'image de la caméra IR contenant les informations relatives à la segmentation de l'horizon et l'encadrement de la cible.

Analyse des données

Ici, l'objectif est de déterminer si l'implémentation du pilonnement dans le système de détection apporte de meilleurs résultats que le système actuel, c'est à dire sans *heave*, on compare alors les mesures des détections des différents systèmes, la version d'origine, une version avec prise en compte du pilonnement et une troisième avec une mesure prétraitée par la centrale inertielle du pilonnement.

Données filtrées

On représente les données collectées en prenant soin d'exclure des valeurs aberrantes causées par les échecs de détection de l'horizon.

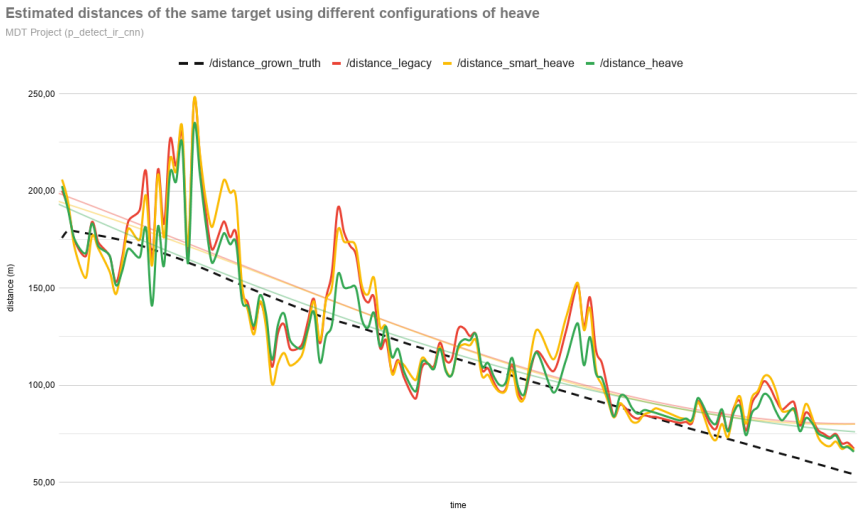


FIGURE 22 : DISTANCES ESTIMEES DE LA CIBLE EN UTILISANT DIFFERENTES CONFIGURATIONS DU PILONNEMENT

- /distance_grown_truth : Distance réelle de la cible
- /distance_legacy : Distance observée par p_detect_ir_cnn → cnn_segm_horizon, nœud de détection dit "*legacy*"
- /distance_heave : Distance observée par p_detect_ir_cnn → heave_cnn, nœud de détection avec prise en compte du pilonnement simple
- /distance_smart_heave : Distance observée par p_detect_ir_cnn → heave_cnn, nœud de détection avec prise en compte du pilonnement prétraitée

Au-delà du fait que la mesure soit assez instable, on remarque que la distance estimée avec la prise en compte du *heave* simple est, d'une part, moins perturbée que ne l'est celle du système actuel. D'autre part, les valeurs de distances semblent individuellement et globalement plus proche de la réalité. On note par ailleurs les performances liées à l'utilisation de la mesure dite *smart heave*, sont en deca de celles du simple *heave*, cela s'explique notamment par le fait que nous n'avons pas pris le temps d'étalonner cette mesure.

Données lissées par une moyenne glissante

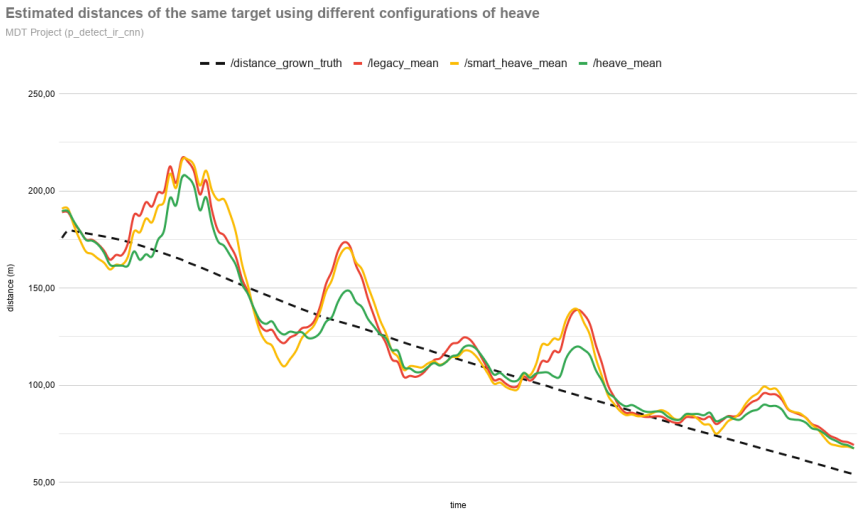


FIGURE 23 : DISTANCES LISSEES DE LA CIBLE EN UTILISANT DIFFERENTES CONFIGURATIONS DU PILONNEMENT

- /distance_grown_truth : Distance réelle de la cible
- /legacy_mean : Moyenne glissante sur 5 valeurs, centrée, de /distance_legacy
- /heave_mean : Moyenne glissante sur 5 valeurs, centrée, de /distance_heave
- /smart_heave_mean : Moyenne glissante sur 5 valeurs, centrée, de /distance_smart_heave

On moyenne les courbes par application d'une moyenne glissante sur 5 valeurs, on se rend un peu mieux compte de l'effet du pilonnement (*heave* simple) sur l'estimation de la distance. En effet la courbe correspondant à la distance estimée à partir du *heave* simple est moins éloignée de la distance réelle que les autres distances (*legacy* et *smart heave*). Notamment sur la deuxième et quatrième oscillation, on remarque qu'elle est nettement moins en erreur que les autres. On en déduit que le *heave* simple permet d'atténuer, bien que légèrement, l'erreur de mesure de la distance de la cible observée par la détection via caméra IR (*p_detect_ir_cnn*).

Conclusion

Devant cette amélioration, en théorie logique mais qui demandait à être vérifiée par la pratique, de l'implémentation des variations de hauteur dans le modèle dynamique MDT, la fonctionnalité a pu être assimilée par la version originale (*master*) du nœud *p_detect_ir_cnn* du projet MDT.

Erreur due à la segmentation de l'horizon

Pour déterminer la distance de la cible observée à travers la caméra IR, le système s'appuie sur les mesures d'angles estimés par l'INS pour projeter les informations de l'image dans son environnement grâce au calcul géométrique détaillé précédemment.

Cependant, le système de segmentation - déduit la ligne d'horizon de la même manière que pourrait le faire un humain - il détermine une ligne qui sépare au mieux le ciel et la Terre. Comme l'œil humain, le système est sujet à l'erreur que l'on peut effectuer sur l'estimation de l'horizon vrai.

En effet, l'horizon vrai, c'est à dire au sens mathématique, est un cercle centré sur l'observateur entre ciel et la surface de l'eau sur Terre (altitude nulle), tenant compte de la courbure de cette dernière [3]. Néanmoins, cette courbure peut largement être approximé à une ligne étant donné la faible altitude de la caméra du système MDT. Mais en beaucoup d'endroits l'horizon n'est pas visible à cause des obstacles, principalement les côtes terrestres. Et puisque le réseau de neurones de segmentation de l'horizon est entrainé pour reconnaitre la séparation de l'eau avec le ciel, comme pourrait le faire l'œil humain, il se peut que la segmentation soit faite avec le trait de séparation entre la mer et la côte.

Lorsque la côte est visible et plus proche que la distance de l'horizon, l'information sur ce dernier est donc biaisée et ce biais entraine logiquement des erreurs sur la mesure des cibles observées par la caméra IR.

L'étude présentée plus tôt met en valeur l'impact de l'erreur causé par le trait de côte en fonction de sa proximité avec l'observateur. On peut constater que dans les conditions standards d'utilisation (hauteur de l'observateur de 2 mètres, paramètres de caméra du système, etc.) ; à partir de 1500 mètres la côte provoque sur la mesure de distance de la cible une erreur non négligeable de 20% et augmente au fur et à mesure que la côte est proche.

Pour supprimer cette erreur, il nous faudrait déterminer l'horizon vrai à travers les obstacles qui obstruent sa visibilité. Cependant, il n'existe pas de solution pour rectifier visuellement le biais causer par la côte de manière systématique puisqu'il n'est pas possible pour la caméra IR de voir l'horizon vrai puisqu'il est caché par la côte, il faut donc conserver ce faux horizon tout en appliquant une correction au système afin d'atténuer l'erreur commise.

En effet, si l'on connait la distance de l'observateur avec la côte, le long d'un angle d'azimut donné, il devient possible d'exploiter cette valeur pour que le système corrige son estimation et qu'il retourne une mesure de distance des cibles plus proche de la réalité.

Justement, le système de services nommé *p_s57_processor* contient un service nommé *get_horizon* destiné à renvoyer la mesure de la distance de l'horizon par rapport à une géo-position fournie, le long d'un azimut donné. Il serait donc dans notre intérêt d'appeler ce service chaque fois que le nœud de projection de la caméra IR du système MDT estime la distance de l'horizon.

Cas de l'horizon couvert par la côte

Dans un premier temps, avant de recourir à l'utilisation d'un service qui nous permet d'obtenir la mesure de la distance du navire à la côte, il est indispensable de vérifier mathématiquement que la connaissance de cette donnée nous permet de l'estimation de la distance d'une cible en mer lorsque l'horizon visible est faux.

Si l'horizon est confondu avec le trait de côte, l'angle gamma correspondant à l'angle fait entre l'horizon et la cible avec l'observateur sera alors erroné. Dans ce cas, il faut envisager l'horizon produit par le trait de côte comme une autre cible dont on connaitrait la distance donnée par le service *get_horizon* du nœud de cartographie *p_s57_processor*.

A partir du *range* ρ_h on peut obtenir l'élévation b_h de l'horizon grâce à la formule obtenue précédemment :

$$\rho_h^2 \cos^2(b_h) + \left((R+h) + \rho_h \sin(b_h)\right)^2 = R^2$$
$$\Leftrightarrow b_h = \sin^{-1}\left(\frac{R^2 - (R+h)^2 - \rho_h^2}{2(R+h)\rho_h}\right)$$

Cette élévation du faux horizon va nous permettre d'obtenir θ, angle entre le nadir et l'horizon avec l'observateur, angle qui jusque-là était obtenu par le calcul théorique basé sur la rotondité de la terre [3] et dépendant uniquement de la hauteur de l'observateur.

$$\theta \neq \tan^{-1}\left(\frac{R}{\sqrt{2Rh}}\right)$$

Cette angle θ est cependant toujours égal à :

$$\theta = \frac{\pi}{2} - \beta_h = \frac{\pi}{2} + b_h$$

avec b_h qui représente l'élévation de l'horizon dépendant de ρ_h, le *range* de l'horizon.

Mais comme on peut déduire le *range* de l'horizon par le service *get_horizon*, il devient possible d'obtenir l'élévation de l'horizon mais si celui ci-est formé par la côte.

Et a maintenant on peut exprimer l'élévation de la cible par rapport à l'élévation de notre faux horizon formé par la côte :

$$b = b\_h - \backslash gamma$$

Puisque γ est estimé à partir du traitement des images et des paramètres caméra, il est dans ce cas, possible de rectifier les estimations des distances des cibles si l'horizon est faussé par la terre.

Utilisation du service d'estimation de distance à la côte pour améliorer la localisation des obstacles

Nous avons aussi constaté que la présence des côtes obstruait la vue de la ligne d'horizon théorique ce qui causait par conséquent une erreur sur la mesure de la distance des cibles (*tracks*) par rapport à l'observateur.

En effet, l'estimation de cette distance se base notamment sur la distance qui sépare l'observateur de l'horizon théorique. Si la côte cache cet horizon, le système de segmentation *p_detect_ir_cnn* identifie la ligne d'horizon par le trait de côte. On parlera alors de faux horizon en contraste avec l'horizon théorique formé normalement par le ciel et la mer. Ce quiproquo aura pour conséquence d'induire une erreur d'appréciation des distances plus importante à mesure que la côte est rapprochée. En sortie de port, par exemple, le système semble fortement perturbé et les distances des obstacles sont anormalement plus élevées.

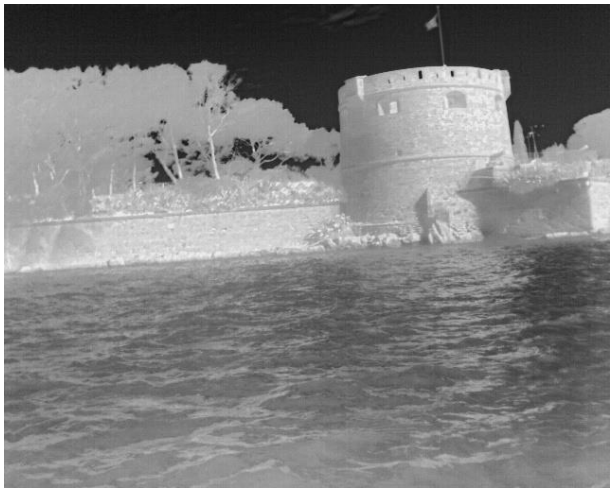


FIGURE 24 : IMAGE DE LA CAMERA IR AVEC LA COTE TRES PROCHE

C'est pourquoi, il serait intéressant d'utiliser le service *get_horizon* au profit du système de détection et de localisation des obstacles en mer, *p_detect_ir_cnn.*

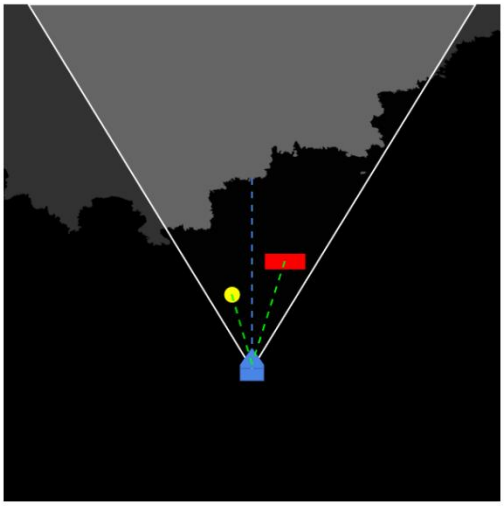
Précédemment, nous avions vu comment le système calculait la distance des *tracks* à l'aide de l'écart entre la cible et l'horizon observé par l'observateur (ici la caméra IR) et de la hauteur de ce dernier par rapport à la surface de l'eau. Par le

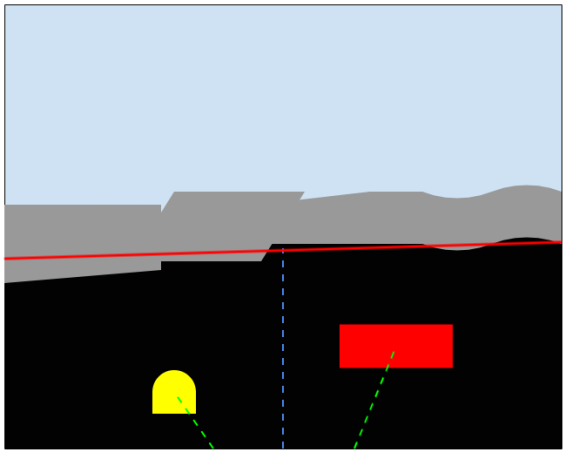
calcul mathématique présenté précédemment et grâce à l'estimation de la distance de l'horizon théorique, on parvient à déduire le "range", c'est à dire, la portée de la cible observée.

La distance de l'horizon théorique par rapport à l'observateur sert de distance de référence dans le calcul de la distance des *tracks* puisque que c'est une donnée facilement accessible. En effet l'horizon théorique dépend uniquement de la hauteur de l'observateur au-dessus de l'eau.

Puisque le système est susceptible de se focaliser sur l'horizon faux, il faut, pour corriger le calcul, associer la bonne distance de cet "horizon" qui sert de référence. Pour cela, il est possible de faire appel service *get_horizon* du module *p_s57_processor* qui renseigne cette donnée. Le nœud *p_s57_processor* est un module de cartographie qui propose des services renvoyant des données relatives à l'environnement du système, il intègre notamment, un service nommé *get_horizon* estimant la distance entre une géo-position donné et les côtes terrestres le long d'un azimut, soit la distance à l'horizon, qu'il soit théorique ou bien formé par les côtes.

En substituant le calcul théorique actuel pour déterminer la distance de l'horizon par la donnée du service *get_horizon* on peut ainsi s'adapter à l'erreur commise par la détection de l'horizon formé par la terre. Puisque le système MDT base son calcul sur le principe que l'horizon est visible et sa distance connue pour projeter tout objets observables sur l'image sur le plan d'eau autour de l'observateur, si l'horizon visible n'est pas nécessairement l'horizon théorique mais que sa distance est connue, on peut projeter de la même manière tout ce qui se situe entre l'observateur et l'horizon.





Concrètement, pour se faire, il nous faudra demander au service *get_horizon* d'effectuer un tracer un rayon sur une carte marine, depuis la position de l'observateur, en direction du cap du navire pour obtenir la distance de l'horizon visible par la caméra IR. Cette information nous permettra par la suite de corriger l'erreur causé par le faux horizon en substituant la distance par celle qui correspond réellement.

Services de cartes

Le nœud *p_s57_processor,* est un nœud ROS permettant d'exploiter les cartes électroniques de navigation officielles (ENC) établies sur la base de données cartographiques vectorielles contenant la description détaillée de chaque objet (marque de balisage, épaves, câbles sous-marins, zones réglementées, sondes, etc.). Le paquet *p_s57_processor* a pour objectif de *parser* (analyser) ces données afin de fournir l'essentiel des informations sur les environs au système MDT. Parmi ces informations utiles on peut citer la carte des côtes sous la forme d'une grille d'occupation, la distance de l'horizon le plus proche (vrai ou terrestre), le taux de couverture des blobs radar avec la terre permettant de fileter les objets flottants des côtes.

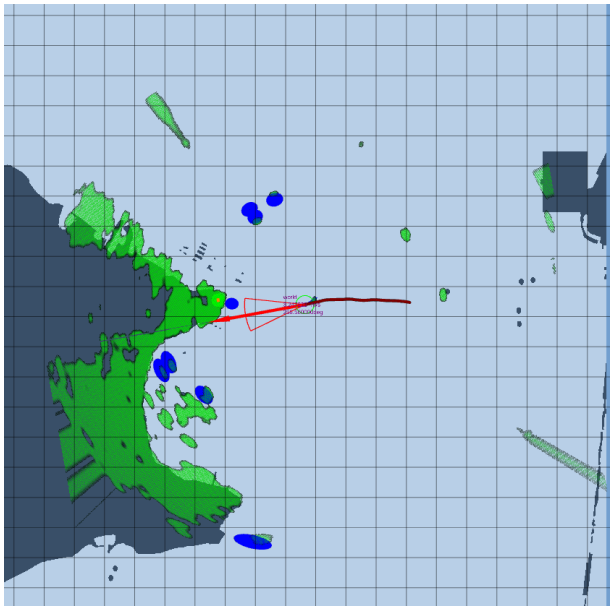


FIGURE 25 : NAVIRE MDT REPRESENTE SUR UNE CARTE MARINE DANS LA RADE DE TOULON

Ce module de cartographie exploite ainsi des données au format vectoriel, en effet, les cartes sont définies comme une liste de polygones et géométries géoréférencées représentant les côtes, les îles, les lacs ou les rivières. La plupart des méthodes utilisées au sein de *p_s57_processor* utilise donc des outils de manipulation d'objets vectoriels tels que des intersections géométriques. Ces calculs sont opérés à l'aide de la librairie nommée *boost* C++ [4] qui offre ces outils.

Deux formats de cartes marines numériques vectorielles (ENC) utilisés

Actuellement, le système MDT prend en compte deux formats de cartes marines et accorde la priorité pour l'utilisation au premier d'entre eux.

Format S-57 (S57)

FIGURE 26 : EXEMPLE D'UNE CARTE DES COTES DE CAVALAIRE-SUR-MER (83) AU FORMAT S-57

- Cartes formées de polygones groupés par régions
- Dispose de plusieurs niveaux de zoom et une large couverture
- Cartes régulièrement misent à jour par le SHOM

Format shape file (SHP)

FIGURE 27 : EXEMPLE D'UNE CARTE DES COTES DE CAVALAIRE-SUR-MER (83) AU FORMAT SHP

- Cartes formées d'un assemblage de polygones partitionnés, très précises mais la couverture y est plus limitée

Recherche d'optimisation du service de distance de l'horizon

Période de rafraichissement

La période de rafraichissement de *p_detect_ir_cnn* est d'environ 4 Hz, soit une période de 250 ms. Il ne faut donc pas que le total du temps passé par la machine à faire tourner les processus du système dépasse ce seuil de 250 ms.

D'après le code existant, l'appel de ce service peut être effectué plusieurs fois consécutives à chaque nouvelle image. Il est donc primordial que le temps de réponse de ce service soit très court pour ne pas ralentir le système, puisque si ce service est trop long, le système ne parviendra pas à maintenir la cadence au fil des images. Or ce service manipule des polygones avec l'aide de la librairie C++ *boost* où il effectue des calculs géométriques et des intersections de polygones pour en déduire notamment, la distance entre le navire et la côte d'après une carte, ce qui représente des opération lourdes pour une machine.

Et en effet, le temps passé à résoudre des intersections géométriques s'avère conséquent, il semblerait que le service *get_horizon* du module de cartographie *p_s57_processor* soit relativement long (70 ms) par rapport à ce cadre de 250 ms et puisque ce service est appelé jusqu'à 4 fois consécutives par le nœud principal de détection par caméra infrarouge, *p_detect_ir_cnn.*

Afin de ne pas ralentir le rythme de fonctionnement du système MDT, ce temps de retour du service doit être diminué pour qu'il puisse être appelé à chaque cycle de la caméra IR. L'objectif sera ici de réduire son temps d'opération d'un facteur 10 au minimum.

Identification du processus responsable

La librairie chrono C++ est utile pour chronomètre ponctuellement le temps passé entre deux bornes dans un code. On l'utilise ici dans le but de déterminer globalement l'origine des ralentissement provoqués par le service *get_horizon.* On passe alors en revue l'ensemble des processus du service pour trouver celui qui occupe la majorité du temps passé au sein du programme.

Temps de transit, appel et retour du service *get_horizon*

Temps de transit du service	~ 1 ms

Le temps de transit du service est négligeable.

Temps de calcul de la distance de l'horizon

Temps de calcul de la distance de l'horizon	côtes proches	45 ms
	absence de côtes	70 ms

On remarque que le temps passé à obtenir une distance de l'horizon représente une fraction importante du temps total passé dans le service pouvant atteindre jusqu'à 70 ms, soit la quasi-totalité.

De plus, le temps de calcul de la distance à la côte semble dépendre de cette dernière. Le temps de calcul est maximal lorsque la côte est absente. En effet, on passe 45 ms lorsque la côte se situe à une distance de l'ordre de 500m et 70 ms pour une côte absente dans la direction de l'azimut. A noté que le tracé de rayon trouve sa limite lorsqu'on atteint la limite de chargement des cartes, soit 1500 km tel que configuré. On en déduit de ces résultats que plus la côte est éloignée ou

absente, plus le temps de calcul est long, ce qui concorde avec le fait que la fonction fonctionne sur une méthode de recherche itérative.

Compréhension du service

Le service *get_horizon* de p_s57_processor permet de retourner la distance maximale de l'horizon.

Pour se faire, le service recueille les données du problème qui lui sont transmises, soit, la hauteur de l'observateur, sa position géographique et l'azimut qui représente la direction dans laquelle on souhaite connaitre la distance de l'horizon le plus proche formé soit par la courbure terrestre [3], soit par la terre.

Avec la hauteur, le service calcule la distance théorique de l'horizon vrai visible en l'absence de côtes, cette valeur est donnée par la formule $D = \sqrt{2Rh}$ qui dépend uniquement de la hauteur de l'observateur h.

Dans un second temps, à travers le service *get_horizon* la fonction nommé *computeCoastDistance(),* on vient récupérer les cartes locales misent à disposition par un autre service nommé *chartLoader* qui effectue entre autres le chargement de la *Region Of Interest* (ROI) en tâche de fond depuis la base donnés afin de fournir uniquement les polygones qui nous intéressent, soient, ceux aux alentours du navire. Ensuite la fonction va pouvoir estimer la distance géodésique de l'horizon en exploitant les cartes dont il dispose. Pour cela elle recourt à un tracé de rayon qui vise à déterminer une intersection avec la frontière terrestre le long de l'azimut donné. Si à l'issue de la recherche, une intersection est trouvée, la fonction retourne la distance associée, autrement, elle retourne une valeur infinie. Enfin, les valeurs des deux horizons sont comparées et le service retourne la distance de l'horizon le plus proche.

La source du problème de performance a pu justement être localisée dans cette fonction *computeCoastDistance().*

Profilage du service get_horizon

Les outils de profilage permettent lors de l'exécution d'un logiciel de contrôler la liste des fonctions appelées, le temps passé dans chacune d'elle, l'utilisation des ressources CPU ou l'utilisation mémoire par exemple.

Une des manières de profiler une application revient à déterminer pourquoi le CPU est occupé. Une façon efficace de faire cela est le profilage par échantillonnage : on envoie à une certaine fréquence une interruption au CPU pour récupérer la stack trace, l'adresse en mémoire de l'instruction en cours d'exécution (Program Counter) ainsi que l'adresse de la fonction. Nous utiliserons l'outil perf pour se faire.

Afin de mesurer le comportement du logiciel, il nous faut dans un premier temps préparer un scénario d'utilisation. Dans notre cas un utiliser le banc d'essai programmé précédemment exécutant la fonction à optimiser *computeCoastDistance*. Ensuite on exécute ce scenario en même temps que l'outil perf récolte les données pertinentes. Pour cela on spécifie le PID du benchmark, sur machine Linux, il s'agit d'un code à chaque chiffre qui permet d'identifier un processus en cours d'exécution.

Une fois les données recueillis, on dispose d'une très grande quantité de log générés que nous devrons analyser. Pour faciliter grandement la lecture de ces données d'historique machine, il nous faut les transformer pour les rendre lisibles. Une bonne méthode consiste à réaliser un graphique "*Flame graph*" [5], une sorte d'histogramme de l'ensemble des fonctions exécutées lors du test mais qui tient aussi compte de la fonction qui l'a engendrée. On obtient alors un histogramme a plusieurs niveaux qui ressemble en général à un feu, d'où le nom "*Flame graph*". Cette forme nous permet de cibler plus facilement les fonctions prennent une place relativement importante sur le CPU et de pouvoir investiguer leur source et leurs causes.

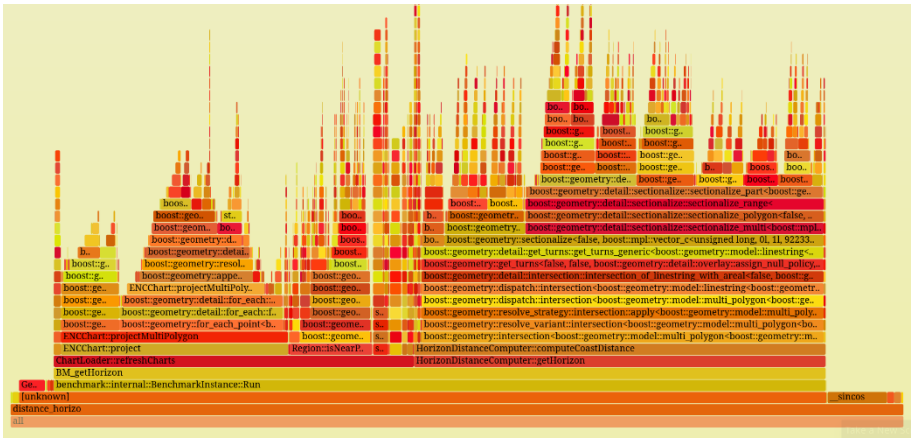


FIGURE 28 : *FLAME GRAPH* DU BENCHMARK DU SERVICE *GET_HORIZON*

Dans notre cas, précisément, on observe rapidement à travers ce graphique que la fonction *getHorizon* prend une place aussi importante que la fonction *refreshCharts*, ce qui alarmant étant donné qu'elle est connue comme étant nécessairement longue puisqu'elle permet de recharger la totalité des cartes seulement une fois tous les 500m parcourus. La fonction en cause, *computeCoastDistance* qui fait intervenir la fonction de la librairie *boost*, *intersection*, semble être directement responsable du temps passé à résoudre le problème de tracé de rayon.

En effet, le problème est posé sous la forme d'une intersection entre le rayon dirigé par l'azimut et le point d'origine spécifié et les côtes terrestres sous forme de polygones dans un espace à deux dimensions. En trouvant le point d'intersection de cette intersection il est alors possible de déduire la distance de la côte. En observant un peu on peut apercevoir que la méthode intersection de la librairie intersection utilise la fonction nommée *get_turns* qui permet justement de localiser le point sur les géométries en question où a lieu l'intersection. Une fois trouvée, les polygones sont sectionnés afin de générer un troisième polygone à l'image de l'intersection géométrique. Ces opérations sont longues notamment parce que les polygones sont complexes et que *boost* cherche à retourner un polygone produit par cette intersection avec une très grande précision. En l'occurrence, ce n'est pas la précision que nous recherchons ici, ni la géométrie qui en résulte mais la performance.

Après recherche, il a été conclu que *boost,* librairie pertinente pour manipuler des objets vectoriels, ne proposait pas de méthode adaptée et optimisée pour effectuer un problème de tracé de rayon comme celui-ci. Cependant, il existe une fonction nommée *simplify* [6]*,* appartenant à *boost*, qui permet de réduire le nombre qui définissent une géométrie, en réduisant le nombre de point on amoindrit automatiquement la complexité du calcul et peut espérer, de cette manière, améliorer les performances du système.

Changement de méthode de calcul (format matriciel des cartes)

Tracé de rayon sur une carte rastérisée

Après de nombreuses tentatives infructueuses pour réduire le temps de calcul du problème de tracé de rayon avec des outils vectoriels sur des objets géométriques (voir les annexes Benchmark du service get_horizon et Simplification des cartes), il est devenu indispensable de réfléchir à une méthode complètement différente. Une méthode qui s'oppose au principe de représentation vectorielle, la manipulation de rasters, autrement dit, la résolution de tracé de rayon par des outils matriciels.

Une tout autre solution envisageable mais qui nécessite une refonte du service consiste à réaliser un tracé de rayon sur une carte au format matricielle telle qu'une image monochrome. Cette image peut justement être générée par un outil de

p_s57_processor nommé *CoastMapComputer* alors la carte des côtes encadrée par la ROI (*Region Of Interest*) du navire avec seulement deux couleurs, noir pour la mer, blanc pour la terre.

L'outil *LineIterator* [7] de la librairie open CV permet d'effectuer une itération sur un segment formé par deux pixels sur une image. Cet outil peut nous permettre de réaliser un tracé de rayon sur la carte avec une précision sur la mesure de l'ordre du pixel, suffisante pour notre cas.

La rastérisation de la carte, c'est à dire, le passage du format vectoriel au format matriciel de la carte locale consiste à transformer les polygones correspondants aux formes de la terre en formes matricielles puis les projeter dans une matrice afin de donner une image en nuances de gris des côtes. La méthode nécessite un certain temps de calcul, mais puisque les opérations seraient effectuées en tâche de fond, lorsque les cartes locales sont rafraichies par le processus qui effectue le chargement des polygones de la ROI, elles n'ront donc pas le service de calcul de la distance à l'horizon. Ainsi on gagne en performance car une fois les données chargées, pour une précision raisonnable d'un mètre par pixel, le tracé de rayon matriciel est bien plus rapide que le vectoriel.

computeCoastDistance() avec méthode raster

computeCoastDistance() avec méthode raster

```
double HorizonDistanceComputer::computeCoastDistance(double _lat, double _lon,
                                                     double _azimut)
{
    // On récupère la carte
    ChartLoader::CostmapData landCstMapData = m_chartLoader->getLandCostMap();
    cv::Mat img = landCstMapData.costmap;

    double delta_x_from_ref_m, delta_y_from_ref_m, delta_z_from_ref;
    // On récupère la distance qui sépare le navire du point de référence de la carte
    m_chartLoader->getGeoRef().Forward(_lat, _lon, 0, delta_x_from_ref_m,
                                       delta_y_from_ref_m, delta_z_from_ref);
    // Init the origin point of the ray we wish to follow
    int x0 = floor(img.cols / 2 + delta_x_from_ref_point_m / imageResolution_m_pix);
    int y0 = floor(img.rows / 2 - delta_y_from_ref_point_m / imageResolution_m_pix);
    // Init the end point of the ray we wish to follow
    int x1 = x0 + floor((img.cols / 2 - .5) * cos(azimut * M_PI / 180.0));
    int y1 = y0 + floor((img.rows / 2 - .5) * sin(-azimut * M_PI / 180.0));
    // Create a iterator to travel the ray pixel after pixel
    cv::LineIterator it(img, cv::Point(x0, y0), cv::Point(x1, y1), 8);
    // Looking for the coast along the considered ray
    for(int i = 0; i < it.count; i++, ++it)
    {
        if(**it == m_searching_value)
        {
                return hypot(-(it.pos().y - y0) * imageResolution_m_pix , (it.pos().x - x0) *
imageResolution_m_pix);
    }
    return std::numeric_limits<double>::max();
}
```

Temps de calcul	0.012 ms

La méthode consiste à récupérer la carte des côtes en tant qu'objet matriciel classique, soit une matrice Mat de la librairie open CV. Cette même librairie propose par ailleurs l'outil *LineIterator* qui permet de créer un itérateur progressant sur une ligne formée par deux pixels. Le premier pixel correspondra donc à la position du navire sur la carte et le second est le point extrême donné par l'azimut et la portée maximale du tracé de rayon. Il est également possible de choisir le degré de connectivité des pixels de la ligne formée, on choisira un degré 8 pour se déplacer sur les pixels diagonaux.

Par la suite, il suffit d'itérer sur cet objet jusqu'à y détecter la côte, autrement, la portée n'est pas suffisamment élevée pour y atteindre le prochain morceau de terre et sa distance est alors supposée infinie.

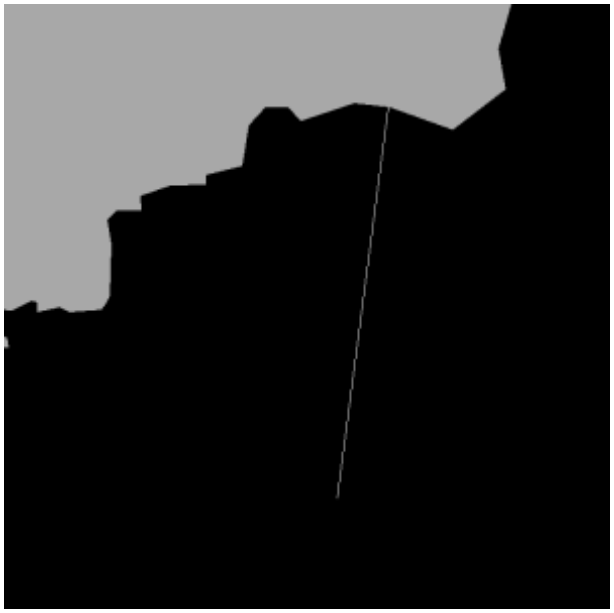


FIGURE 29 : REPRESENTATION D'UN TRACE DE RAYON EFFECTUE AVEC LA METHODE RASTER

Exemple de tracé de rayon, depuis un point au large, le long d'un azimut donné

Bien qu'elle nécessite un changement de format des données à traiter, cette méthode s'avère bien plus performante pour effectuer un tracer de rayon et déduire une intersection géométrique que la méthode de calcul vectorielle, le gain représenté est de l'ordre d'un facteur ×5000. En effet, alors que la méthode d'origine parvenait à obtenir un résultat à en 70 ms, la nouvelle nécessite un temps quasiment négligeable de 0.012 ms en moyenne sur un grand nombre de tracé de rayon dans des configurations variées.

Développement annexe : Détermination du taux de couverture des blobs radar par des outils raster

Puisque l'optimisation du service de distance à l'horizon s'est avérée conséquente et que le gain sur le temps de réponse est loin d'être négligeable, la rastérisation a pu être étendu à d'autres services comme *getLandCoverageRatio,* un service qui permet de calculer le taux de couverture des blobs radar avec la terre afin de filtrer ces détections indésirées. En effet, ce service utilisait auparavant la librairie *boost* pour réaliser des calculs d'intersection géométrique qui peuvent aussi être effectués par la librairie open CV sur des données matricielles. Le temps de retour de ce service est dorénavant nettement meilleur (de ~5 s à moins de 10 ms).

Devant cette amélioration des services de calculs qui exploitent la carte local pour fournir des informations spécifiques à l'environnement du navire - *HorzonDistanceComputer, LandCoverageComputer* et *DetectionComputer* qui réalisent respectivement, la distance de l'horizon le plus proche, le taux de couverture des blobs radar avec la terre et la distance de l'obstacle le plus proche pour chaque degrés autour du navire - un service de fonctions de manipulations d'objets matriciels a été introduit pour permettre à l'avenir de réaliser des calculs sur des données rastérisées et la service qui tourne en second plan établi à présent les cartes rastérisées des côtes, des bouées et autres obstacles en mer sur un rayon de 3 km autour du navire. De nouvelles cartes (de polygones et de grilles d'occupation) sont redéfinies lorsque le navire dépasse un rayon de 500m autour du point de référence.

Intégration de la distance à la côte dans le système de localisation et de suivi des cibles en mer

Le service *get_horizon* du module *p_s57_processor* étant largement optimisé, il est désormais utilisable à fréquence régulière (4Hz) et il est par la même occasion possible d'obtenir la distance de la côte depuis la position de l'observateur à chaque fois qu'une nouvelle image IR est traité par le système MDT.

Range et géodésique

Notons que le service *get_horizon* retourne la distance de l'horizon le plus proche, que ce soit le vrai horizon ou tout autrement le trait de côte. Cependant, les grandeurs de ces deux horizons ne sont pas exactement les mêmes : pour le premier, il s'agit du range, soit la distance la plus courte, aérienne, qui relie l'observateur à l'horizon, pour le second, il s'agit de la distance géodésique à la surface du globe terrestre. Ces deux données doivent être relativement proches étant données la configuration du système (horizon proche, hauteur relativement faible) mais il est indispensable d'estimer à quel point elles divergent. Si, au vu des paramètres du problème, ces deux grandeurs sont sensiblement équivalentes, on sera épargné d'une transformation mathématique.

Pour cela on reprend les expressions déterminées précédemment, mettant en relation le *range* et la géodésique,

d'une part on a exprimé selon les données de *range* et d'élévation de l'horizon :

$$y = (R+h) + \rho_h \sin(b_h)$$
$$b_h = \sin^{-1}\left(\frac{R^2-(R+h)^2-\rho_h^2}{2(R+h)\rho_h}\right) = sin^{-1}(-\frac{2(R+h)h+\rho_h^2}{2(R+h)\rho_h})$$
$$y = (R+h) - \frac{2(R+h)h+\rho_h^2}{2(R+h)} = R - \frac{\rho_h^2}{2(R+h)}$$

d'autre part on a y exprimé selon la distance géodésique de l'horizon :

$$y = R\cos(\Delta) = R\cos(\frac{d}{R})$$

On obtient donc la relation suivante entre ces deux équations :

$$R - \frac{\rho_h^2}{2(R+h)} = R\cos(\frac{d}{R})$$
$$\rho_h = \sqrt{2R(R+h)(1-\cos(\frac{d}{R}))} = \sqrt{4R(R+h)\sin^2(\frac{d}{2R})}$$

Or, dans notre cas, l'horizon terrestre est supposé relativement proche de l'observateur et dans ce cas, sa distance géodésique d devient facilement négligeable par rapport à la circonférence de la Terre d'environ 40 000 km et l'angle $\Delta = \frac{d}{R}$ peut alors être considéré comme très petit. Ce qui nous amène approximer $\frac{\Delta}{2} \approx \sin(\frac{\Delta}{2})$:

$$\rho_h \approx \sqrt{(R+h)\frac{d^2}{R}}$$

Par ailleurs, la hauteur h est aussi très largement négligeable devant la grandeur du rayon terrestre R, ce qui nous donne :

$$\rho_h \approx d$$

On en déduit que le *range* de l'horizon ρ_h peut être aisément approximé par sa propre distance géodésique.

Élévation

Cela nous amène ensuite à exprimer l'élévation b_h de l'horizon en fonction de son *range*.

Puisque la hauteur de l'observateur est négligeable devant le rayon de la Terre, on déduit une expression approximée de l'élévation :

$$b_h = \sin^{-1}\left(\frac{R^2-(R+h)^2-\rho_h^2}{2(R+h)\rho_h}\right)$$
$$b_h = \sin^{-1}\left(-\frac{2Rh+h^2+\rho_h^2}{2(R+h)\rho_h}\right)$$
$$b_h \approx -\sin^{-1}\left(\frac{h}{\rho_h}+\frac{\rho_h}{2(R+h)}\right)$$

Une fois l'élévation estimée, cette grandeur nous permet ensuite de de déduire l'élévation b de la cible observée :

$$b = b_h - \gamma$$

Avec γ estimé à partir des paramètres de la caméra et de l'écart en pixel observé sur l'image :

$$\gamma = \frac{\Delta_v V_{fov}}{V_{pix}}$$

Δ_v, l'écart en pixel sur l'image entre la cible et l'horizon, V_{fov}, les angles FOV de la caméra et V_{pix}, les dimensions en pixel de l'image

Enrichissement du réseau de neurones à convolution de segmentation de l'horizon

Présentation du modèle de réseau U-Net

U-Net est un réseau neuronal convolutif qui a été développé pour la segmentation d'images biomédicales au département d'informatique de l'université de Fribourg [8]. Le réseau est basé sur le *fully convolutional network* et son architecture a été modifiée et étendue afin de travailler avec moins d'images d'entraînement et de produire des segmentations plus précises.

L'idée principale de ce réseau est de compléter un *contracting network* habituel par des couches successives, où les opérations de mise en commun (*pooling*) sont remplacées par des opérateurs de rééchantillonnage. De plus, une couche convolutionnelle successive peut alors apprendre à assembler une sortie précise sur la base de ces informations [8].

Un apport important de U-Net est qu'il y a un grand nombre de canaux de caractéristiques dans la partie de rééchantillonnage, ce qui permet au réseau de propager les informations contextuelles aux couches de plus haute résolution. Par conséquent, le chemin d'expansion est plus ou moins symétrique à la partie de contraction, et donne une architecture en forme de U. Le réseau n'utilise que la partie valide de chaque convolution sans aucune couche entièrement connectée [9]. Pour prédire les pixels de la région frontalière de l'image, le contexte manquant est complété par l'image d'entrée.

Architecture réseau

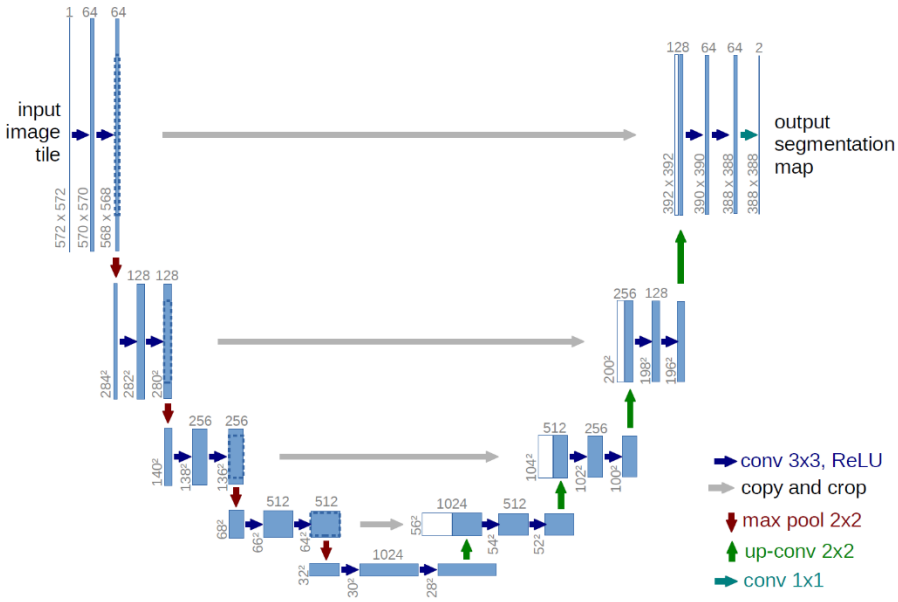


FIGURE 30 : ARCHITECTURE DU RESEAU U-NET

Architecture U-net (exemple pour 32x32 pixels dans la plus basse résolution). Chaque boîte bleue correspond à une carte de caractéristiques multicanaux. Le nombre de canaux est indiqué en haut de la boîte. La taille x-y est indiquée sur le bord inférieur gauche de la boîte. Les boîtes blanches représentent des cartes de caractéristiques copiées. Les flèches indiquent les différentes opérations.

Le réseau se compose d'une voie de contraction et d'une voie d'expansion, ce qui lui donne une architecture en forme de U. Le chemin de contraction est un réseau convolutif typique qui consiste en l'application répétée de convolutions, chacune suivie d'une unité linéaire rectifiée (*ReLU*) et d'une opération de mise en commun maximale. Pendant la contraction, les informations spatiales sont réduites tandis que les informations sur les caractéristiques sont augmentées. La voie d'expansion combine les informations de caractéristiques et spatiales à travers une séquence de convolutions ascendantes et de concaténations avec des fonctionnalités haute résolution provenant de la voie de contraction [10].

Application pour la segmentation de l'horizon du système MDT

Pour rappel, le système MDT (*Marine Detection and Tracking*), est un système permettant de détecter et de localiser les cibles en mer notamment à partir des images de caméra infrarouge dont la portée de la mesure surpasse celles des autres capteurs embarqués tels que les radars, lidars ou AIS. A partir des images infrarouges, le nœud *p_detect_ir_cnn,* consacré à leur traitement, va, dans un premier temps, détecter les cibles en mer et, une par une, les localiser sur l'image avec des boites d'encadrement (*bounding boxes*).

Cette détection s'appuie sur un réseau de neurone convolutif créé pour détecter, encadrer et classifier des objets sur une image. Ce type de réseau est très utilisé pour ce genre d'application car il permet une détection rapide, dynamique et en temps réel d'objets spécifiques sur les images d'une vidéo. En plus de cela, ce réseau encadre ces objets sur les images, ce qui nous permet d'obtenir leur position.

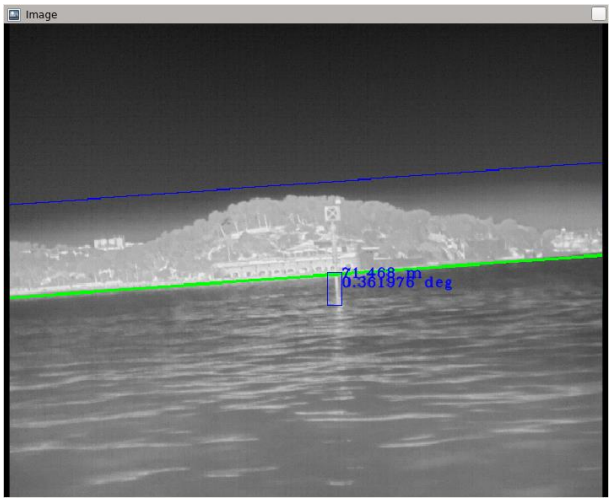


FIGURE 31 : EXEMPLE D'UNE IMAGE IR AVEC SA SEGMENTATION DE L'HORIZON

Une fois les cibles localisées, il nous faut un déterminer le trait de l'horizon sur l'image afin d'obtenir l'écart en pixel entre ces deux entités qui nous permettra ensuite de pouvoir projeter ces cibles dans le repère du navire. Dans le nœud *p_detect_ir_cnn,* on infère un réseau U-Net pré-entrainé pour segmenter de manière précise la séparation entre la mer et le ciel ou la terre. Cette méthode de segmentation est récente et remplace une autre basée su les outils de traitement d'image de la librairie openCV, reconnue mondialement. Le réseau de neurones U-Net s'est montré nettement plus performant que le traitement d'image classique avec openCV, la segmentation étant plus fiable mais aussi plus précise. En effet, tandis que les outils classiques de segmentation openCV permettent de déterminer des formes suffisamment définies sur une image, le réseau de neurones, quant à lui, spécialement entrainé pour résoudre ce problème, c'est à dire, caractériser le trait d'horizon sur l'image.

Cas de la côte proche

Cependant, rappelons-nous que nous cherchions précédemment à améliorer la localisation des cibles en mer dans les situations où la côte, relativement proche, couvre le vrai horizon, situations entrainant une confusion avec le trait de côte, ce qui par conséquent, dérègle la mesure des cibles dans l'espace.

Pour rectifier ce quiproquo, nous avions alors modifier le nœud *p_detect_ir_cnn* effectuant la mesure afin qu'il corrige la confusion grâce aux informations sur la nature de l'horizon détecté et sa distance, données fournies pas le nœud de cartographie *p_s57_processor.* Une fois la modification apportée, il nous a fallu tester cette nouvelle version sur des situations mettant à l'épreuve le système. En recueillant un jeu de données où le navire s'approche progressivement de la côte couvrant le vrai horizon, il s'est révélé que la segmentation de l'horizon sur le trait de côte perdait très largement en fiabilité au fur et à mesure que l'on se rapprochait de la terre. La ligne d'horizon n'est alors plus détectée par le réseau de neurones de segmentation.

Cette défaillance se révèle problématique pour mettre à l'épreuve le nœud en développement *p_detect_ir_cnn* dans cette situation. En effet, si l'horizon formé par la côte n'est pas détecté, impossible de rectifier la mesure.

Ces tentatives de segmentation sans succès s'expliquent notamment par le fait que le réseau est initialement entrainé pour reconnaitre le vrai horizon formé par la mer et le ciel. En ne s'entrainant que sur des jeux de données sans côtes à moins de deux kilomètres, le réseau est logiquement incapable d'inférer correctement sur des images avec la terre proche. Cette situation où l'horizon est confondu avec le trait de côte fut au préalable considéré comme en dehors des cas d'utilisation du système, mais elle s'est présentée de nombreuses fois au cours des essais réalisés et ajoutée une erreur dans la mesure. Il

maintenant nécessaire d'admettre cette confusion non plus comme une erreur mais comme une possibilité. Le trait formé par la mer et la côte sera d'dorénavant considéré comme un horizon sur lequel se focaliser au même titre que l'horizon réel.

Dans cette mesure, pour adapter le système à cette situation il est aujourd'hui indispensable d'enrichir le réseau de cette nouvelle situation en l'entrainant sur des images de ce genre. C'est pourquoi, avec le bateau d'essai de Robopec, nommé "Roboatpec" et embarquant le système MDT fonctionnel, on a pu réaliser des missions dans le but de recueillir de nombreuses images infrarouges avec la présence de la côte. Il est possible d'enregistrer les images de la caméra infrarouge pré-segmentées, c'est à dire que le trait d'horizon à déjà pu être estimé et ses coordonnées en pixel sur l'images sont sauvegardées dans un fichier. JSON afin de labelliser plus facilement les données pour la suite. A la suite de cette mission réalisée avec succès le 4/06/2021, plus de 2500 images pertinentes, montrant des défaillances sur la segmentation ont été retenues pour la labélisation d'un nouveau jeu de données.

Labélisation

La labélisation de données est une étape indispensable du *machine learning* qui consiste à produire un jeu de données contenant les valeurs de référence. Cela revient à associer à chaque image les coordonnées du trait d'horizon correspondant. Ces informations feront office de référence lors de l'apprentissage supervisé. Afin de pouvoir comparer la prédiction avec la réalité, il faut pouvoir connaitre la valeur attendue, ou autrement dit, la *grown truth*.

En effet, lors de l'entrainement, on cherche d'abord à de trouver une relation entre les images et les labels correspondant. On va ensuite mettre à l'épreuve, l'une après l'autre, des fonctions de prédiction généralisées pour s'approcher peu à peu d'une fonction qui minimise les écarts avec un autre jeu de données d'horizons labelisés.

Pour entrainer notre réseau de neurones U-Net, il nous faut construire une base de données suffisamment large et diversifiée, dans laquelle, pour chaque image, on génère :

- L'image originale de la caméra IR
- Un masque de segmentation
- Un masque de pondération

Ces données vont ensuite servir à l'algorithme d'entrainement du réseau de neurones à, d'une part être, conscient de la position sur l'image de l'horizon à prédire et d'autre part, connaitre les zones de l'images qui doivent être misent en avant. L'algorithme se servira alors pour cela, respectivement, du masque de segmentation et du masque de pondération.

Un outil de labellisation d'images infrarouges a été développé à Robopec (pour plus d'information consulter l'annexe Labélisation). J'ai pu l'utiliser et y apporter quelques modifications pour en accroître l'ergonomie. La tâche de labellisation étant longue et l'ergonomie d'un tel outil est très importante.

Entrainement du réseau

Les données labellisées ont pu être ajoutée à la banque d'image d'entrainement, diversifiant un peu plus les types de situations auxquelles le réseau devra adapter sa prédiction. Les données de test du modèle resteront, quant à elles, inchangées si on veut pouvoir comparer les performances de la nouvelle et l'ancienne version.

Entrainer un réseau de neurones complexe nécessite bien souvent une puissance de calcul suffisante pour réaliser les nombreuses opérations de combinaisons matricielles que l'on retrouve notamment dans la fonction d'estimation de coût. Un processeur graphique puissant est d'ailleurs fortement recommandé pour gagner en efficacité et en temps. Robopec a pour cette raison mit en place un ordinateur puissant, nommé "GPU Worker", destiné à ce genre d'utilisation. Au moyen du protocole de communication SSH, il est possible de piloter cet ordinateur à distance et lui faire exécuter le programme d'entrainement.

Matériel / hardware GPU Worker :

- CPU: Ryzen 7 3800X × 8 core
- GPU: Nvidia 2070 Super
- Memory: 32GB

Un entrainement de 1000 *epochs* a donc débuté le 12 juillet sur cette machine et s'est terminé un peu moins de 5 jours plus tard.

A l'aide de l'outil de visualisation TensorBoard, il est possible de consulter les *logs*, autrement dit, les rapports d'entrainement et de tracer les courbes de performance du réseau.

$$IoU = \frac{Aire\ Intersection}{Aire\ Union}$$

L'indice de performance utilisé pour quantifier le succès d'une segmentation est la moyenne des IoU (*Intersection over Union*), un indice largement utilisé pour représenter la justesse d'une segmentation. Cette grandeur se présente comme le rapport de l'aire de l'intersection des *bounding boxes* de l'horizon prédit et labelisé sur leur l'aire de leur union. Un indice proche de 1 indique que la segmentation est précise. Au contraire un résultat proche de 0 signifie qu'elle est peu performante [11].

Dans notre cas, cette moyenne augmente logarithmiquement au fur et à mesure des *epochs* et atteint finalement un résultat très positif car cette moyenne obtenue est sensiblement égale à celle obtenue avec la version antérieure.

Reste à savoir si cette nouvelle version de répartition des poids parvient à s'adapter aux situations de côte proche. Pour vérifier cela, on transforme ces nouveaux poids en un format. ONNX, format accessible par le nœud p_detect_ir_cnn qui infère le réseau de neurone de détection de l'horizon. Après avoir rejoué le système MDT sur le même jeu de données (*rosbag*) qui posait initialement problème au système de segmentation du trait de côte, la détection s'est avérée positive et stable. Maintenant que le réseau est opérationnel, même dans ce genre de situation, il est désormais possible de savoir si la prise en compte de la distance de l'horizon formé par les côtes terrestres améliore ou non la mesure des cibles.

Résultats de détection de cible en situation de côtes proches

Etant donné que l'implémentation de la distance à la côte dans le nœud de détection des cibles en mer dans le but d'améliorer leur mesure de position est théoriquement viable et que le réseau de neurone est d'dorénavant en mesure de détecter le trait de côte comme un horizon, on peut maintenant comparer les performances de cette nouvelle version de *p_detect_ir_cnn* avec l'ancienne version.

On rejoue alors une situation ou le navire Roboatpac équipé du système MDT en faisant tourner dans un premier temps la version d'origine du nœud *p_detect_ir_cnn* dite, *legacy*, puis dans un second temps on relance le même jeu de donnée avec la nouvelle version nommée *devel*. En transformant les données de détection et en affichant la distance réelle de la bouée observée. Pour connaitre la distance réelle de la bouée, il suffit d'estimer les différence entre la position géographique de la bouée et celle du navire avec la méthode *Geodesic::Inverse()* de la librairie *GeographicLib* qui calcul la géodésique entre deux positions GNSS sur le globe terrestre. A noter que lors de cette mission dont on exploite l'enregistrement (*rosbag*), les coordonnées GNSS de la bouée avait été relevée pour, justement, connaitre la réalité terrain (*ground truth*) de cette détection.

Grâce à un nœud ROS réalisé pour l'occasion, on republie les mesures qui nous intéressent et à l'aide de l'outil de visualisation dynamique de données, *PlotJuggler,* on trace les courbes de ces données. Il nous est maintenant possible de comparer les résultats.

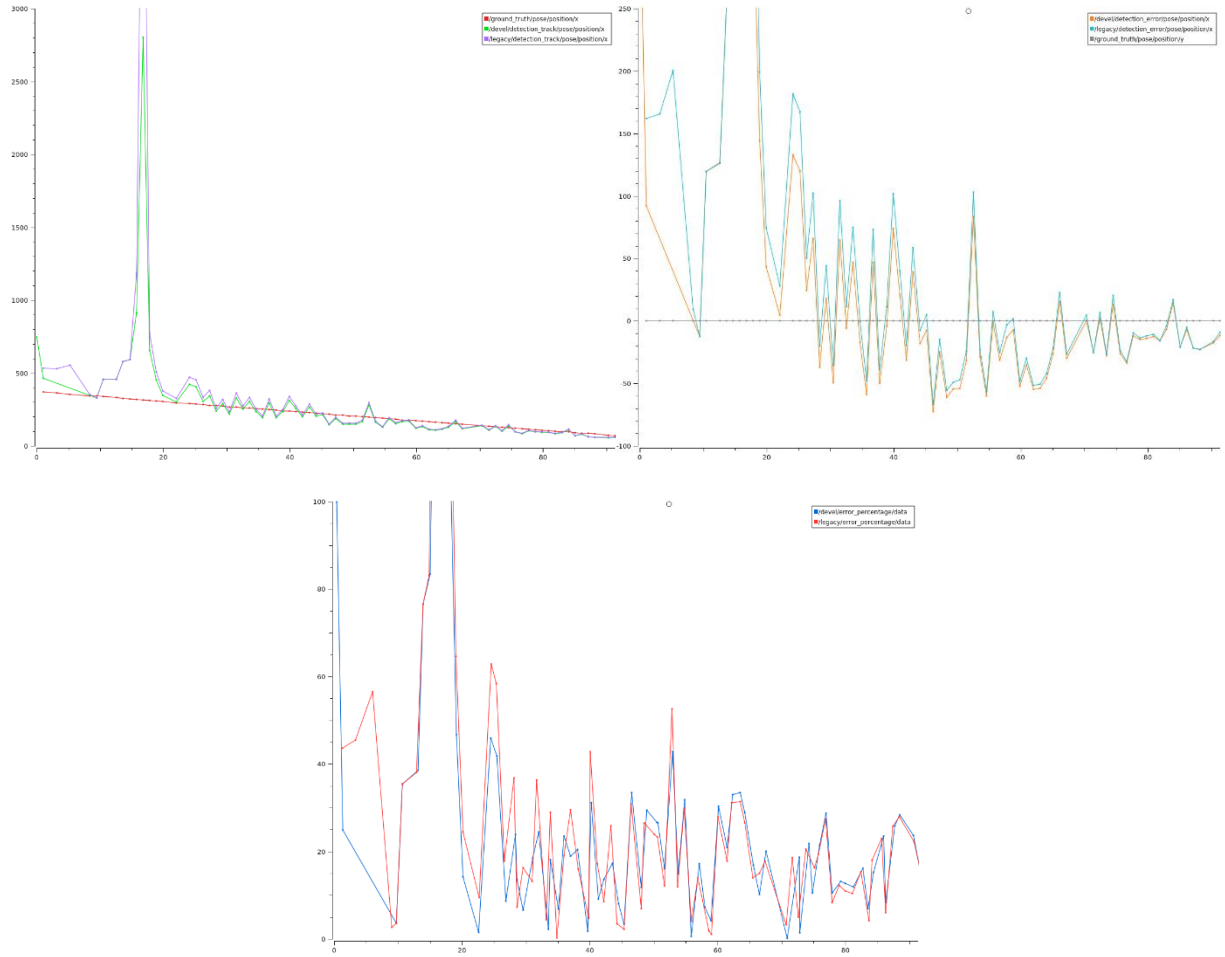


FIGURE 32 : DISTANCE, ERREUR ET POURCENTAGE D'ERREUR DE LA DISTANCE ESTIMEE DE LA CIBLE EN FONCTION DU TEMPS

On réalise ainsi trois graphiques mettant tous en valeur la mesure de la distance de la cible observée (bouée) au cours du temps, sachant que le navire avance droit vers elle.

- Le premier graphique représente la distance réelle (*ground truth*), mesuré avec la version d'origine du système (*legacy*) et la version modifiée (*devel*).
- Le second graphique relève les écarts en distance de ces deux dernières courbes avec la distance réelle.
- Enfin, le dernier graphique montre le pourcentage d'erreur des mesures avec la réalité.

A travers ces courbes, on peut remarquer deux conséquences intéressantes :

- D'une part, la valeur aberrent observée à t=18 s diffère très largement sur les courbes des deux versions du système. Alors que l'actuelle version annonce une distance de la cible de plus de 5000 m, la nouvelle propose une valeur d'environ 2800 m. Cet écart important qui profite à la version en développement s'explique peut-être par le fait que le nœud qui estime la distance doit résoudre une équation du second ordre avec deux solutions. Puisqu'une seule des solutions est valable, le programme retourne la distance de l'horizon à la place de l'autre solution étant donné qu'elle ne peut se trouver au-delà de cette limite du visible.
- D'autre part, de manière générale, les distances annoncées par la nouvelle version du système sont amoindries tout en conservant les mêmes variations. Cette tendance est plutôt bon signe car la mesure du système d'origine avait comme défaut d'estimer des distances assez élevées par rapport à la réalité.

Cependant, cette variante n'est pas sans inconvénient, en effet, le dernier graphique révèle que le pourcentage de l'erreur absolue ne donne pas la nouvelle version gagnante en tous points. Néanmoins, lorsque l'on considère l'erreur de mesure de la distance non absolue, on remarque qu'il ne s'agit sûrement que d'un ajustement d'*offset* (décalage). Ce décalage constant dans la mesure peut être corriger en calibrant les paramètres de montage de la caméra infrarouge tel que la hauteur l'assiette ou le tirant d'eau du navire.

A noter que l’azimut de la cible a pu aussi être observé.

En conclusion, aux vues de ces résultats encourageant en déduit que l'implémentation de la distance à l'horizon observé dans le nœud de détection permettrait de réduire l'erreur de mesure sur la localisation des cibles via la caméra IR, en majorité sur la distance estimée. Reste maintenant à mettre à l'épreuve ce système en conditions réelles sur des situations diverses pour valider la stabilité de cette fonctionnalité avant de pouvoir l'intégrer dans une futur mise à jour du système MDT.

Conclusion

A travers ce PFE, j'ai intégré le développement du système de détection et de localisation de cibles en mer MDT de l'entreprise Robopec. J'ai eu l'occasion non seulement de participer mais aussi de contribuer au projet, en apportant des améliorations et des outils de travail qui permettent de mieux comprendre et appréhender le système. En développant une étude détaillée des causes des erreurs de détection via caméra infra-rouge et leur impact sur la mesure, on a pu mettre en lumière les phénomènes qui engendrent des écarts. Avec la connaissance de ces phénomènes, il a été possible d'envisager des projets d'amélioration du système.

On a notamment pu parler de la prise en compte du pilonnement, rendu réalisable par l'arrivée de cette mesure sur les INS iXbue, qui semble, en partie, régulariser la mesure de la distance des cibles par la caméra infra-rouge.

Dans un autre projet, on a aussi pu corriger l'erreur causée par la confusion de l'horizon avec le trait de côte prenant en compte la distance de ce dernier dans la projection des cibles observées sur l'image. Il nous aura d'abord fallu accepter pleinement ce qui quiproquo en injectant des situations de ce genre dans l'apprentissage du réseau de neurones de segmentation de l'horizon. Puis, en reprenant la théorie du calcul de la distance des cibles, on a pu implémenter le cas où l'horizon est autrement formé que par le ciel et la mer. Ce projet a finalement lui aussi démontrer des résultats plutôt convaincants.

Dans un dernier temps, un ultime projet d'intégration (voir l'annexe Interfaçage MDT et MAVLink ArduPilot via MAVROS) m'a permis de pleinement appréhender MDT en l'interfaçant à un système existant à savoir AutoPilot et d'apporter une preuve de concept démontrant, pour l'avenir, que MDT est tout à fait compatible avec un autre système.

Annexes

Etudes des causes de l'erreur de détection par caméra infra-rouge et quantification de leur impact sur la mesure

Equivalence entre l'erreur causé par le trait de côte et l'erreur en pixel causé par une mauvaise segmentation

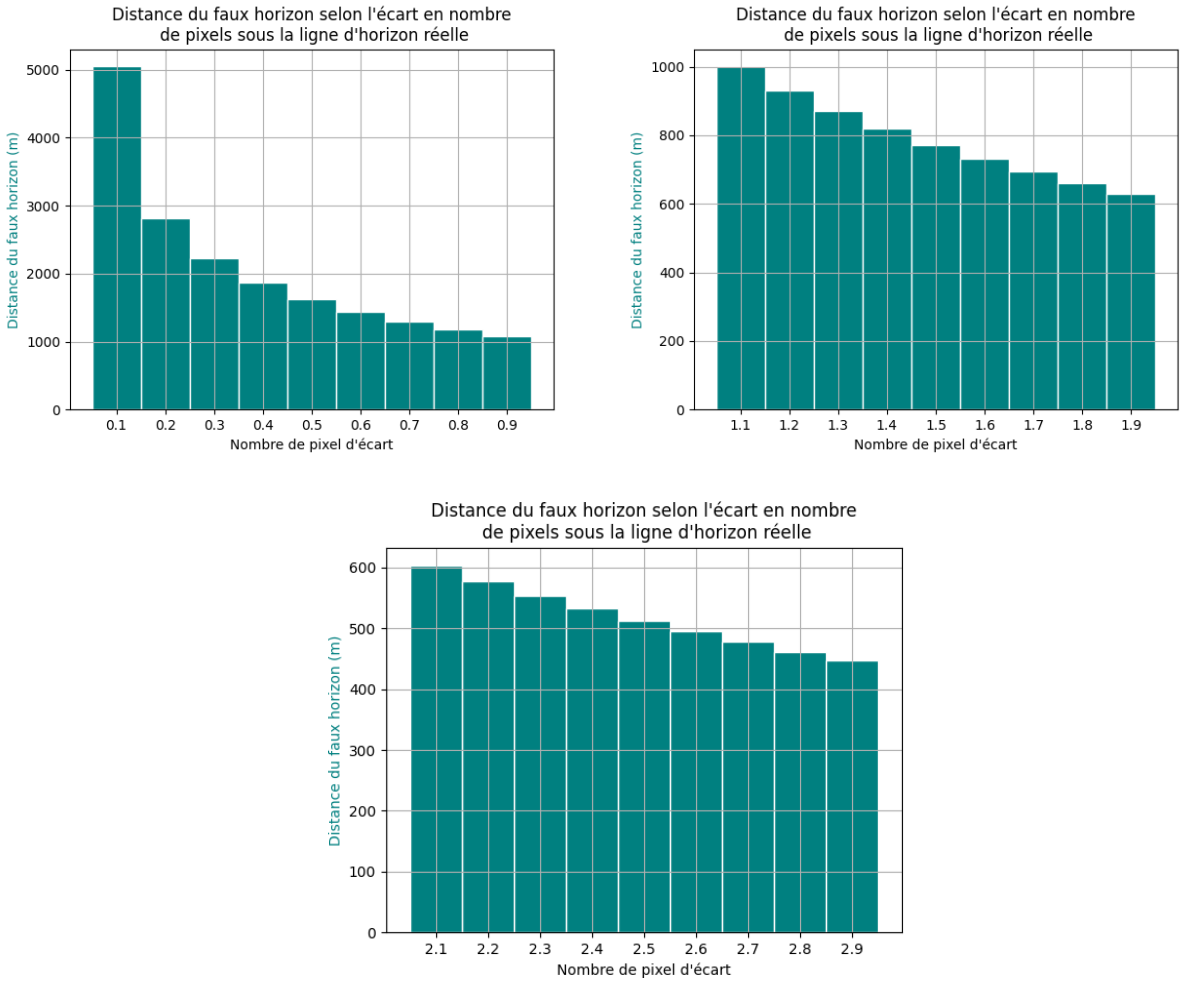


FIGURE 33 : DISTANCE DU FAUX HORIZON SELON L'ECART EN PIXELS SOUS LA LIGNE D'HORIZON REELLE

On peut aisément établir une analogie entre l'erreur causé par le trait de côte et l'erreur en pixel causé par une mauvaise segmentation. C'est pourquoi, on trace la distance de l'horizon erroné en fonction de son écart en pixel sous la ligne d'horizon réelle.

On remarque que dès le premier pixel sous la ligne d'horizon, la distance chute brusquement.

En fonction du nombre de pixels entre l'horizon et la cible

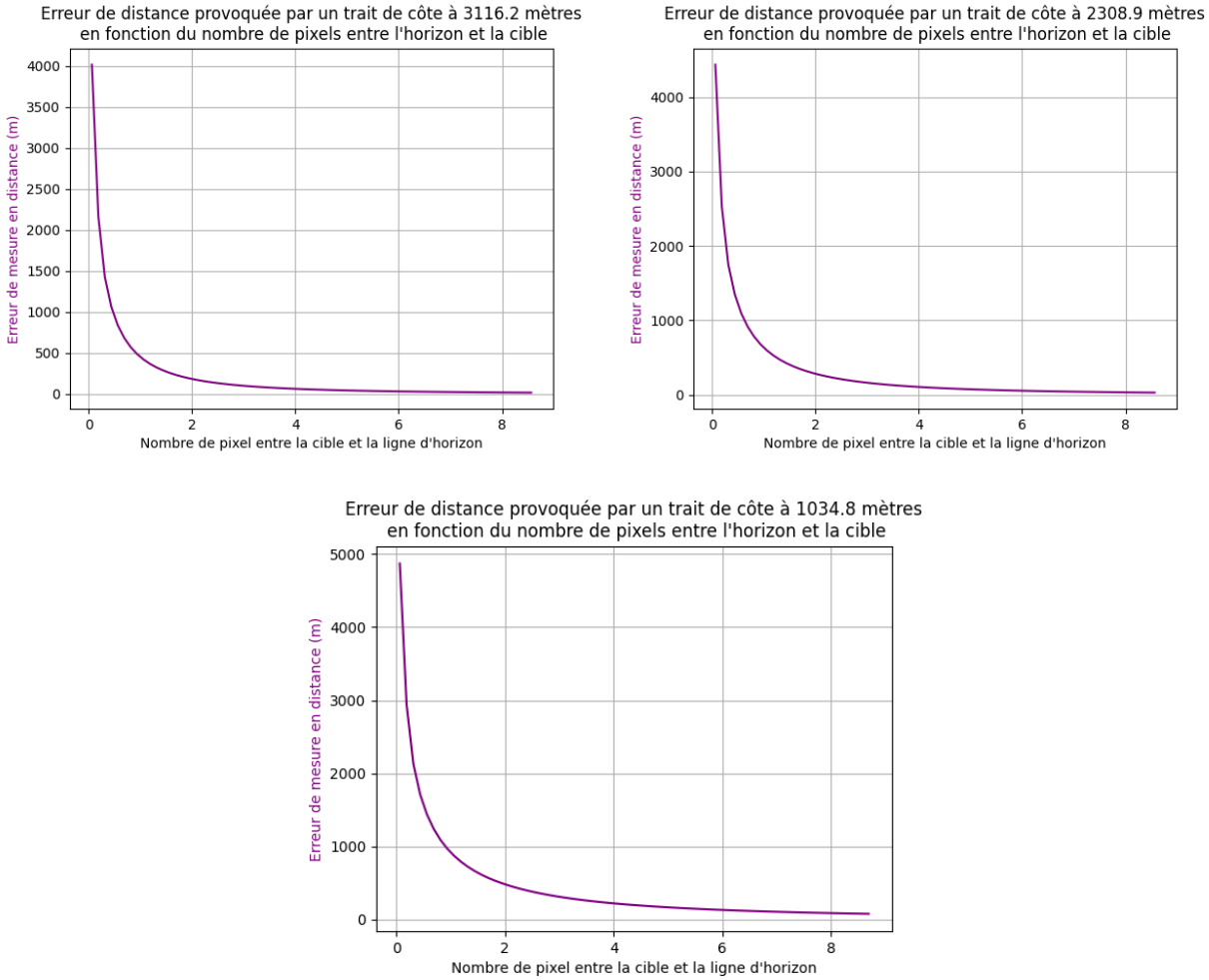


FIGURE 34 : ERREUR DE DISTANCE PROVOQUEE PAR UN TRAIT DE COTE EN FONCTION DU NOMBRE DE PIXELS ENTRE L'HORIZON ET LA CIBLE

Ces trois autres graphiques décrivent l'erreur commise en distance, en fonction du nombre de pixels qui séparent la cible de l'horizon, causée par le trait de côte qui fausse la ligne d'horizon. On y représente les erreurs pour trois distances différentes de l'observateur à la côte.

On remarque que pour un nombre de pixels d'écart faible (correspondant à l'écart entre la cible et l'horizon), l'erreur de mesure liée au trait de côte, reste quasiment équivalente et relativement élevée.

Optimisation du service de distance de l'horizon

Benchmark du service *get_horizon*

Puisque la recherche s'affine et pour gagner en précision sur le chronométrage des temps d'exécution des morceaux de codes traités on préfère utiliser la librairie nommée benchmark.

La librairie benchmark développée par Google rend possible, sur un code, l'estimation le temps moyen passé par une machine dans un processus sur plusieurs itérations. Il est notamment utile lorsque l'on souhaite estimer plus précisément les performances d'un calcul dans une système. Puisque le temps de calcul peut varier d'une itération l'autre, la méthode consiste à chronométrer plusieurs fois consécutives la même fonction jusqu'à obtenir une moyenne stable. En l'occurrence, on souhaite chronométrer le temps de calcul du service *get_horizon.* Pour cela on réalise un code type contenant la fonction à tester qui sera lancée jusqu'à ce que le programme atteigne une moyenne stable du temps calculé.

Voici un le benchmark réalisé :

Benchmark

```
static void BM_getHorizon(benchmark::State& state) {
    // params
    std::shared_ptr<ChartLoader> loader;
    p_s57_processor::S57ProcessorParams params;
    cv::Mat img;
    std::vector<int8_t> grid;
    loader.reset(new ChartLoader("./unit_tests/test_files", "./unit_tests/test_files",
                                 "CATALOG.031", "SHPCatalog.geojson", "./s57data"));
    params.avoid_land = true;
    params.avoid_buoys = true;
    params.avoid_depth = true;

    // On se met près des côtes de Cavalaire
    mdt_msgs::Gps ref;
    ref.latitude = 43.150;
    ref.longitude = 6.500;
    loader->refreshCharts(43.150, 6.500, params);
    HorizonDistanceComputer distanceComputer(loader);
    CostMapComputer costMapComputer(loader);

    costMapComputer.getCostMapImg(43.150, 6.500, 1500, 1500, 2.0, img, params);
    double angle;
    // Les opérations effectuées à l'intérieur de cette boucle for seront
    // passée au benchmark pour estimer le temps moyen de leur execution
    for (auto _ : state)
    {
        state.PauseTiming();
        angle = rand() % 360 - 180;
        state.ResumeTiming();
        distanceComputer.getHorizon(43.150, 6.500, angle, 2.);
    }
}
BENCHMARK(BM_getHorizon)->Unit(benchmark::kMillisecond)->MinTime(2);
BENCHMARK_MAIN();
```

Il est par la même occasion, indispensable de conserver une certaine homogénéité pour chaque test effectué nous permettant de pouvoir ensuite comparer les résultats. Ainsi, nous utiliserons la même machine tout le long de l'analyse. Il s'agit d'un PC portable Linux 64-bit équipé d'un CPU Intel Core 8ème génération i7-8650U cadencé à 1.90GHz et de 16Go de RAM donc suffisamment puissant pour les tâches à réaliser et de faire la moyenne sur un nombre suffisant d'itérations. Plus le temps est court et plus le nombre de chronométrage sera grand pour augmenter en précision. Dans notre cas, on teste les performances des fonctions exécutées lors de l'appel du service *get_horizon.* Pour l'ensemble de cette étude, on prendra le soin qu'aucun autre processus parallèle ne puisse r notablement les performances du CPU.

Hardware information :

- Memory: 15,5GiB
- CPU: Intel® Core™ i7-8650U 1.90GHz × 8

Fonction de tracé de rayon *getCoastDistance()*

Voici un extrait de la fonction qui détermine la distance jusqu'à la côte :

Code original

Code original

```
double HorizonDistanceComputer::computeCoastDistance(double _lat, double _lon, double _azimut)
{
    // Acquire maps
    auto refCharts = m_chartLoader->referencedChartsByScale();

    // On projete la position du bateau dans la ref utilisée pour projeter les cartes
    GeographicLib::LocalCartesian proj(refCharts.refLat, refCharts.refLong, 0);
    double x_os, y_os, z;
    proj.Forward(_lat, _lon, 0., x_os, y_os, z);

    // On trace une ligne entre le bateau et l'horizon dans la direction de l'azimut
    // Pas besoin de tracer une ligne plus grande que l'horizon
    double distance = computeHorizonDistance() + 100;
    double x = x_os + distance*cos(_azimut* M_PI / 180.0);
    double y = y_os + distance*sin(_azimut* M_PI / 180.0);

    tLineString line;
    line.push_back(tPoint(x_os, y_os));
    line.push_back(tPoint(x, y));

    auto it = refCharts.charts.begin();
    while (it != refCharts.charts.end())
    {
        std::vector<tLineString> intersections;

        boost::geometry::intersection(it->second.getCartAreas(ENCChart::areaType::LAND),
                                      line, intersections);
        double minDistance = std::numeric_limits<double>::max();
        for(const tLineString& l : intersections)
        {
            for(const tPoint& point : l)
            {
                double dist = std::hypot(point.x() - x_os, point.y() - y_os);
                minDistance = std::min(dist, minDistance);
            }
        }
        // Si on a trouvé une distance on la retourne
        if(minDistance < std::numeric_limits<double>::max())
        {
            return minDistance;
        }
        it++;
    }
    return std::numeric_limits<double>::max();
}
```

Temps de calcul	70.4 ms

En effet, avec un temps de calcul de l'ordre de 70 ms, il semblerait que cette fonction soit responsable de l'ensemble du temps passé dans le service.

Dans cette fonction, on utilise des méthodes de la libraire *boost* open source C++, adaptée aux calculs faisant intervenir des objets géométriques tels que des polygones, des lignes ou même des cercles. Etant donné que les cartes électroniques marines sont décrites en formats vectoriels, l'utilisation de la librairie *boost* pour effectuer des opérations telles que des intersections est tout à fait pertinente. C'est pourquoi les opérations géométriques du nœud *p_s57_processor* sont en majorité effectués à l'aide des outils de *boost geometry*.

Temps d'acquisition de la carte au sein du service

Dans un premier temps, on détermine le temps correspondant à l'acquisition des cartes sous forme de polygones.

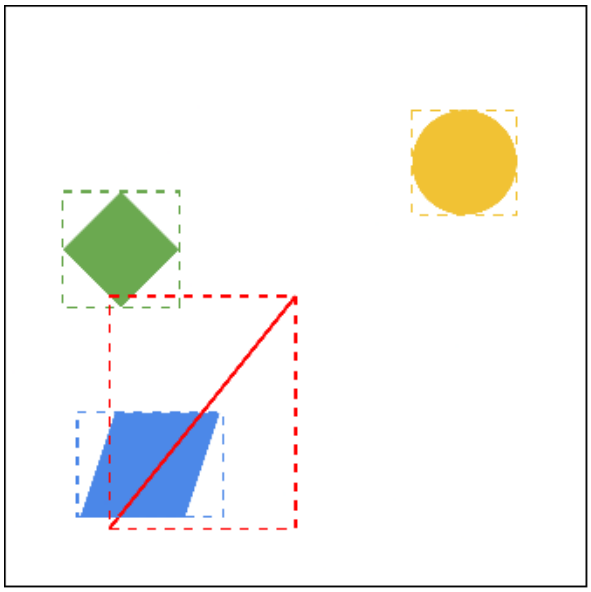
Temps d'acquisition de la carte	< 1 ms

Ce temps est négligeable par rapport aux 70 ms que représente le service *get_horizon.* Le temps de transite des données au client du service n'est pas la source du problème ce qui confirme que c'est bien le tracé de rayon qui mobilise la plupart du temps passé.

En cherchant, il s'est révélé que l'intersection géométrique qui y est effectué à l'aide de l'outil intersection () de la librairie *boost* C++ est assez longue pour notre cas, bien que la librairie soit justement optimisée pour manipuler efficacement des géométries. Puisqu'on utilise cette méthode qui est ici répétées autant de fois que le nombre de polygones à traiter, le temps total de calcul devient rapidement conséquent.

Pour réduire suffisamment ce temps de calcul afin d'arriver à un délai correct, plusieurs méthodes ont alors été envisagées.

Filtrage des polygones dont la *bounding box* intersecte pas avec la *bounding box* de la ligne de l'azimut



Filtrage des polygones dont la *bounding box* intersecte avec la *bounding box* de la ligne de l'azimut

Filtrage des polygones dont la bounding box intersecte avec la bounding box de la ligne de l'azimuth

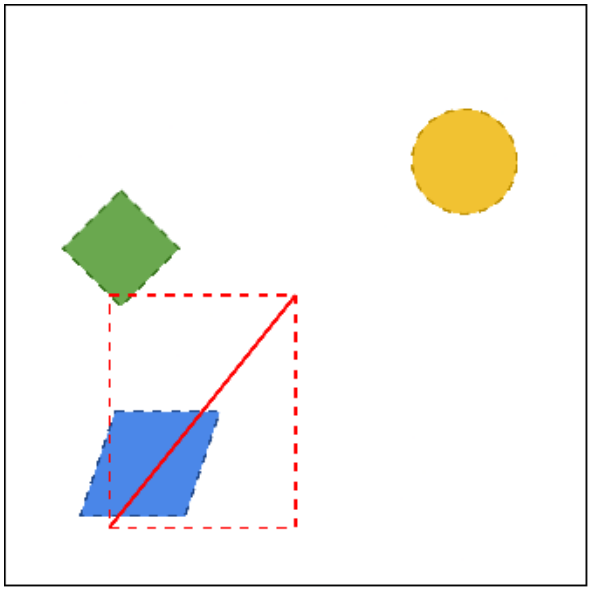
```
auto it = refCharts.charts.begin();
while (it != refCharts.charts.end())
{
    std::vector<tLineString> intersections;
    auto areas = it->second.getCartAreas(ENCChart::areaType::LAND);
    double minDistance = std::numeric_limits<double>::max();
    for(auto polygon: areas) {
        boost::geometry::model::box<boost::geometry::model::d2::point_xy<double>> box;
        boost::geometry::model::box<boost::geometry::model::d2::point_xy<double>> line_box;
        boost::geometry::envelope(polygon, box);
        boost::geometry::envelope(polygon, line_box);
        if (boost::geometry::intersects(box, line_box))
        {
            boost::geometry::intersection(polygon, line, intersections);
            for(const tLineString &l: intersections) {
                for(const tPoint &point: l) {
                    double dist =  std::hypot(point.x()-x_os, point.y()-y_os);
                            minDistance = std::min(dist, minDistance);
                                    }
                            }
        }
    }
    // Si on a trouvé une distance on la retourne
    if(minDistance < std::numeric_limits<double>::max())
    {
        return minDistance;
    }
    it++;
}
return std::numeric_limits<double>::max();
```

Temps de calcul	92.5 ms

D'abord, les polygones définissant les côtes ont été triés pour ne garder que ceux susceptibles d'être utilisés. Pour cela on encadre les polygones par un polygone simple rectangulaire nommé *bounding box*, littéralement, boîte d'encadrement et on retient les polygones dont cette boîte intersecte avec celle du rayon en question. On effectue par la suite le tracé de rayon telle la manière originale mais cette fois ci le nombre de polygones a été réduit.

Malheureusement, ici encore la performance est moins bonne que l'originale, elle aura plutôt comme conséquence de ralentir d'avantage le système en tentant de réduire sans grand succès le nombre de polygone.

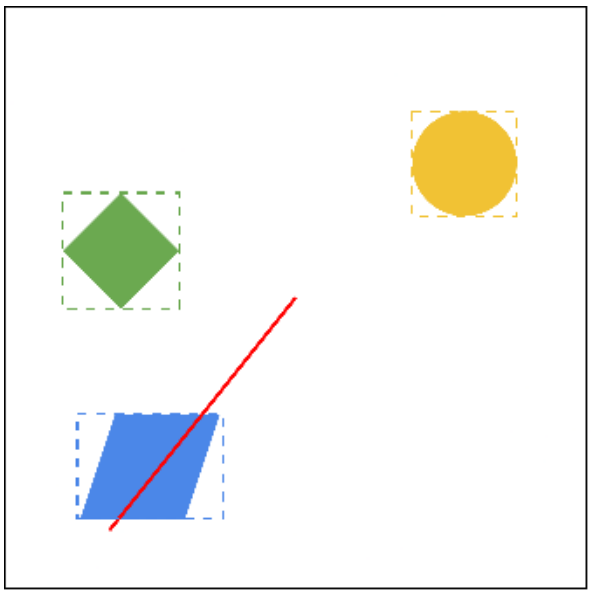
Filtrage des polygones qui n'intersectent pas avec la *bounding box* de la ligne d'azimut



Temps de calcul	71.4 ms

Dans un second temps, un cherche à ne retenir que les polygones qui intersectes avec la *bounding box* de la ligne d'azimut. Mais ce filtrage ne semble pas avoir d'effet positif sur le calcul puisque le temps des opérations reste quasiment inchangé.

Filtrage des polygones dont la *bounding box* intersecte pas avec la ligne de l'azimut



Filtrage des polygones dont la *bounding box* intersecte avec la ligne de l'azimut

Filtrage des polygones dont la bounding box intersecte avec la ligne de l'azimuth

```
auto it = refCharts.charts.begin();
while (it != refCharts.charts.end())
{
    std::vector<tLineString> intersections;
    auto areas = it->second.getCartAreas(ENCChart::areaType::LAND);
    double minDistance = std::numeric_limits<double>::max();
    for(auto polygon: areas) {
        boost::geometry::model::box<boost::geometry::model::d2::point_xy<double>> box;
        boost::geometry::envelope(polygon, box);
        if (boost::geometry::intersects(box, line))
        {
            boost::geometry::intersection(polygon.outer(), line, intersections);
            for(const tLineString &l: intersections) {
                for(const tPoint &point: l) {
                    double dist =  std::hypot(point.x()-x_os, point.y()-y_os);
                            minDistance = std::min(dist, minDistance);
                                    }
                            }
        }
    }
    // Si on a trouvé une distance on la retourne
    if(minDistance < std::numeric_limits<double>::max())
    {
        return minDistance;
    }
    it++;
}
return std::numeric_limits<double>::max();
```

Temps de calcul	55.3 ms

Enfin, la dernière stratégie ici est de garder les polygones dont la boîte d'encadrement chevauche le rayon donné.

Cette solution apporte un gain de temps d'environ 20%, ce qui nous offre un temps de réponse diminué mais toujours pas acceptable. Nous voudrions que le temps de calcul direct passe en dessus des 10 ms.

Simplification des cartes

La fonction *simplify* de la librairie *boost* qui permet de réduire le nombre de points qui définit une géométrie sans réduire le nombre de géométries.

Pour cela, la fonction a besoin d'un paramètre de précision qui est déterminant pour reconstruire un nouveau polygone contenant potentiellement moins de points. Dans notre cas, on peut accepter une précision de 1 mètre, ce qui se traduit dans la méthode par une distance maximale entre chaque point d'au maximum 1 afin d'y éjecter si possible les points trop rapprochés et non nécessaires qui définissent les géométries. On utilise alors cette fonction pour simplifier les polygones qui représentent les côtes terrestres et ensuite pouvoir observer si cela apporte ou non une amélioration sur les performances du système.

Dans un premier temps, il est nécessaire de constater si le nombre de points qui définissaient les polygones des côtes a bien diminué et dans ce cas cela se traduirait par une potentielle optimisation de l'ensemble du module de cartographie *p_s57_processor*.

Nombre de polygones	Nombre de points	
Multipolygones brut ou simplifiés (puisque le nombre est inchangé)	Multipolygone brut	Multipolygone simplifié à 1 m
3	2099	2028
43	11509	11472
40	8728	8570
21	7432	7419

Dans cette figure suivante, on observe que pour une échelle de simplification à 1 mètre, le gain nombre de points qui définissent les polygones des terres est très faible. Pour carte se basant sur 43 polygones par exemple, eux même définies par un total de 11509 points, la simplification a permis de diminuer le nombre de points de seulement 0.3%, ce qui est de toute évidence négligeable. Pour 3 polygones de 2099 points au total, le gain est de 3.4% mais la conclusion reste la même. Une telle simplification ne permettra pas d'obtenir une amélioration nette des performances de calcul, pour cela il aurait fallu espérer une réduction de l'ordre de 50%.

Simplification des données

Filtrage des polygones qui n'intersectent pas avec la ligne de l'azimut

Filtrage des polygones qui n'intersectent pas avec la ligne de l'azimuth

```
auto it = refCharts.charts.begin();
while (it != refCharts.charts.end())
{
    std::vector<tLineString> intersections;
    auto areas = it->second.getCartAreas(ENCChart::areaType::LAND);
    tMultiPolygon simplified;
    boost::geometry::simplify(areas, simplified, 1000);
    double minDistance = std::numeric_limits<double>::max();
    for(auto polygon : simplified) {
        if (boost::geometry::intersects(polygon, line))
        {
            boost::geometry::intersection(polygon, line, intersections);
            for(const tLineString &l: intersections) {
                for(const tPoint &point: l) {
                    double dist =  std::hypot(point.x()-x_os, point.y()-y_os);
                    minDistance = std::min(dist, minDistance);
                }
            }
        }
    }
    // Si on a trouvé une distance on la retourne
    if(minDistance < std::numeric_limits<double>::max()) {
        return minDistance;
    }
    it++;
}
return std::numeric_limits<double>::max();
```

En implémentant cette méthode au sein de la fonction qui réalise le tracé de rayon il devient possible de passer cette solution sur le *benchmark* afin de comparer les résultats.

Temps de calcul	simplification en direct	> 500 ms
	simplification en amont	70 ms

La conclusion est sans appel, comme attendu la simplification à 1 mètre n'apporte aucune nette amélioration. Elle retarde même le système si la méthode *simplify* est utilisé pour directement les polygones dont un cherche l'intersection.

Labélisation

L'outil qui permet de labeliser graphiquement les images une par une et d'enregistrer les coordonnées de l'horizon renseigné a été développé par Robopec. Il s'agit de MDT Horizon Labeling. Cet outil se présente comme une interface sur laquelle on peut parcourir une banque d'images IR brutes et y modifier directement la segmentation de l'horizon sur chaque image. Avec le pointeur de la souris on rejoint une position sur l'image, clic gauche pour définir un premier point de passage de la ligne d'horizon, clic droit pour le second. Lorsqu'on enregistre le programme déduit de cette ligne les hauteurs en pixel de chaque extrémité définissant les coordonnées de l'horizon.

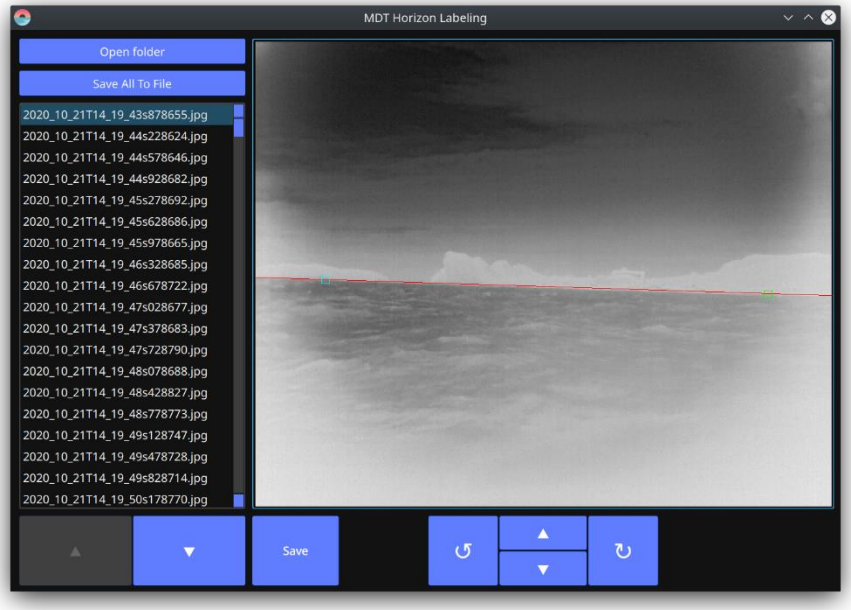


FIGURE 35 : INTERFACE DE LABELISATION DES IMAGES IR

La labélisation est une tâche manuelle qui nécessite du temps et de la patiente. Occasionnellement, certain trait d'horizon pré-segmenté par le système MDT ne s'écarte que très légèrement de l'horizon tel qu'il est perçu par l'utilisateur et là un décalage de quelques pixels suffirait, un nouveau dessin compet de la ligne à la souris est dispensable. C'est la raison pour laquelle, quelques modifications ont été apporté à l'interface basée Qt afin de proposer quatre boutons supplémentaires à l'interface, permettant de déplacer le trait d'horizon pixel par pixel avec la possibilité d'effectuer une translation verticale ou un ajustement du roulis. Cette modification a permis de gagner du temps pour labeliser un total de plus de 2500 nouvelles images d'horizons formés par les côtes.

Interfaçage MDT et MAVLink ArduPilot via MAVROS

En fin de PFE, pour apporter un dernier travail concret sur le système MDT, le dernier objectif consistait à interfacer le système avec un auto-pilote bien connu, ArduPilot [12]. ArduPilot est un logiciel de pilotage automatique de véhicules sans pilote capable de contrôler de manière autonome. La mission ici consiste à reprendre le simulateur MDT développé par Robopec et d'y substituer la partie qui gère la physique du navire avec le SITL (*Systeme In The Loop*) d'ArduPilot gérant le guidage et le contrôle du véhicule. La partie navigation elle reste à la charge du système MDT afin d'éviter les potentiels obstacles durant la mission. Le simulateur actuel dispose d'une interface sur laquelle il est possible de programmer divers scénarios (traversée d'une île, croisement d'un obstacle en mouvement, etc.). Lors de la lecture d'un scénario, des détections virtuelles vont alors être envoyées pour voir comment le système s'adapte à ces situations et s'il parvient a éviter la collision à temps.

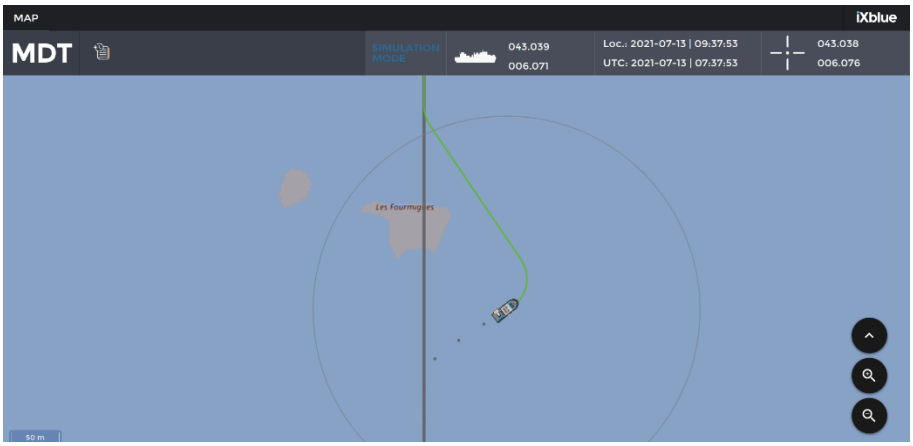


FIGURE 36 : INTERFACE DU SIMULATEUR MDT

Après un travail de développement, la tâche a pu être accomplie avec l'aide d'un projet open-source nommé MAVROS permettant d'interfacer un système de communication MAVLink tel que ArduPilot avec ROS. Il est alors possible d'accéder aux informations essentielles telles que la géolocalisation et l'orientation du système mais aussi de transmettre des ordres de mission pour que le véhicule rejoigne des positions de manière autonome.

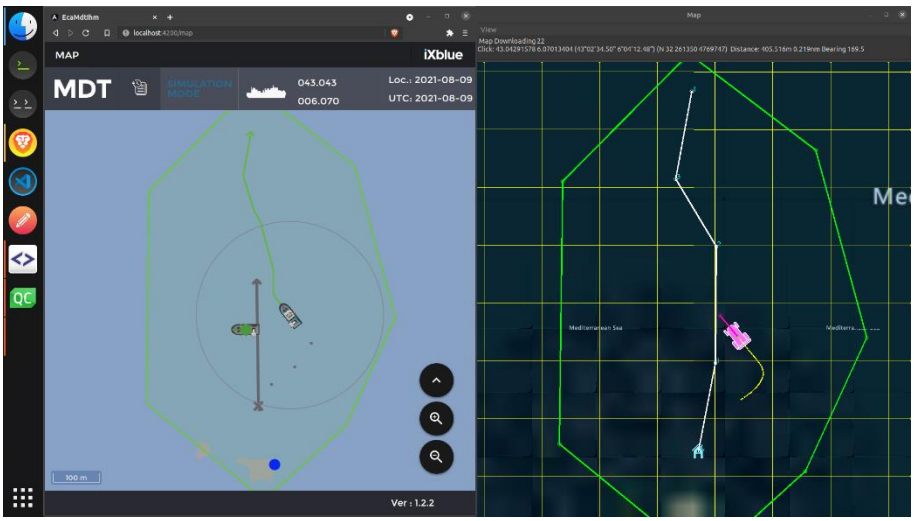


FIGURE 37 : MDT ET ARDUPILOT FONCTIONNANT DE PAIR POUR SUIVRE UNE TRAJECTOIRE DEFINIE TOUT EN EVITANT DYNAMIQUEMENT UN OBSTACLE

Sur cette figure on représente un navire (type Rover sur ArduPilot) s'écartant de sa trajectoire initiale pour éviter un obstacle mobile grâce au système MDT l'ayant détecté. Le système de guidage est capable de mettre à jour dynamiquement sa mission et changer de trajectoire lorsqu'il est en mouvement. Le cadre vert représente la zone d'inclusion dans laquelle le navire peut circuler. Cette zone est partagée par le système MDT et ArduPilot qui peuvent se transmettre de nouvelles frontières avant le départ de la mission.

Bibliographie

[1] « Horizon (physique) », *Wikipédia*. oct. 12, 2020. Consulté le: juillet 15, 2021. [En ligne]. Disponible sur: https://fr.wikipedia.org/w/index.php?title=Horizon_(physique)&oldid=175497873

[2] « Pilonnement », *Wikipédia*. nov. 07, 2020. Consulté le: juillet 15, 2021. [En ligne]. Disponible sur: https://fr.wikipedia.org/w/index.php?title=Pilonnement&oldid=176340417

[3] « Courbure terrestre », *Wikipédia*. mai 26, 2021. Consulté le: juillet 15, 2021. [En ligne]. Disponible sur: https://fr.wikipedia.org/w/index.php?title=Courbure_terrestre&oldid=183286582

[4] « Boost C++ Libraries ». https://www.boost.org/ (consulté le juillet 15, 2021).

[5] « Flame Graphs ». https://www.brendangregg.com/flamegraphs.html (consulté le juillet 15, 2021).

[6] « simplify - 1.50.0 ». https://www.boost.org/doc/libs/1_50_0/libs/geometry/doc/html/geometry/reference/algorithms/simplify/simplify_3.html (consulté le juillet 15, 2021).

[7] « OpenCV: cv::LineIterator Class Reference ». https://docs.opencv.org/4.5.1/dc/dd2/classcv_1_1LineIterator.html (consulté le juillet 15, 2021).

[8] O. Ronneberger, P. Fischer, et T. Brox, « U-Net: Convolutional Networks for Biomedical Image Segmentation », in *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015*, vol. 9351, N. Navab, J. Hornegger, W. M. Wells, et A. F. Frangi, Éd. Cham: Springer International Publishing, 2015, p. 234-241. doi: 10.1007/978-3-319-24574-4_28.

[9] J. Long, E. Shelhamer, et T. Darrell, « Fully Convolutional Networks for Semantic Segmentation », nov. 2014, Consulté le: juillet 15, 2021. [En ligne]. Disponible sur: https://arxiv.org/abs/1411.4038v2

[10] « Defect Detection in Products using Image Segmentation | by Vinithavn | Analytics Vidhya | Medium ». https://medium.com/analytics-vidhya/defect-detection-in-products-using-image-segmentation-a87a8863a9e5 (consulté le juillet 15, 2021).

[11] J. Brownlee, « Gentle Introduction to the Adam Optimization Algorithm for Deep Learning », *Machine Learning Mastery*, juill. 02, 2017. https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/ (consulté le juillet 15, 2021).

[12] « Welcome to the ArduPilot Development Site — Dev documentation ». https://ardupilot.org/dev/index.html (consulté le juillet 15, 2021).

Table des illustrations