

INSA Lyon - Département Informatique

Rapport

de

PROJET DE FIN D'ÉTUDES

**Navigation et contrôle multi-robots pour l'inspection
acoustique de structures métalliques [R&D]**

***Multi-robot navigation and control for acoustic
inspection of metal plate structures [R&D]***

Brandon ALVES

Soutenance le 27 juin 2023

Projet réalisé du **9 janvier 2023** au **30 juin 2023**

dans la structure d'accueil

INRIA (Villeurbanne, France)

Référent	:	Cédric PRADALIER, Professeur	GT Europe
Référent	:	Olivier SIMONIN, Professeur	INSA Lyon
Tuteur	:	Mathieu MARANZANA, Maître de conférences	INSA Lyon

Remerciements

TODO: Remerciements

List of Figures

1	Modèle de crawler utilisé pour l'inspection acoustique de structures métalliques.	3
2	Stratégie de navigation peinture au rouleau.	5
3	Stratégie de navigation ski nordique.	7
4	Stratégie de navigation investigation polygonale.	8
5	Angle du signal émis et reçu pour la stratégie de navigation ski nordique.	10
6	Évolution du score de Cohen et du temps d'exécution de l'algorithme de peinture au rouleau en fonction de la densité du monde et de la distance entre les robots.	20
7	Évolution du κ de Cohen et du temps d'exécution de l'algorithme de peinture au rouleau en fonction de la distance qui sépare les deux crawlers.	21
8	Exemple de zone fantôme situé en bas à gauche de la carte.	22
9	Évolution du score de Cohen et du temps d'exécution de l'algorithme de peinture au rouleau en fonction de la densité du monde pour différentes valeurs de la distance entre les deux crawlers.	23
10	Évolution du score de Cohen et du temps d'exécution de l'algorithme de peinture au rouleau en fonction de la densité du monde pour différentes valeurs du pas entre les deux crawlers.	24
11	Évolution du κ de Cohen et du temps d'exécution de l'algorithme <i>ski nordique</i> en fonction de la distance qui sépare les deux crawlers.	25
12	Évolution du κ de Cohen et du temps d'exécution de l'algorithme <i>ski nordique</i> en fonction du pas des crawlers.	26
13	Différents environnements de test.	45

List of Tables

1	Interprétation du κ de Cohen selon Landis et Koch.	16
2	Paramètres expérimentaux utilisés pour chaque stratégie de navigation.	18
3	Résultats attendus pour chaque stratégie de navigation.	19

List of Algorithms

1	Processus de mise à jour de la grille d'occupation à l'aide de l'algorithme de tracé de segment de Bresenham.	13
2	Algorithme du κ de Cohen.	15

Listings

1	Implémentation de l'algorithme de peinture au rouleau	29
---	---	----

2	Implémentation de l'algorithme de ski nordique	31
3	Implémentation de l'algorithme d'investigation polygonale	35

Contents

1	Introduction	1
2	Étude bibliographique	2
3	Propositions scientifiques et techniques	2
3.1	Définitions préliminaires	3
3.2	Proposition de solution	4
3.3	Étude théorique de propriétés de la solution proposée	10
4	Implémentations techniques	11
5	Expérimentations, validations et évaluations	16
6	Bilan personnel	27
7	Conclusion et perspectives	27
	References bibliographiques	27

Annexes

A	Implémentations techniques	29
A.1	Implémentation de l'algorithme <i>peinture au rouleau</i>	29
A.2	Implémentation de l'algorithme <i>ski nordique</i>	31
A.3	Implémentation de l'algorithme <i>investigation polygonale</i>	35
B	Environnements de test	45
C	Résultat d'investigation	46

INFO: Ceci est un rapport d'environ 30 pages (hors annexes, hors première, deuxième, troisième et quatrième de couverture, hors table des matières et éventuelles autres tables des figures, des définitions, des algorithmes...).

Introduction / Background / Motivation:

TODO: What did you try to do? What problem did you try to solve? Articulate your objectives using absolutely no jargon.

TODO: How is it done today, and what are the limits of current practice?

TODO: Who cares? If you are successful, what difference will it make?

TODO: What data did you use? Provide details about your data, specifically choose the most important aspects of your data mentioned here: Datasheets for Datasets (<https://arxiv.org/abs/1803.09010>). Note that you do not have to choose all of them, just the most relevant.

Approach:

TODO: What did you do exactly? How did you solve the problem? Why did you think it would be successful? Is anything new in your approach?

TODO: What problems did you anticipate? What problems did you encounter? Did the very first thing you tried work?

Experiments and Results:

TODO: How did you measure success? What experiments were used? What were the results, both quantitative and qualitative? Did you succeed? Did you fail? Why? Justify your reasons with arguments supported by evidence and data. Make sure to mention any code repositories and/or resources that you used!

1 Introduction

Ce projet de fin d'étude s'inscrit dans le contexte plus large du projet européen BugWright2, qui vise à résoudre la problématique de l'inspection autonome et la maintenance de grandes structures métalliques avec des flottes hétérogènes de robots mobiles. Dans ce projet, nous nous concentrons sur le développement de stratégies de navigation pour un ensemble de robots mobiles utilisant des ondes ultrasoniques guidées pour réaliser l'inspection des plaques métalliques. En effet, les ondes guidées ont la particularité de se propager le long d'une plaque en interagissant avec la matière qui la compose, et en étant affectées par des changements de géométrie liés, en particulier, à la corrosion.

Le problème principal est donc de définir des stratégies de navigation multi-robot pour optimiser l'acquisition des données permettant de réaliser une tomographie des surfaces métalliques. Pour atteindre cet objectif, nous allons dans un premier temps effectuer une recherche bibliographique, puis mettre en place des méthodes de navigation dans un environnement de simulation. Enfin, nous envisagerons un déploiement sur différents robots en fonction des résultats obtenus. Ce projet sera réalisé sous la supervision de Olivier Simonin (INSA Lyon CITI lab) et de Cédric Pradalier (CNRS IRL2958 GT).

Les contributions attendues de ce projet sont les suivantes:

- Développement de stratégies de navigation multi-robot pour l'inspection acoustique de structures métalliques.
- Optimisation de l'acquisition de données pour la réalisation de la tomographie.
- Résolution des problèmes de coordination entre les robots et de synchronisation des horloges.
- Implémentation des méthodes de navigation dans un environnement de simulation et leur déploiement sur des robots réels.

Ce rapport présente le travail effectué dans le cadre de notre projet de fin d'étude sur la navigation et le contrôle multi-robots pour l'inspection acoustique de structures métalliques. Dans la première section, nous introduisons le sujet du rapport et présentons les objectifs de notre projet. La deuxième section est consacrée à l'étude bibliographique, où nous résumons les recherches et les publications existantes sur le sujet. Dans la troisième section, nous proposons une solution pour la navigation et le contrôle multi-robots pour l'inspection acoustique de structures métalliques. Cette section est divisée en trois sous-sections : définitions préliminaires, proposition de solution et étude théorique de propriétés de la solution proposée. La quatrième section décrit les détails de l'implémentation technique de notre solution proposée. La cinquième section présente les résultats de nos expérimentations, validations et évaluations. Dans la sixième section, nous faisons un bilan personnel de notre expérience de travail sur ce projet. Enfin, dans la septième section, nous concluons notre rapport en résumant les résultats obtenus, les limites du projet et les perspectives pour des recherches futures.

2 Étude bibliographique

TODO: Étude bibliographique

3 Propositions scientifiques et techniques

Nous proposons trois stratégies de navigations multi-robots pour l'inspection acoustique de structures métalliques afin d'optimiser l'acquisition de données qui permettrons de réaliser la tomographie des surfaces métalliques. Ces trois stratégies sont les suivantes:

- Stratégie de navigation *peinture au rouleau*
- Stratégie de navigation *ski nordique*
- Stratégie de navigation *investigation polygonale*

Parmi ces stratégies, deux sont non réactives et peuvent être considérées comme des stratégies d'exploration grossières, le but étant de rapidement obtenir une couverture globale de la surface à inspecter (*peinture au rouleau* et *ski nordique*). La troisième stratégie est réactive et permet d'optimiser l'acquisition de données pour la réalisation de la tomographie (*investigation polygonale*). Dans cette section, nous présentons les définitions préliminaires, la proposition de solution et l'étude théorique de propriétés de la solution proposée.

3.1 Définitions préliminaires

Ici, nous allons expliciter les hypothèses et les définitions préliminaires qui seront utilisées dans la suite de ce rapport. Premièrement, nous considérons un environnement plan, borné et de taille connue. Nous ne nous intéressons pas à la localisation des robots dans l'environnement, mais nous supposons que chaque robot est capable de connaître sa position dans l'environnement. Nous supposons également que les obstacles sont localisés dans l'environnement. Seules les zones de corrosion ne sont pas localisées.

Nous utilisons des robots de type "crawlers". Ces robots sont équipés de deux roues motrices et d'une roue folle. Un exemple de crawler est présenté sur la figure 1. La pose du robot est définie par un triplet (x, y, θ) où x et y sont les coordonnées du robot dans l'environnement et θ est l'orientation du robot dans l'environnement. Nous supposons que la pose du robot est connue. Nous supposons également que les robots sont capables de se synchroniser afin de pouvoir se déplacer de manière simultanée ou bien de manière alternée. On note cr le coût de rotation du robot et ct le coût de translation du robot.

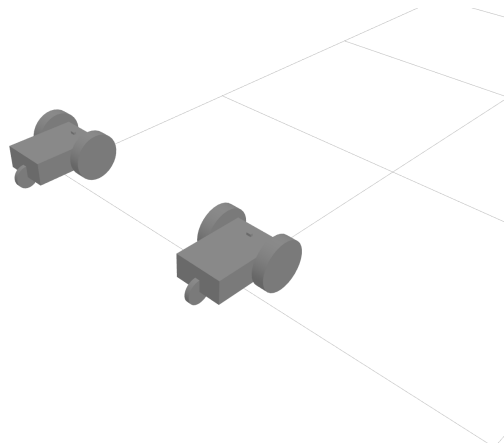


Figure 1 – Modèle de crawler utilisé pour l'inspection acoustique de structures métalliques.

Chaque robot est soit émetteur, soit récepteur, soit les deux. Les crawlers sont équipés de différents capteurs. Parmi eux:

- un capteur IMU (Inertial Measurement Unit)

- un capteur UGW (Ultrasonic Guided Waves)
- un capteur LIDAR (Light Detection And Ranging)

Le capteur IMU permet de connaître l'orientation du robot dans l'environnement. Le capteur UGW permet de détecter les zones de corrosion sur la surface métallique en émettant et en recevant des ondes ultrasoniques. Le capteur LIDAR permet de détecter les obstacles dans l'environnement. Les obstacles considérés ici, sont principalement les différents robots inspectant la surface métallique. La détection de ces zones de corrosion est réalisée par l'émission d'ondes ultrasoniques par un robot et la réception de ces ondes par un autre robot. Dans la mesure où l'onde reçue par un des crawler est altérée, alors il existe un point de corrosion entre le robot émetteur et le robot récepteur. La détection de ces zones de corrosion est réalisée en temps réel. La portée maximale des ondes ultrasoniques est notée d_{max} . Nous approximations le temps de propagation des ondes ultrasoniques dans la surface métallique par un temps nul.

Nous utilisons une grille d'occupation pour modéliser l'environnement dans lequel les robots évoluent lors de l'inspection acoustique des structures métalliques. Cette grille nous permet de représenter et de catégoriser les différents états des zones de la surface métallique. La grille d'occupation est composée de cellules, où chaque cellule correspond à une petite région de l'environnement. En particulier, nous avons utilisé une résolution de 0.05 mètre par cellule. Nous utilisons trois états pour caractériser ces cellules : inconnu, vide et occupé. L'état inconnu désigne les zones dont l'état n'a pas encore été déterminé ou détecté. L'état vide indique les zones où il n'y a pas de corrosion détectée, c'est-à-dire que la surface métallique est saine. Enfin, l'état occupé représente les zones de corrosion identifiées, où la présence de défauts ou de détérioration est détectée.

En utilisant cette grille d'occupation, nous pouvons suivre et mettre à jour en temps réel l'état des différentes zones de la surface métallique pendant l'inspection. Cela nous permet de planifier les mouvements des robots, d'optimiser leur trajectoire et d'assurer une couverture complète de la surface à inspecter. De plus, cette représentation nous offre une vision claire de l'état de corrosion de la structure métallique, facilitant ainsi l'analyse et l'évaluation des résultats de l'inspection.

Dans la suite de notre proposition, nous détaillerons les algorithmes et les méthodes utilisées pour mettre à jour la grille d'occupation en fonction des informations recueillies par les capteurs des robots. Nous discuterons également des stratégies de navigation multi-robots qui exploitent cette modélisation pour optimiser l'acquisition de données et améliorer l'efficacité de l'inspection acoustique.

3.2 Proposition de solution

Nous présentons notre proposition de solution pour l'inspection acoustique de structures métalliques en utilisant des stratégies de navigation multi-robots. Nous avons développé trois stratégies spécifiques pour optimiser l'acquisition de données et permettre la réalisation de la tomographie des surfaces métalliques.

Stratégie de navigation *peinture au rouleau*

La première stratégie de navigation que nous proposons est la stratégie de navigation *peinture au rouleau*. Cette stratégie repose sur une exploration grossière de la surface à inspecter, où les robots se déplacent en ligne droite sur des trajectoires parallèles, garantissant une couverture globale de la zone d'inspection. Il s'agit donc ici d'effectuer un quadrillage de la surface à inspecter.

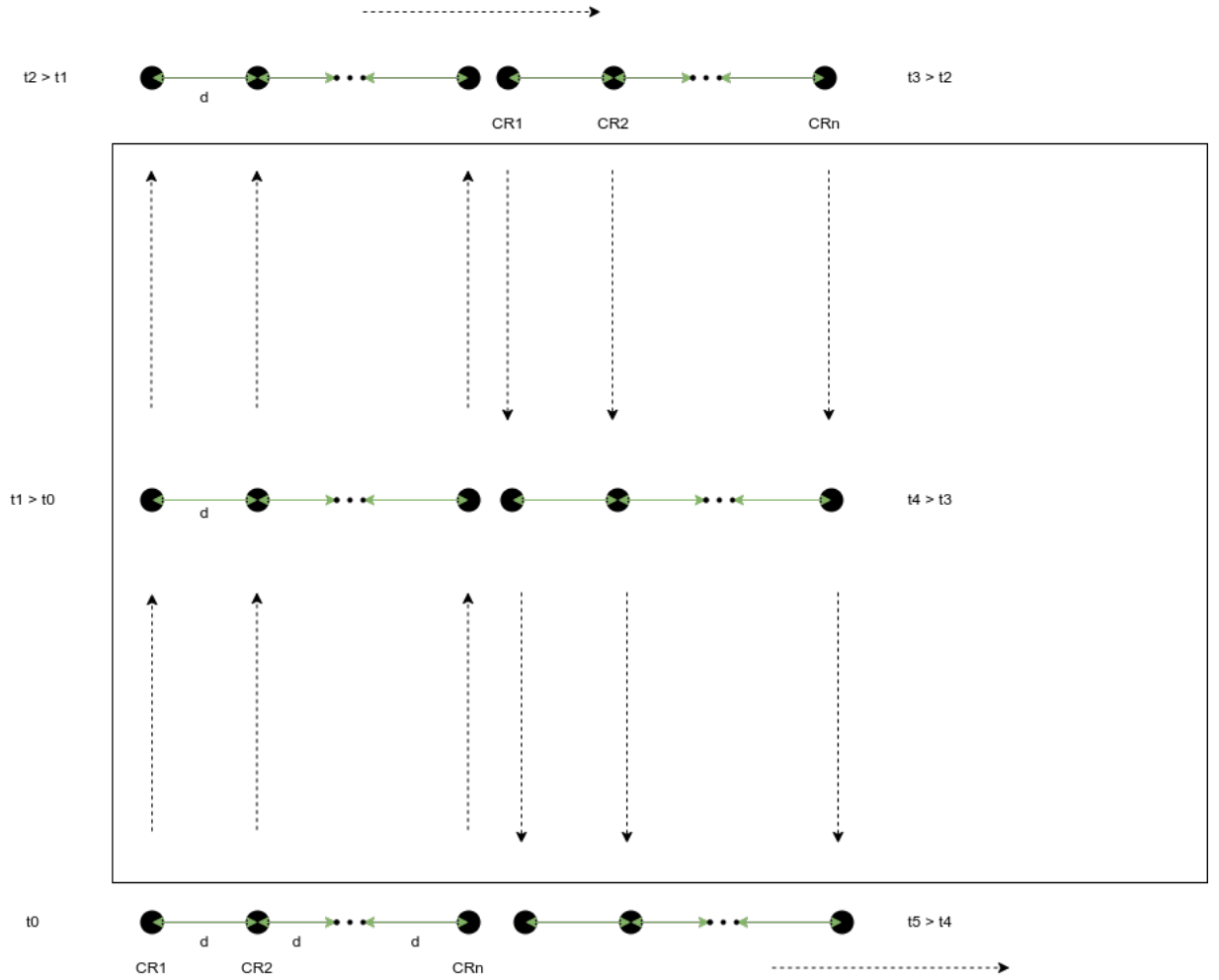


Figure 2 – Stratégie de navigation peinture au rouleau.

Nous présentons à la figure 2 un schéma décrivant la stratégie de navigation peinture au rouleau. Cette stratégie consiste en deux phases, une phase de déplacement vertical et une phase de déplacement horizontal. La figure 2 présente la première phase de déplacement vertical. Afin de réaliser cette stratégie, un minimum de $n \geq 2$ robots, aligné horizontalement et séparé par une distance d , est utilisé. Ces robots se déplacent verticalement, en simultané, en suivant

une trajectoire parallèle. Une fois l'extrémité de la surface à inspecter atteinte, les robots effectuent une rotation de 90 degrés et se translatent horizontalement, en simultané, d'une distance $2 * d$. Ils effectuent ensuite une nouvelle rotation de 90 degrés et se déplacent de nouveau verticalement, en ligne droite, en simultané, en suivant une trajectoire parallèle entre eux, jusqu'à atteindre l'autre extrémité de la surface à inspecter. Ce procédé est répété jusqu'à ce que la surface métallique soit entièrement inspectée. Le même procédé est ensuite répété, mais cette fois-ci, horizontalement.

Le fait que les robots se déplacent en suivant une trajectoire parallèle et en simultané, implique que les rayons du signal émis par le robot émetteur et reçu par le robot récepteur, ont toujours une orientation de 0 pour la phase verticale et une orientation de $\frac{\pi}{2}$ pour la phase horizontale. Il n'y a donc pas une grande variation de l'orientation du signal émis et reçu. Ainsi, cette stratégie ne pourra approcher les enveloppes convexes des zones de corrosion que par des rectangles.

Stratégie de navigation *ski nordique*

La deuxième stratégie que nous proposons est la stratégie de navigation *ski nordique*. Cette stratégie consiste toujours en des déplacements en ligne droite et en suivant des trajectoires parallèles, mais cette fois-ci, les robots se déplacent de manière séquentielle et non plus de manière simultanée. Dans cette stratégie, nous avons voulu accroître la diversité d'orientation des rayons du signal émis et reçu, afin d'approcher plus précisément les enveloppes convexes des zones de corrosion.

La figure 3 présente un schéma décrivant la stratégie de navigation *ski nordique*. Cette stratégie consiste également en deux phases, une phase de déplacement vertical et une phase de déplacement horizontal. La figure 3 présente la première phase de déplacement vertical. Afin de réaliser cette stratégie, un minimum de $n \geq 2$ robots, alignés horizontalement et séparés par une distance d , est utilisé. Par souci de clarté, nous avons représenté à la figure 3 uniquement deux robots. Ces robots se déplacent verticalement, en suivant une trajectoire parallèle, mais de manière séquentielle. Le premier robot se déplace en ligne droite d'une distance s et s'arrête. Le second robot se déplace ensuite en ligne droite d'une distance $2 * s$ et s'arrête. Ce procédé est répété jusqu'à ce que l'extrémité de la surface soit atteinte. Ensuite les robots répètent ce même procédé, dans le sens inverse et de manière à ce que les points d'arrêt des robots ne soient pas les mêmes que ceux précédemment. Les robots se déplacent ensuite horizontalement d'une distance d et répètent le même procédé jusqu'à ce que la surface métallique soit entièrement inspectée. Le même procédé est ensuite répété, mais cette fois-ci, horizontalement.

Le fait que les robots se déplacent en suivant une trajectoire parallèle, mais de manière séquentielle, implique que les rayons du signal émis par le robot émetteur et reçu par le robot récepteur, ont une orientation d'une plus grande variation. Ainsi, cette stratégie permet d'approcher les enveloppes convexes des zones de corrosion par des formes plus diverses et précises que des rectangles.

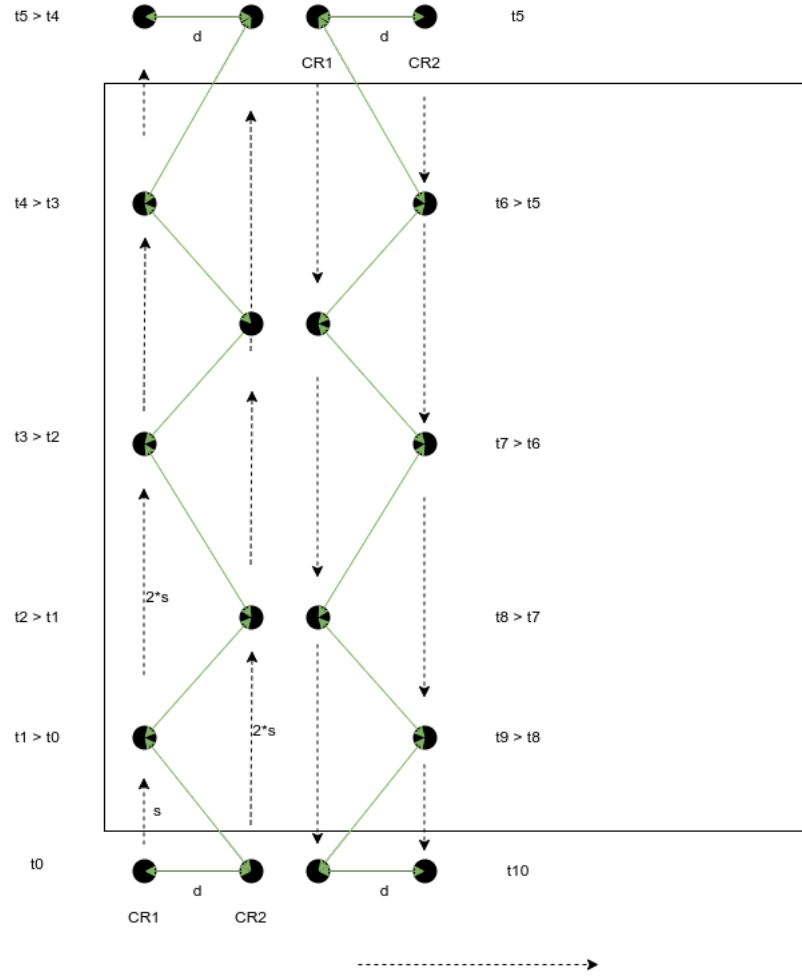


Figure 3 – Stratégie de navigation ski nordique.

Stratégie de navigation *investigation polygonale*

La troisième stratégie que nous proposons est la stratégie de navigation *investigation polygonale*. Nous avons vu, précédemment, qu'à l'issue de la réalisation de la stratégie de navigation *peinture au rouleau*, l'enveloppe convexe des zone de corrosion est approximée par un rectangle. Cette approximation est un peu plus précise pour la stratégie de navigation *ski nordique*. Il serait intéressant de pouvoir un plus grand degré de précision autour des zones potentielles de corrosion. C'est ce que nous proposons avec la stratégie de navigation *investigation polygonale*. Cette stratégie consiste à investiguer autour des zones potentielles de corrosion, détectées au préalable par une des deux stratégies de navigation précédentes. Elle consiste à positionner les robots autour des zones de corrosion et à les faire se déplacer en suivant une trajectoire polygonale, de manière à ce que les rayons du signal émis et reçu aient une orientation d'une plus

grande variation autour même de ces zones.

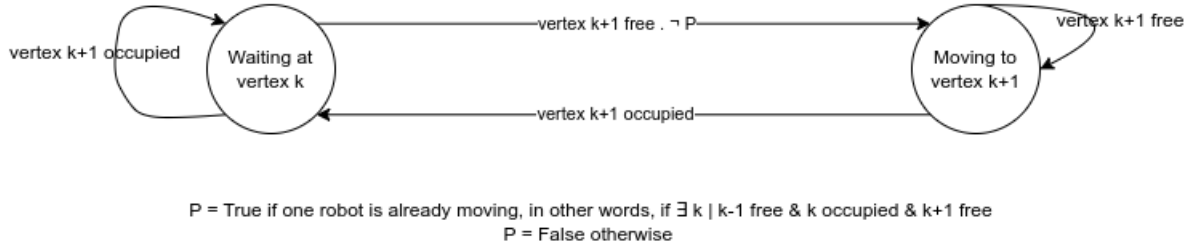


Figure 4 – Stratégie de navigation investigation polygonale.

Nous présentons à la figure 4 un automate à états finis décrivant la stratégie de navigation *investigation polygonale*. Au début de la stratégie de navigation *investigation polygonale*, les $n \in \mathbb{N}$ robots constitutifs de $k \in \mathbb{N}$ équipes sont positionnés sur des sommets consécutifs d'un polygone englobant la zone potentielle de corrosion. Dans cette dernière, chaque robot a deux états. Le premier consiste à attendre et le second consiste à se déplacer en suivant la trajectoire polygonale, à savoir parcourir les différents sommets constituant le polygone. Le robot capable d'avancer, c'est-à-dire, dont le sommet suivant n'est pas occupé par un autre robot, avance. Les autres attendent jusqu'à ce que le robot qui avance atteigne le dernier sommet libre du polygone. Le procédé est ensuite répété pour chaque robot de chaque équipe jusqu'à ce que les sommets occupés par les robots soient les mêmes que ceux occupés au début de la stratégie de navigation *investigation polygonale*.

Cette stratégie a deux avantages. Le premier est qu'elle permet de rapidement éliminer les zones fantômes. Le second est qu'elle permet d'approcher les enveloppes convexes des zones de corrosion par des formes plus diverses et précises que des rectangles du fait de la grande variation de l'orientation des rayons du signal émis et reçu par les robots autour de chaque sommet du polygone.

Cette stratégie nécessite deux étapes préalables à son exécution :

1. l'extraction des zones de corrosion détectées par une des stratégies de navigation précédentes.
2. la détermination de l'ordre d'investigation des zones de corrosion.

La première étape peut être résolue en utilisant un algorithme de décomposition de graphe en composantes fortement connexes. Une composante fortement connexe est définie à la définition 1. Nous considérons alors notre image comme un graphe non orienté $G = (V, E)$, où V est l'ensemble des sommets du graphe, correspondant aux pixels de l'image et E sont les arêtes du graphe, correspond aux pixels adjacents. Ce problème est bien connu et il existe des algorithmes simples pour les résoudre comme l'algorithme de Tarjan de complexité temporelle linéaire $O(|V| + |E|)$. Nous ne nous pencherons pas plus sur ce problème et confions sa résolution à la bibliothèque OpenCV.

Définition 1 (Composante fortement connexe (SCC)). Une composante fortement connexe d'un graphe orienté $G = (V, E)$ est un sous-ensemble C de V tel que pour tout couple de sommets $(u, v) \in C^2$, il existe un chemin de u à v et un chemin de v à u .

La seconde étape peut être résolue en utilisant un algorithme de TSP (*Travelling Salesman Problem*) dans le cas où le nombre d'équipe $k = 1$ et un algorithme de mTSP (*multiple depot multiple Travelling Salesman Problem*) dans le cas où le nombre d'équipe $k > 1$. Il existe plusieurs paradigmes de résolution pour résoudre ce genre de problème. Un premier est de trouver une solution exacte en utilisant un algorithme de programmation linéaire en nombres entiers. Un second est de trouver une solution approchée en utilisant une méta-heuristique.

Définition 2 (Problème du voyageur de commerce (TSP)). Étant donné une liste de villes et les distances entre chaque paire de villes, quel est l'itinéraire le plus court possible qui visite chaque ville exactement une fois et revient à la ville d'origine ?

Le problème du voyageur de commerce est un problème NP-complet et il peut être traité comme un problème d'optimisation linéaire en nombres entiers. Pour ce faire, nous utilisons la formulation présentée à l'équation 1.

$$\begin{aligned}
& \text{minimiser} && \sum_{i \in V} \sum_{j \in V} c_{ij} x_{ij} \\
& \text{soumis à} && \sum_{i \in V} x_{ij} = 1 && \forall j \in V \\
& && \sum_{j \in V} x_{ij} = 1 && \forall i \in V \\
& && \sum_{i \in S} \sum_{j \in S} x_{ij} \leq |S| - 1 \quad \forall S \subset V, 2 \leq |S| \leq |V| - 1 \\
& && x_{ij} \in \{0, 1\} && \forall i \in V, \forall j \in V
\end{aligned} \tag{1}$$

La fonction objectif à minimiser de la formulation 1 est la somme des distances entre chaque paire de villes. Les deux premières contraintes assurent que chaque ville est visitée exactement une fois. La troisième contrainte assure que le cycle formé par les villes visitées est simple, c'est-à-dire, qu'il ne contient pas de sous-cycles. La dernière contrainte assure que les variables de décision x_{ij} sont binaires, avec $x_{ij} = 1$ si le robot se déplace de la ville i à la ville j et $x_{ij} = 0$ sinon.

Définition 3 (Problème du voyageur de commerce multiple avec dépôts multiples (mTSP)). Étant donné une liste de villes et les distances entre chaque paire de villes, quel est l'itinéraire le plus court possible qui visite chaque ville exactement une fois et revient à la ville d'origine, en considérant que l'on a $k \in \mathbb{N}$ voyageurs ?

Le problème du voyageur de commerce multiple avec dépôts multiples est également un problème NP-complet. Celui-ci peut être résolu en utilisant une méta-heuristique comme un algorithme génétique [1, 2]

Dans les prochaines sections, nous détaillerons chaque stratégie de navigation, en exposant les algorithmes et les mécanismes spécifiques utilisés pour mettre en œuvre notre proposition de solution. Nous analyserons également les performances et les résultats obtenus à travers des expérimentations et des évaluations approfondies.

3.3 Étude théorique de propriétés de la solution proposée

Stratégie de navigation *ski nordique*

Proposition 1. *L'angle du signal émis et reçu par les robots, pour la stratégie de navigation ski nordique, varie entre $-\tan^{-1}(\frac{s}{d})$ et $\tan^{-1}(\frac{s}{d})$.*

Proof. Pour démontrer la proposition 1, nous nous appuyons sur les propriétés de trigonométrie. Nous explicitons à la figure 5 la démarche entreprise pour trouver $\alpha = -\tan^{-1}(\frac{s}{d})$

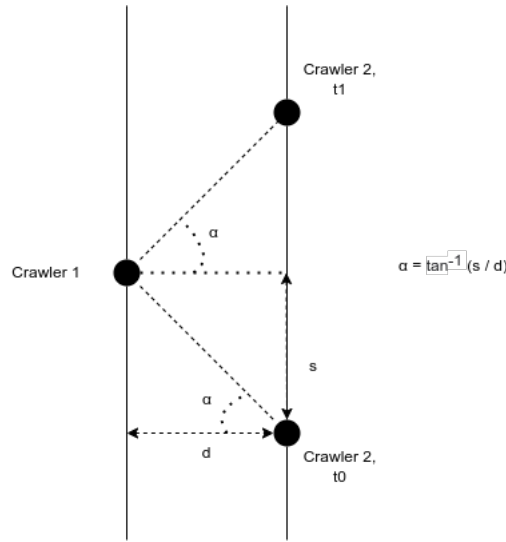


Figure 5 – Angle du signal émis et reçu pour la stratégie de navigation ski nordique.

□

Stratégie de navigation *investigation polygonale*

Proposition 2. *L'enveloppe convexe des zones de corrosion à l'issue de l'investigation polygonale, avec un polygone à p sommets, $p \in \mathbb{N}$, est un polygone d'au plus $2p$ sommets.*

Proof. Démontrons la proposition 2.

- On a, pour chaque sommet du polygone, il existe deux droites passant par ce point et touchant la zone de corrosion sans la traverser.

- On a donc pour un sommet du polygone, au plus deux droites qui participent à la construction de l'enveloppe convexe et donc, dont un segment de ces droites est une arête de l'enveloppe convexe.
- On a donc pour un sommet du polygone, au plus deux arêtes de l'enveloppe convexe.
- On a donc pour un polygone à p sommets, au plus $2p$ arêtes de l'enveloppe convexe.
- On a donc pour un polygone à p sommets, au plus $2p$ sommets de l'enveloppe convexe.

□

Proposition 3. *TODO: Qu'est ce que je souhaite démontrer ?*

Proof. Démontrons la proposition 3. **TODO: Faire la preuve.**

□

4 Implémentations techniques

Dans cette section, nous mettons en évidence certaines des différentes implémentations techniques que nous avons développées pour soutenir nos solutions de navigation et de contrôle multi-robots dans le contexte de l'inspection acoustique de structures métalliques. Nous commençons par décrire notre adaptation de l'algorithme de tracé de sagement de Bresenham, largement utilisé pour déterminer quels sont les points d'un plan discret qui doivent être tracés afin de former une approximation de segment de droite entre deux points donnés. Ensuite, nous abordons l'implémentation de l'algorithme *peinture au rouleau*, qui permet aux robots de se déplacer de manière simultanée, en suivant des trajectoires parallèles. Nous poursuivons avec l'implémentation de l'algorithme *ski nordique*, qui permet aux robots de se déplacer de manière alternée, en suivant des trajectoires parallèles, modifiant ainsi l'orientation du vecteur représentant la direction de déplacement de l'onde émise et reçue par la paire de robot. De plus, nous examinons l'implémentation de l'algorithme *investigation polygonale*, qui permet aux robots d'examiner plus précisément des zones suspectes de corrosion. Enfin, nous présentons l'algorithme de calcul du κ de Cohen, utilisé pour évaluer la qualité et la fiabilité des résultats de l'inspection acoustique. Nous discutons en détail de notre implémentation de cet algorithme, qui fournit des mesures quantitatives pour évaluer la performance des robots dans l'inspection des structures métalliques. Chacune de ces implémentations techniques contribue à l'efficacité et à la précision de notre approche de navigation et de contrôle multi-robots, et sera examinée en détail dans les sous-sections suivantes.

Algorithme de tracé de segment de Bresenham

L'algorithme de tracé de segment de Bresenham est couramment utilisé pour déterminer les points d'un plan discret qui doivent être tracés afin de former une approximation de segment

de droite entre deux points donnés. Lors du balaiement de la surface à inspecter par une paire de robot émetteur et récepteur, le robot émetteur émet une onde acoustique dans la structure métallique, qui est ensuite reçue par le robot récepteur. La détection étant considérée comme parfaite, le robot récepteur reçoit l'onde émise par le robot émetteur, sans quasi altération de la puissance du signal, si et seulement si le segment de droite entre les deux robots ne traverse pas une zone de corrosion. Il est ainsi possible de déterminer si une zone de corrosion est présente entre les deux robots en vérifiant si le signal reçu par le robot récepteur est suffisamment puissant. Dans la mesure où il n'y a pas de détection de corrosion entre l'émetteur et le récepteur, alors le segment de droite entre les deux robots est considéré comme étant libre de corrosion. Dans le cas contraire, alors les points du segment de droite entre les deux robots sont considérés comme étant de la corrosion, à l'exception des points préalablement perçus comme étant libre de corrosion.

Nous utilisons donc l'algorithme de tracé de segment de Bresenham pour déterminer les points du segment de droite entre les deux robots. L'algorithme est présenté à l'algorithme 1. La partie adaptée à notre problème se trouve entre les lignes 12 et 17 de ce dernier. À cet endroit, nous vérifions si la puissance du signal est suffisamment altérée et si le point du segment de droite entre les deux robots n'a pas déjà été perçu comme étant libre de corrosion. Si c'est le cas, alors le point considéré est marqué comme étant de la corrosion, modélisé par la valeur `OCCUPIED`. Si la puissance du signal n'est pas suffisamment altérée, alors le point considéré est marqué comme étant libre de corrosion, modélisé par la valeur `EMPTY`. Une fois que tous les points du segment ont été parcourus, la grille G est mise à jour avec les nouvelles informations. L'algorithme de tracé de segment de Bresenham contribue ainsi à la construction de la grille d'occupation qui permet de localiser les zones de corrosion détectées par les robots lors de l'inspection acoustique des structures métalliques.

Algorithme *peinture au rouleau et ski nordique*

Nous présentons dans cette sous-section les implémentations des algorithmes *peinture au rouleau* et *ski nordique*, présentées en annexe A, au listing 1 et au listing 2.

L'implémentation de ces algorithmes, ont été réalisées en utilisant le langage de programmation Python et les bibliothèques ROS (Robot Operating System). Dans ces implémentations, nous utilisons le framework Task Manager [3] pour gérer les tâches des robots inspecteurs. Tout d'abord, nous initialisons le nœud ROS et créons un client de tâches. Ensuite, nous récupérons les paramètres nécessaires tels que la vitesse des crawlers, l'identifiant du crawler, la distance entre les crawlers, le chevauchement ou encore les dimensions de la surface à inspecter.

Les algorithmes sont ensuite exécutés en suivant une séquence de mouvements précis. Pour chaque crawler, nous définissons des trajectoires verticales et horizontales en utilisant une boucle itérative et des calculs mathématiques. Les crawlers se déplacent le long des trajectoires définies, en utilisant les fonctions du client de tâches telles que `AlignWithTarget` et `FollowLine` pour maintenir une trajectoire précise.

Pendant l'exécution des algorithmes, les crawlers se synchronisent en utilisant les fonctions

Algorithm 1: Processus de mise à jour de la grille d’occupation à l’aide de l’algorithme de tracé de segment de Bresenham.

Data: $P_1 \in \mathbb{R}^2$, $P_2 \in \mathbb{R}^2$, $pw \in \mathbb{R}$, $threshold \in \mathbb{R}$, G :

$l \times w \rightarrow [\text{UNKNOWN}, \text{EMPTY}, \text{OCCUPIED}]$, $l \in \mathbb{N}$, $w \in \mathbb{N}$

with P_1 and P_2 the two points to connect, pw the power of the UGW, $threshold$ the threshold above which the power of the UGW is considered undistributed and G the grid to update.

Result: The updated grid.

```

1  $p_0 \leftarrow \text{from\_position\_to\_grid\_coordinate}(P_1)$ 
2  $p_1 \leftarrow \text{from\_position\_to\_grid\_coordinate}(P_2)$ 
3 if  $\text{is\_out\_of\_grid}(p_0)$  or  $\text{is\_out\_of\_grid}(p_1)$  then
4   | return
5 end
6  $dx \leftarrow p_1.x - p_0.x$ 
7  $dy \leftarrow p_1.y - p_0.y$ 
8  $sx \leftarrow \text{sign}(dx)$ 
9  $sy \leftarrow \text{sign}(dy)$ 
10  $err = dx - dy$ 
11 while  $p_0 \neq p_1$  do
12   | if  $pwd \leq threshold$  and  $G(p_0) = \text{UNKNOWN}$  then
13     |  $G(p_0) \leftarrow \text{OCCUPIED}$ 
14   | end
15   | else if  $pwd > threshold$  then
16     |  $G(p_0) \leftarrow \text{EMPTY}$ 
17   | end
18   |  $e2 \leftarrow 2 \times err$ 
19   | if  $e2 > -dy$  then
20     |  $err \leftarrow err - dy$ 
21     |  $p_0.x \leftarrow p_0.x + sx$ 
22   | end
23   | if  $e2 < dx$  then
24     |  $err \leftarrow err + dx$ 
25     |  $p_0.y \leftarrow p_0.y + sy$ 
26   | end
27 end

```

SetStatusSync et WaitForStatusSync du client de tâches. Cela garantit que les crawlers exécutent les mouvements de manière coordonnée et se positionnent correctement pour couvrir toute la surface métallique. À la fin de chaque étape de mouvement, le statut est mis à jour et

la synchronisation est effectuée avec le partenaire correspondant.

L’implémentation des deux algorithmes *peinture au rouleau* et *ski nordique* permettent aux crawlers d’inspecter la surface métallique de manière méthodique et complète. En utilisant des trajectoires verticales et horizontales, les crawlers parcourent la surface en chevauchant les zones précédemment inspectées pour s’assurer d’une couverture optimale. Ici, nous avons utilisé un chevauchement de 10 cm entre les différentes trajectoires verticales et horizontales.

Une fois les algorithmes terminés, le temps d’exécution est enregistré, fournissant une indication de la durée nécessaire pour inspecter la surface métallique. La grille d’occupation est également enregistré afin de calculer le score de l’inspection. Cette implémentation constitue une étape essentielle dans notre proposition de solution pour l’inspection acoustique de structures métalliques et permet de garantir une couverture complète et efficace de la surface à inspecter.

Algorithme *investigation polygonale*

Nous présentons dans cette sous-section l’implémentation de l’algorithme *investigation polygonale*, présentée en annexe A, au listing 3.

L’implémentation de cet algorithme, a également été réalisée en utilisant le langage de programmation Python et les bibliothèques ROS (Robot Operating System). Dans cette implémentation, nous utilisons toujours le framework Task Manager [3] pour gérer les tâches des robots inspecteurs. Premièrement, nous initialisons le nœud ROS et récupérons les différents paramètres et plus particulièrement la carte des zones potentielles et grossières de corrosion, sur laquelle nous nous basons pour l’inspection. Ensuite nous extrayons les composantes fortement connexes de la carte en utilisant la fonction `connectedComponentsWithStats` de la bibliothèque d’OpenCV. Cette fonction utilise l’algorithme spaghetti de Bolleli [4] pour extraire les composantes fortement connexes d’une image. Pour chacune de ces composantes nous récupérons son centre et ses dimensions. Ensuite, nous construisons un polygone à $p \in \mathbb{N}$ côtés autour de chaque centre d’une composante. Pour ce faire nous plaçons p sur une ellipse centrée sur le centre de la composante et dont les axes sont les dimensions de la composante. Nous avons donc pour chaque zone potentielle de corrosion un polygone à p côtés qui l’entoure. Il ne reste plus qu’à trouver le plus court chemin qui passe par tous les polygones. Pour cela, nous utilisons la bibliothèque Gurobi pour résoudre un simple TSP dans le cas où le nombre d’équipe de robots $k = 1$. Lorsque $k > 1$, nous utilisons l’algorithme génétique proposé par Elad Kivelevitch [5] pour résoudre le problème du mTSP avec plusieurs dépôts.

TODO: investigation

Algorithme de calcul du κ de Cohen

L’évaluation de la qualité et de la fiabilité des résultats de l’inspection acoustique est essentielle pour garantir des mesures précises de l’état des structures métalliques. Dans cette sous-section, nous présentons l’algorithme du calcul du κ de Cohen, présenté à l’algorithme 2,

une mesure statistique couramment utilisée pour évaluer l'accord entre les résultats obtenus par les robots et une référence humaine.

Algorithm 2: Algorithme du κ de Cohen.

Data: $I_0: l \times w \times 3 \rightarrow [0..255]$, $I: l \times w \times 3 \rightarrow [0..255]$, $l \in \mathbb{N}$, $w \in \mathbb{N}$
 with I_0 the ground truth image and I the image to score.

Result: $\kappa \in [0, 1]$

```

1  $TP \leftarrow 0$ 
2  $TN \leftarrow 0$ 
3  $FP \leftarrow 0$ 
4  $FN \leftarrow 0$ 
5 for  $i \leftarrow 1$  to  $l$  do
6   for  $j \leftarrow 1$  to  $w$  do
7     if  $is\_label\_1(I_0(i, j))$  then
8       if  $is\_label\_1(I(i, j))$  then
9          $TP \leftarrow TP + 1$ 
10      end
11      else
12         $FN \leftarrow FN + 1$ 
13      end
14    end
15    else
16      if  $is\_label\_1(I(i, j))$  then
17         $FP \leftarrow FP + 1$ 
18      end
19      else
20         $TN \leftarrow TN + 1$ 
21      end
22    end
23  end
24 end
25  $f_c \leftarrow \frac{(TN+FN)(TN+FP)+(FP+TP)(FN+TP)}{TP+TN+FN+FP}$ 
26  $\kappa \leftarrow \frac{TP+TN-f_c}{TP+TN+FN+FP-f_c}$ 

```

L'algorithme de calcul du κ de Cohen se base sur la notion de concordance et de discordance entre les résultats des inspections réalisées par les robots et celles réalisées par des inspecteurs humains. Il prend en compte les résultats positifs, négatifs, faux positifs et faux négatifs obtenus lors de l'inspection acoustique. Ces informations sont utilisées pour calculer la valeur du coefficient de Cohen, noté κ , avec $\kappa = \frac{p_o - p_e}{1 - p_e}$, où p_o est le taux d'accord observé et p_e le taux d'accord attendu.

L’algorithme se déroule en plusieurs étapes. Tout d’abord, les résultats des inspections réalisées par les robots et réel répartition des zones de corrosion sont comparés pour chaque zone inspectée. Ensuite, les résultats sont regroupés en quatre catégories : concordance positive, concordance négative, discordance positive (faux positifs) et discordance négative (faux négatifs). Ces catégories sont utilisées pour calculer les taux d’observation et d’accord observés entre les robots et la véritable répartition des zones de corrosion. Le κ de Cohen est ensuite calculé à partir des taux d’observation et d’accord observés, prenant en compte la possibilité de concordance due au hasard. Plus le κ de Cohen se rapproche de 1, plus il y a un accord élevé entre les résultats des robots et ceux des inspecteurs humains. En revanche, un κ proche de 0 indique un faible niveau d’accord, tandis qu’un κ négatif suggère une discordance entre les résultats. Une Interprétation du κ de Cohen selon Landis et Koch est présentée dans le tableau 1.

κ	Interprétation
< 0	Désaccord
0.00 – 0.20	Accord très faible
0.21 – 0.40	Accord faible
0.41 – 0.60	Accord modéré
0.61 – 0.80	Accord fort
0.81 – 1.00	Accord presque parfait

Table 1 – Interprétation du κ de Cohen selon Landis et Koch.

Nous avons implémenté cet algorithme dans le cadre de notre projet, en utilisant les résultats des inspections acoustiques effectuées par les robots et des cartes composées de zones de corrosion comme base de comparaison. Cette implémentation nous permet d’obtenir des mesures quantitatives pour évaluer la performance de notre approche de navigation et de contrôle multi-robots dans l’inspection des structures métalliques. Dans les prochaines sections, nous détaillerons les résultats obtenus grâce à l’application de cet algorithme du calcul du κ de Cohen.

5 Expérimentations, validations et évaluations

Dans cette section, nous présentons les expérimentations que nous avons menées pour valider et évaluer nos différentes stratégies de navigation et de contrôle multi-robots dans le contexte de l’inspection acoustique de structures métalliques. Ces expérimentations visent à démontrer l’efficacité, la précision et la fiabilité de notre système dans la détection et la localisation des zones de corrosion.

Pour mener à bien ces étapes, nous avons choisi d’effectuer nos expériences en utilisant Gazebo, un environnement de simulation bien établi dans le domaine de la robotique. Nous

avons commencé par construire plusieurs cartes de tests. Ces cartes modélisent une surface plane sur laquelle sont placées des formes géométriques simples, des rectangles et des cercles, et des formes plus complexes, des polygones entre 3 et 8 sommets. Ces différentes formes géométriques représentent les zones de corrosion que nous souhaitons détecter et localiser. Nous présentons en annexe B, à la figure 13 les cartes que nous avons construites pour nos expérimentations. Chacune de ces cartes est de taille 6 mètres par 6 mètres. Le nombre de zones de corrosion varie entre 5, 8, 11, 15, 20 et 30 zones. La taille et l’emplacement des zones de corrosion sont générés aléatoirement. Pour les cartes de 5, 8, 11 et 15 zones, nous avons généré 5 cartes différentes afin d’avoir des résultats plus représentatifs. Nous ne nous sommes pas permis de générer plusieurs cartes pour les cartes de 20 et 30 zones, le temps d’investigation polygonale étant trop conséquent.

Nous avons également simulé le capteur UGW par un capteur Ultra wideband (UWB). Ce capteur UWB permet d’émettre une impulsion et de la recevoir. En mesurant la puissance du signal, nous sommes en mesure de savoir si le signal a traversé un objet ou non. Le comportement de ce capteur UWB est donc similaire à celui du capteur UGW, à savoir qu’il permet de détecter la présence d’un objet entre deux points, mais pas de le localiser.

Nous avons évalué les performances des trois stratégies de navigation en terme de κ de Cohen et de temps d’inspection. Pour la stratégie *peinture au rouleau* et *ski nordique*, nous avons uniquement utilisé 2 robots. Pour ces deux stratégies, nous avons également fait varier la distance d entre les robots. Pour la stratégie *ski nordique*, nous avons également fait varier le pas s entre les robots. Pour la stratégie *investigation polygonale*, nous faisons varier le nombre de robots n , le nombre d’équipes k et le nombre de côtés p des polygones utilisés. Nous utilisons le résultat de la stratégie de navigation *peinture au rouleau* comme point de départ. Nous justifions ce choix par le fait que cette stratégie est la rapide et la moins précise et donc la plus susceptible de bénéficier d’une amélioration de la part de la stratégie *investigation polygonale* sans atteindre des temps d’inspection trop longs. Nous faisons donc varier le paramètre d de cette stratégie. Nous résumons les paramètres expérimentaux utilisés pour chaque stratégie dans le tableau 2.

Au cours de ces simulations, nous nous attendons à avoir certains résultats. Parmi eux, nous nous attendons à ce que la stratégie *peinture au rouleau* soit la plus rapide, mais également la moins précise. À l’inverse, nous nous attendons à ce que la stratégie *investigation polygonale* soit la plus précise, mais également la plus lente. Nous nous attendons également à ce que le paramètre d ait un impact sur la précision et le temps d’inspection des stratégies *peinture au rouleau* et *ski nordique*. Une distance d faible devrait permettre d’obtenir une meilleure précision, mais devrait également augmenter le temps d’inspection. De plus, nous nous attendons à ce que le paramètre s ait également un impact sur la précision et le temps d’inspection de la stratégie *ski nordique*. Une distance s faible devrait permettre d’obtenir une meilleure précision, mais devrait également augmenter le temps d’inspection. Nous nous attendons également à ce que le paramètre p ait un impact sur la précision et le temps d’inspection de la stratégie *investigation polygonale*. Un nombre de côtés p faible devrait permettre d’obtenir

Stratégie	Paramètre	Valeurs
<i>peinture au rouleau</i>	n	2
	d	1, 2, 3, 6 (mètres)
<i>ski nordique</i>	n	2
	d	1, 2, 3, 6 (mètres)
	s	1, 2, 3, 6 (mètres)
<i>investigation polygonale</i>	stratégie initiale	<i>peinture au rouleau</i>
	d	1, 2, 3, 6 (mètres)
	n	2, 4
	k	1, 2
	p	4, 6

Table 2 – Paramètres expérimentaux utilisés pour chaque stratégie de navigation.

une meilleure précision, mais devrait également augmenter le temps d’inspection. Ensuite, nous nous attendons à ce que les paramètres k et n aient un impact sur le temps d’inspection de la stratégie *investigation polygonale*. Un nombre d’équipes k ou un nombre de robots n élevé devrait permettre d’obtenir un temps d’inspection plus faible. Enfin, nous nous attendons à ce que le nombre de zones de corrosion ait un impact sur le temps d’inspection de la stratégie *investigation polygonale* mais pas sur les stratégies *peinture au rouleau* et *ski nordique*. Plus le nombre de zones de corrosion est élevé, plus le temps d’inspection devrait être élevé pour la stratégie *investigation polygonale*. Finalement, nous nous attendons à ce que le nombre de zones de corrosion ait un impact sur la précision des différentes stratégies. Plus le nombre de zones de corrosion est élevé, moins la précision devrait être élevée.

Par soucis de clarté, nous résumons dans le tableau 3 les résultats attendus pour chaque stratégie de navigation. Le tableau présente les résultats attendus pour chaque stratégie de navigation, en fonction des différents paramètres expérimentaux. Les stratégies de navigation incluent *peinture au rouleau*, *ski nordique* et *investigation polygonale*. Les paramètres comprennent la densité, la distance d , le pas s , le nombre de côtés d’un polygone utilisé dans l’investigation polygonale p , le nombre d’équipes k et le nombre de robots n .

Pour chaque stratégie, le tableau indique les scores et temps attendus associés à chacun des paramètres comparés aux scores et temps attendus pour les paramètres intermédiaires. Les scores sont représentés par des symboles "+" et "-" et indiquent le niveau de précision attendu. Les temps sont également indiqués par des symboles "+" et "-" et reflètent la lenteur d’inspection attendue. Ainsi un score ++ est considéré comme plus précis qu’un score +, et un temps - - est considéré comme plus lent qu’un temps - par exemple.

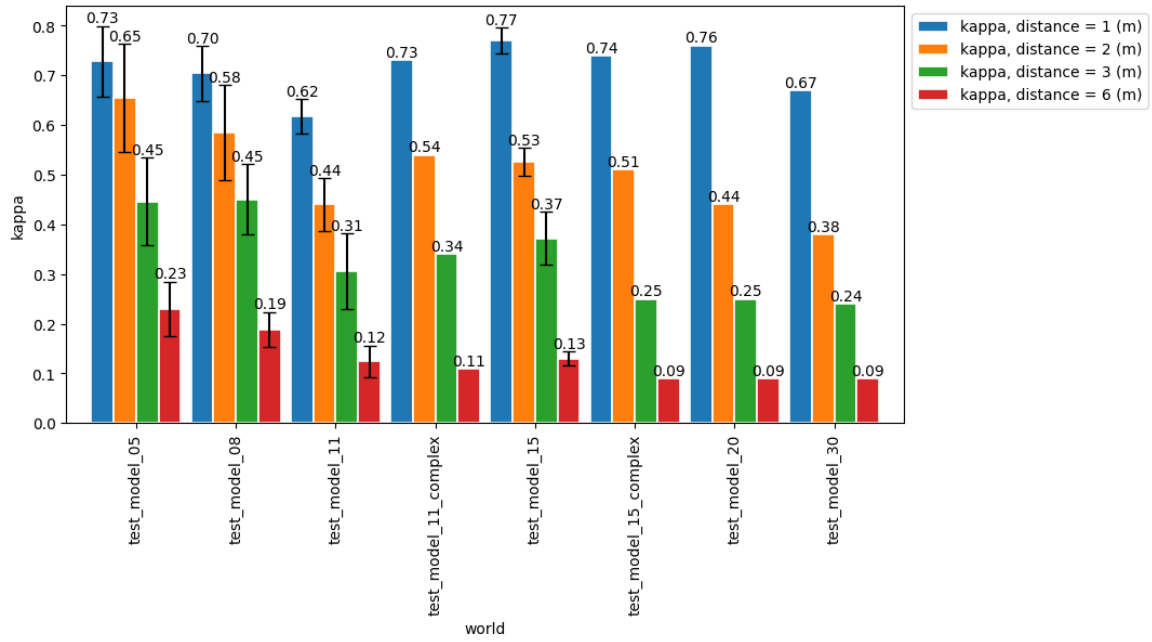
Stratégie	Paramètres	Score	Temps
<i>peinture au rouleau</i>	densité moyenne, d moyen	-	++
	faible densité	+	++
	forte densité	--	++
	faible d	+	+
	fort d	--	+++
<i>ski nordique</i>	densité moyenne, d moyen, s moyen	++	-
	faible densité	+++	-
	forte densité	+	-
	faible d	+++	--
	fort d	+	+
	faible s	+++	--
<i>investigation polygonale</i>	fort s	+	+
	densité moyenne, n moyen, k moyen, p moyen	+++	-
	faible densité	++++	+
	forte densité	++	--
	faible n	+++	--
	fort n	+++	+
	faible k	+++	--
	fort k	+++	+
	faible p	++	+
	fort p	++++	--

Table 3 – Résultats attendus pour chaque stratégie de navigation.

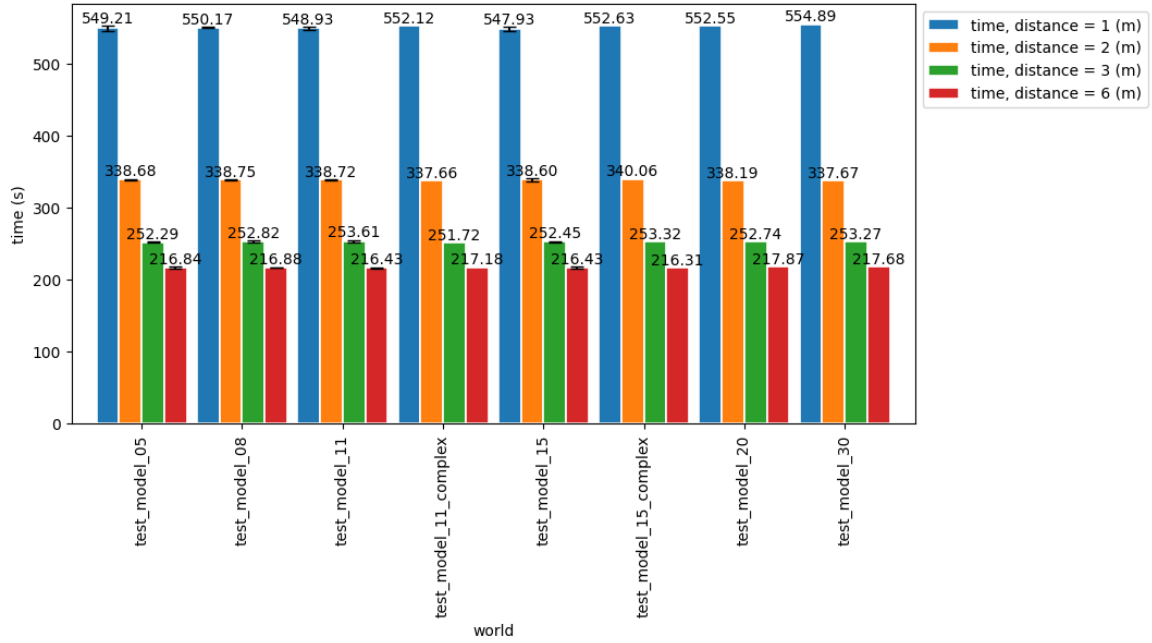
Stratégie de navigation *peinture au rouleau*

Nous résumons à la figure 6a l'évolution du score de Cohen en fonction de la densité du monde pour chaque valeur de d . Nous résumons également à la figure 6b l'évolution du temps d'inspection en fonction de la densité du monde pour chaque valeur de d .

Premièrement, nous pouvons observer que le score de Cohen diminue, de manière générale, avec le nombre de zones de corrosion. Il existe des exceptions, notamment pour la carte composée de 15 zones de corrosion, où le score de Cohen est plus élevé que pour les cartes composées de 5, 8 et 11 zones de corrosion. Cela s'explique du fait que dans les cartes composées de 5, 8 et 11 zones de corrosion, nous avons introduit des zones de corrosion de formes allongées contrairement à la carte composée de 15 zones de corrosion où les zones de corrosion sont toutes des cercles. En effet, les zones de corrosion de formes allongées ont une probabilité plus grande de faire apparaître des zones fantômes, illustrées à la figure 8, que les zones de corrosion de forme circulaire. Ces zones fantômes sont des zones libres de corrosion qui sont détectées par les crawlers. Il s'agit donc de faux positifs qui diminuent le score de Cohen. Ces zones fantômes ont également plus de chance d'apparaître lorsque la densité du monde est élevée et que donc



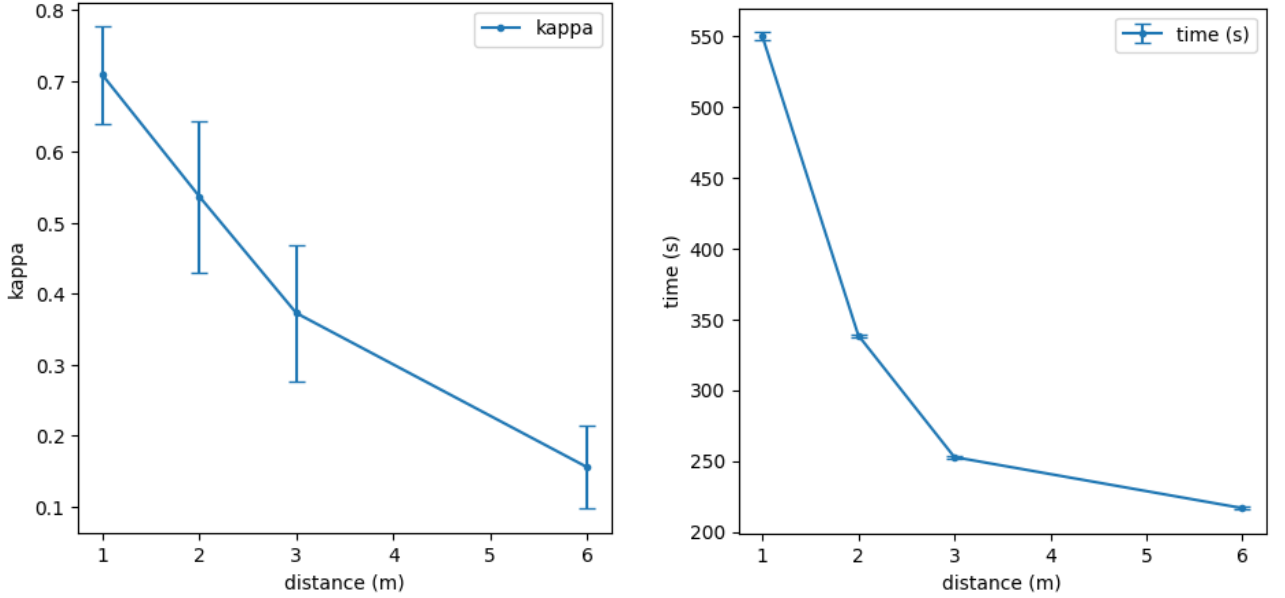
(a) κ en fonction de la densité du monde



(b) Temps d'exécution en fonction de la densité du monde

Figure 6 – Évolution du score de Cohen et du temps d'exécution de l'algorithme de peinture au rouleau en fonction de la densité du monde et de la distance entre les robots.

les zones de corrosion sont plus proches les unes des autres ou encore lorsque la distance d entre les deux crawlers est élevée. C'est bien ce que nous pouvons observer à la figure 7a où le score de Cohen diminue lorsque la distance d entre les deux crawlers augmente. Nous observons qu'il semble exister une relation linéaire entre le score de Cohen et la distance d entre les deux crawlers.



(a) κ en fonction de la distance entre les deux crawlers (b) Temps d'exécution en fonction de la distance entre les deux crawlers

Figure 7 – Évolution du κ de Cohen et du temps d'exécution de l'algorithme de peinture au rouleau en fonction de la distance qui sépare les deux crawlers.

Ensuite, nous observons que le temps d'exécution de l'algorithme *peinture au rouleau* est constant pour chaque valeur du nombre de zones de corrosion. Cela était attendu du fait que l'algorithme en question est un algorithme a priori et ne dépend donc pas du nombre de zones de corrosion. En revanche le temps d'exécution dépend de la distance d entre les deux crawlers. Comme nous pouvons observer à la figure 7b, le temps d'exécution augmente lorsque la distance d entre les deux crawlers diminue. Cela s'explique du fait que plus la distance d est grande, moins les crawlers doivent effectuer de déplacements pour couvrir la carte. Il semble également exister une relation linéaire entre le temps d'exécution et la distance d entre les deux crawlers, excepté pour la distance $d = 6$.

Nous avons également introduit deux cartes avec des formes plus complexes que les cartes de base. Malheureusement nous n'avons pas pu, par soucis de temps, faire varier la position des zones de corrosion, comme nous l'avons fait avec les cartes de faible densité. Néanmoins, il semble ne pas exister de différence significative entre les cartes de formes complexes et les cartes

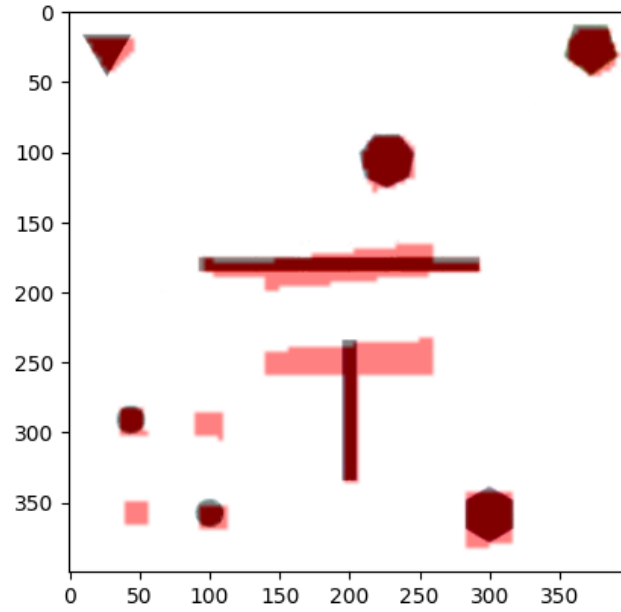


Figure 8 – Exemple de zone fantôme situé en bas à gauche de la carte.

de formes simples. Par exemple pour les cartes avec 15 formes de corrosion et la carte avec 15 formes de corrosion complexes, le score de Cohen ne varie que de 0.02 pour une distance $d = 1$ et de 0.04 pour une distance $d = 6$.

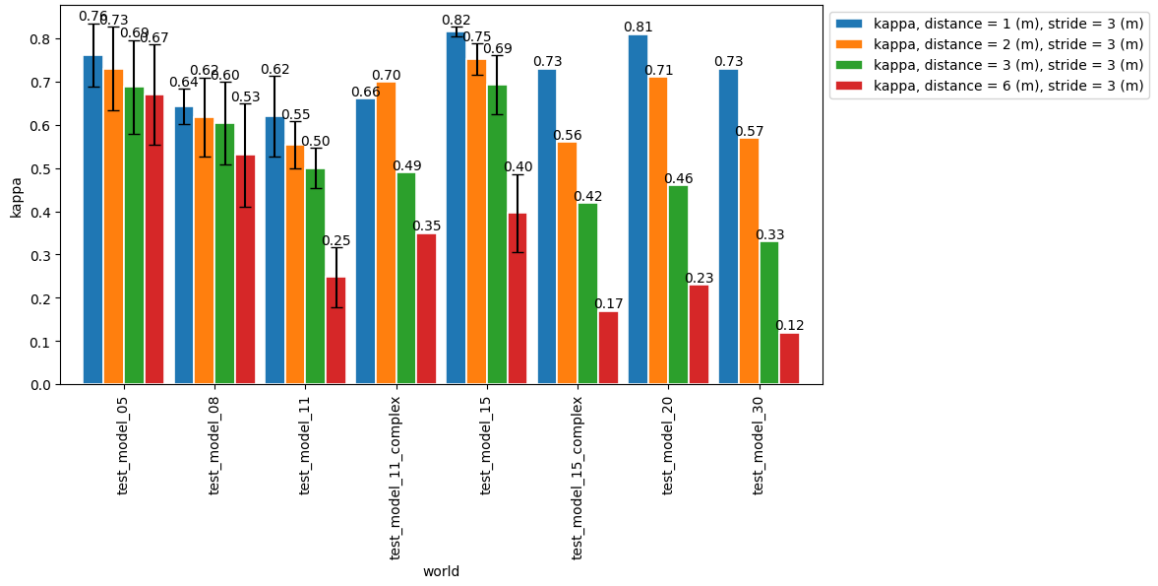
Dans la suite de ce rapport nous considérerons une distance $d = 3$ entre les deux crawlers pour l'algorithme *peinture au rouleau*.

Stratégie de navigation *ski nordique*

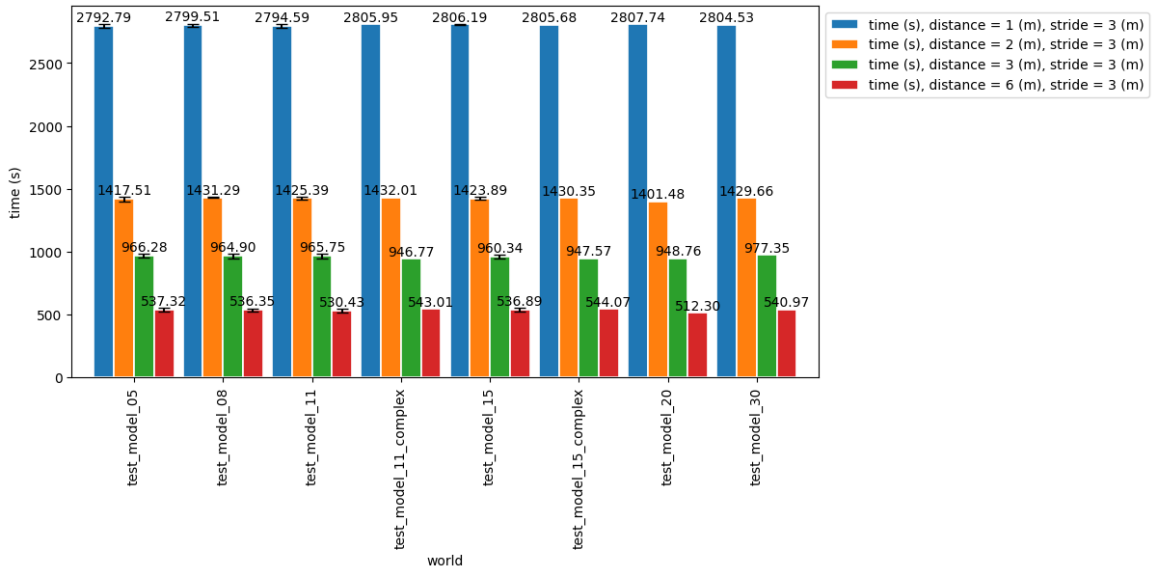
Nous allons maintenant analyser les résultats obtenus pour l'algorithme *ski nordique*. Comme pour l'algorithme *peinture au rouleau*, nous avons fait varier la densité du monde et la distance d entre les deux crawlers mais également le pas utilisé entre les deux crawlers. La figure 9 présente l'évolution du score de Cohen et du temps d'exécution de l'algorithme *ski nordique* en fonction de la densité du monde pour différentes valeurs de la distance d entre les deux crawlers et un pas de 3 mètres.

Nous avons des résultats très similaires à ceux obtenus pour l'algorithme *peinture au rouleau*. En effet, nous observons à la figure 9a que le score de Cohen diminue de manière générale lorsque la densité du monde augmente. De plus, le temps d'exécution de l'algorithme *ski nordique*, observé à la figure 9b, est constant pour chaque valeur de la densité du monde.

Nous pouvons observer à la figure 10 l'évolution du score de Cohen et du temps d'exécution de l'algorithme *ski nordique* en fonction de la densité du monde pour différentes valeurs du pas entre les deux crawlers et une distance entre les crawlers de 3 mètres. À la figure 10a, nous



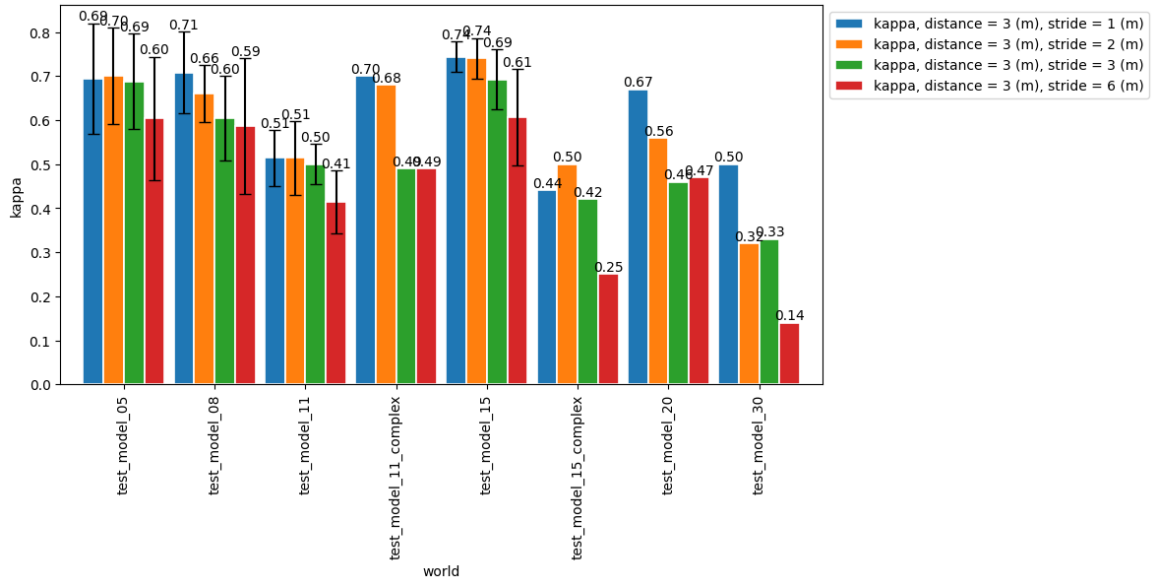
(a) κ en fonction de la densité du monde



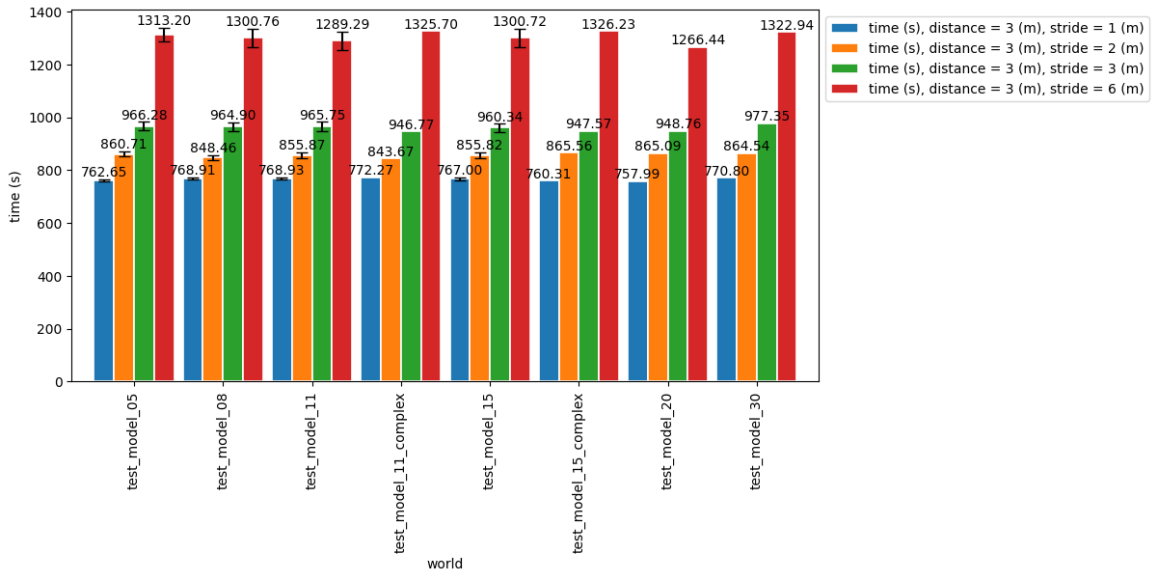
(b) Temps d'exécution en fonction de la densité du monde

Figure 9 – Évolution du score de Cohen et du temps d'exécution de l'algorithme de peinture au rouleau en fonction de la densité du monde pour différentes valeurs de la distance entre les deux crawlers.

observons que le score de Cohen est le plus bas pour des grandes valeurs de densités et des grandes valeurs de d , comme pour $d = 6$ et les cartes avec 30 et 20 zones de corrosion. Ceci



(a) κ en fonction de la densité du monde



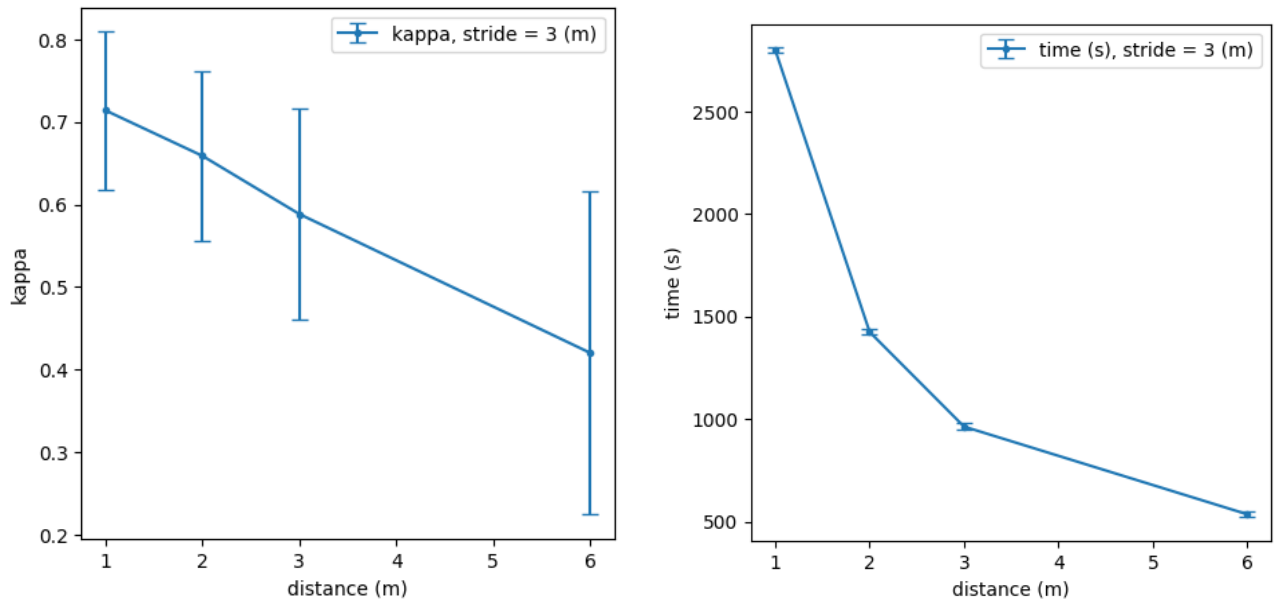
(b) Temps d'exécution en fonction de la densité du monde

Figure 10 – Évolution du score de Cohen et du temps d'exécution de l'algorithme de peinture au rouleau en fonction de la densité du monde pour différentes valeurs du pas entre les deux crawlers.

s'explique que pour de grandes valeurs de densités et de d , la probabilité que les rayons du signals traversent des zones de corrosions est plus élevée. Il y a donc plus de chance que des

zones fantômes soient créées ce qui fait diminuer le score de Cohen. Les formes allongées des zones de corrosion sont également un facteur qui fait diminuer le score de Cohen comme étudié précédemment. C'est pourquoi nous observons que le score de Cohen est le plus haut pour la carte avec la plus petite densité et sans formes allongées de corrosion, c'est-à-dire la carte avec 15 zones de corrosion.

À la figure 10b, nous observons que le temps d'exécution de l'algorithme *ski nordique* est constant pour chaque valeur de la densité du monde. Ceci était attendu comme pour la stratégie *peinture au rouleau*. Cependant, nous observons que le temps d'exécution avec le pas utilisé. Nous nous serions plutôt attendu à ce que le temps d'exécution reste constant avec le pas des crawlers. En effet, indépendamment de la valeur du pas, la distance verticale et horizontale à parcourir par les crawlers est la même. Cette différence significative du temps d'exécution est due à la manière dont nous avons implémenté l'algorithme *ski nordique* qui n'est pas optimale. Nous n'avons pas fait arrêter les crawlers aux extrémité des plaques mais nous les avons fait continuer d'une valeur du pas en plus par simplicité d'implémentation sans penser que l'impact sur le temps d'exécution serait significatif.

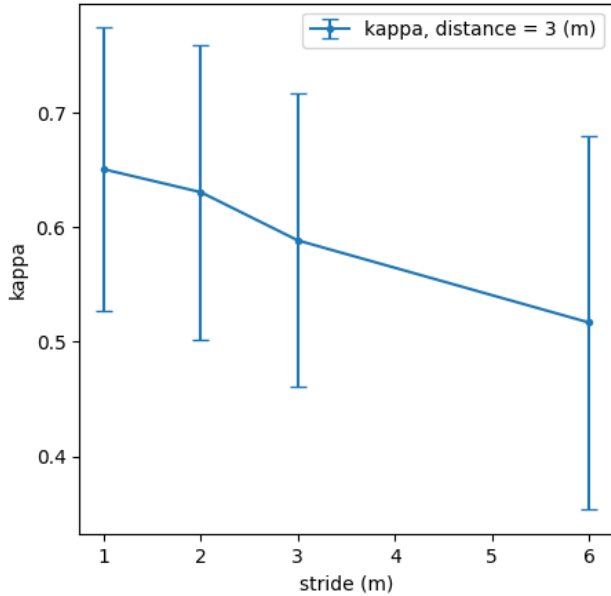


(a) κ en fonction de la distance entre les deux crawlers (b) Temps d'exécution en fonction de la distance entre les deux crawlers

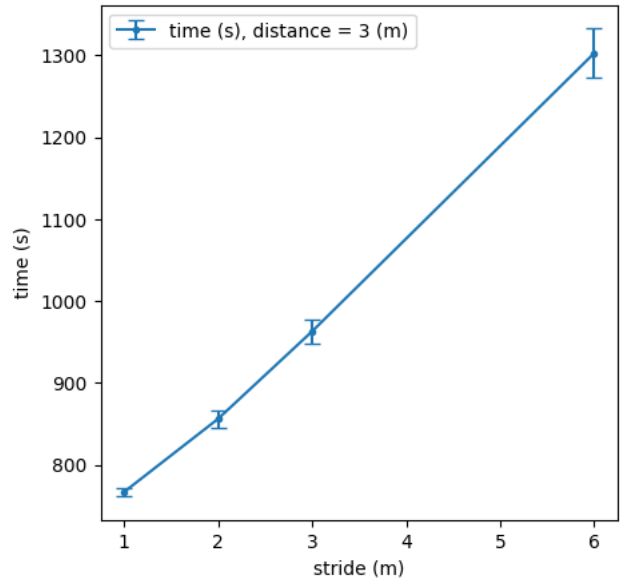
Figure 11 – Évolution du κ de Cohen et du temps d'exécution de l'algorithme *ski nordique* en fonction de la distance qui sépare les deux crawlers.

À la figure 11, nous observons l'évolution du score de Cohen et du temps d'exécution de l'algorithme *ski nordique* en fonction de la distance qui sépare les deux crawlers pour un pas de 3 mètres. Le score semble comme pour la stratégie *peinture au rouleau* suivre une relation

lineaire avec la distance qui sépare les deux crawlers. Le temps d'exécution semble également suivre une relation linéaire avec la distance qui sépare les deux crawlers. Le fait que la courbe à figure 11b ne soit pas une droite est dû au fait que plus la distance entre les crawlers est petite et plus le nombre de rotation que les crawlers doivent effectuer est grand. Or le temps de rotation n'est pas négligeable dans les temps d'exécution des algorithmes.



(a) κ en fonction du pas des crawlers



(b) Temps d'exécution en fonction du pas des crawlers

Figure 12 – Évolution du κ de Cohen et du temps d'exécution de l'algorithme *ski nordique* en fonction du pas des crawlers.

À la figure 12, nous observons l'évolution du score de Cohen et du temps d'exécution de l'algorithme *ski nordique* en fonction du pas des crawlers pour une distance de 3 mètres. Le score semble suivre une relation linéaire avec le pas des crawlers. Plus le pas est grand et plus le score est grand. Cela entre en concordance avec ce que nous expliquions précédemment. Plus le pas est grand et plus il y a de chance de créer des zones fantômes et donc de faire diminuer le score de Cohen. Le temps d'exécution lui semble suivre une relation linéaire avec le pas des crawlers. Comme expliqué précédemment, ce dernier aurait dû être linéaire mais notre implémentation fait dépendre le temps d'exécution du pas des crawlers.

Ici encore il semble que le score et le temps ne soient pas affectés par le fait que les formes soient convexes ou non.

Dans la suite de ce rapport nous considérerons une distance $d = 3$ entre les deux crawlers et un pas $s = 3$ pour l'algorithme *ski nordique*.

Stratégie de navigation *investigation polygonale*

TODO: Analyse

Comparaisons et discussions

6 Bilan personnel

TODO: Bilan personnel

7 Conclusion et perspectives

TODO: Conclusion et perspectives

References

- [1] Dharm Singh, Manoj Kumar Singh, Tarkeshwar Singh, and Rajkishore Prasad. Genetic algorithm for solving multiple traveling salesmen problem using a new crossover and population generation. *Computación y Sistemas*, 22, 07 2018.
- [2] András Király and János Abonyi. *Optimization of Multiple Traveling Salesmen Problem by a Novel Representation Based Genetic Algorithm*, pages 241–269. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.
- [3] ROS Task Manager. <https://dream.georgiatech-metz.fr/research-projects/ros-task-manager/>. Dernier accès : 2023-05-31.
- [4] Federico Bolelli, Stefano Allegretti, Lorenzo Baraldi, and Costantino Grana. Spaghetti labeling: Directed acyclic graphs for block-based connected components labeling. *IEEE Transactions on Image Processing*, PP:1–1, 10 2019.
- [5] Elad Kivelevitch. MDMTSPV_GA - multiple depot multiple traveling salesmen problem solved by genetic algorithm. https://www.mathworks.com/matlabcentral/fileexchange/31814-mdmtspv_ga-multiple-depot-multiple-traveling-salesmen-problem-solved-by-genetic-algorithm. Dernier accès : 2023-06-02.

Annexes

Annexe A: Implémentations techniques

Annexe A.1: Implémentation de l'algorithme *peinture au rouleau*

```
1 #!/usr/bin/env python3
2 import rospy
3 import numpy
4 from math import *
5 from task_manager_lib.TaskClient import *
6
7 rospy.init_node('task_client')
8 server_node = rospy.get_param("~server", "/crawler_id/task_server")
9 default_period = rospy.get_param("~period", 0.05)
10 tc = TaskClient(server_node, default_period)
11 rospy.loginfo("Mission connected to server: " + server_node)
12 vel = rospy.get_param("~velocity", 0.5)
13 crawler_id = rospy.get_param("~crawler_id", 0)
14 crawlers_distance = rospy.get_param("~crawlers_distance", 1.0)
15 overlap = rospy.get_param("~overlap", 0.1)
16 width = rospy.get_param("~width", 20.0)
17 height = rospy.get_param("~height", 20.0)
18
19 d = crawlers_distance
20 k = 0
21 stat = 0
22
23 start = rospy.Time.now()
24
25 try:
26     tc.SetStatusSync(status=stat)
27     if crawler_id == 0:
28         tc.WaitForStatusSync(partner="crawler_1", status=stat)
29     elif crawler_id == 1:
30         tc.WaitForStatusSync(partner="crawler_0", status=stat)
```

```

31 stat += 1
32
33 # vertical
34 if crawler_id == 0:
35     for i in numpy.arange(-width/2, width/2-d+1, d):
36         tc.AlignWithTarget(goal_x=i-overlap, goal_y=height/2*(-1)**k)
37         tc.FollowLine(goal_x=i-overlap, goal_y=height/2*(-1)**k, max_velocity=vel)
38         tc.AlignWithTarget(goal_x=i+d-overlap, goal_y=height/2*(-1)**k)
39         tc.FollowLine(goal_x=i+d-overlap, goal_y=height/2*(-1)**k, max_velocity=vel)
40         k = (k+1) % 2
41         tc.SetStatusSync(status=stat)
42         tc.WaitForStatusSync(partner="crawler_1", status=stat)
43         stat += 1
44 elif crawler_id == 1:
45     for i in numpy.arange(-width/2+d, width/2+1, d):
46         tc.AlignWithTarget(goal_x=i+overlap, goal_y=height/2*(-1)**k)
47         tc.FollowLine(goal_x=i+overlap, goal_y=height/2*(-1)**k, max_velocity=vel)
48         tc.AlignWithTarget(goal_x=i+d+overlap, goal_y=height/2*(-1)**k)
49         tc.FollowLine(goal_x=i+d+overlap, goal_y=height/2*(-1)**k, max_velocity=vel)
50         k = (k+1) % 2
51         tc.SetStatusSync(status=stat)
52         tc.WaitForStatusSync(partner="crawler_0", status=stat)
53         stat += 1
54
55 if crawler_id == 0:
56     tc.GoTo(goal_x=width/2+d, goal_y=-height/2, max_velocity=vel)
57     tc.SetStatusSync(status=stat)
58     tc.WaitForStatusSync(partner="crawler_1", status=stat)
59     stat += 1
60     tc.GoTo(goal_x=width/2, goal_y=-height/2, max_velocity=vel)
61     tc.SetStatusSync(status=stat)
62     tc.WaitForStatusSync(partner="crawler_1", status=stat)
63 elif crawler_id == 1:
64     tc.GoTo(goal_x=width/2+d, goal_y=-height/2+d, max_velocity=vel)
65     tc.SetStatusSync(status=stat)
66     tc.WaitForStatusSync(partner="crawler_0", status=stat)
67     stat += 1
68     tc.GoTo(goal_x=width/2, goal_y=-height/2+d, max_velocity=vel)
69     tc.SetStatusSync(status=stat)
70     tc.WaitForStatusSync(partner="crawler_0", status=stat)
71 stat += 1
72 k = 1
73
74 # horizontal
75 if crawler_id == 0:
76     for i in numpy.arange(-height/2, height/2-d+1, d):
77         tc.AlignWithTarget(goal_x=width/2*(-1)**k, goal_y=i-overlap)
78         tc.FollowLine(goal_x=width/2*(-1)**k, goal_y=i-overlap, max_velocity=vel)
79         tc.AlignWithTarget(goal_x=width/2*(-1)**k, goal_y=i+d-overlap)

```

```

80     tc.FollowLine(goal_x=width/2*(-1)**k, goal_y=i+d-overlap, max_velocity=vel)
81     k = (k+1) % 2
82     tc.SetStatusSync(status=stat)
83     tc.WaitForStatusSync(partner="crawler_1", status=stat)
84     stat += 1
85     elif crawler_id == 1:
86         for i in numpy.arange(-height/2+d, height/2+1, d):
87             tc.AlignWithTarget(goal_x=width/2*(-1)**k, goal_y=i+overlap)
88             tc.FollowLine(goal_x=width/2*(-1)**k, goal_y=i+overlap, max_velocity=vel)
89             tc.AlignWithTarget(goal_x=width/2*(-1)**k, goal_y=i+d+overlap)
90             tc.FollowLine(goal_x=width/2*(-1)**k, goal_y=i+d+overlap, max_velocity=vel)
91             k = (k+1) % 2
92             tc.SetStatusSync(status=stat)
93             tc.WaitForStatusSync(partner="crawler_0", status=stat)
94             stat += 1
95
96 except TaskException as e:
97     rospy.logerr("Exception caught: " + str(e))
98
99 end = rospy.Time.now()
100 time = (end-start).to_sec()
101 rospy.loginfo("Mission completed in " + str(time) + " seconds")
102

```

Listing 1 – Implémentation de l’algorithme de peinture au rouleau

Annexe A.2: Implémentation de l’algorithme *ski nordique*

```

1  #!/usr/bin/env python3
2  from time import sleep
3  import rospy
4  import numpy
5  from math import *
6  from task_manager_lib.TaskClient import *
7
8  rospy.init_node('task_client')
9  server_node = rospy.get_param("~server", "/crawler_id/task_server")
10 default_period = rospy.get_param("~period", 0.05)
11 tc = TaskClient(server_node, default_period)
12 rospy.loginfo("Mission connected to server: " + server_node)
13 vel = rospy.get_param("~velocity", 0.5)
14 crawler_id = rospy.get_param("~crawler_id", 0)
15 crawlers_distance = rospy.get_param("~crawlers_distance", 1.0)
16 stride_size = int(rospy.get_param("~stride_size", 1.0))
17 overlap = rospy.get_param("~overlap", 0.1)
18 grid_width = rospy.get_param("~grid_width", 6) // 2
19 grid_height = rospy.get_param("~grid_height", 6) // 2
20

```

```

21 d = crawlers_distance
22 k = 0
23 if crawler_id == 0: overlap = -overlap
24
25 start = rospy.Time.now()
26
27 # vertical pass
28
29 if crawler_id == 0: begin_w, end_w = -grid_width, grid_width + 1 - d
30 elif crawler_id == 1: begin_w, end_w = -grid_width + d, grid_width + 1
31
32 stat = 0
33 tc.SetStatusSync(status=stat)
34 if crawler_id == 0: tc.WaitForStatusSync(partner="crawler_1", status=stat+1)
35
36 for i in numpy.arange(begin_w, end_w, d):
37
38     if crawler_id == 0: begin_h, end_h = -grid_height + 2*stride_size, grid_height + 1 + stride_size
39     elif crawler_id == 1: begin_h, end_h = -grid_height + stride_size, grid_height + 1 + stride_size
40
41     for j in numpy.arange(begin_h, end_h, 2*stride_size):
42         tc.GoTo(goal_x=i+overlap, goal_y=j, max_velocity=vel, relative=False)
43         stat += 1
44         tc.SetStatusSync(status=stat)
45         if crawler_id == 0: tc.WaitForStatusSync(partner="crawler_1", status=stat+1)
46         elif crawler_id == 1: tc.WaitForStatusSync(partner="crawler_0", status=stat)
47
48     tc.SetStatusSync(status=stat+1)
49     if stride_size % 2 == 1:
50         if crawler_id == 0: tc.GoTo(goal_x=i+overlap, goal_y=grid_height + stride_size, max_velocity=vel,
51             ↳ relative=False)
52         if crawler_id == 1: tc.GoTo(goal_x=i+overlap, goal_y=grid_height, max_velocity=vel, relative=False)
53     elif stride_size % 2 == 0:
54         if crawler_id == 0: tc.GoTo(goal_x=i+overlap, goal_y=grid_height, max_velocity=vel, relative=False)
55         if crawler_id == 1: tc.GoTo(goal_x=i+overlap, goal_y=grid_height + stride_size, max_velocity=vel,
56             ↳ relative=False)
57     stat = 0
58     tc.SetStatusSync(status=stat)
59     if crawler_id == 1 and stride_size % 2 == 1: tc.WaitForStatusSync(partner="crawler_0", status=stat+1)
60     elif crawler_id == 0 and stride_size % 2 == 0: tc.WaitForStatusSync(partner="crawler_1", status=stat
61         ↳ +1)
62
63     if stride_size % 2 == 1:
64         if crawler_id == 0: begin_h, end_h = grid_height - stride_size, -grid_height - 1 - stride_size
65         elif crawler_id == 1: begin_h, end_h = grid_height - 2*stride_size, -grid_height - 1 - stride_size
66     elif stride_size % 2 == 0:
67         if crawler_id == 0: begin_h, end_h = grid_height - 2*stride_size, -grid_height - 1 - stride_size
68         elif crawler_id == 1: begin_h, end_h = grid_height - stride_size, -grid_height - 1 - stride_size

```



```

67 for j in numpy.arange(begin_h, end_h, -2*stride_size):
68     tc.GoTo(goal_x=i+overlap, goal_y=j, max_velocity=vel, relative=False)
69     stat += 1
70     tc.SetStatusSync(status=stat)
71     if crawler_id == 0 and stride_size % 2 == 1: tc.WaitForStatusSync(partner="crawler_1", status=stat)
72     if crawler_id == 0 and stride_size % 2 == 0: tc.WaitForStatusSync(partner="crawler_1", status=stat
    ↪ +1)
73     elif crawler_id == 1 and stride_size % 2 == 1: tc.WaitForStatusSync(partner="crawler_0", status=stat
    ↪ +1)
74     elif crawler_id == 1 and stride_size % 2 == 0: tc.WaitForStatusSync(partner="crawler_0", status=stat)
75
76     tc.SetStatusSync(status=stat+1)
77     if crawler_id == 0: tc.GoTo(goal_x=i+d+overlap, goal_y=-grid_height, max_velocity=vel, relative=
    ↪ False)
78     if crawler_id == 1: tc.GoTo(goal_x=i+d+overlap, goal_y=-grid_height - stride_size, max_velocity=vel
    ↪ , relative=False)
79     stat = 0
80     tc.SetStatusSync(status=stat)
81     if crawler_id == 0: tc.WaitForStatusSync(partner="crawler_1", status=stat+1)
82
83 tc.SetStatusSync(status=stat+1)
84
85 # repositioning
86
87 stat = 0
88 tc.SetStatusSync(status=stat)
89
90 if crawler_id == 0: tc.GoTo(goal_x=grid_width, goal_y=-grid_height, max_velocity=vel, relative=False)
91 elif crawler_id == 1: tc.GoTo(goal_x=grid_width + stride_size, goal_y=-grid_height + d, max_velocity
    ↪ =vel, relative=False)
92
93 if crawler_id == 0: tc.WaitForStatusSync(partner="crawler_1", status=stat+1)
94
95 # horizontal pass
96
97 if crawler_id == 0: begin_h, end_h = -grid_height, grid_height + 1 - d
98 elif crawler_id == 1: begin_h, end_h = -grid_height + d, grid_height + 1
99
100 # stat = 0
101 # tc.SetStatusSync(status=stat)
102 # if crawler_id == 0: tc.WaitForStatusSync(partner="crawler_1", status=stat+1)
103
104 for j in numpy.arange(begin_h, end_h, d):
105
106     if crawler_id == 0: begin_w, end_w = grid_width - 2*stride_size, -grid_width - 1 - stride_size
107     elif crawler_id == 1: begin_w, end_w = grid_width - stride_size, -grid_width - 1 - stride_size
108
109     for i in numpy.arange(begin_w, end_w, -2*stride_size):
110         tc.GoTo(goal_x=i, goal_y=j+overlap, max_velocity=vel, relative=False)

```

```

111     stat += 1
112     tc.SetStatusSync(status=stat)
113     if crawler_id == 0: tc.WaitForStatusSync(partner="crawler_1", status=stat+1)
114     elif crawler_id == 1: tc.WaitForStatusSync(partner="crawler_0", status=stat)
115
116     tc.SetStatusSync(status=stat+1)
117     if stride_size % 2 == 1:
118         if crawler_id == 0: tc.GoTo(goal_x=-grid_width - stride_size, goal_y=j+overlap, max_velocity=vel,
119             ↪ relative=False)
120         if crawler_id == 1: tc.GoTo(goal_x=-grid_width, goal_y=j+overlap, max_velocity=vel, relative=False
121             ↪ )
122     elif stride_size % 2 == 0:
123         if crawler_id == 0: tc.GoTo(goal_x=-grid_width, goal_y=j+overlap, max_velocity=vel, relative=False
124             ↪ )
125         if crawler_id == 1: tc.GoTo(goal_x=-grid_width - stride_size, goal_y=j+overlap, max_velocity=vel,
126             ↪ relative=False)
127     stat = 0
128     tc.SetStatusSync(status=stat)
129     if crawler_id == 1 and stride_size % 2 == 1: tc.WaitForStatusSync(partner="crawler_0", status=stat+1)
130     if crawler_id == 0 and stride_size % 2 == 0: tc.WaitForStatusSync(partner="crawler_1", status=stat+1)
131
132     if stride_size % 2 == 1:
133         if crawler_id == 0: begin_w, end_w = -grid_width + stride_size, grid_width + 1 + stride_size
134         elif crawler_id == 1: begin_w, end_w = -grid_width + 2*stride_size, grid_width + 1 + stride_size
135     elif stride_size % 2 == 0:
136         if crawler_id == 0: begin_w, end_w = -grid_width + 2*stride_size, grid_width + 1 + stride_size
137         elif crawler_id == 1: begin_w, end_w = -grid_width + stride_size, grid_width + 1 + stride_size
138
139     for i in numpy.arange(begin_w, end_w, 2*stride_size):
140         tc.GoTo(goal_x=i, goal_y=j+overlap, max_velocity=vel, relative=False)
141         stat += 1
142         tc.SetStatusSync(status=stat)
143         if crawler_id == 0 and stride_size % 2 == 1: tc.WaitForStatusSync(partner="crawler_1", status=stat)
144         if crawler_id == 0 and stride_size % 2 == 0: tc.WaitForStatusSync(partner="crawler_1", status=stat
145             ↪ +1)
146         elif crawler_id == 1 and stride_size % 2 == 1: tc.WaitForStatusSync(partner="crawler_0", status=stat
147             ↪ +1)
148         elif crawler_id == 1 and stride_size % 2 == 0: tc.WaitForStatusSync(partner="crawler_0", status=stat)
149
150     if crawler_id == 1: tc.SetStatusSync(status=stat+1)
151     if crawler_id == 0: tc.GoTo(goal_x=grid_width, goal_y=j+d+overlap, max_velocity=vel, relative=False)
152     if crawler_id == 1: tc.GoTo(goal_x=grid_width + stride_size, goal_y=j+d+overlap, max_velocity=vel,
153         ↪ relative=False)
154     stat = 0
155     tc.SetStatusSync(status=stat)
156     if crawler_id == 0: tc.WaitForStatusSync(partner="crawler_1", status=stat+1)
157
158     tc.SetStatusSync(status=stat+1)

```

```

153 end = rospy.Time.now()
154 time = (end-start).to_sec()
155 rospy.loginfo("Mission completed in " + str(time) + " seconds")
156
157

```

Listing 2 – Implémentation de l’algorithme de ski nordique

Annexe A.3: Implémentation de l’algorithme *investigation polygonale*

```

1  #!/usr/bin/env python3
2  # ROS specific imports
3  import roslib; roslib.load_manifest('floor_nav')
4  import rospy
5  import os
6  import tf
7  from task_manager_lib.TaskClient import *
8  from nav_msgs.msg import OccupancyGrid
9  from visualization_msgs.msg import MarkerArray
10 from visualization_msgs.msg import Marker
11 from geometry_msgs.msg import Point
12 from CrawlerOccupancyGrid import *
13 import numpy as np
14 from oct2py import octave
15 from math import sqrt
16 from time import sleep
17
18 DEFAULT_UWB_POWER_RX_THRESHOLD = 0
19 DEFAULT_N_CRAWLERS = 2
20 DEFAULT_N_POINTS = 4
21 DEFAULT_SCALE = 10
22 DEFAULT_DELTA = 5.0
23
24 def mission(centroids_map, polygons_map, radii_map, crawler_id, n_crawlers, n_points, cruise_velocity,
25             ↪ investigation_velocity, idx=None, tm=None):
26     print("Starting mission")
27     start = rospy.Time.now()
28
29     print(f'Number of crawlers: {n_crawlers}')
30
31     if n_crawlers == 2:
32         for centroid in range(len(centroids_map)):
33             if idx is not None and centroid < idx: continue
34
35             print(f'Investigating centroid {centroid}')
36
37             file = open("/home/chroma/Downloads/mission.txt", "w")
38             file.write(f'{centroid}\n')

```

```

38 end = rospy.Time.now()
39 if tm is not None: time = (end-start).to_sec() + tm
40 else: time = (end-start).to_sec()
41 file.write(f'{time}\n')
42 file.close()
43 os.system("rosservice call /save_occgrid")
44
45 tc.AlignWithTarget(goal_x=polygons_map[centroid][crawler_id][0], goal_y=polygons_map[centroid][
46 ↪ crawler_id][1], angle_threshold=0.05)
47 tc.FollowLine(goal_x=polygons_map[centroid][crawler_id][0], goal_y=polygons_map[centroid][
48 ↪ crawler_id][1], max_velocity=cruise_velocity)
49
50 first_turn = True
51
52 stat = 0
53 tc.SetStatusSync(status=stat)
54 if crawler_id == 0: tc.WaitForStatusSync(partner="crawler_1", status=stat)
55 elif crawler_id == 1: tc.WaitForStatusSync(partner="crawler_0", status=stat)
56
57 crawler_position = crawler_id
58 polygon = polygons_map[centroid]
59 radii = radii_map[centroid]
60
61 area_x = radii[0] - 0.1
62 area_w = radii[2] + 0.3
63 area_y = radii[1] - 0.1
64 area_h = radii[3] + 0.3
65
66 stat = 0
67 tc.SetStatusSync(status=stat)
68 if crawler_id == 1: tc.WaitForStatusSync(partner="crawler_0", status=stat+1)
69
70 for _ in range(0, n_points, n_crawlers):
71     w4Completion = tc.WaitForCompletion(x=area_x, y=area_y, w=area_w, h=area_h, foreground=
72 ↪ False)
73     tc.addCondition(ConditionIsCompleted("Completion detector", tc, w4Completion))
74     try:
75         for _ in range(1, n_points - n_crawlers + 1):
76             crawler_position = (crawler_position - 1) % n_points
77             tc.AlignWithTarget(goal_x=polygon[crawler_position][0], goal_y=polygon[crawler_position][1])
78             tc.FollowLine(goal_x=polygon[crawler_position][0], goal_y=polygon[crawler_position][1],
79 ↪ max_velocity=investigation_velocity)
80             first_turn = False
81         except TaskConditionException as e:
82             rospy.loginfo("Path following interrupted on condition: %s" % " or ".join([str(c) for c in e.conditions]))
83 ↪ )
84         except TaskException as e:
85             print(e)

```

```

82     tc.clearConditions()
83     tc.stopTask(w4Completion)
84
85     stat += 1
86     tc.SetStatusSync(status=stat)
87     if crawler_id == 0 and not first_turn: tc.WaitForStatusSync(partner="crawler_1", status=stat)
88     elif crawler_id == 1 and not first_turn and stat < n_points: tc.WaitForStatusSync(partner="
↪ crawler_0", status=stat+1)
89     # if crawler_id == 0: tc.WaitForStatusSync(partner="crawler_1", status=stat)
90     # elif crawler_id == 1 and stat < n_points: tc.WaitForStatusSync(partner="crawler_0", status=stat
↪ +1)
91
92     tc.SetStatusSync(status=stat+1)
93     tc.clearConditions()
94     tc.stopTask(w4Completion)
95 elif n_crawlers == 3:
96     for centroid in range(len(centroids_map)):
97         if idx is not None and centroid < idx: continue
98
99         print(f'Investigating centroid {centroid}')
100
101         file = open("/home/chroma/Downloads/mission.txt", "w")
102         file.write(f'{centroid}\n')
103         end = rospy.Time.now()
104         if tm is not None: time = (end-start).to_sec() + tm
105         else: time = (end-start).to_sec()
106         file.write(f'{time}\n')
107         file.close()
108         os.system("rosservice call /save_occgrid")
109
110         tc.AlignWithTarget(goal_x=polygons_map[centroid][crawler_id][0], goal_y=polygons_map[centroid][
↪ crawler_id][1], angle_threshold=0.05)
111         tc.FollowLine(goal_x=polygons_map[centroid][crawler_id][0], goal_y=polygons_map[centroid][
↪ crawler_id][1], max_velocity=cruise_velocity)
112
113         stat = 0
114         tc.SetStatusSync(status=stat)
115         if crawler_id == 0: tc.WaitForStatusSync(partner="crawler_1", status=stat)
116         if crawler_id == 0: tc.WaitForStatusSync(partner="crawler_2", status=stat)
117         if crawler_id == 1: tc.WaitForStatusSync(partner="crawler_0", status=stat)
118         if crawler_id == 1: tc.WaitForStatusSync(partner="crawler_2", status=stat)
119         if crawler_id == 2: tc.WaitForStatusSync(partner="crawler_0", status=stat)
120         if crawler_id == 2: tc.WaitForStatusSync(partner="crawler_1", status=stat)
121
122         print(f'Ready crawler {crawler_id}')
123
124         crawler_position = crawler_id
125         polygon = polygons_map[centroid]
126         radii = radii_map[centroid]

```

```

127
128 area_x = radii[0] - 0.1
129 area_w = radii[2] + 0.3
130 area_y = radii[1] - 0.1
131 area_h = radii[3] + 0.3
132
133 stat = 0
134 tc.SetStatusSync(status=stat)
135 if crawler_id == 1: tc.WaitForStatusSync(partner="crawler_0", status=stat+1)
136 elif crawler_id == 2: tc.WaitForStatusSync(partner="crawler_1", status=stat+1)
137
138 for _ in range(0, n_points, n_crawlers + crawler_id):
139     w4Completion = tc.WaitForCompletion(x=area_x, y=area_y, w=area_w, h=area_h, foreground=
↪ False)
140     tc.addCondition(ConditionIsCompleted("Completion detector", tc, w4Completion))
141     try:
142         for _ in range(0, n_points - n_crawlers):
143             crawler_position = (crawler_position - 1) % n_points
144             tc.AlignWithTarget(goal_x=polygon[crawler_position][0], goal_y=polygon[crawler_position][1])
145             tc.FollowLine(goal_x=polygon[crawler_position][0], goal_y=polygon[crawler_position][1],
↪ max_velocity=investigation_velocity)
146         except TaskConditionException as e:
147             rospy.loginfo("Path following interrupted on condition: %s" % " or ".join([str(c) for c in e.conditions]))
↪ )
148         except TaskException as e:
149             print(e)
150
151     tc.clearConditions()
152     tc.stopTask(w4Completion)
153
154     stat += 1
155     tc.SetStatusSync(status=stat)
156     if crawler_id == 0 and stat < 2: tc.WaitForStatusSync(partner="crawler_2", status=stat)
157     elif crawler_id == 1: tc.WaitForStatusSync(partner="crawler_0", status=stat+1)
158     elif crawler_id == 2: tc.WaitForStatusSync(partner="crawler_1", status=stat+1)
159
160 if crawler_id == 1 and stat == 1: tc.SetStatusSync(status=stat+1)
161 if crawler_id == 2 and stat == 1:
162     tc.AlignWithTarget(goal_x=polygons_map[centroid][1][0] - 0.5, goal_y=polygons_map[centroid][1][1]
↪ + 0.5, angle_threshold=0.05)
163     tc.FollowLine(goal_x=polygons_map[centroid][1][0] - 0.5, goal_y=polygons_map[centroid][1][1] + 0.5,
↪ max_velocity=cruise_velocity)
164     tc.Wait(duration=5.0)
165
166 end = rospy.Time.now()
167 if tm is not None: time = (end-start).to_sec() + tm
168 else: time = (end-start).to_sec()
169 rospy.loginfo("Mission completed in " + str(time) + " seconds")
170

```

```

171 os.system("rm /home/chroma/Downloads/mission.txt")
172
173 def solve_TSP(centroids_map):
174     import math
175     from itertools import combinations
176     import gurobipy as gp
177     from gurobipy import GRB
178
179     # Callback – use lazy constraints to eliminate sub-tours
180     def subtourelim(model, where):
181         if where == GRB.Callback.MIPSOL:
182             vals = model.cbGetSolution(model._vars)
183             # find the shortest cycle in the selected edge list
184             tour = subtour(vals)
185             if len(tour) < n:
186                 # add subtour elimination constr. for every pair of cities in tour
187                 model.cbLazy(gp.quicksum(model._vars[i, j] for i, j in combinations(tour, 2)) <= len(tour)-1)
188
189     # Given a tuplelist of edges, find the shortest subtour
190     def subtour(vals):
191         # make a list of edges selected in the solution
192         edges = gp.tuplelist((i, j) for i, j in vals.keys() if vals[i, j] > 0.5)
193         unvisited = list(range(n))
194         cycle = range(n+1) # initial length has 1 more city
195         while unvisited: # true if list is non-empty
196             thiscycle = []
197             neighbors = unvisited
198             while neighbors:
199                 current = neighbors[0]
200                 thiscycle.append(current)
201                 unvisited.remove(current)
202                 neighbors = [j for i, j in edges.select(current, '*') if j in unvisited]
203             if len(cycle) > len(thiscycle):
204                 cycle = thiscycle
205         return cycle
206
207     n = len(centroids_map)
208     points = [(centroids_map[i][0], centroids_map[i][1]) for i in range(n)]
209
210     # Dictionary of Euclidean distance between each pair of points
211     dist = {(i, j): math.sqrt(sum((points[i][k]-points[j][k])**2 for k in range(2))) for i in range(n) for j in range(i
        ↪ )}
212
213     m = gp.Model()
214
215     # Create variables
216     vars = m.addVars(dist.keys(), obj=dist, vtype=GRB.BINARY, name='e')
217     for i, j in vars.keys():
218         vars[j, i] = vars[i, j] # edge in opposite direction

```

```

219
220 # Add degree-2 constraint
221 m.addConstrs(vars.sum(i, '*') == 2 for i in range(n))
222
223 # Optimize model
224 m._vars = vars
225 m.Params.LazyConstraints = 1
226 m.optimize(subtourelim)
227
228 vals = m.getAttr('X', vars)
229 tour = subtour(vals)
230 assert len(tour) == n
231
232 print(f'Optimal tour: {tour}')
233 print(f'Optimal cost: {m.objVal}')
234
235 return tour
236
237 def solve_mTSP(centroids_map, perimeters_map, n_crawlers):
238     """
239     Returns a permutation of the centroids_map that minimizes the total distance traveled by the crawlers
240     """
241
242     def d(x1, y1, x2, y2):
243         return sqrt((x1-x2)**2 + (y1-y2)**2) + 50.0
244
245     octave.addpath('/home/chroma/Documents/Multi-
246         ↳ robot_navigation_and_control_for_acoustic_inspection_of_metal_plate_structures/scripts/')
247     octave.eval('pkg load statistics')
248
249     XY = np.array(centroids_map)
250     max_salesmen = n_crawlers
251     depots = np.array([
252         [-3, -4.0],
253         [3.0, -4.0]
254     ])
255     CostType = 2
256     MIN_TOUR = 1
257     POP_SIZE = 256
258     NUM_ITER = 100
259     SHOW_PROG = True
260     SHOW_RES = True
261     DMAT = np.zeros((len(XY), len(XY)))
262     for i in range(len(XY)):
263         for j in range(len(XY)):
264             DMAT[i, j] = d(XY[i, 0], XY[i, 1], XY[j, 0], XY[j, 1])
265     [min_dist, best_tour, generation] = octave.mdmtspv_ga(XY, max_salesmen, depots, CostType,
266         ↳ MIN_TOUR, POP_SIZE, NUM_ITER, SHOW_PROG, SHOW_RES, DMAT, nout=3)
267     print("Minimum distance: ")

```



```

266 print(min_dist)
267 print("Best tour: ")
268 print(best_tour)
269 print("Generation: ")
270 print(generation)
271
272 sleep(10000000)
273
274 if __name__ == '__main__':
275     rospy.init_node('task_client')
276     server_node = rospy.get_param("~server", "/crawler_id/task_server")
277     default_period = rospy.get_param("~period", 0.05)
278     tc = TaskClient(server_node, default_period)
279     investigation_velocity = rospy.get_param("~investigation_velocity", 0.3)
280     cruise_velocity = rospy.get_param("~cruise_velocity", 1.)
281     uwb_power_rx_threshold = rospy.get_param("~uwb_power_rx_threshold",
        ↪ DEFAULT_UWB_POWER_RX_THRESHOLD)
282     n_crawlers = rospy.get_param("~n_crawlers", DEFAULT_N_CRAWLERS)
283     n_points = rospy.get_param("~n_points", DEFAULT_N_POINTS)
284     scale = rospy.get_param("~scale", DEFAULT_SCALE)
285     delta = rospy.get_param("~delta", DEFAULT_DELTA)
286     crawler_id = rospy.get_param("~crawler_id", 0)
287     depot_x = rospy.get_param("~depot_x", -3.0)
288     depot_y = rospy.get_param("~depot_y", -3.0)
289     filename = rospy.get_param("~filename", "")
290     occ_grid = CrawlerOccupancyGrid(
291         rospy.get_param("~grid_width", rospy.get_param("~grid_width", DEFAULT_GRID_WIDTH)),
292         rospy.get_param("~grid_height", rospy.get_param("~grid_height", DEFAULT_GRID_HEIGHT)),
293         rospy.get_param("~grid_resolution", rospy.get_param("~grid_resolution",
        ↪ DEFAULT_GRID_RESOLUTION))
294     )
295
296     listener = tf.TransformListener()
297
298     pub_centroid = rospy.Publisher("~circle", MarkerArray, queue_size=1)
299     pub_polygone = rospy.Publisher("~poly", MarkerArray, queue_size=1)
300     pub_occgrid = rospy.Publisher("~occ_grid", OccupancyGrid, queue_size=1)
301
302     occ_grid.restore_from_image(filename)
303
304     centroids_grid, radii_grid = occ_grid.find_centroid_of_clusters()
305     centroids_map = []
306     radii_map = []
307     for i in range(len(centroids_grid)):
308         centroids_map.append((centroids_grid[i][0] * occ_grid.resolution - occ_grid.width / 2.0, centroids_grid[i]
        ↪ ][1] * occ_grid.resolution - occ_grid.height / 2.0))
309         radii_map.append((radii_grid[i][0] * occ_grid.resolution - occ_grid.width / 2.0, radii_grid[i][1] *
        ↪ occ_grid.resolution - occ_grid.height / 2.0, radii_grid[i][2] * occ_grid.resolution, radii_grid[i][3] *
        ↪ occ_grid.resolution))

```

```

310 polygons_grid = []
311 polygons_map = []
312 for i in range(len(centroids_grid)):
313     vertices_grid = occ_grid.get_polygone_around_centroid(n_points, centroids_grid[i], radii_grid[i][2:4] /
314         ↪ 2.0, scale, delta)
315     vertices_map = []
316     for j in range(len(vertices_grid)):
317         vertices_map.append((vertices_grid[j][0] * occ_grid.resolution - occ_grid.width / 2.0, vertices_grid[j]
318             ↪ [1] * occ_grid.resolution - occ_grid.height / 2.0))
319     polygons_grid.append(vertices_grid)
320     polygons_map.append(vertices_map)
321
322 perimeters_grid = []
323 perimeters_map = []
324 for i in range(len(centroids_grid)):
325     perimeters_grid.append(occ_grid.get_perimeter_of_polygone(polygons_grid[i]))
326     perimeters_map.append(occ_grid.get_perimeter_of_polygone(polygons_map[i]))
327
328 tour = solve_TSP(centroids_map)
329 depot = (depot_x, depot_y)
330 # find closest centroid to depot
331 min_dist = 100000
332 min_index = 0
333 for i in range(len(centroids_map)):
334     dist = sqrt((centroids_map[i][0] - depot[0]) ** 2 + (centroids_map[i][1] - depot[1]) ** 2)
335     if dist < min_dist:
336         min_dist = dist
337         min_index = i
338 # shift tour so that the closest centroid to depot is the first one
339 tour = tour[min_index:] + tour[:min_index]
340
341 # reorder centroids, radii, polygons and perimeters according to tour
342 centroids_map = [centroids_map[i] for i in tour]
343 radii_map = [radii_map[i] for i in tour]
344 polygons_map = [polygons_map[i] for i in tour]
345 perimeters_map = [perimeters_map[i] for i in tour]
346
347 print(f"tour: {tour}")
348 print(f"centroids_map: {centroids_map}")
349
350 if os.path.isfile("/home/chroma/Downloads/occupancy_grid.png"):
351     occ_grid.restore_from_image("/home/chroma/Downloads/occupancy_grid.png")
352
353 marker_array = MarkerArray()
354 for i in range(len(centroids_map)):
355     marker = Marker()
356     marker.header.frame_id = "world"
357     marker.header.stamp = rospy.Time.now()
358     marker.ns = "circle"

```

```

357     marker.id = i
358     marker.type = Marker.CYLINDER
359     marker.action = Marker.ADD
360     marker.pose.position.x = centroids_map[i][0]
361     marker.pose.position.y = centroids_map[i][1]
362     marker.pose.position.z = 0.0
363     marker.pose.orientation.x = 0.0
364     marker.pose.orientation.y = 0.0
365     marker.pose.orientation.z = 0.0
366     marker.pose.orientation.w = 1.0
367     marker.scale.x = 0.1
368     marker.scale.y = 0.1
369     marker.scale.z = 0.1
370     marker.color.a = 1.0
371     marker.color.r = 0.0
372     marker.color.g = 1.0
373     marker.color.b = 0.0
374     marker_array.markers.append(marker)
375
376 marker_array_2 = MarkerArray()
377 for i in range(len(centroids_map)):
378     for j in range(n_points):
379         marker = Marker()
380         marker.header.frame_id = "world"
381         marker.header.stamp = rospy.Time.now()
382         marker.ns = "poly"
383         marker.id = i * n_points + j
384         marker.type = Marker.LINE_STRIP
385         marker.action = Marker.ADD
386         marker.pose.position.x = 0.0
387         marker.pose.position.y = 0.0
388         marker.pose.position.z = 0.0
389         marker.pose.orientation.x = 0.0
390         marker.pose.orientation.y = 0.0
391         marker.pose.orientation.z = 0.0
392         marker.pose.orientation.w = 1.0
393         marker.scale.x = 0.01
394         marker.scale.y = 0.01
395         marker.scale.z = 0.01
396         marker.color.a = 1.0
397         marker.color.r = 1.0
398         marker.color.g = 0.0
399         marker.color.b = 0.0
400         point = Point()
401         point.x = polygons_map[i][j][0]
402         point.y = polygons_map[i][j][1]
403         point.z = 0.0
404         marker.points.append(point)
405         point = Point()

```

```

406     point.x = polygons_map[i][(j + 1) % n_points][0]
407     point.y = polygons_map[i][(j + 1) % n_points][1]
408     point.z = 0.0
409     marker.points.append(point)
410     marker_array_2.markers.append(marker)
411
412     pub_centroid.publish(marker_array)
413     pub_polygone.publish(marker_array_2)
414     occ_grid.publish(pub_occgrid, rospy.Time.now())
415
416     if os.path.isfile("/home/chroma/Downloads/mission.txt"):
417         file = open("/home/chroma/Downloads/mission.txt", "r")
418         lines = file.readlines()
419         file.close()
420         idx = int(lines[0])
421         tm = float(lines[1])
422         mission(centroids_map, polygons_map, radii_map, crawler_id, n_crawlers, n_points, cruise_velocity,
423                 ↪ investigation_velocity, idx, tm)
424     else:
425         mission(centroids_map, polygons_map, radii_map, crawler_id, n_crawlers, n_points, cruise_velocity,
426                 ↪ investigation_velocity)

```

Listing 3 – Implémentation de l’algorithme d’investigation polygonale

Annexe B: Environnements de test

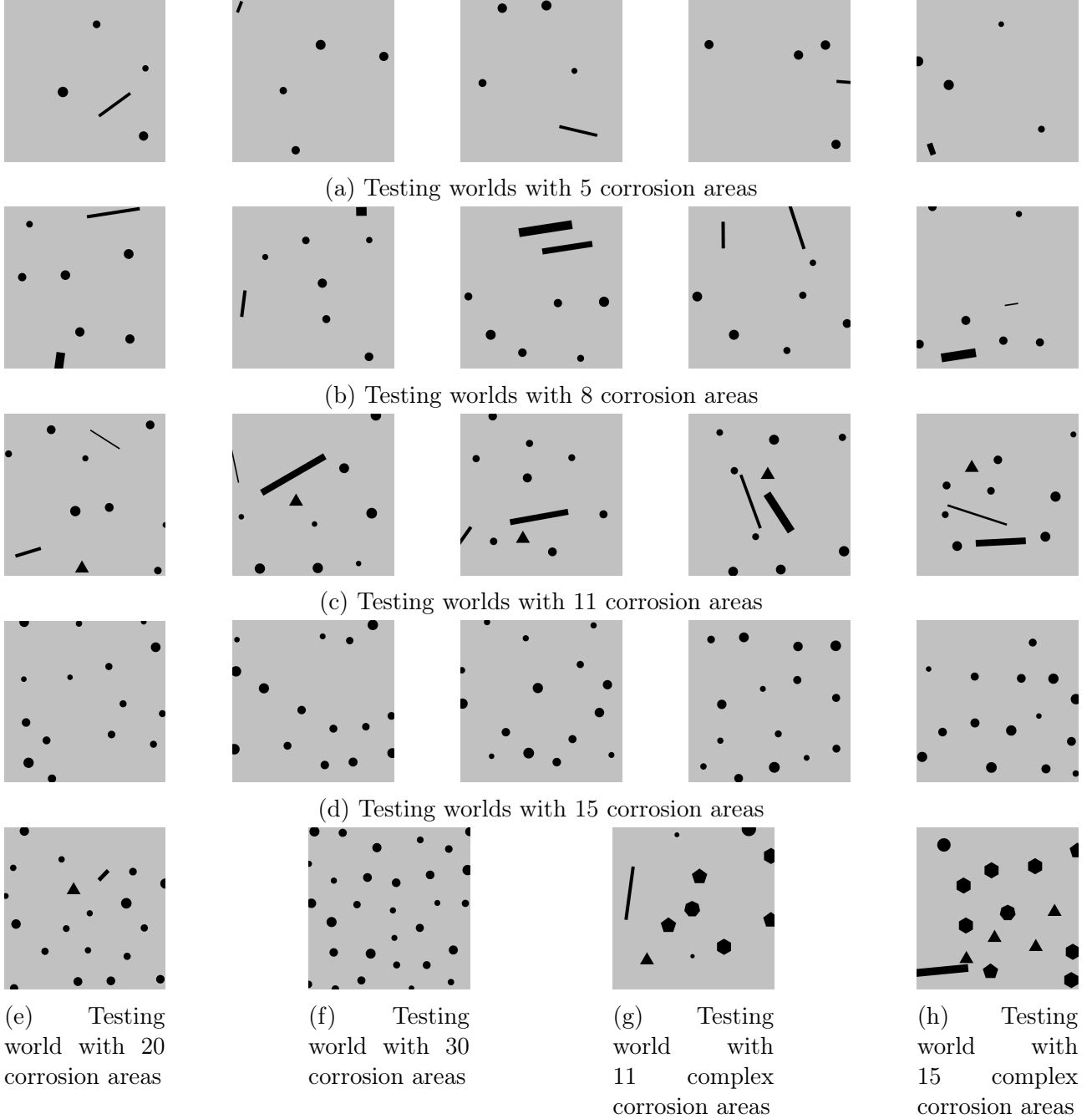


Figure 13 – Différents environnements de test.

Annexe C: Résultat d'investigation

TODO: figure avec la superposition des images

Navigation et contrôle multi-robots pour l'inspection acoustique de structures métalliques [R&D]

Brandon ALVES

Résumé

Ce projet fait partie du projet européen BugWright2 qui a pour objectif de résoudre la problématique de l'inspection de grandes structures métalliques en utilisant des flottes hétérogènes de robots mobiles. Le projet se concentrera sur le développement de stratégies de navigation pour des robots mobiles utilisant des ondes ultrasoniques guidées pour réaliser l'inspection de plaques métalliques. Les ondes guidées ont la capacité de se propager le long d'une plaque en interagissant avec la matière qui la compose et en étant affectées par des changements de géométrie, tels que la corrosion. En combinant des mesures entre un système émetteur et un système récepteur distant, il est possible de réaliser une tomographie de la zone à inspecter et de potentiellement identifier et localiser des points de corrosion.

Mots-clés : Navigation ; Multi-Robot ; Tomographie ; Ondes Guidées Ultrasoniques ; Inspection.

Abstract

This project is part of the European project BugWright2, which aims to address the problem of inspecting large metal structures using heterogeneous fleets of mobile robots. The project will focus on developing navigation strategies for mobile robots using guided ultrasonic waves to perform the inspection of metal plates. Guided waves have the ability to propagate along a plate by interacting with the material that makes it up and being affected by changes in geometry, such as corrosion. By combining measurements between a transmitter and a distant receiver system, it is possible to perform a tomography of the area to be inspected and potentially identify and locate points of corrosion.

Keywords: Navigation ; Multi-Robot ; Tomography ; Ultrasonic Guided Waves ; Inspection.