Architectures pour des systèmes de localisation et de cartographie simultanées

Bastien Vincke

UNIVERSITÉ PARIS SUD

Comprendre le monde, construire l'avenir®

UNIVERSITE PARIS-SUD

ÉCOLE DOCTORALE : STITS
Institut d'Electronique Fondamentale

***DISCIPLINE** PHYSIQUE*

THÈSE DE DOCTORAT

soutenue le 03/12/2012

par

Bastien VINCKE

Architectures pour des systèmes de localisation et de cartographie simultanées

Directeur de thèse :	Alain MERIGOT	Professeur, Université Paris-Sud, Orsay
Encadrants de thèse :	Abdelhafid EL OUARDI	Maître de Conférences, Université Paris-Sud, Orsay
	Alain LAMBERT	Chargé de Recherche, IFSTTAR, LIVIC, Versailles

Composition du jury :

Rapporteurs :	Etienne COLLE	Professeur, Université Val d'Essonne, Evry
	Michel DEVY	Directeur de Recherche, LAAS, Toulouse
Examinateur :	Mohamed SHAWKY	Professeur, Université de Technologie, Compiègne

Bastien Vincke : Architectures pour des systèmes de localisation et de cartographie simultanées, 2012

Remerciements

En premier lieu, je tiens à remercier Monsieur Michel Devy pour m'avoir fait l'honneur de présider mon jury de thèse, mais également pour son rôle en tant que rapporteur de ma thèse. Ces remarques et conclusions m'ont été d'une grande utilité pour tirer les enseignements de mon travail de thèse. Je tiens également à remercier Monsieur Etienne Colle pour avoir accepté de rapporter mon travail de thèse et Monsieur Mohamed Shawky pour accepté d'examiner mes travaux. Leurs remarques, corrections et questions m'ont permis d'améliorer mon manuscrit et de valoriser mon travail.

Je remercie également Monsieur Alain Mérigot, mon directeur de thèse, pour ces conseils et orientations, Monsieur Abdelhafid Elouardi pour m'avoir encadré, guidé, orienté, aidé et soutenu durant l'intégralité de ma thèse. Enfin, je remercie Monsieur Alain Lambert pour son encadrement, ses précieux conseils algorithmiques et son aide importante lors de la rédaction de publications.

J'adresse également mes plus sincères remerciements à l'ensemble de l'équipe du département ACCIS de l'institut d'électronique fondamentale, ayant pour responsable Monsieur Roger Reynaud, pour m'avoir permis de réaliser mes travaux de thèses au sein du département. Je remercie également l'équipe SAAVE de Monsieur Samir Bouaziz pour m'avoir fait entièrement confiance et m'avoir intégrer dans leur programme de recherche.

Je tiens en particulier à remercier Franck Bimbard pour ses multiples conseils et sa relecture approfondie de mon manuscrit et Michèle Gouiffes pour ses remarques et conseils concernant ma présentation. Je pense enfin que ma thèse n'aurait pas été la même sans mes deux collègues de bureau Cyrille André et Arnaud Roquel. Nos discussions animées, autour d'un bon café Senseo, m'ont très souvent ouvert l'esprit et orienté dans la bonne direction.

Finalement mes derniers remerciements iront à mes frères Matthieu et Guillaume pour tous les bons moments passés et à mes parents sans qui je n'aurai pas réalisé le parcours universitaire que j'ai suivi. Je vous serai éternellement reconnaissant de m'avoir encouragé dans les moments difficiles. Enfin, je tiens à remerciement la personne sans qui rien n'aurait été possible, ma fiancée Céline, qui m'a soutenue et poussée durant ces trois longues années.

Résumé

La robotique mobile est un domaine en plein essor. Aujourd'hui, de nombreuses recherches sont effectuées pour accroître l'autonomie des robots. L'un des domaines de recherche consiste à permettre à un robot de cartographier son environnement tout en se localisant dans l'espace. Couplés à un algorithme de planification, ces techniques permettent à un robot d'explorer des espaces inconnus. Les techniques couramment employées de SLAM (Simultaneous Localization And Mapping) restent généralement coûteuses en termes de puissance de calcul. En effet, les robots mobiles sont généralement équipés d'un ordinateur portable ayant d'importantes capacités de calculs. La tendance actuelle vers la miniaturisation impose de restreindre les ressources embarquées. L'ensemble de ces constatations nous ont guidés vers l'intégration d'algorithmes de SLAM sur des architectures adéquates dédiées pour l'embarqué.

Le problème de localisation et de cartographie simultanées requière l'utilisation de capteurs fournissant des informations sur l'environnement mais aussi sur les déplacements du robot. Classiquement, le SLAM peut être résolu en utilisant des odomètres ainsi qu'un télémètre laser. Cependant, l'utilisation de caméras est de plus en plus courante pour résoudre la problématique du SLAM. Les algorithmes SLAM sont rarement implantés sur des systèmes embarqués. Aujourd'hui, l'évolution de la technologie des calculateurs permet de disposer de nombreuses architectures ayant des spécificités rarement utilisées dans le contexte du SLAM.

Les premiers travaux de cette thèse ont consisté à définir une architecture permettant à un robot mobile de se localiser. Cette architecture doit respecter certaines contraintes, notamment celle du temps réel, des dimensions réduites et de la faible consommation énergétique. Nous avons développé un système incluant des capteurs extéroceptifs et proprioceptifs, des architectures hétérogène et homogène et les interfaces associées. L'implantation optimisée d'un algorithme (EKF-SLAM) a permis de démontrer l'efficacité de l'adéquation algorithme architecture permettant à un robot mobile de se localiser en temps réel. Notre approche nous a amené à étudier l'algorithme en le découpant sous forme de blocs fonctionnels. Il a été nécessaire de revoir l'implantation de chaque bloc en utilisant au mieux les spécificités architecturales du système (capacités des processeurs, implantation multi-cœurs, calcul vectoriel ou parallélisation avec processeur DSP). Notre étude montre l'importance de l'adéquation algorithme

architecture en particulier pour des systèmes à fortes contraintes.

Durant la seconde partie de ma thèse, une deuxième approche a été explorée ayant pour objectif la définition d'un système à base d'une architecture reconfigurable. L'algorithme choisi est le FastSLAM, car après avoir étudié les algorithmes existants, il est apparu que cet algorithme pouvait bénéficier d'une architecture fortement parallèle. Nous avons donc choisi de concevoir une architecture matérielle à base de FPGA. L'architecture définie permet de cartographier un environnement plus large que celle définie pour le premier système à base d'une architecture hétérogène et a été évaluée en utilisant une méthodologie HIL (Hardware in the Loop).

La troisième partie de cette thèse a été consacrée à l'exploration des algorithmes SLAM basés sur la théorie ensembliste. En effet, les principaux algorithmes de SLAM sont conçus autour de la théorie des probabilités. Ces théories permettent une localisation précise mais ne garantissent en aucun cas les résultats de localisation. Un algorithme de SLAM basé sur la théorie ensembliste a été défini garantissant l'ensemble des résultats obtenus. Plusieurs améliorations algorithmiques sont ensuite proposées. Elles sont évaluées par simulation et expérimentation. Une comparaison avec les algorithmes probabilistes a mis en avant la robustesse de l'approche ensembliste.

Ces travaux de thèse ont permis de mettre en avant deux contributions principales. La première consiste à affirmer l'importance d'une conception algorithme-architecture pour résoudre la problématique du SLAM. La seconde est la définition d'une méthode ensembliste permettant de garantir les résultats de localisation.

Publications

Revue internationale avec comité de lecture

1. "Real Time Simultaneous Localization and Mapping : Towards Low-cost Multiprocessor Embedded Systems"
   B. Vincke, A. Elouardi, A. Lambert, EURASIP Journal on Embedded Systems, SpringerOpen, Juin 2012. DOI :10.1186/1687-3963-2012-5

Congrès internationaux avec actes et comités de lecture

1. "Efficient Implementation of EKF-SLAM on a Multi-Core Embedded System"
   B. Vincke, A. Elouardi, A. Lambert, A. Merigot, IECON 2012. Montreal, Canada. Octobre 27-28, 2012.
2. "Experimental Comparison of Bounded-Error State Estimation and Constraints Propagation"
   B. Vincke, A. Lambert, International Conference on Robotics and Automation ICRA 2011. Shanghai, Chine. Mai 9-13, 2011 ISBN : 978-1-61284-386-5, Pages 4724 - 4729
3. "Multiprocessing Improvements on a Low-cost System Based Simultaneous Localization and Mapping"
   B. Vincke, A. Elouardi, A. Lambert, International Conference on Multimedia Computing and Systems ICMCS 2011. Ouarzazate, Maroc. Avril 7-9, 2011 ISBN : 978-1-61284-730-6, Pages 1-5
4. "Design and Evaluation of an Embedded System Based SLAM Applications"
   B. Vincke, A. Elouardi, A. Lambert International Symposium on System Integration SII 2010. Sendai, Japan. December 21-22, 2010 ISBN : 978-1-4244-9314-2, Pages 224 - 229
5. "A Practical Approach for EKF-SLAM Implementation Using a Low-cost System"
   B. Vincke, A. Elouardi, A. Lambert, International Conference on Test and Measurement, ICTM 2010. Phuket, Thailand. December 1-2, 2010 ISBN : 978-1-61284-031-4, Pages 402 - 405
6. "Static and Dynamic Fusion for Outdoor Vehicle Localization"
   B. Vincke, A. Lambert, D. Gruyer, A. Elouardi, E. Seignez, International Conference on Control, Automation, Robotics and Vision, ICARCV 2010. Singapore 7-10 December 2010, ISBN : 978-1-4244-7813-2, Pages 437 - 442
7. "Consistent outdoor vehicle localization by bounded-error state estimation"
   A. Lambert, D. Gruyer, B. Vincke, E. Seignez International Conference on Intelligent Robots and Systems, IROS 2009. Saint Louis, USA 11-15 October ISBN : 978-1-4244-3803-7, Pages 1211-1216

Table des matières

Table des figures

Liste des tableaux

Introduction Générale

Depuis plusieurs années, le secteur de la robotique est en plein essor. De plus en plus de systèmes robotisés sont intégrés dans des produits grand public comme par exemple les robots-aspirateur, les robots de surveillance ou encore les jouets. Ces systèmes acquièrent de jour en jour de nouvelles capacités, comme par exemple dans les domaines de la localisation ou de la cartographie. Couplés à un algorithme de planification, ces techniques permettent à un robot d'explorer des espaces inconnus. Ces capacités sont d'autant plus importantes qu'elles sont indispensables à l'autonomie de tout robot.

La localisation d'un robot consiste à chercher sa position dans un espace connu. La cartographie revient, quant à elle, a modéliser l'environnement entourant le robot. Le SLAM (Simultaneous Localization And Mapping) consiste à résoudre simultanément ces deux problèmatiques. Les robots bénéficient alors de connaissances accrues du monde qui les entoure : ils peuvent ainsi se déplacer de manière autonome dans des espaces inconnus.

Le problème de localisation et de cartographie simultanées requiert l'utilisation de capteurs fournissant des informations sur l'environnement mais aussi sur les déplacements du robot. Les premiers algorithmes de SLAM étaient principalement basés sur l'utilisation de télémètres lasers à balayage. Ce type de capteur fournit un ensemble de distances très précises rendant possible une cartographie rapide et efficace de l'environnement. Malheureusement, ces capteurs restent généralement très coûteux. Neato a développé son propre télémètre laser (Figure 1), adapté à une utilisation en environnement d'intérieur, en réduisant son coût de production afin de pouvoir l'intégrer dans un produit grand public. Traditionnellement, un télémètre laser coûte plus d'un millier d'euros. Neato a réduit le cout de production à environ 30 euros, permettant d'inclure ce capteur dans leur aspirateur autonome vendu environ 500 euros dans le commerce. Ce capteur rend le produit de Neato très intéressant et surtout efficace par rapport aux autres produits concurrents. Ce type d'optimisation rend envisageable l'utilisation du SLAM dans des produits grand public permettant ainsi une utilisation efficace de la localisation et de la cartographie simultanées.

Le SLAM ne se focalise pas sur l'utilisation de télémètre laser, en particulier de nombreuses recherches sont effectuées pour reconstruire des environnements à l'aide de caméras. L'utilisation de plusieurs caméras permet d'obtenir une cartographie en trois dimensions dès les premières images en fusionnant leurs informations. En utilisant une simple caméra monoculaire, il est nécessaire d'effectuer un déplacement pour définir la scène en utilisant plusieurs images réalisées à différents points de vue. La reconstruction de la scène nécessite la connaissance du déplacement du robot entre chaque image. Ce déplacement peut être calculé à partir des images elle-mêmes ou en utilisant un capteur supplémentaire tel qu'une centrale inertielle ou des odomètres.
Comme pour le télémètre laser, l'utilisation de caméras par des algorithmes de SLAM permet d'envisager le déploiement à grande échelle de ce type de système. En effet, les caméras se sont fortement démocratisées devenant ainsi des composants grand public à faible coût. Pour preuve, aujourd'hui, tout le monde possède un téléphone portable incluant une caméra. L'utilisation du SLAM pourrait permettre, par exemple, la localisation d'une personne dans une ville ou encore dans un musée simplement à l'aide de son téléphone. Aucun équipement supplémentaire ne sera nécessaire si le système de SLAM était intégré aux téléphones.

Les techniques couramment employées de SLAM restent généralement coûteuses en termes de puissance de calcul. En effet, les robots mobiles sont généralement équipés d'un ordinateur portable ayant d'importantes capacités de calculs. L'embarquabilité des systèmes est un problème majeur. La tendance actuelle vers la miniaturisation impose de restreindre les ressources embarquées. En effet, la majorité des algorithmes de localisation et de cartographie nécessitent une puissance de calcul considérable, inadaptée à une utilisation nomade. Des recherches sont effectuées pour réduire les ressources nécessaires à ce type d'algorithme, ce qui augmenterait le champ d'utilisation des systèmes de SLAM. De nombreuses optimisations sont nécessaires pour obtenir des résultats satisfaisants. En effet, le SLAM ne se limite pas à un algorithme mais il inclut aussi un ou plusieurs capteurs ainsi que les calculateurs utilisés pour divers traitements. Les optimisations peuvent, en particulier, se concentrer sur l'optimisation de la qualité des résultats, la diminution des ressources de calcul nécessaires ou encore sur l'adéquation entre l'algorithme et l'architecture du système.

Aujourd'hui, l'électronique embarquée dispose de nombreux calculateurs possédant des spécificités propres qui leurs permettent une grande efficacité. Or, ces spécificités ne sont que rarement utilisées, en particulier dans le contexte du SLAM. La définition d'un couple algorithme/architecture est essentiel pour envisager un système performant. L'implantation d'un algorithme sur une architecture existante requiert des modifications algorithmiques afin d'optimiser les performances du système en fonction

des capacités spécifiques de l'architecture. Ce type d'optimisation est souvent réalisée lors de l'utilisation de processeurs multi-coeurs, de GPU (Graphics Processing Unit) ou de processeurs vectoriels. Il est nécessaire de revoir l'implantation de chaque bloc composant l'algorithme pour utiliser au mieux les capacités spécifiques des processeurs embarqués actuels tels que le multi-cœur, la présence d'unité de calcul vectoriel ou encore de processeur hétérogène.

L'utilisation d'une architecture existante réduit les possibilités d'optimisations des systèmes. En effet, ces architectures sont conçues pour permettre une implantation efficace de la plupart des algorithmes. Le second type d'optimisation consiste à définir un algorithme mais aussi une architecture adaptée sur laquelle il sera implanté. L'utilisation d'un FPGA (Field-Programmable Gate Array) permet, dans ce cas, de définir une architecture matérielle spécifique pour maximiser l'efficacité du système embarqué. De plus, l'utilisation d'outils dédiés à la conception d'architectures matérielles permet aujourd'hui d'obtenir de très bonnes performances tout en conservant un temps de développement restreint.

La conception d'un système est couplé à son évaluation. En effet, après avoir défini un système, il est nécessaire d'en valider son bon fonctionnement. Classiquement, cette validation est réalisée en deux étapes. La première utilise un simulateur pour évaluer les performances du système alors que la seconde utilise des données expérimentales. La tendance actuelle est de concevoir un système de validation incluant le système matériel dans la boucle de traitement logicielle.

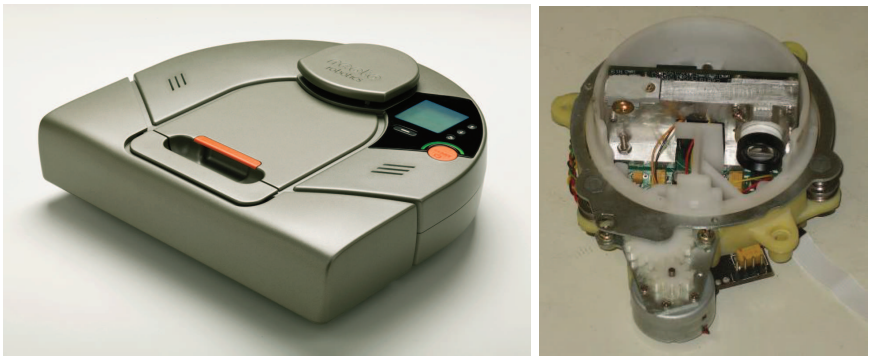


Figure 1: *Aspirateur autonome développé par Neato, et son laser low-cost*

Objectifs et Contributions

L'objectif principal de cette thèse consiste à définir des architectures pour des systèmes embarqués dédiés aux applications SLAM dans un contexte d'Adéquation Algorithme Architecture.

La première contribution consiste à définir un système de SLAM allant du capteur au calculateur en utilisant une architecture grand public. Ce système nous permettra d'explorer les capacités du système embarqué et de définir une implantation efficace d'un algorithme sur l'architecture choisie. Ce système sera ensuite évalué fonctionnellement et temporellement à l'aide de nos outils.
La seconde contribution définira un couple algorithme/architecture adapté à la reconstruction de scène. En effet, l'utilisation d'une architecture programmable met en avant de nombreuses possibilités d'implantations indisponibles sur des architectures classiques. Cette contribution définit un système matériel et logiciel adapté au SLAM. De plus, nous effectuerons des recommandations pour la définition d'architectures en fonction des algorithmes actuels.
Enfin, les études des algorithmes précédents mettront en avant des problèmes de qualités des résultats. Le principal était que les résultats issus de ces algorithmes ne sont pas garantis. Nous définirons donc un algorithme, basé sur la théorie ensembliste, garantissant la localisation et la cartographie de l'environnement. Cet algorithme sera évalué par rapport aux algorithmes traditionnels.

Organisation du mémoire

Le premier chapitre réalise une bibliographie des différents algorithmes et systèmes de SLAM existants. Les algorithmes sont tout d'abord étudiés en fonction de leurs caractéristiques, des ressources nécessaires à leurs fonctionnements ainsi que de leurs résultats. Ensuite, nous étudierons les systèmes embarqués dédiés au SLAM. De nombreux systèmes ont été définis pour résoudre la problématique du SLAM allant du PC classique aux architectures programmables.

Le second chapitre de ce mémoire est consacré à la définition de notre plateforme expérimentale ainsi qu'aux systèmes d'évaluation. La plateforme expérimentale est indispensable pour définir et évaluer les différents algorithmes. Elle fournit les données nécessaires à leur mise au point. Ensuite, des méthodes sont définies pour évaluer les algorithmes choisis mais aussi les systèmes conçus en utilisant une méthodologie HIL (Hardware In the Loop) qui permet de valider des modules matériels en les incluant dans la boucle de traitement utilisant des données simulées ou expérimentales.

L'étude bibliographique montrera que l'implantation optimisée d'un algorithmique de SLAM en adéquation avec une architecture multiprocesseur embarquée représente des recherches qui ont rarement eu lieu. Or, ce type d'optimisation est indispensable pour obtenir un système qui répond aux contraintes de l'embarqué, notamment celle du temps-réel et de la consommation d'énergie. Le troisième chapitre sera donc dédié

à la conception d'un système de SLAM basé sur un module de traitement grand public. Après avoir choisi et étudié l'algorithme, plusieurs optimisations sont proposées pour obtenir une implantation efficace sur une architecture embarquée multiprocesseur. Deux plateformes seront évaluées : une architecture hétérogène et une architecture homogène. Les différentes optimisations mettront en avant les capacités avancées de chaque architecture.

L'optimisation de l'implantation précédente a mis en avant la nécessité de l'adéquation entre l'algorithme et l'architecture cible. Comme nous le verrons, l'utilisation des spécificités de l'architecture a permis un gain important de performances. Cependant, l'utilisation d'une architecture spécifique à un algorithme doit permettre un gain supérieur en terme de performances. Nous choisissons donc de définir, dans le quatrième chapitre, un couple architecture/algorithme adapté au SLAM. L'architecture définie sera validée à l'aide d'une méthodologie HIL puis caractérisée temporellement. Enfin, ces performances seront comparées aux architectures grand publics évaluées et décrites dans le chapitre précédent.

Le dernier chapitre sera consacré à la définition d'un algorithme de SLAM garantissant les résultats de localisation. En effet, l'étude des différents algorithmes de SLAM probabilistes montrera leurs faiblesses en termes de consistance. Nous définirons donc un algorithme basé sur la théorie ensembliste qui garantira les résultats de localisation et de reconstruction de l'environnement. Cet algorithme sera évalué puis comparé aux algorithmes précédemment étudiés.

Finalement, nous conclurons sur les contributions proposées au cours de ce travail de thèse, ainsi que sur les différentes voies de recherches issues de ce celui-ci.

Chapitre 1

Algorithmes SLAM et architectures des systèmes dédiés : Etat de l'art

Sommaire

1.1 Localisation et cartographie simultanées

Le SLAM (Simultaneous Localization And Mapping) est un processus consistant à répondre à deux questions " Où suis-je ? " et " Quelle est la structure de mon environnement ? ". Ces problèmes ne sont pas uniquement posés en robotique mobile mais sont aussi résolus quotidiennement par des êtres humains.
Pour se repérer, les hommes disposent de cinq sens pouvant être chacun exploité séparément puis combinés, par notre cerveau, pour affiner notre hypothèse de localisation et notre connaissance de l'environnement qui nous entoure. Ce phénomène peut être réalisé inconsciemment ou être l'objet d'une réflexion profonde permettant d'exploiter au mieux les différentes informations que nous captons.

La localisation et la cartographie simultanées sont indispensables dans le domaine de la robotique, en particulier pour la robotique mobile. Le SLAM est une tâche préliminaire à de nombreux autres domaines tels que la navigation autonome ou la planification de trajectoire. En effet, obtenir une modélisation réaliste d'un environnement permet au robot d'augmenter son autonomie en prenant des décisions en fonction du contexte général.

1.2 Le SLAM probabiliste

Le SLAM consiste à définir simultanément la localisation d'un mobile et la cartographie d'un environnement. Bien évidemment, les deux problèmes peuvent être résolus séparément. En effet, la localisation dans un environnement connu ainsi que la cartographie à partir de plusieurs points de vue sont des problèmes communément résolus. Cependant, si le mobile réalise une trajectoire inconnue dans un environnement inconnu, le problème de cartographie et de localisation simultanées semble beaucoup plus complexe. Aucune information n'est connue a priori par le mobile qui doit créer un environnement à partir des différentes données provenant des capteurs dont il dispose et se localiser dans l'environnement qu'il vient de reconstruire.

Depuis ces vingts dernières années, de nombreuses recherches ont été menées pour résoudre ce problème. L'une des première approche du SLAM consiste à utiliser la théorie probabiliste pour modéliser l'environnement ainsi que la localisation du mobile.

1.2.1 Première approche algorithmique

Le filtre de Kalman, proposé en 1960 [9], estime les états d'un système dynamique à partir de mesures imprécises ou incomplètes. C'est l'estimateur optimal pour les systèmes linéaires et les distributions gaussiennes. Il a été étendu dans le cas non linéaire (Extended Kalman Filter - EKF) par Maybeck [10] en 1979. Une particularité du filtre de Kalman est de mettre à jour l'intégralité du système en observant une

seule de ses composantes. En effet, les composantes du système sont reliées entre elles par une matrice de covariance qui modélise ces relations entre les différents paramètres.

L'utilisation du filtre de Kalman pour la cartographie a été proposée par Cheeseman *et al.* [11]. Ils proposent de cartographier l'environnement en utilisant la localisation moyenne de points de repères (Appelés amers) ainsi que la covariance les reliant. La carte est construite de manière incrémentale en insérant au fur et à mesure les amers découverts tout au long du trajet du robot. Ce travail a été étendu par Moutarlier et Chatila [12] en incluant la notion de propagation de l'incertitude de localisation du robot dans le temps. Ils utilisent un modèle dynamique qui prédit la localisation du mobile et un modèle d'observations qui utilise les mesures des capteurs extéroceptifs. Cependant, les améliorations apportées sur la qualité des résultats de l'estimation sont associées à la dégradation du temps d'exécution: plus la carte est grande, plus le temps d'estimation est important.

La forte dépendance du temps d'exécution à la taille de la carte a induit l'étude de l'importance de maintenir une matrice de corrélation complète. Une étude expérimentale a été menée par Castellanos *et al.* [13] pour comparer les résultats d'un EKF-SLAM comportant la corrélation totale, une corrélation partielle ou aucune corrélation. Sans corrélation, la carte de l'environnement devient très rapidement inconsistante, de plus la corrélation permet d'améliorer l'intégralité de la reconstruction lors de l'observation d'un simple amer. Ces résultats ont été validés théoriquement par Csorba [14] puis par Dissanayake *et al.* [15] qui ont démontré la nécessité de conserver l'intégralité de la matrice de covariance pour garantir la convergence du filtre.

1.2.2 Vers l'utilisation de capteurs bas coût

Les recherches précédemment citées ont défini des algorithmes de SLAM basés sur un filtre de kalman. Le principe du filtre est indépendant du type de capteur utilisé. Deux types de capteurs sont couramment utilisés :

- les capteurs proprioceptifs qui renseignent sur les déplacements propres du robot (odomètres, accéléromètres ou gyromètres).
- les capteurs extéroceptifs qui renseignent sur l'environnement entourant le robot (télémètres laser ou caméras).

Les premières implémentations d'algorithmes de SLAM ont été effectuées à l'aide de télémètres laser. Ce type de capteur, précis et robuste, génére peu d'erreurs et founit des mesures rapides et précises. Dès 1996, Betge-Brezetz *et al.* [16] proposent une validation expérimentale de l'EKF-SLAM sur un robot équipé d'un télémètre laser, une expérimentation similaire a été réalisée par Csorba [14].

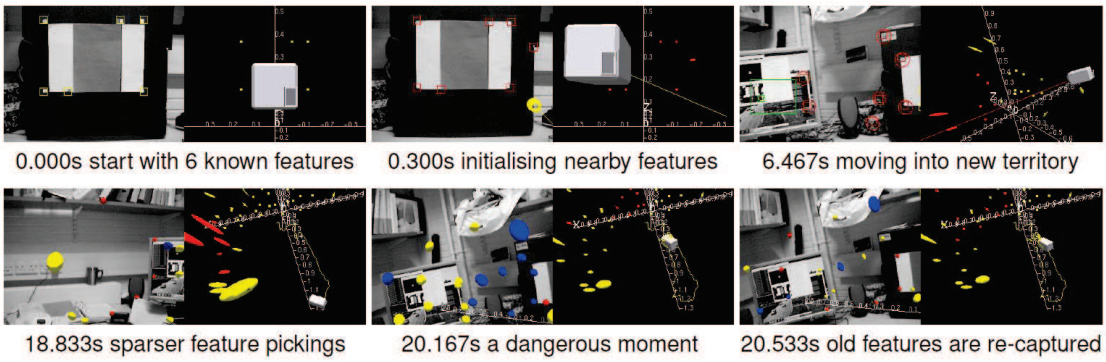


Figure 1.1: *Résultat du MonoSLAM de Davison [1]*

L'utilisation du télémètre laser a montré de bons résultats mais ce type de capteur reste généralement couteux et difficilement accessible pour des produits grand public.

Parallélement à l'utilisation de télémètres, des algorithmes ont été proposés pour utiliser une ou plusieurs caméras. Harris et Pike [17] proposent le premier algorithme de cartographie à l'aide d'une simple caméra. Le système réalise une cartographie séparée de chaque point de répère, la carte ne peut donc pas être consistante. Cependant, il introduit le problème de la paramétrisation des points et de leurs incertitudes qui sera un axe majeur de recherche. De plus, il développe le concept de recherche active qui consiste à rechercher des correspondances à l'intérieur de zones prédéfinies par la carte de l'environnement, qui sera un second axe de recherche important.

En 1997, Neira *et al.* [18] ont présenté le premier système de SLAM basé sur les travaux concernant les cartes entièrement corrélées de Cheeseman *et al.* [11] et Moutarlier et Chatila [12] en utilisant un capteur de vision monoculaire et des odomètres. L'algorithme est basé sur la détection et l'appariement d'amers verticaux, modélisés en 2 dimensions. Son étude met en avant l'importance de l'étape de mise en correspondance. L'utilisation d'amers plus robustes sera réalisées par Se *et al.* [19] en utilisant des amers ponctuels de type SIFT (Scale-Invariant Feature Transform [20]).

L'utilisation de points en trois dimensions sera proposée par Davison et Murray [21], ils ont proposé la première implémentation utilisant une caméra stéréoscopique et des odomètres. Le système représente la carte de manière éparse pour limiter l'augmentation du temps de calcul. Ce travail sera étendu à l'utilisation exclusive d'une caméra stéréoscopique (sans odométre) par Jung et Lacroix [22].

Le domaine du SLAM visuel a connu une grande avancée en 2003 par Davison [1] qui a développé un algorithme de SLAM, appelé MonoSLAM, basé sur une simple caméra ainsi qu'une implémentation en temps-réel de son système sur un ordinateur

grand public. Son système utilise une simple webcam et aucune donnée odométrique. Ce travail est considéré comme le premièr système de SLAM en temps-réel utilisant un capteur bas coût. La carte, ainsi que la position de la caméra, sont représentées en trois dimensions et estimées par un filtre de Kalman incluant une corrélation complète de la carte. Une des limitations du système est l'utilisation d'une étape d'initialisation qui permet au système de connaître une métrique, indispensable à la reconstruction en trois dimensions. Davison propose une méthode d'initialisation des amers basée sur un filtre particulaire. L'incertitude de profondeur, dûe à l'utilisation d'une caméra monoculaire, est réduite au fur et à mesure du déplacement. Une fois l'incertitude assez réduite, l'amer est ajouté à la carte de l'environnement. La figure 1.1 représente les différentes étapes de l'algorithme proposé comme l'initialisation de nouveaux amers, le déplacement de la caméra puis la relocation des amers.
Pour conserver un aspect temps-réel, Davison propose une gestion de la carte de l'environnement permettant de limiter le nombre d'amers présents.

Aujourd'hui, l'EKF-SLAM est toujours le sujet de multiples études, optimisations et implantations. Ces études portent principalement sur l'optimisation des points négatifs de l'algorithme tels que sa forte dépendance à la taille de la carte.

1.2.3 Les limitations de l'EKF-SLAM

Depuis 1988, de nombreuses recherches ont été basées sur l'utilisation principale de l'EKF-SLAM. De nombreuses optimisations ont été proposées pour améliorer la qualité de la localisation ou de la cartographie mais aussi pour permettre l'utilisation de capteurs bas coût. Cependant, cet algorithme souffre de nombreux problèmes, en particulier, il a été démontré, par Csorba [14], Dissanayake *et al.* [15], que pour obtenir une carte consistante, chacun des amers devait être considéré dans le vecteur d'état mais aussi dans la matrice de covariance.

Complexité algorithmique

L'obligation de maintenir une carte complète entièrement corrélée a un coût de stockage en mémoire avec une complexité en $O(N^2)$, avec N le nombre d'amers dans la carte. De plus, la mise à jour d'un seul amer dans la carte mettra à jour l'ensemble du système. De ce fait, la mise à jour aura lieu avec une complexité en $O(N^2)$. On s'aperçoit que, en l'état, l'EKF-SLAM ne sera pas adapté à la cartographie d'une large zone car son temps de traitement a une très forte dépendance au nombre d'amers.

Gestion de la carte de l'environnement

Plusieurs solutions ont été proposées pour réduire la dépendance au nombre d'amers. La première solution, proposée par Dissanayake *et al.* [15], consiste à supprimer des

amers pour conserver une carte de taille limitée. Ils ont démontré qu'il était possible de supprimer des amers de la carte sans affecter la consistante globale. Cependant, il ne sera plus possible de cartographier un grand environnement en réalisant de grandes fermetures de boucles. Cette limitation a aussi mise en avant le problème de sélection des amers à conserver, aussi étudié par Davison [1].

Décorrélation

Une seconde solution a été d'exploiter le caractère éparse de la matrice de covariance, cette matrice est effectivement remplie majoritairement de zéros car plus les amers sont éloignés spatialement, moins ils seront corrélés. Ce constat permet d'envisager la mise à jour partielle du système en sélectionnant uniquement les amers ayant un impact important durant la mise à jour du système.
Thrun *et al.* [23, 24] ont proposé un filtre appelé Sparse Extended Information Filter (SEIF) qui ignore les corrélations faibles entre les amers. Une corrélation est utilisée uniquement entre les amers ayant été observés simultanément. La consistance de cette approche a ensuite été améliorée par Eustice *et al.* [25] et Walter *et al.* [26].

Représentation relative

Classiquement, la carte de l'environnement est construite de manière incrémentale à partir de la position initiale du robot. En 1997, Csorba [14] propose de centrer la carte de l'environnement sur la position actuelle du robot, le filtre estime alors la position relative des amers par rapport au mobile. Le vecteur d'état contient alors N^2 informations car il modélise les informations entre chaque amer. Cependant, la matrice de covariance associée peut être diagonale car les relations sont indépendantes les unes des autres. De ce fait, l'étape d'estimation a une complexité en $O(1)$. Cependant, cette représentation ne permet pas un accès simple à la position du robot dans le repère global, c'est pourtant ce type d'information qui parait important pour un algorithme de localisation.
Cette idée sera reprise et améliorée par Newman [27] en combinant une représentation relative et un système de contrainte afin d'obtenir une estimation consistante de la position des amers tout en améliorant la gestion des incertitudes. L'implantation réalisée a montré des résultats similaires à un EKF-SLAM classique.

Etude de la consistance

Le filtre de Kalman est optimal dans le cas d'un système linéaire, affecté par des bruits gaussiens. Dans le cas de l'EKF-SLAM, les modèles de déplacements et d'observations ont été linéarisés, ceci peut induire des problèmes de consistance. L'étude de la consistance vise à vérifer que la position réelle du robot est incluse dans la zone définie par le vecteur d'état, incluant l'incertitude induite par la matrice de

covariance.
La consistance de l'EKF-SLAM a été étudiée par Julier et Uhlmann [28], qui avaient proposés d'utiliser un UKF (Unscented Kalman Filter) dès 1997 [29]. Ils ont montré plusieurs cas où le filtre ne peut pas être consistant. En particulier, ils utilisent le fait que, à tout instant, l'incertitude de localisation ne peut être inférieure à l'incertitude de localisation initiale. Julier et Uhlmann [28] ont montré plusieurs cas où l'incertitude d'orientation était corrigée de manière abusive, rendant le filtre inconsistant. Ce problème a été confirmé par Castellanos *et al.* [30] qui ont conseillé d'utiliser la représentation relative de la carte pour améliorer la consistance globale.
Bailey *et al.* [31] ont étudié plus précisément les inconsistances du filtre mettant en avant la problèmatique de l'incertitude d'orientation. Lorsque cette incertitude devient trop importante, le filtre réalise des corrections optimistes rendant le résultat inconsistant. La solution proposée est l'utilisation d'un capteur supplémentaire mesurant le cap de manière absolue dans le repère global.
Ces problèmes de consistance ont été démontrés dans un contexte théorique par Huang et Dissanayake [32], Huang *et al.* [33], où il est montré que, "le sous-espace observable du système linéarisé est de dimension supérieure à celle du système réel, non linéaire, menant à des réductions de covariance dans des directions de l'état où aucune information n'est disponible, est une première cause de l'inconsistance". Ces problèmes de consistance sont atténuables de plusieurs manières, par exemple en utilisant une paramétrisation apprioriée pour les amers.

Paramétrisation des amers

Les problèmes de consistance de l'EKF-SLAM ont amené les chercheurs à améliorer la paramétrisation des amers. En effet, l'utilisation d'une simple paramétrisation en trois dimensions (x, y, z) produit des équations d'observations fortement non linéaires. Montiel *et al.* [2] ont proposé d'utiliser une paramétrisation par inverse de profondeur permettant une initialisation directe des amers. En effet, la paramétrisation par inverse de profondeur est plus adaptée à représenter l'incertitude de la profondeur. La figure 1.2 représente l'avantage de cette paramétrisation par rapport a une paramétrisation xy. On remarque que le nuage de points qui représente l'incertitude de localisation de l'amer peut-être plus facilement approximée par une gaussienne dans le cas de l'inverse de profondeur. Enfin, les équations d'observations issues de ce modèle ont un plus faible degré de non-linéarité, ce qui améliore la consistance du filtre.
Plusieurs paramétrisations ont été proposées et étudiées par Sola [34], Solà *et al.* [35]. L'impact positif sur la consistance du filtre a été étudié en simulation et expérimentalement.

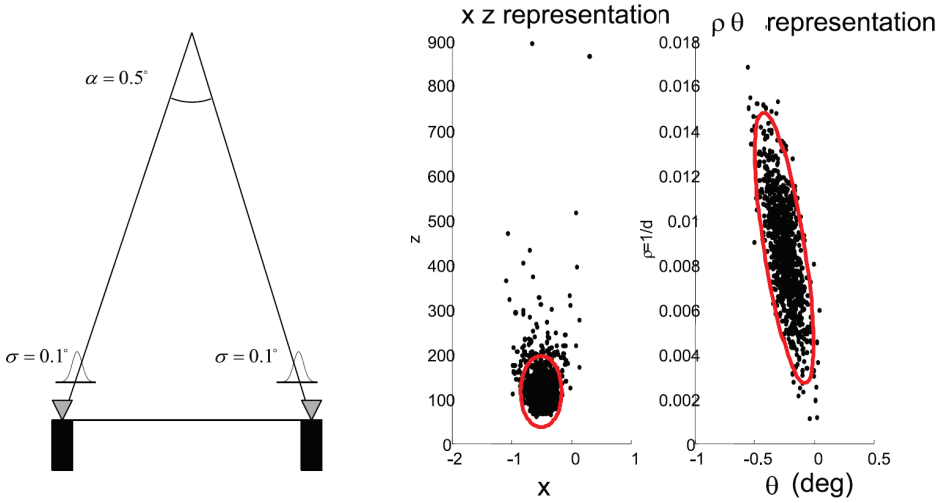


FIGURE 1.2: *Comparaison entre une paramétrisation classique et par inverse de profondeur par Montiel* et al. *[2]*

Utilisation de cartes locales

Deux problèmes majeurs affectent l'EKF-SLAM, le premier est le problème de consistance des résultats et le second la forte complexité algorithmique. Dans ce cadre, des recherches ont été menées pour réaliser une cartographie de type "Diviser pour régner" qui consiste à diviser la cartographie globale en plusieurs cartographies de petits espaces afin de réduire le nombre d'amers présents dans le filtre.
En 1999, Chong et Kleeman [36] proposent le premier algorithme de cartographie locale. L'espace est sous-divisé lorsque l'incertitude de localisation du robot dépasse un seuil. Les différentes cartes sont ensuite mises en commun pour former une cartographie globale. Pour détecter les fermetures de boucles, l'algorithme utilise la carte globale pour vérifier des passages dans des zones déja cartographiées.
Leonard et Feder [37] proposent, au contraire, de réaliser des cartes locales de tailles spatiales fixes. Ils discrétisent la carte globale sous forme de grille, chaque case étant cartographiée séparément mais reliée dans la grille globale. Plusieurs algorithmes ont été étudiés pour réaliser la transition entre les différentes cartes. Ils seront ensuite améliorés par Leonard et Newman [38].
De nombreuses autres méthodes de cartographie locale ont été développées par Bailey [39], Bosse *et al.* [40] ou Estrada *et al.* [41] afin d'améliorer la consistance globale ou l'efficacité de la fermeture de boucle.

1.2.3.1 Gestion de l'appariement

L'EKF-SLAM utilise des amers pour cartographier l'environnement puis se localiser. Cependant, le filtre ne tolère pas les erreurs d'appariement qui provoquent des problèmes de consistance. Dès 2003, Davison [1] propose une méthode d'appariement active

qui projette l'incertitude des amers sur l'image de caméra pour calculer une zone de recherche dans laquelle est sélectionnée l'observation. Ce type de recherche active a été associé à d'autres algorithmes comme le FastSLAM par Eade [42].
Chli et Davison [43] ont proposé une amélioration du concept de recherche active en tirant un profit immédiat de l'appariement d'un amer. En effet, le filtre réalise l'appariement d'un seul amer puis met à jour les zones de recherches des autres amers. Cette mise à jour réduit le temps de traitement de la mise en correspondance. De plus, un système d'arbre est ajouté afin d'éviter les problèmes de mauvaises mises en correspondance.
Récemment, l'appariement a été amélioré par Civera *et al.* [44] en utilisant RANSAC pour éviter les erreurs d'appariement.

1.3 Approche particulaire : Le FastSLAM

L'EKF-SLAM est encore aujourd'hui communément utilisé, malgré ses défauts de complexité ou de consistance, il possède l'avantage d'avoir été très étudié. Des recherches ont été effectuées pour concevoir une alternative à l'EKF-SLAM afin de corriger son principal défaut: sa forte complexité algorithmique.
Montemerlo *et al.* [3] ont défini un algorithme appelé FastSLAM utilisant un filtre particulaire à la place du filtre de Kalman. L'intérêt majeur est la décorrélation de l'estimation de la localisation du robot de celle des amers. En effet, la localisation du robot est échantillonnée sous forme de particules, chacune représente une hypothèse de localisation du mobile. La probabilité de chaque hypothèse est ensuite mise à jour en fonction des observations, enfin, une étape de rééchantillonage concentre les hypothèses sur la localisation la plus probable. La figure 1.3 représente les résultats du FastSLAM pour le jeu de données du Victoria Park.

Le FastSLAM réduit le temps de calcul par rapport à l'EKF-SLAM, cependant, plusieurs problèmes apparaissent. Le premier consiste à définir théoriquement le nombre de particules nécessaires au bon fonctionnement de l'algorithme. Ce nombre semble corrélé à la longueur du trajet réalisé entre deux fermetures de boucle (Eade [42]). Le second problème est, comme pour l'EKF-SLAM, un problème de consistance (étudié par Bailey *et al.* [45]). En effet, l'algorithme réalise une étape de rééchantillonage qui supprime des particules et donc des hypothèses de localisation des amers. La consistance des cartes restantes n'est donc plus garantie.

1.4 Le SLAM par optimisation de graphes

L'EKF-SLAM et le FastSLAM sont basés sur la théorie probabiliste. Ils souffrent tous les deux d'un problème de consistance. C'est pourquoi, un nouveau type

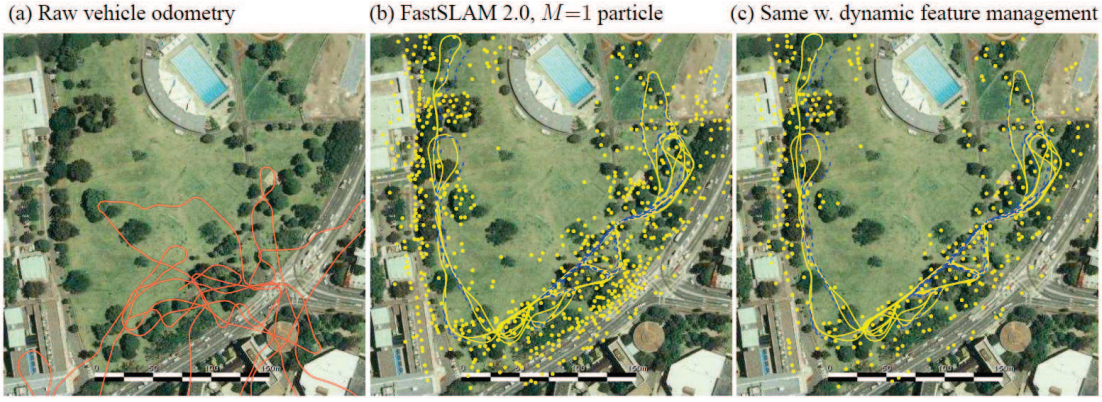


FIGURE 1.3: *Résultats du FastSLAM de Montemerlo* et al. *[3] pour le jeu de données de Victoria Park*

d'algorithme est apparu récemment, basé sur l'optimisation de graphes (Thrun et Montemerlo [46], Folkesson et Christensen [47]). Il consiste dans le cas du SLAM visuel à minimiser les erreurs de reprojection afin de recalculer la position de la caméra mais aussi celles des amers.
Plusieurs problèmes affectent ce type d'approche: le temps de traitement important, le manque de flexibilité de l'étape de mise en correspondance et la fermeture de boucle. Cependant, l'implantation réalisée par Klein et Murray [48] démontre qu'il est envisageable d'utiliser ce type d'algorithme sur des plateformes ayant des ressources limitées (ici un iphone) pour cartographier de petites scènes. L'une des limitations de ces approches est de conserver assez de pessimisme pour réaliser de grandes fermetures de boucle [49].

Les développements actuels utilisent principalement le GraphSLAM pour réaliser des modèles denses à partir d'une simple caméra. En particulier, Newcombe et Davison [50] ont proposé un algorithme réalisant un modèle dense à partir d'une simple caméra. Cependant, son implantation demande beaucoup de ressources de calculs.

L'utilisation de ce type d'algorithme est conditionnée à l'utilisation d'un calculateur important, les implantations temps réel nécessitent l'utilisation d'une carte GPU très performante. Strasdat *et al.* [51] ont montré que l'utilisation de filtre (EKF-SLAM ou FastSLAM) plutôt que de solutions basées sur l'optimisation de graphes était efficace uniquement lors de l'utilisation couplée à un calculateur ayant de faible performance.

1.5 Le SLAM ensembliste

L'étude des différents algorithmes de SLAM a mis en avant des problèmes de consistance en particulier pour les algorithmes probabilistes. Ce type de problèmes peut être résolu en utilisant une approche basée sur l'analyse par intervalles. En effet, l'analyse par intervalles n'est pas affectée par les erreurs de linéarisation ou d'échantillonage. De plus, elle produit des résultats garantis et consistants. Ces avantages ont été mis

en avant dans le cas de localisation de véhicule en environnement extérieur [52, 53] et intérieur [54].

Le premier algorithme de SLAM basé sur la théorie ensembliste a été proposé par Di Marco *et al.* [55] en 2001. La localisation du mobile est représentée par une boite à trois dimensions (x, y et θ) dont le volume évolu en fonction des données proprioceptives (odomètres) et extéroceptives (capteurs types télémètres laser). Les amers sont modélisés par des parallélélogrammes. L'algorithme est divisé en quatre sections:

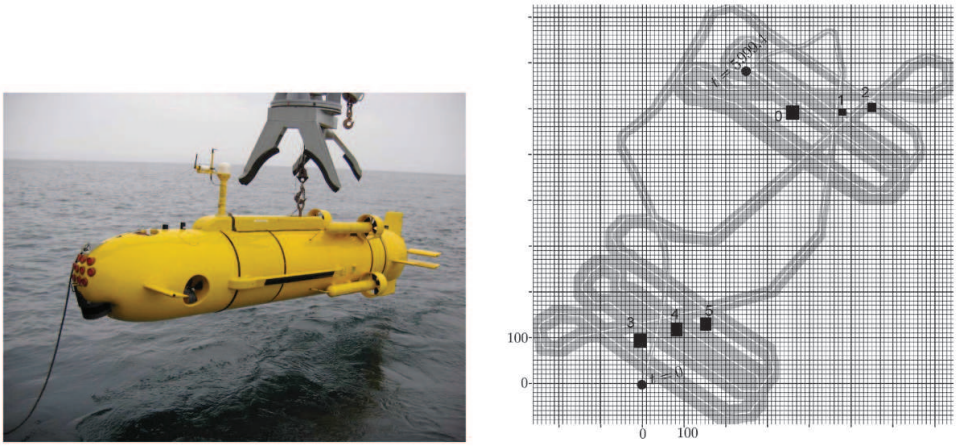
- La prédiction de la localisation du mobile.
- La correction de la localisation du mobile en utilisant les observations réalisées par le capteur extéropceptif.
- L'initialisation de nouveaux amers.
- Une étape de correction dédiée aux amers.

Cet algorithme a ensuite été étendu en incluant une étape de mise en correspondance automatique [56]. Il a été évalué expérimentalement, les résultats obtenus sont consistants [55, 56].

En 2003, Drocourt *et al.* [57] ont proposé un algorithme de SLAM basé sur la théorie ensembliste et l'utilisation de l'algorithme SIVIA (Set Inversion Via Interval Analysis) [58]. Le système utilise des odomètres et une caméra stéréoscopique ommnidirectionnelle qui produit une image en trois dimensions et à 360 degrés de l'environnement entourant le robot. Contrairement à Di Marco *et al.* [55], le robot est représenté par un pavage à 3 dimensions qui implique l'utilisation d'algorithmes comme ImageSP et SIVIA pour gérer les pavages [58]. L'étape de correction de la position du mobile effectue l'intersection entre le pavage prédit et le pavage observé à l'aide de la caméra. La correction de la localisation des amers est réalisée de manière similaire, par intersection de pavages.

En 2005, Porta [59] propose de résoudre la problématique du SLAM comme un ensemble de contraintes entre la localisation du robot, celle des amers et les observations. Cette approche se rapproche du principe du GraphSLAM mais Porta résout le système de contraintes à l'aide de l'algorithme CUIK (Complete Universal Inverse Kinematics) basé sur l'analyse par intervalles. La localisation du robot ainsi que celle des amers est modélisée par une boite à trois dimensions pour faciliter l'utilisation de CUIK. L'algorithme a été évalué expérimentalement en utilisant un sonar et a démontré des résultats consistants.

En 2006, Jaulin [60, 4] a présenté un algorithme basé sur la propagation des contraintes, pour localiser un robot sous-marin (Figure 1.4a). Expérimentalement, de nombreux capteurs ont été utilisés : un GPS, un sonar, un "loch doppler", un gyrocompas et un baromètre. Le robot est modélisé par une boite à 6 dimensions (x,y,z,φ,θ,ψ) (contrairement à l'utilisation d'une boite en trois dimensions).
La position initiale du robot est calculée à l'aide du GPS, puis les capteurs proprio-



(a) (b)

Figure 1.4: *a) Robot sous-marin développé par Jaulin [4], b) Résultats de localisation obtenus*

ceptifs permettent de prédire la position du robot. Enfin cette position est corrigée en utilisant le sonar. Cependant, les données issues du sonar doivent être pré-traitées par un opérateur, la détection ainsi que la mise en correspondance des amers sont réalisées manuellement. Les modèles de déplacement et d'observations définissent un ensemble de contraintes qui est résolu en utilisant la propagation de contraintes associée à l'analyse par intervalles. Les résultats expérimentaux (Figure. 1.4b) ont montré la consistance de cette approche. De plus, la comparaison avec une méthode probabiliste (EKF-SLAM) a mis en avant les avantages de l'utilisation de l'analyse par intervalles. Plusieurs points ont été mis en avant tels que la garantie des résultats, la consistance, l'absence de linéarisation ou encore la validation simplifiée du système.

En 2008, Joly et Rives [61] proposent un algorithme similaire pour la localisation d'un robot mobile simulé, embarquant des odomètres et une caméra omnidirectionnelle. Ils ont ajouté le concept de boites orientées, contrairement à l'utilisation de boites simples telles que celles réalisées par Porta [59] et Jaulin [4]. Ce concept a été utilisé pour la représentation de la localisation des amers et de la localisation du robot. Il permet de réduire le pessimisme de localisation induit par l'utilisation de l'analyse par intervalles. De plus, Joly et Rives [61] ont montré la supériorité de leur approche par rapport au GraphSLAM dans le cas de données biaisées. Dans le cas de bruits gaussiens ou uniformes, les résultats sont similaires mais plus pessimistes.

En 2012, Bars *et al.* [62] améliorent l'algorithme proposé par Jaulin [4] en ajoutant une détection automatisée des amers, basée sur l'observation de données fugaces (observations réalisées à un instant très précis et inconnu). De plus, ils introduisent la gestion des tubes d'intervalles adaptée à la robotique. Un tube peut être considéré comme un intervalle de trajectoire. Enfin, Sliwka *et al.* [63] ont proposé une méthode pour gérer la présence de données aberrantes. Ils ont validé leur approche dans le cadre de la localisation de robot sous-marin.

1.6 Le SLAM embarqué bas-coût

De nombreux algorithmes ont été étudiés précédemment, cependant ils sont principalement basés sur l'utilisation de capteurs coûteux et nécessitent de grandes capacités de traitement. L'utilisation de capteurs bas-coût nécessite des modifications algorithmiques Hoppenot *et al.* [64]. Plusieurs algorithmes de SLAM ont été conçus pour utiliser des capteurs bas coûts comme une simple caméra [65]. Cependant, l'implantation de ces algorithmes sur des système embarqués a très peu été traitée. L'utilisation du SLAM sur des systèmes embarqués étant encore très récente.

Abrate *et al.* [66] ont implanté un EKF-SLAM sur un robot Khepera, embarquant de simples capteurs infrarouges. Ils ont choisi d'implanter leur algorithme sur un microprocesseur Motorala 68331 cadencé à seulement 25 Mhz. La faible puissance de calcul disponible a imposé une optimisation importante de l'algorithme. Pour améliorer les performances du systèmes, ils ont choisi d'utiliser des primitives linéaires afin de compenser la mauvaise qualité des données issues de leurs capteurs. De plus, en utilisant des lignes, ils ont réduit le nombre de primitives utilisé par le filtre de Kalman et donc ils ont diminué le temps global de calcul. Les résultats expérimentaux ont montré l'importance de la caractérisation du capteur, du choix des primitives et de la mise en correspondance des amers.

Comme Abrate et al., Yap et Shelton [67] utilisent des capteurs bas coût, des télémètres ultrasons embarqués sur un robot P3-DX. Afin de compenser la mauvaise qualité des données issues des capteurs, ils choisissent de réaliser une hypothèse d'orthogonalité des murs composant la carte. L'hypothèse de base est que la carte de l'environnement est composée de murs orthogonaux. Cette hypothèse permet de simplifier la mise en correspondance des données capteurs ainsi que la création de nouveaux murs.

En plus de l'utilisation de capteurs bas coût, il faut utiliser des calculateurs embarquables alors que de nombreuses implantations requièrent des performances importantes. Gifford *et al.* [5] ont présenté une approche bas coût au SLAM multirobot. Chaque robot est équipé d'une carte Gumstix (600 Mhz), de six capteurs infrarouges, d'un gyromètre trois axes et de deux odomètres. La figure 1.5 représente le robot d'exploration conçu par Gifford *et al.*. L'algorithme utilisé est un DP-SLAM, exécuté sur la carte embarquée. Son exécution est incompatible avec le temps-réel, elle prend plus de 10 secondes avec 25 particules. Gifford et al. ont mis en évidence la difficulté de correctement paramétriser un algorithme de SLAM pour qu'il soit efficacement exécuté sur un système embarqué disposant de peu de ressources et ayant des contraintes d'éxécution en temps réel. De plus, les résultats expérimentaux sont très prometteurs.

Magnenat *et al.* [6] ont présenté un algorithme de SLAM basé sur l'utilisation conjointe

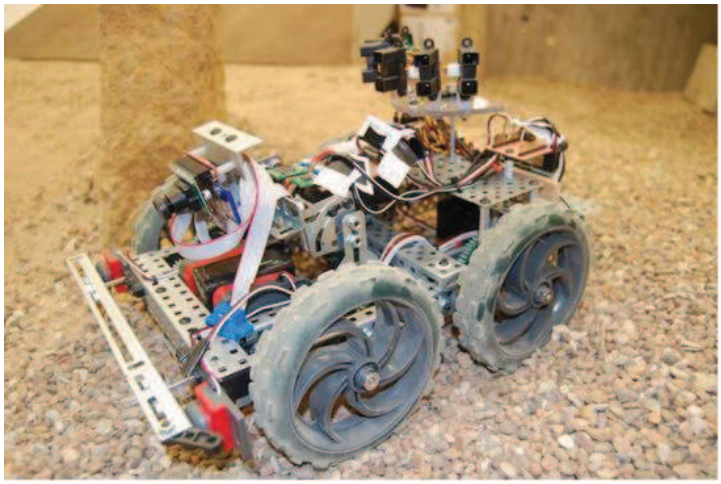
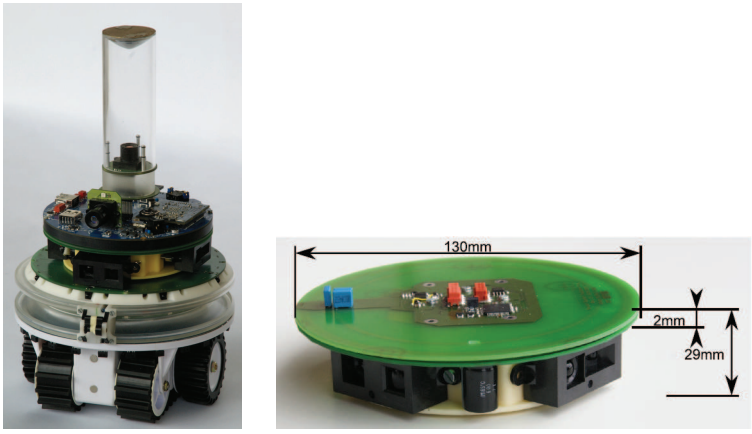


Figure 1.5: *Robot d'exploration conçu par Gifford* et al. *[5]*



(a) (b)

Figure 1.6: *a) Robot conçu par Magnenat* et al. *[6], b) Capteur développé*

d'un capteur bas coût, d'un algorithme de SLAM, d'un calculateur et d'une méthode d'optimisation. Ils ont développé un petit robot (Figure. 1.6a) capable d'embarquer l'ensemble de l'électronique nécessaire à l'exécution d'un algorithme de SLAM. Le robot embarque un calculateur ARM, cadencé à 533 Mhz, exécutant un algorithme Fast-SLAM 2.0 et un second calculateur dédié au contrôle du robot. L'équipe a développé un capteur spécifique (Figure. 1.6b), bas coût et surtout adapté à la taille du robot. Ce capteur est constitué d'un télémètre infrarouge rotatif. Ce capteur permet d'obtenir les informations nécessaires à l'algorithme de SLAM. Enfin, Magnenat *et al.* [6] utilisent une stratégie d'évolution pour paramétrer au mieux l'algorithme utilisé. Une bonne paramétrisation permet de diminuer le temps de calcul nécessaire en limitant le nombre d'amers présents dans la carte.

Récemment, Meier *et al.* [7] ont développé un drône quadrirotor équipé d'un module de traitement basé sur un processeur dual-core core2duo cadencé à 1.86 ghz. Cette puissance de calcul leur permet de réaliser du SLAM embarqué sur leur drône. De

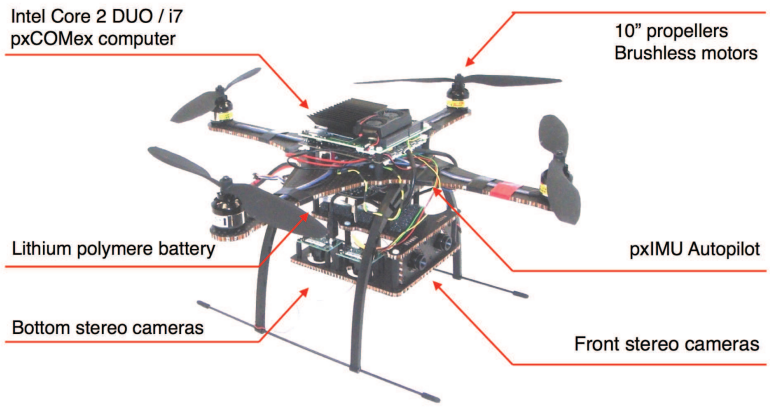


Figure 1.7: *Drône quadrirotor conçu par Pixhawk [7]*

plus, le drône est équipé d'un second contrôleur ARM7 à 60 Mhz dédié au contrôle de la plateforme. La figure 1.7 représente la plateforme développée.

Les études de Magnenat *et al.* [6] et Gifford *et al.* [5] soulignent l'importance de paramètrer efficacement les algorithmes pour atteindre des performances compatibles avec le temps réel sur des architectures embarquées. Cette paramétrisation est couplée à l'utilisation d'améliorations algorithmiques étudiées précédemment telle que la limitation de la taille du vecteur d'état pour l'EKF-SLAM. Certaines optimisations sont conçues pour améliorer l'embarquabilité des algorithmes, par exemple Schröter *et al.* [68] optimisent l'espace mémoire nécessaire à un algorithme de cartographie basé sur une grille et un algorithme FastSLAM.

Cependant, les optimisations en adéquation entre un algorithme et une architecture sont rares, les implantations proposées précédemment ne tirent pas profit des spécificités des processeurs sur lesquels elles sont embarquées. Aujourd'hui, les processeurs disposent de capacités spécifiques tels que les architectures multicœurs ou des architectures ayant des coprocesseurs vectoriels. Nous allons montrer que ce type de ressources permet une implantation plus efficace des algorithmes.

1.7 Vers des architectures dédiées

Nous avons vu que peu d'implantations sur des architectures embarquées avaient été réalisées. De plus, elles ne sont pas réalisées en adéquation avec l'architecture cible. La première implantation sur un système embarqué, réalisée en adéquation avec les capteurs a été proposée par la société Neato qui a développé un aspirateur autonome implémentant un système de navigation basé sur un algorithme de SLAM. Le robot est équipé d'un télémètre laser réalisé par la société pour minimiser le coût de production [69]. L'utilisation de ce capteur bas coût permet d'envisager de nouvelles fonctionnalités pour les robots de demain. En effet, le prix de revient d'un télémètre laser classique est de plus de 1000 euros contre seulement 30 euros pour celui développé par Neato. De

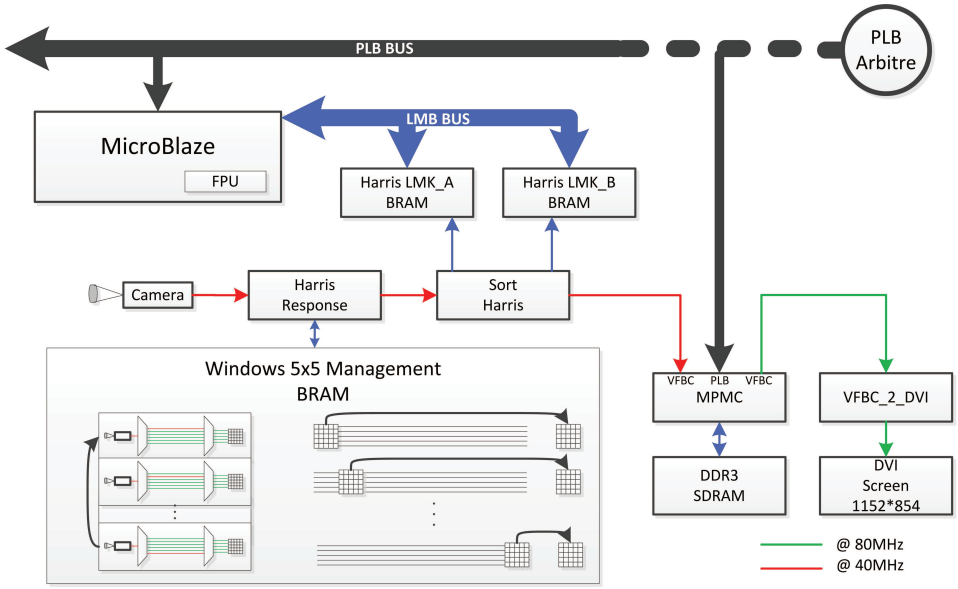


Figure 1.8: *Architecture dédiée au SLAM par Botero* et al. *[8]*

plus, sa précision et sa fréquence n'ont rien à envier au télémètre plus haut de gamme. Cependant Neato ne communique pas sur l'architecture de leur contrôleur principal.

L'utilisation de capteurs dédiés permet l'implantation sur une plateforme grand public. Cependant, l'utilisation de circuits dédiés pour le calcul du coeur de l'algorithme permettrait une implantation plus efficace. L'utilisation d'architectures massivement parallèles, réalisées sur des FPGA, a déjà été réalisée pour la détection de primitives. En particulier Idris *et al.* [70] proposent une sélection d'implantation d'algorithmes de détection de primitives sur des FPGA. L'idée principale de la majorité des implantations est que l'utilisation d'architectures reconfigurables conçues en adéquation avec les algorithmes permet le traitement parallèle de plusieurs pixels.

Botero *et al.* [8] proposent une architecture en codesign pour exécuter un EKF-SLAM. La détection des primitives (Harris) est effectuée à la fréquence pixelique à l'aide d'une architecture matérielle alors que le coeur de l'algorithme est implémenté sur un processeur softcore (Microblaze). La figure 1.8 représente l'architecture mise en place par Botero *et al.* [8]. Les performances obtenues par leur système sont prometteuses. A 30 Hz, le filtre de Kalman peut contenir jusqu'à 30 amers et réaliser jusqu'a 6 observations par images.

L'utilisation d'un FPGA pour implanter un EKF-SLAM a été réalisée par Idris *et al.* [71] et Bonato *et al.* [72]. Les équations de la phase d'estimation du filtre ont été optimisées sur une architecture programmable. Les performances obtenues sont très intéressantes. Le principe de base de leur architecture est d'utiliser la redondance des données utilisées pour la multiplication matricielle en évitant les chargements multiples. Des mémoires supplémentaires sont utilisées pour éviter ces chargements, de plus elles permettent d'augmenter le débit mémoire. Leur architecture met en évidence la

nécessité de concevoir des accès mémoires efficaces, surtout dans le cas de matrices de grandes dimensions. Les résultats obtenus permettent de stocker 1800 amers en deux dimensions dans le vecteur d'état et sa matrice de covariance. Cependant, la mise à jour n'est réalisée que pour un seul amer.

1.8 Synthèse

Le tableau 1.1 résume les systèmes récents de SLAM. A partir de l'année 2003, l'utilisation en temps réel est apparue avec le MonoSLAM de Davison [1]. Cette avancée majeure a montré la possibilité de réaliser la cartographie et la localisation simultanée d'une simple webcam en temps réel sur un ordinateur de bureau. Ensuite, des capteurs bas coût ont été utilisés pour concevoir des systèmes de SLAM embarqués. L'implantation réalisée par Klein et Murray [48] sur un iphone est la preuve de la possibilité de résoudre la problématique à l'aide de capteurs bas coût et de calculateurs disposant de ressources limitées.

Plus récemment, l'utilisation d'architectures dédiées est apparue. Ces architectures matérielles sont utilisées pour réaliser de nombreux calculs en parallèle. Botero *et al.* [8] utilisent une architecture matérielle pour réaliser la détection des amers alors que Bonato *et al.* [72] ont conçus une architecture pour paralléliser le calcul matriciel de l'EKFSLAM.

1.9 Conclusion

La résolution du problème de localisation et de cartographie simultanées est étudiée depuis plusieures dizaines d'années. Cependant, il n'apparait pas aujourd'hui de solution idéale. En effet, les algorithmes probabilistes semblent être une solution adaptée aux problèmes lors de l'utilisation de faibles ressources. Cependant, ce type d'approche a montré des faiblesses en particulier lors de l'étude de la consistance des résultats ou de la complexité algorithmique.

Plusieurs propositions ont été réalisées afin d'améliorer la consistance des algorithmes. En particulier, l'utilisation de l'analyse par intervalles permet de garantir l'ensemble des résultats de localisation et de cartographie. Cependant, ce type de méthode n'est performant que dans le cas de bruits bornés.

Les implantations embarquées d'algorithmes de SLAM ne sont pas nombreuses, en particulier si on se limite à l'utilisation de capteurs bas coût et de calculateurs à faibles ressources. Cependant, plusieurs implantations ont fait ressortir l'intérêt principal de l'adéquation algorithme architecture. En effet, pour obtenir des performances satisfaisantes sur une architecture embarquée, il est absolument nécessaire de tirer profit au maximum des capacités spécifiques du calculateur.

Cette adéquation algorithme architecture a poussé des recherches vers l'implantation d'algorithmes sur des architectures programmables. En particulier, des recherches sont actuellement menées pour définir une architecture adéquate à l'EKF-SLAM. Cette architecture a comme objectif d'avoir des performances supérieures aux architectures classiques.

Cette étude bibliographique nous permet de définir plusieurs axes de recherche. Le premier axe consiste à définir une implantation optimisée d'un algorithme de SLAM sur une architecture embarquée. Cette implantation devrait exploiter au maximum les spécificités architecturales du système. Le second axe de recherche consiste à définir une architecture matérielle dédiée à un algorithme de SLAM. En effet, les performances décrites dans les recherches récentes semblent donner de très bons résultats. Enfin, d'un point de vue algorithmique, un axe majeur de recherche apparu ces dernières années est la consistance des algorithmes de SLAM. Nous chercherons donc à améliorer la consistance d'une méthode de SLAM basé sur une méthode ensembliste.

Année	Systèmes	Algorithmes	Capteurs	Calculateur			Avantages / Inconvénients
				PC	RISC	Reconfigurable	
2003	Système de SLAM monoculaire proposé par Davison [1]	EKFSLAM	Webcam	Ordinateur de bureau	-	-	Fonctionnement en temps réel, cartographie d'une petite zone. Nécessite une étape d'initialisation au lancement de l'algorithme.
2007	Abrate *et al.* [66] proposent une implantation basé sur le robot Khepera	EKFSLAM	Télémètre infrarouge	-	Motorala 68331 (25 Mhz)	-	Utilisation de droites pour simplifer la gestion des données capteurs. Réduction du nombre d'amers présents dans le vecteur d'état pour limiter le temps de traitement.
2007	Klein et Murray [48] proposent un algorithme de SLAM par optimisation en décorrélant la cartograhie de la localisation	GraphSLAM	Caméra	-	Iphone (Cortex A8)	-	Le GraphSLAM est implanté sur un processeur à faible ressources. L'utilisation d'une simple caméra embarquée dans un téléphone permet de cartographier de petites zones.
2008	Gifford *et al.* [5] proposent une implantation de SLAM embarqué, multi-robot.	DP-SLAM (environ 20 particules)	Télémètre infrarouge, gyromètres, odométres	-	Gumstix (600 Mhz)	-	La reconstruction de l'environnement est effectuée par coopération entre plusieurs robots. Le principale inconvénient est la lenteur de l'algorithme: 10secondes pour une image à jour avec 25 particules.
2009	Bonato *et al.* [72] proposent une architecture matérielle dédiée à l'EKFSLAM	EKFSLAM	-	-	-	FPGA	L'utilisation d'une architecture matérielle incluant une gestion efficace des accès mémoires permet d'obtenir des performances importantes en particulier pour la multiplication matricielle de l'EKFSLAM.
2010	Magnenat *et al.* [6] proposent un système embarqué dédié au SLAM allant du capteur au calculateur	FastSLAM	Fabriquation d'un télémètre laser rotatif	-	ARM (533 Mhz)	-	L'utilisation d'un capteur conçu pour le SLAM a permis d'obtenir de bon résultats avec des composants trés bas-coût. La bonne paramétrisation de l'algorithme permet de diminuer ces besoins en termes de temps de traitement.
2012	Botero *et al.* [8] proposent une architecture en codesign pour l'EKFSLAM	EKFSLAM	Caméra	-	Micro-Blaze	FPGA	L'utilisation d'un FPGA permet de réaliser la détection des amers à la fréquence pixel. L'algorithme est ensuite implanté de manière logicielle sur le processeur MicroBlaze.

Table 1.1: *Exemples de systèmes embarqués de SLAM depuis 2003*

Chapitre 2

Plateforme expérimentale et méthodologies d'évaluation

Sommaire

2.1 Introduction

Pour un laboratoire, posséder une plateforme expérimentale est généralement très couteux lors de son achat et de l'entretien. Son exploitation demande des ressources financières et humaines importantes. Cependant, une plateforme est indispensable pour valider expérimentalement des concepts, souvent développés en simulation. Initialement, elle fournit un ensemble de données permettant une évaluation différée. Une fois la mise au point terminée, une implantation peut être réalisée directement sur la plateforme.

L'implantation d'un algorithme est souvent précédée d'une étude par simulation, qui ne nécessite pas de ressource matérielle. Comme les données expérimentales, les données issues d'une simulation permettent de valider un algorithme. Elle a l'avantage de pouvoir l'évaluer en utilisant des scénarios extrêmes, souvent impossible à tester en expérimentation. De plus, les résultats sont comparés à une réalité terrain exacte, contrairement aux expérimentations où sa précision est parfois contestable. En effet, il est difficille de connaitre la localisation ainsi que le cap d'un robot mobile durant une expérimentation, de plus cette référence doit être synchronisée avec les données capteurs.

Cette méthodologie est utilisée pour définir et évaluer des algorithmes. De même, pour une architecture, l'évaluation expérimentale est indispensable. Cependant, il est plus compliqué de modifier l'architecture d'un véhicule que son logiciel. L'utilisation d'une plateforme de taille réduite facilite cette évaluation. En effet, les coûts d'achat mais aussi de modification et d'entretien sont nettement inférieurs à ceux d'une plateforme de taille réelle. Nous avons choisi de développer notre plateforme expérimentale pour évaluer les différents systèmes conçus.

La première partie de ce chapitre est consacrée à la présentation de notre plateforme expérimentale. Nous développerons sa conception mécanique, électronique et informatique. La seconde partie est dédiée à la méthodologie d'évaluation d'algorithmes et de systèmes. Finalement, nous présenterons la méthodologie de validation HIL (Hardware in the loop).

2.2 Instrumentation d'une plateforme expérimentale

2.2.1 Objectifs principaux

Le département ACCIS (Architectures, Contrôle, Communication, Images, Systèmes) possède deux plateformes robotiques : la première de taille réelle ([73]) et la seconde de taille réduite ([74]). La plateforme de taille réduite a été conçue pour évaluer des algorithmes de localisation en environnement intérieur. Ces algorithmes sont basés sur la fusion de données provenant de différents capteurs et d'une carte de l'envi-

ronnement. Plus précisément, le mobile est équipé de deux odomètres ainsi que d'une dizaine de capteurs ultrasons.

Ces algorithmes de localisation et la conception de la plateforme MiniTruck ont permis de tirer plusieurs conclusions. Tout d'abord, l'ensemble des algorithmes développés, a pu être évalué, de façon expérimentale, grâce à cette plateforme. Cette évaluation est fondamentale dans le développement d'algorithmes. De plus, cette expérimentation est facilitée par la taille réduite de la plateforme. En comparaison avec le véhicule de taille réelle de l'équipe, la plateforme dispose d'une facilité de mise en œuvre. MiniTruck a été équipé de plusieurs cartes électroniques afin d'exécuter différents algorithmes de manière autonome. Des tests ont démontrés la possibilité d'obtenir une plateforme entièrement autonome. Le principal défaut de MiniTruck est sa conception mécanique. Elle a été conçue sur la base d'un châssis de modèle réduit, reproduisant exactement le modèle d'un camion (allant jusqu'à inclure une boite de vitesse). Cette ressemblance permet de développer des algorithmes pouvant être adaptés sur des véhicules de tailles réelles. Cependant, la plateforme dispose d'un angle de braquage assez faible, très limitant lors d'exploration d'environnement de taille réduite. De par sa conception mécanique, le robot est incapable de réaliser un demi tour dans un espace restreint.

Fort de ces expériences, j'ai choisi de développer une plateforme autonome entièrement adaptée à un environnement d'intérieur. Celle-ci est capable de se déplacer dans un environnement simple composé de couloirs et de bureaux. Elle embarque l'ensemble des outils nécessaires à la reconstruction de scènes : des capteurs, des actionneurs, des cartes contrôleurs mais aussi une source d'alimentation et un moyen de communication avec une base extérieure. Cette plateforme a permis d'évaluer l'ensemble des systèmes proposés.

2.2.2 Mise en oeuvre d'une plateforme : MiniB

Les robots embarquant des algorithmes de SLAM sont souvent coûteux, complexes et pas réellement adaptés. Par exemple, [75] dispose d'un robot de grande taille, incluant plusieurs cartes mères d'ordinateur et plusieurs capteurs onéreux. Plusieurs robots de plus petites tailles ont été développés, en particulier pour le défi CAROTTE. Steux *et al.* [76] a conçu une plateforme adaptée à une environnement d'intérieur, de taille réduite mais équipé d'un ordinateur, non adapté à une utilisation embarquée. De même, la plateforme développée par Baillie *et al.* [77] embarque trois ordinateurs portables. De nombreuses plateformes ont été conçues, cependant il est rare de concevoir un système allant de la plateforme matérielle au logiciel.

Notre plateforme MiniB (Fig. 2.1) est construite sur la base d'un robot mobile pour lui permettre d'évoluer facilement dans un environnement de taille réduite. MiniB a été adapté pour embarquer l'intégralité de l'électronique nécessaire à l'évaluation d'un algorithme de SLAM. La plateforme embarque plusieurs capteurs (Kinect, odomètres)

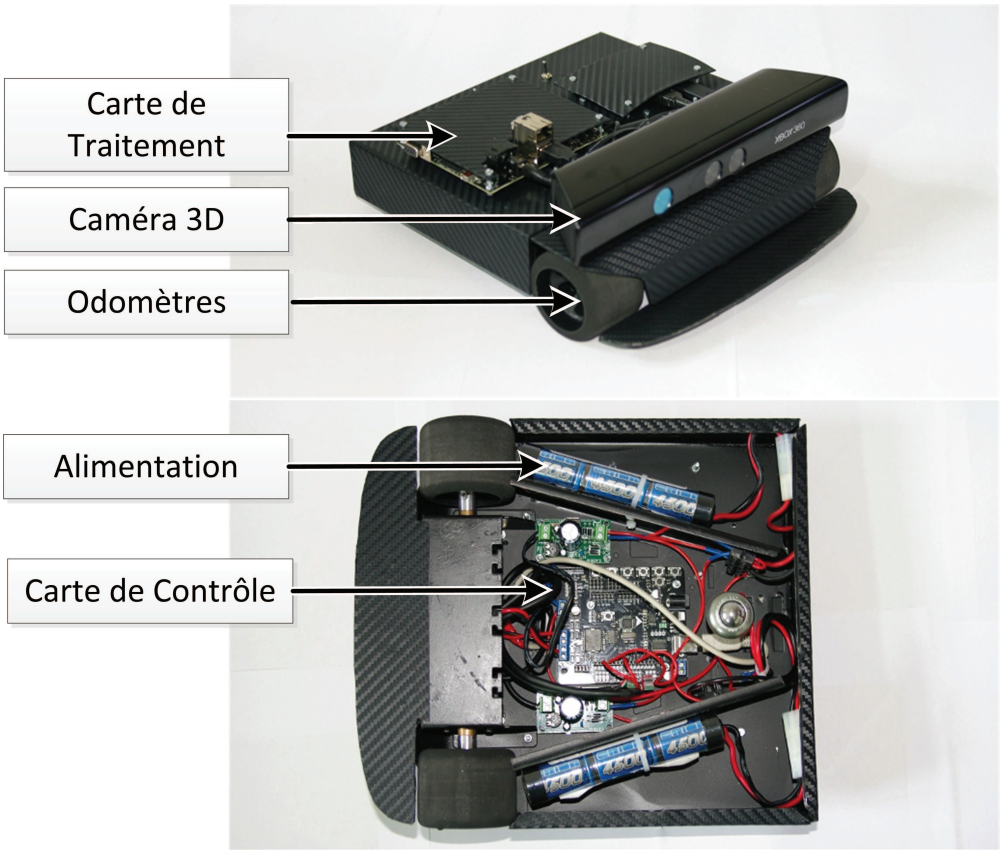


Figure 2.1: *MiniB : Plateforme instrumentée*

ainsi qu'une carte de contrôle.

La figure 2.2 représente la structure électronique de la plateforme MiniB, composée de deux parties. La première partie est dédiée au contrôle de la plateforme : elle réalise l'interfaçage des capteurs et des actionneurs. La seconde partie est dédiée aux traitements de plus haut niveau : l'algorithme de SLAM sera implémenté sur le calculateur embarqué.

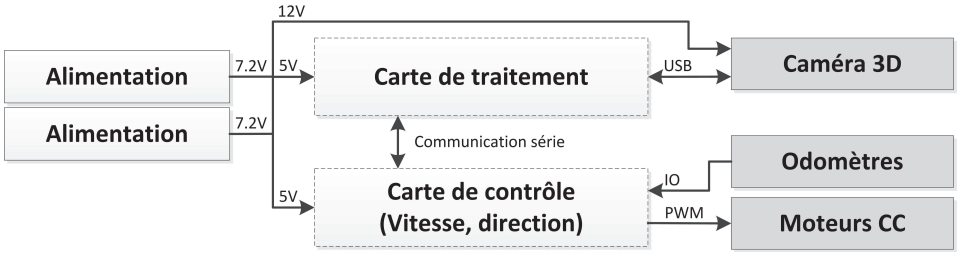


Figure 2.2: *L'architecture du système embarqué par MiniB*

Motorisation

MiniB est équipé de deux moteurs à courant continu (CC) reliés directement aux deux roues, contrôlables de manière indépendante. Ce type de motorisation, couramment utilisé par des robots d'explorations, facilite le déplacement même dans un espace

réduit. En effet, elle donne à la plateforme la capacité d'effectuer une rotation sur un point fixe.

Capteurs

La plateforme embarque deux types de capteurs : des capteurs proprioceptifs et un capteur extéroceptif.
Les capteurs proprioceptifs renseignent sur les mouvements du robot. Nous avons choisi d'équiper le mobile de deux odomètres afin de calculer le déplacement de chaque roue. Les odomètres sont des capteurs fiables et précis. De plus, leur mise en œuvre mécanique est simple : ils sont fixés sur l'arbre de sortie arrière de chaque moteur.
La plateforme est équipée d'un capteur extéroceptif : une caméra (Kinect). Cette dernière fournit une image couleur et une image 3D de la scène observée. Les données 3D n'ont pas été exploitées lors de cette thèse. On se limite à l'utilisation de l'image en niveau de gris (640×480 pixels).

Carte de contrôle

La carte de contrôle, de type Arduino, inclut un microprocesseur ARM Atmega168 (cadencé à 16Mhz). Cette carte contrôle les deux moteurs, décode les données issues des odomètres et communique avec la carte de traitement par le biais d'une liaison série RS232.

Carte de traitement

Les algorithmes de reconstructions de scène seront implanté sur cette carte. MiniB a été conçu pour évaluer différents systèmes et donc recevoir différentes cartes de traitement de nature différentes. En particulier, ces cartes de traitements sont équipées de processeur homogène ou hétérogène. Enfin, ces cartes disposent d'une liaison wifi permettant une liaison bidirectionnelle avec l'opérateur.

Alimentation

La plateforme embarque deux batteries NIMH de 7.2V 3300mAh qui alimentent la caméra Kinect (12V), les moteurs (7.2V) et l'ensemble des autres composants (5V).

Spécifications Techniques

Motorisation

La plateforme est équipée de deux moteurs à courant continu, alimentés par une tension de 7.2V. Avec une vitesse de rotation maximale de 160 tr/min et des roues de 4cm de diamètres, la vitesse maximale atteinte par le robot est de 35 cm/s. Elle est adaptée à notre problématique d'exploration d'environnement intérieur.

Odomètres

Un odomètre de 624 pas/tour est fixé sur chaque moteur des roues. Avec cette précision, une impulsion correspond à un déplacement de 0.2mm. Chaque odomètre fournit des données en quadrature, indispensable pour connaitre le sens de rotation de la roue.

Caméra embarquée

MiniB est équipé d'une caméra 3D (Kinect), placée de manière frontale. Elle fournit des images de 640×480 pixels à la fréquence de 30 images par seconde.

Carte de contrôle

Un coprocesseur (ATMega168) est mis en place pour prétraiter les données issues des capteurs et contrôler les moteurs. Le coprocesseur régule la vitesse du robot et sa direction au moyen de deux PWM (Pulse-Width Modulation), décode les signaux issus des odomètres et communique avec la carte principale à l'aide d'une liaison série RS232. Le processeur principal utilise cette liaison pour récupérer les données odomètriques et envoyer des commandes de vitesse et de direction.

Cartes de traitement

La plateforme est conçue pour évaluer différentes cartes de traitement, deux familles ont été évaluées :

- La première famille est conçue autour d'un processeur hétérogème incluant plusieurs coeurs de natures différentes. Expérimentalement, la carte évaluée est la Gumstix, équipée d'un OMAP3530 constitué d'un coeur ARM Cortex A8 et d'un DSP C64x.
- La seconde catégorie est conçue autour d'un processeur homogène disposant de plusieurs coeurs de même nature. La carte utilisée est la Pandaboard, équipée d'un OMAP4460 disposant de deux coeurs ARM Cortex A9.

Architecture hétérogène Le premier module de traitement est basé sur un processeur OMAP3530. Il s'agit d'une architecture hétérogène composée d'un processeur ARM Cortex-A8 cadencé à 500 MHz et d'un DSP C64x cadencé à 430 Mhz. Le processeur dispose d'un coprocesseur graphique (PowerVR) permettant une accélération 3D.
Le processeur principal, le ARM Cortex-A8, dispose d'un coprocesseur vectoriel SIMD NEON. L'unité de calcul NEON est l'équivalent de SSE sur un processeur X86.
Cette architecture dispose de 512 Mo de mémoire SDRAM et d'une connexion WiFi (802.11g) permettant de contrôler la plateforme à distance.

Concernant la consommation électrique, la carte consomme environ 500 mA, sous 5V avec le wifi activé.

Architecture homogène Le second module de traitement est basé sur un processeur OMAP4460. Ce processeur dispose de deux cœurs ARM Cortex A9, cadencé à une fréquence de 1 Ghz et disposant d'un coprocesseur vectoriel SIMD NEON.
Le cœur Cortex-A9 fournit d'excellents niveaux de performance. Ce processeur est une solution idéale pour les systèmes nécessitant des performances élevées et une consommation électrique faible. Aujourd'hui, les Cortex-A9 sont les cœurs les plus performants fabriqués par ARM et disponibles pour le grand public.
Ce processeur est associé à 1 Go de mémoire DDRAM. Enfin, la carte dispose d'une connexion Wifi permettant un contrôle à distance. Comparé à la carte Gumstix, la Pandaboard dispose d'un processeur de génération plus récente. Elle intègre deux cœurs (contre un pour la gumstix) qui sont cadencés à une fréquence deux fois supérieures (1Ghz contre 500Mhz).
Concernant la consommation électrique, la carte consomme entre 600 et 700 mA, sous 5V, avec le wifi activé.

2.3 Traitement de données capteurs

La plateforme est équipée de plusieurs capteurs fournissant des informations sur l'environnement ou sur le comportement du robot. Pour être correctement interprété, les données issues des capteurs requièrent un traitement. Les données odomètriques sont utilisées pour calculer les déplacements du robot. Les images de la caméra servent à l'identification de points de localisation.

2.3.1 Préambule

On considère que le robot se déplace sur un plan $(\vec{x}_{glob}, \vec{y}_{glob})$. On définit trois repères :

- Le repère global : $R_{glob} = \{\vec{x}_{glob}, \vec{y}_{glob}, \vec{z}_{glob}\}$ définit par la position et l'orientation du mobile à l'instant initial.
- Le repère mobile : $R_{mob} = \{\vec{x}_{mob}, \vec{y}_{mob}, \vec{z}_{mob}\}$ attaché au mobile. L'origine est positionnée au dessus du centre de l'entraxe des roues dans le repère global.
- Le repère camera : $R_{cam} = \{\vec{x}_{cam}, \vec{y}_{cam}, \vec{z}_{cam}\}$ définit par la position de la camera dans le repère global. Elle est orientée vers l'avant du robot.

La figure 2.3 représente les trois repères. Plusieurs changements de repère sont requis pour exprimer les coordonnées d'un point dans les différents repères.

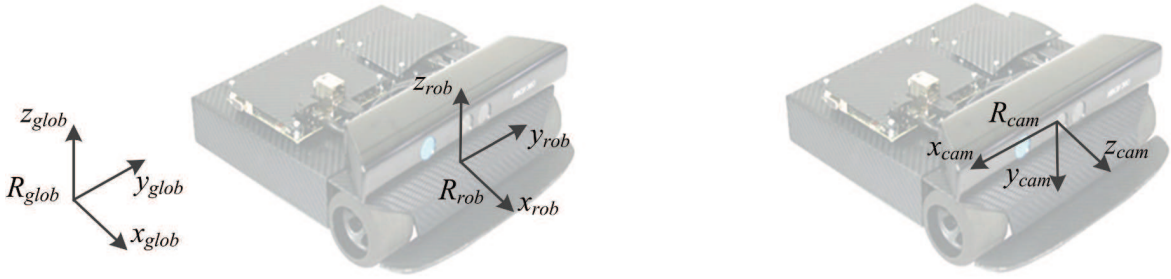


Figure 2.3: *Définitions des repères global, mobile et caméra*

2.3.1.1 La localisation du mobile

Dans le repère R_{glob}, la position du mobile $\mathbf{x}$ est définie par :
- (x, y) les coordonnées 2D de l'entraxe des roues.
- θ l'orientation du mobile représentant l'angle orienté formé par les vecteurs $\vec{x}_{glob}$ et $\vec{x}_{mob}$.

Ces variables définissent l'état du mobile. Elles sont liées par des relations cinématiques et dynamiques : les équations d'états. L'objectif du SLAM est de définir ces grandeurs de manière fiable, rapide et précise.

2.3.1.2 Passage du repère global au repère mobile

Les algorithmes de SLAM nécessitent d'exprimer les coordonnées du robot ou d'un amer dans les différents repères définis précédemment. Pour exprimer des coordonnées de R_{glob} dans R_{mob}, il faut réaliser une translation de vecteur $(x, 0, z)^T$, correspondant à la position du robot, puis une rotation d'angle θ autour de l'axe $\vec{z}_{mob}$. Pour un point $(x_{glob}, y_{glob}, z_{glob})$ de R_{glob}, ces coordonnées $(x_{mob}, y_{mob}, z_{mob})$ dans R_{mob} sont :

$$\begin{pmatrix} x_{mob} \\ y_{mob} \\ z_{mob} \end{pmatrix} = \mathrm{T}_{\mathrm{glob,mob}}(\mathrm{x_{glob}}, \mathrm{y_{glob}}, \mathrm{z_{glob}}) = R_{glob,mob} \left(\begin{pmatrix} x_{glob} \\ y_{glob} \\ z_{glob} \end{pmatrix} - T_{glob,mob} \right) \quad (2.1)$$

$$= \begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) \end{bmatrix} \left(\begin{pmatrix} x_{glob} \\ y_{glob} \\ z_{glob} \end{pmatrix} - \begin{pmatrix} x \\ y \\ 0 \end{pmatrix} \right) \quad (2.2)$$

Avec $R_{glob,mob}$ la matrice de rotation entre le repère globale et le repère mobile et $T_{glob,mob}$ la translation entre les deux repères. La transformation inverse (passage du repère mobile au repère global) est définie par :

$$\begin{pmatrix} x_{glob} \\ y_{glob} \\ z_{glob} \end{pmatrix} = \mathrm{T}_{\mathrm{mob,glob}}(\mathrm{x_{mob}}, \mathrm{y_{mob}}, \mathrm{z_{mob}}) = R_{mob,glob} \begin{pmatrix} x_{glob} \\ y_{glob} \\ z_{glob} \end{pmatrix} + T_{mob,glob} \quad (2.3)$$

$$= \begin{bmatrix} \cos(-\theta) & 0 & \sin(-\theta) \\ 0 & 1 & 0 \\ -\sin(-\theta) & 0 & \cos(-\theta) \end{bmatrix} \begin{pmatrix} x_{mob} \\ y_{mob} \\ z_{mob} \end{pmatrix} + \begin{pmatrix} x \\ y \\ 0 \end{pmatrix} \quad (2.4)$$

2.3.1.3 Passage du repère mobile au repère caméra

Pour exprimer des coordonnées de R_{mob} dans R_{cam}, il faut réaliser une translation de vecteur (x_t, y_t, z_t), correspondant à la position de la caméra par rapport aux odomètres, puis une rotation d'angle (α, β, γ), suivant son orientation, autour des axes $(\vec{\mathbf{x}}_{mob}, \vec{\mathbf{y}}_{mob}, \vec{\mathbf{z}}_{mob})$. Pour un point $(x_{mob}, y_{mob}, z_{mob})$ défini dans le repère mobile ces coordonnées $(x_{cam}, y_{cam}, z_{cam})$ dans le repère caméra sont :

$$\begin{pmatrix} x_{cam} \\ y_{cam} \\ z_{cam} \end{pmatrix} = \mathrm{T}_{\mathrm{mob,cam}}(\mathrm{x_{mob}}, \mathrm{y_{mob}}, \mathrm{z_{mob}}) = R_{mob,cam} \left(\begin{pmatrix} x_{mob} \\ y_{mob} \\ z_{mob} \end{pmatrix} - T_{mob,cam} \right) \quad (2.5)$$

$$= \begin{bmatrix} c(\gamma)c(\beta) & c(\gamma)s(\beta)s(\alpha) - s(\gamma)c(\alpha) & c(\gamma)s(\beta)c(\alpha) + s(\gamma)s(\alpha) \\ s(\gamma)c(\beta) & s(\gamma)s(\beta)s(\alpha) + c(\gamma)c(\alpha) & s(\gamma)s(\beta)c(\alpha) - s(\gamma)s(\alpha) \\ -s(\beta) & c(\beta)s(\alpha) & c(\beta)c(\alpha) \end{bmatrix} \left(\begin{pmatrix} x_{mob} \\ y_{mob} \\ z_{mob} \end{pmatrix} - \begin{pmatrix} x_t \\ y_t \\ z_t \end{pmatrix} \right) \quad (2.6)$$

avec $c(*) = \cos(*)$ et $s(*) = \sin(*)$.

On définit la transformation inverse, du repère caméra au repère mobile :

$$\begin{pmatrix} x_{mob} \\ y_{mob} \\ z_{mob} \end{pmatrix} = \mathrm{T}_{\mathrm{cam,mob}}(\mathrm{x_{cam}}, \mathrm{y_{cam}}, \mathrm{z_{cam}}) = R_{cam,mob} \begin{pmatrix} x_{glob} \\ y_{glob} \\ z_{glob} \end{pmatrix} + T_{cam,mob} \quad (2.7)$$

$$= \begin{bmatrix} c(\gamma')c(\beta') & c(\gamma')s(\beta')s(\alpha') - s(\gamma')c(\alpha') & c(\gamma')s(\beta')c(\alpha') + s(\gamma')s(\alpha') \\ s(\gamma')c(\beta') & c(\gamma')s(\beta')s(\alpha') + c(\gamma')c(\alpha') & s(\gamma')s(\beta')c(\alpha') - c(\gamma')s(\alpha') \\ -s(\beta') & c(\beta')s(\alpha') & c(\beta')c(\alpha') \end{bmatrix} \begin{pmatrix} x_{mob} \\ y_{mob} \\ z_{mob} \end{pmatrix} + \begin{pmatrix} x_t \\ y_t \\ z_t \end{pmatrix} \quad (2.8)$$

avec $\alpha' = -\alpha$, $\beta' = -\beta$ et $\gamma' = -\gamma$.

2.3.1.4 Passage du repère global au repère caméra

Les changements de repère peuvent être combinés. Pour exprimer des coordonnées de R_{glob} dans R_{cam}, on réalise la double transformation : on exprime d'abord les coordonnées dans le R_{mob} puis dans le R_{glob}. On obtient la formule :

$$\begin{pmatrix} x_{cam} \\ y_{cam} \\ z_{cam} \end{pmatrix} = \mathrm{T}_{\mathrm{mob,cam}}(\mathrm{T}_{\mathrm{glob,mob}}(\mathrm{x}_{\mathrm{glob}}, \mathrm{y}_{\mathrm{glob}}, \mathrm{z}_{\mathrm{glob}})) \tag{2.9}$$

Inversement pour exprimer des coordonnées de R_{mob} dans R_{glob}, on utilise la formule suivante :

$$\begin{pmatrix} x_{glob} \\ y_{glob} \\ z_{glob} \end{pmatrix} = \mathrm{T}_{\mathrm{mob,glob}}(\mathrm{T}_{\mathrm{cam,mob}}(\mathrm{x}_{\mathrm{cam}}, \mathrm{y}_{\mathrm{cam}}, \mathrm{z}_{\mathrm{cam}})) \tag{2.10}$$

L'ensemble de ces formules permettent d'exprimer les coordonnées d'un point dans les différents repères.

2.3.2 Exploitation des données proprioceptives

2.3.2.1 Présentation des capteurs

La plateforme est équipée de deux odomètres fixés sur chacune des roues motrices. Les odomètres quantifient les déplacements du robot en évaluant la rotation des roues. Un odomètre est constitué d'un disque troué, d'une source infrarouge et de deux capteurs infrarouges. Sur la figure 2.4, on observe le disque troué (624 fois) et la structure comportant la source infrarouge et les capteurs.
Si le trou est placé entre l'émetteur et le récepteur alors le signal est à l'état haut sinon à l'état bas. Les signaux issus des deux capteurs infrarouges étant en quadrature, il est possible de déterminer s'il y a eu ou non un changement de sens de rotation.

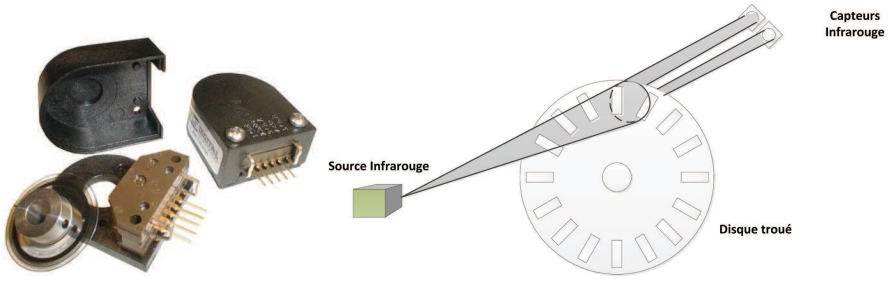


Figure 2.4: *Fonctionnement d'un odomètre*

La figure 2.5 représente les deux signaux émis par un odomètre. Le sens de rotation (de la gauche vers la droite) est déterminé grâce aux instants t et $t-1$, il faut remarquer que lors des interruptions $t-1$ et t, le canal B est à l'état Bas. Pour l'instant $t+1$, deux cas sont possibles :

- Soit la roue a tourné dans le même sens. L'état suivant est l'état $t+1$: le canal B est toujours à l'état bas au moment de l'interruption.
- Soit il y a eu un changement de sens de rotation. L'état suivant est l'état $t'+1$: le canal B est a l'état haut au moment de l'interruption.

Le canal B permet de détecter le sens de rotation.

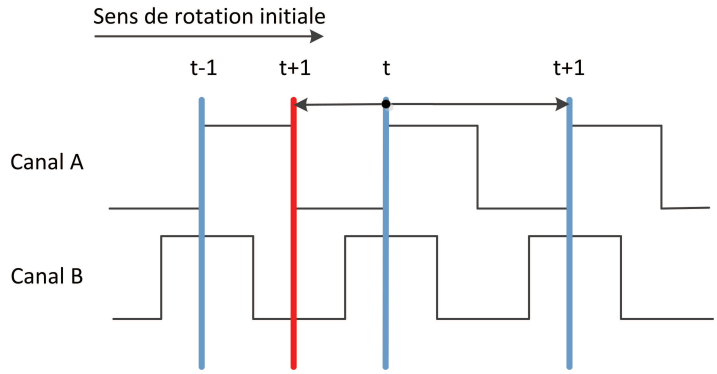


Figure 2.5: *Signaux émis par un odomètre.*

Pour calculer la position du robot, ces déplacements sont modélisés par des morceaux de courbes. La distance parcourue par chaque roue (δ_r, δ_l) est obtenue à partir des données odomètriques (le nombre de pas parcourus par la roue np_r, np_l), de la précision des odomètres (nombre de pas par tour de roue N_{pt}) et du rayon des roues (r). La distance parcourue par chaque roue est définie par :

$$\delta_i = \frac{2\pi r}{N_{pt}} np_i \tag{2.11}$$

Grâce à la distance parcourue par la roue droite (δ_r) et la roue gauche (δ_l), on définit δs le déplacement longitudinal qui correspond à la longueur de l'arc de cercle défini par le déplacement du robot et $\delta\theta$ le déplacement rotationnel qui correspond à l'angle représentant l'arc de cercle. La figure 2.6 représente ces deux grandeurs. On définit analytiquement δs et $\delta\theta$ par :

$$\begin{pmatrix} \delta s \\ \delta\theta \end{pmatrix} = \mathbf{g}(\delta_l, \delta_r) \tag{2.12}$$

$$= \left(\frac{\delta_r + \delta_l}{2}, \frac{\delta_l - \delta_r}{2e} \right) \tag{2.13}$$

Avec $2e$ la taille de l'entraxe en mètre.

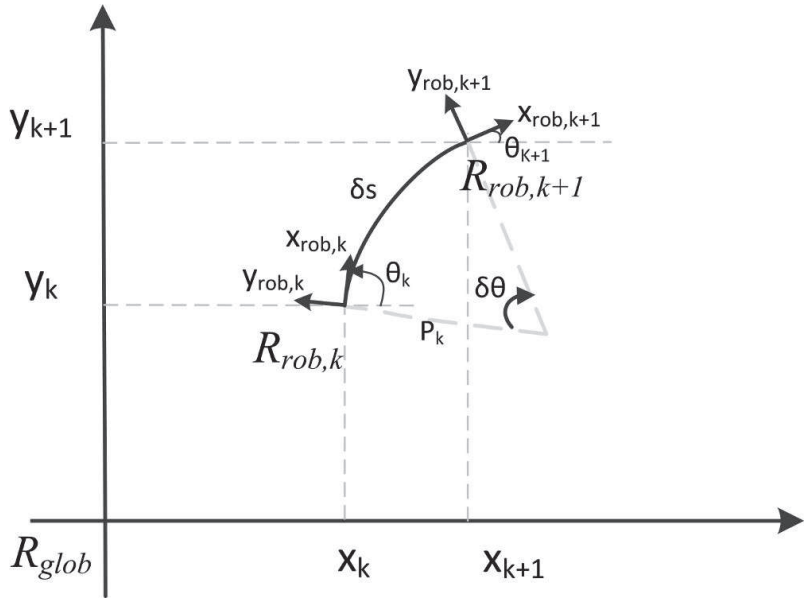


Figure 2.6: *Définition de δs et $\delta\theta$*

2.3.2.2 Localisation par codeur incrémental

Le modèle d'intégration des données odomètriques est une approximation discrète d'un modèle continu par intégration des équations aux différences finies ([78]). A chaque instant k, la configuration du véhicule est donnée par le vecteur d'état $\mathbf{x}_k = (x_k, y_k, \theta_k)^T$ où $(x_k, y_k)^T$ représente les coordonnées 2D du milieu de l'entraxe des roues et θ_k l'orientation du robot. Les odomètres fournissent le vecteur $\mathbf{u}_k = (\delta s, \delta\theta)^T$ représentant le déplacement angulaire et linéaire des roues.

Après un déplacement $\mathbf{u}_k = (\delta s, \delta\theta)^T$, la nouvelle configuration du véhicule est [79] :

$$\begin{aligned} (x_{k+1})_{R_{rob}} &= \rho_k \sin\delta\theta \\ (y_{k+1})_{R_{rob}} &= \rho_k(1-\cos\delta\theta) \\ (\theta_{k+1})_{R_{rob}} &= \delta\theta \end{aligned} \tag{2.14}$$

ou $(x_{k+1}, y_{k+1}, \theta_{k+1})_{R_{rob}}$ est la nouvelle configuration du robot dans le repère R_{rob} (Fig. 2.6) et ρ_k est le rayon de courbure, $\rho_k = \frac{\delta s}{\delta\theta}$. On effectue ensuite le changement de R_{rob} vers le repère global R_{glob} en utilisant la formule :

$$(\mathbf{x}_{k+1})_{R_{glob}} = \begin{pmatrix} \cos\theta_k & \sin\theta_k & 0 \\ -\sin\theta_k & \cos\theta_k & 0 \\ 0 & 0 & 1 \end{pmatrix} (\mathbf{x}_{k+1})_{R'} + (\mathbf{x}_k)_R \tag{2.15}$$

En développant l'équation (2.15), on obtient la nouvelle configuration du mobile :

$$(\mathbf{x}_{k+1})_{R_{glob}} = \begin{pmatrix} x_k + \rho_k(\cos\theta_k \sin\delta\theta - \sin\theta_k(1-\cos\delta\theta)) \\ z_k + \rho_k(\sin\theta_k \sin\delta\theta + \cos\theta_k(1-\cos\delta\theta)) \\ \theta_k + \delta\theta \end{pmatrix} \tag{2.16}$$

En simplifiant l'équation, on obtient :

$$(\mathbf{x}_{k+1})_{R_{glob}} = \begin{pmatrix} x_k + 2\frac{\delta s}{\delta\theta} \sin\frac{\delta\theta}{2} \cos(\theta_k + \frac{\delta\theta}{2}) \\ z_k + 2\frac{\delta s}{\delta\theta} \sin\frac{\delta\theta}{2} \sin(\theta_k + \frac{\delta\theta}{2}) \\ \theta_k + \delta\theta \end{pmatrix} \tag{2.17}$$

En utilisant une approximation des petits angles $\sin\frac{\delta\theta}{2} \approx \frac{\delta\theta}{2}$, la nouvelle configuration du mobile est décrite par l'équation (2.18) :

$$(\mathbf{x}_{k+1})_{R_{glob}} = \begin{pmatrix} x_k + \delta s \cos(\theta_k + \frac{\delta\theta}{2}) \\ z_k + \delta s \sin(\theta_k + \frac{\delta\theta}{2}) \\ \theta_k + \delta\theta \end{pmatrix} \tag{2.18}$$

Ce modèle odomètrique calcule la position de mobile par intégration successive des données issues des odomètres.

2.3.2.3 Résultats de localisation

La figure 2.7 représente les résultats issus de l'intégration des données odomètriques lors d'une expérimentation réalisée avec MiniB. Le robot parcourt un trajet de 2 ×50m le long d'un couloir. A la fin de l'expérimentation, l'erreur euclidienne sur la position est de plus de 35m. Cette erreur est relativement importante car la localisation est obtenue par intégration des données : chaque erreur se propage le long du trajet.

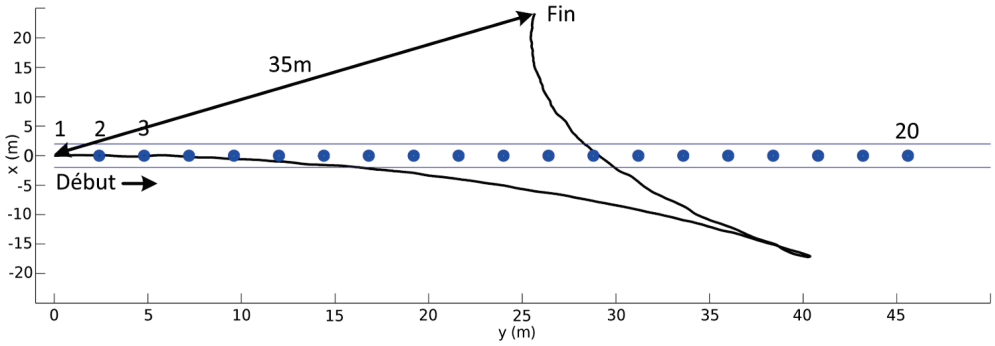


Figure 2.7: *Intégration des données odométriques*

2.3.2.4 Erreurs de mesures

En intégrant les données odomètriques, on connait la position du mobile. Toutefois, le modèle utilisé est soumis à plusieurs types d'erreurs :

- des erreurs sur les caractéristiques du véhicule : il est impossible de calculer exactement la longueur de l'entraxe (e) ainsi que le rayon des roues (r). De plus, ces variables peuvent varier légèrement au cours de l'expérimentation.

- des erreurs sur les données provenant des capteurs odomètriques : Les odomètres sont des capteurs qui discrétisent la distance parcourue. Ils fournissent un intervalle de distance parcourue.

- des erreurs dues aux glissements des roues : les roues peuvent subir des glissements et engendrer des écarts entre la distance réellement parcourue et la distance retournée par le capteur.

2.3.2.5 Intégration sur la plateforme

Notre plateforme expérimentale est équipée d'une carte électronique dédiée aux prétraitements des données capteurs. Cette carte (Arduino) est conçue autour d'un processeur Atmega168. Ce processeur 8bits comptabilise les impulsions issues des deux odomètres. Pour cela, les sortie digitales des odomètres sont connectées à des entrés dédiées du microprocesseur. A chaque front montant, une interruption est générée, le processeur vérifie le sens de rotation puis comptabilise l'impulsion. La carte principale est reliée à cette carte par une liaison série (RS232). Grâce à cette liaison, elle peut consulter le nombre d'impulsion de chaque odomètre. Le nombre d'impulsion est remis à zéro après chaque consultation.

2.3.3 Exploitation des données extéroceptives

2.3.3.1 Présentation

Le robot est équipé d'une caméra frontale monoculaire, elle fournit des images de l'environnement. Elle est utilisée pour détecter des amers qui serviront, par la suite, de repères pour la localisation du mobile. Une caméra est un capteur projetant un environnement à trois dimensions sur un plan à deux dimensions. La caméra couleur embarquée fournie des images de 640×480 pixels à la fréquence de 30 images par seconde.

2.3.3.2 Le modèle Pinhole

Les algorithmes de SLAM utilisent des amers pour reconstruire l'environnement puis se localiser. Il est fréquent de projeter un point de l'environnement sur le plan caméra ou inversement de projeter un point du plan caméra dans l'environnement $3D$.

Le modèle Pin-Hole ([80]) décrit la relation entre les coordonnées d'un point (u, v) sur l'image et sa position $(x^{cam}, y^{cam}, z^{cam})$ dans le repère caméra.

$$\begin{pmatrix} su \\ sv \\ s \end{pmatrix} = \begin{bmatrix} fk_u & fs_{uv} & c_u \\ 0 & fk_v & c_v \\ 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} x_{cam} \\ y_{cam} \\ z_{cam} \end{pmatrix} \tag{2.19}$$

Avec :
- (u, v) la position du point sur l'image en pixel.
- f la distance focale en mm.
- (k_u, k_v) les facteurs d'agrandissement de l'image.
- (c_u, c_v) les coordonnées de la projection du centre optique de la caméra sur le plan image en pixel.
- s_{uv} la non-orthogonalité des lignes et des colonnes de cellules électroniques photosensibles qui composent le capteur de la caméra. Dans notre cas, cette variable est considéré comme nulle.
- $(x_{cam}, y_{cam}, z_{cam})$ la position de (u, v) dans le repère caméra.

En faisant l'hypothèse que $s_{uv} = 0$, le modèle Pinhole devient :

$$\begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} c_u + fk_u \frac{x_{cam}}{z_{cam}} \\ c_v + fk_v \frac{y_{cam}}{z_{cam}} \end{pmatrix} \tag{2.20}$$

On remarque que ce modèle n'est pas inversible car on a 3 inconnues $(x_{cam}, y_{cam}, z_{cam})$ pour seulement deux équations. Ce problème confirme l'impossibilité d'obtenir l'information de profondeur avec seulement une image provenant d'une caméra monoculaire.

L'ensemble des paramètres intrinsèques de la caméra (f, k_u, k_v, c_u, c_v) ont été obtenus en calibrant la caméra avec une toolbox Matlab ("Camera Calibration Toolbox for Matlab"). Numériquement, on obtient $fk_u = fk_v = 520$, $c_u = 319$, $c_v = 230$.

2.3.3.3 Détection de points d'intérêt

Le robot se localise par rapport à des amers, il est obligatoire de les détecter de manière fiable et robuste. Couramment, les algorithmes de SLAM utilisent des points d'intérêts pour se localiser. Ces points d'intérêts correspondent à des points distinctifs de l'environnement. Plusieurs algorithmes de détection de points d'intérêts existent : SURF [81], SIFT [82], Harris [83], Shi et Tomasi[84]. L'algorithme FAST (Features from Accelerated Segment Test, [85]) détecte rapidement des points d'intérêts et produit des résultats suffisamment stable pour notre application. De plus, [86] a comparé l'ensemble des détecteurs de points d'intérêts dans le contexte d'un algorithme de SLAM. Son étude a montré que SURF donne les meilleurs résultats de détection, cependant son

temps de traitement est incompatible avec une implantation embarquée. De ce fait, nous avons choisi le détecteur FAST qui permet une détection rapide et fiable des amers.

FAST utilise une image en niveau de gris. Pour chaque pixel p de l'image, des tests sont réalisés sur les pixels appartenant à un cercle discrétisé de rayon 3 pixels. La figure 2.8 représente les 16 pixels qui sont testés pour détecter si le pixel p est un amer potentiel.

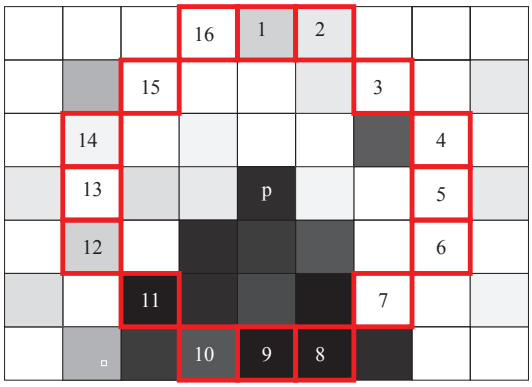


Figure 2.8: *Pixels considérés pour le test du pixel "p"*

Si n (classiquement $n = 9$) pixels voisins consécutif sont plus "lumineux" ou plus "sombre" d'au moins ε que le pixel p, alors le centre est considéré comme un point d'intérêt potentiel.

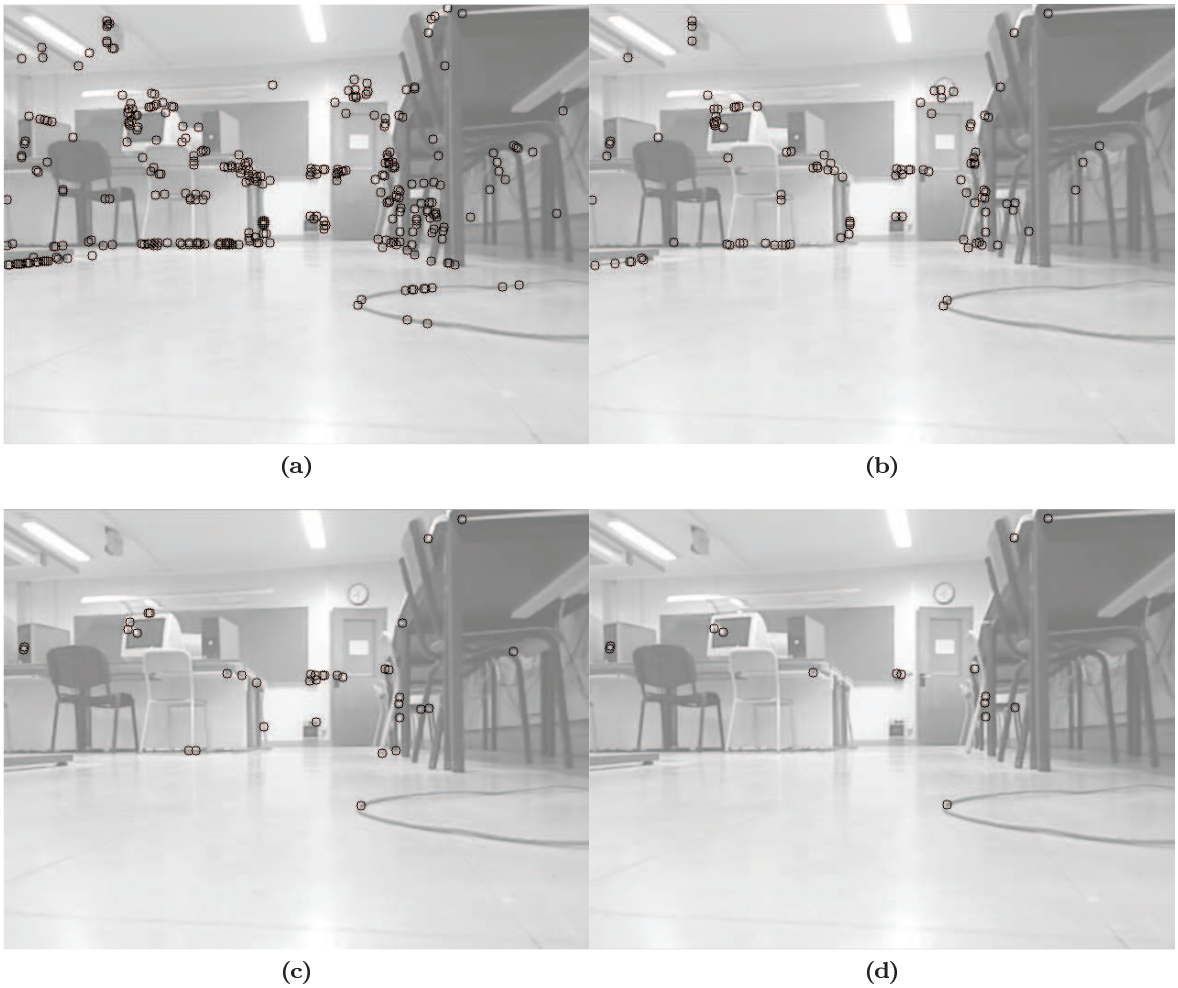
En utilisant ce test, il est fréquent de détecter plusieurs points d'intérêts potentiels dans une zone très restreinte. Un filtrage est réalisé pour ne garder que les maximas locaux. On associe un poids à chaque point d'intérêt potentiel en utilisant la formule définie dans [85] :

$$V_p = \max\left(\sum_{x \epsilon S_{bright}} |I_{p\to x} - I_p| - \varepsilon, \sum_{x \epsilon S_{dark}} |I_{p\to x} - I_p| - \varepsilon\right) \tag{2.21}$$

avec :

- S_{bright} l'ensemble des pixels plus lumineux d'au moins ε que le pixel p.
- S_{dark} l'ensemble des pixels plus sombre d'au moins ε que le pixel p.
- $I_{p\to x}$ la valeur d'un pixel appartenant à S_{bright} ou S_{dark}.
- I_p la valeur du pixel p codé sur 8 bits.
- ε le seuil pour considérer un pixel plus lumineux ou sombre. (classiquement $\varepsilon \approx 75$)

Enfin, on supprime les non-maxima locaux dans un voisinage de 5 pixels. Rosten *et al.* [85] montrent que les points détectés sont stables et que son détecteur est très rapide : la détection est réalisé en environ 5ms pour une image de 768×288 pixels sur un Pentium



(a) (b)

(c) (d)

Figure 2.9: *Détection de points d'intérêts sur des images expérimentales avec un seuil =30, 50, 80 ,100*

III 850Mhz. La figure 2.9 présente le résultat de l'algorithme sur une image extraite d'une séquence de test. Plus le seuil est élevé, plus le nombre de points d'intérêts détecté est faible.

Cet algorithme a été défini en 2008. Depuis, il est fréquemment utilisé par des algorithmes de SLAM ou plus généralement de traitement d'images. Il a fait l'objet de deux implantations importantes. La première, utilisée par Rosten *et al.* [85], consiste à utiliser un algorithme de "Machine Learning" pour accélérer les calculs ([85]). Cette implantation est utilisée par nos algorithmes de SLAM, le code du programme étant disponible sur le site de l'auteur.
La seconde est une implantation sur une architecture programmable type FPGA ([87]). Celle-ci montre la possibilité d'utiliser une architecture dédiée pour cet algorithme, les performances sont très intéressantes : l'implantation peut traiter 100 images de taille 512×512 par seconde avec une fréquence de seulement 100Mhz.

2.3.3.4 Mise en correspondance

Les algorithmes de SLAM utilisent les points d'intérêts pour cartographier l'environnement et se localiser. Il est nécessaire de redétecter des amers ayant déjà été vus. Cette étape de mise en correspondance entre des amers connus et de nouvels amers permet au robot de se relocaliser dans l'environnement mais aussi d'améliorer la carte de l'environnement.

La mise en correspondance de points d'intérêts est un domaine très étudié. Classiquement, un descripteur est associé à chaque point d'intérêt, il consiste soit en une fenêtre de pixels entourant le point soit en un détecteur plus complexe calculé à l'aide des pixels voisins. Puis, on utilise une distance entre les deux descripteurs pour calculer un taux de similarité. Pour permettre une implantation efficace, nous avons choisi d'utiliser un descripteur simple : la fenêtre d'image entourant le point d'intérêt de dimension 16×16 pixels, déja utilisé par Davison [1]. Schmidt et al. ([86]) a étudié plusieurs distances utilisées pour la mise en correspondance de ce type de descripteurs : SAD (Sum of Absolute Differences), NCC (Normalized Cross Correlation), SSD (Sum of Squared Differences). SSD est la distance qui donne les meilleurs résultats de mise en correspondance. Nous avons choisi d'utiliser une version légèrement modifiée du SSD, le ZMSSD (Zero Mean Sum of Squared Differences), qui améliore le comportement de la distance par rapport aux changements de luminosité. ZMSSD s'écrit :

$$N_p = \sum_{i,j}((d(i,j) - m_d) - (im(p_x + i - \frac{des}{2}, p_y + j - \frac{des}{2}) - m_i))^2 \quad (2.22)$$

avec

- $i \in [0; des - 1]$, $j \in [0; des - 1]$ avec des la taille du descripteur en pixel.
- d le descripteur.
- m_d et m_i la moyenne des pixels du descripteurs et de la fenêtre d'image.
- im l'image.

2.3.3.5 Résultats expérimentaux

La figure 2.10 représente la mise en correspondance des points d'intérêts entre deux images consécutives. La zone de recherche de correspondance est définie comme étant proche de la position précédente (d<30pixels). Cette zone de recherche sera calculée plus précisément par nos algorithmes de SLAM, à partir de la carte et de la position du mobile. L'algorithme cartographie l'environnement permettant une projection des amers dans l'image. Cette projection définira une zone de recherche pour l'étape de mise en correspondance.

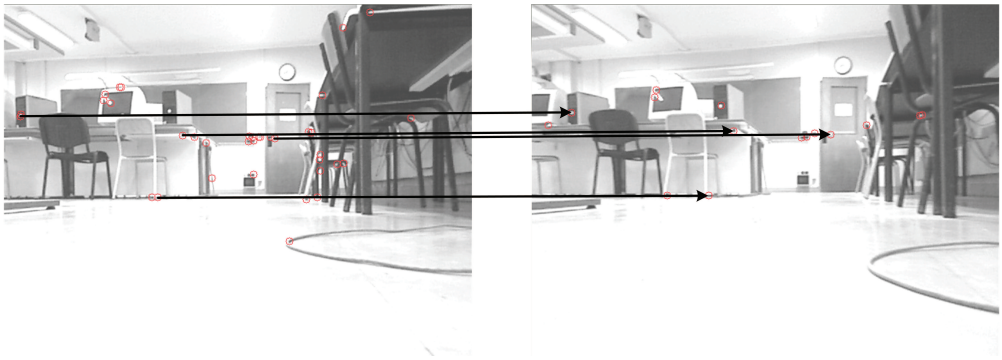


Figure 2.10: *Résultats de la mise en correspondance des points d'intérêts*

2.4 Méthodologie d'évaluation

De nombreuses recherches ont été effectuées pour développer des algorithmes de SLAM. Beaucoup d'algorithmes ont été évalués expérimentalement ou en simulation. L'évaluation d'un algorithme vérifie ses bons résultats. Elle est souvent divisée en deux parties. La première évalue les résultats à l'aide d'un simulateur puis la seconde utilise des données expérimentales.
Nous avons choisi d'évaluer les algorithmes à l'aide de données simulées mais aussi de données réelles. Les premières sont issues d'un simulateur que j'ai développé et les suivantes sont collectées à l'aide de la plateforme précédemment décrite.
L'évaluation des systèmes et des architectures définis requiert une méthode différente. Les systèmes sont évalués à l'aide de données réelles, simulées mais aussi en situation réelle. Enfin, ils sont validés à l'aide d'une méthodologie HIL (Hardware In the Loop).

2.4.1 Développement d'outils de simulations

2.4.1.1 Développement d'un simulateur

Les simulateurs de robot et d'environnement sont très nombreux. Nous avons cependant choisi de développer notre propre simulateur pour conserver les formats de données et les interfaces que nous avons défini avec notre plateforme.
Le simulateur a été développé sous Matlab. Il simule un environnement de taille variable, constitué d'un ensemble de mur et d'amers permettant de modéliser des environnements structurés tel que des couloirs ou des étages complets de batiments. Les amers sont disposés aléatoirement dans l'environnement. Le trajet suivi par le mobile est composé de points de passage, reliés par un ensemble de courbes et de droites.
Pour les données proprioceptives, durant le trajet, le simulateur renvoie les données odomètriques issues du parcours du mobile. Ces données peuvent être altérées en utilisant différents bruits (gaussien, uniforme, biaisé).
Pour les données extéroceptives, le simulateur modélise une caméra sur le robot, il

calcule la projection de chaque amer sur l'image de la caméra. Puis, il envoie le numéro de l'amer détecté ainsi que sa position sur l'image. L'association entre un amer nouvellement détecté et un amer précédemment détecté est supposée connue grâce au numéro des amers. Le simulateur ne gère pas les problèmes d'appariement.

2.4.1.2 Définition du modèle du robot

Notre plateforme expérimentale a été modélisée pour permettre une simulation réaliste. Nous avons défini les variables suivantes :

- Diamètre de la roue : 4cm.
- Entraxe : 20cm.
- Caméra : $fku = fkv = 520$.
- Caméra: $c_u = 320$, $c_v = 240$.
- Taille de l'image: 640×480pixels.
- La caméra est située au dessus du centre de l'essieu.

L'ensemble de ces paramètres a été défini en fonction de notre plateforme. Chaque variable est utilisée à la fois pour les données simulées mais aussi pour les données réelles.

2.4.1.3 Définition des environnements de tests

Les algorithmes de SLAM ont des comportements différents en fonction de plusieurs paramètres de l'environnement. Le résultat de la localisation dépend du nombre d'amers présents dans l'environnement mais aussi de la taille de ce dernier ou encore de la visibilité des amers. On définit trois cartes de test qui seront utilisées pour évaluer nos différents algorithmes.

Environnement 1

Le premier environnement représente une petite salle, constituée de 4 murs. Elle a une taille de 7m×7m soit 49m^2. Cette carte modélise, par exemple, le cas du déplacement dans une salle de type "salle de classe" ou une "salle de musée".
La figure 2.11 a) représente la carte ainsi que le trajet parcouru par le mobile. Cette carte peut être remplie avec un nombre d'amers variables.

Environnement 2

Le second environnement reproduit une grande salle et inclut deux murs. Cette pièce a une taille de 15m×15m soit 225m^2. Cette carte schématise une salle de plus grande taille où la visibilité n'est pas possible d'un bout à l'autre de la salle.
La figure 2.11 b) représente la carte définie. Comme pour la première carte, cette carte peut être remplie avec un nombre d'amers variables.

Environnement 3

Le troisième environnement synthétise un ensemble de couloirs. Les algorithmes de SLAM sont régulièrement utilisés pour schématiser un bâtiment complet. Ces couloirs ont une largeur de 3 à 4m.
La figure 2.11 c) représente la carte et le trajet. Ce dernier explore la totalité de la carte en réalisant plusieurs fermetures de boucle. Cela correspond au passage du mobile dans un espace déjà exploré. La principale difficulté d'une exploration de couloir est la faible visibilité des amers.

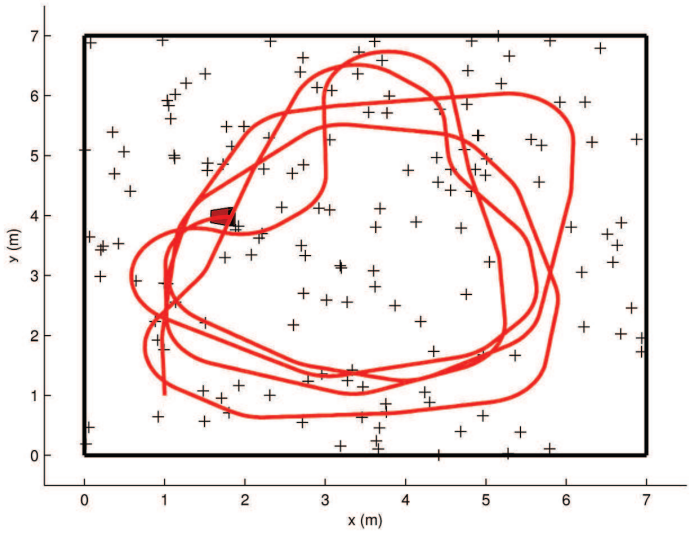
Synthèses des différents environnements

Trois environnements ont été définis. Ils représentent des situations différentes : L'exploration d'une petite salle, l'exploration d'une grande salle et l'exploration d'un ensemble de couloirs. Les trois situations présentent des difficultés diverses pour un algorithme de SLAM. En particulier, elles mettent en évidence le problème de la taille de l'environnement ou celui de la visibilité des amers.

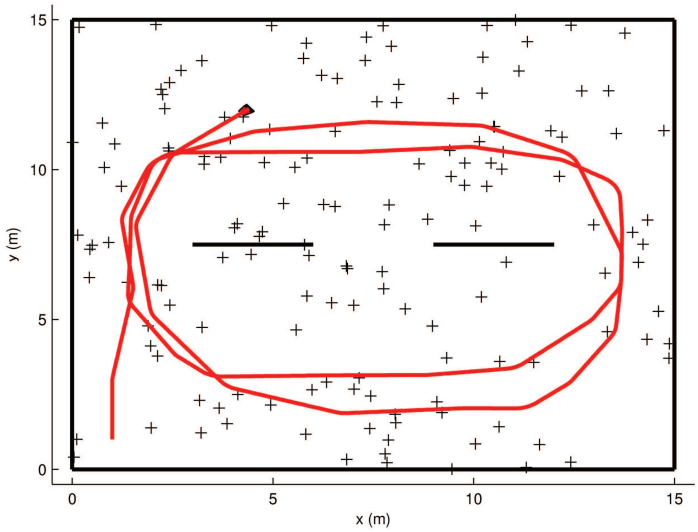
2.4.1.4 Données odomètriques

Le simulateur dispose d'un trajet préprogrammé constitué de points de passage. Au fur et à mesure du trajet, le simulateur envoie des données odomètriques en fonction des déplacements du mobile. Nous avons défini précédemment les caractéristiques de notre plateforme, en particulier, le rayon de la roue (2cm) et l'entraxe (20cm).
Les odomètres sont des capteurs imprécis soumis à plusieurs types d'erreurs. Pour bruiter les données odomètriques, deux bruits sont paramétrables. Le premier correspond aux bruits de glissement des roues : les roues peuvent subir des glissements et engendrer des écarts entre la distance réellement parcourue et la distance retournée par le capteur. Ce bruit affecte directement les valeurs issues du compteur de pas odomètriques. Le second bruit affecte les valeurs des caractéristiques du robot : la valeur réelle du rayon de la roue ou de l'entraxe peut être légèrement différente de la valeur prévue.
Ces valeurs (rayon, entraxe) peuvent être biaisées, dans ce cas, les deux valeurs restent constantes durant l'expérimentation et modélisent une petite erreur de mesure. Soit on affecte un bruit sur chacune des deux valeurs, ces valeurs évoluent donc au fils du temps, en effet le rayon de la roue peut varier légèrement en fonction du sol mais aussi de la trajectoire.

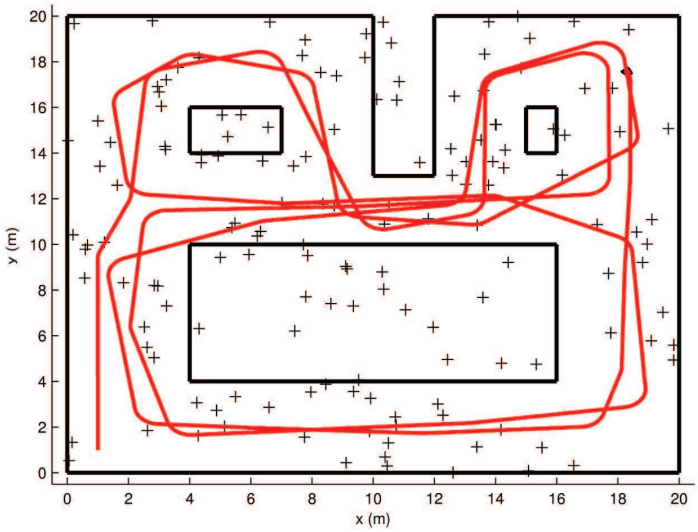
La figure 2.12 représente la trajectoire calculée à l'aide des odomètres (rouge) et la trajectoire réelle (noir). L'entraxe du mobile a été biaisé de -5mm. Au début de la trajectoire (1,1), les deux courbes se chevauchent parfaitement. Au fur et à mesure de l'expérimentation, la trajectoire décrite par l'intégration des données odomètriques



(a) *Environnement 1*



(b) *Environnement 2*



(c) *Environnement 3*

Figure 2.11: *Définition des environnements et trajets de simulation*

s'éloigne de la trajectoire réelle.

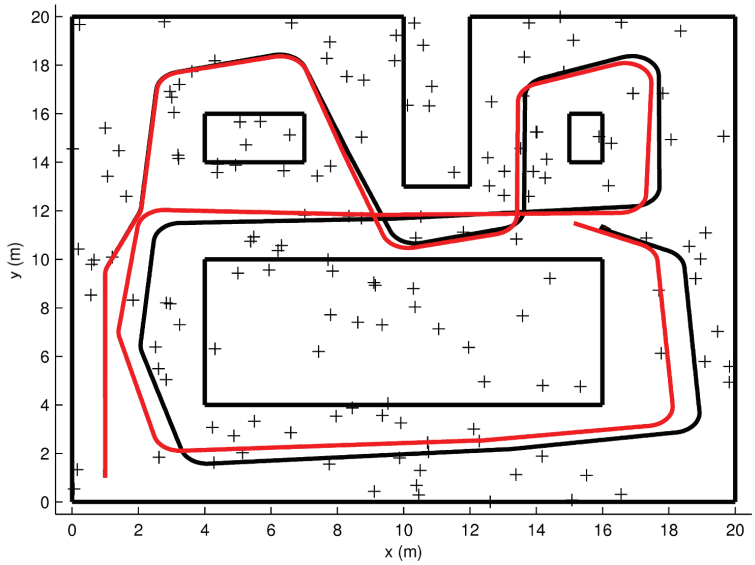


Figure 2.12: *Trajectoire odomètrique bruitée*

2.4.2 Données expérimentales

Après avoir évalué en simulation les algorithmes, il faut les valider expérimentalement. Un jeu de données a été collecté pour réaliser une évaluation "offline". Le jeu de donnée consiste en un parcours dans une salle de travaux pratiques d'environ 10m×10m. La figure 2.13 représente le trajet réalisé ainsi que l'intégration odomètrique.
Le trajet a été réalisé à une vitesse réduite correspondant à une problématique d'exploration. Durant cette expérimentation, des points de référence ont été tracés sur le trajet pour permettre une évaluation de la localisation. Ces de points passages sont représentés sur la figure 2.13.

L'ensemble des systèmes est aussi évalué à l'aide d'expérimentation réalisée en temps réel. Cependant, durant ces expérimentations, il n'est pas possible d'enregistrer les données issues des capteurs. En effet, notre électronique ne permet pas le traitement et l'enregistrement simultanés des données à cause d'une limitation de débit des cartes SD.

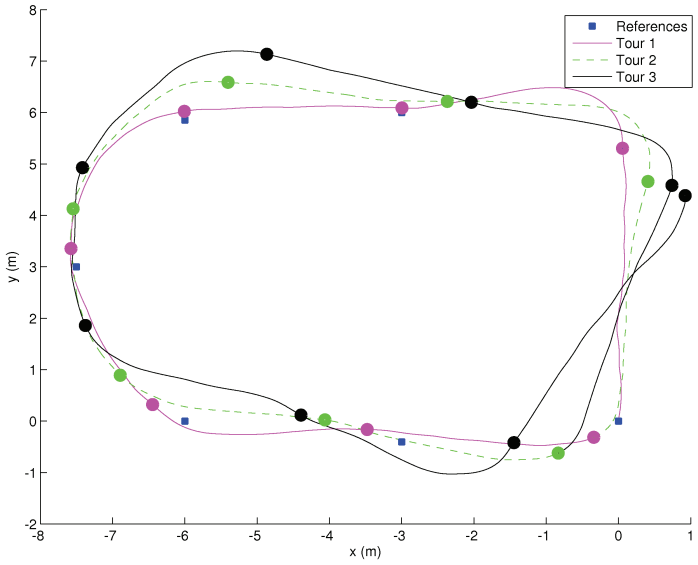


Figure 2.13: *Parcours réalisé lors de l'expérimentation*

2.4.3 Critères d'évaluation

Pour évaluer correctement un algorithme ou un système, des critères sont définis pour quantifier ses performances. Les algorithmes de SLAM localisent un mobile et cartographient un environnement. Il faut évaluer la localisation du robot mais aussi la reconstruction de l'environnement. Plusieurs critères sont couramment utilisés comme la distance euclidienne ou le NEES ([35]). Enfin, on étudie le temps de traitement des algorithmes en fonction des architectures.

2.4.3.1 Evaluation de la localisation du mobile

Position de référence

Pour évaluer la localisation du mobile, une position de référence est indispensable. Elle correspond à la position réelle de la plateforme. En simulation, il est facile d'obtenir cette référence car le simulateur la calcule. Lorsque l'on réalise des expérimentations, il est plus compliqué de l'obtenir. Pour créer cette référence, lors des expérimentations, nous avons tracé au sol des points de passage qui serviront de positions de référence. Cependant, celle-ci est moins précise que celle utilisée en simulation. De plus, elle ne nous renseigne pas sur l'orientation du mobile.

Distance euclidienne

Pour évaluer la qualité de la localisation du mobile, on se concentre tout d'abord sur la distance euclidienne entre la position réelle du mobile et sa position estimée. Cette distance représente l'erreur commise par l'algorithme à chaque instant de la trajectoire. On utilise la distance euclidienne entre la position de référence $\mathbf{x}_k = (x_{k,ref}, y_{k,ref})$ du

robot et sa position estimée $\hat{\mathbf{x}}_k = (x_k, y_k)$ à chaque instant t.
La distance est calculée à l'aide de la formule :

$$d = \sqrt{(x_{k,ref} - x_k)^2 + (y_{k,ref} - y_k)^2} \tag{2.23}$$

Cette distance représente l'erreur commise par l'algorithme sur la localisation du mobile.

Couloir d'incertitude

La distance euclidienne renseigne sur l'erreur commise sur la localisation estimée. En plus de la position estimée, les algorithmes de localisation fournissent une incertitude sur celle-ci. Chaque estimée (x_k, y_k, θ_k) est associée à une variance $(\sigma_{x_k}, \sigma_{y_k}, \sigma_{\theta_k})$. Pour prendre en compte la variance de chaque variable, on trace un couloir d'incertitude. On définit trois couloirs e_x, e_y, e_θ correspondant aux trois variables estimées:

$$e_x = \begin{pmatrix} x_{k,ref} - x_k - k\sigma_{x_k} \\ x_{k,ref} - x_k + k\sigma_{x_k} \end{pmatrix} \tag{2.24}$$

$$e_y = \begin{pmatrix} y_{k,ref} - y_k - k\sigma_{y_k} \\ y_{k,ref} - y_k + k\sigma_{y_k} \end{pmatrix} \tag{2.25}$$

$$e_\theta = \begin{pmatrix} \theta_{k,ref} - \theta_k - k\sigma_{\theta_k} \\ \theta_{k,ref} - \theta_k + k\sigma_{\theta_k} \end{pmatrix} \tag{2.26}$$

A partir de ces couloirs, on définit la notion de consistance. La localisation issue d'un algorithme est consistante si la position réelle du mobile est incluse dans la zone définie par la localisation estimée du mobile incluant l'incertitude. Par conséquence, un algorithme est consistant si son couloir d'incertitude inclut à chaque instant la valeur zéro. En effet, ce cas valide l'inégalité $\hat{\mathbf{x}}_k - k\sigma_{\mathbf{x}_k} < \mathbf{x}_k < \hat{\mathbf{x}}_k + k\sigma_{\mathbf{x}_k}$.

NEES

Le couloir d'incertitude renseigne sur la consistance de la localisation. Cependant, le couloir ne prend pas en compte la forme de l'ellipse d'incertitude : son orientation n'est pas réellement utilisée. Pour combler ce manque, on utilise le NEES défini pour N expérimentations par :

$$\epsilon_{rob} = \frac{1}{N}\sum_{j=1}^{N}(\hat{\mathbf{x}}_k - \mathbf{x}_k)^T \hat{\sigma}_{\hat{\mathbf{x}}_k}(\hat{\mathbf{x}}_k - \mathbf{x}_k) \tag{2.27}$$

Pour 3 dimensions, et N = 50, la zone de 95% de probabilité est définie par l'intervalle

[2.36, 3.72]. Si ϵ est supérieur à la borne supérieure de l'intervalle alors le filtre est optimiste : la localisation du robot n'inclut pas la position réelle. Si ϵ est inférieur à la borne inférieure alors le filtre est conservateur : la zone de localisation du robot est plus grande que la normale.

2.4.3.2 Evaluation de la carte de l'environnement

Bien que la qualité de la localisation est fortement corrélée à la qualité de la cartographie Huang et Dissanayake [88], on choisit d'étudier la qualité de la carte reconstruite par les algorithmes.

Distance Euclidienne

Pour évaluer la carte de l'environnement reconstruite par un algorithme de SLAM, une carte de référence est indispensable. Malheureusement, il n'est pas possible d'évaluer la qualité de la carte lors d'une expérimentation. Nos algorithmes de SLAM utilisent des amers ponctuels, il est impossible d'établir une carte de référence comportant la position 3D de ces amers.
En simulation, la carte de référence est connue par le simulateur. La position réelle de chaque amer $\mathbf{A}_i$ est définie par le vecteur $\mathbf{A}_i = (x_{i,ref}, y_{i,ref}, z_{i,ref})$. On définit la distance euclidienne entre la position de référence et la position estimée d'un amer $\hat{\mathbf{A}}_i = (x_i, y_i, z_i)$:

$$d_{amer} = \sqrt{(x_{i,ref} - x_i)^2 + (y_{i,ref} - y_i)^2 + (z_{i,ref} - z_i)^2} \tag{2.28}$$

Cette distance représente l'erreur d'estimation entre la position réelle de l'amer et la position estimée par le filtre. Plus la carte est précise, plus la distance moyenne est faible.

NEES

En plus de fournir la position de chaque amer $\hat{\mathbf{A}}_i = (x_i, y_i, z_i)$, les algorithmes de SLAM fournissent une incertitude sur chaque variable $\hat{\sigma}_{A_i} = (\sigma_{x_i}, \sigma_{y_i}, \sigma_{z_i})$. Pour prendre en compte l'incertitude de chaque variable, on définit le NEES (Normalized Estimation Error Squared). Pour calculer le NEES, plusieurs expérimentations sont nécessaires (N expérimentation). Après avoir réalisé N expérimentations, le NEES est calculé en utilisant la formule :

$$\epsilon_{amer} = \frac{1}{N}\sum_{j=1}^{N}(\hat{\mathbf{A}}_i - \mathbf{A}_i)^T \hat{\sigma}_{A_i}(\hat{\mathbf{A}}_i - \mathbf{A}_i) \tag{2.29}$$

Comme précédemment, pour 3 dimensions, et N = 50, la zone de 95% de probabilité est définie par l'intervalle [2.36, 3.72].

2.4.3.3 Evaluation du temps de traitement

Après avoir évalué les résultats d'un algorithme, il faut évaluer son temps de traitement. En première approche, on calcule le temps d'éxecution de l'intégralité de l'algorithme. Ces temps sont calculés soit en microsecondes à l'aide d'un timer (disponible sur chaque carte de traitement), soit en nombre de cycles (pour notre architecture programmable).
Généralement, le temps d'execution d'un algorithme de SLAM n'est pas constant. Il est donc nécessaire de décomposer l'algorithme sous forme de blocs fonctionnels ayant des temps de traitement constant. Ces blocs permetent de comparer les temps d'executions entre les différentes architectures.

2.4.4 Validation Hardware In the Loop

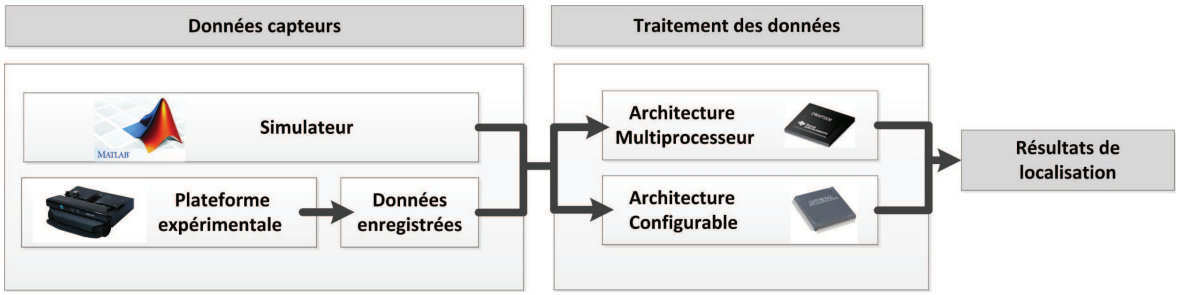


FIGURE 2.14: *Outils de développement et de validation*

Nous avons défini précédemment différentes mesures pour évaluer les résultats d'un algorithme de SLAM. Cette évaluation est basée sur la qualité de la localisation du mobile ou de la localisation des amers. Le SLAM ne se limite pas à un algorithme mais peut être considéré comme un système incluant des capteurs, des calculateurs et une plateforme. Les algorithmes de SLAM augmentent l'autonomie des robots, qui doivent avoir la capacité d'embarquer l'intégralité du système.
Pour évaluer un système de SLAM, il faut valider l'algorithme mais aussi le système dans son ensemble. Pour cela, nous avons développé un système de validation HIL (Hardware in the loop). La validation HIL est couramment utilisée pour valider un système matériel ([89, 90]), elle consiste à inclure des cartes de traitements matériels dans la chaîne de traitement logiciel. Dans notre cas, cet outil permet d'évaluer le comportement d'un système de SLAM en utilisant plusieurs sources de données et plusieurs méthodes de traitements. L'intérêt majeur est d'évaluer nos systèmes à l'aide du simulateur et d'une référence précise.

La figure 2.14 détaille l'intégralité des possibilités de notre système d'évaluation "Hardware In the Loop". Le schéma différencie la partie capteur de la partie traitement. Deux sources sont disponibles pour fournir des données capteurs :

- Le simulateur Matlab : Il peut être utilisé comme source de données capteurs. Le simulateur fournit des données odomètriques et la position des amers sur l'image. Les données du simulateur sont transmises par liaison RS232. La liaison RS232 a été choisie car le volume de données à transmettre est beaucoup moins important que pour la transmission d'une image complète.
- Le rejeux de données : après avoir effectué une expérimentation, il est possible de rejouer les données capteurs. Ce mode de fonctionnement est principalement utilisé pour mettre au point et évaluer nos algorithmes. Un prétraitement des données peut être effectué, par exemple pour extraire les amers des images.

Deux sources sont disponibles pour traiter les données capteurs :

- Des architectures multiprocesseurs : deux cartes processeurs ont été interfaçées avec notre système. Il s'agit d'une carte Gumstix et d'une Pandaboard. Grâce à ce système, on peut valider les différents calculateurs embarqués.
- Une architecture configurable (FPGA) : dans le chapitre 4, nous définissons une architecture programmable adaptée à la problématique du SLAM. Cette architecture a été validée grâce à des données simulées mais aussi à des données enregistrées. Cette méthode de validation permet de valider l'intégralité de notre architecture.

Le système d'évaluation mis en place permet de tester à la fois les résultats des algorithmes en utilisant des données simulées mais aussi des données réelles. Il valide des systèmes en interfaçant directement des architectures matérielles au simulateur ou à des données réelles.

2.5 Bilan

La mise au point d'un système de SLAM requiert une grande réflexion concernant les parties matérielle et logicielle. Chacune de ces parties doit être évaluée pour valider le bon fonctionnement de l'ensemble.

Durant ce chapitre, nous avons conçu et instrumenté une plateforme robotisée pour disposer des moyens nécessaires à la conception et l'évaluation d'un système de SLAM embarqué. La plateforme est adaptée à un environnement d'intérieur lui permettant de se déplacer facilement dans un espace réduit. Elle embarque plusieurs capteurs fournissant les données indispensables à la reconstruction d'une scène. Enfin, elle intégre la puissance de calcul nécessaire pour reconstruire la carte et se localiser de manière autonome.

L'évaluation des algorithmes et des architectures matérielles requiert plusieurs outils.

Un simulateur a été développé pour évaluer les différents systèmes et algorithmes dans toutes les situations possibles. De plus, un système HIL a été développé pour évaluer des systèmes matériels en incluant le matériel dans la chaîne de traitement.

Au delà du travail réalisé, la plateforme a été conçue pour recevoir de nouveaux capteurs, de nouveaux actionneurs ou encore de nouvelles cartes de traitement. MiniB est une base solide dédiée aux développements et l'évaluation de systèmes. Elle peut être utilisée pour d'autre type de recherches, par exemple, l'implantation d'un système de navigation complet incluant la reconstruction de scène mais aussi un système de planification autonome. Les outils d'évaluation ont été conçus pour être modulaire, ils permettent facilement l'ajout de nouvelles sources de données ou de nouvelles cartes de traitements.

Chapitre 3

Optimisation logicielle sur architectures low-cost : Application à l'EKF-SLAM

Sommaire

3.1 Introduction

Aujourd'hui, le SLAM (Simultaneous Localization And Mapping) est perçu comme un ensemble d'algorithme dont l'objectif est de localiser un robot mobile tout en modélisant son environnement. Une autre approche du SLAM est de considérer les algorithmes comme une partie d'un système complet incluant aussi des calculateurs, des capteurs et une plateforme expérimentale.

Un robot mobile est équipé de plusieurs capteurs, qui lui permettent d'obtenir des informations sur son propre comportement ainsi que sur l'environnement qui l'entoure. On distingue les capteurs proprioceptifs renseignant les déplacements du robot et les capteurs extéroceptifs fournissant des informations sur l'environnement extérieur.
Classiquement, les capteurs utilisés sont de bonnes qualités et souvent très onéreux. Par exemple, de nombreux algorithmes utilisent des données issues d'un télémètre laser à balayage, voir un télémètre multi-nappe. Ce type de capteurs est, à ce jour, incompatible avec des systèmes bas-coût qui pourraient cependant tirer profit des résultats d'un algorithme SLAM.
L'utilisation de capteurs bas coût s'est répandue récemment. Ces capteurs permettent d'envisager de nouvelles utilisations pour les algorithmes SLAM. Par exemple, les algorithmes utilisant une simple caméra intégrée dans un téléphone portable. En particulier PTAM [48] a été conçu pour développer les aspects de réalité augmentée des jeux vidéo embarqués sur un téléphone.
Ces capteurs sont utilisés par des systèmes adaptés, disposant majoritairement de peu de ressources de calcul. Cependant, aujourd'hui, les systèmes embarqués possèdent des fonctionnalités qui permettent des implantations très efficaces. Par exemple, de nombreux téléphones portables sont équipés d'un Cortex A8 ayant la spécificité d'intégrer une unité de calcul vectorielle. Une implantation optimisée, utilisant l'intégralité des fonctionnalités d'un système, est nécessaire pour obtenir des performances satisfaisantes, en particulier pour des systèmes embarqués disposant de peu de ressources.

L'objectif de ce premier travail est de démontrer la possibilité de concevoir un système embarqué réalisant du SLAM en utilisant des composants COTS (Components Off The Shelf) et des calculateurs bas coût. Pour cela, nous définirons plus précisément la problématique algorithmique du SLAM. Ensuite, un algorithme (EKF-SLAM) est étudié pour permettre une première approche de la reconstruction de scènes dans un environnement d'intérieur. Enfin, après une étude approfondie de l'algorithme, modifications seront proposées, permettant une implantation logicielle efficace sur des architectures embarquées disposant de peu de ressource de calculs mais surtout de spécificités permettant une implantation optimisée et efficace.

3.2 Définition du problème

3.2.1 Les états à observer

Un robot mobile utilise un algorithme de SLAM (Simultaneous Localization And Mapping) pour cartographier un environnement tout en se localisant dans la carte reconstruite. Le SLAM est décomposable en deux parties dépendantes très fortement l'une de l'autre :

- La localisation du robot mobile.
- La cartographie de l'environnement.

Ces deux parties sont entièrement corrélées, il est impossible de se localiser sans carte de l'environnement mais il est aussi impossible de cartographier un environnement sans connaitre sa position.

3.2.2 Déroulement global d'un algorithme SLAM

Ces algorithmes cartographient l'environnement en utilisant des primitives de natures diverses : ponctuelles, linéaires ou encore surfaciques. Cependant, un algorithme SLAM est généralement décomposable en plusieurs étapes (Fig. 3.1) :

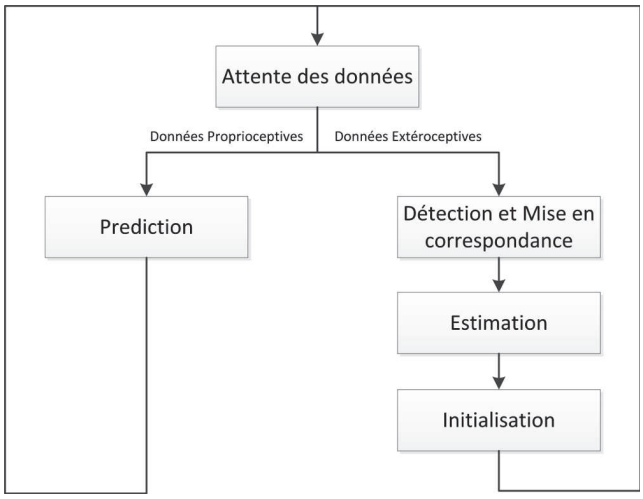


Figure 3.1: *Algorithme SLAM Générique*

- La prédiction : elle prédit l'état du mobile à partir des données proprioceptives.
- La détection et la mise en correspondance des amers : elle détecte les amers qui servent de points de repère pour se localiser. Une fois détectés, elle met en correspondance les amers observés et les amers déjà cartographiés.
- L'estimation : elle utilise les amers observés pour améliorer la localisation du mobile mais aussi celle de la carte.
- L'initialisation : l'algorithme ajoute de nouveaux amers pour agrandir la carte de l'environnement.

3.2.3 Choix de l'algorithme

Le SLAM a été très étudié depuis une quinzaine d'année. Les premiers algorithmes définis sont basés sur la théorie probabiliste, en particulier l'EKF-SLAM. Cet algorithme a été évalué de nombreuses fois, de plus, il a été adapté à l'utilisation de nombreux capteurs.
L'implantation la plus connue de l'EKF-SLAM est le MonoSLAM de Davison [1]. Il a proposé un algorithme utilisant une simple webcam, qui cartographie un environnement relativement restreint en taille mais en temps réel sur un ordinateur de bureau.

SLAM par Graph Récemment de nombreux algorithmes de SLAM basés sur l'optimisation de graphes sont apparus. En particulier, la première implantation temps réel sur un système embarqué a été réalisée par Klein et Murray [48] sur un iphone. Leur algorithme ne peut cependant cartographier qu'une petite zone de l'espace et se destine a être utilisé par des systèmes de réalité augmentée. Le choix de notre algorithme a été, en partie, guidé par Strasdat *et al.* [51] qui explique que le seul domaine où les algorithmes basés sur un filtre reste supérieur aux approches par graphs est le cas où les ressources processeurs sont très limitées. De plus, ils ont montré que les performances en terme de qualité de localisation pouvait être similaire entre les deux méthodes. Nous avons donc choisi pour notre premier système d'utiliser un algorithme probabiliste très utilisé : l'EKF-SLAM.

3.2.4 Définitions des variables

L'EKF-SLAM utilise un filtre de Kalman [9] pour localiser le robot mobile tout en cartographiant la scène. Dans le repère R_{glob}, les coordonnées du mobile sont $\mathbf{X}_k = (x,\ y,\ \theta)$ avec x , y les coordonnées du robot dans le plan et θ son orientation. Chaque amer est répresenté par sa position 3D dans l'espace, soit $\mathbf{a}_i = [x_{ai},\ y_{ai},\ z_{ai}]$.

A l'instant k, le SLAM estime la position du mobile ($\mathbf{X}_k$) ainsi que celle des amers ($\mathbf{a}$) en fonction des déplacements du mobile ($\mathbf{u}_k$) et des observations ($\mathbf{z}_k$). La problématique probabiliste s'écrit sous la forme :

$$p(\mathbf{x}_k, \mathbf{a}|\mathbf{z}_k, \mathbf{u}_k) \tag{3.1}$$

Si l'environnement contient M amers, on définit le vecteur d'état $\mathbf{x}_k$ suivant :

$$\mathbf{x}_k \quad = \quad (\mathbf{X}_k, \mathbf{a}_1, ..., \mathbf{a}_M)^T \tag{3.2}$$

On associe à ce vecteur, la matrice de covariance $\mathbf{P}_k$:

$$\mathbf{P}_k = \begin{bmatrix} P_{xx} & P_{xy} & P_{x\theta} & P_{xx_{a_1}} & .. & P_{xz_{a_M}} \\ P_{yx} & P_{yy} & P_{y\theta} & P_{yx_{a_1}} & .. & P_{yz_{a_M}} \\ P_{\theta x} & P_{\theta y} & P_{\theta\theta} & P_{\theta x_{a_1}} & .. & P_{\theta z_{a_M}} \\ P_{x_{a_1}x} & P_{x_{a_1}y} & P_{x_{a_1}\theta} & P_{x_{a_1}x_{a_1}} & .. & P_{x_{a_1}z_{a_M}} \\ .. & .. & .. & .. & .. & .. \\ P_{z_{a_M}x} & P_{z_{a_M}y} & P_{z_{a_M}\theta} & P_{z_{a_M}x_{a_1}} & .. & P_{z_{a_M}z_{a_M}} \end{bmatrix} \tag{3.3}$$

$$= \begin{bmatrix} \mathbf{P}_{RR} & \mathbf{P}_{Ra_1} & .. & \mathbf{P}_{Ra_M} \\ \mathbf{P}_{a_1R} & \mathbf{P}_{a_1a_1} & .. & \mathbf{P}_{a_1a_M} \\ .. & .. & .. & .. \\ \mathbf{P}_{a_MR} & \mathbf{P}_{a_Ma_1} & .. & \mathbf{P}_{a_Ma_M} \end{bmatrix} \tag{3.4}$$

Cette matrice de covariance symbolise l'ensemble des dépendances entre les variables. L'estimation de la position de chaque amer est corrélée, il est impossible d'estimer la position d'un amer indépendamment. Smith et Cheeseman [91] et Durrant-Whyte [92] ont démontré qu'il était nécessaire d'estimer la localisation de l'ensemble des amers dans un seul vecteur d'état, dans le cas contraire la carte peut devenir inconsistante.

3.2.5 Etape de prédiction

L'étape de prédiction utilise les données proprioceptives issues des odomètres pour prédire la position du robot. Les équations du filtre de Kalman pour la phase de prédiction sont les suivantes :

$$\begin{aligned} \mathbf{x}_{k|k-1} &= \mathbf{f}(\mathbf{x}_{k-1|k-1}, \mathbf{u}_k) + \mathbf{v}_k \\ \mathbf{P}_{k|k-1} &= \mathbf{F}_k \mathbf{P}_{k-1|k-1} \mathbf{F}_k^T + \mathbf{Q}_k \end{aligned} \tag{3.5}$$

Avec les données odométriques $\mathbf{u}_k$ supposées constantes entre les instants $k-1$ et k, le bruit d'état $\mathbf{v}_k$ définit blanc, gaussien et de matrice de covariance $\mathbf{Q_k}$.
$\mathbf{x}_{k-1|k-1}$ représente le vecteur d'état à l'instant $k-1$, $\mathbf{x}_{k|k-1}$ représente le vecteur d'état après avoir intégré les données odométriques et $\mathbf{x}_{k|k}$ représentera le vecteur après l'étape d'estimation.

A l'instant k, le modèle de déplacement $\mathbf{f}(\mathbf{x}_{k-1|k-1}, \mathbf{u}_k)$ est défini par :

$$\mathbf{x}_{k|k-1} = \mathbf{f}(\mathbf{x}_{k-1|k-1}, \delta s, \delta\theta) = \begin{pmatrix} x_{k-1|k-1} + \delta s_k \cos\left(\theta_{k-1|k-1} + \frac{\delta\theta_k}{2}\right) \\ y_{k-1|k-1} + \delta s_k \sin\left(\theta_{k-1|k-1} + \frac{\delta\theta_k}{2}\right) \\ \theta_{k-1|k-1} + \delta\theta_k \\ x_{a_1,k-1} \\ y_{a_1,k-1} \\ z_{a_1,k-1} \\ .. \\ .. \\ x_{a_N,k-1} \\ y_{a_N,k-1} \\ z_{a_N,k-1} \end{pmatrix} \tag{3.6}$$

avec :

– δs_k : la distance parcourue par le véhicule entre l'instant $k-1$ et k définie dans la section 2.3.2.

– $\delta\theta_k$: la variation de cap entre l'instant $k-1$ et k définie dans la section 2.3.2.

On remarque que le modèle de déplacement ne fait évoluer que la position du robot mobile, les coordonnées des amers n'évoluent pas durant cette étape. La matrice $\mathbf{F}_k$ est définie comme la jacobienne de $\mathbf{f}$ tel que :

$$\mathbf{F}_k = \frac{\partial \mathbf{f}}{\partial \mathbf{x}}|_{\mathbf{x_{k-1|k-1}}} \tag{3.7}$$

Dans le cas où il n'y a aucun amer dans la carte, on obtient la matrice suivante :

$$\mathbf{F}_{robot,k} = \begin{bmatrix} 1 & 0 & \delta s_k \cos(\theta_{k-1|k-1} + \frac{\delta\theta_k}{2}) \\ 0 & 1 & -\delta s_k \sin(\theta_{k-1|k-1} + \frac{\delta\theta_k}{2}) \\ 0 & 0 & 1 \end{bmatrix} \tag{3.8}$$

Avec M amers, on obtient la matrice de dimension $(3M+3)(3M+3)$ suivante :

$$\mathbf{F}_k = \left[\begin{array}{c|c} \mathbf{F}_{robot,k} & 0 \\ \hline 0 & Id(3M, 3M) \end{array}\right] \tag{3.9}$$

La matrice de bruit $\mathbf{Q}_k$ est définie dans [93] par :

$$\mathbf{Q}_k = \begin{bmatrix} k_{ss}\delta s_k|\cos(\theta_k)| & 0 & 0 \\ 0 & k_{ss}\delta s_k|\sin(\theta_k)| & 0 \\ 0 & 0 & k_{s\theta}|\delta s_k| + k_{\theta\theta}|\delta\theta_k| \end{bmatrix} \tag{3.10}$$

avec :
- k_{ss} le coefficient de dérive odométrique sur s causée par δs.
- $k_{s\theta}$ le coefficient de dérive odométrique sur θ causée par $\delta\theta_k$.
- $k_{\theta\theta}$ le coefficient de dérive odométrique sur θ causée par $\delta\theta_k$.

3.2.6 Etape d'estimation

L'étape d'estimation utilise les données issues de la caméra pour améliorer la position issue de l'étape de prédiction.
Elle utilise la prédiction de localisation des amers sur l'image. Pour calculer cette position prédite, on calcule la position de l'amer dans le repère caméra puis on utilise le modèle Pinhole pour projeter l'amer i sur l'image :

$$\mathbf{h}_{i,k}(\mathbf{x}_{k|k-1}) = \begin{pmatrix} u_i \\ v_i \end{pmatrix} = \begin{pmatrix} c_u + fk_u\frac{x_{ai,cam}}{z_{ai,cam}} \\ c_v + fk_v\frac{y_{ai,cam}}{z_{ai,cam}} \end{pmatrix} \tag{3.11}$$

avec :
- $(x_{ai,cam}, y_{ai,cam}, z_{ai,cam})$ la position de l'amer i dans le repère caméra.
- (u_i, v_i) la position de l'amer i sur l'image.
- (c_u, c_v) les coordonnées de la projection du centre optique de la caméra sur le plan image.
- f la distance focale de la caméra.
- (k_u, k_v) le nombre de pixel par unité de mesure.

Si l'amer n'est pas présent sur l'image, alors il n'est pas pris en compte lors de cette étape. Durant l'étape d'estimation, l'algorithme prédit la position de N amers avec N$\leqslant$M (le nombre d'amer total). On forme le vecteur de prédiction à l'aide de l'ensemble des projections : $\mathbf{h}_k(\mathbf{x}_{k|k-1}) = \begin{pmatrix} \mathbf{h}_{0,k} \\ .. \\ \mathbf{h}_{N-1,k} \end{pmatrix}$.

Après avoir prédit la position de chaque amer, son observation est calculé à l'aide de l'étape de mise en correspondance (définie section 3.3.2), elle correspond à sa position réelle sur l'image, $\hat{\mathbf{z}}_{i,k} = (u_{i,obs}, v_{i,obs})$. On forme ainsi le vecteur d'observations qui correspond à la position de chaque amer prédit :

$$\hat{\mathbf{z}}_k = \begin{pmatrix} \hat{\mathbf{z}}_{0,k} \\ ... \\ \hat{\mathbf{z}}_{N-1,k} \end{pmatrix} \tag{3.12}$$

L'innovation $\mathbf{Y}_k$, correspondant à la différence entre la valeur observée et la valeur prédite, est définie par :

$$\mathbf{Y}_k = \hat{\mathbf{z}}_k - \mathbf{h}_k(\mathbf{x}_{k|k-1}) \tag{3.13}$$

On lui associe sa matrice de covariance $\mathbf{S}_k$:

$$\mathbf{S}_k = \mathbf{H}_k \mathbf{P}_{k|k-1} \mathbf{H}_k^T + \mathbf{R}_k \tag{3.14}$$

Avec $\mathbf{H}_k$ la jacobienne de $\mathbf{h}_k$ et $\mathbf{R}_k$ la matrice de covariance du bruit. Cette matrice $\mathbf{R}_k$, définie dans le repère caméra, est calibrée expérimentalement à une variance de 1 pixel ($2\sigma = 2pixels$).

Enfin, les équations d'estimation du filtre de Kalman sont définies comme suit :

$$\begin{aligned} \mathbf{K}_k &= \mathbf{P}_{k|k-1}\mathbf{H}_k\mathbf{S}_k^{-1} \\ \mathbf{x}_{k|k} &= \mathbf{x}_{k|k-1} + \mathbf{K}_k\mathbf{Y}_k \\ \mathbf{P}_{k|k} &= (\mathbf{I} - \mathbf{K}_k\mathbf{H}_k)\mathbf{P}_{k|k-1} \end{aligned} \tag{3.15}$$

avec $\mathbf{K}_k$ le gain de Kalman.

L'étape d'estimation du filtre met à jour la postion du robot, des amers et de leurs incertitudes associées grâce aux observations réalisées. Les observations sont basées sur la carte, que l'algorithme construit au fur et à mesure du déplacement, et sur la localisation des amers réalisée grâce à la caméra. Il est important de remarquer que plus la carte est grande, plus la dimension du vecteur d'état et de sa matrice de covariance est grande. Le temps de calcul de l'étape d'estimation dépend fortement du nombre d'amers dans l'environnement. En effet, la complexité algorithmique de la multiplication matricielle est importante.

3.3 Problèmatiques associées au SLAM

L'algorithme décrit précédemment ne considère pas toutes les problématiques existantes. En effet, plusieurs aspects n'ont pas été abordés :

– la détection des amers.
– l'appariement des amers.
– l'initialisation des amers.

Chacun de ces points est crucial pour le SLAM. En effet, sans détection d'amers, il est impossible de cartographier un environnement. De même sans initialisation ou appariement, l'algorithme ne pourra pas fonctionner car il ne localisera ou relocalisera aucun point de repère.

3.3.1 Détection des amers

Le robot se localise dans une carte, constituée d'un ensemble d'amers. Il les localisent puis les utilisent pour corriger sa position. Il est nécessaire de détecter de manière fiable et robuste ces amers pour se relocaliser correctement dans la carte reconstruite. Plusieurs algorithmes de détection de points d'intérêts existent. Les plus connus sont Harris [83], Shi et Tomasi [84], FAST [85], SIFT [20], SURF [81]. Schmidt *et al.* [86] ont comparés l'ensemble des détecteurs de points d'intérêt dans le contexte d'un algorithme de SLAM. Cette étude a montré que SURF donnait les meilleurs résultats de détection et donc de localisation. Cependant son temps de traitement est incompatible avec une implantation embarquée temps réel. Après avoir évalué les différents détecteurs dans notre contexte [94], nous avons choisi le détecteur FAST qui permet une détection rapide et fiable des amers (voir Section. 2.3.3.3).

3.3.2 Appariement des amers

Après avoir détecté un amer, il faut pouvoir le redétecter durant le trajet du robot. Pour cela, on lui associe un descripteur qui est l'équivalent d'une carte d'identité pour l'amer.

La position de l'amer sur l'image peut-être prédite, lors de l'acquisition de l'image, le mobile connait sa position grâce à l'étape de prédiction de plus il connait la position de l'amer dans le repère global. Il faut donc projeter la position de l'amer sur l'image en utilisant le modèle Pinhole (Eq. 2.20). Cette projection est associée à une incertitude qui prend en compte l'incertitude de localisation de l'amer et celle du robot. Enfin, on choisit le pixel correspondant à l'amer à l'intérieur de l'ellipse d'incertitude en calculant la distance ZMSSD pondérée par le poids de la projection pour chaque pixel $p : (p_x, p_y)$:

$$\begin{aligned} N_p &= w(p_x, p_y) \\ &\sum_{i,j}((d(i,j) - m_d) - (im(p_x + i - \frac{des}{2}, p_y + j - \frac{des}{2}) - m_i))^2 \end{aligned} \tag{3.16}$$

avec :

- $w(p_x, p_y)$ le poids défini par la projection.
- $i \in [0; des - 1]$ et $j \in [0; des - 1]$ avec des la taille du descripteur.
- d le descripteur.
- m_d et m_i respectivement la moyenne des pixels du descripteur et la moyenne des pixels de l'imagette de test.
- im l'imagette.

On sélectionne ensuite l'observation p_{obs} comme étant le pixel candidat ayant le plus petit N_p dans l'ellipse d'incertitude : $p_{obs} = (p \in \tau | N_p = min(N_{p_j}), \forall p_j \in \tau)$.

3.3.3 Recherche active

L'utilisation de la projection de l'incertitude de l'amer sur l'image permet de réduire considérablement la zone de recherche de correspondance. Des contributions ont été proposées, en particulier par Handa *et al.* [95], pour réduire encore les zones de recherches.
La mise en correspondance est réalisée pour un amer puis les zones de recherches des autres amers sont mises à jour en utilisant l'observation réalisée. Un système d'arbre permet de gérer les problèmes de mauvaises mises en correspondance. Ce système n'a pas été mis en oeuvre dans notre implantation, car nous considérons un robot d'exploration lent et donc les zones de recherches ne devraient pas être très vastes. De plus, nous optimiserons le calcul du ZMSSD pour réduire considérablement le temps de calcul. Cependant, cette contribution pourrait être incluse dans notre système final.

3.3.4 Initialisation d'un amer

L'initialisation des amers est une étape cruciale des algorithmes de SLAM, en particulier pour une caméra monoculaire. En effet, ce type de caméra ne permet pas de connaitre directement le paramètre de profondeur d'une observation : elle ne renseigne que sur l'angle d'observation. Cependant, la détermination de la profondeur est possible en observant le même amer à deux positions différentes. La précision de la profondeur dépend de l'angle induit par les deux observations.

Deux méthodes réalisent cette initialisation :

- La première consiste à initialiser un amer sans aucun délai [96], en utilisant une paramétrisation par inverse de profondeur. L'amer est ajouté directement dans le vecteur d'état lors de sa première observation puis il est utilisé lors pour l'étape d'estimation. Ce type d'initialisation est très souvent utilisé par des systèmes de localisation basés sur une simple caméra monoculaire (sans odométres) et n'ayant aucune connaissance à priori de l'environnement. En effet, elle permet d'utiliser l'ensemble des observations pour améliorer la localisation, cependant, l'ajout des amers dans le système augmentera le temps de traitement. De plus, ce type de

méthode nécessite une paramétrisation améliorée utilisant 6 variables, la taille du système croit donc de manière plus importante.

– La seconde initialise les amers avec un délai [65], elle définit la composante de profondeur à l'aide de plusieurs observations successives puis l'insère dans la carte.

Nous utiliserons la méthode avec délai car nous disposons de données odométriques permettant de se localiser sans vision durant un court laps de temps. De plus, Munguia et Grau [97] ont montré que la méthode d'initialisation sans délai a approximativement les mêmes résultats que la méthode avec délai. Enfin, notre optimisation nous permettra de réaliser une parrallèlisation des étapes d'initialisation et d'estimation des amers, de ce fait un processeur sera dédié à l'initialisation de nouveaux amers et un second à l'estimation du système. L'utilisation d'une initialisation sans délai ne semble donc pas indispensable.

3.3.4.1 Méthode de Davison

Pour définir les coordonnées initiales d'un amer, Davison *et al.* [65] initialisent une demi-droite composée de 100 particules, ayant pour origine le centre de la caméra et suivant la direction de l'observation. Chaque particule représente une localisation potentielle de l'amer en 3 dimensions ($\mathbf{a_{i,p}} = [a_{i,p,x}, a_{i,p,y}, a_{i,p,z}]$). En environnement intérieur, on limite la longueur de la demi-droite à 10m.
A chaque nouvelle image, l'amer est observé et le poids de chaque particule est calculé en fonction de l'observation et de l'ellipse d'incertitude issue de la projection. Cette incertitude est calculée à l'aide de la jacobienne de la fonction d'observation. Plus précisément, on utilise l'innovation $\mathbf{S}_k$ issu des équations d'estimation du filtre de Kalman.

Lorsque le rapport de l'incertitude de la profondeur (σ_{depth}) sur la profondeur ($depth$) est suffisamment réduit ($< \epsilon_{depth}$), on ajoute l'amer dans le vecteur d'état et sa matrice de covariance. On utilise le test suivant $\frac{\sigma_{depth}}{depth} < \epsilon_{depth}$. Davison *et al.* [65] définissent le seuil ϵ_{depth} égale à 0.3.

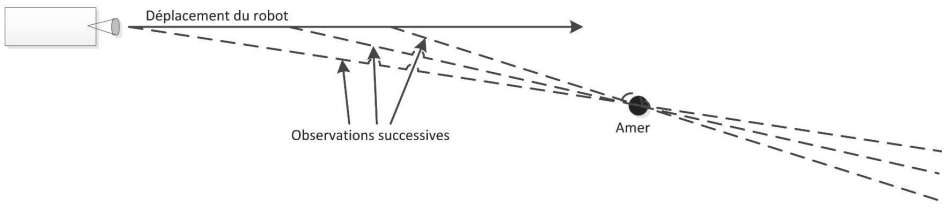


Figure 3.2: *Initialisation des amers par la méthode de Davison.*

3.4 Validation de l'EKF-SLAM

Un simulateur a été développé pour évaluer le comportement des différents algorithmes. Il permet de valider les résultats de localisation car on dispose d'une position

de référence exacte pour le robot ainsi que pour les amers (ce qui n'est pas le cas lors de la validation expérimentale). Plusieurs environnements ont été modélisés pour valider l'algorithme EKF-SLAM dans différentes situations (Sec. 2.4.1.3). De plus, plusieurs critères d'évaluation ont été définis pour évaluer la qualité et la consistance de la localisation du mobile mais aussi de celle des amers (Sec. 2.4.3).

Pour ces simulations, nous avons fixé un bruit blanc gaussien pour le rayon de la roue avec $3\sigma = 0.2mm$, pour l'entraxe $3\sigma = 5mm$ et pour les observations $3\sigma = 2pixels$. Les figures 3.3 représentent les différents résultats issus de 25 simulations dans l'environnement 1, les résultats pour l'environnement 2 et 3 se trouvent en annexes. La première constatation est l'influence du nombre d'amers sur la qualité de la localisation. Il est immédiat que plus le nombre d'amers est important, plus la qualité de la localisation est bonne. De plus, pour les environnement 2 et 3, la localisation est stabilisée : l'erreur euclidienne ne croit pas au cours du temps.

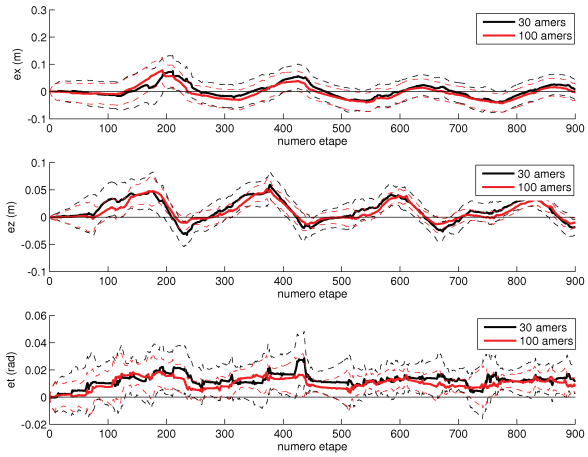
Malgré une bonne qualité de localisation, on remarque la mauvaise consistance du filtre : la valeur 0 n'est pas toujours incluse dans le couloir de consistance. Quelque soit le nombre d'amers ou l'environnement, le filtre est inconsistant pour la localisation des amers et la position. Ces problèmes d'inconsistances sont présents quelque soit la paramétrisation du filtre.

3.5 Récapitulatif de l'algorithme

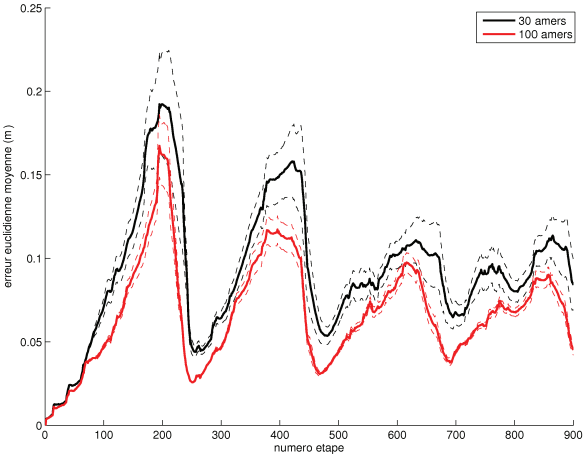
L'algorithme (3.1) résume les différentes étapes nécessaires à la reconstruction de la carte et à la localisation du robot. L'algorithme se déroule en deux étapes : la prédiction (L. 4) et la mise à jour (L. 10). L'étape de prédiction intègre les données odométriques (L. 7) pour définir la position du robot dans l'environnement (L. 8). Cette position est associée à une incertitude car les données proprioceptives sont imprécises (L. 9). Ensuite, l'étape de mise à jour corrige cette position à l'aide de plusieurs sous étapes : la mise en correspondance, l'estimation et l'initialisation. La première étape (mise en correspondance) consiste à rechercher les points de localisation (L. 12) puis à les utiliser pour relocaliser le robot lors de l'étape d'estimation (L. 22). De plus, cette étape permet d'améliorer la carte de l'environnement (L. 25, L. 26). L'étape d'initialisation permet d'ajouter de nouveau points de localisation (L. 27) à la carte (L. 34).

3.6 Validation expérimentale

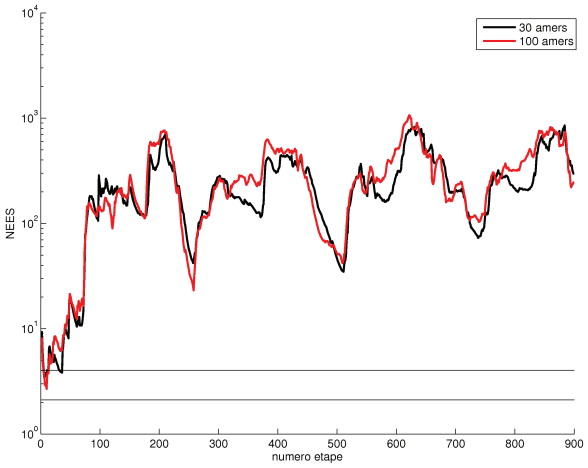
Afin d'évaluer les performances expérimentales de l'EKF-SLAM, on utilise le jeu de données collecté par notre plateforme. Ce jeu de données correspond à 3 tours approximativement similaire dans une salle de travaux pratiques. Nous avons vu que l'EKF-



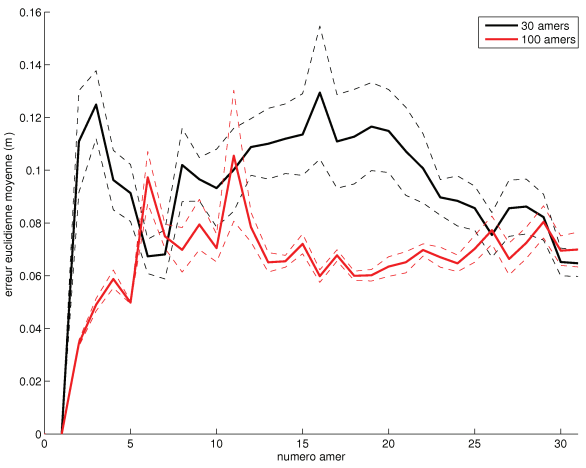
(a) *Couloir d'incertitude sur la position x, y, θ*



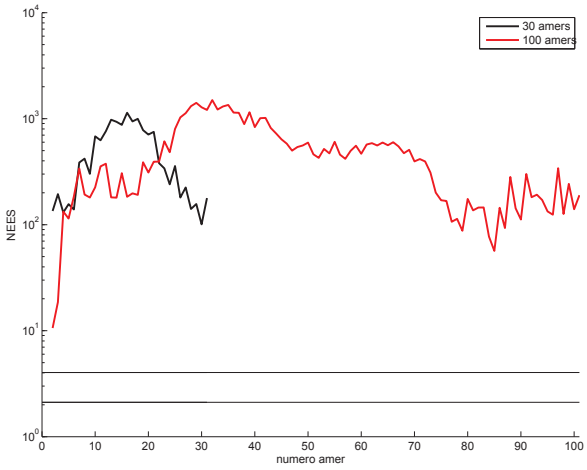
(b) *Erreur euclidienne de la position x, y, θ*



(c) *NEES de la position x, y, θ*



(d) *Erreur euclidienne de la position des amers*



(e) *NEES pour la position des amers*

Figure 3.3: *Résultats de l'EKF-SLAM pour l'environnement 1*

Algorithm 3.1 EKF-SLAM

$\chi \leftarrow \emptyset$ ▷ Liste des amers en cours d'initialisation
Initialisation de la position du robot
while 1 **do**
  DATA ← Acquisition des données capteurs
  if DATA $= (\varphi_l, \varphi_r)$ **then** ▷ Données odométriques
    PREDICTION :
    $(\delta s\ ,\ \delta\theta) \leftarrow \mathbf{g}(\varphi_l, \varphi_r)$ (see Eq. (2.12))
    $\mathbf{x}_{k|k-1} \leftarrow \mathbf{f}(\mathbf{x}_{k-1|k-1}, \delta s, \delta\theta)$ (see Eq. (3.6))
    $\mathbf{P}_{k|k-1} \leftarrow \frac{\partial \mathbf{f}}{\partial \mathbf{x}} \mathbf{P}_{k-1|k-1} \frac{\partial \mathbf{f}}{\partial \mathbf{x}}^T + \mathbf{Q}_k$ (see Eq. (3.5))
  else if DATA = Camera **then**
    Détecteur FAST
    MISE EN CORRESPONDANCE :
    for Each Amer $\mathbf{N_i} \in \mathbf{x}_{k|k-1}$ **do**
      $u_i, v_i, \tau_i \leftarrow pinhole(\mathbf{x}_{k|k-1}, \mathbf{N_i})$ (see Eq. (2.19))
      if $(u_i, v_i) \in$ Image **then**
        $\hat{\mathbf{z}}_k \leftarrow ZMSSD(\tau_i, \mathbf{N_i})$ (see Eq. (2.22))
        $\mathbf{h}_k \leftarrow (u_i, v_i)$
        $\mathbf{Y}_k \leftarrow \hat{\mathbf{z}}_k - \mathbf{h}_k$
        $\mathbf{H}_k \leftarrow \frac{\partial \mathbf{h}_k}{\partial \mathbf{x}}\big|_{\mathbf{x}_{k|k-1}}$
      end if
    end for
    ESTIMATION :
    $\mathbf{S}_k \leftarrow \mathbf{H}_k \mathbf{P}_{k|k-1} \mathbf{H}_k^T + \mathbf{R}_k$ (see Eq. (3.14))
    $\mathbf{K}_k \leftarrow \mathbf{P}_{k|k-1} \mathbf{H}_k \mathbf{S}_k^{-1}$ (see Eq. (3.14))
    $\mathbf{x}_{k|k} \leftarrow \mathbf{x}_{k|k-1} + \mathbf{K}_k \mathbf{Y}_k$ (see Eq. (3.15))
    $\mathbf{P}_{k|k} \leftarrow (\mathbf{I} - \mathbf{K}_k \mathbf{H}_k) \mathbf{P}_{k|k-1}$ (see Eq. (3.15))
    INITIALISATION :
    for Each $\mathbf{L} \in \chi$ **do** ▷ **L** : Nouveaux amers potentiels
      $\mathbf{L}_{obs} \leftarrow ZMSSD(\mathbf{L})$ (see Eq. (2.22))
      Mise à jour du poids des particules
      Calcule de σ_{depth}, *depth*
      if $\frac{\sigma_{depth}}{depth} < \epsilon$ **then**
        Calcule de $\mathbf{L}$, $\mathbf{P_L}$
        $insérer(\mathbf{x_{k|k-1}}, \mathbf{L})$; $insérer(\mathbf{P_{k|k-1}}, \mathbf{P_L})$
        $retirer(\chi, \mathbf{L})$
      end if
    end for
    if Présence de zones blanches **then** ▷ see [8]
      ajouter(χ, Liste_amers_potientiels)
    end if
  end if
end while

SLAM n'était pas très adapté à des environnements très larges, nous nous concentrons donc sur un environnement de taille réduite.

Les figures 3.4 représente les résultats expérimentaux. La figure 3.4a représente les résultats du premier tour. On remarque que l'EKF-SLAM améliore la localisation par rapport à la localisation par intégration des données odométriques. En effet, le trajet passe correctement par les points de référence. Cependant, on remarque que l'incertitude de localisation n'évolue que très peu. En effet, les ellipses d'incertitude représentées en rouge restent très petite. L'EKF-SLAM est optimiste sur l'incertitude de localisation du robot.
Concernant la qualité de localisation, les mêmes conclusions peuvent être tirés pour le second tour. La figure 3.4b représente le trajet du second tour. La différence avec l'intégration odométrique est encore plus marqué, la trajectoire odométrique dévie au cours du temps. L'EKF-SLAM conserve la bonne trajectoire. Le troisième tour est représenté sur la figure 3.4c. On remarque que la fin du trajet souffre de plusieurs problèmes. La trajectoire se dévie assez fortement des points de référence. Ceci confirme le problème de consistence à long terme de notre algorithme.

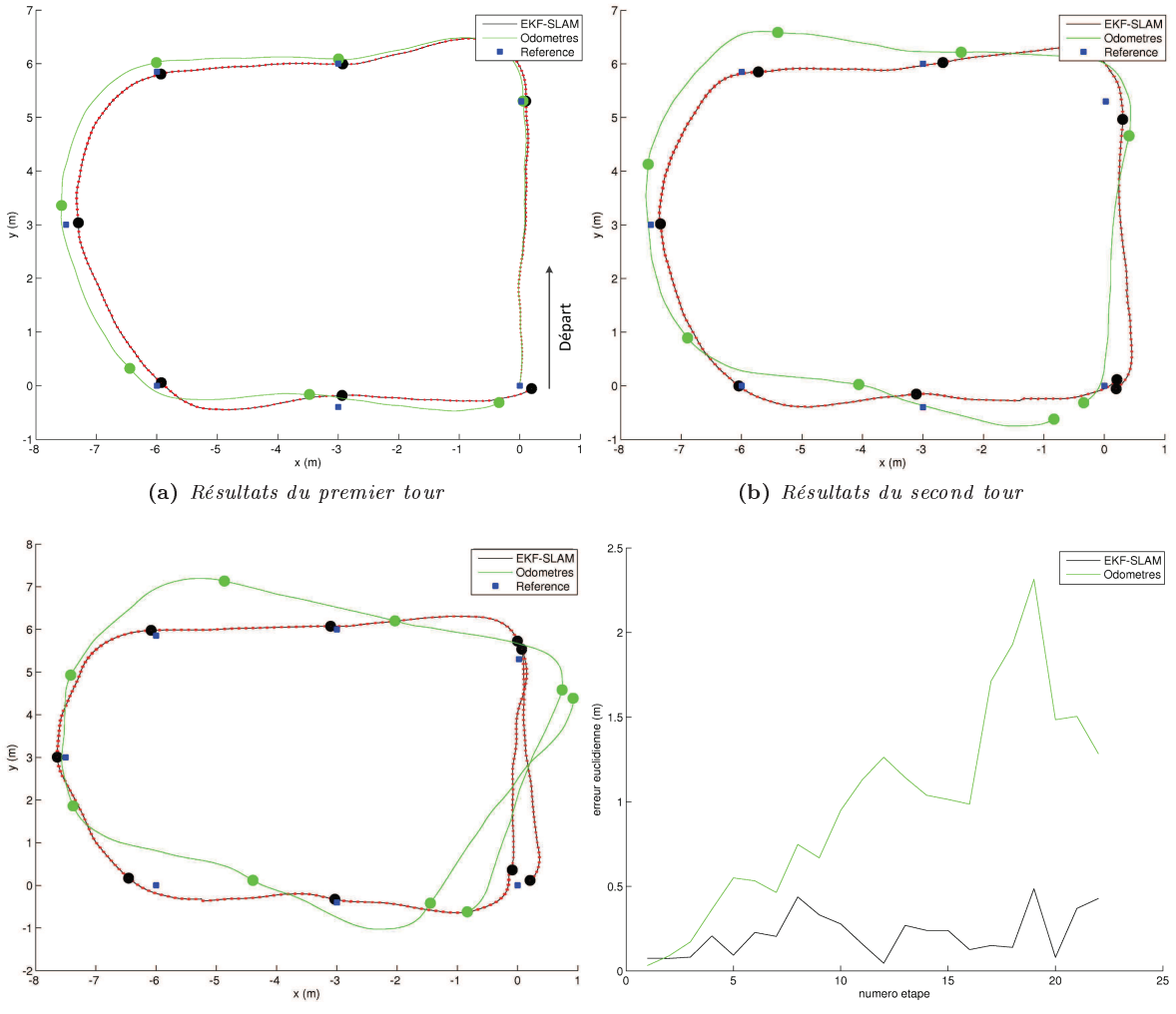
Nous avons évalué visuellement la différence entre l'intégration odométrique et l'EKF-SLAM. La figure 3.4d confirme les conclusions précédentes. L'intégration odométrique s'éloigne progressivement de la trajectoire de référence. L'erreur euclidienne entre la localisation et la référence augmente au cours du temps. Quant à lui, l'EKF-SLAM a une erreur bornée sur l'ensemble de la trajectoire. Les résultats obtiennent une erreur moyenne d'environ 0.25 cm. Cette erreur est faible et confirme les performances acceptables de notre algorithme.

Cette expérimentation nous confirme les bonnes performances de l'EKF-SLAM. Cependant, elle confirme aussi que l'EKF-SLAM n'est pas adapté à des expérimentations longues. Cependant, ce type de problème pourrait être diminué en utilisant par exemple une paramétrisation par inverse de profondeur ou encore un système de cartes locales. On choisit de conserver cet algorithme car notre objectif est de concevoir un système de SLAM en respectant le concept d'adéquation algorithmes architectures et non de définir un nouvel algorithme de SLAM.

3.7 Analyse temporelle de l'algorithme

3.7.1 Choix de l'architecture

L'implantation d'un algorithme dépend très fortement de l'architecture cible. Chaque architecture dispose de spécificités propres qui permettent une implantation efficace. Une architecture dédiée permet d'obtenir des résultats interressants en particulier lors de calculs fortement parrallèles. Botero *et al.* [8] ont proposé une architecture réalisée en co-design implantant un algorithme EKF-SLAM conjointement à une détection de



(a) *Résultats du premier tour*

(b) *Résultats du second tour*

(c) *Résultats du troisième tour*

(d) *Erreur euclidienne de l'EKF-SLAM pour les 22 points de référence*

Figure 3.4: *Résultats expérimentaux de l'EKF-SLAM*

point d'intérêt Harris, une paramétrisation par inverse de profondeur et une méthode de recherche active. L'implantation des équations du filtre de Kalman est réalisée de manière logicielle, la détection de points d'intérêt est réalisée de manière cablée car les traitements sont parrallélisables. De plus, l'implantation des équations de Kalman sur une architecture programmable a été réalisée par Kraft *et al.* [87], et leurs résultats semblent prometteur.

Contrairement à la conception d'un système dédié, on propose d'étudier l'implantation optimisée de notre algorithme sur des architectures grand public, multiprocesseurs et embarquables. Ces architectures sont aujourd'hui très largement distribuées en particulier dans les smartphones. Notre première architecture est conçue autour d'un OMAP3530 et la seconde autour d'un OMAP4430. Ces deux processeurs ont des spécificités qui permettent une implantation très optimisée telle que la présence d'unité de calcul vectorielle ou de DSP.

3.7.2 Méthodologie d'évaluation

La méthode d'évaluation nécessite l'identification des parties fonctionnelles de l'algorithme qui demandent un temps de calcul important.
Elle est composée de plusieurs étapes : nous analysons d'abord le temps d'exécution de chaque tâche ainsi que sa dépendance aux paramètres de l'algorithme. Les différentes tâches n'ont pas des temps d'exécution constants, il est donc nécessaire de décomposer l'algorithme en blocs fonctionnels (BF) ayant des temps de traitement constants. Ensuite, chaque bloc est évalué afin de déterminer précisément son temps de calcul ainsi que son nombre d'occurrences moyen lors d'une expérimentation. Les blocs fonctionnels nécessitant des ressources importantes sont ensuite optimisés pour réduire le temps de traitement global. Cette optimisation est réalisée en adéquation avec l'architecture sur laquelle est implanté l'algorithme. Elle nécessite parfois des modifications algorithmiques pour s'adapter au mieux aux caractéristiques de l'architecture.

Nous allons étudier chaque étape de l'algorithme et définir les dépendances aux différents paramètres. Certains paramètres pourront être fixés pour rendre constant ou borner le temps de traitement de l'étape.

3.7.3 Etape de prédiction

L'étape de prédiction met à jour la position du robot mobile ($\mathbf{x}_{k|k-1}$) en fonction des données proprioceptives acquises à partir des odomètres (φ_l, φ_r). Le temps de traitement de ce processus n'est pas constant. Il met à jour le vecteur 3D contenant la position du robot, cette étape est réalisée en temps constant. Cependant, la mise à jour de la matrice de covariance dépend du nombre d'amers. En effet, l'ensemble de la carte est corrélée à la localisation du robot.

3.7.4 Etape de mise à jour

Le temps de traitement global du processus de mise à jour n'est pas constant. Nous allons étudier le temps de traitement de chaque tâche composant cette étape ainsi que leurs dépendances :

Mise en correspondance

Cette étape calcule la position d'un amer sur l'image à l'aide d'une mesure de corrélation. Chaque amer, présent dans le vecteur d'état, est projeté sur l'image en utilisant le modèle Pinhole (L. 14). Puis, l'algorithme lui cherche une correspondance en utilisant la mesure de corrélation ZMSSD. Le temps de calcul des projections ne dépend que du nombre d'amers présents dans le vecteur d'état (L. 13). Cependant, le temps de recherche de correspondance ne peut être borné, il dépend du nombre de pixels candidats qui est défini par l'incertitude de localisation du robot et des amers.
Le temps de traitement de cette tâche dépend de plusieurs paramètres :

- Le nombre d'amers dans le vecteur d'état.
- Le nombre d'amers visibles dans l'image.
- La taille du descripteur.
- L'incertitude de localisation du mobile et de chaque amer.

Le nombre d'amers n'est pas limité : plus le robot parcourt un trajet important, plus le nombre d'amers augmente. De plus, il n'est pas possible de borner facilement les incertitudes de localisation du robot ou des amers.

Estimation

La tâche d'estimation utilise les équations de Kalman pour mettre à jour à la fois la position du robot, la position des amers ainsi que l'incertitude sur chacune des variables. Le temps de traitement de la tâche d'estimation est très important car cette étape est composée de multiplications matricielles dont la taille dépend du nombre d'amers présents dans le vecteur d'état ainsi que du nombre d'amers observés.
Le temps de traitement de cette tâche dépend donc de deux paramètres :

- Le nombre d'amers dans le vecteur d'état.
- Le nombre d'amers observés.

Ces différents paramètres pourront être bornés en limitant le nombre d'observations à chaque étape ainsi que le nombre d'amers présents dans le vecteur d'état. La gestion de la carte d'amers est détaillée au paragraphe 3.7.6.

Initialisation

Pour chaque amer en cours d'initialisation, l'algorithme sélectionne une observation à l'aide de la position définie par la projection des particules et de la corrélation. La

probabilité de chaque particule est ensuite mise à jour et l'incertitude de localisation de l'amer est évaluée pour vérifier si elle est assez réduite pour l'insérer dans la carte de l'environnement.
Le temps de traitement de l'étape d'initialisation dépend de :

- Le nombre d'amers en cours d'initialisation.
- La taille du descripteur.
- L'incertitude de la localisation du mobile.

Comme pour l'étape de mise en correspondance ou d'estimation, le temps de calcul de cette étape n'est pas constant. Il dépend de l'incertitude de localisation du robot et des amers en cours d'initialisation.

3.7.5 Définition des seuils

Dans les sections précédentes, nous avons vu que le temps de calcul de chaque tâche de l'algorithme dépend de nombreuses variables. Pour implanter l'algorithme sur un système embarqué, il est important d'obtenir un temps de calcul constant, ou borné. Plusieurs paramètres peuvent être définis pour limiter les variations de temps de traitement. On définit :

- le nombre maximum d'amers dans le vecteur d'état, sa taille sera fixe. Par conséquent, aucune allocation dynamique de mémoire ne sera nécessaire, le vecteur ainsi que sa matrice de covariance seront déclarés au lancement de l'algorithme. Le temps de l'étape de prédiction sera constant.
- le nombre maximum d'amers observés. Le temps de calcul de l'étape d'estimation sera constant car elle est constituée de multiplications matricielles de taille fixe.
- le nombre maximum d'amers en cours d'initialisation. Malheureusement, le temps de calcul de la tâche d'initialisation ne sera pas constant car l'étape de mise en correspondance dépendra toujours de l'incertitude de localisation du robot.

3.7.6 Gestion de la carte de l'environnement

Le vecteur d'état comprend deux types d'amers, ceux dont l'initialisation a été réalisée récemment et ceux qui ont déjà été suivis depuis plusieurs étapes. Pour conserver la taille du vecteur d'état constante, il est obligatoire de supprimer certains amers de la carte lors de l'insertion de nouveaux.
Auat Cheein et Carelli [98] proposent une méthode efficace pour sélectionner les amers utilisés par la tâche d'estimation. Ils basent leur choix sur l'évaluation de l'influence d'une observation sur la convergence de la matrice de covariance du vecteur d'état. La méthode calcule l'observation correspondante à chaque amer présent dans le vecteur d'état puis le coefficient $(\mathbf{I} - \mathbf{K}_k\mathbf{H}_k)$ des équations du filtre de Kalman (Eq. 3.15). S'agissant d'une implantation embarquée, nous ne pourrons pas mettre en œuvre cette méthode exactement comme proposée par [98] en raison des contraintes de temps de

calcul.
Nous avons choisi d'ajouter des amers, sur la base de l'étape d'estimation précédente, en sélectionnant les observations qui ont eu la plus grande influence sur la convergence de la matrice de covariance du vecteur d'état. A l'instant k, nous sélectionnons en priorité les amers qui ont eu la plus petite valeur $(\mathbf{I} - \mathbf{K}_{k-1}\mathbf{H}_{k-1})$. De plus, nous ajoutons la possibilité pour l'algorithme de choisir en priorité les amers ayant eu une initialisation très récente pour améliorer leur localisation.

3.7.7 Définition des blocs fonctionnels

Toutes les tâches définies précédemment n'ont pas un temps de calcul constant, le temps dépend de l'expérimentation. En effet, l'incertitude du robot ou celle des amers ne peut pas être bornée facilement. On définit des blocs fonctionnels (BF) ayant un temps de calcul fixe pour pouvoir identifier les blocs nécessitants des ressources importantes. Le temps de calcul d'un bloc est constant, il ne dépend pas de l'expérimentation. Cependant, le nombre d'itérations de certains blocs (3,4,5,6,7,8 et 9) dépend de l'expérimentation. A partir de l'algorithme 3.1, on définit neuf blocs explicités dans la table 3.1 :

Blocs Fonctionnels (BF)	Description	Lignes
1. Prédiction	L'étape de prédiction	7 à 9
2. FAST	La détection de points d'intérêt	11
3. Projection	La projection d'un amer sur l'image	14
4. ZMSSD-M	Le calcul de la correlation entre deux imagettes lors de la mise en correspondance	16
5. $\mathbf{H}_i$	Le calcul de $\mathbf{H}_i$	19
6. Estimation	Les équations de l'étape d'estimation du filtre de Kalman	23 à 26
7. ZMSSD-I	Le calcul de la correlation entre deux imagettes lors de l'initialisation	29
8. Mise à jour des poids	La mise à jour des poids des particules lors de l'initialisation	30, 31
9. Ajout d'un amer	L'ajout d'un amer pour l'initialiser	39

Table 3.1: *Définition des blocs fonctionnels*

Chaque bloc fonctionnel a un temps de calcul constant mais peut être éxecuté plus d'une fois lors d'une étape de mise à jour ou de prédiction.

3.7.8 Temps d'execution global

Pour effectuer une première évaluation du temps de calcul de l'algorithme, ce dernier est implanté séquentiellement sur le processeur OMAP3530 cadencé à 500 MHz (aucune optimisation n'est réalisée). De plus, nous avons choisi d'utiliser le nombre maximum

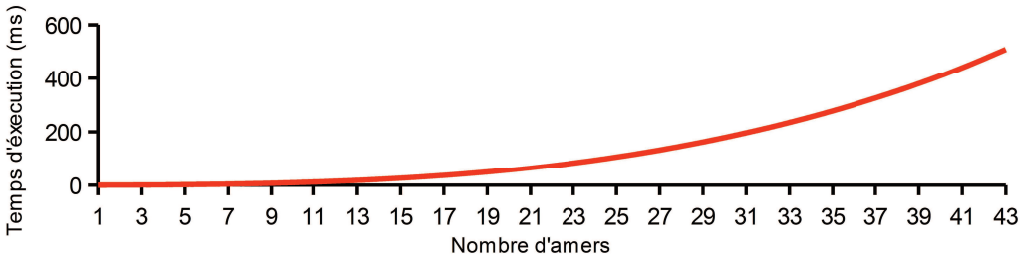


Figure 3.5: *Temps d'éxecution de la tâche d'estimation*

d'images car le suivi des amers sera facilité si le mouvement entre deux images est faible. On dispose donc de 33ms de traitement, car la caméra fourni des images à une fréquence de 30Hz. Le temps d'acquisition des données est constant. En effet :

• L'acquisition des données odométriques est réalisée en 0.7ms (ce temps de traitement est due à la communication I2C avec le processeur Atmega168, voir Architecture 2.2.2).

• Chaque acquisition d'image prend 1.8ms (en raison du transfert des données par communication USB).

L'étape de prédiction ne nécessite pas un temps de calcul important, seulement 0.093ms par itération. Comme pour la tâche de mise en correspondance, la tâche d'estimation ne peut pas avoir un temps de traitement constant. Le temps de traitement de l'estimation dépend du nombre total d'amers ainsi que du nombre d'observation. La figure 3.5 représente le temps de traitement de la tâche d'estimation en fonction du nombre d'amers présents dans le vecteur d'état, en considérant que chaque amer présent dans la carte est observé. Évidemment, il sera impossible de prendre en compte tous les amers observés : le temps de calcul sera beaucoup plus élevé que les 33ms disponible. Il faudra trouver un compromis entre le nombre d'amers et le temps de traitement afin de respecter les contraintes de temps réel.

3.7.9 Temps d'execution des blocs fonctionnels

Le temps de calcul des blocs fonctionnels a été évalué indépendamment de l'expérimentation. On étudie le temps de traitement de chaque bloc pour chaque architecture (Cortex A8 500Mhz, Cortex A9 1Ghz).

Cortex A8 Pour étudier les temps de traitement, plusieurs paramètres ont été fixés :

- La taille de l'image à 320×240 pixels.
- La taille du descripteur à 16×16 pixels.
- Le nombre maximum d'amers dans le vecteur d'état : 25.
- Le nombre maximum d'observation : 25.
- Le nombre maximum d'amers en cours d'initialisation : 20.

Blocs Fonctionnels (BF)	Temps d'execution par itération (μs)	Nombre moyen d'occurence par mise à jour	Temps moyen d'execution (μs)
2. FAST	3400	1	3400
3. Projection	9	19	180
4. ZMSSD-M	11.29	233	2630
5. $\mathbf{H}_i$	14.5	4.5	66
6. Estimation	88845	0.8	70568
7. ZMSSD-I	11.29	123	1388
8. Mise à jour poids	638	4.0	2586
9. Ajout d'une amer	103	0.18	18

Table 3.2: *Temps d'execution des blocs fonctionnels sur le processeur Cortex A8 (500 Mhz), image 320×240 pixels, 25 amers*

Tout d'abord, nous pouvons analyser le temps d'exécution des 8 blocs de l'étape de mise à jour précédemment définis. Le temps de traitement est calculé en utilisant le timer posix fourni par le système d'exploitation. Le tableau 3.2 résume, le temps de traitement par itération, la moyenne du nombre d'itérations et la moyenne du temps de traitement par étape de correction pour le Cortex A8 cadencé à 500Mhz.
Le temps de traitement moyen est d'environ 80.8ms, ce qui correspond à la somme de tous les temps de traitement : prédiction (BF1) et mise à jour (BF2 à BF9). Le temps de traitement de la tâche d'estimation (BF6) est d'environ 70.5ms, il représente environ 87% du temps de traitement global. Le détecteur FAST (BF2) prend 3.4ms quant à ZMSSD-M (BF4), il prend 2.63ms. Enfin, la tâche d'initialisation (BF7, BF8 et BF9) prend 3.9ms. Ces 6 BFs représentent 99,6% du temps de traitement global.

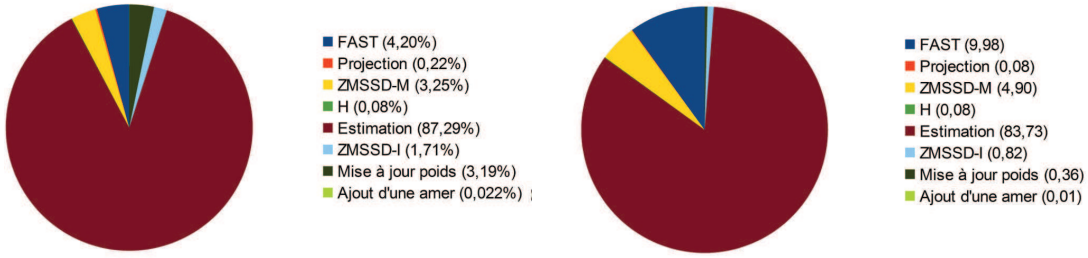
Cortex A9 Pour étudier les temps de traitement du Cortex A9, on choisit des paramètres plus importants car il dispose de plus de puissance de calcul. Les paramètres choisis sont :

- La taille de l'image à 640×480 pixels.
- La taille du descripteur à 16×16 pixels.
- Le nombre maximum d'amers dans le vecteur d'état : 50.
- Le nombre maximum d'observation : 50.
- Le nombre maximum d'amers en cours d'initialisation : 50.

La prédiction (BF1) s'exécute en 0.026 ms. Le tableau 3.3 résume, pour les BFs, le temps de traitement par itération, le nombre moyen d'occurrences et le temps de traitement moyen par processus de mise à jour. Le temps de traitement moyen par image est d'environ 65,90 ms. Le temps de traitement de la tâche d'estimation (BF6) est d'environ 55,1 ms. Il représente environ 83% du temps de traitement global. Le détecteur FAST (BF2) s'exécute en 6,5 ms. Le bloc ZMSSD-M (BF4) prend 3,23 ms par estimation. Enfin, les tâches d'initialisation (BF7 à BF9) prennent 0,8 ms. Ces

Blocs Fonctionnels (BF)	Temps d'execution par itération (μs)	Nombre moyen d'occurence par estimation	Temps moyen d'execution (μs)
2. FAST	6581	1	6581
3. Projection	1.37	40.9	56.03
4. ZMSSD-M	3.46	933.7	3230.60
5. $\mathbf{H}_i$	3.42	15.86	54.24
6. Estimation	61317	0.90	55185.30
7. ZMSSD-I	3.46	158.03	546.78
8. Mise à jour poids	47.13	5.07	238.94
9. Ajout d'une amer	32.18	0.26	8.36

Table 3.3: *Temps d'execution des blocs fonctionnels sur le processeur Cortex A9 (1 Ghz), image 640×480 pixels, 50 amers*



(a) *Répartition des temps d'éxecution pour le Cortex A8* **(b)** *Répartition des temps d'éxecution pour le Cortex A9*

Figure 3.6: *Comparaison de la répartition des temps d'éxécution*

blocs (2, 4, 6 et 7) représentent 98,6% du temps de traitement global.

Comparaison La figure 3.6 représente la répartition des temps d'exécution de chaque implantation sur les deux architectures et les paramétrisations correspondantes. La première constatation est que la tâche d'estimation consomme énormément de ressources sur les deux architectures. Deux tâches annexes représentent aussi une consommation importante : le calcul de la corrélation ZMSSD et la détection des points d'intérêts (FAST). La répartition des temps est très similaire sur les deux architectures. La seule différence majeure concerne la mise à jour des poids lors de l'étape d'initialisation des amers. En comparaison avec les temps de traitement du Cortex A8, à fréquence égale, l'architecture du Cortex A9 a des résultats significativement meilleurs. En effet le Cortex A8 est pénalisé par l'absence d'une unité de calcul des nombres flottants. Cette unité est présente pour le Cortex A9. Par conséquence, la mise à jour des poids est réalisée très rapidement sur le Cortex A9 grâce à l'unité de calcul flottante, contrairement au Cortex A8.
Pour définir une implantation adéquate, nous nous sommes concentrés sur une optimisation de ces BFs pour améliorer le temps de traitement global.

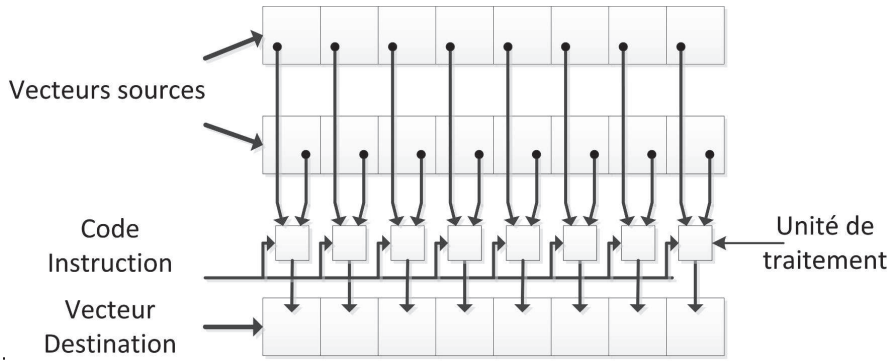


Figure 3.7: *Traitement des données par le coprocesseur vectoriel NEON*

3.8 Implantation optimisée sur une architecture de calcul vectorielle

3.8.1 Le calcul vectoriel embarqué

Le SIMD (Single Instruction on Multiple Data) désigne un mode de fonctionnement des ordinateurs dotés de plusieurs unités de calcul fonctionnant en parallèle. Ces unités de calcul appliquent la même instruction à plusieurs données différentes pour produire plusieurs résultats. Cependant, aujourd'hui, les calculs vectoriels sont encore peu développés sur les processeurs embarqués alors que ce type de calcul a démontré sa très grande efficacité lors de calculs réguliers sur des ordinateurs, par exemple pour le calcul matriciel ou certains algorithmes de traitements d'images.
Nos deux systèmes sont basés sur des architectures de types OMAP (OMAP3530, OMAP4430), ces deux architectures possèdent une ou plusieurs unités de calcul vectoriel associées au processeur. Cette unité, appellée NEON, permet l'exécution de calcul vectoriel de manière similaire aux instructions SSE (Streaming SIMD Extensions) pour les processeurs de type X86.

NEON possède deux opérateurs flottants, un opérateur entier et un opérateur 128bits dédié aux chargements et à la sauvegarde des données. Le coprocesseur NEON est optimisé pour effectuer des instructions type SIMD (Single Instruction Multiple Data). Il réalise simultanément le même calcul sur des données différentes (Figure 3.7). Le type d'opérations réalisées simultanément dépend du type de données traitées. Les opérations sont réalisées sur des vecteurs composés d'éléments d'un même type : entier/flottant, signé/non signé, 8-bits, 16-bits, 32-bits ou 64-bits.

3.8.2 Adaptation à notre algorithme

Dans l'algorithme 3.1, les blocs fonctionnels nécessitant des ressources importantes sont : le bloc d'estimation (BF6), d'initialisation (BF7, BF8 et BF9), de détection de primitives (BF2) et de corrélation ZMSSD (BF4).
Le détecteur FAST est déjà optimisé grâce à une méthode de machine learning [85].

De plus, cet algorithme a déjà été synthétisé sur une architecture hardware par Kraft *et al.* [87].
Nous avons choisi d'optimiser les autres BFs. Le premier bloc optimisé est le calcul de ZMSSD qui représente la corrélation entre deux imagettes. Ce bloc effectue la même opération (addition, soustraction, multiplication et comparaison) sur des données 8bits. Le calcul de ZMSSD peut être optimisé en utilisant l'architecture vectorielle SIMD NEON disponible sur les processeurs ARM. De même, la tâche d'estimation est basée sur plusieurs multiplications matricielles en nombres flottants. Ces multiplications sont implantables sur une architecture vectorielle, le gain sera conséquent car le Cortex A8 n'inclut pas d'unité de calcul flottant. Enfin, l'optimisation de l'étape d'initialisation sera détaillée dans le paragraphe 3.10.

3.8.3 Optimisation de l'opérateur ZMSSD

L'EKF-SLAM réalise l'appariement des amers en utilisant le Zero-Mean Sum of Squared Differences (ZMSSD). Cette mesure est utilisée pour calculer la corrélation de chaque imagette avec le descripteur de l'amer. Nous avons défini la taille du descripteur à 16×16 pixels de 8 bits car l'unité de calcul vectorielle peut traiter jusqu'à 16 nombres entiers de 8 bits simultanément.

Implantation scalaire La première implantation du ZMSSD consiste a utiliser la formule défini dans la section 2.22 :

$$N_p = \sum_{i,j}((d(i,j) - m_d) - (im(p_x + i - \frac{des}{2}, p_y + j - \frac{des}{2}) - m_i))^2 \tag{3.17}$$

avec
- $i \in [0; des-1]$, $j \in [0; des-1]$, le paramètre des représente la taille du descripteur en pixels.
- d est le descripteur.
- m_d et m_i représente respectivement la moyenne des pixels du descripteur et de la fenêtre d'image.
- im correspond à l'image.

Cette formule nécessite le précalcul de m_i (m_d peut être précalculé lors de l'initialisation de l'amer). Ensuite, le calcul utilise une boucle sur l'ensemble des pixels concernés. L'implantation scalaire est la suivante :

Algorithm 3.2 Calcul du ZMSSD scalaire

1: $m_i \leftarrow 0$
2: $N_p \leftarrow 0$
3: **for Each** $i \in [0; des-1]$, $j \in [0; des-1]$ **do**
4: $\quad m_i \leftarrow m_i + im(p_x + i - \frac{des}{2}, p_y + j - \frac{des}{2})$
5: **end for**
6: $m_i \leftarrow m_i/(des \times des)$
7: **for Each** $i \in [0; des-1]$, $j \in [0; des-1]$ **do**
8: $\quad N_p \leftarrow N_p + \sum_{i,j}((d(i,j) - m_d) - (im(p_x + i - \frac{des}{2}, p_y + j - \frac{des}{2}) - m_i))^2$
9: **end for**

Sur le Cortex A8, cette implantation prend 12.60 μs. Sur le Cortex A9, le temps de traitement est de 3.95 μs . Cette différence s'explique d'abord par la fréquence des deux processeurs (1Ghz pour le Cortex A9 contre 500Mhz pour le Cortex A8). De plus le Cortex A9 est associé à une mémoire plus rapide (DDR2) et dispose d'une architecture générale plus rapide que le Cortex A8.

Implantation scalaire optimisée Cette implantation requiert le pré-calcul de la moyenne des pixels (m_d et m_i) des deux fenêtres d'images avant de calculer $ZMSSD$. Ce calcul nécessite le chargement en mémoire de chaque pixel deux fois. Pour éviter ce double chargement, on reformule $ZMSSD$. On pose $d = d(i,j)$ et $im = im(p_x + i - \frac{des}{2}, p_y + j - \frac{des}{2})$, $ZMSSD$ devient :

$$ZMSSD = \sum_{i,j}((d - m_d) - (im - m_i))^2 \tag{3.18}$$

En developant $ZMSSD$, on obtient l'équation,

$$\begin{aligned} ZMSSD &= \sum_{i,j}((d - m_d)^2 - 2(d - m_d)(im - m_i) + (im - m_i)^2) && (3.19)\\ &= \sum_{i,j}(d^2 - 2d.m_d + m_d^2 - 2d.im + 2d.m_i + 2m_d.im - 2m_d.m_i && (3.20)\\ &\quad +im^2 - 2im.m_i + m_i^2) && (3.21) \end{aligned}$$

Les deux moyennes des pixels s'écrivent :

$$\begin{aligned} m_d &= \sum_{kl} \frac{d(k,l)}{des \times des} && (3.22)\\ m_i &= \sum_{kl} \frac{im(p_x + k - \frac{des}{2}, p_y + l - \frac{des}{2})}{des \times des} && (3.23) \end{aligned}$$

On sait que la moyenne des pixels ne dépend ni de i ni de j, d'où

$$\sum_{i,j} m_d = \sum_{i,j}\sum_{kl} \frac{d(k,l)}{des \times des} \tag{3.24}$$

$$= \sum_{kl} d(k,l) \tag{3.25}$$

$$\sum_{i,j} m_i = \sum_{i,j}\sum_{kl} \frac{im(p_x + k - \frac{des}{2}, p_y + l - \frac{des}{2})}{des \times des} \tag{3.26}$$

$$= \sum_{kl} im(p_x + k - \frac{des}{2}, p_y + l - \frac{des}{2}) \tag{3.27}$$

L'équation devient :

$$ZMSSD = \left[2\sum_{i,j} dim - (\sum_{i,j} d)^2 - (\sum_{i,j} im)^2\right] /(des \times des) + \sum_{i,j} d^2 + \sum_{i,j} im^2 - 2\sum_{i,j} dim \tag{3.28}$$

On pose les notations suivantes :
- $Sd = \sum_{i,j} d(i,j)$ la somme des pixels du descripteur.
- $Si = \sum_{i,j} im(p_x + i - \frac{des}{2}, p_y + j - \frac{des}{2})$ la somme des pixels de la fenêtre de l'image.
- $SSi = \sum_{i,j} im(p_x + i - \frac{des}{2}, p_y + j - \frac{des}{2}) \times im(p_x + i - \frac{des}{2}, p_y + j - \frac{des}{2})$ la somme du carré des pixels de la fenêtre de l'image.
- $SSd = \sum_{i,j} d(i,j) \times d(i,j)$ la somme du carré des pixels du descripteur.
- $Sdi = \sum_{i,j} d(i,j) im(p_x + i - \frac{des}{2}, p_y + j - \frac{des}{2})$ la somme du produit des pixels du descripteur et de l'image.

Finalement, $ZMSSD$ s'écrit :

$$ZMSSD = [((2Sd \times Si) - Sd^2 - Si^2)/(des \times des)] + SSi + SSd - 2Sdi \tag{3.29}$$

Cette équation limite les accès mémoire. Il n'est plus nécessaire de précalculer les deux moyennes (m_d et m_i). De plus, plusieurs sommes peuvent être pré-calculées : Sd et SSd. L'algorithme devient :

Algorithm 3.3 Implantation scalaire optimisée du ZMSSD

$Si \leftarrow 0$
$SSi \leftarrow 0$
$Sdi \leftarrow 0$
for Each $i \in [0; des-1]$, $j \in [0; des-1]$ **do**
  $Si \leftarrow Si + im(p_x + i - \frac{des}{2}, p_y + j - \frac{des}{2})$
  $SSi \leftarrow SSi + im(p_x + i - \frac{des}{2}, p_y + j - \frac{des}{2}) \times im(p_x + i - \frac{des}{2}, p_y + j - \frac{des}{2})$
  $Sdi \leftarrow Sdi + im(p_x + i - \frac{des}{2}, p_y + j - \frac{des}{2}) \times d(i,j)$
end for
$Np \leftarrow (((2Sd \times Si) - Sd^2 - Si^2)/256) + SSi + SSd - 2Sdi$

Cet algorithme n'utilise plus qu'une simple boucle, le temps de traitement pour le Cortex A8 passe de 12.60 μs à 11.29 μs, et celui du Cortex A9 passe de 3.95 μs à 3.46 μs

Implantation vectorielle Le coprocesseur NEON permet d'effectuer la même opération sur plusieurs données. Son utilisation est possible lors du calcul régulier (exemple du calcul de ZMSSD qui répète les mêmes opérations sur les différents pixels). Il est possible de vectoriser l'implantation du calcul de ZMSSD grâce à notre reformulation. Notre implantation utilise l'algorithme suivant :

Algorithm 3.4 Implantation vectorielle de ZMSSD

$V8x8\ Vimage \leftarrow 0$ ▷ V8x8: Défini un vecteur 8×8 bits
$V8x8\ Vdescriptor \leftarrow 0$
$V16x8\ VSi \leftarrow 0$ ▷ V16x8: Défini un vecteur 8×16 bits
$V32x4\ VSSi \leftarrow 0$ ▷ V32x4: Défini un vecteur 4×32 bits
$V32x4\ VSdi \leftarrow 0$
for Each $i \in [0; des-1]$, $j = 0, 8$ **do**
  $Vimage \leftarrow load_8(im(p_x + i - \frac{des}{2}, p_y + j - \frac{des}{2}))$ ▷ Chargement de 8 pixels
  $Vdescriptor \leftarrow load_8(d(i,j))$
  $VSi \leftarrow VSi + Vimage$
  $VSSi \leftarrow VSSi + Vimage \times Vimage$
  $VSdi \leftarrow VSdi + Vimage \times Vdescriptor$
end for
$Si \leftarrow sum(VSi)$ ▷ Somme des composantes d'un vecteur
$SSi \leftarrow sum(VSSi)$
$Sdi \leftarrow sum(VSdi)$
$Np \leftarrow (((2Sd \times Si) - Sd^2 - Si^2)/256) + SSi + SSd - 2Sdi$

Cette implantation traite 8 pixels à la fois. L'architecture NEON permet d'effectuer 8 additions ou 8 multiplications simultanément. Le temps de traitement sur le Cortex A8 est de 1.27 μs, il est de 0.58 μs sur le Cortex A9.

Comparaison des résultats Le tableau 3.4 résume l'ensemble des résultats de l'optimisation du calcul de corrélation ZMSSD. Tout d'abord, le gain de temps entre la version scalaire et la version vectorielle est très important pour les deux architectures. Ensuite, la différence de temps de traitement entre les deux architectures est conséquente. Cependant elle est à pondérer par la fréquence des processeurs.

	Cortex A8 - 500 Mhz (μs)	Cortex A9 - 1 Ghz (μs)
Implantation scalaire	12.60	3.95
Implantation scalaire optimisée	11.29	3.46
Implantation vectorielle	1.27	0.58

Table 3.4: *Temps de calcul de ZMSSD sur les deux architectures*

3.8.4 Optimisation de la mise à jour (BF 6)

L'étape d'estimation du filtre de Kalman est principalement constituée de multiplications matricielles, dont la taille dépend du nombre d'amers présents dans la carte ainsi que des observations réalisées. Plus la carte est grande, plus le temps de calcul est important.
L'optimisation de la multiplication matricielle est un problème souvent traité. Cependant, il n'existe à notre connaissance qu'une seule librairie qui intègre une version optimisée de cette multiplication pour des systèmes embarqués. Il s'agit de la librairie Eigen3 [99] qui implante l'algorithme défini par Goto et Geijn [100].
On se concentre sur la multiplication de matrice carré de taille N. Les matrices sont stockées en mémoire sous la forme "colonne major" : les données de la matrice sont stockées en mémoire dans l'ordre des colonnes. La multiplication matricielle ($C = A \times B$) s'écrit classiquement :

Algorithm 3.5 Multiplication matricielle

```
for Each j ∈ [0; N] do
    for Each k ∈ [0; N] do
        for Each i ∈ [0; N] do
            C(i, j) += A(i, k) × B(k, j)
        end for
    end for
end for
```

Goto et Geijn [100] proposent une optimisation basée sur le chargement en cache des lignes de la matrice B. En effet, la matrice B est parcourue en ligne alors que ses données sont sauvegardées en colonne. Le chargement en cache permettra des accès linéaires en mémoire. Cette optimisation est incluse dans la librairie Eigen3.
Pour utiliser l'unité de calcul vectorielle, les matrices sont divisées en sous matrices de taille 4×4 car le coprocesseur peut traiter des vecteurs de taille 128bits (4×32bits). L'algorithme devient :

Algorithm 3.6 Multiplication matricielle vectorielle

```
for Each j ∈ [0 : 4 : N] do
    for Each k ∈ [0 : 4 : N] do
        for Each i ∈ [0 : 4 : N] do
            C(i : i + 3, j : j + 3)+ = A(i : i + 3, k : k + 3) × B(k : k + 3, j : j + 3)
        end for
    end for
end for
```

Pour utiliser l'algorithme précédent, les matrices doivent avoir une taille multiple de 4. On arrondira la taille de la matrice au multiple de 4 supérieur. La multiplication matricielle de taille 4×4 est implantée comme proposée par ARM. La multiplication des deux matrices se décomposent comme suit :

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} & A_{14} \\ A_{21} & A_{22} & A_{23} & A_{24} \\ A_{31} & A_{32} & A_{33} & A_{34} \\ A_{41} & A_{42} & A_{43} & A_{44} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} & B_{13} & B_{14} \\ B_{21} & B_{22} & B_{23} & B_{24} \\ B_{31} & B_{32} & B_{33} & B_{34} \\ B_{41} & B_{42} & B_{43} & B_{44} \end{bmatrix} = \begin{bmatrix} A_{11}B_{11} + A_{12}B_{21} + A_{13}B_{31} + A_{14}B_{14} & \dots & \dots & \dots \\ A_{21}B_{11} + A_{22}B_{21} + A_{23}B_{31} + A_{24}B_{14} & \dots & \dots & \dots \\ A_{31}B_{11} + A_{32}B_{21} + A_{33}B_{31} + A_{34}B_{14} & \dots & \dots & \dots \\ A_{41}B_{11} + A_{42}B_{21} + A_{43}B_{31} + A_{44}B_{14} & \dots & \dots & \dots \end{bmatrix} \tag{3.30}$$

On remarque que le résultat contient la multiplication du premier vecteur colonne $\begin{bmatrix} A_{11} & A_{21} & A_{31} & A_{41} \end{bmatrix}$ par B_{11} puis le résultat est additionné au produit du second vecteur colonne $\begin{bmatrix} A_{12} & A_{22} & A_{32} & A_{42} \end{bmatrix}$ par B_{21}. On utilise cette propriété pour calculer l'ensemble de la matrice 4×4 .

L'implantation Eigen utilise 12 vecteurs : 4 pour A, 4 pour B et 4 pour C. Cependant il est important de réduire au maximum les chargements/sauvegardes de données surtout lorsqu'on dispose d'un coprocessor vectoriel. En effet, les temps de chargement ne sont pas négligeables lors des traitements rapides comme la multiplication matricielle. On propose de limiter ces chargements/sauvegardes en utilisant l'intégralité de l'architecture NEON : les 16 registres vectoriels disponibles sont utilisés. Pour cela, on calcule simultanément deux blocs de la matrice C. L'algorithme utilisé est :

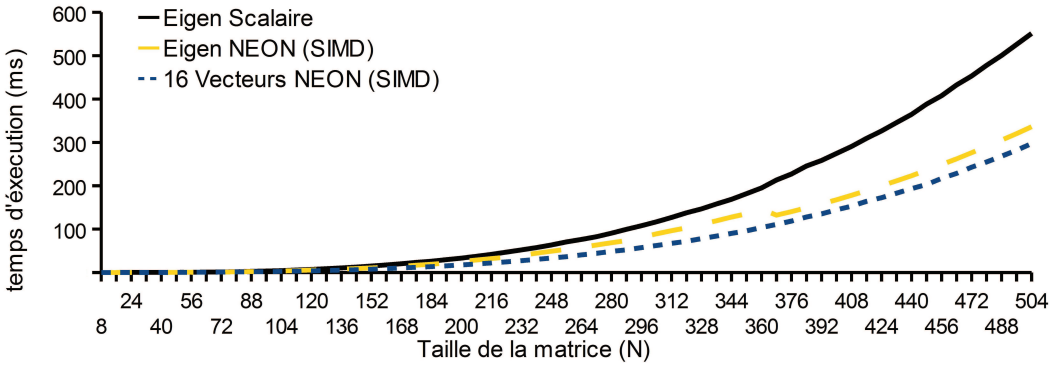


Figure 3.8: *Temps d'éxecution de la multiplication matricielle sur le processeur Cortex A9 et le coprocesseur NEON*

Algorithm 3.7 Multiplication matricielle vectorielle adaptée au coprocesseur NEON

for Each $j \in [0:8:N]$ **do**
  for Each $k \in [0:4:N]$ **do**
    for Each $i \in [0:4:N]$ **do**
      $C(i:i+3, j:j+4) += A(i:i+4, k:k+4) \times B(k:k+4, j:j+4)$
      $C(i+4:i+7, j:j+4) += A(i+4:i+7, k:k+4) \times B(k:k+4, j:j+4)$
    end for
  end for
end for

Cet implantation utilise les 16 vecteurs disponibles : 8 vecteurs pour C, 4 vecteurs pour A, 4 vecteurs pour B. De plus, il n'est pas nécessaire de recharger la matrice $B(k:k+4, j:j+4)$ ou de sauvegarder les 8 vecteurs de C.

La figure 3.8 représente les résultats des deux architectures monocore Cortex A8, Cortex A9. Tout d'abord, l'utilisation du coprocesseur vectorielle permet un gain important. Pour une taille de 504, l'implantation de Eigen passe de 551ms à 336ms soit un gain de 39%. Avec notre implantation, le temps de traitement diminue à 297ms soit un gain de 46% par rapport à la version scalaire et un gain de 11% par rapport à l'implantation de Eigen.

3.9 Implantation optimisée sur une architecture multi-cœurs homogène

Des blocs fonctionnels ont été optimisés grâce à l'utilisation du coprocesseur vectoriel. Cependant, la seconde spécificité du processeur OMAP4430 est la disposition de deux cœurs Cortex A9 possédants chacun sa propre unité de calcul vectoriel. On propose des adaptations de chaque partie de l'algorithme pour utiliser les deux processeurs disponibles. Expérimentalement, ces optimisations ont été programmées à l'aide de OpenMP.

Détection de primitives Le bloc fonctionnel FAST a été parallélisé sur les deux cœurs du processeur. Pour cela, l'image provenant de la caméra est séparée en deux parties. Les deux imagettes sont traitées par les deux processeurs. Ces calculs sont indépendants les uns des autres. On ne réalise pas de traitement spécifique pour regroupper les résultats. En effet, le détecteur ajoutera peu de points d'intérêt le long de la ligne de coupe. Le temps de traitement évalué sur l'architecture dual-core est de $3701\mu s$ contre $6581\mu s$ pour l'architecture monocore.

Mise en correspondance Pour optimiser efficacement la tâches de mise en correspondance sur l'architecture dual-core, nous avons choisi de paralléliser la tâche entière. L'exécution de la boucle est parallélisée sur l'ensemble des amers traités, chaque cœur processeur traitera l'appariement d'un amer simultanément. Sans parallélisation, l'appariement prend en moyenne un temps équivalent à $670{,}35\mu s$ (pour BF3, BF4 et BF5). Avec l'optimisation sur le dual-core (double-ARM et double-NEON), le temps de traitement est réduit à $454.51\mu s$.

Estimation L'étape d'estimation utilise le coprocesseur vectoriel NEON. Le gain en temps de traitement est considérable grâce à cette optimisation. En utilisant les deux cœurs du processeur OMAP4430, il est possible de réduire encore le temps de traitement de la multiplication. Cette parallélisation est réalisée en divisant la matrice A en deux sous matrice suivant les lignes. La figure 3.9 représente les résultats obtenus, le gain en temps de traitement est considérable. Pour la taille 504, l'implantation de Eigen décroit son temps de traitement de 336ms à 211ms soit 37% de gain, notre implantation passée d'un temps de 297ms à 153ms soit un gain de 48%. Le gain final de notre implantation par rapport à celle de Eigen est de 27% mais surtout le gain est de 72% entre la version scalaire et la version vectorielle (de 551ms à 153ms).

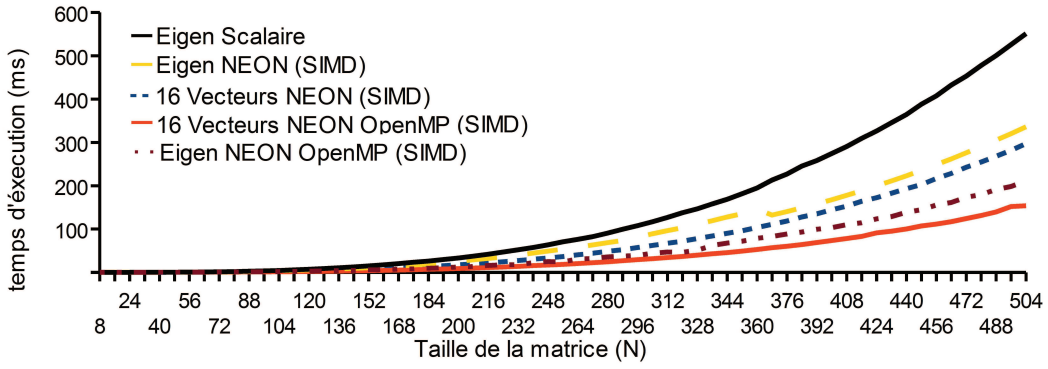


Figure 3.9: *Temps d'éxecution de la multiplication matricielle sur les deux cœurs de l'OMAP4430*

Initialisation Comme pour la tâche d'appariement, l'étape d'initialisation est paralléliser au niveau des amers traités : chaque cœur réalise l'initialisation d'un amer.

		Temps de traitement moyen par image (μs)	
Tâches	BF	Monocœur	Multicœur
FAST	2	6581	3701
Appariement	3, 4, 5	670.35	454.51
Estimation	6	38145.25	19584.26
Initialisation	7, 8, 9	338.97	225.6

Table 3.5: *Temps de traitement pour l'architecture double cœur Cortex A9 (2*1 Ghz), image 640×480 pixels, 50 amers*

Cette parallélisation permet de réduire le temps global de traitement de $794.08\mu s$ à $338.97\mu s$.

Résultat L'ensemble de l'algorithme a été parallélisé sur les deux cœurs de l'architecture OMAP4430. Le tableau 3.5 résume les différents résultats obtenus. Le temps de traitement de chaque tâche est considérablement réduit grâce à la parallélisation des traitements. Pour certaines tâches le gain de temps est pratiquement un coefficient 2 (Fast, Estimation). Pour les deux autres tâches le gain est moindre car elles n'effectuent que peu de traitement par rapport au nombre de chargement des données.

3.10 Implantation optimisée sur une architecture hétérogène

L'OMAP3530 a la particularité d'inclure un DSP. Ce type de processeur est intégré dans la plupart des systèmes de vision. Ils intègrent des fonctionnalités qui contribuent à améliorer la polyvalence des systèmes de traitement d'image. L'utilisation d'un DSP et de la mémoire partagée permet au traitement d'images d'être parallélisé.
Avec un traitement sur DSP, il est possible de paralléliser l'algorithme EKF SLAM sur les deux processeurs (ARM et DSP) : le temps global de traitement sera diminué. La prédiction ne nécessite pas de ressources processeur importante. Les tâches d'appariement et de mise à jour doivent être traitées de manière séquentielle. Toutefois la tâche d'initialisation peut être traitée en parallèle avec la tâche d'appariement / de mise à jour. Ce traitement en parallèle implique une légère complication de la phase d'initialisation. En effet, cette tâche comprend une phase d'appariement qui dépend de l'incertitude de localisation du mobile. Plus l'incertitude du mobile est grande, plus la zone de recherche de candidat potentiel τ sera grande et donc plus les erreurs d'appariement seront possibles. Cette tâche sera donc plus délicate si l'incertitude de localisation du robot est plus importante. Toutefois, nous avons vérifié que cette parallélisation n'affecte pas les résultats globaux en vérifiant que les résultats des deux algorithmes, sur un même jeu de données, sont similaires.
L'algorithme parallélisé devient :

Algorithm 3.8 Répartiation des tâcles de l'EKF SLAM sur l'OMAP3530

Initialisation de la position du robot
while 1 **do**
  if DATA = Odometres **then**
    PREDICTION
  else if DATA = Camera **then**

ARM Cortex A8	DSP C64x
MISE EN CORRESPONDANCE ET ESTIMATION	INITIALISATION

  end if
end while

3.10.1 Implantation

L'architecture de l'OMAP3530 permet de faire communiquer les processeurs ARM et DSP à l'aide d'une mémoire partagée. La figure 3.10 montre le mécanisme de transfert de données en utilisant une zone mémoire partagée. Pour chaque image acquise, le processeur ARM charge l'image (320×240 pixels), la position du robot et son incertitude dans la mémoire partagée. Lorsque le traitement d'initialisation est terminé, le DSP retourne la position et l'incertitude des nouvelles amers. Le processeur ARM choisit d'initialiser ou non les amers renvoyées en fonction des capacités de traitements (définis par un nombre d'amers maximal dans le vecteur d'état).

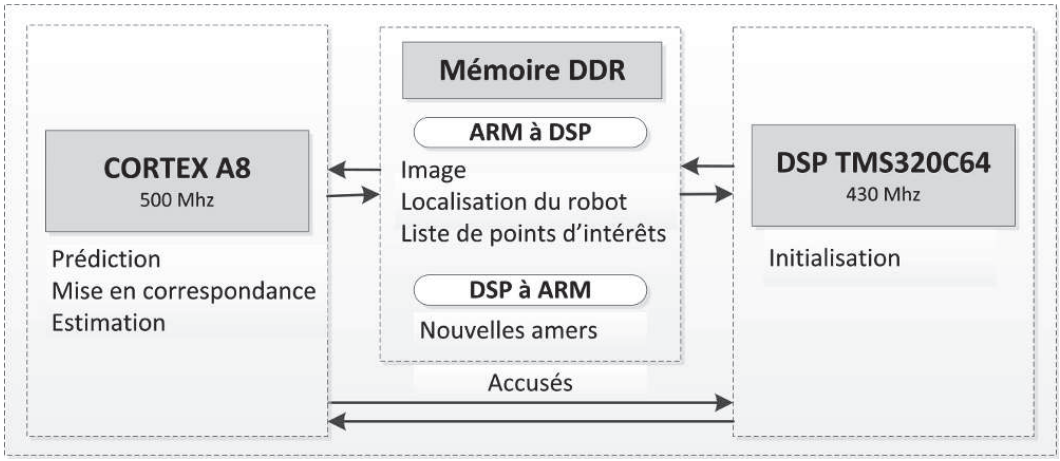


Figure 3.10: *ARM-DSP mémoire partagée*

3.10.2 Résultats

Nous avons amélioré la mise en œuvre de l'EKF-SLAM sur le processeur OMAP3530 en utilisant le coprocesseur NEON SIMD mais aussi le DSP. Le tableau 3.6 résume le temps de traitement par itération et le temps de traitement moyen par image de chaque BF. Le temps de calcul de la tâche d'initialisation (blocs 7, 8 et 9) mis en œuvre sur le processeur DSP est d'environ 4.0ms. Le processeur DSP exécute la tâche

	Implantation non optimisée (μs)		Implantation optimisée (μs)		
Blocs Fonctionels	Tps / iter (μs)	Tps moyen / image (μs)	Tps / iter (μs)	Tps moyen / image (μs)	Processeur utilisé
1. Prediction	93	93	93	93	ARM
2. FAST	3400	3400	3400	3400	ARM
3. Projection	9	180	9	180	ARM
4. ZMSSD-M	11.29	2630	1.27	295	NEON
5. $\mathbf{H}_i$	14.5	66	14.5	66	ARM
6. Estimation	88845	70568	15690	12552	NEON
Initialisation (7, 8, 9)	3992	3922	4025	4025	DSP
Total	-	**80859**	-	**16586**	-

Table 3.6: *Temps d'execution des BFs sur l'OMAP3530, image 320×240 pixels, 25 amers*

d'initialisation, tandis que les processeurs ARM-NEON éxécutent les phases de prédiction, de détection, de mise en correspondance et d'estimation.
Avec ces optimisations le temps de traitement global est composé de la prédiction (0.093ms), de la détection (3.4ms), de la tâches d'appariement (0.4ms) et de l'estimation (13.0ms). Le temps moyen de traitement par image (implantation optimisée) est de 17.6ms (nous ajoutons 1 ms pour le transfert de données entre le processeur ARM et le DSP), tandis que l'implantation non optimisée a un temps de traitement de 80.85ms. L'implantation optimisée représente seulement 22% du temps d'exécution de la version non optimisée, il a été réduit de 78%.

3.11 Comparaison des performances

L'implantation de l'EKF-SLAM a été optimisée pour deux architectures : la première est une architecture homogène possédant deux cœurs ARM Cortex A9 cadencés à une fréquence de 1Ghz, la seconde est architecturée autour d'un OMAP3530 intégrant un cœur Cortex A8 et un DSP C64x. L'utilisation d'une architecture homogène a permis une mise en œuvre plus aisée de l'algorithme, l'architecture hétérogène nécessite un travail important de gestion des communication entre les processeurs et surtout une programmation séparée des deux cœurs. La parallélisation sur l'architecture homogène est facilitée par l'utilisation de librairies, tel que OpenMP, qui permet de répartir facilement le traitement réalisé sur les différents processeurs. De plus, ce type de librairie peut gérer les différents accès concurrents à la mémoire partagée.
Au niveau des performances, l'implantation sur l'architecture hétérogène a moins tiré profit de la parallélisation, en effet il est difficile de répartir de manière efficace le traitement sur les deux cœurs. Ils sont répartis suivant des tâches complètes à cause de l'utilisation de la mémoire partagée, alors que l'architecture homogène permet de répartir des tâches ayant des temps de calcul moins importants. L'architecture hété-

rogène est moins adaptée à notre système, toutefois, pour améliorer légèrement les performances de l'implantation sur l'architecture OMAP3530, on pourrait implanter le détecteur FAST sur le DSP.

3.12 Conclusions

L'étude de l'EKF-SLAM nous a permis de définir un système complet permettant de résoudre la problématique du SLAM. Pour cela, nous avons optimisé son implantation sur une architecture homogène et une architecture hétérogène. Cette optimisation a nécessité des modifications importantes, en particulier lors de l'utilisation du coprocesseur vectoriel et du DSP. Ces modifications ont rendu l'algorithme compatible avec le temps réel sur notre plateforme expérimentale. De plus, l'adéquation entre l'algorithme et l'architecture est indispensable pour obtenir des performances satisfaisantes. En effet, les temps de traitement globaux ont été en moyenne divisé par un facteur 4. Nous avons dû limiter le nombre d'amers utilisés par le filtre pour respecter les contraintes de temps réel. Malgré une optimisation importante, la taille de l'environnement reste relativement restreinte, ceci est dû à la complexité algorithmique de l'EKF-SLAM qui dépend très fortement du nombre de points de repère présents dans la carte. De plus, la suppression des amers rend le filtre légèrement inconsistant. Une première solution à ce problème serait l'utilisation de carte locale qui permettrait de gérer les dépendances entre plusieurs cartes crées par notre système. Une seconde solution serait d'utiliser un autre algorithme qui aurait une dépendance plus faible par rapport au nombre d'amers présents dans la carte.
Pour pouvoir cartographier une plus large zone, nous avons choisi d'étudier, implanter et optimiser un second algorithme largement utilisé dans la communauté robotique : le FastSLAM. Ce filtre peut cartographier une large zone tout en conservant un temps de traitement restreint. De plus, l'étude de l'EKF-SLAM nous a montré qu'il était important de définir une implantation optimisée dépendant de l'architecture du système. Nous définirons dans le chapitre suivant une architecture matérielle entièrement dédiée au FastSLAM.

Chapitre 4

Implantation sur une architecture programmable et validation HIL : Application au FastSLAM

Sommaire

Dans le chapitre précédent, nous avons défini et évalué un système embarqué dédié à la localisation d'un robot mobile. Après avoir évalué le temps de calcul des différents blocs fonctionnels constituant l'algorithme, il est apparu que la majeure partie du temps de traitement était consacré aux équations d'estimation de l'EKF-SLAM. De plus, pour obtenir une carte et une localisation consistante et précise, il est obligatoire de conserver l'intégralité des amers observés au cours du parcours. Malheureusement, l'ajout d'amers dans le vecteur d'état et dans sa matrice de covariance augmente fortement le temps de calcul : l'EKF-SLAM souffre d'un problème de complexité algorithmique. Malgré une optimisation importante des calculs effectués lors de la phase d'estimation, nous avons dû limiter le nombre d'amers estimés par le filtre.

Pour limiter cette forte dépendance du temps de traitement aux nombre d'amers, nous avons choisi de changer d'algorithme. Le FastSLAM résoud ce problème de corrélation en utilisant un filtre particulaire pour estimer la position du mobile. Il décorrèle l'estimation de la position du robot et celle de chaque amer. Cet algorithme peut cartographier une large zone car son temps de traitement dépend faiblement de la taille de la carte. Nous avons vu précédemment qu'une optimisation de l'implantation de l'algorithme permettait d'obtenir des résultats de temps de traitement intéressants. Cette optimisation est absolument nécessaire pour obtenir un système efficace, opérant en temps réel.

Tout d'abord, nous effectuerons au paragraphe 4.1 une étude algorithmique du FastSLAM ainsi que son évaluation à l'aide de nos outils. Ensuite, après avoir étudié les différents paramètres de l'algorithme, nous définirons au paragraphe 4.4 une architecture matérielle adéquate, qui nécessitera des adaptations algorithmiques. Cette architecture sera évaluée grâce à une méthodologie HIL. Ces performances seront comparées à celles obtenues par d'autres architectures grand public.

4.1 Etude Algorithmique

Comme l'EKF-SLAM, le FastSLAM utilise des capteurs proprioceptifs pour calculer une position prédite puis corrige cette position en utilisant des capteurs extéroceptifs. Nous considérons toujours un robot à roues utilisant deux odomètres (fixés sur les roues) et embarquant une caméra frontale.

4.1.1 Hypothèse d'indépendance

Rappelons que le SLAM a pour but de localiser un mobile tout en modélisant son environnement. La problèmatique s'écrit sous la forme :

$$p(\mathbf{x}_k, \mathbf{a}_k | \mathbf{z}_k, \mathbf{u}_k) \tag{4.1}$$

avec :

– $\mathbf{x}_k$ la position du mobile à l'instant k.

– $\mathbf{a}_k$ la position des amers à l'instant k.

– $\mathbf{u}_k$ les déplacements du mobile de l'instant k.

– $\mathbf{z}_k$ les observations de l'instant 0 à l'instant k.

L'évolution de la position du robot ($\mathbf{x}_k$) dépend de la commande ($\mathbf{u}_k$) et du modèle de déplacement. Les observations ($\mathbf{z}_k$) dépendent de la localisation des amers ($\mathbf{a}_k$) et de celle du robot ($\mathbf{x}_k$). Toutes les estimations de localisation des amers sont corrélées entre elles à cause de leurs liens avec la position du robot. C'est le problème majeur de l'EKF-SLAM : cette formulation oblige à conserver l'intégralité des amers observés dans le vecteur d'état si on veut obtenir une carte et une localisation consistante.

Contrairement à l'EKF-SLAM, le FastSLAM proposé par Montemerlo *et al.* [3] représente la position du robot par un filtre particulaire avec N particules représentant chacune une localisation possible du robot. La localisation d'une particule est connue sans aucune incertitude. Toutes les estimations des positions des amers sont réalisées de manière indépendante car elles ne sont plus corrélées à l'incertitude de localisation du robot. Avec M amers dans la carte, la problématique du FastSLAM s'écrit :

$$p(\mathbf{x}_k, \mathbf{a}_k|\mathbf{z}_k, \mathbf{u}_k) = p(\mathbf{x}_k|\mathbf{z}_k, \mathbf{u}_k)\prod_{M} p(\mathbf{a}_k|\mathbf{z}_k, \mathbf{u}_k) \tag{4.2}$$

Le problème est décomposé en $M + 1$ problèmes d'estimations :

– Le premier problème est d'estimer la trajectoire du robot : $p(\mathbf{x}_k|\mathbf{z}_k, \mathbf{u}_k)$.

– Les M autres problèmes représentent l'estimation de la localisation des M amers : $p(\mathbf{a}_k|\mathbf{z}_k, \mathbf{u}_k)$.

Cette factorisation est toujours applicable dans le problème du SLAM. Contrairement à l'EKF-SLAM, la localisation des M amers est effectuée de manière indépendante.

4.1.2 Représentation des données

La position du robot est modélisée par un filtre particulaire. Chaque particule Pi représente une hypothèse sur la localisation du robot (x_i, y_i, θ_i) et est associée à un poids (ρ_i) représentant sa probabilité. De plus, cette particule possède sa propre carte représentée par un arbre binaire (Sec. 4.1.5). Chaque carte est composée de M filtres de Kalman estimant la localisation des M amers (L). Chaque filtre se trouve aux extrémités des branches de l'arbre.

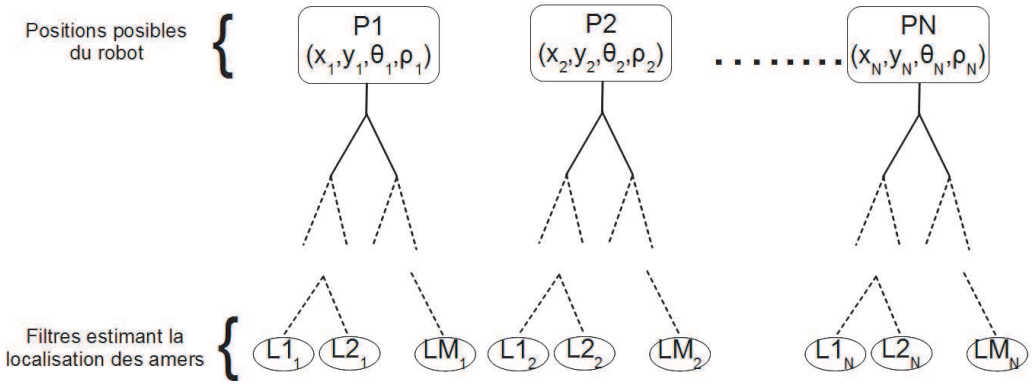


FIGURE 4.1: *Représentation du FastSLAM*

4.1.3 Etape de prédiction

L'étape de prédiction utilise les capteurs proprioceptifs embarqués : les odomètres. Chaque odomètre retourne la distance (en nombre de pas) parcourue par une roue (nt_r, nt_l). On utilise le même modèle de déplacement que précédemment (Sec. 2.18).

Les données odométriques sont sujettes à trois types d'erreurs :

- Erreurs sur les caractéristiques du véhicule : il est impossible de calculer la longueur de l'entraxe (e) ainsi que le rayon des roues (r) de manière exacte. De plus, ces variables peuvent varier légèrement au cours du temps.

- Erreurs sur les données provenant des capteurs odométriques : les odomètres sont des capteurs qui discrétisent la distance parcourue. Ils fournissent un intervalle de distance parcourue.

- Erreurs dues au glissement des roues : les roues peuvent subir des glissements et engendrer des écarts entre la distance réellement parcourue et la distance retournée par le capteur.

Pour intégrer ces trois types d'erreurs, nous ajoutons un bruit (ϵ_r,ϵ_e) pour les caractéristiques du mobile (rayon, entraxe), un bruit (ϵ_R,ϵ_L) sur les données (np_r, np_l) issues des odomètres et un bruit de glissement ($\epsilon_{g,r}$,$\epsilon_{g,l}$). Pour chaque particule i, on obtient :

$$\begin{aligned}(np_r^i, np_l^i) &= \phi(np_r, np_l, \epsilon_r, \epsilon_l, \epsilon_{g,r}, \epsilon_{g,l}) \\ &= \big(np_r(1+\epsilon_r^i+\epsilon_{g,r}^i), np_l(1+\epsilon_l^i+\epsilon_{g,l}^i)\big)\end{aligned} \tag{4.3}$$

Puis, on calcule le déplacement de chaque roue pour chaque particule (δ_r^i, δ_l^i) :

$$(\delta_r^i, \delta_l^i) = (\tfrac{2\Pi(R+\epsilon_R^i)}{N_{pt}} np_r^i, \tfrac{2\Pi(R+\epsilon_R^i)}{N_{pt}} np_l^i) \tag{4.4}$$

Pour chaque particule, on obtient les déplacements :

$$\begin{aligned}(\delta s^i, \delta\theta^i) &= g(\delta_r^i, \delta_l^i) \\ &= \left(\frac{\delta l_t^i + \delta r_t^i}{2},\ \frac{\delta l_t^i - \delta r_t^i}{2(e+\epsilon_e^i)}\right)\end{aligned} \tag{4.5}$$

Enfin, on calcule la nouvelle position de chaque particule en appliquant le modèle défini en section 2.18. A la fin de cette étape, on obtient un nuage de points symbolisants la position du robot, ce nuage prend en compte les bruits des capteurs proprioceptifs. Les bruits (ϵ_R,ϵ_e) et ($\epsilon_{g,r}$,$\epsilon_{g,l}$) sont modélisés par un bruit gaussien. Le bruit de glissement pourra être considéré comme faible si le robot se déplace à une vitesse réduite. Les bruits (ϵ_R,ϵ_e) auront une variance faible (autour du demi millimètre). Pour les bruits (ϵ_r,ϵ_l), on les modélise par un bruit uniforme compris entre -1 et 1 pas odométrique.

La figure 4.2 représente le trajet des particules (50 particules pour le graphique du haut et 200 particules pour le graphique du bas) lors d'un mouvement rectiligne suivant l'axe x. On remarque que le trajet prend en compte les différents bruits qui peuvent affecter les données odométriques. En effet, au fur et à mesure du trajet, l'ensemble des particules se dispersent dans l'environnement, elles représentent l'ensemble des hypothèses de trajet du mobile. Enfin, on remarque que plus le nombre de particules est élevé, plus la probabilité de représenter la trajectoire réelle est importante. Dans le cas des 50 particules, à la fin du trajet, plusieurs hypothèses ne sont plus représentées contrairement au cas 200 particules. La zone de localisation finale est moins dense.

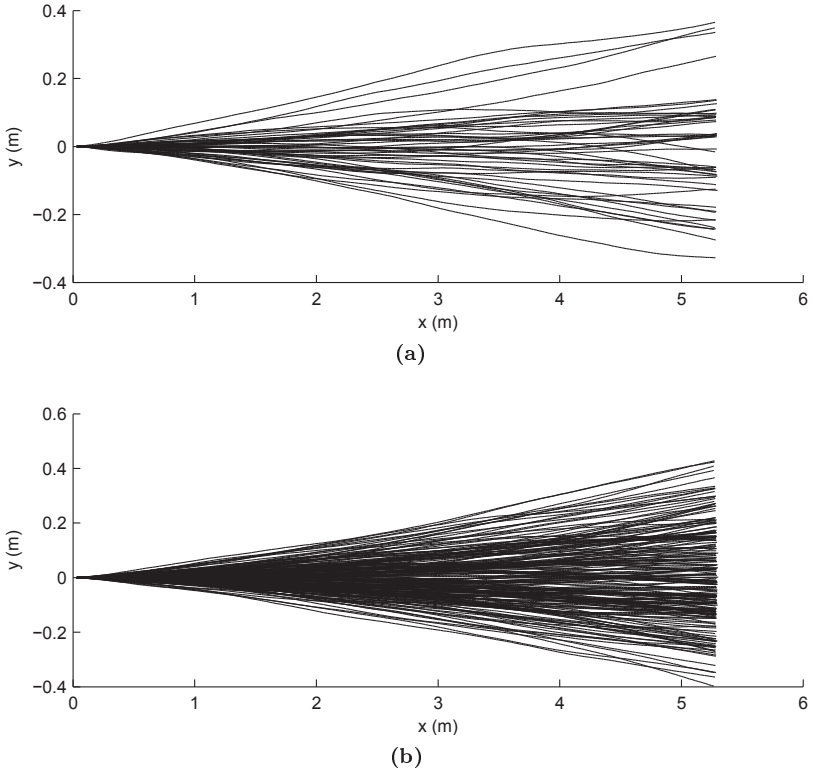


Figure 4.2: *Evolution de la position des particules lors d'un déplacement suivant l'axe x : (a) 50 particules (b) 200 particules*

4.1.4 Etape d'estimation

Chaque particule possède sa propre carte modélisée par un arbre binaire. Chaque amer est représenté par un filtre de Kalman pour chaque particule. Comme Montiel *et al.* [2], les coordonnées de chaque amer sont modélisées par l'inverse de la profondeur ρ_i et non la profondeur d_i ($\rho_i = \frac{1}{d_i}$). Cette paramétrisation, communément utilisée dans les algorithmes de SLAM ([101, 102]), produit des équations d'observation ayant un plus haut degré de linéarité qu'une simple paramétrisation xyz.

Un amer (L_i) est défini par un vecteur à 5 dimensions :

$$\mathbf{y}_i = (x_i, y_i, \theta_i, \phi_i, \rho_i) \tag{4.6}$$

Avec :
- x_i, y_i : la position de la caméra lors de la première observation.
- θ_i, ϕ_i : l'azimuth et l'élévation du vecteur pointant vers l'amer $\overrightarrow{m}(\theta_i, \varphi_i)$.
- ρ_i : l'inverse de la profondeur suivant le vecteur directeur.

La figure 4.3 représente l'ensemble de ces paramètres.

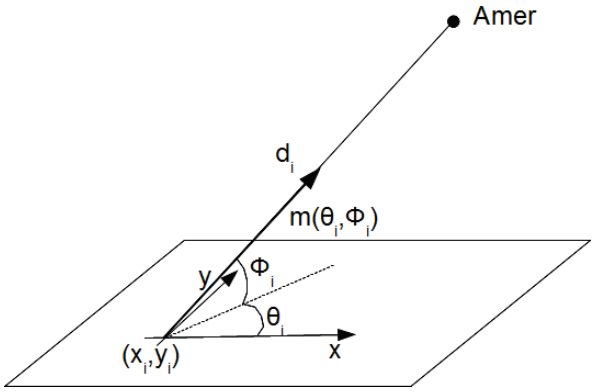


Figure 4.3: *Paramètres d'un amer*

La position de l'amer dans le repère global est calculée en utilisant la formule :

$$\mathbf{A_i} = \begin{pmatrix} x_{landmark} \\ y_{landmark} \\ z_{landmark} \end{pmatrix} = \begin{pmatrix} x_i \\ y_i \\ 0 \end{pmatrix} + \frac{1}{\rho_i}\overrightarrow{m}(\theta_i, \varphi_i) \tag{4.7}$$

avec :

$$\overrightarrow{m}(\theta_i, \varphi_i) = \begin{pmatrix} \cos(\theta_i)\cos(\varphi_i) \\ -\sin(\theta_i)\cos(\varphi_i) \\ \sin(\varphi_i) \end{pmatrix} \tag{4.8}$$

En utilisant la position du robot $\mathbf{x_k}$ et la position de l'amer $\mathbf{A_i}$, on en déduit que la position de l'amer dans le repère caméra est :

$$\mathbf{A}_{i,cam} = T_{cam,glob}(\mathbf{A}_i) \tag{4.9}$$

avec $T_{cam,glob}$ la transformation entre le repère mobile et le repère global définie par l'équation 2.5.

En remplacant $\mathbf{A}_i$ dans l'expression précédente, on obtient :

$$\mathbf{A}_{i,cam} = T_{cam,glob}(\begin{pmatrix} x_i \\ y_i \\ 0 \end{pmatrix} + \frac{1}{\rho_i}\vec{\mathbf{m}}(\theta_i, \varphi_i)) \tag{4.10}$$

La caméra n'observe pas directement $\mathbf{A}_{i,cam}$ mais sa projection sur l'image. Grâce au modèle Pinhole, on obtient :

$$\mathbf{h}_i = \begin{pmatrix} u_{i,pred} \\ v_{i,pred} \end{pmatrix} = \begin{pmatrix} c_u - fk_u\frac{A_{i,cam,x}}{A_{i,cam,z}} \\ c_v - fk_v\frac{A_{i,cam,y}}{A_{i,cam,z}} \end{pmatrix} \tag{4.11}$$

avec :

– f : la focale de la caméra.

– c_u, c_v : les coordonnées de la projection du centre optique de la caméra sur l'image.

– k_u, k_v : les facteurs d'agrandissement de l'image.

Quand un amer (i) est observé, nous mettons à jour ses coordonnées pour chaque particule en utilisant les équations du filtre Kalman. On définit l'innovation $\mathbf{Y}_i$:

$$\mathbf{Y}_i = \hat{\mathbf{z}}_i - \mathbf{h}_i \tag{4.12}$$

Avec $\hat{\mathbf{z}}_i$ l'observation réalisée.

La matrice de covariance de l'innovation $\mathbf{S}_i$ est donnée par :

$$\mathbf{S}_i = \mathbf{H}_i\mathbf{P}_i\mathbf{H}_i^T + \mathbf{R} \tag{4.13}$$

Avec $\mathbf{H}_i$ la jacobienne de $\mathbf{h}_i$ et $\mathbf{R}$ la matrice de covariance du bruit.

Ensuite, les équations d'estimations classiques du filtre de Kalman sont :

$$\begin{aligned} \mathbf{K}_i &= \mathbf{P}_i\mathbf{H}_i\mathbf{S}_i^{-1} \\ \mathbf{x}_i &= \mathbf{x}_i + \mathbf{K}_i\mathbf{Y}_i \\ \mathbf{P}_i &= (\mathbf{I} - \mathbf{K}_i\mathbf{H}_i)\mathbf{P}_i \end{aligned} \tag{4.14}$$

Après cette étape, nous avons mis à jour la position d'un amer observé pour une particule : ces calculs doivent être réalisés pour chaque amer observé et pour chaque particule.
On remarque que le filtre de Kalman n'estime que trois variables pour chaque amer : θ_i, ϕ_i et ρ_i. Les variables x_i et y_i sont supposées être connues exactement, elles représentent la position de la particule lors de la première observation de l'amer.

4.1.5 Utilisation d'un arbre binaire

Sans optimisation, le FastSLAM a une complexité en $O(NMK)$, avec N le nombre de particules, M le nombre d'amers et K le nombre d'amers observés. En effet, chaque amer présent dans la carte doit être mis en correspondance, puis si une observation est réalisée l'estimation est mise à jour pour chaque particule. De plus, lors du ré-échantillonnage, il faut copier entièrement la carte de l'environnement. Montemerlo *et al.* [3] proposent une implantation optimisée de l'algorithme en représentant la carte de l'environnement, de chaque particule, sous la forme d'un arbre binaire.

La figure 4.4 représente la mise à jour des paramètres de l'amer 4 sur la carte de la particule P1. Pour mettre à jour les paramètres d'un amer, l'algorithme va créer des feuilles intermédiaires (1′,3′). Ces dernières ont une branche qui pointe vers le reste de l'arbre qui ne change pas (respectivement 2, $L3_1$) et une branche qui pointe vers les nouvelles parties créées (1′vers 3′, 3′vers$P4'_1$). La feuille finale $P4'_1$ contient les nouveaux paramètres de l'amer. La recopie partielle est indispensable à cause de l'étape de rééchantillonnage. En effet, après cette étape, plusieurs particules peuvent partager des sous-parties d'arbre. A la fin d'une mise à jour, il est important de désalouer l'ensemble des feuilles inutilisées.

Cette implantation diminue considérablement les besoins en terme de mémoire pour stocker les paramètres des amers : une particule peut avoir des feuilles de l'arbre en commun avec d'autres particules. En effet, les cartes ne sont pas copiées, seul le pointeur vers l'arbre est mis à jour.

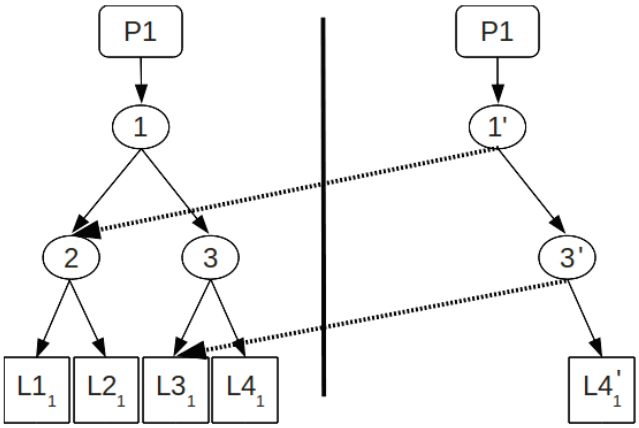


FIGURE 4.4: *Mise à jour de l'amer 4 pour la particule 1*

4.1.6 Mise à jour des poids des particules

Après avoir estimé la localisation d'un amer, le poids de chaque particule est mis à jour. La particule qui convient le mieux est celle dont la carte est le plus en adéquation avec l'observation. On utilise l'équation de mise à jour suivante, définie dans [3] :

$$\rho_j = \rho_j \exp\left(\frac{-0.5 Y_i^T S_i^{-1} Y_i}{\sqrt{||S_i||}}\right) \tag{4.15}$$

Avec Y_i le vecteur d'innovation et S_i sa matrice de covariance définie lors de l'estimation du filtre de Kalman. Il faut ensuite normaliser le poids des particules pour conserver un poids total égal à 1 :

$$\rho_i = \frac{\rho_i}{\sum_{j=1}^{N} \rho_j} \tag{4.16}$$

A la fin de cette étape, on obtient pour chaque particule un poids qui représente l'adéquation entre la position du robot, sa carte et les observations correspondantes.

4.1.7 Rééchantillonnage

Après avoir mis à jour le poids des particules, nous supprimons les particules dont la propabilité est trop faible (le seuil à été défini expérimentalement à $\rho < 0.0001$). Pour supprimer une particule, on la remplace par une copie de la particule ayant le poids le plus élevé et on divise le poids de ces deux particules par deux. Enfin, on met à jour la carte de la nouvelle particule suivant la figure 4.5. Dans l'exemple (Fig. 4.5), la particule $P2$ est supprimée et remplacée par la particule $P1$:

– On place les coordonnées de la particule au même endroit que $P1$: $x_2 = x_1$, $y_2 = y_1$, $\theta_2 = \theta_1$
– On partage le poids de$P1$ entre les deux particules : $\rho_1 = \rho_2 = \rho_1/2$
– On remplace sa carte en pointant vers la carte de $P1$.

Ce procédé de copie d'arbre évite la copie d'un arbre entier lors du remplacement d'une particule. Il suffit de copier un pointeur vers l'arbre de la particule ayant le poids le plus élevé. Cette méthode est possible grâce à l'algorithme de mise à jour d'une feuille qui réalise une copie partielle de l'arbre (Sec. 4.1.5).

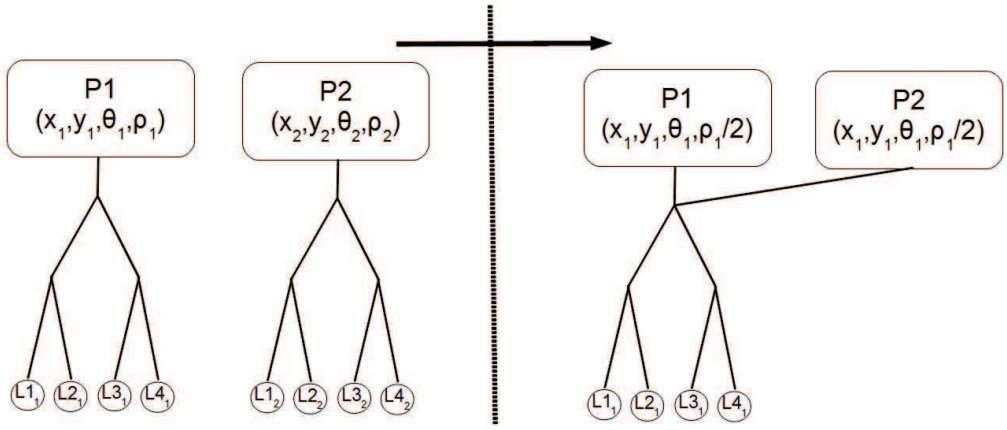


Figure 4.5: *Réchantillonnage de la Particle 2 à partir de la Particle 1*

4.1.8 Détection et appariement des amers

Pour la détection des amers, on utilise le même algorithme que l'EKF-SLAM : le détecteur FAST (voir Section 2.3.3.3). Cet algorithme détecte des points d'intérêts de manière fiable, robuste et rapide.

Nous utilisons le même algorithme de mise en correspondance que pour l'EKF-SLAM (Sec. 2.3.3.4). On projette la position de l'amer sur l'image puis on utilise la corrélation ZMSSD pour sélectionner l'observation qui sera utilisée par le filtre.
Il n'est pas envisageable de réaliser cet appariement pour chaque particule car cette étape est coûteuse en temps de calcul. Comme Eade [42], l'association n'est réalisée que pour la particule ayant le poids le plus élevé.

4.1.9 Initialisation des amers

Pour initialiser un nombre d'amers restreint et uniformément répartis, l'image est découpée en plusieurs rectangles. Après avoir projeté chaque amer sur l'image, si un rectangle ne contient pas d'amers, alors on initialise un nouvel amer (pour chaque particule). Les paramètres initiaux d'un amer sont :

– x_i, y_i : la position de la caméra lors de la première observation.
– θ_i, ϕ_i : l'azimuth et l'élévation.

– ρ_i : l'inverse de la profondeur. Donc notre cas, on la fixe à 0.26 et $\sigma_\rho = 0.25$ comme proposé par Montiel *et al.* [2].

4.1.10 Gestion des cartes

Il est probable que certains amers posent des problèmes. Par exemple, l'amer peut être localisé sur un objet mobile, être caché temporairement par un obstacle ou le processus de mise en correspondance peut échouer. On propose d'utiliser la méthode de gestion de carte définie par Eade [42]. Pour éviter de conserver des amers inutiles dans la carte, on calcule le rapport entre le nombre d'échec et le nombre de succès d'appariement. Un succès correspond à une mise en correspondance réussie entre la position prédite de l'amer sur l'image et une position observée. Un échec correspond à une impossibilité de trouver un pixel correspondant à une observation dans l'ellipse d'incertitude. Si ce rapport dépasse un certain seuil ($\frac{2}{3}$, défini par [42]), l'amer est considéré comme instable, il est supprimé de toutes les cartes.

La gestion des cartes n'a pas besoin d'être aussi efficace que pour l'EKF-SLAM car le FastSLAM est capable de cartographier un environnement incluant un nombre important d'amers.

4.2 Analyse de l'algorithme

L'algorithme 4.1 résume le déroulement de l'algorithme FastSLAM. On retrouve la même structure générale que l'EKF-SLAM : étape de prédiction, d'initialisation, de mise en correspondance et d'estimation. Des étapes supplémentaires ont été introduites : la mise à jour du poids des particules et le rééchantillonnage.
Le temps de calcul de chaque bloc ne sera pas équivalent à celui de l'EKF-SLAM. En particulier, le bloc d'initialisation n'est plus un consommateur important de ressources. En effet, grâce à l'initialisation sans délai décrite au paragraphe 4.1.9, les amers sont ajoutés directement au module d'estimation. On confirmera dans la suite que cette étape ne représente plus une partie importante de l'algorithme en terme de temps de calcul.
Pour l'étape de mise en correspondance, il n'est pas possible de réaliser un appariement par amer et par particule. Il n'est réalisé qu'une fois par amer.
L'étape d'estimation met à jour la position de chaque amer pour chaque particule. Cette tâche est la plus gourmande en temps de calcul.

Le figure 4.6 représente l'algorithme sous forme de blocs fonctionnels en incluant les données en entrées et les accès mémoire. Les étapes ne sont pas indépendantes, en effet, pour réaliser l'étape de mise en correspondance, il est nécessaire d'avoir détecté les points d'intérêts sur l'image. Pour l'étape d'estimation, il est nécessaire d'avoir calculé les observations correspondantes aux amers.

Algorithm 4.1 FastSLAM

Initialisation des particules
while 1 **do**
  DATA ← Acquisition des données capteurs
  if DATA ← (δ_r, δ_l) **then** ▷ Odometer's data
    PREDICTION :
    for Each Particle **do**
      $(nt_r^n, nt_l^n) \leftarrow \phi(nt_r, nt_l, \epsilon_r, \epsilon_l)$ (see Eq. 4.3)
      $(\delta_r^n, \delta_l^n) \leftarrow (\frac{2\Pi r}{precision} nt_r^n, \frac{2\Pi r}{precision} nt_l^n)$ (see Eq. 4.4)
      $(\delta s^n, \delta\theta^n) \leftarrow g(\delta_r^n, \delta_l^n)$ (see Eq. 4.5)
      $\mathbf{x}_k \leftarrow \mathbf{f}(\mathbf{x}_{k-1}, \delta s^n, \delta\theta^n)$ (see Eq. **??**)
    end for
  else if DATA = Image **then**
    MATCHING :
    Sélectionner la particule P ayant le poids le plus élevé
    for Each Amer $i \in$ carte(P) **do**
      $\mathbf{h} = (u_i, v_i)$ (see Eq. 4.11)
      if $(u_i, v_i) \in Camera\ Frame$ **then**
        $\tau \leftarrow$la zone de recherche
        Sélectionner l'observation $\hat{\mathbf{z}}_k$ dans τ en utilisant ZMSSD
      end if
    end for
    ESTIMATION :
    for Each Amer visible i **do**
      for Each particle P **do**
        $\mathbf{A}_i \leftarrow kalman(\mathbf{A}_i, \mathbf{h}, u_{i,obs}, v_{i,obs})$ (see Sec. 4.1.4)
        Mise à jour du poids des particles (see Eq. 4.15)
      end for
      Normalisation du poids des particules (see Eq. 4.16)
    end for
    RÉÉCHANTILLONAGE :
    for Each particle P **do**
      if $\rho_P < \varepsilon$ **then**
        rééchantillonage(P) (see Sec. 4.1.7)
      end if
    end for
    INITIALISATION :
    if Présence de zone blanche **then** (see Sec. 4.1.9)
      FAST dans la zone
      Ajout de l'amer pour chaque particule
    end if
  end if
end while

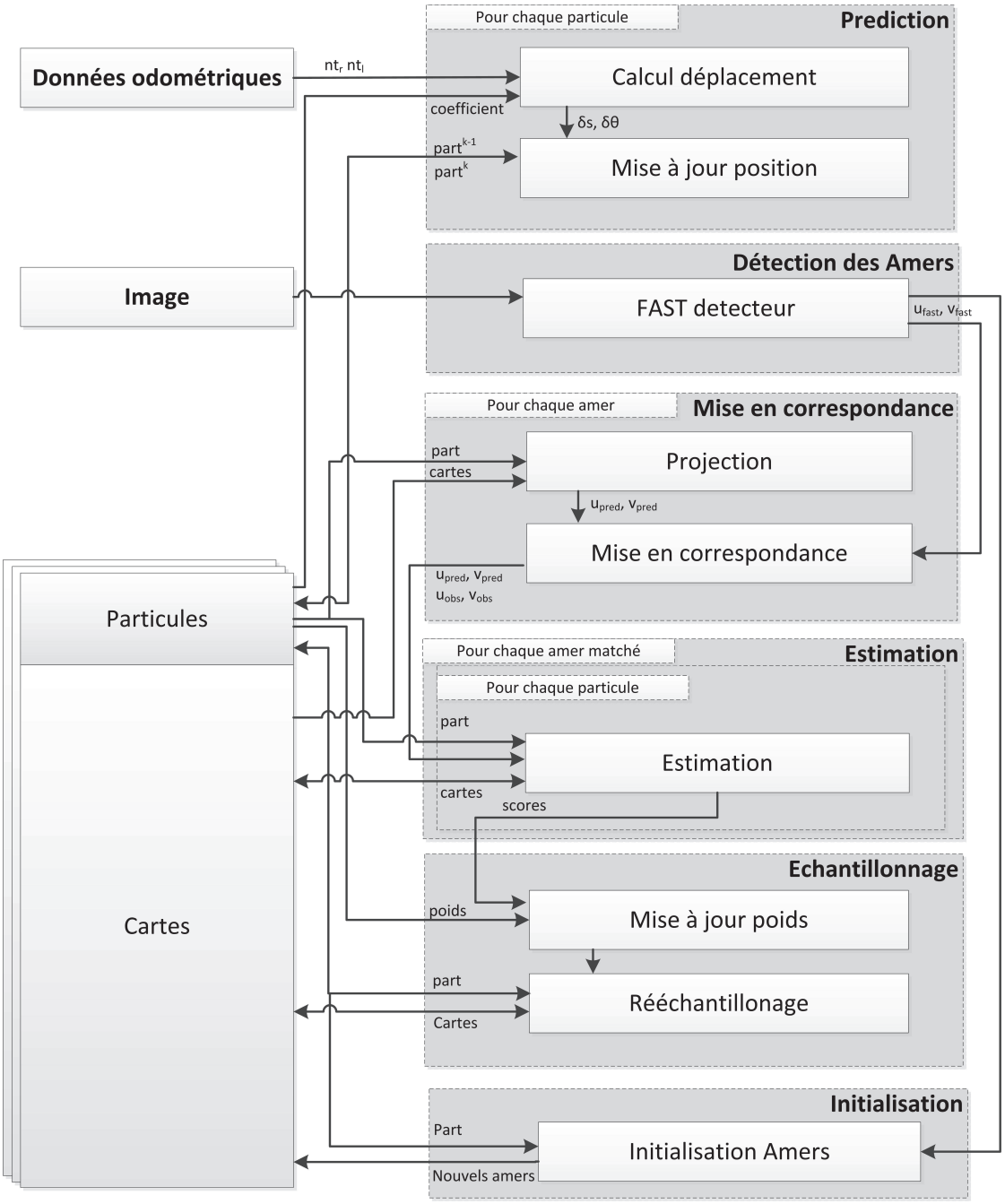


Figure 4.6: *Algorithme FastSLAM sous forme de bloc fonctionnel*

4.2.1 Etape de Prédiction

Cette étape met à jour la position de chaque particule en fonction des données proprioceptives. La première étape consiste à calculer un déplacement bruité pour chaque particule puis mettre à jour la position de la particule. Le calcul du déplacement ainsi que la mise à jour de la position sont indépendants des autres particules. Il est possible de réaliser ces traitements en parallèle pour chaque particule.

Le temps de traitement de l'étape de prédiction dépend uniquement du nombre de particule. Ce nombre est fixé dès le lancement de l'algorithme. On peut donc considérer

ce temps comme constant. De plus, le temps de calcul de cette étape est relativement limité.

4.2.2 Etape de mise à jour

La détection de point d'intérêt

Comme pour l'EKF-SLAM, nous avons choisi d'utiliser l'algorithme FAST pour détecter les amers potentiels. Le temps de traitement de cette étape dépend du nombre de points d'intérêts détectés.

La mise en correspondance

Cette étape met en correspondance les amers de la carte, avec ceux nouvellement détectés. Pour limiter le temps de traitement de cette tâche, nous avons choisi (comme Eade [42]) de la réaliser qu'une seule fois.
Chaque amer est projeté sur l'image pour définir la zone de recherche. Cette projection est réalisée en utilisant le modèle Pinhole (Eq. 4.20). Le temps de calcul de cette projection est constant. Pour chaque amer sur l'image, nous définissons l'observation correspondante à l'aide de ZMSSD (Sec. 2.3.3.4). La taille du descripteur a un impact sur le temps de calcul de la corrélation (voir Eq. 2.22). Enfin, la taille de la zone de recherche dépend de l'incertitude de localisation de l'amer et du mobile.

Le temps de traitement de cette tâche dépend des paramètres suivants :
- Le nombre d'amers dans la carte.
- Le nombre d'amers visible sur l'image.
- La taille du descripteur.
- Les incertitudes de localisation des amers et du robot.
- Le nombre de particules.

L'estimation

L'étape d'estimation met à jour la localisation de chaque amer pour chaque particule en utilisant le résultat des observations issues de l'étape de mise en correspondance. L'intérêt majeur du FastSLAM, par rapport à l'EKF-SLAM, est que le temps de calcul de cette étape dépend du nombre d'amers dans la carte. En effet, l'étape d'estimation consiste à réaliser uniquement des opérations matricielles de taille 3×3 alors que pour l'EKF-SLAM, la taille des matrices augmentait en fonction du nombre d'amers présents dans la carte.

Le temps de traitement de cette tâche est important et dépend des paramètres suivants :
- Le nombre d'amers dans la carte.
- Le nombre d'amers observés.

– Le nombre de particules.

L'initialisation

L'étape d'initilisation ajoute des amers dans la carte si il y a des zones blanches sur l'image (sans amers présents). Le temps de traitement de l'étape d'initialisation de nouveaux amers dépend du :

– Nombre d'amers à initialiser.
– Nombre d'amers dans la carte.
– Nombre de particules.

Contrairement à l'EKF-SLAM, le temps de calcul de cette étape ne représente pas une réelle contrainte car nous avons choisi d'utiliser une initialisation directe grâce à la paramétrisation par inverse de profondeur. De plus, l'étape de projection des amers sur le plan image est commune avec l'étape de mise en correspondance. On pourra donc éviter de réaliser cette projection deux fois si les deux étapes sont réalisées de manière séquentielle.

Echantillonage

Cette étape met à jour le poids de chaque particule en fonction des résultats de l'étape d'estimation. Cette mise à jour dépend du :

– Nombre d'estimations réalisées.
– Nombre de particules.

Après avoir mis à jour le poids de chaque particule, cette étape vérifie qu'une particule n'a pas un poids trop faible. Si une particule représente une probabilité trop faible, elle est remplacée par une copie de la particule ayant le poids le plus élevé. Le temps de cette étape dépend du nombre de particule à remplacer.

4.3 Validation du FastSLAM

Dans le chapitre 2, des environnements de simulation ont été définis pour évaluer les algorithmes de SLAM. Ces trois environnements représentent une petite salle, une grande salle et un ensemble de couloirs. Plusieurs expérimentations ont été enregistrées pour évaluer l'algorithme de manière "off-line". Ces expérimentations ont été réalisées grâce à la plateforme miniB (voir Section. 2.2.2). Dans cette section, le FastSLAM est évalué à partir des simulations et des expérimentations définies dans le chapitre 2.

4.3.1 Validation par Simulation

L'algorithme est évalué sur les trois environnements de simulation. Les figures 4.7 représentent les résultats obtenus dans l'environnement 2. Les résultats des environnements 1 et 3 se trouvent en annexes figure 6.3 et 6.4.

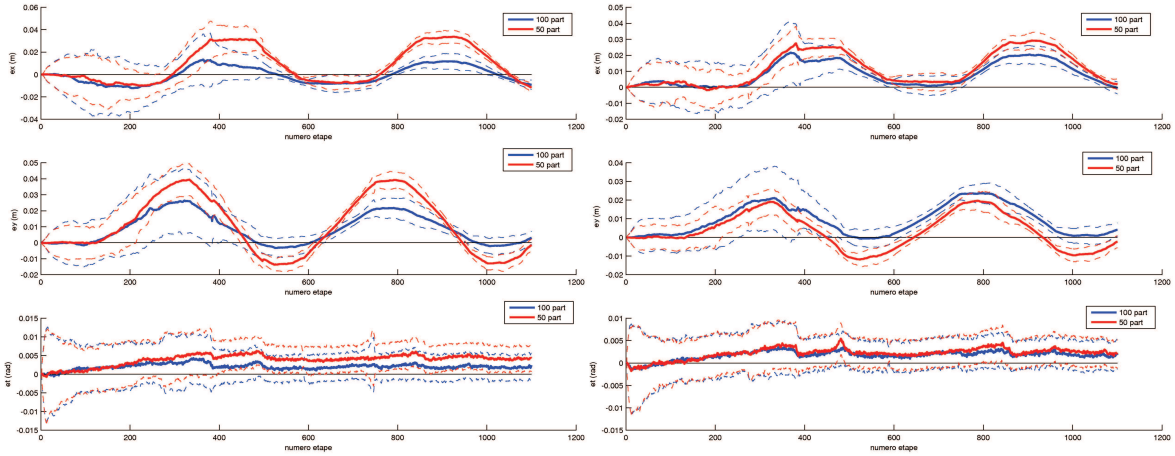
Tout d'abord, l'analyse de l'erreur euclidienne moyenne (Fig. 3.4d et Fig. 4.7d) montre que l'erreur cumulée le long du parcours est faible. De plus, cette erreur semble stabilisée à partir de l'étape 400 (fermeture de boucle), bornée et périodique. De même, la localisation de l'ensemble des amers est précise : l'erreur euclidienne moyenne est d'environ $0.2\,rad \times m^2$.
Après avoir étudié la précision de l'algorithme, il faut vérifier la consistance des résultats. Notre algorithme fournit une position précise mais aussi une incertitude associée à cette précision. Les couloirs d'incertitudes (Fig. 4.7a et 4.7b) montrent que l'algorithme est consistant durant le premier tour (jusqu'à l'étape 400) : le couloir d'incertitude inclu la valeur 0, après, il devient inconsistant quelque soit le nombre d'amers ou le nombre de particules. Cette inconsistance est vérifiée grâce aux deux graphiques représentant le NEES sur la position du mobile (Fig. 4.7e et 4.7f) et des amers (Fig. 4.7i et 4.7j). La position des amers, comme celle de la position du mobile, est très optimiste, l'incertitude est trop faible comparée à l'erreur commise. Cependant, l'erreur euclidienne commise sur la position des amers reste faible (Fig. 4.7g et 4.7h)
Cette inconsistance a été étudiée par Bailey *et al.* [45]. Il met en évidence l'inconsistance du filtre sur le long terme. En effet, l'étape de rééchantillonnage supprime des particules pour se concentrer sur la position la plus probable. Ce rééchantillonnage provoque la suppression de cartes qui rend l'ensemble inconsistant. Il est tout de même remarquable que ces inconsistances ne perturbent que faiblement les résultats de précision.
Les conclusions issues de cette environnement reflète les résultats des deux autres environnements. Cependant, on remarque que la qualité de la localisation dépend de la taille de l'environnement : plus l'environnement est grand, plus la localisation est dégradée.

En comparaison avec l'EKFSLAM, les résultats de localisation sont de meilleure qualité, l'erreur euclidienne moyenne est plus faible.

4.3.2 Validation Expérimentale

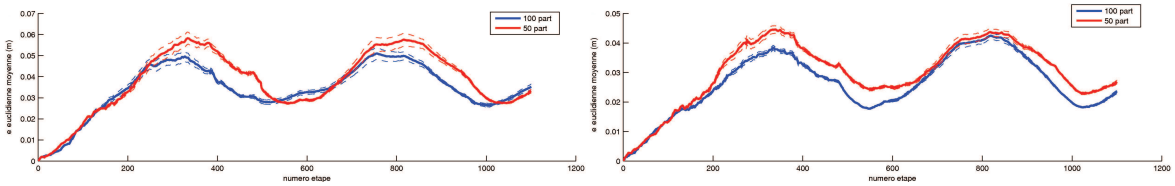
Après avoir validé en simulation le fonctionnement du FastSLAM, on le valide expérimentalement en utilisant le jeu de données collectées par notre plateforme. Pendant ce jeu de données, le robot parcourt trois tours approximativement similaires. Durant l'expérimentation, le robot est passé au dessus de plusieurs points de passage. Ces points de référence permettent de quantifier la qualité des résultats issus de l'algorithme. Les figures 4.8 représentent l'ensemble des résultats concernant cette expérimentation.

La figure 4.8a représente le trajet des particules durant le premier tour de l'expérimentation. Le nuage de points est approximé par une ellipse. Le tracé noir représente la trajectoire du mobile, les ellipses rouges représentent l'incertitude de localisation. On remarque que le trajet décrit par le nuage de particules suit correctement le trajet décrit



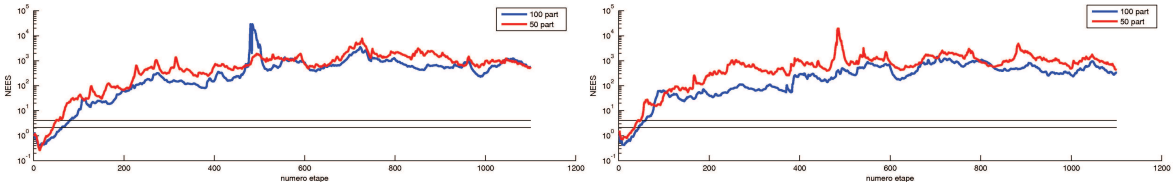
(a) *Couloir d'incertitude sur la position x, y, θ pour l'environnement 2 rempli de 100 amers*

(b) *Couloir d'incertitude sur la position x, y, θ pour l'environnement 2 rempli de 500 amers*



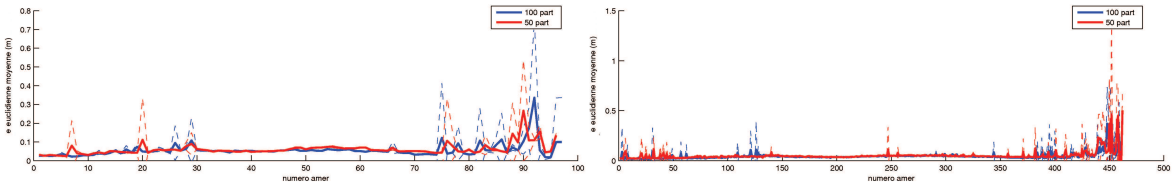
(c) *Erreur euclidienne de la position x, y, θ pour l'environnement 2 rempli de 100 amers*

(d) *Erreur euclidienne de la position x, y, θ pour l'environnement 2 rempli de 500 amers*



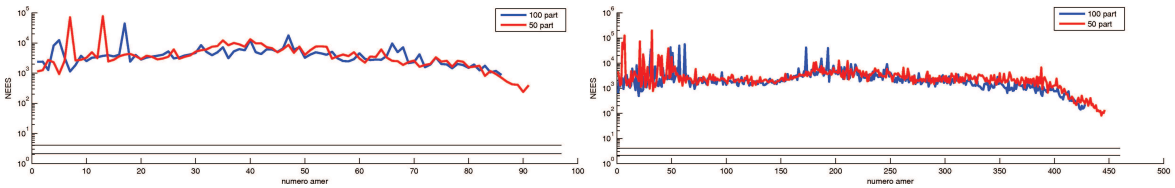
(e) *NEES de la position x, y, θ pour l'environnement 2 rempli de 100 amers*

(f) *NEES de la position x, y, θ pour l'environnement 2 rempli de 500 amers*



(g) *Erreur euclidienne de la position des amers pour l'environnement 2 rempli de 100 amers*

(h) *Erreur euclidienne de la position des amers pour l'environnement 2 rempli de 500 amers*



(i) *NEES pour la position des amers pour l'environnement 2 rempli de 100 amers*

(j) *NEES pour la position des amers pour l'environnement 2 rempli de 500 amers*

Figure 4.7: *Résultats du FastSLAM pour l'environnement 2*

par les points de passage. De plus, contrairement à l'EKF-SLAM, l'algorithme arrive à conserver une incertitude qui grandit au cours de l'expérimentation. L'incertitude de localisation est maximale avant la fermeture de boucle. Cette fermeture de boucle correspond au moment où le robot retourne dans un espace précédemment cartographié. On remarque à ce moment que l'incertitude de localisation diminue fortement.

Les figures 4.8b et 4.8c représente le second et le troisième tour de l'expérimentation. Ces deux tours sont relativement similaires. La qualité de localisation est bonne comparativement à l'intégration des données odométriques. La trajectoire définie par le FastSLAM suit correctement le trajet de référence. Cependant, on remarque que durant ces deux tours, l'incertitude de localisation du robot est très fortement réduite. En effet, les ellipses de localisation sont beaucoup trop petites : les ellipses n'incluent pas les positions de référence. Le filtre devient inconsistant malgré ces bons résultats en terme de localisation.

La figure 4.8d confirme la qualité des résultats du FastSLAM comparativement à l'intégration odométrique. En effet, l'intégration odométrique souffre d'une erreur cumulative qui augmente au fur et à mesure de l'expérimentation alors que le FastSLAM a une erreur stabilisée.

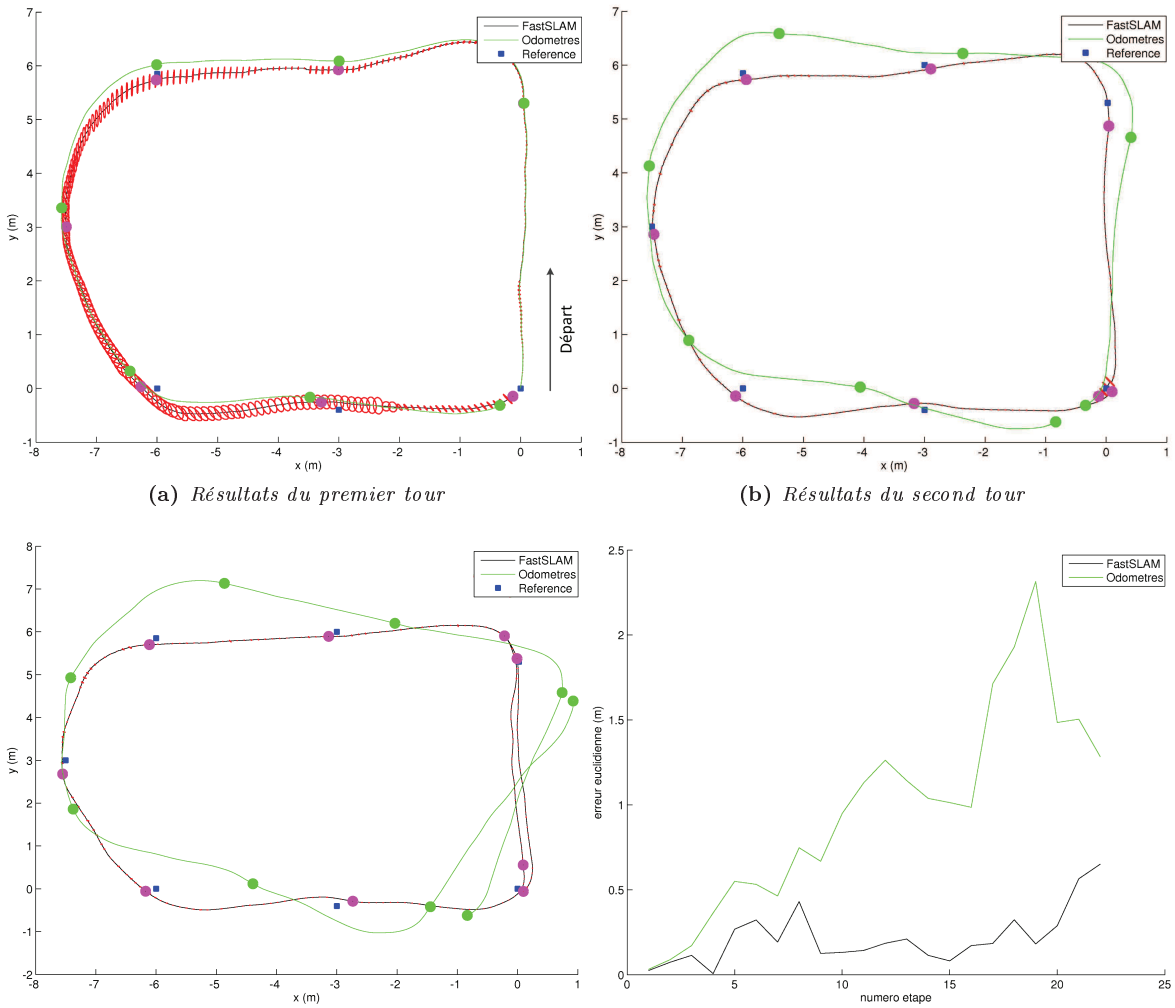
L'efficacité du FastSLAM est confirmée : le trajet résultat est meilleur que l'intégration des données odométriques. Afin de concevoir une architecture adaptée, nous utiliserons le fait que le FastSLAM réalise de nombreuses opérations de manière itératives. Les opérations sont majoritairement réalisées pour chaque particule. Nous allons donc pouvoir utiliser cette particularité pour concevoir une architecture massivement parallèle en adéquation avec notre algorithme.

4.4 Définition d'une Architecture Programmable

L'algorithme décrit précédemment contient l'intégralité des étapes permettant de reconstruire l'environnement et de se localiser. Nous avons choisi de nous focaliser sur l'implémentation du cœur de l'algorithme : le filtre particulaire. Nous ne traiterons pas l'étape de détection de primitive ni l'étape de mise en correspondance. L'étape de détection de primitive (algorithme FAST) a déjà été implémentée avec succès sur une architecture programmable [87]. De même, l'étape de mise en correspondance a déjà fait l'objet de nombreux travaux [103]. On se place dans le cas où les capteurs nous retournent l'identifiant d'un amer ainsi que sa position sur l'image.

4.4.1 Définition des blocs fonctionnels

Pour pouvoir concevoir un système efficace, il est nécessaire d'étudier les besoins de notre algorithme. Nous avons étudié les dépendances temporelles de chaque partie de notre algorithme. Cependant, les temps de traitement de ces parties ne sont pas



(a) *Résultats du premier tour*

(b) *Résultats du second tour*

(c) *Résultats du troisième tour*

(d) *Erreur euclidienne pour les 22 points de référence*

Figure 4.8: *Résultats expérimentaux du FastSLAM*

constants. Comme nous l'avons fait au chapitre 3, on propose de découper chaque partie en plusieurs blocs fonctionnels ayant un temps de traitement constant.

Blocs Fonctionnels (BF)	Description	Lignes
1. Calcul δs et $\delta\theta$	Calcul des déplacements à partir des données odométriques	7 à 9
2. Modèle de déplacement	Déplacement du nuage de particules	10
3. Initialisation d'un amer	Initialisation d'un amer	39
4. Projection	Projection d'un amer sur l'image	16
5. Calcul de **H**	Calcul de la jacobienne **H**	25
6. Estimation	Estimation de la localisation d'un amer	25
7. Mise à jour	Mise à jour du poids des particules	26, 28
8. Rééchantillonnage	Rééchantillonnage du nuage de particules	32 à 34

Table 4.1: *Définition des blocs fonctionnels*

Chaque bloc fonctionnel défini dans le tableau 4.1 a un temps de traitement constant si le système n'utilise qu'une seule particule. On exprimera donc le temps de traitement des blocs en fonction du nombre de particules.

4.4.2 Choix d'une architecture adaptée

Le FastSLAM permet à un robot mobile de se localiser dans un environnement inconnu. Ce type d'algorithme est principalement utilisé en robotique mobile pour permettre au robot d'obtenir une plus grande autonomie. Cette condition nous impose d'utiliser une architecture qui peut être embarquée sur une plateforme de taille réduite. Dans les chapitres précédents, nous avons étudié deux architectures différentes : un OMAP3530 et un OMAP4460. Cependant, ces architectures ne sont pas réellement adaptées au FastSLAM. En effet, le FastSLAM réalise plusieurs boucles de calculs indépendants pour chaque particule : ces calculs peuvent être réalisés de manière parallèle. Les processeurs OMAP3530 et le OMAP4460 ne disposent que de deux cœurs et ne sont donc pas adaptés au calcul massivement parallèle.
Les architectures massivement parallèles se développent de manière très importante depuis quelques années. Le GPU (Graphics Processing Unit) est un exemple d'architecture massivement parallèle. Cependant, les GPU embarquables sur un robot mobile sont encore aujourd'hui très rares et peu performants. L'une des plateformes massivement parallèles embarquées est IMAPCAR [104]. Ce processeur, conçu par NEC, est dédié à la réalisation de reconnaissance d'images en utilisant des fonctions avancées de traitement parallèle pour les systèmes de sécurité automobile. Cette plateforme reste cependant difficilement accessible.
Pour obtenir une architecture massivement parallèle, adéquate à l'algorithme FastSLAM, nous avons opté pour une architecture reconfigurable à base de FPGA (Field-Programmable Gate Array). L'utilisation d'une architecture dédiée a démontré de bons

Blocs Fonctionnels (BF)	Temps d'execution par itération (nombre de cycle / par particule)	Nombre moyen d'occurence par mise à jour	Temps moyen d'execution pour 200 particules (μs)
1. Calcul δs et $\delta\theta$	42	1	168
2. Modèle de déplacement	109	1	436
3. Initialisation d'un amer	86	0.4	136
4. Projection	416	14.1	23460
5. Calcul de **H**	1339	9.6	51416
6. Estimation	2115	9.6	81216
7. Mise à jour	168	9.6	6450
8. Rééchantillonnage	22000	1.3	572
		Total :	163.85 ms

Table 4.2: *Temps d'execution par particule des blocs fonctionnels sur le processeur NIOS2*

résultats dans le cas de calculs massivement parallèles. En particulier, Botero *et al.* [8] ont proposé une architecture dédiée, dans le cadre d'un algorithme de SLAM, pour la détection d'amers. Cependant, le filtre (EKF-SLAM) est exécuté sur un processeur softcore (MicroBlaze). L'utilisation d'une architecture dédiée a permis d'obtenir une détection des primitives à la fréquence pixel. Dans notre cas, nous nous concentrerons sur la définition d'une architecture, fortement parallèle, matérielle dédiée au FastSLAM.

4.4.3 Outils d'évaluation temporelle

Pour anaylser les performances d'un bloc fonctionnel, il est nécessaire de calculer son temps de traitement. Pour cela, on utilise un timer matériel qui permet de calculer le temps d'éxecution d'un bloc en nombre de cycle processeur. Le temps d'éxecution en secondes dépend du nombre de cycle ainsi que de la fréquence du processeur.
Le temps d'éxecution dépend très souvent dans notre cas du nombre de particules. Il sera donc normalisé par rapport au nombre de particules. Pour cela, on calculera le temps de traitement pour 100 particules puis on le divisera par 100.

4.4.4 Temps d'exécution des blocs fonctionnels

On effectue une première analyse du temps de traitement de chaque bloc fonctionnel sur le processeur NIOS2. Pour cette étude, nous avons choisi d'utiliser le NIOS2 le plus performant incluant des multiplicateurs et des diviseurs matériels. Le temps total de traitement est d'environ 163.85 ms pour 200 particules. Comme l'EKFSLAM, la majeure partie du temps de traitement est consacrée aux calculs de l'estimation de la position des amers. Ce bloc de calculs représente 49.5% du temps total de traitement.

4.4.5 Simplifications des hypothèses

4.4.5.1 Modélisation du robot

Précédemment, nous avons défini trois repères différents : le repère global, le repère caméra et celui du mobile. Expérimentalement, la position du repère caméra est très proche du centre de l'essieu (repère mobile). Pour simplifier les équations de changement de repère, on considère que la caméra est positionnée exactement au même niveau que le repère mobile.

4.4.5.2 Représentation de la carte

Montemerlo *et al.* [3] ont proposé une implantation de l'algorithme en utilisant un arbre binaire pour représenter la carte de l'environnement. Cependant, ce mode de représentation a été relativement contesté. Par exemple, Sim [105] montre que le temps de calcul de l'algorithme n'atteint pas la complexité souhaitée mais reste plus important que prévu. Plusieurs hypothèses sont émises. En particulier, Sim [105] souligne le fait que l'étape de rééchantillonnage va fragmenter la mémoire. En effet, les cartes sont représentées par des arbres puis copiées partiellement. Il est impossible que les cartes soient représentées de manière linéaire en mémoire. Or, l'accès non linéaire en mémoire est plus coûteux en temps d'accès qu'un accès linéaire. Cette fragmentation provoque une hausse importante du temps de calcul.
Nous choisissons de représenter les cartes par un tableau linéaire. Cette représentation permet de ne pas fragmenter la mémoire et d'obtenir un temps de traitement déterministe lors de la copie d'un arbre ou de l'estimation d'un amer. Enfin, ce choix supprime toute allocation dynamique de mémoire, les cartes sont allouées au lancement du processus.

4.4.6 Représentation des variables

L'analyse globale de l'algorithme a permis de décomposer le traitement en blocs fonctionnels. Elle a mis en avant les besoins importants en terme de mémoire dus à l'utilisation d'un nombre important de particules et d'amers.
Les temps d'accès à la mémoire sont un des facteurs limitant pour une architecture matérielle. Il est nécessaire de représenter le plus efficacement possible les différentes variables du système pour minimiser ces accès.

4.4.6.1 Position du robot

Le FastSLAM utilise un filtre particulaire pour représenter la position du robot. Chaque particule est représentée par quatre variables x, y, θ, p. Les deux premières variables représentent la position en deux dimensions du mobile dans le plan, θ son orientation et ρ la probabilité de la particule.

On représente les variables sous la forme :
- x/y : un entier signé sur 32 bits avec une précision au centième de millimètre.
- θ : un entier non signé sur 32 bits représentant l'orientation du mobile de $-\pi$ à π.
- p : un entier non signé sur 32 bits représentant le poids de la particule allant de 0 à 4294967295.

Chaque particule est représentée par 4 entiers de 32 bits. Des simulations ont été réalisées pour valider le choix de ces représentations (Sec. 4.6). En représentant les variables sur seulement 16 bits, des approximations très importantes sont réalisées et les résultats ne sont plus satisfaisants.

4.4.6.2 Représentation de la carte

Chaque particule possède sa propre carte de l'environnement, composée d'amers. Un amer est représenté par 5 variables $(x_i, y_i, \theta_i, \varphi_i, \rho_i)$:
- x, y : la position de la caméra lors de la première observation.
- θ, φ : l'azimuth et l'élévation du vecteur pointant vers l'amer.
- ρ : l'inverse de la profondeur suivant le vecteur directeur.

On représente ces variables par :
- x/y : un entier signé sur 32 bits avec une précision au dixième de millimètre.
- θ/φ : deux entiers non signés 32 bits représentant l'azimuth de $-\pi$ à π et l'élévation de $-\pi/2$ à $\pi/2$.
- ρ : un entier non signé 32 bits.

On associe aux variables θ, φ, ρ une matrice de covariance de dimension 3×3 formée de 9 entiers signés codés sur 32 bits. Chaque amer est donc représenté par 14 entiers de 32 bits soient 448 bits par amer.

4.4.6.3 Capacité mémoire

Nous avons vu précédemment que le FastSLAM traite de nombreuses particules et que chacune dispose de sa propre carte. Le nombre d'amers dans une carte ne peut pas être limité sauf dans le cas ou l'on supprimerait des amers pour en insérer de nouveaux. De même, il n'existe pas un nombre optimal de particules : il dépend de la taille de l'environnement et de la qualité de localisation souhaitée.
La figure 4.9 représente l'espace mémoire maximum nécessaire en fonction du nombre d'amers et du nombre de particules. Par exemple, pour 100 particules et 50 amers, l'algorithme a besoin de 0.26Mo de mémoire. Pour 500 particules et 500 amers, l'espace mémoire nécessaire est de 13.3Mo.

4.4.7 Outils de développement

La création de bloc de traitement hardware est souvent longue et compliquée lorsque l'utilisateur doit développer directement en langage de description matériel. Cependant

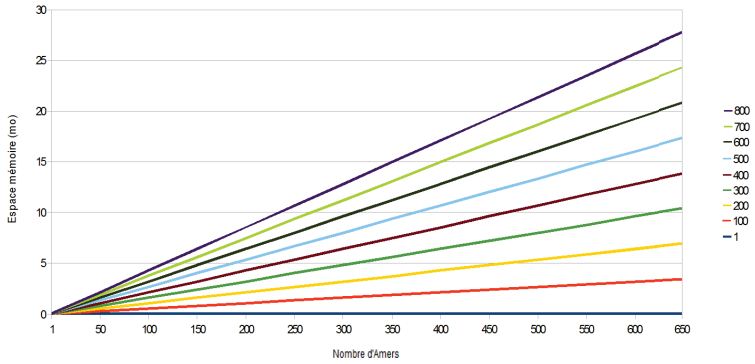


Figure 4.9: *Espace mémoire nécessaire en fonction du nombre d'amers et du nombre de particules*

des outils existent pour permettre à l'utilisateur de créer des fonctions hardware à partir de programmes écrits en langage C. En particulier, le compilateur C2H de Altera est un outil utilisé pour convertir des fonctions C en briques matérielles pour un FPGA. Cette transformation permet d'améliorer de manière significative les performances des fonctions. Cette conversion nécessite quelques changements du code C pour la gestion des bus de données, de la mémoire et des opérateurs de calculs (MUL, DIV, ADD...).

4.4.7.1 Parallélisme

C2H permet de paralléliser l'ensemble des calculs réalisés tant qu'il n'y a pas de dépendance entre les résultats des calculs. La figure 4.10 symbolise la parallélisation réalisée par notre outil de développement, plusieurs accélérateurs matériels sont utilisés en parallèle. Le nombre d'accélérateurs est défini par le facteur limitant, ce facteur est fréquemment la vitesse d'acquisition des données.

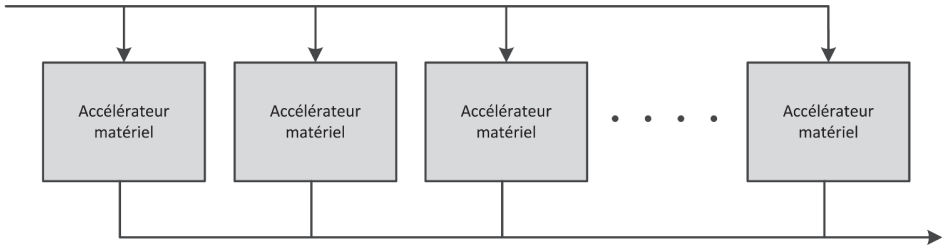


Figure 4.10: *Parallélisation du bloc fonctionnel*

4.4.7.2 Exemple de conception de bloc

Pour illustrer les principes de la conception de blocs matériels à l'aide de C2H, on se propose d'utiliser un exemple simple. La fonction à implanter est :

Algorithm 4.2 Fonction à implanter à l'aide de C2H

```
1: #define size 100
2: void fonction(alt_32 *a, alt_32 *b, alt_64 *c, alt_64 *d){
3: alt_u32 i;
4: for (i = 0; i < 100; i++) do
5:     c[i] = a[i] × b[i];
6:     d[i] = a[i] + b[i];
7: end for
8: }
```

La figure 4.11 représente le diagramme temporel de l'exécution séquentielle. On remarque que chaque chargement, sauvegarde ou calcul est réalisé séquentiellement.

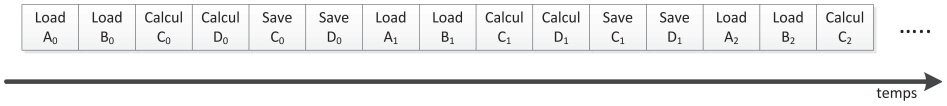


Figure 4.11: *Diagramme temporel de l'exécution séquentielle*

En utilisant une implantation logicielle sur le processeur NIOS2, le temps de traitement de cette fonction est de 9334 cycles processeurs. En utilisant cette implantation directement avec C2H, le gain est de seulement 7.3%, le temps est réduit à 8654 ticks.

Il est possible de réaliser une première optimisation en indiquant au compilateur que les données des quatres tableaux (a, b, c ,d) sont indépendantes. On utilise pour cela le qualificatif "__restrict__" lors de la définition de la fonction :

Algorithm 4.3 Utilisation de la directive __restrict__

```
1: void fonction(alt_32 * __restrict__ a, alt_32 * __restrict__ b, alt_64 * __restrict__ c,
   alt_64 * __restrict__ d)
```

En utilisant ce mot clé, le compilateur peut paralléliser les calculs. La figure 4.12 représente le diagramme temporel incluant les calculs parallèles, les calculs de $c[i]$ et $d[i]$ sont réalisés en même temps.

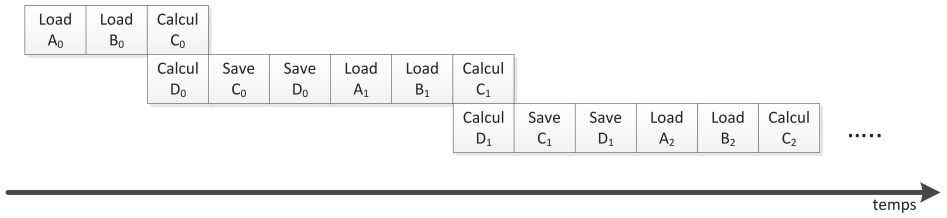


Figure 4.12: *Diagramme temporel de l'exécution avec calculs parallèles*

Le temps de traitement est maintenant de 4955 cycles processeurs soit un gain de 47% par rapport à la version logicielle.

La seconde optimisation possible est de stocker les quatres tableaux dans quatres mémoires différentes. Dans ce cas, il sera possible de charger en mémoire ou de sauvegarder plusieurs données en même temps. On utilise la directive "#pragma altera_accelerate

connect_variable * to *" pour spécifier l'association entre une mémoire et une variable. Le programme devient finalement :

Algorithm 4.4 Fonction optimisée à implanter à l'aide de C2H

```
1: #define size 1000
2: #pragma altera_accelerate connect_variable fonction/a to stock_a
3: #pragma altera_accelerate connect_variable fonction/b to stock_b
4: #pragma altera_accelerate connect_variable fonction/c to stock_c
5: #pragma altera_accelerate connect_variable fonction/d to stock_d
6: void fonction(alt_32 *a, alt_32 *b, alt_64 *c, alt_64 *d){
7: alt_u32 i;
8: for (i = 0; i < 100; i++) do
9:     c[i] = a[i] × b[i];
10:    d[i] = a[i] + b[i];
11: end for
12: }
```

Il est nécessaire de vérifier que les quatres blocs mémoires sont uniquement reliés à l'accélérateur matériel. L'ensemble des chargements et des sauvegardes peut être parallélisé. La figure 4.13 représente la version entièrement parallélisée de notre fonction. Plusieurs chargements, sauvegardes et opérations peuvent être réalisés simultanément. Une fois les premières étapes effectuées, notre accélérateur matériel réalise 6 opérations en un seul cycle CPU.

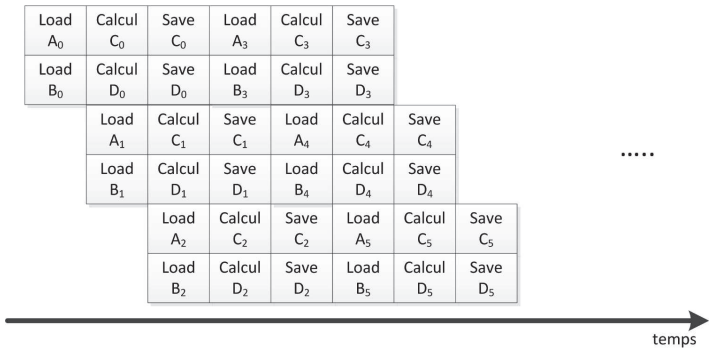


Figure 4.13: *Diagramme temporel de l'exécution parallèle*

Le temps de traitement de cette version optimisée est de 1596 cycles soit un gain de 67% par rapport à la version ayant une seule mémoire et un gain de 82% par rapport à la version logicielle. Le gain en temps de traitement dans ce cas est très important car les différents calculs peuvent être aisément réalisés en parallèle.

4.4.8 Outils d'évaluation des blocs matériels

Pour évaluer une architecture, nous avons défini précédemment une méthodologie et des critères d'évaluation. Cette dernière est composée de deux étapes : la première consiste à vérifier les résultats issus de l'architecture et la seconde concerne la caractérisation temporelle.

4.4.8.1 Analyse fonctionnelle

Le FastSLAM a été évalué fonctionnellement au paragraphe 4.3. La validation fonctionnelle de l'architecture consiste à vérifier que les résultats issus de cette nouvelle architecture confirment les résultats précédemment obtenus. La principale difficulté est l'utilisation unique de nombres entiers. En effet, C2H ne gère pas les nombres flottants. De plus, la création de blocs matériels utilisant des nombres flottants serait trop consommatrice de ressources.
Chaque bloc défini est évalué fonctionnellement. Pour cela, une simulation est réalisée en tirant aléatoirement les données d'entrées. Les résultats sont comparés à ceux de l'algorithme évalué au paragraphe 4.3.
Après avoir évalué chaque bloc indépendamment, plusieurs simulations sont réalisées pour évaluer le fonctionnement de l'intégralité de l'architecture.

4.4.8.2 Intensité arithmétique

Nous avons vu dans le paragraphe 4.4.7.2 que l'utilisation de bloc matériel pouvait diminuer très fortement le temps de traitement des fonctions associées. Deux types d'optimisation ont été mises en avant. La première consiste à paralléliser les calculs et la seconde à stocker les données dans différentes mémoires pour accélérer leurs chargements ou sauvegardes.
L'intensité arithmétique (IA), définie par Dally *et al.* [106], relie les deux composantes utilisées dans nos optimisations : les opérations et les accès mémoire. L'intensité arithmétique correspond au ratio entre le nombre d'opérations arithmétiques et le nombre d'accès mémoire. Plus la valeur de l'intensité arithmétique est élevé, plus le nombre d'opérations par rapport aux accès mémoire est important. A l'inverse, plus l'intensité arithmétique est faible, plus le nombre d'accès mémoire est important par rapport au nombre d'opérations.

L'intensité arithmétique a été principalement utilisée pour la conception d'optimisations d'algorithmes sur processeur graphique (GPU). En effet, les calculs sur GPU sont fortement contraints par les accès mémoire qui ralentissent les optimisations possibles. L'objectif des travaux est de maximiser l'intensité arithmétique des algorithmes pour pouvoir réaliser une implantation efficace sur des processeurs disposant de nombreux processeurs comme les GPU. Par exemple, Buck *et al.* [107] utilisent ce critère pour définir des optimisations. Castaño-Díez *et al.* [108] utilisent ce même critère pour justifier les performances de différentes implantations d'algorithmes sur GPU.

Dans notre cas, nous utiliserons l'intensité arithmétique pour concevoir l'architecture correspondant à chaque bloc étudié. Si l'intensité arithmétique est faible, il faudra privilégier une optimisation des accès mémoire en répartissant par exemple les données dans plusieurs mémoires différentes. Inversement, si l'intensité arithmétique est forte, on pourra se consacrer à la parallélisation des calculs. Cependant, la conception des

blocs matériels devra réaliser un compromis entre ces deux optimisations.

4.4.9 Définitions des blocs matériels

Dans ce paragraphe, nous allons définir et évaluer l'ensemble des blocs matériels nécessaire au FastSLAM. Nous commencerons au paragraphe 4.4.9.1 par l'étape de prédiction puis l'étape d'estimation (paragraphe 4.4.9.2)

4.4.9.1 Prédiction

Ce bloc (BF1) met à jour la position de chaque particule en fonction des données proprioceptives. Il est séparé en deux sous parties : le calcul des déplacements et la mise à jour de la position des particules.

Calcul de δs et $\delta\theta$

Traitement réalisé Ce bloc calcule, pour chaque particule, le déplacement longitudinal (δs) et rotation ($\delta\theta$). Il utilise les données proprioceptives (nt_r et nt_l) ainsi que le rayon de la roue (r) et l'entraxe (e). Les déplacements sont calculés, pour chaque particule, en utilisant la formule :

$$\begin{aligned} \delta s &= \frac{\Pi R_b(nt_{l,b} + nt_{r,b})}{N_{pt}} \\ \delta\theta &= \frac{\Pi R_b(nt_{l,b} - nt_{r,b})}{e_b N_{pt}} \end{aligned} \tag{4.17}$$

avec :
- R_b : le rayon bruité.
- e_b : la dimension de l'entraxe bruitée.
- N_{pt} : le nombre de pas / tour de roue d'un odométre.
- $nt_{l,b}$ et $nt_{r,b}$: le nombre de pas parcouru bruité. Chaque nombre de pas est codé sur un entier signé de 16 bits.

Chaque particule est associée à un rayon de roue et une taille d'entraxe bruité. Le calcul du rayon et de l'entraxe bruités est réalisé à la création de la particule. De plus, on réalise un précalcul des coefficients $c_1 = \Pi R_b/N_{pt}$ et $c_2 = \Pi R_b/(e_b N_{pt})$ permettants d'éviter des calculs dont une division coûteuse en temps de calcul. Les coefficients sont stockés en entiers non signés sur 16bits, les simulations du paragraphe 4.6.1 valideront ce choix.

La formule devient :

$$\begin{aligned} \delta s &= c_1(nt_{l,b} + nt_{r,b}) \\ \delta\theta &= c_2(nt_{l,b} - nt_{r,b}) \end{aligned} \tag{4.18}$$

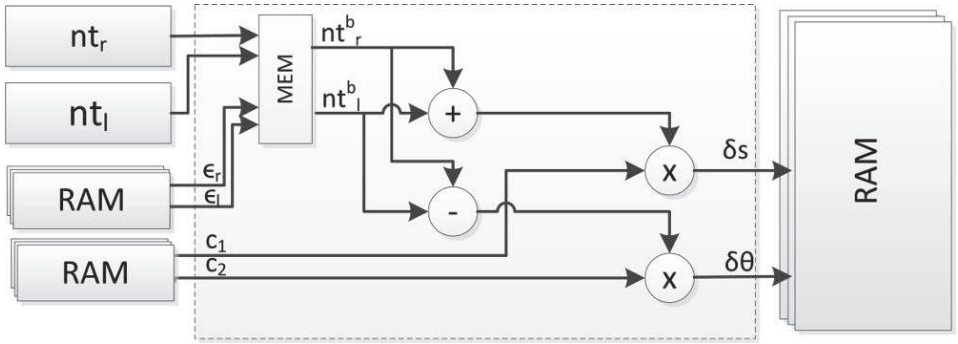


Figure 4.14: *Architecture du bloc fonctionnel "Calcul déplacement"*

On remarque que les deux calculs sont réalisables en paralléle, il n'y a pas de dépendance entre les deux calculs. Le facteur limitant est le temps d'accès à la mémoire.

Architecture proposée La figure 4.14 représente l'architecture proposée. Les bruits sont générés aléatoirement puis stockés dans une LUT. Après avoir bruité les données odométriques, les variables sont combinées pour obtenir les déplacements longitudinals et rotationels. Ce traitement est réalisé indépendemment pour chaque particule, l'ensemble des calculs est donc pipeliné.

Analyse fonctionnelle Pour évaluer les résultats issus du bloc, on tire aléatoirement des valeurs :

- $nt_r, nt_l \in [-1000, 1000]$.
- $r \in [0.01, 0.05]$.
- $e \in [0.1, 0.2]$.
- $N_{pt} \in [500, 4000]$.

Après traitement, on compare les données de sortie δs et $\delta\theta$ avec les valeurs réelles issues du calcul en double précision. Le tableau 4.3 présente les résultats issus d'une simulation de 500 tests. Les erreurs moyennes mais aussi maximales et minimales sont très faibles. L'erreur maximale pour le déplacement longitudinal est de l'ordre du millimètre. Elle est de l'ordre du dixième de radian pour le déplacement rotationnel. Les erreurs moyennes étant proches de zéro, on peut considérer que les résultats issus de ce bloc ne sont pas biaisés.

Variables	Erreur moyenne	Ecart type	Erreur Min	Erreur Max
δs	$1,43 \times 10^{-5}$ m	0,00024 m	−0,0018 m	0,0030 m
$\delta\theta$	$-2,12 \times 10^{-5}$ rad	0,0022 rad	−0,0181 rad	0,0325 rad

Table 4.3: *Résultats du bloc "Calcul de δs et $\delta\theta$"*

Analyse temporelle Le bloc est implémenté sur un FPGA (Cyclone II EP2C70). Expérimentalement, les coefficients sont stockés dans une mémoire SDRAM et les résultats dans une mémoire RAM. Le tableau 4.4 représente l'intensité arithmétique du

bloc. Pour ce bloc, elle est de 2 : il y a deux fois plus d'opérations que d'accès mémoires. Cette intensité est relativement faible, ce qui va restreindre l'efficacité de notre architecture parallèle.

	Nombre d'opérations					Nombre d'accès mémoire	
	ADD	SUB	MUL	DIV	SHIFT	LOAD	STORE
	1	1	2	0	4	2	2
	8					4	
IA	2						

Table 4.4: *Intensité Arithmétique pour le bloc de prédiction*

En implantation logicielle, sur le NIOS2, ce bloc consomme 52 cycles par itération et par particule. Ce nombre est réduit à 6 avec notre implantation matérielle. Le gain est de 8.6 fois.

Modèle de déplacement

Traitement réalisé Ce bloc (BF2) intégre les données odométriques pour calculer la position du mobile. Les positions des particules (x^i_{k-1}, y^i_{k-1}, θ^i_{k-1}) sont mises à jour à l'aide du modèle de déplacement et des données issues du bloc "Calcul de δs et $\delta\theta$". La fonction réalisée est basée sur le modèle :

$$\mathbf{f}(\mathbf{x}_{k-1}, \delta s_k, \delta\theta_k) = \begin{pmatrix} x_{k-1} + \delta s. \cos\left(\theta_{k-1} + \frac{\delta\theta}{2}\right) \\ y_{k-1} + \delta s. \sin\left(\theta_{k-1} + \frac{\delta\theta}{2}\right) \\ \theta_{k-1} + \delta\theta \end{pmatrix} \tag{4.19}$$

En sortie de ce bloc, on obtient la position de chaque particule intégrant les données odométriques.

Architecture proposée La figure 4.15 représente l'architecture proposée. Toutes les particules sont stockées dans une seule mémoire. Les déplacements δs et $\delta\theta$ sont enregistrés dans une mémoire tampon pour réaliser la transition entre les deux blocs de traitement. Une LUT (Look Up Table) est utilisée pour calculer le sinus et le cosinus nécessaires au modèle de déplacement. Les données en sortie sont stockées dans une troisième mémoire. L'utilisation de plusieurs mémoires permet de charger ou de sauvegarder simultanément les données. De plus, les données relatives aux particules sont sauvegardées de manière continue en mémoire. La carte de développement utilisée intégre de la mémoire SDRAM, optimisée pour la lecture de données contigus.

Analyse fonctionnelle Pour évaluer les résultats issus du bloc, on génére aléatoirement les valeurs suivantes :

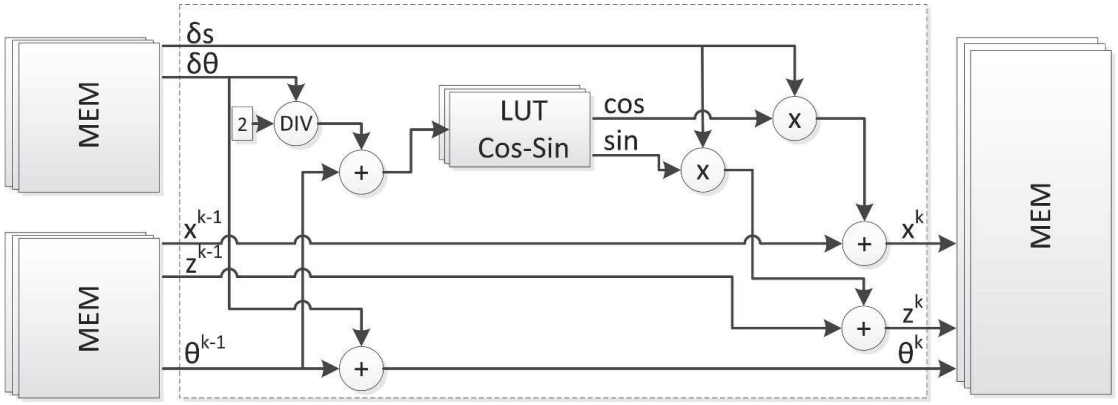


Figure 4.15: *Architecture du bloc fonctionnel "Modèle déplacement"*

- $x^i_{k-1}, y^i_{k-1} \in [-10, 10]$.
- $\theta^i_{k-1} \in [0, 6.28]$.
- $\delta s \in [-2, 2]$.
- $\delta\theta \in [-0.5, 0.5]$.

Après traitement, on compare les nouvelles positions des particules (x_k, y_k, θ_k) avec les valeurs réelles issues du calcul en flottant double précision. Le tableau 4.5 résume les résultats issus d'une simulation de 500 tests. Le déplacement longitudinal maximum est de 2 m, alors que l'erreur maximale commise est au alentour de 3 mm. Cette erreur semble relativement faible par rapport aux grandeurs mesurées. De même, pour l'orientation, les erreurs commises sont très faibles ($<10^{-5}$ rad).

Variables	Erreur moyenne	Ecart type	Erreur Min	Erreur Max
x_k	$6,60 \times 10^{-5}$ m	0,0010 m	$-0,0031$ m	0,0030 m
y_k	$9,90 \times 10^{-4}$ m	0,0010 m	$-0,0038$ m	0,0022 m
θ_k	$4,90 \times 10^{-6}$ rad	$6,43 \times 10^{-6}$ rad	$-8,86 \times 10^{-6}$ rad	0,00002 rad

Table 4.5: *Résultats du bloc "Modèle de déplacement"*

Analyse temporelle Expérimentalement, les données des particules d'entrées sont stockées dans la mémoire principale de l'architecture utilisée de type SDRAM. Les déplacements sont enregistrés dans une mémoire tampon entre les deux blocs de traitement. Les nouvelles positions des particules sont sauvegardées dans une troisième mémoire de type SSRAM. L'intensité arithmétique du bloc, détaillée dans le tableau 4.6, est de 2. Les performances obtenues sur notre architecture est de 109 cycles par particules par itération pour une implantation avec un softcore NIOS2 et 16 cycles pour une implantation matérielle, le gain est de 6.81 fois.

	ADD	SUB	MUL	DIV	SHIFT	LOAD	STORE
Nombre	6	0	7	0	3	5	3
Nombre	16					8	
IA	2						

Table 4.6: *Intensité arithmétique pour le bloc numéro*

4.4.9.2 Estimation

L'estimation est composée de plusieurs blocs. Elle comporte l'initialisation d'amers, leurs estimations ainsi que leurs projections. De plus, on inclut la gestion des particules (mise à jour des poids et rééchantillonage).

Initialisation d'un amer

Traitement réalisé L'initialisation d'un amer (BF3) a lieu lors de la première observation par le mobile. Elle consiste à définir les paramètres de l'amer : x_i, y_i ,θ_i, ϕ_i et ρ_i et sa matrice de covariance associée. La matrice de covariance de l'amer est initialisée lors de l'initialisation du FPGA car elle ne dépend d'aucune variable. Pour un nouvel amer ayant été observé en position (u, v), les paramètres sont initialisés par :

- $x_i = x_k^i$, $y_i = y_k^i$: la position de la particule i.
- $\theta_i = \theta_k^i + u_{angle}$ avec $u_{angle} = -\arctan(\frac{u-c_u}{fk_u})$.
- $\varphi_i = \arctan(\frac{(v-c_v)}{fk_u\sqrt{\frac{(v-c_v)^2}{fk_v{}^2}+1}})$.
- $\rho_i = 0.26$.

Trois variables (x_i, y_i, ρ_i) sont initialisées sans calcul à l'aide d'une copie de variables ou d'une initialisation par une constante. La composante θ_i dépend de θ_k^i et de u_{angle}. La variable u_{angle} dépend uniquement de l'observation u. L'initialisation de φ_i dépend de deux variables : v et u_{angle}.

Architecture proposée Le bloc "Initialisation d'un amer" ne nécéssite pas de calculs intensifs : seulement deux variables doivent être calculées. Le résultat u_{angle} ne dépend que de la variable u. Cette variable a des valeurs comprises entre 0 et 639 car l'image a une largeur de 640 pixels. Pour accélérer le traitement, on crée une LUT contenant le résultat de u_{angle}. Pour limiter la taille de la LUT, on pourra utiliser la propriété de l'arctangente : $\arctan(x) = \arctan(-x)$.
Le paramètre φ_i dépend de deux variables : u et v. Il est calculé à l'aide de $\arctan(X)$ avec $X = \frac{(v-c_v)}{fk_u\sqrt{\frac{(v-c_v)^2}{fk_v{}^2}+1}}$. La valeur de X est bornée car les valeurs de u et v sont bornées ($u \in [0;639]$ et $v \in [0;479]$, l'image a une taille de 640×480 pixels). Après avoir calculé les valeurs minimale et maximale de X, on réalise une LUT de 4096 valeurs permettant un temps de traitement très court. De plus, φ_i étant estimé par le filtre de Kalman, une petite erreur de calcul n'influencerait que très peu le résultat.
Enfin, il est remarquable que les valeurs issues des deux LUT soient identiques pour chaque particule, la calcul ne doit être réalisé qu'une seule fois. Ce bloc n'inclut qu'une simple addition.

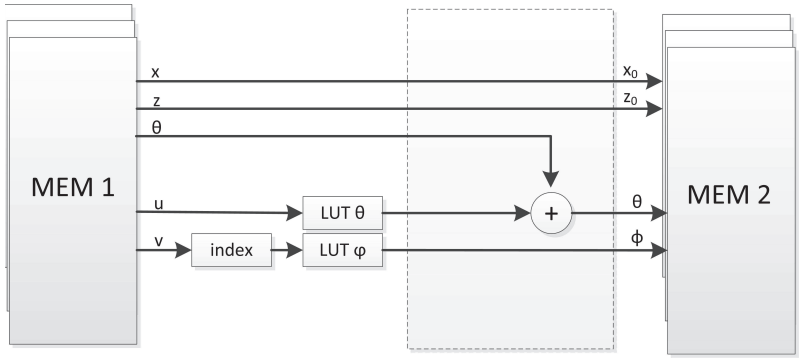


Figure 4.16: *Architecture du bloc fonctionnel "Initialisation d'un amer"*

Analyse fonctionnelle Pour évaluer les résultats issus du bloc, on génère aléatoirement des valeurs :

– $u \in [0, 639]$.

– $v \in [0, 479]$.

puis on calcule les paramètres θ_i et φ_i. Après avoir calculé ces paramètres, on les compare avec les valeurs de référence issues de l'algorithme initial. Le tableau 4.7 résume les résultats issus de cette simulation. L'ensemble des erreurs est très faible. Les erreurs maximale et minimale sont de l'ordre du millième de radian ce qui semble tout à fait raisonnable comme erreur.

Variables	Erreur moyenne	Ecart type	Erreur Min	Erreur Max
θ_i	$-5,37 \times 10^{-6}$ rad	$6,90 \times 10^{-6}$ rad	$-1,97 \times 10^{-5}$ rad	$8,97 \times 10^{-6}$ rad
φ_i	$-0,00035$ rad	$0,00025$ rad	$-0,00084$ rad	$0,00020$ rad

Table 4.7: *Résultats du bloc "Initialisation d'un amer"*

Analyse temporelle Expérimentalement, lors d'une initialisation, les paramètres des particules seront stockés dans la mémoire SSRAM alors que les données de la carte seront dans la mémoire SDRAM. L'intensité arithmétique du bloc de traitement est de seulement 0.714, notre implantation matérielle apporte un gain de 3.18, le temps de traitement est réduit de 86 cycles par itération à 27. Notre optimisation a été limitée par les temps d'accès à la mémoire.

	ADD	SUB	MUL	DIV	LOAD	STORE
Nombre	3	0	2	0	4	3
Nombre	5				7	
IA	0.714					

Table 4.8: *Intensité Arithmétique du bloc d'initialisation*

Projection d'un amer

Traitement réalisé La projection d'un amer (BF4) est utilisée pour calculer la prédiction de l'observation. Cette position permet au filtre de Kalman de calculer l'inno-

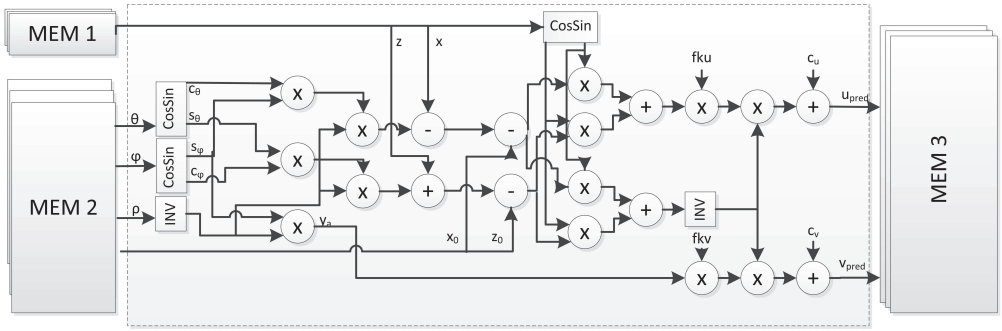


Figure 4.17: *Architecture du bloc fonctionnel "Projection d'un amer"*

vation. La projection est réalisée grâce au modèle Pinhole suivant :

$$\begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} c_u + fk_u \frac{x_{i,cam}}{z_{i,cam}} \\ c_v + fk_v \frac{y_{i,cam}}{z_{i,cam}} \end{pmatrix} \tag{4.20}$$

avec :

- (u, v) la position de l'amer sur l'image.
- $(x_{i,cam}, y_{i,cam}, z_{i,cam})$ la position de (u, v) dans le repère caméra.
- fk_u, fk_v, c_u, c_v des constantes modélisants la caméra.

Les coordonnées des amers ne sont pas définies dans le repère caméra mais dans le repère global par cinq paramètres. Il faut tout d'abord calculer la position de l'amer dans le repère global :

$$\begin{pmatrix} x_{i,glob} \\ y_{i,glob} \\ z_{i,glob} \end{pmatrix} = \begin{pmatrix} x_i \\ y_i \\ 0 \end{pmatrix} + \frac{1}{\rho_i} \mathbf{m}(\theta_i, \varphi_i) \tag{4.21}$$

Puis, un changement de repère est réalisé :

$$\begin{pmatrix} x_{i,cam} \\ y_{i,cam} \\ z_{i,cam} \end{pmatrix} = \begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) \end{bmatrix} \left(\begin{pmatrix} x_{i,glob} \\ y_{i,glob} \\ z_{i,glob} \end{pmatrix} - \begin{pmatrix} x \\ y \\ 0 \end{pmatrix} \right) \tag{4.22}$$

Après avoir calculé les coordonnées de l'amer dans le repère caméra, on utilise le modèle 4.20 pour calculer sa projection sur l'image.

Architecture proposée L'architecture proposée reprend les trois étapes de calculs. Elle calcule les coordonnées de l'amer dans le repère global en utilisant deux LUT pour éviter les calculs de cosinus/sinus. Pour éviter d'effectuer plusieurs divisions, l'inverse de ρ est calculé une seule fois puis utilisé pour les trois coordonnées. Le changement de

repère pour le repère caméra est effectué en utilisant une seconde LUT cosinus/sinus. Le modèle Pinhole est ensuite appliqué aux coordonnées. Pour éviter une division, on précalcule l'inverse de $z_{i,cam}$.

Analyse fonctionnelle Pour évaluer les résultats issus du bloc, on tire aléatoirement des valeurs :

- $x^i_{k-1}, y^i_{k-1} \in [-10, 10]$.
- $\theta^i_{k-1} \in [0, 2\pi]$.
- $x_i, y_i \in [-10, 10]$.
- $\theta_i = \theta^i_k + u_{angle}$ avec $u_{angle} = -\arctan(\frac{u-c_u}{fk_u})$.
- $\varphi_i = \arctan(\frac{(v-c_v)}{fk_u\sqrt{\frac{(v-c_v)^2}{fk_v{}^2}+1}})$.
- $\rho_i = 0.26$.

Le tableau 4.9 résume les résultats issus du bloc de projection. L'erreur moyenne est d'environ un demi pixel, son écart type est de 0.3 pixels. Ces erreurs de projection seront incluses dans le filtre de Kalman en surestimant légèrement la matrice de variance de l'innovation.

Variables	Erreur moyenne	Ecart type	Erreur Min	Erreur Max
u	$-0,66$ pixels	$0,30$ pixels	$-2,081$ pixels	$0,074$ pixels
v	$-0,52$ pixels	$0,35$ pixels	$-2,351$ pixels	$0,584$ pixels

Table 4.9: *Résultats du bloc "Projection d'un amer"*

Analyse temporelle Expérimentalement, les paramètres des particules sont stockés dans une mémoire SSRAM alors que les données de la carte sont dans une SDRAM. L'intensité arithmétique du bloc est évaluée, elle vaut 3.187 (Tableau 4.10). Cependant notre architecture logicielle bénéficie déjà de multiplieurs et diviseurs matériels. Le temps de calcul logiciel (implantation sur NIOS2) est de 416 cycles par itération par particules et le temps de calcul avec une implantation matérielle est de 61 cycles soit un gain de 6.8.

	ADD	SUB	MUL	DIV	SHIFT	LOAD	STORE
Nombre	5	3	13	2	28	14	2
Nombre	51					16	
IA	3.187						

Table 4.10: *Intensité Arithmétique pour le bloc de projection*

Calcul de la jacobienne

Traitement réalisé Le filtre de Kalman définit la matrice H comme la jacobienne de la fonction d'observation (BF5). L'expression de la fonction d'observation étant

compliquée, sa jacobienne est calculée à l'aide de matlab puis C2H crée son architecture. On vérifie ensuite que son exécution est correctement parallélisée.

Analyse fonctionnelle L'analyse fonctionnelle sera réalisée lors de la simulation au paragraphe 4.6.2.

Analyse temporelle L'intensité arithmétique de ce bloc est très élevée : 12. Elle permet une implantation matérielle très efficace qui réduit le temps de calcul de 1339 cycles par itération et par particule à seulement 83 cycles. Le gain est de 16.13 fois. Ceci confirme l'efficacité d'une implantation matérielle fortement parallèle lorsque l'intensité arithmétique est élevée.

	ADD	SUB	MUL	DIV	SHIFT	LOAD	STORE
Nombre	10	10	79	2	103	11	6
Nombre	204					17	
IA	12						

Table 4.11: *Intensité Arithmétique pour le bloc du calcul de la jacobienne*

Estimation de la localisation d'un amer

Traitement réalisé La localisation de chaque amer est estimée pour chaque particule. Ce bloc de calcul (BF6) est très important, il est exécuté pour chaque amer observé et pour chaque particule. Le bloc implémente les équations d'estimation du filtre de Kalman :

$$\begin{aligned} \mathbf{S}_i &= \mathbf{H}_i\mathbf{P}_i\mathbf{H}_i^T + \mathbf{R} \\ \mathbf{K}_i &= \mathbf{P}_i\mathbf{H}_i\mathbf{S}_i^{-1} \\ \mathbf{x}_i &= \mathbf{x}_i + \mathbf{K}_i\mathbf{Y}_i \\ \mathbf{P}_i &= (\mathbf{I} - \mathbf{K}_i\mathbf{H}_i)\mathbf{P}_i \end{aligned} \tag{4.23}$$

Ces calculs sont des calculs matriciels. Ils sont constitués d'une multitude de multiplications et d'additions.

Architecture proposée La figure 4.18 représente l'architecture proposée. Les données d'entrées sont issues du bloc "Projection d'un amer" et "Calcul de H" mais aussi les paramètres des amers. En sortie, le bloc met à jour les paramètres des amers ainsi que la matrice de covariance.

L'architecture du bloc a été définie par l'utilitaire C2H qui transforme le code C en bloc matériel. L'utilisation de C2H a permis de définir rapidement la structure du bloc alors que celui-ci comporte un nombre important d'opérations et d'accès mémoire. Les

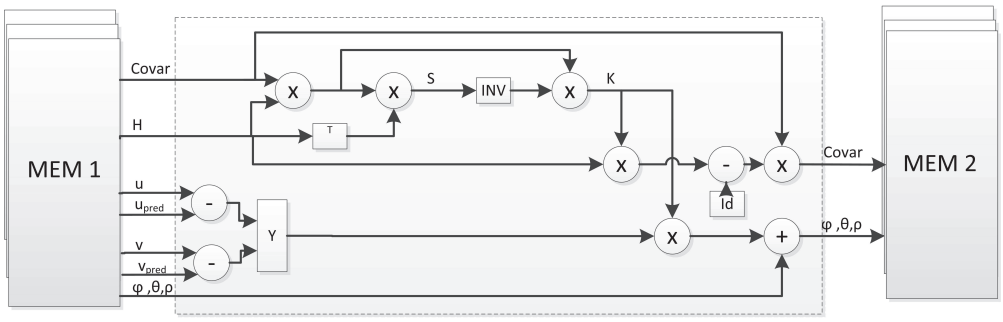


Figure 4.18: *Architecture du bloc fonctionnel "Estimation de la localisation d'un amer"*

opérateurs sont des opérateurs matriciels, sauf le calcul de l'inverse de la matrice S qui est définit par l'inverse d'une matrice de taille 2×2.

Analyse fonctionnelle L'analyse fonctionnelle sera réalisée lors de la simulation au paragraphe 4.6.2 car les résultats de ce bloc dépendent de nombreux paramètres.

Analyse temporelle L'intensité arithmétique du bloc (tableau 4.12) est élevée : elle devrait permettre un gain important lors de l'étape d'estimation. Cependant, ce bloc lit et écrit les données de la carte dans la mémoire SDRAM. Les performances logicielles sont de 2115 cycles par itération et par particule contre 238 cycles pour la version cablée : le gain est de 8.8.

	ADD	SUB	MUL	DIV	SHIFT	LOAD	STORE
Nombre	55	17	105	1	103	21	13
Nombre	281					34	
IA	8.264						

Table 4.12: *Intensité Arithmétique pour le bloc d'estimation*

Mise à jour des poids

Traitement réalisé Après avoir estimé la position d'un amer, le poids de chaque particule est mis à jour (BF7). On utilise l'équation de mise à jour suivante :

$$\rho_j = \rho_j \exp\left(\frac{-0.5 Y_i^T S_i^{-1} Y_i}{\sqrt{||S_i||}}\right) \tag{4.24}$$

Avec Y_i le vecteur d'innovation et S_i sa matrice de covariance définis précédemment lors de l'estimation du filtre de Kalman.

Pour éviter plusieurs mises à jour, on propose de n'effectuer qu'une seule fois cette étape après avoir réalisé l'ensemble des estimations. Pour l'ensemble des N observations, on obtient la mise à jour suivante :

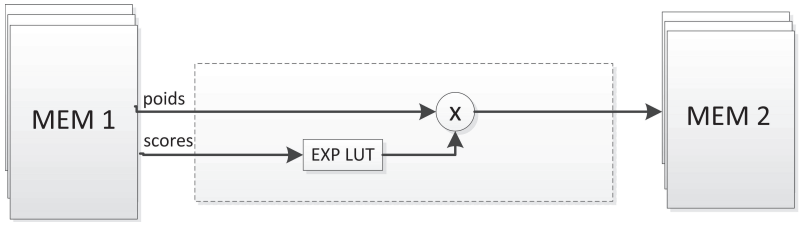


Figure 4.19: *Architecture du bloc fonctionnel "Mise à jour des poids"*

$$\rho_j = \rho_j \exp(\frac{-0.5Y_1^T S_1^{-1} Y_1}{\sqrt{||S_1||}}) \exp(\frac{-0.5Y_2^T S_2^{-1} Y_2}{\sqrt{||S_2||}}) ..... \exp(\frac{-0.5Y_N^T S_N^{-1} Y_N}{\sqrt{||S_N||}}) \quad (4.25)$$

$$= \rho_j \prod_{i=1:N} \exp(\frac{-0.5Y_i^T S_i^{-1} Y_i}{\sqrt{||S_i||}}) \quad (4.26)$$

$$= \rho_j \exp(\sum_{i=1:N} \frac{-0.5Y_i^T S_i^{-1} Y_i}{\sqrt{||S_i||}}) \quad (4.27)$$

Pour effectuer une seule mise à jour, on somme l'ensemble des scores successifs puis on met le poids à jour.

Architecture proposée La figure 4.19 schématise l'architecture générée, composée exclusivement d'une LUT et d'un multiplieur. L'utilisation d'une seule étape de mise à jour de poids permet de réduire considérablement le nombre de calculs effectués. Le gain obtenu est de 4.8 fois, avec un temps de 35 cycles par itération et par particule pour l'opérateur cablé contre 168 cycles pour l'opérateur logiciel.

Analyse temporelle L'architecture utilisée est relativement simple, les poids sont stockés avec les particules dans la mémoire SDRAM et les scores dans une RAM. L'intensité arithmétique est faible : seulement 0.75 (Tableau 4.13)

	ADD	SUB	MUL	DIV	LOAD	STORE
Nombre	1	0	2	0	3	1
Nombre	3				4	
IA	0.75					

Table 4.13: *Intensité arithmétique du bloc de mise à jour du poids des particules*

Rééchantillonnage

Traitement réalisé Après avoir mis à jour le poids des particules, nous supprimons les particules dont la propabilité est trop faible (BF8). Pour supprimer une particule, on la remplace par une copie de la particule ayant le poids le plus élevé et on divise le poids de ces deux particules par deux. Enfin, on met à jour les données de la carte de

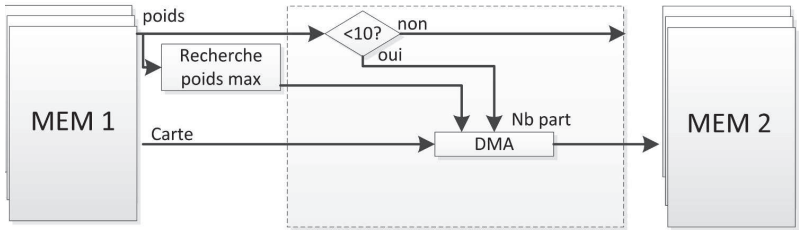


Figure 4.20: *Architecture du bloc fonctionnel "Rééchantillonnage"*

la nouvelle particule en copiant ceux de la carte de la particule ayant le poids le plus élevé.

Architecture proposée La figure 4.20 représente l'architecture mise en place. Pour éviter de rechercher le poids maximum à chaque fois qu'une particule doit être remplacée, on recherche les M maximas de la liste en une seule fois. On remplacera ensuite au maximum M particules. Cette recherche limite les accès à la mémoire.

Analyse fonctionnelle L'analyse fonctionnelle de ce bloc sera réalisée en même temps que l'intégralité de l'algorithme au paragraphe 4.6.3.

Analyse temporelle Ce bloc effectue une seule opération arithmétique sur les données, son intensité arithmétique est proche de 0. Cependant l'implantation hardware crée un accès direct à la mémoire ce qui permet une implantation efficace. La copie d'une carte de 100 amers prend 22000 cycles. Cependant, la recherche de la particule de poids maximum augmente encore le temps de traitement. Le temps de recherche pour un maximum dans une liste de 250 particules est de 193 cycles (969 cycles pour la version logicielle). Cependant, la version cablée recherche les 10 maximas en seulement 385 cycles (2260 cycles pour la version logicielle). Le gain de temps entre la recherche des 10 maximas contre 10 recherches d'un maximum est un coefficient 5.

4.5 Architecture globale

Le schéma 4.21 représente l'architecture proposée, en incluant les différents blocs fonctionnels, les mémoires tampons et les LUT. Quatre grands blocs sont représentés : la prédiction, l'estimation, l'échantillonnage et l'initialisation. Chaque bloc nécessite les résultats des autres blocs. Ils ne peuvent pas opérer parallélement.
En effet, les données odométriques permettent la mise à jour de la position des particules. Cette position est ensuite utilisée par les blocs fonctionnels de la phase d'estimation. Lors de l'observation d'un amer, la position estimée est projetée pour chaque particule puis son estimation est mise à jour. Enfin le poids de chaque particule est actualisé et les particules ayant une probabilité faible sont remplacées.

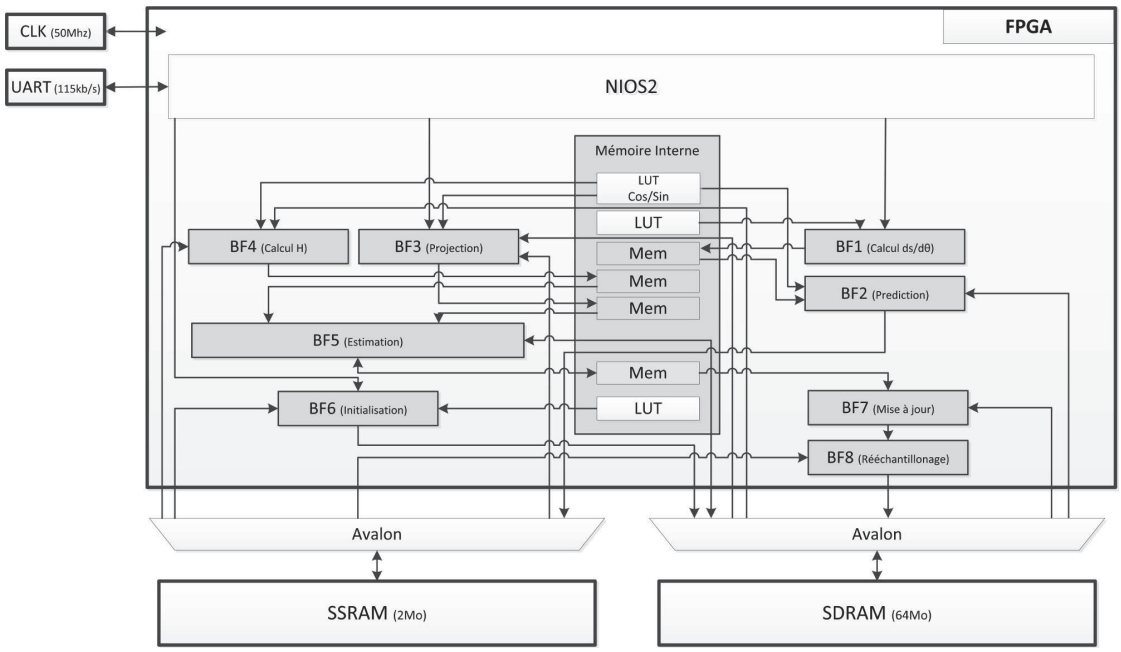


Figure 4.21: *Architecture proposée*

4.6 Validation Hardware In the Loop

L'algorithme a été validé pour le fonctionement en double précision. L'implantation optimisée a nécessité l'utilisation de variables entières à la place de variables flottantes doubles précisions ainsi que l'utilisation de LUT. Reste à vérifier si cette modification n'engendre pas de dégradation des résultats de localisation. On vérifie donc les résultats issus de notre architecture à l'aide de notre module HIL. Cette validation est réalisée sur l'environnement 2 (Paragraphe 2.4.1.3), on compare les résultats issus de notre architecture et ceux de l'implantation double précision.

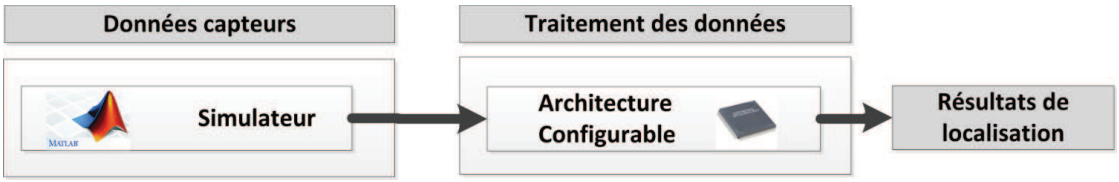


Figure 4.22: *Validation Hardware In the Loop*

4.6.1 Etape de prédiction

L'étape de prédiction intègre les déplacements du mobile en déplaçant le nuage de points. L'incertitude des données odométriques imposent la dispersion du nuage au court du temps. La figure 4.23 représente les couloirs d'incertitude issus des deux implantations. Tout d'abord, on remarque que les résultats de l'implantation en entier et double précision sont similaires. Les deux courbes sont presque chevauchées. De plus, la courbe bleue (précision entière) englobe la courbe rouge (double précision) : l'ensemble

défini par la double précision est inclu dans l'ensemble défini par l'implantation cablée. L'étape de prédiction est correctement réalisée, elle est légèrement plus pessimiste mais reste très approchante des résultats en nombres flottants.

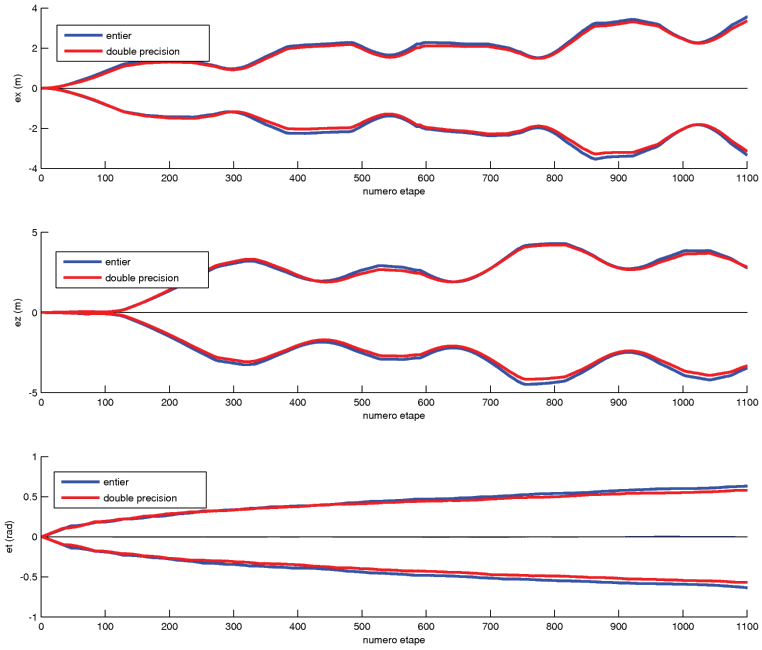


Figure 4.23: *Couloir d'incertitudes issus de la phases de prédiction - Comparaison avec l'implantation en double précision*

4.6.2 Etape d'estimation

Pour évaluer l'étape d'estimation, on réalise une simulation sans aucun bruit odométrique pour pouvoir quantifier les erreurs de l'estimation de la localisation des amers. Ainsi, l'algorithme n'utilise qu'une seule particule qui représente la position réelle du mobile simulé. Le filtre effectue l'estimation des amers qui est ensuite comparée avec les résultats issus de l'estimation en double précision. La figure 4.24 représente l'erreur euclidienne commise par l'algorithme sur la position des amers. L'analyse globale confirme que la différence entre les deux filtres est relativement restreinte. Cependant, l'erreur moyenne est tout de même légèrement plus élevée pour le filtre utilisant des données entières.

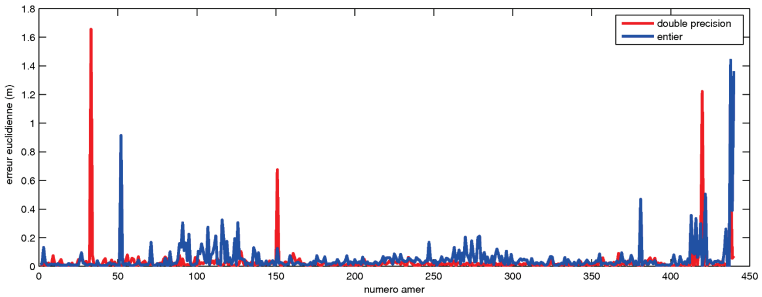


FIGURE 4.24: *Erreur euclidienne sur la position des amers - Comparaison avec l'implantation en double précision*

4.6.3 FastSLAM

Après avoir étudié l'étape de prédiction puis l'étape d'estimation, il faut analyser les résultats de l'algorithme complet incluant l'étape de mise à jour des poids et de rééchantillonnage. La figure 4.25 représente l'erreur euclidienne commise sur la position du mobile. Les erreurs issues des deux implantations sont très similaires : la différence de localisation ne justifie pas l'utilisation de données en double précision. L'utilisation de nombres entiers n'engendre pas une hausse importante de l'erreur de localisation. La figure 4.26 représente les couloirs d'incertitudes des deux implantations. Ces couloirs confirment les conclusions précédentes : le filtre ne peut pas être consistant sur le long terme. Après la fermeture de boucle de l'étape 500, les deux filtres deviennent inconsistants : la position réelle n'est pas incluse dans l'incertitude de localisation. Il n'y a pas de différence remarquable entre les deux implantations.

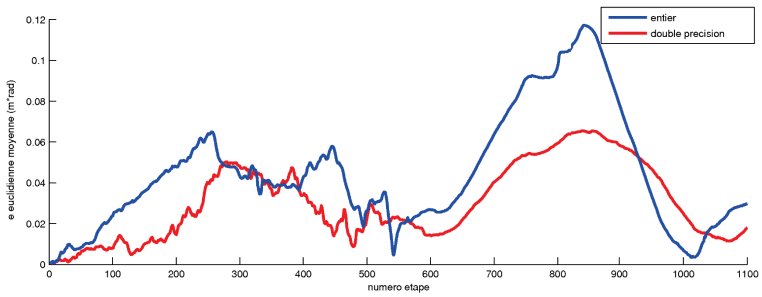
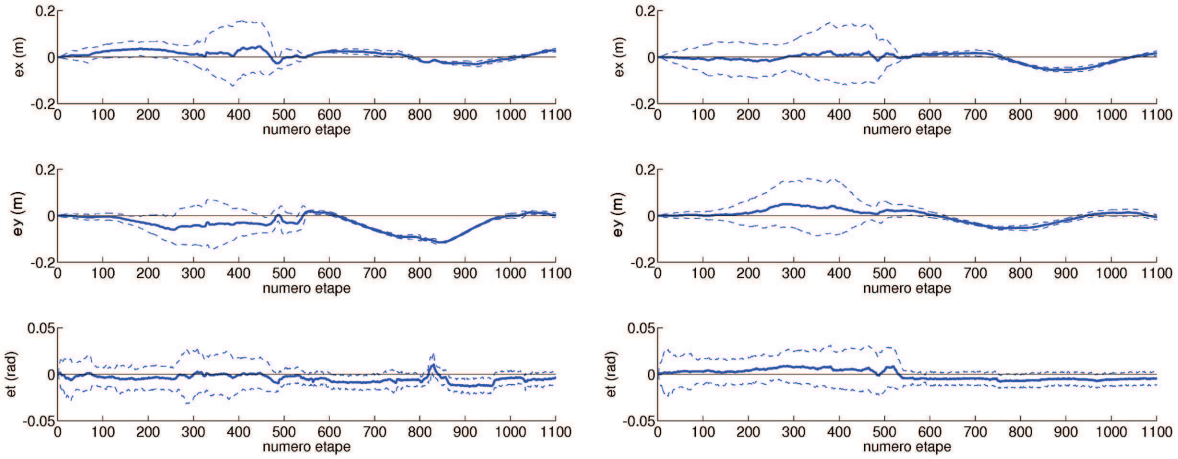


FIGURE 4.25: *Erreur euclidienne sur la position du mobile - Comparaison avec l'implantation en double précision*



(a) *Résultats issus de l'implantation en entier* (b) *Résultats issus de l'implantation en double précision*

Figure 4.26: *Couloir d'incertitude sur la position du mobile - Comparaison avec l'implantation en double précision*

4.7 Evaluation de Performances

Les résultats de localisation issus du filtre ont été évalués pour valider l'implantation cablée. Cette implantation ne dégrade que très faiblement les résultats de localisation. L'utilisation de nombres entiers a permis de définir une architecture matérielle, synthétisable sur FPGA. Nous allons résumer les performances de cette architecture puis nous étudierons les résultats de notre implantation optimisée sur les architectures utilisées précédemment : l'OMAP3530 et l'OMAP4430. L'utilisation de nombres entiers et de LUT permet d'obtenir des temps de calculs restreints par rapport à l'utilisation d'une implantation logicielle.

4.7.1 Evaluation des Performances de l'Architecture Proposée

Le tableau 4.14 résume les gains obtenus par l'architecture proposée comparés à une implantation logicielle sur un NIOS2. Les gains obtenus sont considérables. Le gain maximum est de 16 fois pour le bloc de calcul de la jacobienne. En majorité, les blocs ayant une intensité arithmétique forte bénéficient fortement de leur architecture matérielle. Plus l'intensité arithmétique est importante, plus le gain lors de l'utilisation d'une architecture matérielle est important. Cependant, le nombre de LU (Logical Unit) utilisé est assez important pour les deux blocs principaux (Calcul de H et Estimation).

Le tableau 4.15 représente le gain temporel obtenu pour une expérimentation. Globalement, on obtient un temps de calcul divisé par 9.2 par rapport à la version logicielle.

Blocs Fonctionnels	Intensité arithmétique	cycle /iter /part (SOFT)	cycle /iter /part (HARD)	Nombre de LU utilisé	Gain
1. Calcul δs et $\delta\theta$	2	42	6	1496	7
2. Modèle de déplacement	2	109	16	1333	6.8
3. Initialisation d'un amer	0.71	86	27	1241	3.1
4. Projection	3.187	416	61	6094	6.8
5. Calcul de **H**	12	1339	83	10176	16.1
6. Estimation	8.26	2115	238	13092	8.8
7. Mise à jour	0.75	168	35	2745	4.8
8. Rééchantillonnage	0	22000	22000	2039	1

Table 4.14: *Analyse des performances temporelles de l'architecture définie*

Blocs Fonctionnels	Temps moyen /iter (SOFT, en μs)	Temps moyen /iter (HARD, en μs)
1. Calcul δs et $\delta\theta$	168	24
2. Modèle de déplacement	436	64
3. Initialisation d'un amer	136	43
4. Projection	23460	3440
5. Calcul de **H**	51416	3187
6. Estimation	81216	9139
7. Mise à jour	6450	1344
8. Rééchantillonnage	572	572
Total :	163.8 ms	17.8 ms

Table 4.15: *Analyse des performances temporelles de l'architecture définie pour l'expérimentation pour 200 particules*

4.7.2 Evaluation des Performances sur une Architectures RISC Multiprocesseur

Dans le chapitre précédent, nous avons implanté l'EKF-SLAM sur deux architectures : l'OMAP3530 et l'OMAP4430. Ces architectures sont basées sur des architectures Cortex A8 et A9. Ces processeurs conçus par ARM sont très répandus, en particulier dans les systèmes embarqués.

On étudie le temps de calcul de chaque bloc sur les deux architectures. Le tableau 4.16 résume les différents résultats. L'utilisation de nombre entiers et de LUT a permis un gain de temps de calcul important pour les deux architectures. Ensuite, on remarque que l'architecture du Cortex A9 a de meilleurs performances que le Cortex A8 en particulier lors de l'utilisation de nombres flottants, ce gain est dû à l'absence d'unité de calcul flotant dans le Cortex A8. En nombres entiers, les différences de performance entre les deux architectures ne sont pas très importantes. Le Cortex A8 fait presque jeu égal avec le Cortex A9.

L'OMAP4430 a la particularité de disposer de deux processeurs. La parallélisation permet un gain de temps conséquent dans la majorité des cas. Lors d'un calcul rapide, la

création des threads pénalise le gain, en particulier pour le cas des nombre entiers où le temps de calcul est très faible (calcul déplacement, prédiction, initialisation, mise à jour). Cependant, il sera possible d'améliorer légèrement ces résultats en créant deux threads de calculs dès le lancement du programme.
Pour conclure, le gain de performance dû à l'utilisation de nombre entier et de LUT est très important. La différence entre les deux architectures monocore est relativement faible en nombre entier. Cependant, la possibilité d'utiliser le second cœur de l'architecture OMAP4430 permet d'obtenir de meilleurs résultats que l'OMAP3530.

Blocs Fonctionnels	OMAP3530		OMAP4430 monocore		OMAP4430 dualcore	
	cycle /iter /part		cycle /iter /part		cycle/iter /part	
	Flotant	Entier	Flotant	Entier	Flotant	Entier
1. Calcul δs et $\delta\theta$	78	38	63	30	59	40
2. Modèle de déplacement	1134	52	473	49	279	68
3. Initialisation d'un amer	1492	86	524	68	310	81
4. Projection	2219	473	1395	424	796	357
5. Calcul de **H**	2618	734	1971	714	584	466
6. Estimation	2249	903	1754	740	783	344
7. Mise à jour	862	129	274	61	244	70
8. Rééchantillonnage	5995	5995	1242	1242	1242	1242

Table 4.16: *Analyse des performances temporelle*

4.7.3 Comparaison des Résultats

Après avoir étudié les résultats des trois architectures (FPGA, Cortex A8, Cortex A9), on compare leurs trois résultats. Le tableau 4.17 résume les résultats obtenus sur les trois architectures : l'OMAP3530, l'OMAP4430 et notre architecture matérielle. Le gain obtenu est important, en particulier pour les blocs disposant d'une forte intensité arithmétique. Le seul gain négatif est pour le bloc de copie des cartes, en effet notre architecture dispose de mémoire SDRAM plus lente que la mémoire DDRAM2 disponible sur la plateforme OMAP4430. Il est indispensable d'envisager l'utilisation d'une mémoire plus rapide pour notre architecture. En effet, cette étape de copie reste une étape cruciale bien qu'elle pourrait être limitée en utilisant l'algorithme FastSLAM 2.0 Montemerlo *et al.* [109] qui permet une utilisation plus efficace de chaque particule.

Bloc Fonctionnel	Intensité arithmétique	OMAP3530 (500Mhz)	OMAP4430 (2×1Ghz)	FPGA (50Mhz)	Gain / A8	Gain / dual-A9
1. Calcul δs et $\delta\theta$	2	38	30	6	6.3	5.0
2. Modèle de déplacement	2	52	49	16	3.2	3.0
3. Initialisation d'un amer	0.71	86	68	27	3.2	2.5
4. Projection	3.18	473	357	61	7.7	5.8
5. Calcul de **H**	12	734	466	83	8.8	5.6
6. Estimation	8.26	903	344	238	3.8	1.4
7. Mise à jour	0.75	129	61	35	3.6	1.7
8. Rééchantillonnage	0	5995	1242	22000	0.3	0.1

Table 4.17: *Comparaison des performances des différentes architectures en (cycle /iter /part)*

4.8 Perspectives

4.8.1 Architecture

L'architecture proposée dispose de très bonnes performances pour les blocs fonctionnels ayant une forte intensité arithmétique et disposant de plusieurs mémoires. Elle permet de réaliser en parallèle des traitements très importants. Cette architecture est très adaptée dans le cas du filtre particulaire qui nécessite de nombreux calculs indépendants.
La principale limitation de notre système est la lenteur de notre mémoire embarquée (SDRAM). Cette limitation apparait en particulier lors de la copie des données de la cartes de l'environnement. Cependant, l'utilisation d'une mémoire plus rapide type DDR2 devrait permettre des performances satisfaisantes. Les performances souhaitées lors d'une copie sont obtenues par l'architecture OMAP4430.

4.8.2 Interfacage

Pour utiliser l'architecture définie dans un système, il est nécessaire de réaliser un interfaçage adéquat. Nous envisageons de réaliser un système complet incluant un processeur ARM interfacé avec notre architecture à l'aide du bus GPMC disponible sur les différents processeurs ARM (Figure 4.27). Ce bus de données permet une liaison très rapide entre le processeur et notre architecture. De plus, ce type d'interfaçage permet d'inclure le traitement d'image (détection et appariement) sur notre FPGA. Le

processeur ARM sera alors dédié aux traitements plus haut niveau tel que la planification de trajectoire. L'utilisation conjointe d'un processeur ARM et d'une architecture programmable permet une flexibilité importante tout en profitant de la possibilité de réaliser de nombreux calculs en parallèle.

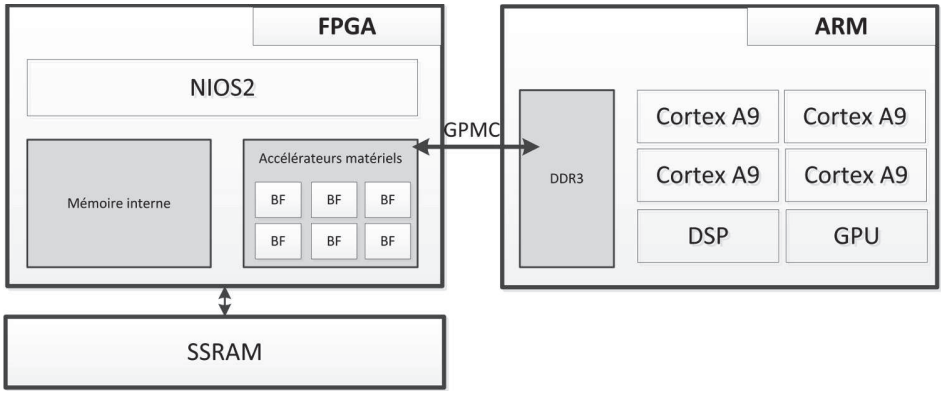


Figure 4.27: *Architecture envisageable*

4.8.3 Perspectives

La figure 4.28 représente une évolution possible de notre architecture. Actuellement, chaque bloc est parallélisé et pipeliné. Cependant, il est possible de créer plusieurs blocs matériels réalisant la même fonction. Chacun de ces blocs devra être couplé à une mémoire indépendante pour ne pas être limité par la vitesse du bus de données. L'utilisation de plusieurs blocs en parallèle nécessitera l'utilisation d'un superviseur pour assembler les résultats des différents blocs de traitement.
Par rapport à notre architecture actuelle, cette nouvelle version disposera de plusieurs mémoires dédiées permettant l'augmentation de la vitesse de traitement globale. Cependant, elle nécessitera d'utiliser un FPGA de taille bien supérieure à celui utilisé actuellement.

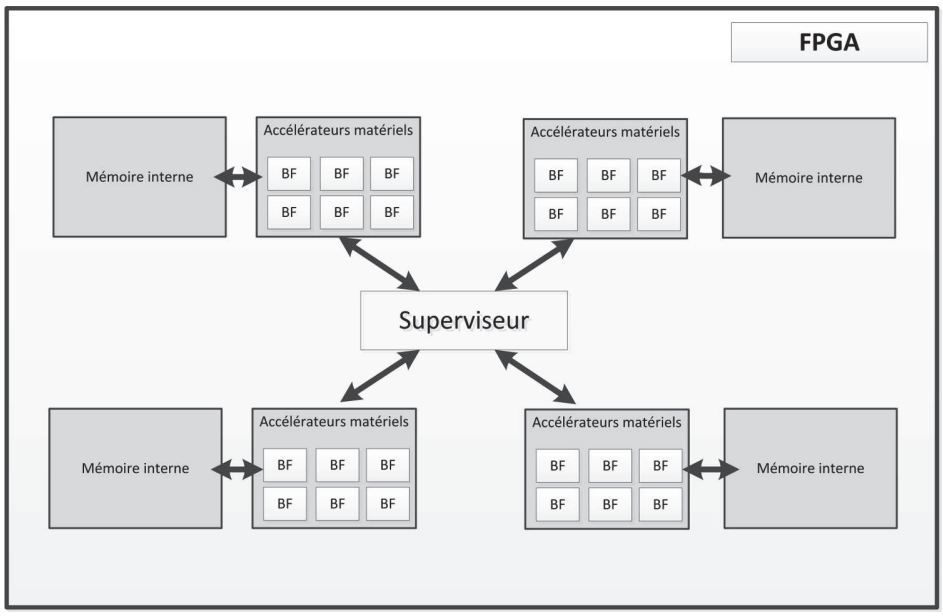


FIGURE 4.28: *Architectures envisageables*

4.9 Conclusions

Le développement d'un système dédié aux applications SLAM nécessite plusieurs étapes. En particulier, après avoir choisi les capteurs ainsi que l'algorithme, il est nécessaire de définir une implantation efficace permettant d'obtenir des résultats satisfaisants en fonction des besoins. Cette implantation peut être modulée en fonction des besoins en terme de précision ou de taille de carte par exemple.
L'utilisation du FastSLAM a permis de définir une architecture matérielle adaptée, fortement parallèle. En effet, cet algorithme réalise de nombreux traitements sans interactions permettant une implantation fortement parallèle. Cette architecture a nécessité des modifications de l'algorithme ou de son implantation. En particulier, l'utilisation de nombres entiers ou de LUT a permis d'obtenir des gains importants. Après avoir défini notre architecture, nous l'avons caractérisée temporellement et fonctionnellement puis nous avons évalué ses résultats en termes de localisation pour vérifier que notre implantation ne dégrade pas les résultats comparés à une implantation logicielle.
De nombreuses architectures embarquées existent, nous avons comparé les résultats de notre architecture par rapport à deux systèmes grand public développés par ARM. Les gains obtenus sont conséquents car les architectures embarquables actuellement ne disposent pas de la possibilité d'un fort parallélisme de calcul. Seul l'OMAP4430 dispose de deux cœurs homogènes.

Actuellement, de nombreuses architectures disposant de plusieurs cœurs de calcul sont en conception. Cette tendance permettra l'implantation efficace d'algorithmes parallélisables. Cependant, il serait profitable d'obtenir une architecture disposant de nombreux cœurs à faible fréquence ainsi que d'une mémoire séparée plutôt qu'une ar-

chitecture ayant deux ou quatre cœurs à forte fréquence et une simple mémoire centrale. La multiplication des cœurs de calcul permettrait, dans notre cas, une meilleure répartition des calculs. De plus l'utilisation de plusieurs mémoires réduira les limitations de performances dues aux accès mémoires.

Finalement, l'utilisation d'une architecture programmable associée à un processeur standard type ARM semble être, à ce jour, la meilleure solution. Les traitements non parallélisables sont réalisés à une fréquence élevée à l'aide du processeur ARM et les traitements fortement parallèlisables sont traités sur le FPGA. L'interfaçage devra être réalisé avec un bus de données très rapide pour éviter de limiter les performances à cause des accès mémoire.

Chapitre 5

Algorithmes SLAM : vers une approche ensembliste

Sommaire

5.1 Introduction

L'utilisation de la théorie probabiliste a permis de concevoir des algorithmes de SLAM ayant de bonnes performances en termes de localisation ou de temps de traitement. Ces algorithmes souffrent cependant de problèmes de consistance. En effet, l'EKF-SLAM [31] et le FastSLAM [45] produisent des résultats inconsistants sur le long terme. La position estimée par les algorithmes n'incluent pas la position réelle. De nombreuses améliorations ont été apportées pour accroître la consistance de ces algorithmes comme par exemple l'utilisation de nouvelles paramétrisations pour les amers [35], cependant ce type de méthode reste inconstant.

Nous proposons d'utiliser la théorie ensembliste pour concevoir un algorithme. Cette théorie a déjà été utilisée dans le cas de localisation de véhicule en environnement intérieur [54], extérieur [52, 53] et même dans le cas du SLAM [60, 110]. Les résultats de ces différentes approches ont toujours montré des résultats consistants à condition que les bruits soient bornés. En effet, l'hypothèse de base de l'utilisation de la théorie ensembliste est que les bruits affectant les données issues des capteurs soient bornés et de bornes connues. Nous proposons d'améliorer les algorithmes de SLAM existant en proposant une intégration améliorée des données proprioceptives, une nouvelle paramétrisation des amers, une étude expérimentale et une étude de l'impact sur l'utilisation de nouveaux capteurs.

Pour concevoir notre algorithme de SLAM ensembliste, nous allons tout d'abord faire une introduction générale à l'analyse par intervalles et à la propagation de contraintes. Puis, nous définirons l'étape de prédiction qui nous permettra d'introduire une gestion améliorée des données odométriques. Ensuite, pour pouvoir définir l'étape d'estimation, nous définirons une paramétrisation des amers adaptée à l'utilisation d'une caméra monoculaire utilisant 5 paramètres. L'algorithme ainsi défini sera évalué en simulation. Des optimisations seront ensuite proposées pour améliorer les performances de notre approche. En particulier, nous utiliserons une proposition de Joly et Rives [61] permettant d'améliorer de manière significative les résultats. L'algorithme optimisé sera alors évalué en simulation puis expérimentalement. Enfin, les conclusions issues de cette évaluation nous permettront d'émettre des propositions d'amélioration et de les valider en simulation. Ces propositions se concentrent sur l'utilisation de capteurs supplémentaires permettant d'augmenter la redondance des données.

5.2 L'analyse par intervalles

Pour représenter des nombres, les ordinateurs utilisent un système binaire dont la précision et l'étendue sont limitées par le nombre de bits sur lesquels ceux-ci sont codés. Le codage binaire rend impossible la représentation exacte de l'ensemble des nombres réels.

L'accumulation des imprécisions peut devenir, au fur et à mesure des calculs, un réel problème. L'analyse par intervalles a donc été développée par Moore [111] pour obtenir des plages de valeurs contenant obligatoirement la valeur exacte plutôt qu'une valeur approchée. Un réel est alors représenté, non plus par une valeur approchée, mais par un intervalle contenant la valeur exacte. Par exemple, pour exprimer le nombre Pi avec une précision au dixième, on ne donnera pas la valeur 3.14 mais l'intervalle [3.14,3.15].

5.2.1 Outils mathématiques d'analyse par intervalles

Notre système de reconstruction d'environnement et de localisation simulanées est appelé SLAM par méthode ensembliste à base d'analyse par intervalles et de propagation de contraintes. Deux outils sont utilisés par ce système : l'analyse par intervalles qui permet la manipulation des ensembles et la propagation de contraintes qui intègre les différentes contraintes induites par le modèle de déplacement ou d'observation.
Comme sur une dimension un réel est représenté par un intervalle, avec trois dimensions, un point ou un amer est représenté par une boîte à trois dimensions contenant sa position exacte. La localisation du mobile ainsi que la carte de l'environnement sont modélisées par des boites à trois dimensions. Chaque boite inclut de manière garantie la position réelle du robot ou d'un amer. De plus, les boites incluent les différentes erreurs de modèle pour garantir la consistance des résultats. Plus l'incertitude est importante, plus la taille de la boite sera grande.
Une particularité de l'analyse par intervalles est de considérer l'ensemble des erreurs comme étant bornées et de bornes connues.

L'utilisation de l'analyse par intervalles nécessite, bien-sûr, des opérateurs appropriés à cette représentation pour traiter les différents calculs. L'intégration des données odométriques ou caméra sous forme d'intervalles engendre des opérations entre plusieurs intervalles alors régies par les lois de l'analyse par intervalles.

Avant d'étudier notre algorithme, nous allons procéder à une courte introduction des outils d'analyses par intervalles et de la propagation de contraintes. Ces outils sont indispensables à la compréhension de la méthode.

5.2.2 Opérations sur des intervalles

Les opérations (addition, multiplication, soustraction, division) sont réalisées classiquement sur des nombres. Cependant, la méthode ensembliste doit manipuler non plus des nombres mais des intervalles.
Tout d'abord, on définit un intervalle : c'est un sous-ensemble fermé borné connexe de $\mathbb{R}$. On le note :

$$[x] = [\underline{x}, \overline{x}] = \{x \in \mathbb{R} | \underline{x} \leq x \leq \overline{x}\} \tag{5.1}$$

où $\underline{x}$ et $\overline{x}$ sont respectivement sa borne inférieure et supérieure.

Les intervalles étant des ensembles, les notions d'égalité, d'appartenance, d'inclusion et d'intersection sont définies. La réunion de deux intervalles est problématique puisqu'elle n'est en général pas un intervalle. On réalise alors l'union convexe de deux intervalles $\underline{\cup}$ correspondant au plus petit intervalle contenant l'union de ces deux intervalles :

$$[x]\underline{\cup}[y] = [\min([x],[y]), \max([x],[y])] \tag{5.2}$$

On étend les opérations usuelles comme l'addition, la soustraction, la multiplication et la division à l'analyse par intervalles :

- Addition : $[x] + [y] = [\underline{x} + \underline{y}, \overline{x} + \overline{y}]$
- Soustraction : $[x] - [y] = [\underline{x} - \overline{y}, \overline{x} - \underline{y}]$
- Multiplication : $[x] \times [y] = [\min(\underline{x}.\underline{y}, \underline{x}.\overline{y}, \overline{x}.\underline{y}, \overline{x}.\overline{y}), \max(\underline{x}.\underline{y}, \underline{x}.\overline{y}, \overline{x}.\underline{y}, \overline{x}.\overline{y})]$
- Division : si $0 \notin$ [y], $[x]/[y] = [\underline{x}] \times [1/\overline{y}, 1/\underline{y}]$

5.2.3 Fonctions d'inclusion

Pour n'importe quelle fonction f, l'analyse par intervalles définit la fonction d'inclusion $f_{[]}$ dont le résultat est un intervalle englobant de l'ensemble résultat de f. C'est à dire :

$$\forall x \in D,\ f([x]) \subset f_{[]}([x]) \tag{5.3}$$

La méthode la plus simple pour obtenir cette fonction d'inclusion est de remplacer l'ensemble des variables par leurs intervalles. Par exemple, pour la fonction suivante :

$$\forall x \in D,\ f(x) = x^2 + x \tag{5.4}$$

On définit la fonction d'inclusion comme étant :

$$\forall x \in D,\ f_{[]}(x) = [x]^2 + [x] \tag{5.5}$$

Dans la fonction précédente, chaque variable a été remplacée par un intervalle. La fonction d'inclusion calcule l'image d'un intervalle par une fonction.

5.2.4 Librairies

Le calcul par intervalles est simplifié par l'utilisation de librairie telle que CXSC qui permet de manipuler les intervalles comme des nombres réels. Toutes les fonctions mathématiques de base sont mises en œuvre pour accepter des nombres réels ainsi que des intervalles.

5.3 La propagation de contraintes

La propagation de contraintes a été développée dans les années 1970 par Waltz [112] et principalement utilisée en intelligence artificielle [113]. Elle consiste à réduire le domaine de définition d'une variable tout en maintenant l'ensemble des valeurs possibles cohérent avec l'ensemble des contraintes du problème.

La propagation de contraintes a été utilisée conjointement avec l'analyse par intervalles à la fin des années 80 [114]. Elle a été appliquée à la localisation de véhicule en environnement extérieur par Gning et Bonnifait [115]. L'approche consiste à définir un ensemble de contraintes qui est ensuite résolu en utilisant la propagation de contraintes, couplée à l'analyse par intervalles. Ce système permet d'obtenir une localisation garantie si les hypothèses réalisées sur les bruits affectant les données sont respectées.

5.3.0.1 Définition d'une contrainte

Dans le cadre de l'analyse par intervalles, une contrainte C_i est une relation arithmétique qui lie plusieurs intervalles $[a], ..., [z]$, elle est définie par :

$$C_i([a], ..., [z]) = 0 \tag{5.6}$$

5.3.0.2 Contraction

Une contrainte peut être utilisée pour réduire la taille des intervalles. Prenons, par exemple, la contrainte reliant les trois intervalles $[x], [y], [z]$:

$$[z] = [x] + \ln([y]) \tag{5.7}$$

L'utilisation de la contrainte permet de réduire le domaine de définition de chaque intervalle. Il est parfois nécessaire de réaliser plusieurs itérations consécutives pour réduire au maximun le domaine de définition de chaque intervalle. Ces contractions successives sont réalisées tant qu'une itération réduit au moins un intervalle.

Numériquement, on pose $x \in [1;10]$, $y \in [3;50]$, $z \in [-5;5]$, en utilisant la contrainte définie précédemment, les intervalles contractés sont :

$$\begin{aligned}
[z] &\in [z] \bigcap ([x] + ln([y])) \\
&\in [-5;5] \bigcap ([1;10] + ln([3;50])) \\
&\in [-5;5] \bigcap [2.09;13.92] \\
&\in [2.09;5]
\end{aligned} \tag{5.8}$$

$$\begin{aligned}
[x] &\in [x] \bigcap ([z] - ln([y])) \\
&\in [1;10] \bigcap ([2.09;5] - ln([3;50])) \\
&\in [1;10] \bigcap [-1.92;3.91] \\
&\in [1;3.91]
\end{aligned} \tag{5.9}$$

$$\begin{aligned}
[y] &\in [y] \bigcap exp([z] - [x]) \\
&\in [3;50] \bigcap exp([2.09;5] - [1;3.91]) \\
&\in [3;50] \bigcap [0.14;408400] \\
&\in [3;50]
\end{aligned} \tag{5.10}$$

Après contractions, on obtient les intervalles $x \in [1;3.91]$, $y \in [3;50]$, $z \in [2.09;5]$. Les intervalles résultats sont de taille inférieure ou égale à leur taille initiale. On remarque que, dans notre cas, il n'est pas nécessaire d'effectuer plusieurs étapes de contractions, les intervalles ne peuvent plus être réduits après la première étape de propagation.

5.3.0.3 Propagation de contraintes

L'objectif d'un problème de satisfaction de contraintes (CSP, Constraints Satisfaction Problem) [58] est d'obtenir les plus petits intervalles satisfaisants un ensemble de contraintes. La solution sera l'ensemble des plus petits intervalles satisfaisants toutes les contraintes.

Un CSP est résolu en réalisant des contractions successives des boîtes initiales. Les boîtes résultats contiennent l'ensemble des points satisfaisants l'ensemble des contraintes. Ces contractions peuvent être réalisées par une propagation Forward/Backward [58, 115]. Les contraintes sont écrites sous la forme $f([x]) = [y]$ où $[x]$ et $[y]$ peuvent

être mesurées. Puis, les contraintes sont propagées de y à x dans une première étape et, de x à y dans une seconde étape.

Par exemple, nous ajoutons une seconde contrainte à notre exemple précédent, le CSP devient :

$$\begin{cases} [z] = [x] + \ln_{[]}([y]) \\ [y] = [z]^2 \end{cases} \tag{5.11}$$

La contraction est réalisée en deux étapes :

1. La propagation Forward qui réduit les termes à gauche de l'équation (5.11)

$$[z] \in [z] \bigcap ([x] + \ln_{[]}([y])) \tag{5.12}$$
$$[y] \in [y] \bigcap [z]^2 \tag{5.13}$$

2. La propagation Backward qui réduit les termes à droite de l'équation (5.11)

$$[x] \in [x] \bigcap ([z] - \ln_{[]}([y])) \tag{5.14}$$
$$[y] \in [y] \bigcap \exp_{[]}([z] - [x]) \tag{5.15}$$
$$[z] \in [z] \bigcap \sqrt{[y]} \tag{5.16}$$

Pour des CSP complexes, il peut être nécessaire de réaliser plusieurs contractions successives jusqu'à ce qu'il n'y ait plus de contraction d'intervalle. Ces contractions sont réalisées sous formes de boucles de contractions Forward / Backward, réalisées tant qu'au moins un intervalle est contracté.

Numériquement, on reprend les résultats du paragraphe précédent : $x \in [1; 3.91]$, $y \in [3; 50]$, $z \in [2.09; 5]$. On contracte les intervalles en utilisant la contrainte supplémentaire :

1. Forward :

$$[y] \in [y] \bigcap [z]^2 \in [3; 50] \bigcap [2.09; 5]^2 \in [4.36; 25] \tag{5.17}$$
$$[z] \in [z] \bigcap ([x] + \ln_{[]}([y])) \in [2.09; 5] \bigcap [1; 3.91] + \ln_{[]}([4.36; 25]) \tag{5.18}$$
$$\in [2.47; 5] \tag{5.19}$$

2. Backward :

$$[x] \in [x] \bigcap ([z] - \ln_{[]}([y])) \in [1; 3.91] \bigcap [2.47; 5] - \ln_{[]}([4.36; 25]) \in [1; 3.53] \tag{5.20}$$
$$[y] \in [y] \bigcap \exp_{[]}([z] - [x]) \in [4.36; 25] \bigcap \exp_{[]}([2.47; 5] - [1; 3.53]) \tag{5.21}$$
$$\in [4.36; 25] \tag{5.22}$$
$$[z] \in [z] \bigcap \sqrt{[y]} \in [2.47; 5] \bigcap \sqrt{[4.36; 25]} \in [2.47; 5] \tag{5.23}$$

Après la première étape de Forward/Backward, il est encore possible de recontracter les intervalles (Seul les équations impliquant une réduction d'intervalle sont explicitées) :

$$[y] \in [y] \bigcap [z]^2 \in [4.36; 25] \bigcap [2.47; 5]^2 \in [6.10; 25] \quad (5.24)$$

$$[z] \in [z] \bigcap ([x] + \ln_{[]}([y])) \in [2.47; 5] \bigcap [1; 3.53] + \ln_{[]}([6.10; 25]) \in [2.80; 5] \quad (5.25)$$

$$[x] \in [x] \bigcap ([z] - \ln_{[]}([y])) \in [1; 3.53] \bigcap [2.80; 5] - \ln_{[]}([6.10; 25]) \in [1; 3.20] \quad (5.26)$$

La contraction est effectuée tant qu'au moins un intervalle est réduit. La propagation de contraction est réalisée en Annexes 6.2, Equations 6.2. La propagation de contraintes est maintenant stabilisée : aucun intervalle ne peut plus être contracté. Il est à remarquer que de multiples itérations Backward/Forward ont été nécessaires pour contracter au maximum les intervalles.

5.4 Définition de l'algorithme de SLAM ensembliste

Nous avons étudié les différents outils nécessaires à la conception d'un algorithme de SLAM ensembliste par propagation de contraintes. Notre objectif principal est de définir un algorithme dont les résultats sont consistants et garantis.
L'algorithme utilise des données odomètriques pour prédire la position du robot (comme Di Marco *et al.* [55], Porta [59]) et des données provenant d'une caméra monoculaire pour corriger cette position. La position du mobile est représentée par un pavé en trois dimensions : $[\mathbf{x}] = ([x][y][\theta])^{\mathrm{T}}$ avec x, y la position du robot dans le repère global et θ son orientation. On suppose que le robot se déplace sur un plan.

5.5 Etape de Prédiction

L'étape de prédiction utilise les capteurs proprioceptifs embarqués sur notre plateforme expérimentale : les odomètres. L'EKF-SLAM ainsi que le FastSLAM utilisent un modèle de déplacement probabiliste pour intégrer les données odométriques. Nous utiliserons ce modèle pour obtenir un modèle d'intégration ensembliste, défini par Seignez *et al.* [54]. L'utilisation de ce modèle montrera le pessimisme important des résulats. Nous proposons donc une solution pour limiter le pessimisme induit par le modèle d'intégration des données.

5.5.1 Modèle probabiliste

Chaque odomètre retourne la distance (nt_r, nt_l, en nombre de pas) parcourue par une roue. On utilise le même modèle de déplacement que pour l'EKF-SLAM ou le

FastSLAM :

$$\mathbf{f}(\mathbf{x}_{k-1}, \delta s_k, \delta\theta_k) = \begin{pmatrix} x_{k-1} + \delta s \cos\left(\theta_{k-1} + \frac{\delta\theta}{2}\right) \\ y_{k-1} + \delta s \sin\left(\theta_{k-1} + \frac{\delta\theta}{2}\right) \\ \theta_{k-1} + \delta\theta \end{pmatrix} \tag{5.27}$$

avec :

– δs_k : la distance parcourue par le véhicule entre l'instant $k-1$ et k.

– $\delta\theta_k$: la variation de cap entre l'instant $k-1$ et k.

Le déplacement longitudinal $\delta\theta$ correspond à la longueur de l'arc de cercle défini par le déplacement du robot et le déplacement rotationnel δs correspond à l'angle représentant l'arc de cercle (Voir Section. 2.3.2). Ces deux variables sont calculées à l'aide des données proprioceptives. Les données proprioceptives (np_r, np_l) sont utilisées pour calculer le déplacement (δ_r, δ_l) de chaque roue en mètres :

$$\delta_{r,l} = \frac{2\Pi r}{N_{pt}} np_{r,l} \tag{5.28}$$

avec r le rayon de la roue et N_{pt} le nombre de pas par tour de notre odomètre.

On peut ensuite calculer le déplacement longitudinal (δs) et rotationnel ($\delta\theta$) :

$$\begin{pmatrix} \delta s \\ \delta\theta \end{pmatrix} = \mathbf{g}(\delta_l, \delta_r) = \begin{pmatrix} \frac{\delta_r + \delta_l}{2} \\ \frac{\delta_l - \delta_r}{2e} \end{pmatrix} \tag{5.29}$$

Avec $2e$ la taille de l'entraxe (en mètre).

5.5.2 Modèle Ensembliste

Pour le modèle ensembliste, il est nécessaire de borner l'ensemble des incertitudes. En effet, les paramètres tels que le rayon de chaque roue ou l'entraxe des roues arrières ne peuvent pas être connus avec précision. On définit donc :

– $[r]$: l'intervalle contenant la valeur réelle du rayon des roues.

– $[e]$: l'intervalle contenant la valeur réelle de l'entraxe des roues arrières.

– $[np_{r,l}] = [np_{r,l} - 1, np_{r,l} + 1]$: l'intervalle contenant la distance parcourue par une roue (en nombre de pas) en supposant que le glissement des roues est nul.

L'ajout de deux pas d'incertitude $[nt_{r,l} - 1, nt_{r,l} + 1]$ est nécessaire pour obtenir de manière garantie le déplacement d'un odomètre. La figure 5.1 représente un déplacement ainsi que le résultat de l'accumulation des pas odométriques. La distance réelle parcourue est d'environ 6.5 pas odométriques, alors que le comptage de pas odométriques est de 7 pas.

Quatres distances parcourues peuvent être choisies en fonction de l'emplacement du

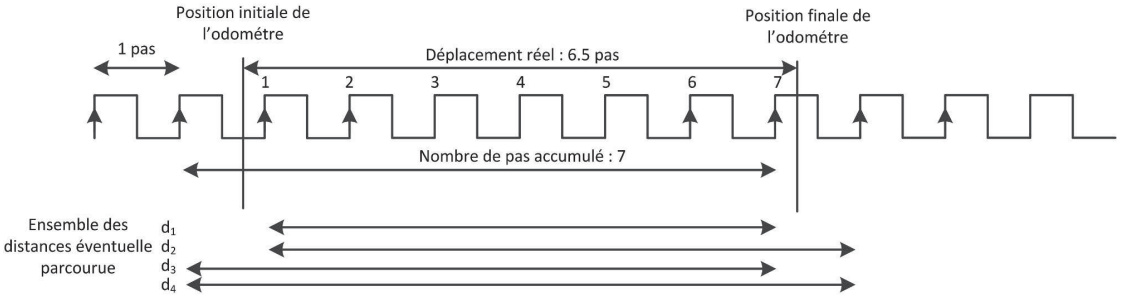


Figure 5.1: *Utilisation des données odométriques*

mobile à l'intérieur d'un pas odométrique :

- La distance d_1 choisit de placer le robot à la fin du premier pas odométrique et au début du dernier pas, la distance ainsi parcourue est 6 pas.
- La distance d_2 place le robot à la fin du premier pas et à la fin du dernier pas, la distance est maintenant de 7 pas.
- La distance d_3 place le robot au début du premier pas et du dernier pas, la distance est de 7 pas.
- La distance d_4 place le robot au début du premier pas et à la fin du dernier pas, la distance est maintenant de 8 pas.

Notre méthode doit inclure de manière garantie l'ensemble des données capteurs, il est donc nécessaire d'inclure l'ensemble des distances d_1à d_4 dans un intervalle. Dans notre exemple, la distance utilisée sera $d_1 \bigcup d_2 \bigcup d_3 \bigcup d_4 = [6; 8]$. Elle inclut donc deux pas d'erreurs : un pas en plus de la valeur retournée par l'odomètre et un pas en moins.

Pour obtenir le déplacement en mètre de chaque roue, on utilise la version ensembliste du modèle utilisé précédemment (Section. 2.3.2) :

$$[\delta_{r,l}] \quad = \quad \frac{2[\pi][r]}{N_{pt}}[np_{r,l}] \tag{5.30}$$

La valeur exacte du nombre π n'étant pas connue, il est nécessaire de la définir comme un intervalle. On calcule le déplacement longitudinal et le déplacement rotationnel :

$$\begin{pmatrix} [\delta s] \\ [\delta\theta] \end{pmatrix} = \mathbf{g}([\delta_l], [\delta_r]) \quad = \quad \begin{pmatrix} \frac{[\delta_r]+[\delta_l]}{2} \\ \frac{[\delta_l]-[\delta_r]}{2[e]} \end{pmatrix} \tag{5.31}$$

Enfin, on remplace les variables dans le modèle pour obtenir le modèle défini par Seignez *et al.* [54] :

$$\mathbf{f}([\mathbf{x}_{k-1}],[\delta s_k],[\delta\theta_k]) = \begin{pmatrix} [x_{k-1}] + [\delta s_k]\cos\left([\theta_{k-1}] + \frac{[\delta\theta_k]}{2}\right) \\ [y_{k-1}] + [\delta s_k]\sin\left([\theta_{k-1}] + \frac{[\delta\theta_k]}{2}\right) \\ [\theta_{k-1}] + [\delta\theta] \end{pmatrix} \tag{5.32}$$

Ce modèle permet d'obtenir à chaque instant un pavé contenant obligatoirement la position réelle du mobile. Le pavé inclut les incertitudes odométriques.

5.5.3 CSP Prédiction

On utilise le modèle de prédiction pour former un premier problème de satisfaction de contraintes. En effet, chacune des positions successives du mobile sont reliées par le modèle de déplacement et les données odométriques. Pour former le problème de satisfaction de contraintes, on exprime chacune des variables individuellement.
Par exemple, la mise à jour de la variable $[x_k]$ par l'expression :

$$[x_{k-1}] + [\delta s_k]\cos\left([\theta_{k-1}] + \frac{[\delta\theta_k]}{2}\right)$$

Nous permet de définir 4 équations de Backward. En effet, les variables $[x_{k-1}]$, $[\delta s_k]$, $[\theta_{k-1}]$ et $[\delta\theta_k]$ peuvent être exprimées en fonction des autres. Après avoir exprimé chaque variable, on obtient un jeu de 10 contraintes de Backward et 3 contraintes de Forward, la table 5.1 résume l'ensemble des contraintes définies.

On remarque que le tableau 5.1 ne recalcule pas les constantes propres au véhicule : le rayon de la roue ou l'entraxe. Nous avons choisi de considérer ces paramètres comme pouvant évoluer légèrement au cours d'une expérimentation.

5.5.4 Premiers résultats

Pour évaluer notre modèle, on réalise une simulation dans un environnement simplifié. L'utilisation d'un trajet simplifié permet d'expliciter plus aisément le comportement de l'algorithme. Le robot réalise un parcours ovale. En réalisant 900 étapes, le robot parcourt un peu plus d'un tour complet. La figure 5.2 représente le trajet suivi par le robot avec le simulateur. Ce trajet représente un parcours de plus de 50m.

La figure 5.3 représente la boite de localisation du robot toutes les 100 prédictions. On remarque que l'aire de la boite de localisation augmente de manière très rapide. Pour quantifier cette augmentation, on utilise le volume de la boite de localisation ainsi que le couloir d'incertitude.

A chaque instant, la position réelle $(x_{reel}, y_{reel}, \theta_{reel})$ du robot est connue par le simulateur. On calcule les couloirs d'incertitude en utilisant les formules :

Propagation Backward		
1. $[x_{k-1}]$	=	$[x_{k-1}] \bigcap ([x_k] - [\delta s_k] \cos([\theta_{k-1}] + \frac{[\delta\theta_k]}{2}))$
2. $[y_{k-1}]$	=	$[y_{k-1}] \bigcap ([y_k] - [\delta s_k] \sin([\theta_{k-1}] + \frac{[\delta\theta_k]}{2}))$
3. $[\delta s_k]$	=	$[\delta s_k] \bigcap (([x_k] - [x_{k-1}]) / \cos([\theta_{k-1}] + \frac{[\delta\theta_k]}{2}))$
4. $[\delta s_k]$	=	$[\delta s_k] \bigcap (([y_k] - [y_{k-1}]) / \sin([\theta_{k-1}] + \frac{[\delta\theta_k]}{2}))$
5. $[\theta_{k-1}]$	=	$[\theta_{k-1}] \bigcap ([\theta_k] - [\delta\theta_k])$
6. $[\theta_{k-1}]$	=	$[\theta_{k-1}] \bigcap (\arccos(([x_k] - [x_{k-1}])/[\delta s_k]) - \frac{[\delta\theta_k]}{2})$
7. $[\theta_{k-1}]$	=	$[\theta_{k-1}] \bigcap (\arcsin(([y_k] - [y_{k-1}])/[\delta s_k]) - \frac{[\delta\theta_k]}{2})$
8. $[\delta\theta_k]$	=	$[\delta\theta_k] \bigcap (\arccos(([x_k] - [x_{k-1}])/[\delta s_k]) - [\theta_{k-1}])$
9. $[\delta\theta_k]$	=	$[\delta\theta_k] \bigcap (\arcsin(([y_k] - [y_{k-1}])/[\delta s_k]) - [\theta_{k-1}])$
10. $[\delta\theta_k]$	=	$[\delta\theta_k] \bigcap ([\theta_k] - [\theta_{k-1}])$
Propagation Forward		
11. $[x_k]$	=	$[x_k] \bigcap ([x_{k-1}] + [\delta s_k] \cos([\theta_{k-1}] + \frac{[\delta\theta_k]}{2}))$
12. $[y_k]$	=	$[y_k] \bigcap ([y_{k-1}] + [\delta s_k] \sin([\theta_{k-1}] + \frac{[\delta\theta_k]}{2}))$
13. $[\theta_k]$	=	$[\theta_k] \bigcap ([\theta_{k-1}] + [\delta\theta_k])$

Table 5.1: *Définition du CSP prédiction*

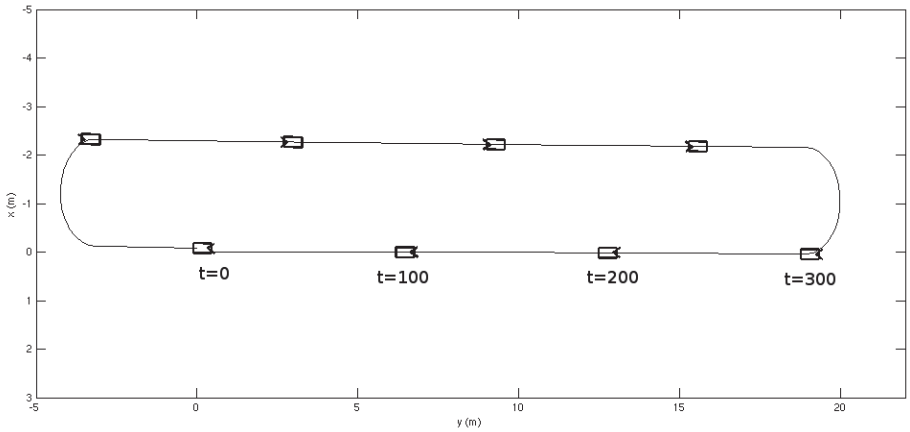


Figure 5.2: *Trajet du robot simulé*

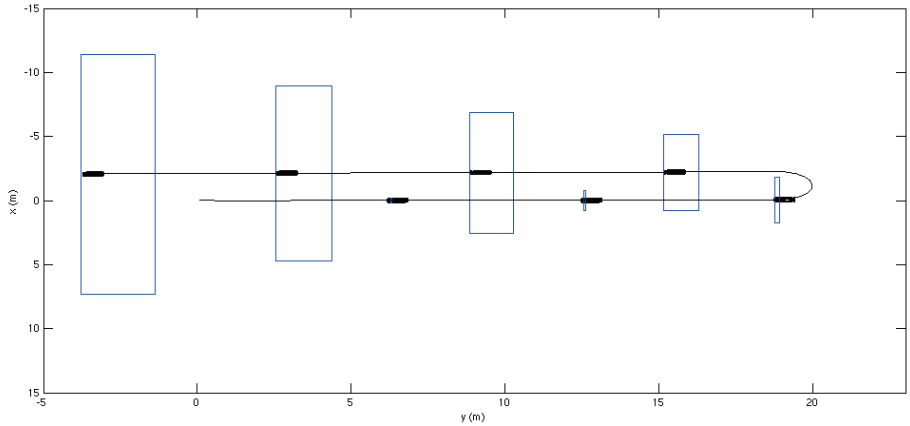


Figure 5.3: *Trajet du robot simulé, résultat de l'étape de prédiction*

$$\begin{aligned} [couloir_x] &= [x] - x_{reel} \\ [couloir_y] &= [y] - y_{reel} \\ [couloir_\theta] &= [\theta] - \theta_{reel} \end{aligned} \tag{5.33}$$

Ce couloir permet de vérifier que le résultat de l'algorithme est consistant. En effet, si le résultat est consistant alors la valeur zéro sera incluse dans le couloir. C'est à dire que la position réelle est incluse dans l'ensemble résultat.

La figure 5.4a montre les couloirs d'erreurs obtenus. On remarque que les résultats sont consistants. Cependant, à la fin de l'expérimentation, le couloir est très large (>40m de large). On peut considérer, dans ce cas, que le robot est perdu. Deux hausses importantes de la taille du couloir sont observables aux étapes 320 et 720 sur l'axe y, ces hausses sont dues aux changements de cap du robot. En effet, sur la première partie du trajet, l'incertitude de cap agrandit la boite de localisation suivant l'axe x. Après la première rotation, le volume augmentera principalement suivant l'axe y. L'incertitude du cap produit une augmentation rapide du volume du pavé de localisation.

On réalise ensuite une seconde simulation en ajoutant des incertitudes sur la mesure du rayon de la roue et celle de l'entraxe. On utilise les paramètres suivant :

- $r = [0.04 - 0.0005; 0.04 + 0.0005]$ m.
- $e = [0.10 - 0.005; 0.10 + 0.005]$ m.

La figure 5.4b représente les couloirs d'incertitudes obtenus. Les résultats sont similaires aux résultats obtenus précédemment, cependant, les couloirs d'incertitudes sont un petit peu plus large. La figure 5.5 représente le volume de la boite de localisation. Conformément aux intuitions, le volume de la boite sans incertitude est inférieure à celui de la boite prenant en compte l'intégralité des incertitudes.

5.5.5 Améliorations de la gestion des données odométriques

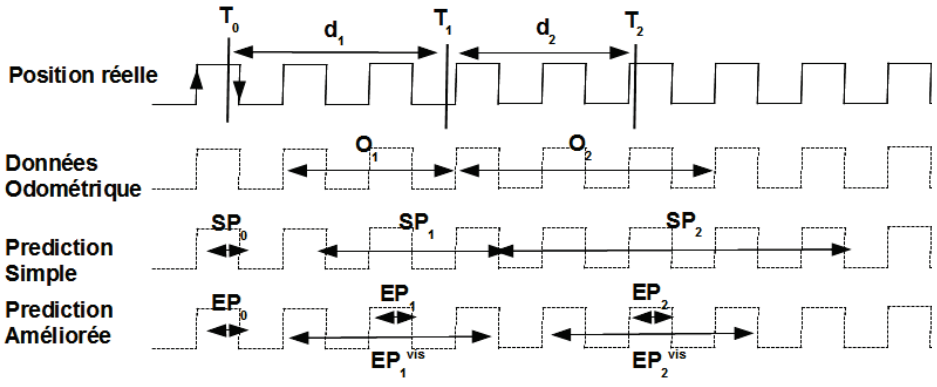
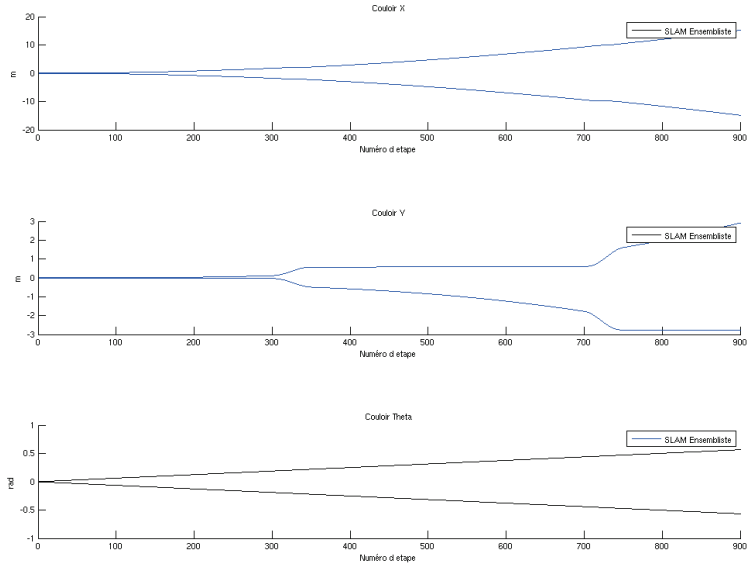
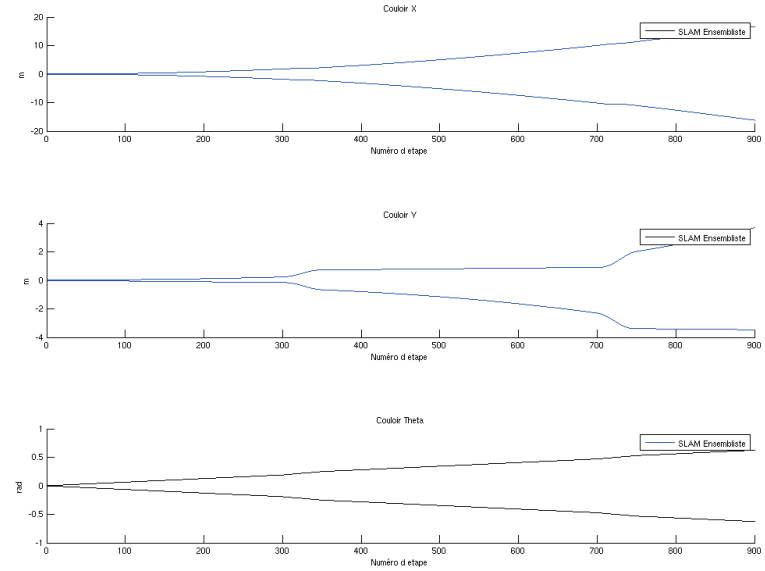


Figure 5.6: *Amélioration de la gestion des données odométriques*



(a) *Couloir d'incertitude sans incertitude sur les constantes*



(b) *Couloir d'incertitude incluant des incertitudes sur les constantes*

Figure 5.4: *Couloirs d'incertitude issus de la prédiction*

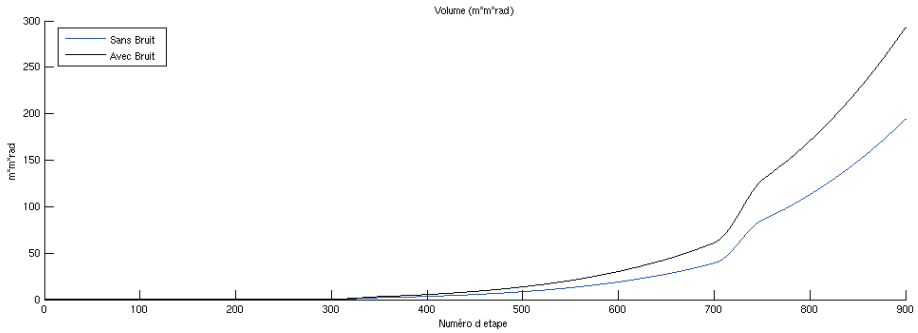


Figure 5.5: *Volume de la boite de localisation*

Précédemment, nous avons constaté que le volume de la boite de localisation augmentait de manière importante. En effet, nous avons remarqué au paragraphe 5.5.2 que l'intégration des donnés odométriques, de manière ensembliste, obligeait d'inclure deux pas d'incertitudes à chaque intégration des données. On propose une amélioration de la prise en compte des données odométriques pour diminuer le pessimisme de l'étape de prédiction. On se place dans le cas ou il n'y a aucun glissement de roues.

La figure 5.6 représente les signaux issus d'un odomètre lorsque la roue a réalisé un mouvement $d = d_1 + d_2$. Le système compte les fronts montants issus du signal odomètrique.
Classiquement, les résultats de localisation sont modélisés par le "Simple Prediction" (SP), notre contribution obtient les résultats "Prediction Améliorée " (EP). Au lancement de l'algorithme, la boite de localisation est très petite (SP_0, EP_0).

Déplacement réalisé Pour notre exemple, la roue réalise deux mouvements successif : le premier mouvement a une longeur d_1 et correspond aux instants de T_0 à T_1, le second mouvement a une longueur d_2 et correspond aux instants T_1 à T_2.
Les données odométriques correspondant à ce moment sont O_1 qui représente deux fronts montants et O_2 trois fronts montants.

Prédiction Simple Pour intégrer le déplacement d_1, le modèle classique, SP ajoute deux pas d'erreurs (Voir paragraphe 5.5.2) et obtient la boite ($[SP_1]$). La taille de la boite résultat est d'environ 2 pas. On vérifie que le résultat est consistant, $[SP_1]$ inclutla localisation réelle T_1.

Ensuite, pour le déplacement d_2, SP ajoute de nouveau deux pas d'erreurs et obtient la boite $[SP_2]$), la taille de la boite est d'environ 4 pas. On vérifie que le résultat est consistant, $[SP_2]$ inclut la localisation réelle T_2.

Les deux déplacements successifs ont impliqué l'ajout de 2×2 pas d'erreurs. Or si le déplacement avait été réalisé en une seule fois ($d = d_1+d_2$) alors l'algorithme n'aurait ajouté qu'un seul pas d'erreur.

Par conséquent, nous affirmons que l'erreur due à la quantification des données odométriques est non cumulative. Une telle erreur est la même après un déplacement de 1 m ou 1 km et il n'est pas nécessaire de propager une erreur le long du chemin suivi. Néanmoins nous devons prendre en compte cette erreur à chaque fois que nous avons besoin d'utiliser la position exacte du robot.

Prédiction Améliorée Nous proposons donc de calculer l'image de la boite de localisation en utilisant le même modèle odométrique mais sans ajouter de pas supplémentaire. Il est nécessaire d'ajouter les pas d'erreurs uniquement pour connaitre la boite incluant de manière sûre la position réelle. Après le déplacement d_1, EP calcule deux boites :

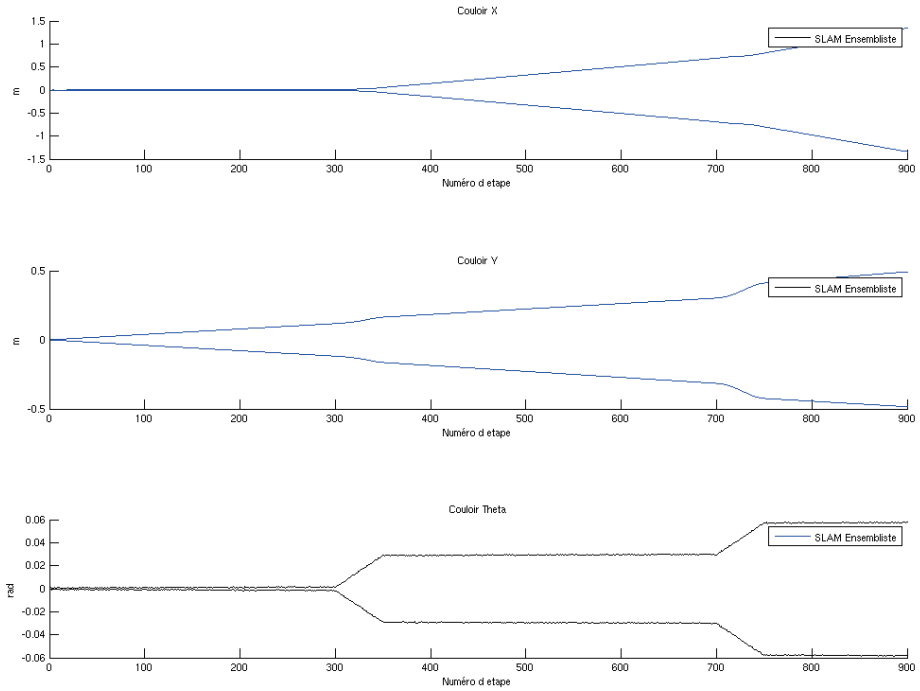


Figure 5.7: *Couloir d'incertitude avec des incertitudes sur les constantes*

- la boite EP_1 qui sera utilisée pour la prochaine prédiction n'inclut pas de pas d'erreur.
- la boite EP_1^{vis} qui inclut les pas d'erreurs, cette boite sera utilisée pour l'étape d'estimation.

A la fin d'une étape d'estimation, on choisira la plus petite boite entre EP_1 et EP_1^{vis} pour la prochaine prédiction. Dans notre cas, la boite EP_1 a une taille inférieure à EP_1^{vis}, on utilisera cette boite pour intégrer le déplacement d_2. L'intégration de ce déplacement donnera les boites EP_2 et EP_2^{vis}. On remarquera que la taille de EP_1^{vis} est inférieure à celle de SP_2.

Résultats obtenus La figure 5.7 représente les couloirs d'incertitudes. On obtient un résultat dix fois meilleur que précédemment. A la fin du parcours, la largeur du couloir en x ou y est d'environ 4m alors qu'elle était de 40m précédemment. Le gain est confirmé par le calcul du volume de la figure 5.8.

5.6 Etape d'Estimation

L'étape d'estimation consiste à définir une cartographie de l'environnement tout en améliorant la localisation du robot. Dans notre cas, cette étape est réalisée à partir d'une caméra monoculaire. Nous proposons d'introduire l'utilisation d'une modèle pinhole ensembliste. De plus, nous proposons une paramétrisation des amers permettant une initialisation directe incluant une incertitude infinie sur la variable de profondeur.

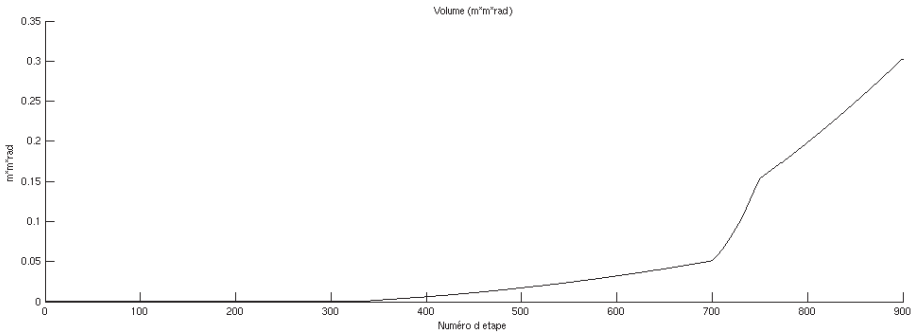


Figure 5.8: *Volume de la boite de localisation*

L'étape d'estimation constituera un second problème de satisfaction de contraintes. On propose d'étudier le cas d'équation simplifiée en deux dimensions pour étudier au mieux le comportement de l'algorithme.

5.6.1 Paramétrisation des amers

En SLAM probabiliste, les amers sont classiquement représentés en utilisant une paramétrisation par inverse de profondeur [2]. Elle permet de limiter les problèmes dus à la linéarisation du modèle d'observation et d'initialiser des amers ayant une incertitude de profondeur importante sans aucun délai.
Idéalement, l'incertitude de la caméra doit être représentée par un cône de profondeur infinie. L'analyse par intervalles peut représenter ce cône en utilisant la paramétrisation :

$$[\mathbf{y}_i] = ([x_i], [y_i], [\varphi_i], [\theta_i], [d_i]) \tag{5.34}$$

avec :

• $[x_i], [y_i]$: les intervalles incluant la position de la caméra lors de la première observation de l'amer.

• $[\phi_i], [\theta_i]$: les intervalles contenant l'élévation et l'azimuth du vecteur pointant vers l'amers $\overrightarrow{m}(\phi_i, \theta_i)$.

• $[d_i]$: l'intervalle contenant la distance de la caméra à l'amer.

Cette paramétrisation permet une initialisation sans délai des amers, les observations seront ajoutées directement dans le CSP d'estimation. De plus, en utilisant notre paramétrisation, l'incertitude de profondeur est représentée de manière uniforme, aucune hypothèse n'est réalisée sur la profondeur supposée de l'amer.

5.6.2 Initialisation d'un amer

Pour initialiser un amer, il faut réaliser un changement de repère entre le repère caméra et le repère global. Les changements de repère sont réalisés de la même manière

que précédemment sauf que l'on utilise des intervalles à la place de simples nombres. On définit par exemple la transformation ensembliste du repère global au repère mobile $(\mathrm{T}_{[],\mathrm{glob,mob}}(\mathrm{x_{glob}}, \mathrm{y_{glob}}, \mathrm{z_{glob}}))$:

$$\begin{pmatrix} [x_{mob}] \\ [y_{mob}] \\ [z_{mob}] \end{pmatrix} = \begin{bmatrix} \cos_{[]}([\theta]) & 0 & \sin_{[]}([\theta]) \\ 0 & 1 & 0 \\ -\sin_{[]}([\theta]) & 0 & \cos_{[]}([\theta]) \end{bmatrix} \left(\begin{pmatrix} [x_{glob}] \\ [y_{glob}] \\ [z_{glob}] \end{pmatrix} - \begin{pmatrix} [x] \\ [y] \\ 0 \end{pmatrix} \right) \quad (5.35)$$

avec :

– $[x_{mob}]$, $[y_{mob}]$, $[z_{mob}]$: la position d'un cube dans le repère mobile.

– $[x_{glob}]$, $[y_{glob}]$, $[z_{glob}]$: la position de ce cube dans le repère global.

– $[x]$, $[y]$, $[\theta]$: la position du robot dans le repère global.

Lors de l'initialisation d'un amer, les paramètres sont définis par :

• $[x_i] = [x_{camera}]$ et $[y_i] = [y_{camera}]$: le premier point de vue est égal à la position de la caméra .

• $[\theta_i] = [\theta] + [u_{angle}]$ and $[\varphi_i] = [v_{angle}]$: θ les valeurs d'observation u et v sont converties en angle. La caméra est un capteur angulaire : chaque pixel correspond à un angle d'ouverture et un angle d'élévation, définis en utilisant le modèle Pinhole :

$$[u_{angle}] = -\arctan_{[]}(\frac{[c_u] - [u_{obs}]}{[f][k_u]}) \quad (5.36)$$

$$[v_{angle}] = -\arctan_{[]}(\frac{[c_v] - [v_{obs}]}{[f][k_v] cos_{[]}([u_{angle}])}) \quad (5.37)$$

avec u_{obs}, v_{obs} les pixels correspondant à l'observation, f la distance focale, (k_u, k_v) la taille des pixels et (c_u, c_v) le centre optique. Les paramètres sont obtenus à l'aide de la calibration de la caméra, on a ajouté un bruit correspondant à 3 sigmas du modèle probabiliste.

• $[d_i] =]0; +\infty[$: la distance de la caméra à l'amer est inconnue lors de la première observation.

La figure 5.9 représente l'initialisation de deux amers. Le premier cas représente une initialisation au début de la trajectoire. Le mobile étant correctement localisé, l'incertitude angulaire est relativement faible, on remarque que le cône d'incertitude inclut bien l'incertitude infinie de profondeur.

Le second cas représente une initialisation en milieu de trajet avec une incertitude d'orientation importante, la zone de localisation définie par le cône est beaucoup plus importante dans ce cas. En effet, l'incertitude due à la caméra est la même que dans le cas précédent, cependant, durant le trajet le mobile a accumulé une incertitude d'orientation. C'est pourquoi le cône d'incertitude de l'amer a une ouverture plus importante.

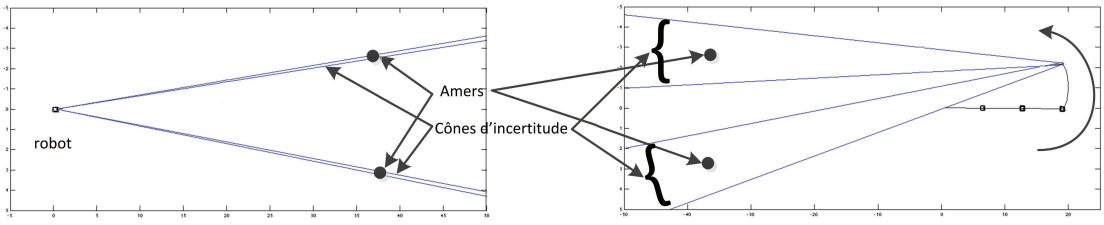


Figure 5.9: *Initialisation d'amers avec et sans incertitude d'orientation du mobile*

L'un des intérêts majeurs de la théorie ensembliste est de pouvoir modéliser le plus précisément possible les capteurs. En effet, l'incertitude de profondeur est correctement initialisée et inclut l'ensemble des profondeurs possibles. De plus, il n'y a aucune hypothèse a priori sur la profondeur.

5.6.3 Estimations

La phase d'estimation utilise la propagation de contraintes. Dans notre cas, plusieurs contraintes sont disponibles :

- le modèle de déplacement (Eq. 5.32) définit un ensemble de contraintes. En effet, on utilise toutes les équations de déplacement de l'instant t=0 à l'instant présent.
- l'observation des amers nous donne un second ensemble de contraintes.

L'observation des amers est réalisée comme pour le FastSLAM mais en utilisant l'analyse par intervalles. La position de l'amer dans le repère global est calculée en utilisant la formule suivante :

$$[\mathbf{A}_i] = \begin{pmatrix} [x_i] \\ [y_i] \\ 0 \end{pmatrix} + [d_i]\overrightarrow{m}_{[]}([\theta_i],[\varphi_i]) \tag{5.38}$$

avec :

$$\overrightarrow{m}_{[]}([\theta_i],[\varphi_i]) = \begin{pmatrix} \cos_{[]}([\theta_i])\cos_{[]}([\varphi_i]) \\ -\sin_{[]}([\theta_i])\cos_{[]}([\varphi_i]) \\ \sin_{[]}([\varphi_i]) \end{pmatrix} \tag{5.39}$$

En utilisant la position du robot $[\mathbf{x}_k]$ et la position de l'amer $[\mathbf{A}_i]$, on en déduit la position de l'amer dans le repère caméra :

$$[\mathbf{A}_{i,cam}] = T_{[],mob,cam}(R_{[]}([\mathbf{A}_i] - [\mathbf{x}_k])) \tag{5.40}$$

avec $T_{[],mob,cam}$ la transformation ensembliste entre le repère mobile et le repère caméra, $R_{[]}$ la matrice de rotation entre le repère du robot et le repère global.

En remplaçant $[\mathbf{A}_i]$ dans l'expression précédente, on obtient :

$$[\mathbf{h}_i^c] = T_{[],mob,cam}(R_{[]}(\begin{pmatrix} [x_i] \\ [y_i] \\ 0 \end{pmatrix} + [d_i]\overrightarrow{m}_{[]}([\theta_i],[\varphi_i]) - [\mathbf{x}_k])) \tag{5.41}$$

La caméra n'observe pas directement $[\mathbf{h}_i^c]$ mais sa projection sur le plan image. Grâce au modèle Pinhole, on obtient :

$$[\mathbf{h}_i] = \begin{pmatrix} [u_{i,pred}] \\ [v_{i,pred}] \end{pmatrix} = \begin{pmatrix} [c_u] - [f][k_u]\frac{[h_{i,x}^c]}{[h_{i,z}^c]} \\ [c_v] - [f][k_v]\frac{[h_{i,y}^c]}{[h_{i,z}^c]} \end{pmatrix} \tag{5.42}$$

Chaque constante du modèle Pinhole $[c_u],[f],[k_u],[c_v],[f],[k_v]$ est définie par un intervalle, contenant la valeur réelle. Cette méthode inclut une imprécision sur chaque variable du modèle.

Pour mettre à jour la position d'un amer, on réalise l'intersection entre chaque paramètre connu $([x_i],[y_i],[\theta_i],[\varphi_i],[d_i])$ et chaque paramètre observé. Pour cela, on inverse les équations 5.42 et on exprime chaque paramètre $([x_i],[y_i],[\theta_i],[\varphi_i],[\rho_i])$ en fonction des autres variables.

On réalise ensuite une propagation de contraintes de type Backward/Forward sur l'ensemble des contraintes (modèle de déplacement et d'observation) pour réduire les intervalles.

5.6.3.1 Cas 2 dimensions

Définition des équations de propagation Pour pouvoir exprimer les équations de propagation, on se concentre sur un cas 2D avec des équations simplifiées. La figure définit les repères utilisés, le repère caméra est confondu avec le repère mobile.

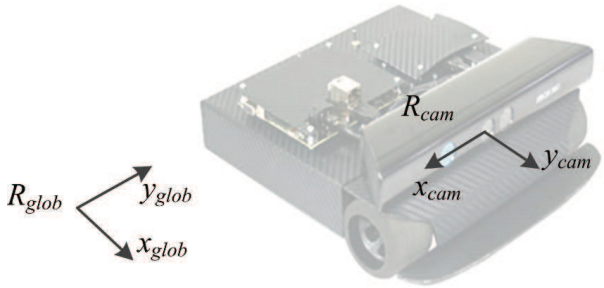


Figure 5.10: *Définition des repères simplifiés*

Le robot est représenté par une boite $[\mathbf{x}] = ([x][y][\theta])^{\mathrm{T}}$, la caméra est positionnée exactement au même endroit que les odomètres. Les amers considérés sont uniquement en deux dimensions $[x_a], [y_a]$.

Dans le repère caméra, la position d'un amer est :

$$\begin{aligned} [A_x^c] &= \cos(\theta - \frac{\pi}{2})(x - x_a) + \sin(\theta - \frac{\pi}{2})(y - y_a) \\ [A_y^c] &= -\sin(\theta - \frac{\pi}{2})(x - x_a) + \cos(\theta - \frac{\pi}{2})(y - y_a) \end{aligned} \tag{5.43}$$

On pose $\theta' = \theta - \frac{\pi}{2}$. Dans notre cas 2D, on prédit la position de l'amer sur l'image $[u_{pred}]$ en utilisant la formule :

$$[u_{pred}] = [c_u] + [f][k_u]\frac{[A_x^c]}{[A_y^c]} \tag{5.44}$$

En développant $[u_{pred}]$, on obtient :

$$[u_{pred}] = [c_u] + [f][k_u]\frac{\cos_{[]}([\theta'])([x] - [x_a]) + \sin_{[]}([\theta'])([y] - [y_a])}{-\sin_{[]}([\theta'])([x] - [x_a]) + \cos_{[]}([\theta'])([y] - [y_a])} \tag{5.45}$$

L'équation de forward de notre problème de satisfaction de contraintes correspond à l'intersection entre la prédiction et l'observation réalisées :

$$[u_{obs}] = [u_{obs}] \bigcap [u_{pred}] \tag{5.46}$$

$$= [u_{obs}] \bigcap [c_u] + [f][k_u]\frac{\cos_{[]}([\theta'])([x] - [x_a]) + \sin_{[]}([\theta'])([y] - [y_a])}{-\sin_{[]}([\theta'])([x] - [x_a]) + \cos_{[]}([\theta'])([y] - [y_a])} \tag{5.47}$$

On exprime ensuite chaque variable en fonction des autres pour obtenir les équations de backward, le CSP est : (les intervalles sont représentés sans [])

$$x = x \bigcap x_a - \frac{\sin(\theta')(y - y_a) + \cos(\theta')(c_u - u_{obs})(y - y_a)/fk_u}{\cos(\theta') - \sin(\theta')(c_u - u_{obs})/fk_u} \tag{5.48}$$

$$y = z \bigcap y_a - \frac{\cos(\theta')(x - x_a) - \sin(\theta')(c_u - u_{obs})(x - x_a)/fk_u}{\sin(\theta') + \cos(\theta')(c_u - u_{obs})/fk_u} \tag{5.49}$$

$$x_a = x_a \bigcap x + \frac{\sin(\theta')(y - y_a) + \cos(\theta')(c_u - u_{obs})(y - y_a)/fk_u}{\cos(\theta') - \sin(\theta')(c_u - u_{obs})/fk_u} \tag{5.50}$$

$$y_a = y_a \bigcap y + \frac{\cos(\theta')(x - x_a) - \sin(\theta')(c_u - u_{obs})(y - y_a)/fk_u}{\sin(\theta') + \cos(\theta')(c_u - u_{obs})/fk_u} \tag{5.51}$$

$$\theta' = \theta' \bigcap \arccos(\frac{-\sin(\theta')(y - y_a) + \sin(\theta')(c_u - u_{obs})(x - x_a)/fk_u}{x - x_a + (c_u - u_{obs})(y - y_a)/fk_u}) \tag{5.52}$$

$$\theta' = \theta' \bigcap \arcsin(\frac{\cos(\theta')(x - x_a) + \cos(\theta')(c_u - u_{obs})(y - y_a)/fk_u}{y_a - y + (cu - u_{obs})(y - y_a)/fk_u}) \tag{5.53}$$

Chaque amer observé définit un ensemble de 5 contraintes pour chaque observation. L'ensemble de ces observations définit le problème de satisfaction de contraintes.

Exemple numérique On se focalise dans le cas où $u_{obs} = c_u$: l'observation est réalisée exactement au centre optique de la caméra. Les équations de propagation simplifiées sont :

$$x = x \bigcap x_a - \frac{\sin(\theta')(z)}{\cos(\theta')} \tag{5.54}$$

$$y = y \bigcap y_a - \frac{\cos(\theta')(x - x_a)}{\sin(\theta')} \tag{5.55}$$

$$x_a = x_a \bigcap x + \frac{\sin(\theta')(y - y_a)}{\cos(\theta')} \tag{5.56}$$

$$y_a = y_a \bigcap z + \frac{\cos(\theta')(x - x_a)}{\sin(\theta')} \tag{5.57}$$

$$\theta' = \theta' \bigcap \arccos(\frac{-\sin(\theta')(y - y_a)}{x - xa}) \tag{5.58}$$

$$\theta' = \theta' \bigcap \arcsin(\frac{\cos(\theta')(x - x_a)}{z_a - z}) \tag{5.59}$$

La figure 5.11 représente la correction de la boite de localisation du robot lors d'une observation. Dans ce cas, la position de l'amer est supposée connue parfaitement ($[x_a] = 10$, $[y_a] = 0$), dans le plan du robot et l'incertitude d'observation est nulle.

A t_1, la boite de localisation est orientée avec $\theta = [0, 0]$, $\theta' = [-\pi/2, -\pi/2]$et $[x] = [y] = [-2, 2]$, la position de l'amer est connu exactement.

A t_2, le robot observe l'amer et le met en correspondance avec un amer de sa carte.

A t_3, le robot corrige sa position à l'aide de l'observation réalisée. Dans notre cas, l'observation est réalisée suivant l'axe x, elle corrigera donc la boite de localisation suivant l'axe z.

Numériquement, on prend $\theta' = [-\pi/2, -\pi/2]$, $[x] = [y] = [-2, 2]$, $[x_a] = 10$ et $[y_a] = 0$. On propage l'observation :

$$x = [-2, 2] \bigcap x_a - \frac{\sin(\theta')(y - y_a)}{\cos(\theta')} = [-2, 2] \bigcap]-\infty; +\infty[= [-2, 2] \tag{5.60}$$

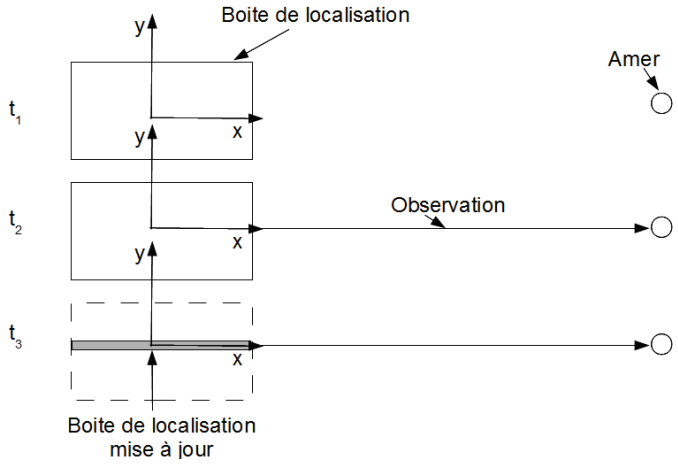


Figure 5.11: *Correction de la boite de localisation dans le cas 2D*

$$y = [-2,2] \bigcap y_a - \frac{\cos(\theta')(x - x_a)}{\sin(\theta')} = [-2,2] \bigcap [0,0] = [0,0] \tag{5.61}$$

$$x_a = [10,10] \bigcap x + \frac{\sin(\theta')(y - y_a)}{\cos(\theta)} = [10,10] \bigcap]-\infty;+\infty[= [10,10] \tag{5.62}$$

$$y_a = [0,0] \bigcap y + \frac{\cos(\theta')(x - x_a)}{\sin(\theta')} = [0,0] \bigcap [-2,2] = [0,0] \tag{5.63}$$

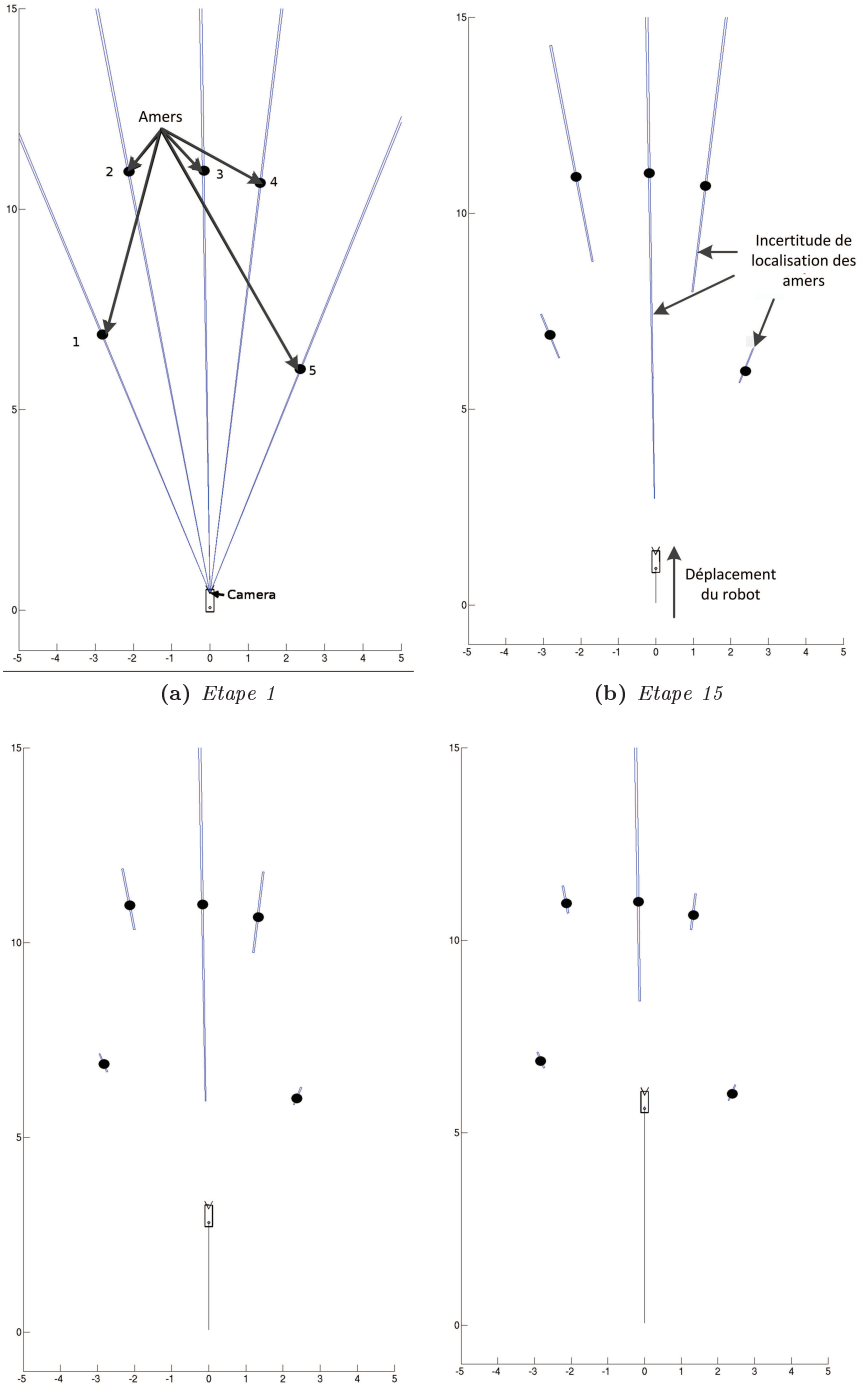
$$\begin{aligned} \theta' &= [-\pi/2, -\pi/2] \bigcap \arccos(\frac{-\sin(\theta')(y - y_a)}{x - x_a}) \\ &= [-\pi/2, -\pi/2] \bigcap \arccos([0,0]) = [0,0] \end{aligned} \tag{5.64}$$

$$\begin{aligned} \theta' &= [-\pi/2, -\pi/2] \bigcap \arcsin(\frac{\cos(\theta')(x - x_a)}{y_a - y}) \\ &= [-\pi/2, -\pi/2] \bigcap \arcsin(]-\infty;+\infty[) = [1,1] \end{aligned} \tag{5.65}$$

Cette observation a grandement amélioré la qualité de localisation suivant l'axe z. Cette correction est particulièrement efficace car la localisation de l'amer était connue exactement et aucun bruit d'observation n'a été ajouté.

5.6.3.2 Evolution de l'incertitude de localisation des amers

La figure 5.12 représente l'évolution de l'incertitude de localisation des amers en fonction du temps durant un trajet linéaire sans incertitude odométrique. La figure 5.12a représente l'initialisation des amers, ils sont initialisés à l'aide d'un cône infini : il n'y a aucune information de profondeur. Les figures 5.12b et 5.12c illustrent l'évolution de l'incertitude de localisation des amers au cours du déplacement du robot. On remarque que plus l'angle d'observation est important, plus l'incertitude se réduit



(a) *Etape 1*

(b) *Etape 15*

(c) *Etape 45*

(d) *Etape 90*

Figure 5.12: *Evolution de la carte de l'environnement*

rapidement. En particulier, les amers 1 et 5 sont localisés précisément très rapidement au contraire de l'amer 3 dont l'incertitude ne diminue que très peu car les observations sont toujours réalisées dans la même direction.

5.7 Résultats de localisation

La figure 5.13 représente les couloirs d'incertitude en fonction du numéro de l'étape pour la localisation dans l'environnement de test numéro deux (Section. 2.4.1.3). On remarque tout d'abord que le processus est stabilisé. En effet, la localisation du robot reste stable malgré le nombre de tour réalisé (un tour=500étapes).

La figure 5.14 représente le volume de la boite de localisation. Cette figure confirme que l'incertitude de localisation est stabilisée. Cependant, la fin du second tour est légèrement moins bonne que le premier tour.

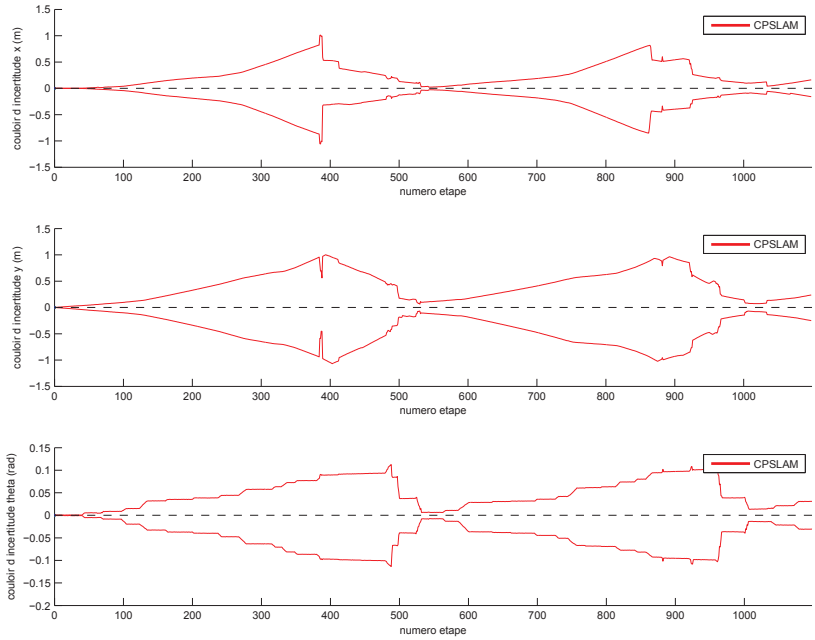


Figure 5.13: *Couloir d'incertitude du SLAM ensembliste*

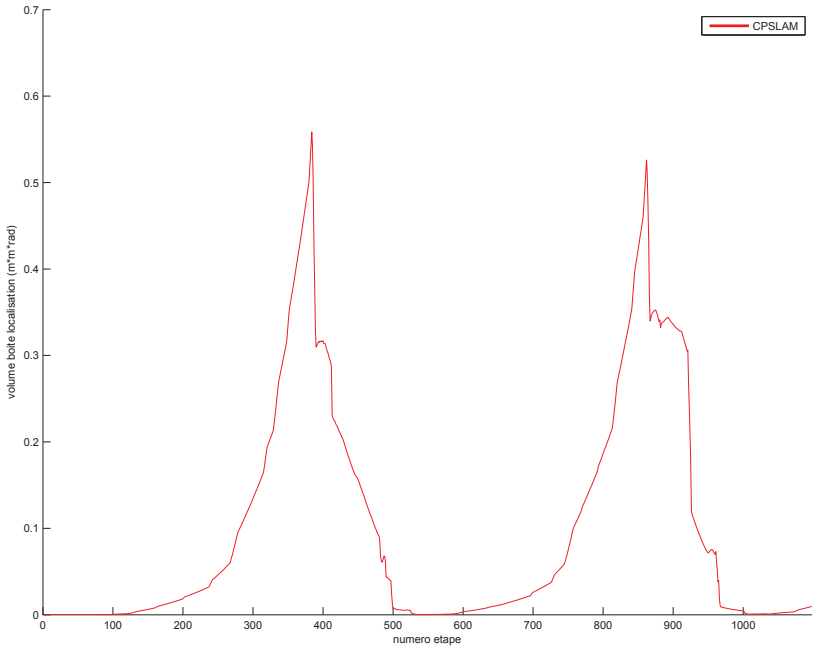


Figure 5.14: *Volume de la boite de localisation du SLAM ensembliste*

5.7.1 Mise en correspondance

Contrairement aux deux algorithmes précédents, l'extraction des primitives est réalisée par l'algorithme SURF (Speeded Up Robust Features) [81]. SURF est beaucoup plus performant que FAST pour la détection mais aussi l'appariement des amers. Ce choix a été nécessaire pour éviter les fausses détections ou les erreurs de mise en correspondance qui peuvent rendre le filtre inconsistant. L'étape de mise en correspondance est constituée de deux étapes:

– la projection du cône représentant l'amer sur l'image, elle définit une zone de recherche ($[u_{search}]$,$[v_{search}]$) :

$$\begin{pmatrix} [u_{search}] \\ [v_{search}] \end{pmatrix} = \begin{pmatrix} [c_u] - [f][k_u]\frac{[\mathbf{A}_{i,x}]}{[\mathbf{A}_{i,z}]} \\ [c_v] - [f][k_v]\frac{[\mathbf{A}_{i,y}]}{[\mathbf{A}_{i,z}]} \end{pmatrix} \tag{5.66}$$

avec $[\mathbf{A}_{i,x}]$, $[\mathbf{A}_{i,y}]$, $[\mathbf{A}_{i,z}]$ la position de l'amer dans le repère caméra (definie par l'équation 5.40) et $[c_u]$, $[f]$, $[k_u]$, $[c_v]$, $[k_v]$ les paramètres de la caméra.

– la sélection, dans la zone de recherche, de l'observation ($[u_{i,obs}]$, $[v_{i,obs}]$) qui a le descripteur SURF le plus proche de l'amer.

L'utilisation du détecteur et du descripteur SURF nous permet d'obtenir des résultats de détection et de mise en correspondance ayant un très faible taux d'erreur. La présence de mauvais appariement peut rendre les résultats inconsistants. Cependant, de nombreuses recherches sont menées pour améliorer le comportement de l'algorithme lors de la présence d'erreurs de détection ou de mise en correspondance. En particulier, Sliwka *et al.* [63, 116] proposent des algorithmes de gestions de données aberrantes.

5.8 Améliorations

L'algorithme défini reconstruit l'environnement tout en localisant le robot. Une amélioration de la gestion des données odométriques a été proposée pour réduire l'incertitude induite par les données proprioceptives et surtout leur modèle d'intégration. On propose plusieurs améliorations de l'algorithme ou de son implantation pour obtenir de meilleurs résultats tant au niveau de la localisation du robot que de la cartographie.

5.8.1 Orientation des repères

L'efficacité du modèle de prédiction et d'estimation dépend de l'orientation du robot. Par exemple, pour l'étape de prédiction, le volume de la boite de localisation va grossir plus vite si le robot se déplace avec un cap $\theta = 45°$ que suivant un des deux axes du repère. Comme pour l'étape de prédiction, la correction souffre du même problème d'orientation.
Nous avons utilisé la contribution proposée par Joly et Rives [61], qui propose de réaliser des changements de repère pour obtenir des résultats les moins pessimistes possibles, quasiment indépendants de l'orientation du robot. Nous avons vérifié expérimentalement que l'utilisation de ces rotations fournissaient des résultats au minimum aussi bon qu'une version sans rotation.
La figure 5.15 représente un cas où il est impossible de tirer profit d'une observation. Comme précédemment, on se place dans le cas où $u_{obs} = c_u$, la boite de localisation est un carré ($taille([x]) = taille([y])$) centré sur la localisation réelle. Numériquement, on prend $[\theta] = -\pi/4$, $[x] = [zy] = [-2, 2]$, $[x_a] = [y_a] = 10$.

En utilisant le système de contraintes et $\cos(-\pi/4) = -\sin(-\pi/4)$, on obtient :

$$\begin{aligned}
x &= x \bigcap x_a - \frac{\sin(\theta)(y - y_a)}{\cos(\theta)} = x \bigcap x_a + y - y_a \\
y &= y \bigcap y_a - \frac{\cos(\theta)(x - x_a)}{\sin(\theta)} = y \bigcap y_a + x - x_a \\
x_a &= x_a \bigcap x + \frac{\sin(\theta)(y - y_a)}{\cos(\theta)} = x_a \bigcap x - y + y_a \\
y_a &= y_a \bigcap y + \frac{\cos(\theta)(x - x_a)}{\sin(\theta)} = y_a \bigcap y - x + x_a \\
\cos(\theta) &= \cos(\theta) \bigcap \frac{-\sin(\theta)(y - y_a)}{x - x_a} \\
\sin(\theta) &= \sin(\theta) \bigcap \frac{\cos(\theta)(x - x_a)}{y_a - y}
\end{aligned} \tag{5.67}$$

Numériquement :

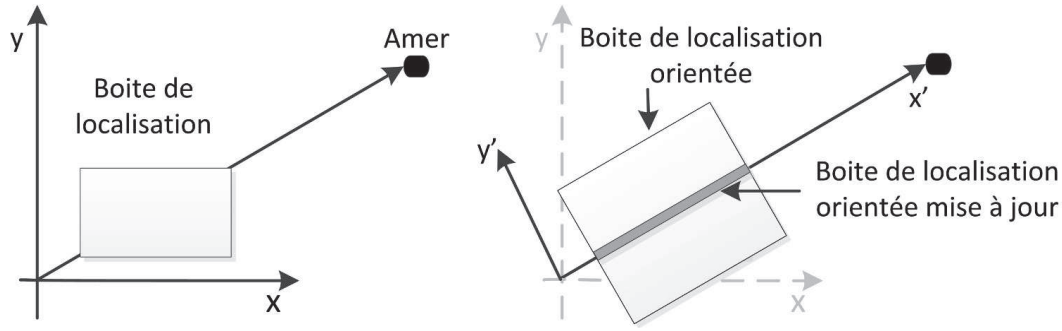


Figure 5.15: *Correction de la boite de localisation en utilisant un repère orienté*

$$
\begin{aligned}
x &= [-2,2] \bigcap [-2,2] = [-2,2] \\
y &= [-2,2] \bigcap [-2,2] = [-2,2] \\
x_a &= 10 \bigcap 10 + [-4,4] = 10 \\
y_a &= 10 \bigcap 10 + [-4,4] = 10 \\
\cos(\theta) &= -0.7 \bigcap \frac{0,70[-12,-8]}{[-12,-8]} = -0.7 \\
\sin(\theta) &= -0.7 \bigcap \frac{-0,70[-12,-8]}{[8,12]} = -0.7
\end{aligned}
\tag{5.68}
$$

Les intervalles ne peuvent être contractés en utilisant cette observation. Pour tirer profit de l'observation, on réalise un changement de repère autour de l'axe y ($(x,y) \rightarrow (x',y')$) (Figure 5.15) en utilisant l'équation :

$$
\begin{aligned}
\begin{pmatrix} [x'] \\ [y'] \end{pmatrix} &= \begin{bmatrix} \cos_{[]}\beta & -\sin_{[]}\beta \\ \sin_{[]}\beta & \cos_{[]}\beta \end{bmatrix} \begin{pmatrix} [x] \\ [y] \end{pmatrix} \\
[\theta'] &= [\theta] - \beta
\end{aligned}
\tag{5.69}
$$

Cette rotation ajoute du pessimisme : la boite de localisation est plus volumineuse après la rotation (Figure 5.15). Cependant, grâce à cette rotation, il est possible de contracter la boite de localisation à l'aide de l'observation en utilisant la même méthode que précédemment, la correction sera indépendante du cap ainsi que de l'angle d'observation.
Expérimentalement, on définit 10 angles de rotation possibles allant de $-\pi/2$ à $\pi/2$. Pour chaque observation, chaque angle est testé. A la fin de l'étape d'estimation, on garde la boite de localisation la plus petite parmi toutes les boites définies.

On réalise l'expérimentation correspondante à l'environnement deux. La figure 5.16 représente les résultats de l'algorithme utilisant 10 angles et la comparaison avec l'utilisation d'un seul angle d'orientation. La différence est relativement faible car le

mobile réalise un parcours principalement suivant les axes x et y. Les différences sont visibles en fin de tour (étape 450 et 900), l'utilisation de plusieurs angles d'orientation permet d'améliorer considérablement la localisation en temps réel. Cette correction est confirmée par le graphique du volume 5.17.

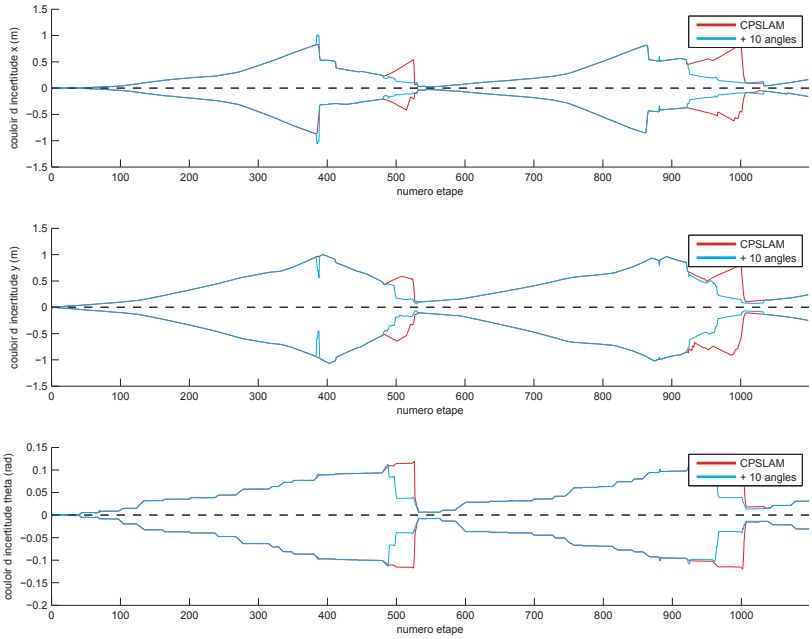


Figure 5.16: *Comparaison entre les couloirs d'incertitudes avec et sans inclinaison de repère*

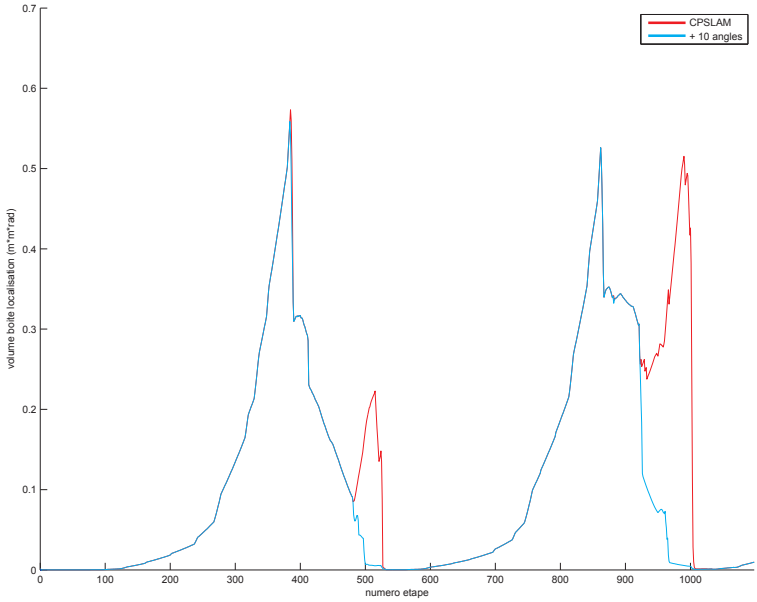


Figure 5.17: *Comparaison entre le volume de la boite de localisation avec et sans inclinaison de repère*

5.8.2 Post-Localisation

CP-SLAM utilise un ensemble de contraintes, construit au fil de l'expérimentation. A chaque étape, on ajoute de nouvelles contraintes dans le CSP : des contraintes sont ajoutées lors de l'étape de prédiction mais aussi lors de l'étape d'estimation pour chaque amer observé. En utilisant la propagation Backward/Forward, tous les intervalles sont contractés, y compris les intervalles des étapes précédentes. En effet, les boîtes de localisation, qui correspondent à chaque étape, peuvent être améliorées en utilisant une nouvelle observation. En particulier lors d'une fermeture de boucle, la localisation passée pourra être considérablement améliorée. De plus, l'amélioration de la localisation en raison d'une observation sera propagée dans le passé.
Par conséquence, on définit deux localisations differentes : la localisation en temps réel correspondant à la localisation actuelle et la post-localisation correspondant à la meilleure localisation utilisant l'ensemble de l'expérimentation.

La figure 5.18 représente les couloirs d'incertitude pour l'environnement 2 (Section. 2.4.1.3). La courbe pointillée représente la post localisation obtenue : elle confirme que l'utilisation du futur permet d'améliorer la localisation présente. En effet, les fermetures de boucles des étapes 500 et 1000 sont propagées en arrière et permettent d'accroître la qualité de la localisation.

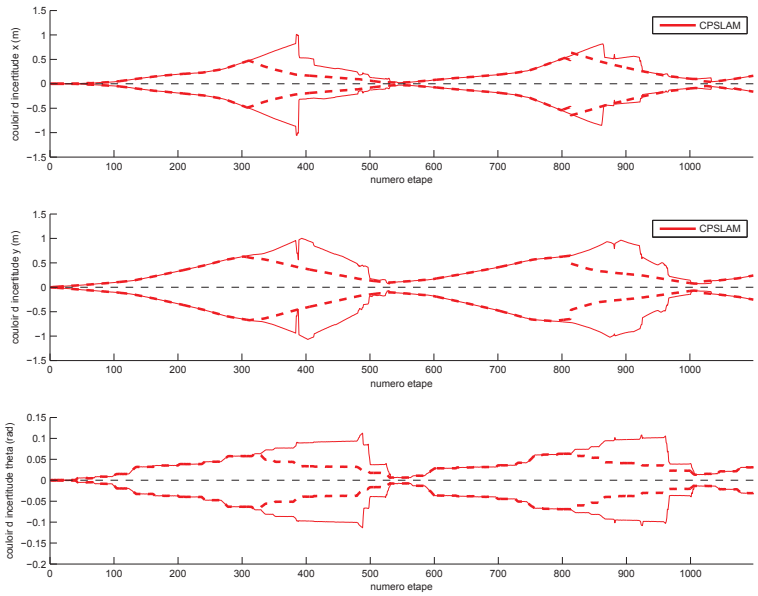


Figure 5.18: *Couloirs d'incertitudes incluant la post localisation pour l'environnement 2*

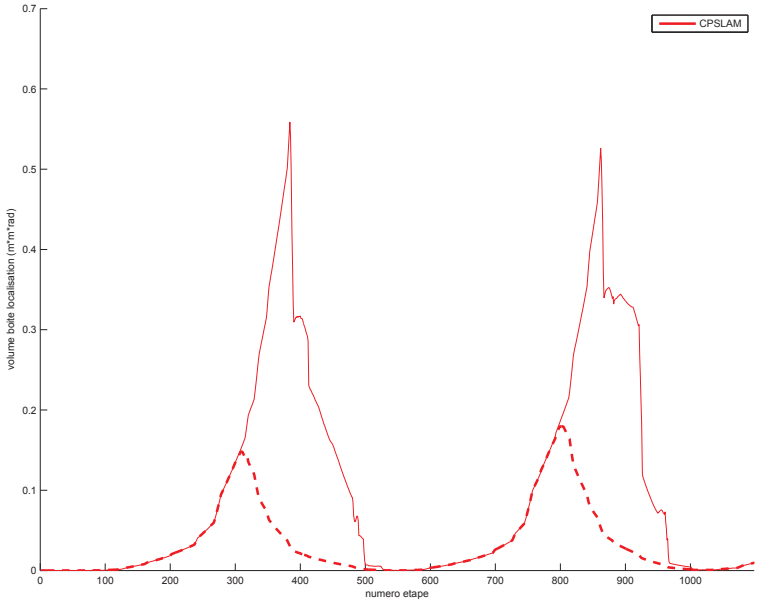


Figure 5.19: *Volume de la boite de localisation et post localisation pour l'environnement 2*

5.9 Validation du CPSLAM par Simulation

Nous avons défini un algorithme de localisation et de cartographie simultanées basé sur la théorie ensembliste et la propagation de contraintes. Cet algorithme localise un robot mobile tout en cartographiant son environnement sous forme d'une carte d'amers. Des premiers résultats ont montré la forte consistance de cette approche mais aussi un fort pessimisme des résultats. Plusieurs optimisations ont été apportées pour améliorer les résultats. On se propose d'évaluer notre algorithme dans nos environnements de test pour évaluer les résultats obtenus. Enfin, nous étudions l'influence du nombre d'amers, d'un bruit biaisé et d'une fenêtre de propagation sur la localisation du robot.

5.9.1 Influence du nombre d'amers

Les figures 5.20 et 5.21 représentent les résultats de l'environnement 2 en fonction du nombre d'amers (50, 100, 200). La première constatation est la grande consistance des résultats, les couloirs d'incertitude incluent tout au long de la simulation la valeur zéro, quel que soit le nombre d'amers. La représentation du volume de la boite de localisation montre l'importance du nombre d'amers dans l'environnement. Plus le nombre d'amers est important, plus la localisation est précise. Seul la précision de la localisation dépend du nombre d'amers, la consistance n'en dépend pas.

Les mêmes conclusions sont tirées des résultats des environnements 1 (Figures 6.5a et 6.5b) et 3 (Figures 6.6a et 6.6b). Les résultats sont consistants quelque soit le nombre d'amers et la qualité de localisation s'accroit avec le nombre d'amers.

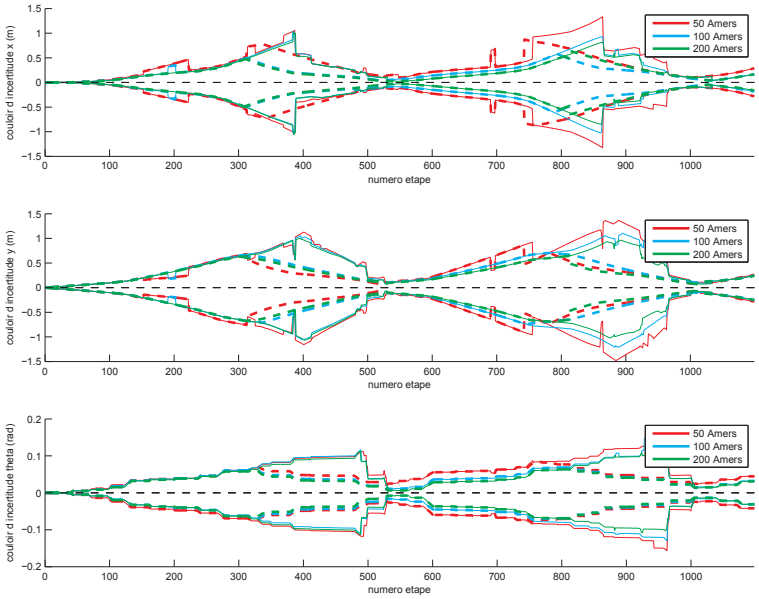


Figure 5.20: *Couloirs d'incertitudes incluant la post localisation pour l'environnement 2 en fonction du nombre d'amers*

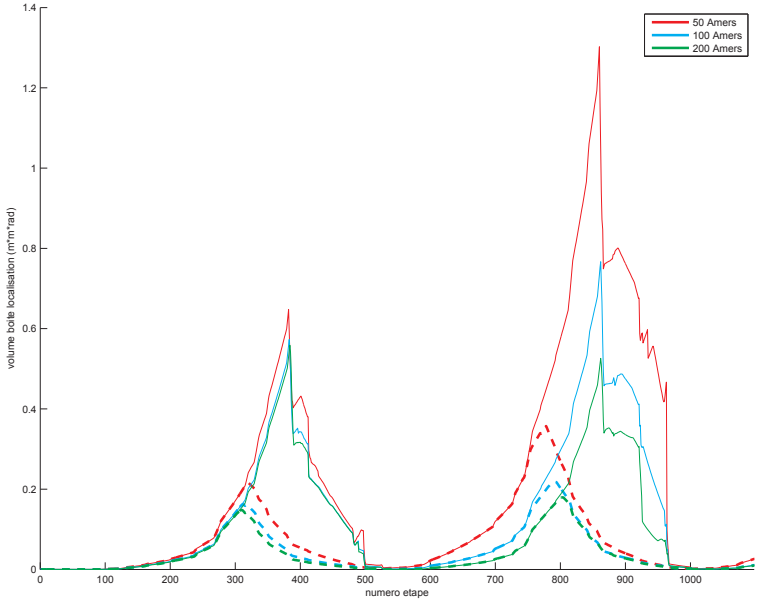


Figure 5.21: *Volume de la boite de localisation et post localisation pour l'environnement 2 en fonction du nombre d'amers*

5.9.2 Influence d'un biais

Contrairement à la théorie probabiliste, l'utilisation de la théorie ensembliste permet d'être insensible aux biais sur les données capteurs ou sur les constantes du modèle de déplacement. On étudie le comportement de l'algorithme avec un biais sur le rayon de la roue et sur l'entraxe. On fixe r=0.04-0.00025 et e=0.2-0.0005.

La figure 5.22 représente les couloirs d'incertitude des algorithmes biaisés ou non

biaisés. On remarque que l'algorithme n'est absolument pas affecté par la présence du biais. En effet, les couloirs sont encore consistants : la valeur 0 est incluse dans chaque couloir. De plus, la figure 5.23 montre que l'algorithme tire parti du biais et que le volume de la boite de localisation est plus petit dans le cas de données biaisées.

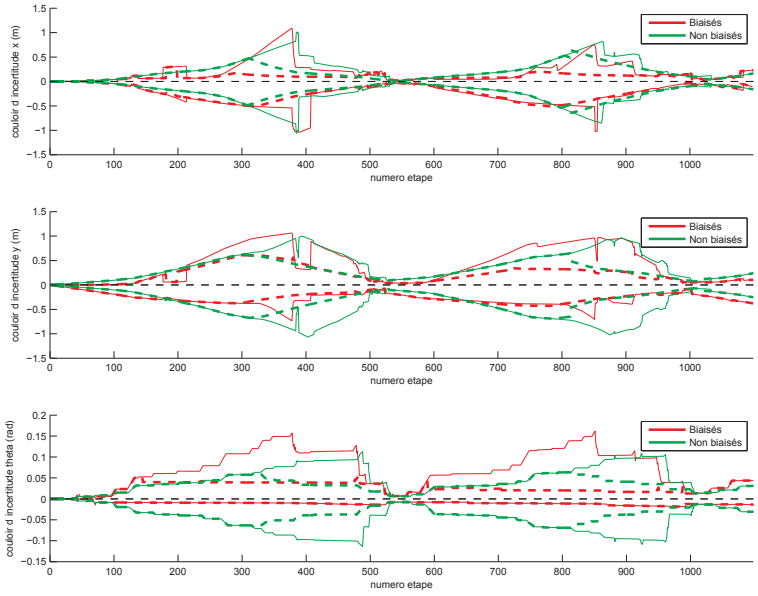


Figure 5.22: *Couloir d'incertitude en incluant un biais sur les paramètres du modèle de déplacement*

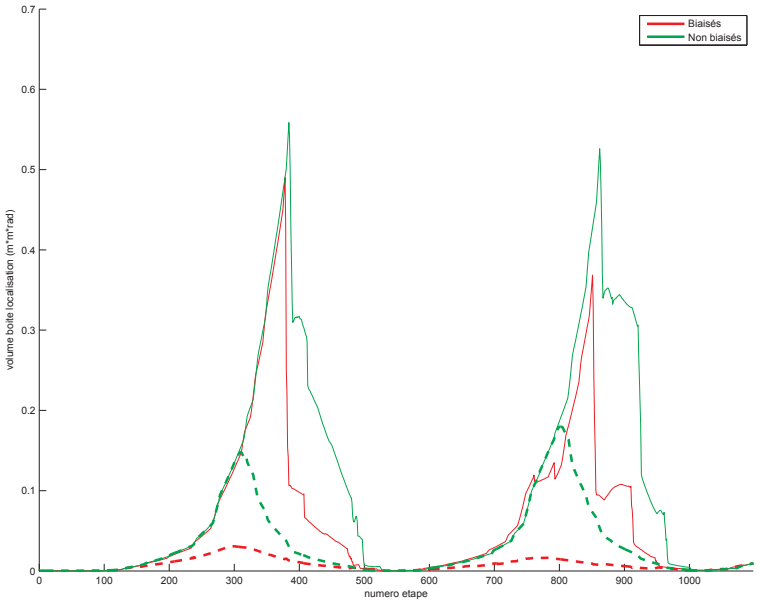


Figure 5.23: *Volume de la boite de localisation en incluant un biais sur les paramètres du modèle de déplacement*

5.9.3 Utilisation d'une fenêtre pour la propagation de contraintes

La propagation de contraintes utilise l'ensemble des contraintes de l'instant 0 à l'instant actuel. Plus le nombre d'étape augmente, plus le nombre de contraintes est important. Il est possible d'utiliser uniquement les "l" dernières contraintes, principalement pour limiter le temps de calcul. La figure 5.24 représente les volumes des boites de localisation pour une taille de fenêtre 0, 100, 200, 1100. La fenêtre 1100 correspond au cas sans fenêtre, il inclut l'ensemble des étapes de la simulation. On remarque tout d'abord que la localisation en temps réel lors du premier tour est approximativement la même quel que soit la fenêtre de propagation, les différents tracés sont confondus. Cependant, on remarque un gain concernant la post localisation pour l'algorithme réalisant une propagation complête. Pour le second tour, plus la fenêtre est grande, plus la localisation est précise. Il n'est cependant pas nécessaire d'utiliser une taille supérieure à 200 car les résultats sont identiques à l'utilisation d'une fenêtre complête.
L'utilisation d'une fenêtre de propagation peut poser des problèmes lors d'une fermeture de boucle. En effet en n'utilisant pas de fenêtre, l'algorithme peut tirer profit de la fermeture de boucle qui permet une amélioration significative de la carte de l'environnement mais aussi de sa localisation antérieure. Lors de l'utilisation d'une fenêtre le gain obtenu ne sera propablement pas repropagé jusqu'à l'étape correspondant à la fermeture de boucle. Les fermetures de boucles sont détectables, il est donc envisageable de réaliser une propagation de contraintes complête uniquement lors de la fermeture de boucle et des N étapes suivantes. Dans notre cas, le l optimal en terme de précision de localisation se situe autour de 250 étapes.

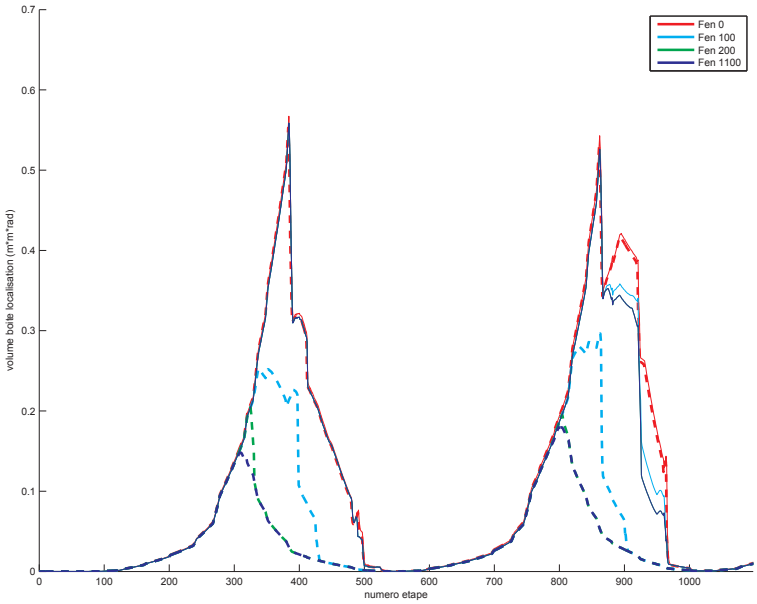


Figure 5.24: *Comparaison entre le volume de la boite de localisation en utilisant différente taille de fenêtre de propagation*

5.10 Validation Expérimentale du CPSLAM

Précédemment, nous avons évalué les performances de notre algorithme en simulation. On utilise maintenant notre jeu de données réalisé avec notre plateforme pour évaluer les résultats expérimentaux de notre algorithme. L'expérimentation consiste en trois tours de salle. Au cours de l'expérimentation, le robot a suivi des points de passage permettant de quantifier la qualité des résultats de localisation.

Expérimentalement, nous avons défini un seuil de mise en correspondance élevé. La définition de ce seuil permet de limiter les erreurs de mises en correspondance. De plus, l'appariement est réalisé en projetant l'incertitude de localisation sur l'image puis en sélectionnant l'observation.

Les figures 5.25 représentent les résultats expérimentaux. Durant le premier tour (Figure 5.25a), on remarque que le trajet défini par les boites de localisation suit correctement le trajet défini par les points de référence. De plus, on remarque que l'ensemble des points de référence sont inclus dans une boite de localisation. Contrairement à l'EKF-SLAM ou au FastSLAM, le filtre est consistant, l'incertitude de localisation inclut la position de référence. Enfin, on remarque que la post localisation bénéficie de la fermeture de boucle réalisée à la fin de la trajectoire. En effet, à la fin du premier tour, la boite de post localisation a une taille inférieur à celle de la localisation temps réel. Ce gain est dû à la fermeture de boucle du premier tour.

Les figures 5.25b et 5.25c représentent les résultats du second et troisième tour. Ces deux tours apportent des résultats similaires. Tout d'abord, la localisation du robot est correcte : les boites de localisation suivent le parcours tracé par les points de référence. De plus, la consistance des résultats est vérifiée, les points de référence sont inclus dans des boites. La figure 5.25e confirme la consistance des résultats en temps réel et en post localisation. La valeur zéro est incluse dans le couloir d'incertitude.

La principale différence de résultats entre le premier et les deux autres tours est la précision des résultats. En effet, les boites de localisation sont plus grandes lors des deux derniers tours. La figure 5.25d confirme que le volume de la boite de localisation est supérieur lors des deux derniers tours. Cependant, la courbe du volume montre le fait que l'algorithme tire profit de la fermeture de boucle. Le volume de la boite de localisation est fortement diminué lors de réobservation d'amers précédemment cartographiés. De plus, on remarque que la post localisation profite fortement de ces fermetures de boucles.

Notre expérimentation nous a permis de confirmer les résultats issus des simulations. La consistance des résultats a été montrée, cependant il est nécessaire d'utiliser un seuil d'appariement élevé pour éviter les problèmes de mises en correspondance. L'utilisation d'un algorithme de gestion des données aberrantes devrait permettre d'abaisser ce seuil et ainsi d'améliorer la précision des résultats. En effet, la seconde conclusion est le manque de précision de la méthode. Malgré plusieurs optimisations, l'incertitude de

localisation du robot reste relativement élevée. Nosu allons donc proposer plusieurs pistes d'optimisation afin d'améliorer la précision de la localisation.

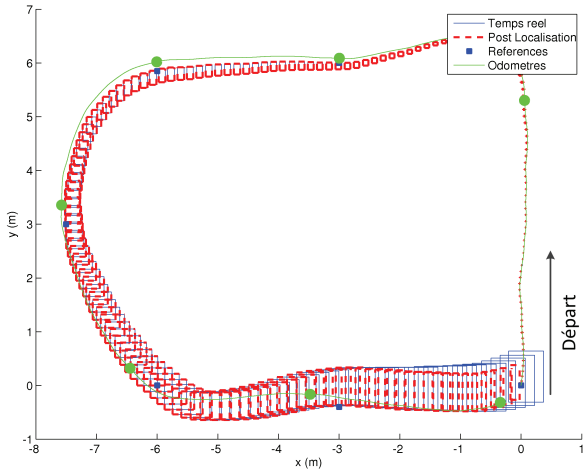
5.11 Utilisation de capteurs supplémentaires

Les résultats de notre système sont consistants mais relativement pessimistes, la zone de localisation du mobile mais aussi des amers est relativement grande. Plusieurs améliorations peuvent être utilisées pour essayer d'optimiser les résultats de l'algorithme.
Dans cette section, nous avons choisi d'étudier l'influence de capteurs supplémentaires sur les résultats de localisation. En effet, la propagation de contraintes utilise la redondance des données pour contracter les intervalles, l'utilisation de capteurs supplémentaires doit permettre d'améliorer les résultats. On propose l'étude en simulation de deux types de capteurs : une caméra 3D et un magnétomètre.

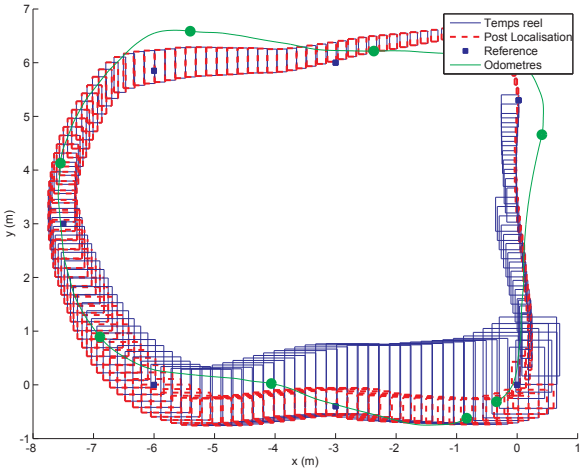
5.11.1 Caméra 3D

La commercialisation par Microsoft de la Kinect a révolutionné le domaine de la robotique. En particulier, de nombreux algorithmes de SLAM utilisant des données 3D sont en cours de développement dans le monde entier. Cette caméra fournit une image RGB de scène mais aussi une image de la profondeur. Chaque pixel est associé à une profondeur.
On modélise une caméra 3D fournissant la profondeur avec une précision dépendant de la profondeur de 2%. Plus l'objet est loin de l'objectif, plus la composante de profondeur est bruitée.

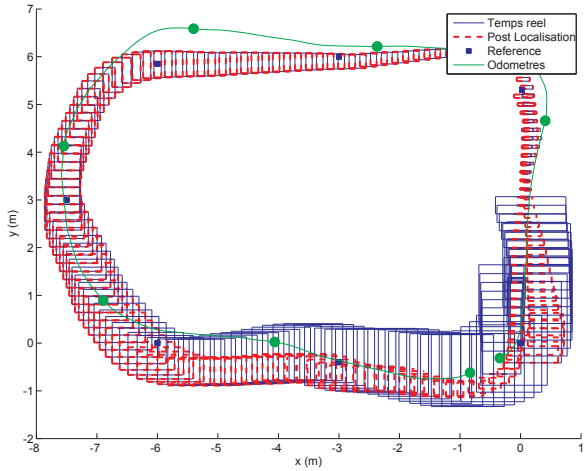
Les figures 5.26 et 5.27 représentent les résultats pour l'environement 2. L'utilisation d'une caméra 3D apporte de très bons résultats, la localisation est toujours consistante et l'incertitude est fortement réduite. Il est remarquable que la localisation en temps réel ait fortement améliorée grâce à ce capteur supplémentaire.



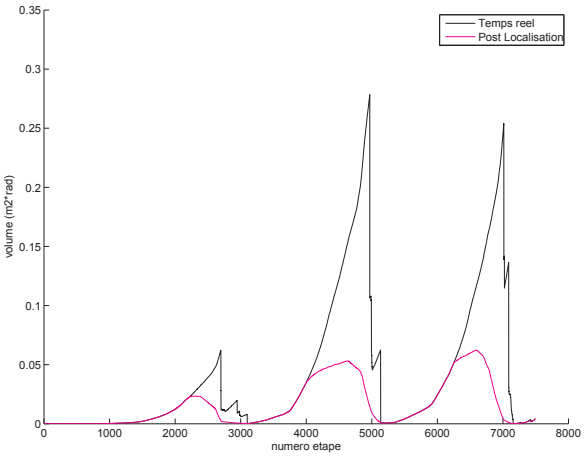
(a) *Résultats du premier tour*



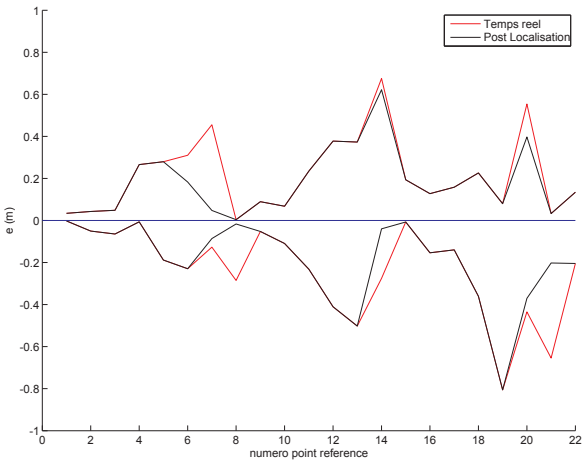
(b) *Résultats du second tour*



(c) *Résultats du troisième tour*



(d) *Volumes de la boite de localisation*



(e) *Couloirs d'incertitude*

Figure 5.25: *Résultats expérimentaux du CPSLAM*

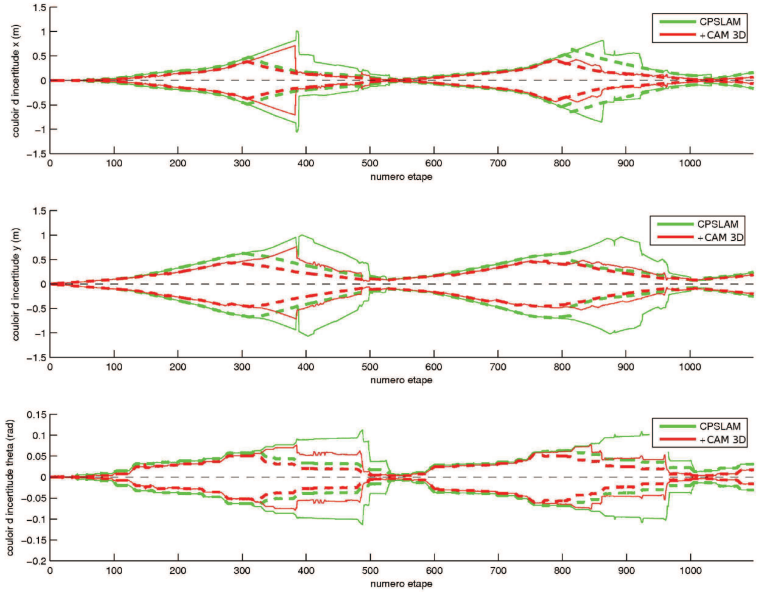


Figure 5.26: *Couloir d'incertitude en utilisant un caméra 3D*

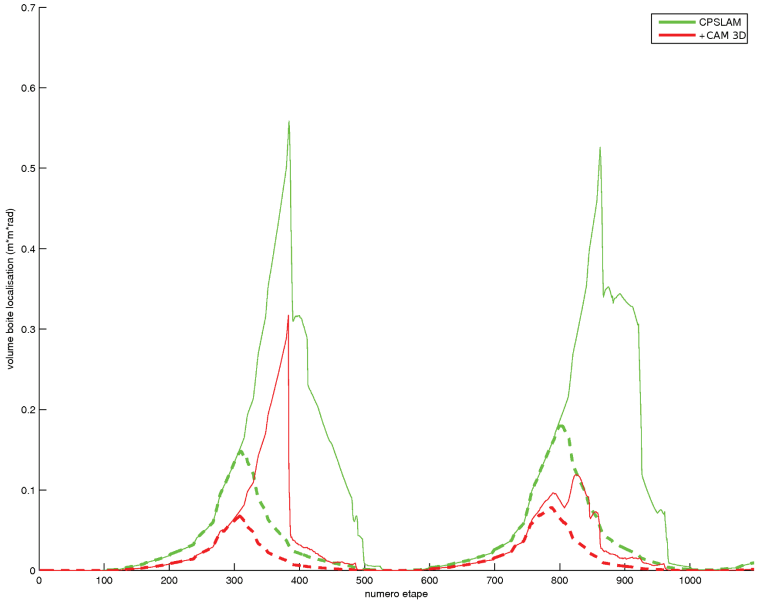


Figure 5.27: *Volume de la boite de localisation en utilisant un caméra 3D*

5.11.2 Magnétomètre

Notre système utilise les odomètres pour réaliser l'étape de prédiction. Ces odomètres sont des capteurs de déplacement relatif, ils fournissent le déplacement entre deux points. Nous proposons de modéliser l'utilisation d'un magnétomètre qui fournit l'orientation absolue du mobile. En plus des données odomètriques, on fournit à l'algorithmique l'orientation du robot $\theta_{magneto}$. On ajoute un bruit blanc gaussien, borné dans un intervalle d'erreur de $[-2.5°; 2.5°]$.

Les figures 5.28 et 5.29 représentent les résultats de l'algorithme incluant l'utilisation du magnétomètre. Les résultats sont consistants et de très bonne qualité. La localisation du cap de manière absolue permet de limiter considérablement l'augmentation du volume de la boite de localisation en particulier lors de la localisation en temps réel.

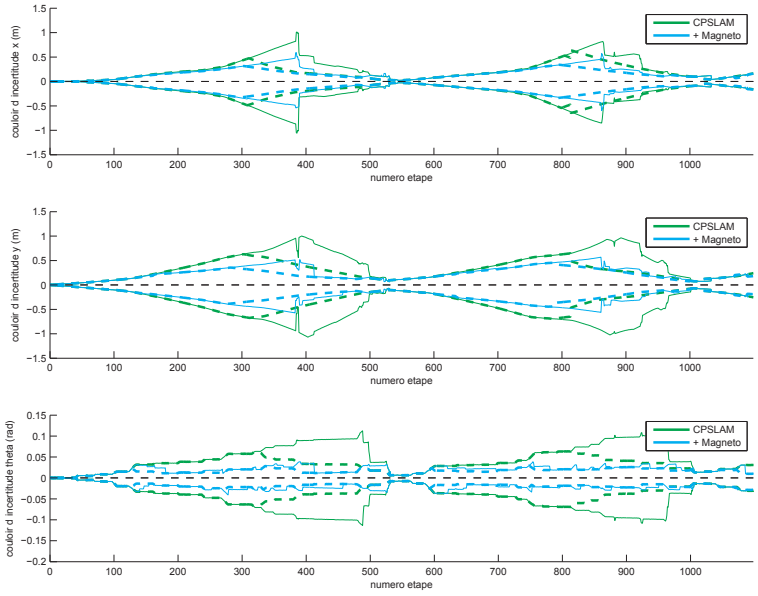


Figure 5.28: *Couloir d'incertitude en utilisant un magnétomètre*

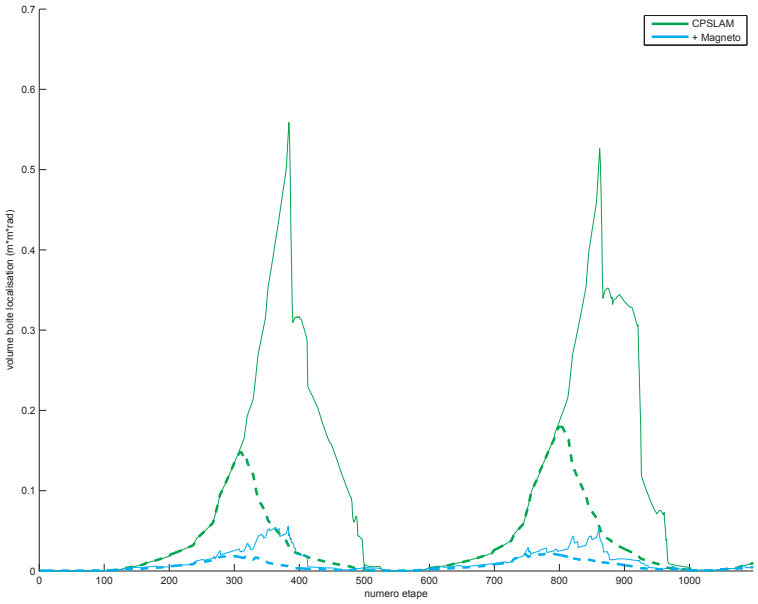


Figure 5.29: *Volume de la boite de localisation en utilisant un magnétomètre*

5.11.3 Utilisation conjointe des capteurs

On se propose d'étudier l'utilisation conjointe des capteurs : odométres, magnétomètre et caméra 3D. La figure 5.30 représente les couloirs d'incertitudes en fonction

de l'utilisation des capteurs. L'algorithme tire fortement profit de la redondance des données issues des capteurs. Les résultats de localisation sont beaucoup plus précis en utilisant l'ensemble des capteurs. Ces résultats sont confirmés par la figure 5.31 qui représente le volume de la boite de localisation. L'utilisation conjointe des capteurs crée un système de propagation de contraintes avec de nombreuses redondances qui permet d'améliorer la localisation.

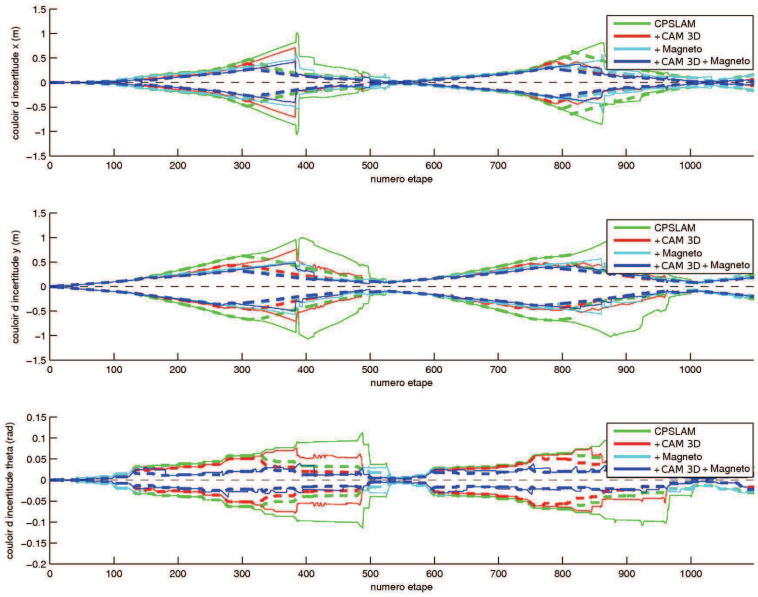


Figure 5.30: *Couloir d'incertitude en utilisant l'ensemble des capteurs*

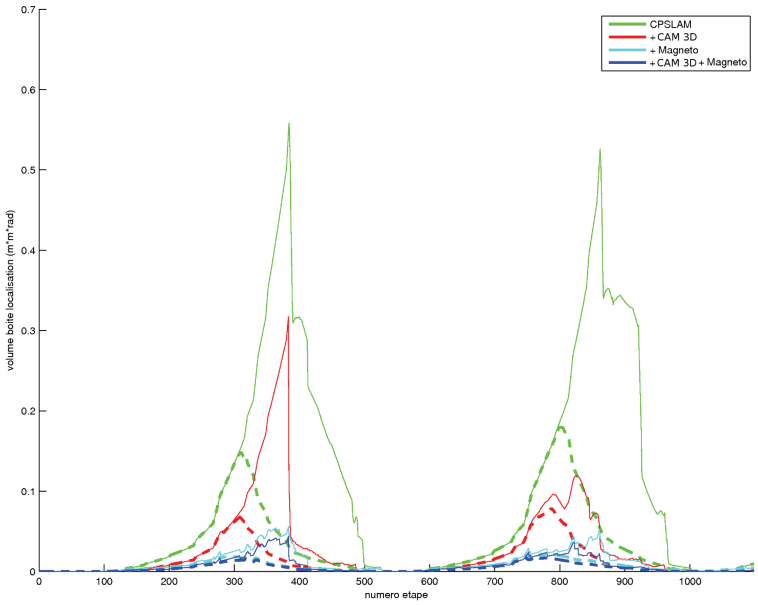


Figure 5.31: *Volume de la boite de localisation en utilisant l'ensemble des capteurs*

5.12 Stratégie d'implantation

Contrairement au chapitre 3 et 4, l'algorithme défini dans ce chapitre a été optimisé au niveau des résultats de localisation et de cartographie. Expérimentalement, en utilisant 10 angles de rotations, le temps de traitement pour une étape d'estimation est en moyenne de 4 secondes pour une expérimentation de 6500 étapes sur le processeur OMAP4430. De plus, le temps de traitement dépend très fortement du nombre d'étape. En effet, nous avons remarqué qu'il était quasiment obligatoire de propager les nouvelles observations sur l'ensemble du système de contraintes.

Le temps de traitement de l'algorithme est très important. Il est principalement dû au fait que pour réaliser des opérations sur des intervalles, il est nécessaire de calculer la borne inférieure de l'intervalle résultat en réalisant un arrondi par défaut. Alors que pour la borne supérieure, l'arrondi doit être réalisé par excès. Sur un processeur classique, le changement de mode d'arrondi réduit considérablement l'intérêt des pipelines. L'utilisation d'une architecture matérielle, dans ce cas particulier, peut avoir un effet très intéressant et améliorer considérablement le temps de traitement.

L'implantation de cet algorithme sur un système embarqué temps réel nécessitera l'utilisation d'une fenêtre de propagation. En effet, il est impossible de pouvoir propager l'ensemble des nouvelles observations sur l'intégralité du système de contraintes. Cependant, il est envisageable d'utiliser une méthode de détection de fermeture de boucle pour optimiser au mieux le gain de localisation obtenu à ce moment.

Enfin, une architecture matérielle pourrait être fortement bénéfique lors de l'utilisation de plusieurs angles de rotations. L'architecture pourra réaliser indépendamment la propagation pour chaque angle et ensuite sélectionner le meilleur. Cette parrallélisation pourra accélérer le temps de traitement.

5.13 Conclusions

Dans ce chapitre, nous avons défini, évalué et optimisé un algorithme de SLAM basé sur l'analyse par intervalles et la propagation de contraintes. L'utilisation de l'analyse par intervalles a déjà prouvé sa grande efficacité à produire des résultats consistants, à condition que les hypothèses de bruits bornés soient respectées.

Nous avons amélioré le modèle d'intégration des données odométriques. En effet, le modèle existant était très pessimiste par rapport à une intégration réelle. L'utilisation d'un modèle de déplacement amélioré nous a permis de réduire considérablement l'incertitude de localisation due à l'intégration des données odométriques.
Pour l'étape d'estimation, nous avons proposé une paramétrisation ensembliste des amers à l'aide de 5 paramètres. Cette paramétrisation réduit l'incertitude par rapport à l'utilisation d'une boite à trois dimensions. De plus, elle permet d'insérer directement les amers dans le système de contraintes et donc d'utiliser chaque observation

sans aucun délai. L'étude des systèmes existants nous a permis d'utiliser la méthode de rotation des repères pour améliorer les performances de localisation de l'algorithme. En effet, le mobile étant représenté par une boite, l'apport d'une observation dépendait de son angle de vision. Grâce à la rotation des repères, cette dépendance disparait.
Après avoir évalué les résultats de l'algorithme, nous avons constaté ces bonnes performances en terme de consistance mais cependant, les résultats sont assez pessimistes. La propagation de contraintes utilise la redondance des données pour réduire les incertitudes. Or, notre système était uniquement redondant dans le temps. Nous avons proposé d'utiliser des capteurs supplémentaires tels que une caméra 3D ou/et un magnétomètre. En simulation, leur utilisation a montré la grande efficacité de notre approche lors de la fusion de plusieurs capteurs redondants.

L'approche incluant de nombreux capteurs permet d'obtenir de meilleurs résultats. Malgré cela, elle met en avant la nécessité de gérer les valeurs de capteurs erronées. En effet, l'hypothèse principale de notre filtre est que les bruits des capteurs sont bornés. Si une donnée ne respecte pas cette hypothèse, les résultats seront inconsistants. Il serait intéressant d'inclure dans notre approche une gestion des données erronées pour en améliorer la robustesse. Cette gestion fait l'objet de multiples recherches Sliwka *et al.* [63, 116] qui pourraient être adaptées dans notre contexte.

Chapitre 6

Conclusion générale et perspectives

6.1 Conclusion et résumé des contributions

L'objectif principal de cette thèse consistait à définir des architectures pour des systèmes embarqués dédiés aux applications SLAM. L'étude des principaux algorithmes développés et des systèmes conçus ces dernières années nous a amené à proposer deux types de contributions. La première contribution a consisté à la conception de systèmes dans une approche "Adéquation Algorithme Architecture". La deuxième contribution a consisté au développement d'un algorithme de SLAM garantissant les résultats.

Dans l'optique de l'embarquabilité, nous avons conçu et évalué un système basé sur un algorithme probabiliste et une architecture multiprocesseur grand public. En effet, les nombreux systèmes de SLAM actuellement développés sont principalement basés sur l'utilisation de capteurs de très haut de gamme et de calculateurs très puissants. Notre objectif était de montrer la possibilité de concevoir un système utilisant une approche d'adéquation algorithme architecture. Le système conçu utilise des composants à bas coût, une architecture adaptée et une optimisation poussée de l'implantation de l'algorithme défini.
Nous avons donc suivi une démarche consistant à définir un système allant du capteur au calculateur. Pour cela, nous avons étudié les différents algorithmes ainsi que les différents capteurs à bas coûts utilisés dans la recherche mais aussi dans les produits grand public. Le choix de l'architecture a été réalisé en fonction des besoins de l'algorithme (EKF-SLAM) mais aussi des possibilités d'adaptation de son implantation. L'algorithme a été décomposé en blocs fonctionnels évalués indépendemment pour définir leurs besoins en terme de puissance de calcul ainsi que les possibilités d'implantations optimisées. L'évaluation temporelle sur des architectures embarquées multiprocesseurs nous a amené à proposer plusieures optimisations au niveau de l'implantation de l'algorithme, basées sur les capacités spécifiques des calculateurs que nous avons choisis. En particulier, la parallélisation et la répartition sur les différents processeurs de l'exécution des blocs fonctionnels a permis un gain de temps considérable. De plus, l'optimisation du calcul matriciel et du calcul de corrélation en utilisant des architectures vectorielles a permis d'avoir des temps de calculs répondant aux contraintes du temps réel.

Le premier système a mis en avant les possibilités offertes aujourd'hui par les nouvelles technologies des architectures bas-coût, en particulier pour la parallélisation multicoeurs et le calcul vectoriel. Cependant, ce système présente un inconvénient, celui de la reconstruction d'environnements de grandes tailles. Une des solutions consiste à utiliser un système de cartographie globale reconstruite à partir de plusieurs cartes locales.
Nous avons défini une architecture reconfigurable permettant l'implantation d'un al-

gorithme parallélisable (FastSLAM). Cet algorithme a été étudié en suivant la même méthodologie que l'algorithme précédent. Il a été décomposé en blocs fonctionnels pour identifier les principaux blocs consommateurs de ressources matérielles et de calculs. L'étude du FastSLAM et sa décomposition en blocs fonctionnels ont permis de définir une architecture reconfigurable parallélisée. L'architecture définie a été validée à l'aide d'une méthologie "Hardware In the Loop" qui valide des systèmes matériels en les incluant directement dans la chaîne de traitement logiciel. Enfin, cette architecture a été comparée à deux architectures basées sur des proceseurs type RISC : ARM CortexA8 et ARM Cortex A9. A fréquence égale, les résultats montrent un gain conséquent pour l'architecture reconfigurable.

La conception des deux systèmes précédents nous a confirmé la possibilité de concevoir un système de SLAM bas coût embarqué. Cependant, la consistance des méthodes de SLAM probabiliste a souvent été remise en question. Nous avons donc développé un algorithme de SLAM basé sur l'analyse par intervalles et la propagation de contraintes. L'utilisation de l'analyse par intervalles a montré de bons résultats en terme de consistance dans des applications de localisation, en particulier pour la localisation de mobile en environnement intérieur ou extérieur. Nous avons donc développé un algorithme basé sur cette théorie en utilisant les mêmes capteurs (odomètres et caméra). Nous avons proposé un algorithme qui fournit un résultat de localisation et de cartographie garanties. Une amélioration a été apportée pour utiliser au mieux les données odométriques et ainsi réduire le pessimisme dû à leur intégration. De plus, nous avons proposé une nouvelle paramétrisation et un système de post-localisation. Cet algorithme a été évalué à l'aide de notre simulateur pour mettre en évidence ses capacités à fournir des résultats consistants. L'analyse des résultats a confirmé la forte consistance de la localisation mais avec un certain degré de pessimisme. La propagation de contraintes étant basée sur l'utilisation de la redondance des données provenant de plusieurs capteurs, nous avons donc étudié, en simulation, l'influence de l'ajout de capteurs sur les résultats de localisation. L'utilisation d'une caméra 3D et d'un magnétomètre a montré la possibilité d'améliorer considérablement les résultats en termes de précision .

En résumé, ce travail de thèse a porté sur l'adéquation algorithme architecture dans le domaine du SLAM. En effet, des contributions ont été apportées sur l'implantation d'un algorithme sur une architecture cible mais aussi sur la conception d'une architecture dédiée. Des contributions ont aussi été apportées sur des concepts algorithmiques pour pallier à des problémes récurrents des algorithmes de SLAM telle que la consistence des résultats.

6.2 Perspectives

L'adéquation algorithme architecture est un domaine en plein essort. Cependant, dans le cas du SLAM, cette problématique est encore rarement étudiée. Malgré tout, des optimisations sont fréquemment proposées pour améliorer les algorithmes mais aussi les architectures des systèmes dédiés. A l'issu de cette thèse, nous envisageons les perspectives suivantes :

Conception d'une architecture matérielle intégrant des briques pour le traitement d'images

Notre approche a mis en avant la possibilité d'obtenir des performances satisfaisantes dans le cadre du SLAM. Ces performances pourront être accrues en utilisant une nouvelle méthode de paramétrisation des amers ou encore une méthode plus élaborée de mise en correspondance par recherche active. Ces différentes améliorations algorithmiques ont montré aujourd'hui leurs intérêts au niveau des résultats de localisation. Cependant, leur intégration sur des systèmes embarqués doit prendre en considération la problématique d'adéquation algorithme architecture.

Parallélement à l'amélioration des résultats de localisation, l'étude d'architectures matérielles dédiées aux traitements d'images devrait permettre de concevoir une architecture matérielle incluant le système de SLAM mais aussi l'ensemble des briques detraitement d'images.

Développement d'un système de SLAM distribué

Généralement, le système de SLAM est centralisé sur un robot. On pourra entreprendre des recherches dans le domaine du SLAM distribué afin de concevoir une architecture matérielle distribuée sur plusieurs robots. Chaque robot pourra communiquer sa carte d'environnement mais aussi bénéficier de la cartographie des autres robots (cas de tremblement de terre). Le SLAM distribué reste encore aujourd'hui peu étudié.

Vers une architecture dédiée au CPSLAM

L'algorithme de SLAM ensembliste développé a la capacité de pouvoir bénéficier de nouveaux capteurs. Il est nécessaire de valider une approche multi-capteurs. Cette étude devrait montrer la nécessité de gérer les données aberrantes émises par les capteurs. De plus, l'utilisation de pavage pour représenter la localisation du mobile et/ou celle des amers semble être une méthode intéressante pour réduire le pessimisme dû à l'intégration des données capteurs. Cependant cette adaptation nécessitera de profondes modifications algorithmiques. Enfin, la conception d'une architecture matérielle devrait permettre d'obtenir des performances compatibles avec le temps-réel.

SLAM en environnement extérieur

A moyen terme, l'ensemble des systèmes conçus pourront être adaptés à un véhicule évoluant dans un environnement extérieur. Cette intégration nécessitera de repenser en partie la conception de l'algorithme et de l'architecture. L'utilisation de capteurs adaptés tel qu'un GPS sera indispensable. De profondes modifications algorithmiques pour pouvoir gérer les objets mobiles devraient permettre d'obtenir un système temps réel dans une approche d'adéquation algorithme architecture. Des recherches sont actuellement menées sur le SLAM dans un environnement incluant des objets mobiles. Ces perspectives pourront être étudiées et améliorées.

Conception de robots autonomes

L'objectif à plus long terme est d'asurer un certain degré d'autonomie des robots. Les résultats d'un algorithme de SLAM permettent à un robot de se localiser dans un environnement inconnu. Cependant, l'objectif est de fournir au robot la capacité de réaliser un maximum d'actions de manière totalement autonome.

L'utilisation conjointe d'un algorithme de SLAM, de planification de trajectoire et d'une intelligence artificielle devrait permettre d'accroître les capacités des robots.

Annexes

6.3 Résultats expérimentaux de l'EKF-SLAM

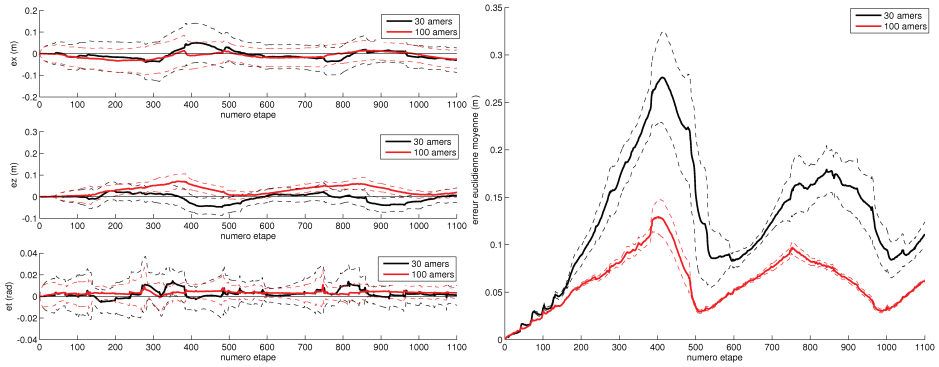
Dans le chapitre 3, nous avons évalué les performances de l'EKF-SLAM en simulation. Trois environnements ont été définis afin d'évaluer le comportement de l'algorithme en fonction de la taille de l'environnement. Les résultats correspondants à l'environnement 1 ont été inclus dans le chapitre 3.4.

6.3.1 Environnement 2

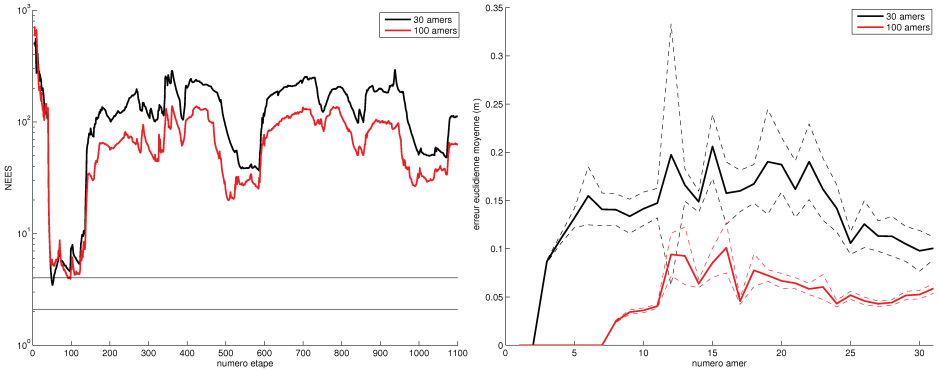
L'environnement 2 est un environnement de taille moyenne (15×15 m) incluant un mur. Le mur est destiné à générer des occultations. Le robot n'est pas capable d'observer les amers à n'importe quel endoit de l'environnement.

Les résultats de localisation sont relativement précis. La figure 6.1b confirme que l'erreur euclidienne entre la position estimée et la référence est faible. Pour 30 amers, l'erreur commise est d'environ 15 cm alors que pour 100 amers l'erreur moyenne diminue à seulement 7.5 cm. On remarque que les résultats d'erreurs sont reproductibles, l'incertitude sur l'erreur euclidienne est relativement faible.

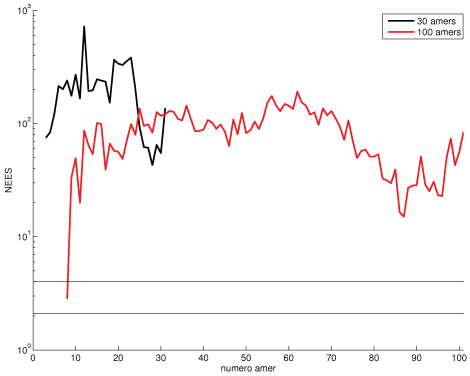
Nous avons vu que la précision de l'algorithme est bonne, cependant il est nécessaire de vérifier la consistance des résultats. En effet, l'incertitude de localisation du robot doit inclure la localisation de référence. Les figures 6.1a et 6.1c permettent d'estimer la consistance de la localisation. La consistance des résultats est relativement mauvaise, les couloirs d'incertitudes n'incluent pas la valeur zéro à chaque instant. Ces inconsistances sont confirmées par la figure 6.1c, le NEES n'est pas inclu dans les bornes de consistance. Enfin la consistance des amers (Figure 6.1e) reflète la consistance de la localisation.



(a) *Couloir d'incertitude sur la position x, y, θ* (b) *Erreur euclidienne de la position x, y, θ*



(c) *NEES de la position x, y, θ* (d) *Erreur euclidienne de la position des amers*



(e) *NEES pour la position des amers*

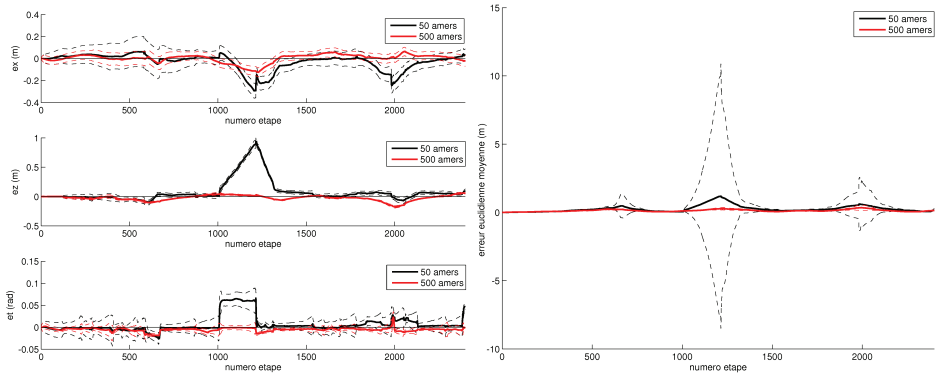
Figure 6.1: *Résultats de l'EKF-SLAM pour l'environnement 2*

6.3.2 Environnement 3

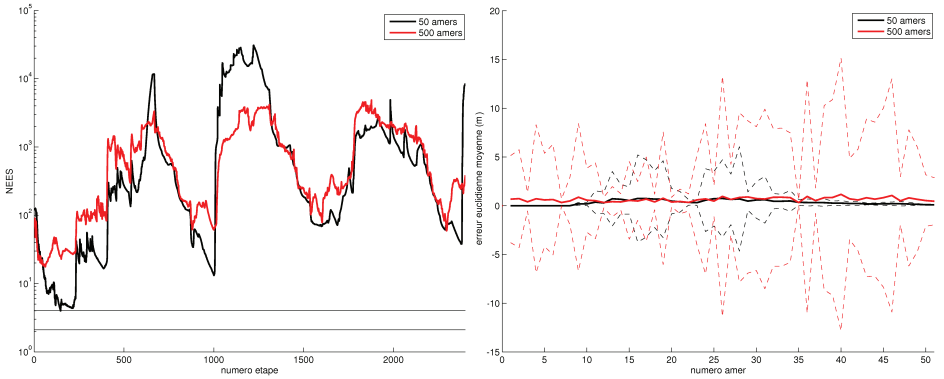
L'environnement 3 est un environnement de plus grande taille (20 ×20m) incluant de nombreux murs. Cet environnement est le plus compliqué des trois environnements de test.

Nous avons remarqué précédemment que notre l'algorithme n'était pas adapté au grand environnement. Cette simulation le confirme, l'erreur euclidienne sur la position du robot (Figure 6.2b) est importante et surtout instable. Ces problèmes se retrouvent aussi pour les couloirs d'incertitude (Figure 6.2a). Notre algorithme n'est pas adapté

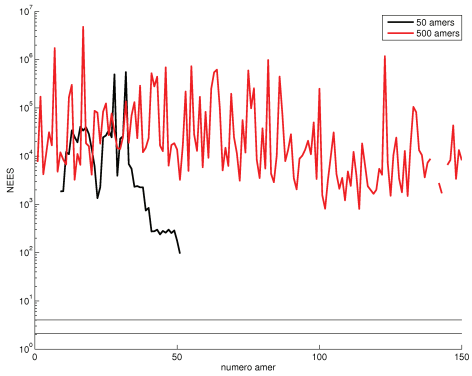
à ce type d'environnement complexe.



(a) *Couloir d'incertitude sur la position x, y, θ* (b) *Erreur euclidienne de la position x, y, θ*



(c) *NEES de la position x, y, θ* (d) *Erreur euclidienne de la position des amers*



(e) *NEES pour la position des amers*

Figure 6.2: *Résultats de l'EKF-SLAM pour l'environnement 3*

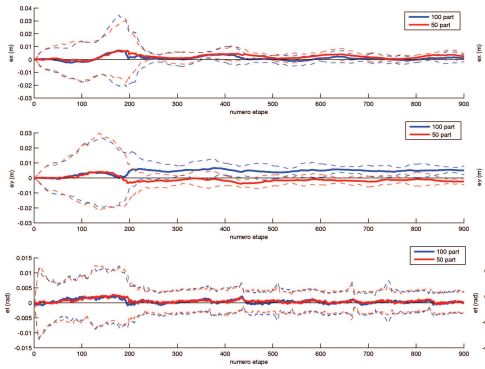
6.4 Résultats expérimentaux du FastSLAM

Dans le chapitre 4, nous avons étudié le FastSLAM. L'algorithme a été évalué en simulation et les résultats de l'environnement 2 ont été étudiés au paragraphe 4.3.1. On étudie maintenant les résultats des environnements 1 et 3.

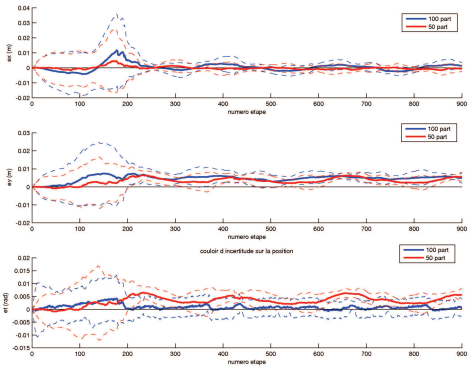
6.4.1 Environnement 1

L'environnement 1 est une petite salle sans aucun mur. Les résultats du FastSLAM sont de bonne qualité, quelque soit le nombre d'amers présents dans la salle.
L'erreur euclidienne commise sur la localisation du robot est très faible (Figure 6.3c et 6.3d). Elle se situe autour de 2cm et reste stable durant l'ensemble de l'expérimentation.

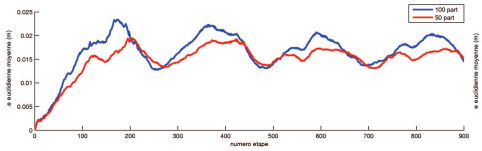
Concernant la consistance des résultats, l'algorithme souffre de problème de consistance comme l'EKF-SLAM. Cependant, on peut observer à la figure 6.3a et 6.3a que la première fermeture de boucle est correctement réalisée (étape 200). L'incertitude de localisation diminue fortement à cette étape. Cependant, après la fermeture de boucle, le filtre devient fortement inconsistant. Cette conclusion est confirmée par l'étude du NEES (Figure 6.3e et 6.3e). Le NEES augmente de manière très importante autour de l'étape 200. L'algorithme devient trop optimiste à partir de la fermeture de boucle.



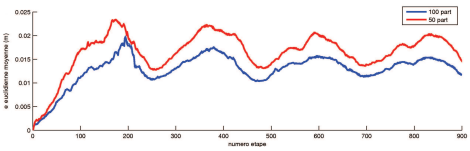
(a) *Couloir d'incertitude sur la position x, y, θ pour l'environnement 1 rempli de 100 amers*



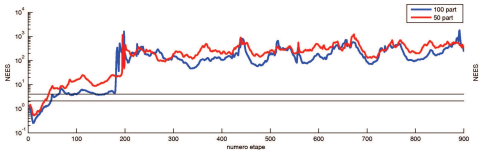
(b) *Couloir d'incertitude sur la position x, y, θ pour l'environnement 1 rempli de 500 amers*



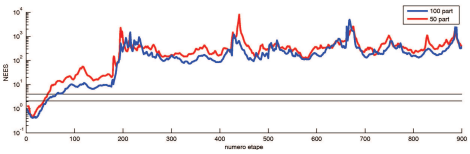
(c) *Erreur euclidienne de la position x, y, θ pour l'environnement 1 rempli de 100 amers*



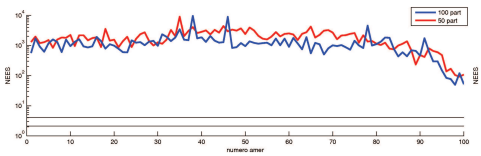
(d) *Erreur euclidienne de la position x, y, θ pour l'environnement 1 rempli de 100 amers*



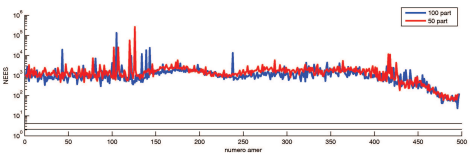
(e) *NEES de la position x, y, θ pour l'environnement 1 rempli de 100 amers*



(f) *NEES de la position x, y, θ pour l'environnement 1 rempli de 500 amers*



(g) *NEES pour la position des amers pour l'environnement 1 rempli de 100 amers*



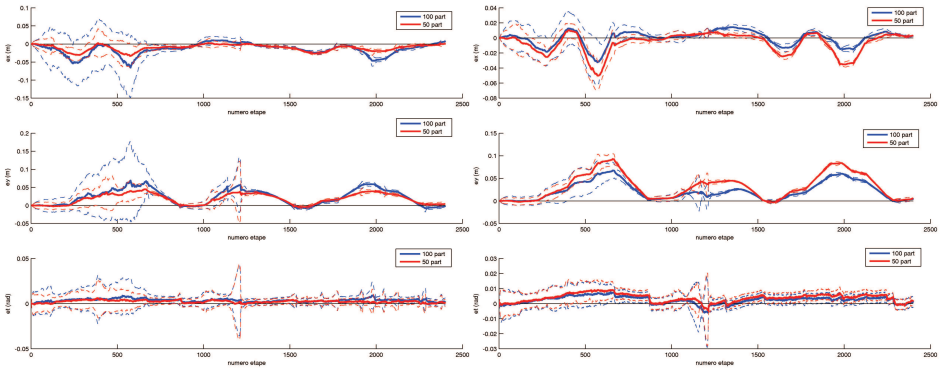
(h) *NEES pour la position des amers pour l'environnement 1 rempli de 500 amers*

Figure 6.3: *Résultats du FastSLAM pour l'environnement 1*

6.4.2 Environnement 3

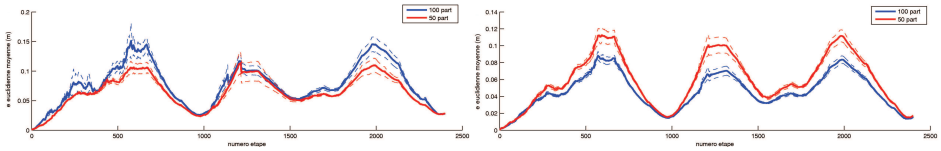
L'environnement 3 est un environnement complexe ayant une taille de 20 ×20m. L'EKF-SLAM n'était pas capable d'apporter de bons résultats pour cet environnement.

Les figures 6.4c et 6.4d montrent que, contrairement à l'EKF-SLAM, le FastSLAM a de bons résultats pour cet environnement. L'erreur euclidienne est bornée et diminue en fonction de la hausse du nombre d'amers. Cependant, les problèmes de consistances évoqués précédemment se confirment (Figure 6.4a et 6.4b). Le NEES a une valeur très élevé pour la position du robot (Figure 6.4e et 6.4f) mais aussi pour les amers (Figure 6.4g et 6.4h).



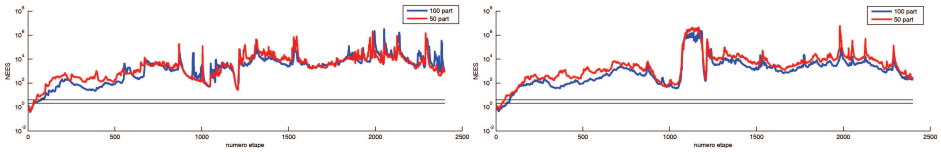
(a) *Couloir d'incertitude sur la position x, y, θ pour l'environnement 3 rempli de 100 amers*

(b) *Couloir d'incertitude sur la position x, y, θ pour l'environnement 3 rempli de 500 amers*



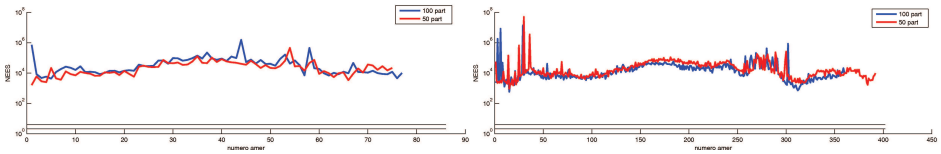
(c) *Erreur euclidienne de la position x, y, θ pour l'environnement 3 rempli de 100 amers*

(d) *Erreur euclidienne de la position x, y, θ pour l'environnement 3 rempli de 100 amers*



(e) *NEES de la position x, y, θ pour l'environnement 3 rempli de 100 amers*

(f) *NEES de la position x, y, θ pour l'environnement 3 rempli de 500 amers*



(g) *NEES pour la position des amers pour l'environnement 3 rempli de 100 amers*

(h) *NEES pour la position des amers pour l'environnement 3 rempli de 500 amers*

Figure 6.4: *Résultats du FastSLAM pour l'environnement 3*

6.5 Problème de satisfaction de contraintes

Dans le Chapitre 5, nous avons défini et resolu un exemple numérique d'un probléme de satisfaction de contraintes. On se propose de détailler l'ensemble des étapes de calculs.

Le CSP est :

$$\begin{cases} [z] = [x] + \ln_{[]}([y]) \\ [y] = [z]^2 \end{cases} \tag{6.1}$$

avec $x \in [1; 3.91]$, $y \in [3; 50]$, $z \in [2.09; 5]$. On obtient alors la propagation suivante:

$$\begin{array}{rcl}
[y] & \in & [y] \bigcap [z]^2 \in [3;50] \bigcap [2.09;5]^2 \in [4.36;25] \\
[z] & \in & [z] \bigcap ([x] + \ln_{[]}([y])) \in [2.09;5] \bigcap [1;3.91] + \ln_{[]}([4.36;25]) \in [2.47;5] \\
[y] & \in & [y] \bigcap [z]^2 \in [4.36;25] \bigcap [2.47;5]^2 \in [6.10;25] \\
[z] & \in & [z] \bigcap ([x] + \ln_{[]}([y])) \in [2.47;5] \bigcap [1;3.53] + \ln_{[]}([6.10;25]) \in [2.80;5] \\
[x] & \in & [x] \bigcap ([z] - \ln_{[]}([y])) \in [1;3.53] \bigcap [2.80;5] - \ln_{[]}([6.10;25]) \in [1;3.20] \\
[y] & \in & [y] \bigcap [z]^2 \in [6.10;25] \bigcap [2.80;5]^2 \in [7.84;25] \\
[z] & \in & [z] \bigcap ([x] + ln([y])) \in [2.80;5] \bigcap [1;3.20] + ln([7.84;25]) \in [3.05;5] \\
[x] & \in & [x] \bigcap ([z] - ln([y])) \in [1;3.20] \bigcap [3.05;5] - ln([7.84;25]) \in [1;2.95] \\
[y] & \in & [y] \bigcap [z]^2 \in [7.84;25] \bigcap [3.05;5]^2 \in [9.30;25] \\
[z] & \in & [z] \bigcap ([x] + ln([y])) \in [3.05;5] \bigcap [1;2.95] + ln([9.30;25]) \in [3.23;5] \\
[x] & \in & [x] \bigcap ([z] - ln([y])) \in [1;2.95] \bigcap [3.23;5] - ln([9.30;25]) \in [1;2.95] \\
[y] & \in & [y] \bigcap [z]^2 \in [9.30;25] \bigcap [3.23;5]^2 \in [10.43;25] \\
[z] & \in & [z] \bigcap ([x] + ln([y])) \in [3.23;5] \bigcap [1;2.95] + ln([10.43;25]) \in [3.34;5] \\
[x] & \in & [x] \bigcap ([z] - ln([y])) \in [1;2.95] \bigcap [3.34;5] - ln([10.43;25]) \in [1;2.59] \\
[y] & \in & [y] \bigcap [z]^2 \in [10.43;25] \bigcap [3.41;5]^2 \in [11.62;25] \\
[z] & \in & [z] \bigcap ([x] + ln([y])) \in [3.41;5] \bigcap [1;2.59] + ln([11.62;25]) \in [3.44;5] \\
[x] & \in & [x] \bigcap ([z] - ln([y])) \in [1;2.59] \bigcap [3.44;5] - ln([11.62;25]) \in [1;2.55] \\
[y] & \in & [y] \bigcap [z]^2 \in [11.62;25] \bigcap [3.44;5]^2 \in [11.90;25] \\
[z] & \in & [z] \bigcap ([x] + ln([y])) \in [3.44;5] \bigcap [1;2.55] + ln([11.90;25]) \in [3.47;5]
\end{array} \tag{6.2}$$

$$[x] \in [x] \bigcap ([z] - ln([y])) \in [1; 2.55] \bigcap [3.47; 5] - ln([11.90; 25]) \in [1; 2.53] \quad (6.3)$$

$$[y] \in [y] \bigcap [z]^2 \in [11.90; 25] \bigcap [3.47; 5]^2 \in [12.04; 25] \quad (6.4)$$

$$[z] \in [z] \bigcap ([x] + ln([y])) \in [3.47; 5] \bigcap [1; 2.53] + ln([12.04; 25]) \in [3.48; 5] \quad (6.5)$$

$$[x] \in [x] \bigcap ([z] - ln([y])) \in [1; 2.53] \bigcap [3.48; 5] - ln([12.04; 25]) \in [1; 2.52] \quad (6.6)$$

$$[y] \in [y] \bigcap [z]^2 \in [12.04; 25] \bigcap [3.48; 5]^2 \in [12.11; 25] \quad (6.7)$$

$$[z] \in [z] \bigcap ([x] + ln([y])) \in [3.48; 5] \bigcap [1; 2.52] + ln([12.11; 25]) \in [3.49; 5] \quad (6.8)$$

$$[x] \in [x] \bigcap ([z] - ln([y])) \in [1; 2.52] \bigcap [3.49; 5] - ln([12.11; 25]) \in [1; 2.51] \quad (6.9)$$

$$[y] \in [y] \bigcap [z]^2 \in [12.11; 25] \bigcap [3.49; 5]^2 \in [12.18; 25] \quad (6.10)$$

$$[z] \in [z] \bigcap ([x] + ln([y])) \in [3.49; 5] \bigcap [1; 2.51] + ln([12.18; 25]) \in [3.49; 5] \quad (6.11)$$

Finalement, on obtient les intervalles $[x] \in [1; 2.51]$, $[y] \in [12.18; 25]$ et $[z] \in [3.49; 5]$.

6.6 Résultats expérimentaux du SLAM ensembliste

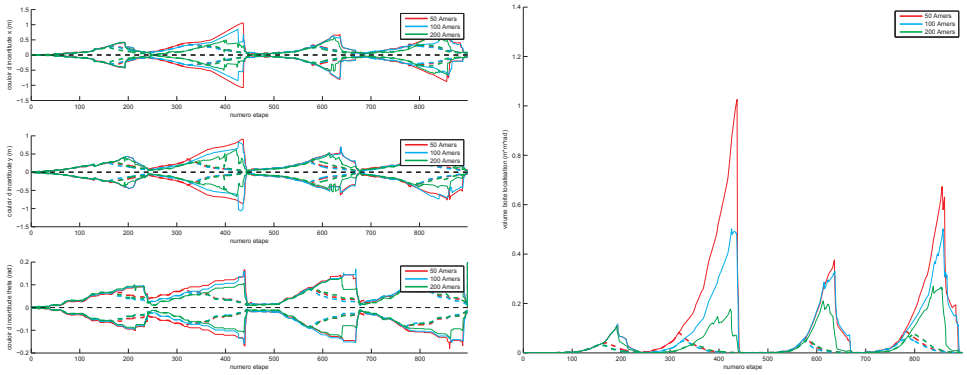
Dans le chapitre 5, nous avons défini et évalué un algorithme de SLAM ensembliste. Au paragraphe 5.9, nous avons évalué cet algorithme en utilisant l'environnement 2. On étudie maintenant les résultats obtenus dans les deux autres environnements.

6.6.1 Environnement 1

Pour l'environnement 1, la figure 6.5a représente les couloirs d'incertitude obtenus. La figure 6.5 représente le volume de la boite de localisation.

La première constatation est la consistance des résultats. En effet, contrairement à l'EKF-SLAM et au FastSLAM, les couloirs d'incertitude incluent la valeur zéro quelque soit le nombre d'amers présents dans l'environnement. De plus, on remarque l'effet de la fermeture de boucle sur la précision de la localisation. Le volume de la boite de localisation diminue fortement lors des fermetures de boucles successives.

Enfin, on remarque que l'algorithme ne tire pas entièrement profit des fermetures de boucles. En effet, la localisation lors du second, troisième et quatrième tours est légèrement moins précise que lors du premier tour.



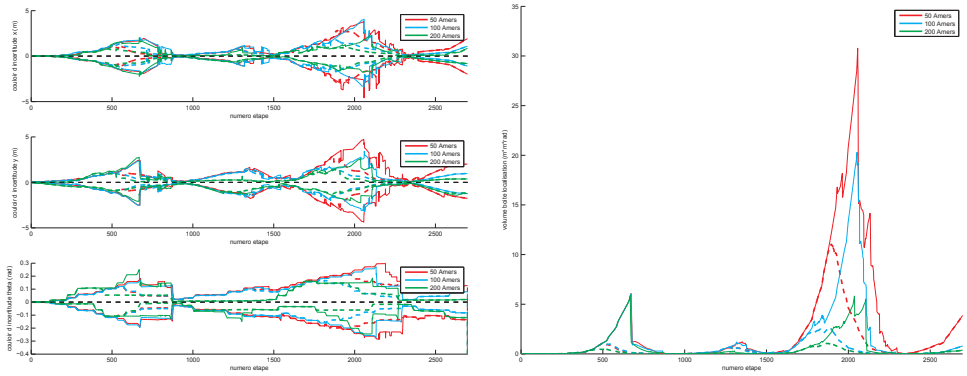
(a) *Couloirs d'incertitudes incluant la post localisation pour l'environnement 1*

(b) *Volume de la boite de localisation et post localisation pour l'environnement 1*

Figure 6.5: *Résultats expérimentaux du SLAM ensembliste pour l'environnement 1*

6.6.2 Environnement 3

Pour l'environnement 3, il s'agit d'un grand environnement composé de nombreux murs. La figure 6.6a représente les couloirs d'incertitudes. Comme pour l'environnement 1, on remarque que les résultats sont consistants et que l'algorithme parvient à ce relocaliser. Cependant, on remarque aussi le manque de précision de l'algorithme. Les couloirs d'incertitudes deviennent relativement large, en particulier autour de l'étape 1800. Ce phénomène est visible quelque soit le nombre d'amers présents. Enfin, ce manque de précision est confirmé par l'étude du volume (Figure 6.6b).



(a) *Couloirs d'incertitudes incluant la post localisation pour l'environnement 3*

(b) *Volume de la boite de localisation et post localisation pour l'environnement 3*

Figure 6.6: *Résultats expérimentaux du SLAM ensembliste pour l'environnement 3*

Bibliographie

[1] A.J. Davison : Real-time simultaneous localisation and mapping with a single camera. *In IEEE International Conference on Computer Vision*, pages 1403–1410, 2003.

[2] J.M.M. Montiel, J. Civera et A.J. Davison : Unified inverse depth parametrization for monocular SLAM. *In International Conference on Robotics : Science and Systems*, pages 16–19, 2006.

[3] M. Montemerlo, S. Thrun, D. Koller et B. Wegbreit : FastSLAM : A factored solution to the simultaneous localization and mapping problem. *In National Conference on Artificial Intelligence*, pages 593–598, 2002.

[4] L. Jaulin : A nonlinear set-membership approach for the localization and map building of an underwater robot using interval constraint propagation. *IEEE Transaction on Robotics*, 25:88–98, 2009.

[5] C.M. Gifford, R. Webb, J. Bley, D. Leung, M. Calnon, J. Makarewicz, B. Banz et A. Agah : Low-cost multi-robot exploration and mapping. *In IEEE International Conference on Technologies for Practical Robot Applications*, pages 74–79, 2008.

[6] S. Magnenat, V. Longchamp, M. Bonani, P. Rétornaz, P. Germano, H. Bleuler et F. Mondada : Affordable slam through the co-design of hardware and methodology. *In IEEE International Conference on Robotics and Automation*, pages 5395–5401, 2010.

[7] L. Meier, P. Tanskanen, L. Heng, G.H. Lee, F. Fraundorfer et M. Pollefeys : Pixhawk : A micro aerial vehicle design for autonomous flight using onboard computer vision. *Autonomous Robots*, pages 1–19, 2012.

[8] D. Botero, A. Gonzalez, M. Devy *et al.* : Architecture embarquée pour le slam monoculaire. *In Reconnaissance des Formes et Intelligence Artificielle*, 2012.

[9] R.E. Kalman : A new approach to linear filtering and prediction problems. *Basic Engineering*, 82:34–45, 1960.

[10] P.S. Maybeck : *Stochastic models, estimation and control*, volume 1. Academic Pr, 1979.

[11] R.S.M.S.P. Cheeseman, R. Smith et M. Self : A stochastic map for uncertain spatial relationships. *In International Symposium on Robotics Research*, 1987.

[12] P. Moutarlier et R. Chatila : Stochastic multisensory data fusion for mobile robot location and environmental modelling. *In International Symposium of Robotics Research*, 1990.

[13] JA Castellanos, JD Tardós et G. Schmidt : Building a global map of the environment of a mobile robot : The importance of correlations. *In IEEE International Conference on Robotics and Automation*, volume 2, pages 1053–1059, 1997.

[14] M. Csorba : *Simultaneous localisation and map building*. Thèse de doctorat, University of Oxford, 1998.

[15] M.W.M.G. Dissanayake, P. Newman, S. Clark, H.F. Durrant-Whyte et M. Csorba : A solution to the simultaneous localization and map building (slam) problem. *IEEE Transactions on Robotics and Automation*, 17(3):229–241, 2001.

[16] S. Betge-Brezetz, P. Hebert, R. Chatila et M. Devy : Uncertain map making in natural environments. *In IEEE International Conference on Robotics and Automation*, volume 2, pages 1048–1053, 1996.

[17] CG Harris et JM Pike : 3d positional integration from image sequences. *Image and Vision Computing*, 6(2):87–90, 1988.

[18] J. Neira, M.A.I. Ribeiro, J.D. Tardós *et al.* : Mobile robot localization and map building using monocular vision. *In In The 5th Symposium for Intelligent Robotics Systems*. Citeseer, 1997.

[19] S. Se, D. Lowe et J. Little : Local and global localization for mobile robots using visual landmarks. *In IEEE/RSJ International Conference on Intelligent Robots and Systems*, volume 1, pages 414–420, 2001.

[20] D.G. Lowe : Object recognition from local scale-invariant features. *In IEEE International Conference on Computer Vision*, volume 2, pages 1150–1157, 1999.

[21] A. Davison et D. Murray : Mobile robot localisation using active vision. *Computer Vision*, pages 809–825, 1998.

[22] I.K. Jung et S. Lacroix : High resolution terrain mapping using low attitude aerial stereo imagery. *In IEEE International Conference on Computer Vision*, pages 946–951, 2003.

[23] S. Thrun, D. Koller, Z. Ghahramani, H. Durrant-Whyte et Ng. A.Y. : Simultaneous mapping and localization with sparse extended information filters. *In International Workshop on Algorithmic Foundations of Robotics*, 2002.

[24] S. Thrun, Y. Liu, D. Koller, A.Y. Ng, Z. Ghahramani et H. Durrant-Whyte : Simultaneous localization and mapping with sparse extended information filters. *International Journal of Robotics Research*, 2004.

[25] R. Eustice, M. Walter et J. Leonard : parse extended information filters : Insights into sparsification. *In IEEE/RSJ International Conference on Intelligent Robots and Systems*, volume 1, page 3281–3288, 2005.

[26] M. Walter, R. Eustice et J. Leonard : A provably consistent method for imposing sparsity in feature-based slam information filters. *Robotics Research*, pages 214–234, 2007.

[27] P. Newman : *On the structure and solution of the simultaneous localisation and map building problem.* Thèse de doctorat, University of Sydney, 1999.

[28] S.J. Julier et J.K. Uhlmann : A counter example to the theory of simultaneous localization and map building. *In IEEE International Conference on Robotics and Automation*, volume 4, pages 4238–4243, 2001.

[29] S.J. Julier et J.K. Uhlmann : A new extension of the kalman filter to nonlinear systems. *In Int. Symp. Aerospace/Defense Sensing, Simul. and Controls*, volume 3, page 26, 1997.

[30] J.A. Castellanos, J. Neira et J.D. Tardós : Limits of to the consistency of the ekf-based SLAM. *In IFAC Symposium on Intelligent Autonomous Vehicles*, 2004.

[31] T. Bailey, J. Nieto, J. Guivant, M. Stevens et E. Nebot : Consistency of the EKF-SLAM algorithm. *In IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3562–3568, 2006.

[32] S. Huang et G. Dissanayake : Convergence analysis for extended kalman filter based slam. *In IEEE International Conference on Robotics and Automation*, pages 412–417, 2006.

[33] G.P. Huang, A.I. Mourikis et S.I. Roumeliotis : Analysis and improvement of the consistency of extended kalman filter based slam. *In IEEE International Conference on Robotics and Automation*, pages 473–479, 2008.

[34] J. Sola : Consistency of the monocular ekf-slam algorithm for three different landmark parametrizations. *In IEEE International Conference on Robotics and Automation*, pages 3513–3518, 2010.

[35] J. Solà, T. Vidal-Calleja, J. Civera et J.M.M. Montiel : Impact of landmark parametrization on monocular ekf-slam with points and lines. *International Journal of Computer Vision*, pages 1–30, 2010.

[36] K.S. Chong et L. Kleeman : Feature-based mapping in real, large scale environments using an ultrasonic array. *The International Journal of Robotics Research*, 18(1):3–19, 1999.

[37] J.J. Leonard et H.J.S. Feder : Decoupled stochastic mapping [for mobile robot & auv navigation]. *Oceanic Engineering, IEEE Journal of*, 26(4):561–571, 2001.

[38] J. Leonard et P. Newman : Consistent, convergent, and constant-time slam. *In International Joint Conference on Artificial Intelligence*, volume 18, pages 1143–1150, 2003.

[39] T. Bailey : *Mobile robot localisation and mapping in extensive outdoor environments*. Thèse de doctorat, University of Sydney, 2002.

[40] M. Bosse, P. Newman, J. Leonard et S. Teller : Simultaneous localization and map building in large-scale cyclic environments using the atlas framework. *The International Journal of Robotics Research*, 23(12):1113–1139, 2004.

[41] C. Estrada, J. Neira et J.D. Tardós : Hierarchical slam : Real-time accurate mapping of large environments. *IEEE Transactions on Robotics*, 21(4):588–596, 2005.

[42] E. Eade : *Monocular simultaneous localisation and mapping*. Thèse de doctorat, PhD thesis, University of Cambridge, 2008.

[43] Margarita Chli et Andrew J. Davison : Active Matching. *In* David Forsyth, Philip Torr et Andrew Zisserman, éditeurs : *European Conference on Computer Vision*, volume 5302, pages 72–85, 2008.

[44] J. Civera, O.G. Grasa, A.J. Davison et JMM Montiel : 1-point ransac for ekf-based structure from motion. *In IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3498–3504, 2009.

[45] T. Bailey, J. Nieto et E. Nebot : Consistency of the FastSLAM algorithm. *In IEEE International Conference on Robotics and Automation*, pages 424–429, 2006.

[46] S. Thrun et M. Montemerlo : The GraphSLAM algorithm with applications to large-scale mapping of urban structures. *International Journal on Robotics Research*, 25(5/6):403–430, 2005.

[47] J. Folkesson et H. I. Christensen : Graphical SLAM - a self-correcting map. *In IEEE International Conference on Robotics and Automation*, pages 383–390, 2004.

[48] Georg Klein et David Murray : Parallel tracking and mapping for small AR workspaces. *In International Symposium on Mixed and Augmented Realit*, Nara, Japan, 2007.

[49] J.S. Gutmann et K. Konolige : Incremental mapping of large cyclic environments. *In IEEE International Symposium on Computational Intelligence in Robotics and Automation*, pages 318–325. IEEE, 1999.

[50] R.A. Newcombe et A.J. Davison : Live dense reconstruction with a single moving camera. *In IEEE Conference on Computer Vision and Pattern Recognition*, pages 1498–1505. IEEE, 2010.

[51] H. Strasdat, JMM Montiel et A.J. Davison : Real-time monocular slam : Why filter ? *In IEEE International Conference on Robotics and Automation*, pages 2657–2664, 2010.

[52] A. Lambert, D. Gruyer, B. Vincke et E. Seignez : Experimental vehicle localization by bounded-error state estimation using interval analysis. *In IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1211–1216, 2009.

[53] B. Vincke et A. Lambert : Experimental comparison of bounded-error state estimation and constraints propagation. *In IEEE International Conference on Robotics and Automation*, pages 4724–4729, 2011.

[54] E. Seignez, M. Kieffer, A. Lambert, E. Walter et T. Maurin : Real-time bounded-error state estimation for vehicle tracking. *IEEE International Journal of Robotics Research*, 28:34–48, 2009.

[55] M. Di Marco, A. Garulli, S. Lacroix et A. Vicino : Set membership localization and mapping for autonomous navigation. *International Journal of robust and nonlinear control*, 11(7):709–734, 2001.

[56] M. Di Marco, A. Garulli, A. Giannitrapani et A. Vicino : A set theoretic approach to dynamic robot localization and mapping. *Autonomous robots*, 16 (1):23–47, 2004.

[57] C. Drocourt, L. Delahoche, E. Brassart, B. Marhic et A. Clerentin : Incremental construction of the robot's environmental map using interval analysis. *In Global optimization and constraint satisfaction : second international workshop*, pages 127–141, 2003.

[58] L. Jaulin, M. Kieffer, O. Didrit et E. Walter : *Applied Interval Analysis*. Springer-Verlag, 2001.

[59] J.M. Porta : CuikSlam : A kinematics-based approach to SLAM. *In International Conference on Robotics and Automation*, pages 2425–2431, 2005.

[60] L. Jaulin : Localization of an underwater robot using interval constraints propagation. *In International Conference on Principles and Practice of Constraint Programming*, 2006.

[61] Cyril Joly et Patrick Rives : Bearing-only SLAM : comparison between probabilistic and deterministic methods. Research Report RR-6602, INRIA, 2008.

[62] F. Le Bars, J. Sliwka, O. Reynet et L. Jaulin : Set-membership state estimation with fleeting data. *Automatica*, 48:381–387, 2012.

[63] J. Sliwka, L. Jaulin, M. Ceberio et V. Kreinovich : Processing interval sensor data in the presence of outliers, with potential applications to localizing underwater robots. *In IEEE International Conference on Systems, Man, and Cybernetics*, pages 2330–2337, 2011.

[64] Philippe Hoppenot, Etienne Colle et Christian Barat : Off-line localisation of a mobile robot using ultrasonic measurements. *Robotica*, 18(3):315–323, mai 2000.

[65] A.J. Davison, I.D. Reid, N.D. Molton et O. Stasse : MonoSLAM : Real-time single camera SLAM. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 1052–1067, 2007.

[66] F. Abrate, B. Bona et M. Indri : Experimental ekf-based slam for mini-rovers with ir sensors only. *In European Conference on Mobile Robots*, 2007.

[67] T.N. Yap et C.R. Shelton : Slam in large indoor environments with low-cost, noisy, and sparse sonars. *In IEEE International Conference on Robotics and Automation*, pages 1395–1401, 2009.

[68] C. Schröter, H.J. Böhme et H.M. Gross : Memory-efficient gridmaps in rao-blackwellized particle filters for slam using sonar range sensors. *In European Conference on Mobile Robots*, pages 138–143, 2007.

[69] K. Konolige, J. Augenbraun, N. Donaldson, C. Fiebig et P. Shah : A low-cost laser distance sensor. *In IEEE International Conference on Robotics and Automation*, pages 3002–3008, 2008.

[70] MYI Idris, H. Arof, EM Tamil, NM Noor et Z. Razak : Review of feature detection techniques for simultaneous localization and mapping and system on chip approach. *Information Technology Journal*, 8(3):250–262, 2009.

[71] M. Idris, N.M. Noor, E.M. Tamil, Z. Razak et H. Arof : Parallel matrix multiplication design for monocular SLAM. *In International Conference on Mathematical/Analytical Modelling and Computer Simulation*, pages 492–497, 2010.

[72] V. Bonato, E. Marques et G.A. Constantinides : A floating-point extended kalman filter implementation for autonomous mobile robots. *Journal of Signal Processing Systems*, 56(1):41–50, 2009.

[73] S. Bouaziz, M. Fan, A. Lambert, T. Maurin et R. Reynaud : Picar : experimental platform for road tracking applications. *In IEEE Intelligent Vehicles Symposium*, pages 495–499, 2003.

[74] E. Seignez, A. Lambert et T. Maurin : An experimental platform for testing localization algorithms. *In IEEE International Conference On Information And Communication Technologies : From Theory To Application*, pages 748–753, 2006.

[75] S. Ceriani, G. Fontana, A. Giusti, D. Marzorati, M. Matteucci, D. Migliore, D. Rizzi, D.G. Sorrenti et P. Taddei : Rawseeds ground truth collection systems for indoor self-localization and mapping. *Autonomous robots*, 27(4):353–371, 2009.

[76] B. Steux, L. Bouraoui, S. Thorel, L. Benazet *et al.* : Corebot m : Le robot de la team corebots préparé pour l'édition 2011 du défi carotte. 2011.

[77] J.C. Baillie, A. Demaille, G. Duceux, D. Filliat, Q. Hocquet, M. Nottale *et al.* : Software architecture for an exploration robot based on urbi. *In National Conference on Control Architectures of Robots*, 2011.

[78] J.L. Crowley : World modeling and position estimation for a mobile robot using ultrasonic ranging. *In IEEE International Conference on Robotics and Automation*, pages 674–680, 1989.

[79] A. Lambert : *Planification de tâches sûres pour robot mobile par prise en compte des incertitudes et utilisation de cartes locales*. Thèse de doctorat, 1998.

[80] R Hartley et A Zisserman : Multiple view geometry in computer vision. 2004.

[81] H. Bay, A. Ess, T. Tuytelaars et L. Van Gool : Speeded-up robust features (SURF). *Computer Vision and Image Understanding*, 110:346–359, 2008.

[82] D.G. Lowe : Distinctive image features from scale-invariant keypoints. *Computer Vision*, 60(2):91–110, 2004.

[83] C. Harris et M. Stephens : A combined corner and edge detection. *In The Fourth Alvey Vision Conference*, pages 147–151, 1988.

[84] J. Shi et C. Tomasi : Good features to track. *In IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 593–600, 1994.

[85] E. Rosten, R. Porter et T. Drummond : Faster and better : A machine learning approach to corner detection. *IEEE Transaction on Pattern Analysis and Machine Intelligence*, pages 105–119, 2009.

[86] A. Schmidt, M. Kraft et A. Kasiński : An evaluation of image feature detectors and descriptors for robot navigation. *Computer Vision and Graphics*, pages 251–259, 2010.

[87] M. Kraft, A. Schmidt et A. Kasinski : High-speed image feature detection using fpga implementation of fast algorithm. *In International Conference on Computer Vision Theory and Applications*, pages 174–179, 2008.

[88] S. Huang et G. Dissanayake : Convergence and consistency analysis for extended kalman filter based slam. *IEEE Transactions on Robotics*, 23(5):1036–1049, 2007.

[89] B. Lu, X. Wu, H. Figueroa et A. Monti : A low-cost real-time hardware-in-the-loop testing approach of power electronics controls. *IEEE Transactions on Industrial Electronics*, 54(2):919–931, 2007.

[90] T. Hwang, J. Roh, K. Park, J. Hwang, K.H. Lee, K. Lee, S.J. Lee et Y.J. Kim : Development of hils systems for active brake control systems. *In IEEE International Joint Conference SICE-ICASE, 2006*, pages 4404–4408, 2006.

[91] R.C. Smith et P. Cheeseman : On the representation and estimation of spatial uncertainty. *International journal of Robotics Research*, 5(4):56, 1986.

[92] H.F. Durrant-Whyte : Uncertain geometry in robotics. *Journal of Robotics and Automation*, 4(1):23–31.

[93] F. Chenavier et J.L. Crowley : Position estimation for a mobile robot using vision and odometry. *In IEEE International Conference on Robotics and Automation*, pages 2588–2593, 1992.

[94] B. Vincke, A. Elouardi et A. Lambert : Design and evaluation of an embedded system based SLAM applications. *In IEEE/SICE International Symposium on System Integration*, pages 224–229, 2010.

[95] A. Handa, M. Chli, H. Strasdat et A.J. Davison : Scalable active matching. *In IEEE Conference on Computer Vision and Pattern Recognition*, pages 1546–1553, 2010.

[96] J. Sola, A. Monin, M. Devy et T. Lemaire : Undelayed initialization in bearing only SLAM. *In IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2499–2504, 2005.

[97] R. Munguia et A. Grau : Delayed feature initialization for inverse depth monocular SLAM. *In European Conference on Mobile Robots*, pages 1–6, 2007.

[98] F.A. Auat Cheein et R. Carelli : Analysis of different feature selection criteria based on a covariance convergence perspective for a slam algorithm. *Sensors*, 11 (1):62–89, 2010.

[99] Eigen, 2012 : URL `http://eigen.tuxfamily.org/`.

[100] K. Goto et R.A. Geijn : Anatomy of high-performance matrix multiplication. *ACM Transactions on Mathematical Software*, 34(3):12, 2008.

[101] N. Sunderhauf, S. Lange et P. Protzel : Using the unscented kalman filter in mono-slam with inverse depth parametrization for autonomous airship control. *In IEEE International Workshop on Safety, Security and Rescue Robotics*, pages 1–6, 2007.

[102] P. Piniés, T. Lupton, S. Sukkarieh et J.D. Tardós : Inertial aiding of inverse depth slam using a monocular camera. *In IEEE International Conference on Robotics and Automation*, pages 2797–2802, 2007.

[103] A. Lindoso et L. Entrena : High performance fpga-based image correlation. *Journal of Real-Time Image Processing*, 2(4):223–233, 2007.

[104] S. Kyo et S. Okazaki : Imapcar : A 100 gops in-vehicle vision processor based on 128 ring connected four-way vliw processing elements. *Journal of Signal Processing Systems*, 62(1):5–16, 2011.

[105] Robert Sim, 2010 : URL `http://www.cs.ubc.ca/ simra/lci/-fastslam/nonlinear.html`.

[106] W.J. Dally, F. Labonte, A. Das, P. Hanrahan, J.H. Ahn, J. Gummaraju, M. Erez, N. Jayasena, I. Buck, T.J. Knight *et al.* : Merrimac : Supercomputing with streams. *In IEEE conference on Supercomputing*, page 35, 2003.

[107] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston et P. Hanrahan : Brook for gpus : stream computing on graphics hardware. *In ACM Transactions on Graphics*, volume 23, pages 777–786, 2004.

[108] D. Castaño-Díez, D. Moser, A. Schoenegger, S. Pruggnaller et A.S. Frangakis : Performance evaluation of image processing algorithms on the gpu. *Journal of structural biology*, 164(1):153–160, 2008.

[109] M. Montemerlo, S. Thrun, D. Koller et B. Wegbreit : FastSLAM 2.0 : An improved particle filtering algorithm for simultaneous localization and mapping that provably converges. *In International Joint Conference on Artificial Intelligence*, pages 1151–1156, 2003.

[110] L. Jaulin, F. Dabe, A. Bertholom et M. Legris : A set approach to the simultaneous localization and map building - application to underwater robots. *In International Conference on Informatics in Control, Automation and Robotics*, pages 65–69, 2007.

[111] R.E. Moore : *Methods and Applications of Interval Analysis*. Studies in Applied Mathematics, 1979.

[112] D. Waltz : *Understanding line drawings of scenes with shadows*. 1975.

[113] A.K. Mackworth : Consistency in networks of relations. *Artificial intelligence*, 8(1):99–118, 1977.

[114] J. Cleary : Logical arithmetic. *Future Computing Systems*, 2(2):125–149, 1987.

[115] A. Gning et P. Bonnifait : Constraints propagation techniques on real intervals for guaranteed fusion of redundant data. application to the localization of car-like vehicles. *Automatica*, pages 1167–1175, 2006.

[116] J. Sliwka, F. Le Bars, O. Reynet, L. Jaulin *et al.* : Using interval methods in the context of robust localization of underwater robots. *In Conference of the North American Fuzzy Information Processing Society*, pages 18–20, 2011.