Architecture embarquée pour le SLAM monoculaire

Diego Botero-Galeano, Aurélien Gonzalez, Michel Devy

Architecture embarquée pour le SLAM monoculaire

Diego Botero[1,2] Aurélien Gonzalez[1,2] Michel Devy[1,2]
[1] CNRS ; LAAS ; 7 avenue du colonel Roche, F-31077 Toulouse Cedex 4, France
[2]Université de Toulouse ; UPS, INSA, INP, ISAE ; UT1, UTM, LAAS ;
F-31077 Toulouse Cedex 4, France
{prenom.nom}@laas.fr

Résumé

Cet article rappelle les principes des algorithmes de SLAM monoculaire mis en oeuvre sur un véhicule pour traiter de sa localisation, puis décrit l'architecture développée pour embarquer ces algorithmes. Notre méthode SLAM exploite le filtre de Kalman étendu pour construire une carte d'amers ponctuels, représentés soit en paramètres IDP si la profondeur est trop imprécise, soit en euclidien ; la position du véhicule est mise à jour à partir d'appariements entre points extraits de l'image et amers de la carte. Cet article présente une implémentation sur une architecture dédiée, réalisée en co-design sur un FPGA. Les algorithmes d'extraction des points de l'image sont exécutés à la fréquence pixel sur FPGA, tandis que la fonction de construction et mise à jour de la carte reste exécutée en logiciel. Les caractéristiques et les performances de cette implémentation sont comparées avec celles obtenues par une version purement logicielle.

Mots Clef

SLAM, amers ponctuels, IDP, FPGA, architecture

Abstract

This paper recalls first bearing-only SLAM algorithms executed on a vehicle to provide its localization, and then, describes the embedded architecture designed for this method. It is an EKF-based SLAM approach, used to build a map of punctual landmarks represented either by IDP parameters when the depth is poorly estimated, or by euclidean coordinates. The vehicle position is updated from matchings between interest points extracted from images and landmarks of the map. This paper presents an embedded architecture developed by co-design on a FPGA. Algorithms required to extract points from images are executed on a FPGA, synchronized on the pixel clock, while functions used to build and update the map are performed on a CPU. Characteristics and performances of this implementation are compared with the ones provided by a purely software version.

Keywords

SLAM, point landmarks, IDP, FPGA, architecture

1 Introduction

Poussées par une forte demande des roboticiens, les méthodes de Localisation et Cartographie Simultanées (SLAM) se sont grandement développées depuis plus de 20 ans. Les progrès faits en Vision d'une part, puis les travaux de Davison en 2003 [3] sur une implémentation temps réel du SLAM visuel monoculaire, ont grandement influencé la communauté robotique. Les recherches sur le *SLAM visuel* ont atteint une certaine maturité en 2005 avec l'arrivée de techniques d'initialisations non retardées, traitées en 2006 avec la représentation d'amers ponctuels de type *Inverse Depth Parametrisation* (IDP) [12].

Plusieurs méthodes d'estimation, Filtre de Kalman Etendu (EKF), Filtre d'Information ou Filtre Particulaire [19] ont été étudiées pour traiter du SLAM en robotique. Pour garantir des performances temps réel, la méthode EKF-SLAM à base du filtre de Kalman étendu reste la référence dans le domaine : l'approche EKF-SLAM en vision monoculaire tourne, depuis 2007, avec des caméras visibles à 30 Hz [4]. Mais pour pallier le problème majeur du filtre de Kalman qui est sa divergence lors de longues séquences, la communauté SLAM s'est tournée peu à peu vers des méthodes d'optimisation. Ainsi, les méthodes fondées sur le *Sparse Bundle Adjustement* (SBA) [11] ou le Graph-SLAM [5] sont connues pour être plus robustes que EKF-SLAM. Mais à l'exception de la version *Parallel Tracking And Mapping* (PTAM) embarquée avec succès sur Iphone [10], ces méthodes ne sont pas compatibles avec les applications temps-réel, les résultats étant souvent obtenus hors-ligne et après post-traitement. L'autre inconvénient de ces approches est la difficulté d'intégrer des sources d'informations hétérogènes (une ou plusieurs caméras, centrale inertielle, GPS...). Avantages et inconvénients des méthodes d'estimation ou d'optimisation pour traiter du SLAM ont été discutées dans [18] où Strasdat et al. concluent que les méthodes de filtrage peuvent être avantageuses dans les cas de faibles ressources allouées.

Cependant, la communauté SLAM n'a pas trop abordé la problématique des systèmes embarqués et de ses applications potentielles. Nous pouvons noter quelques

récents travaux sur le design d'architectures FPGA pour la multiplication matricielle dans le cadre d'un SLAM monoculaire [9], ainsi que l'implémentation d'un EKF sur FPGA [1][21]. Notre projet concerne l'estimation des déplacements d'un véhicule évoluant dans un environnement structuré, puis la localisation du dit véhicule par rapport à une carte a priori de cet environnement. Dans cet article, nous présentons les méthodes développées pour construire une carte stochastique permettant d'estimer la position du véhicule et d'un ensemble de primitives visuelles, ici des points. Nous avons en parallèle développé des algorithmes de traitement d'images et d'extraction de primitives visuelles exécutés à la fréquence pixel sur FPGA [6], et prototypé les algorithmes de SLAM inertiel monoculaire visible et infrarouge pour pouvoir à terme les embarquer avec des contraintes fortes de ressources (mémoire, puissance de calcul...) [8]. Dans l'architecture finale décrite dans cet article, il a fallu faire collaborer ces deux composantes matérielle et logicielle en les intégrant sur un FPGA associé à un processeur.

La section suivante permet de positionner nos travaux par rapport à la communauté SLAM. Dans la section 3 nous présenterons les différentes approches logicielles développées, tandis que le co-design et l'implémentation matérielle seront traités dans la section 4. Enfin, la section 5 résume les contributions du papier et évoque nos travaux en cours.

2 Localisation par SLAM monoculaire

Le but n'est pas dans ce papier de présenter toutes les fonctions à mettre en oeuvre pour traiter du SLAM monoculaire avec amers ponctuels. Nous rappelons simplement les principaux problèmes à considérer.

2.1 Paramétrisation des points

D'abord comment initialiser un amer ponctuel ? La manière dont sont paramétrés et reparamétrés les amers a une influence significative sur la consistence de l'algorithme. Une gestion efficace des amers améliore le filtrage, réduit les temps de calcul et prévient de la divergence du SLAM. De nombreuses méthodes pour la description de points et de lignes sont analysées dans [14] ; pour les points, trois types de paramétrisations ont été évaluées, Anchored Homogeneous Point (AHP) [14], Inverse-Depth Point parametrization (IDP) [12] et Euclidien.

Un amer AHP est représenté par un vecteur de 7 composantes, 3 pour les coordonnées euclidiennes du centre optique lors de l'initialisation du point, $\mathbf{p}_0 = (x_0, y_0, z_0)$, 3 pour le vecteur directeur de l'axe optique $\mathbf{v} = (u, v, w)$, et une dernière composante définie par la distance enclidienne inverse entre $\mathbf{p}_0$ et le point 3D $\mathbf{p}$, notée ρ

$$\mathbf{P}_{AHP} = \left[\mathbf{p}_0^T \ \mathbf{v}^T \ \rho\right]^T = [x_0 \ y_0 \ z_0 \ u \ v \ w \ \rho]^T \in \mathbb{R}^7$$

Un amer IDP est représenté par un vecteur de 6 composantes, 3 pour les coordonnées euclidiennes du centre optique lors de l'initialisation du point $\mathbf{p}_0$, 2 pour l'angle d'élévation et d'azimuth pour définir la direction du rayon optique initial (ϵ, α), et ρ la distance enclidienne inverse entre $\mathbf{p}_0$ et le point 3D $\mathbf{p}$:

$$\mathbf{P}_{IDP} = \left[\mathbf{p}_0^T \ \epsilon \ \alpha \ \rho\right]^T = [x_0 \ y_0 \ z_0 \ \epsilon \ \alpha \ \rho]^T \in \mathbb{R}^6$$

Un amer Euclidien est juste défini par ses 3 composantes cartésiennes dans le repère monde :

$$\mathbf{P}_E = P = [X \ Y \ Z]^T \in \mathbb{R}^3$$

2.2 Reparamétrisation des points

Le principal inconvénient des représentations non euclidiennes est qu'elles occupent 6 ou 7 variables dans la carte stochastique contre 3 seulement pour un amer euclidien. Or, l'occupation mémoire comme la complexité temporelle du filtre de Kalman varie de façon quadratique avec la taille de la carte, ce qui signifie qu'un point non euclidien est environ 3 à 4 fois plus encombrant et long à traiter qu'un point euclidien. Nous avons donc implémenté un mécanisme de *reparamétrisation* des amers IDP ou AHP en amers euclidiens lorsqu'ils ont suffisamment convergé. Nous utilisons pour cela le critère de linéarité proposé par [12] :

$$L = \frac{4 \cdot \sigma_d}{d} \left|\cos \alpha\right|$$

dépendant de l'incertitude sur la distance de l'amer σ_d par rapport à sa distance d au capteur, et du cosinus du changement de point de vue angulaire α que le capteur a eu depuis l'initialisation de l'amer, afin d'évaluer l'observabilité qu'il a eu de sa distance.

2.3 Carte probabiliste

L'un des points clé de notre approche est la gestion d'une carte probabiliste regroupant l'ensemble des amers perçus dans la scène. Dans notre application, nous ne faisons que de la localisation, en exploitant des estimées du mouvement du véhicule dans la phase de prédiction ; pour limiter la taille de la carte, nous considérons que le véhicule ne repasse pas sur un lieu déjà visité. De ce fait, les amers ne sont pas conservés sur le très long terme. On peut donc ne construire qu'une seule carte ; nous n'avons qu'un seul robot à localiser ; ce robot n'est équipé que d'une caméra, et nous n'exploitons que des amers ponctuels, observés dans les images par des points d'intérêt.
Cette carte stochastique peut être représentée comme un vecteur d'état composé du robot et des amers : $X = \left[\mathcal{R} \ \mathcal{M}\right]^T$.

$\mathcal{R}$ est le vecteur d'état du robot. Dans nos applications, différents types de modèles d'évolution ont été implémentés :
• le modèle de type *vitesse constante* défini par $\mathcal{R} = [p\ q\ v\ w]^\mathsf{T}$ où p et q sont la position et l'orientation du robot définie par un quaternion et v et w sont chacun des vecteurs de dimension 3 définissant les vitesses linéaires et angulaires,
• le modèle de type *odométrique*, uniquement défini par $\mathcal{R} = [p\ q]^\mathsf{T}$
• le modèle de type *inertiel* défini par $\mathcal{R} = [p\ q\ v\ a_b\ w_b\ g]^\mathsf{T}$ où v défini la vitesse linéaire, a_b et w_b représentent respectivement les biais des accéléromètres et des gyromètres de notre centrale inertielle, et g est la gravité suivant les 3 axes.
$\mathcal{M}$ est composé de l'ensemble des amers, chaque amer étant codé avec 3 ou 6 paramètres, dépendant de sa paramétrisation Euclidienne ou IDP : $\mathcal{M} = \left[\mathcal{L}_1 \dots \mathcal{L}_n\right]^T$.

Dans un filtre de Kalman étendu, les densités *a posteriori* sont approximées par des densités gaussiennes, dont les matrices de variance et covariance sont définies par :

$$\hat{X} = \begin{bmatrix} \hat{\mathcal{R}} \\ \hat{\mathcal{M}} \end{bmatrix} \quad \mathbf{P} = \begin{bmatrix} \mathbf{P}_{\mathcal{R}\mathcal{R}} & \mathbf{P}_{\mathcal{R}\mathcal{M}} \\ \mathbf{P}_{\mathcal{M}\mathcal{R}} & \mathbf{P}_{\mathcal{M}\mathcal{M}} \end{bmatrix} \tag{1}$$

où $\mathbf{P}$ est carrée symétrique.

2.4 Filtre de Kalman Etendu

Le filtre de Kalman étendu est une solution simple pour résoudre le problème d'estimation posé par le SLAM, et qui a été beaucoup utilisée en robotique mobile, même si elle est aujourd'hui concurrencée par des méthodes d'optimisation globale.
Dans un algorithme de type EKF-SLAM, le vecteur d'état est constitué de la concaténation de l'état du robot $\mathcal{R}$ et des états des différents amers $\mathcal{L}_i$:

$$\mathcal{X} = (\mathcal{R}\ \mathcal{L}_1\ \cdots\ \mathcal{L}_n)^T$$

Dans l'hypothèse d'un environnement statique, l'équation d'état ne concerne que le robot :

$$\mathcal{R}^+ = f\left(\mathcal{R}, \mathbf{u}, \mathbf{n}\right)$$

où $\mathbf{u}$ est la commande et $\mathbf{n}$ un bruit additif gaussien centré. L'équation est alors linéarisée pour pouvoir appliquer les équations du filtre de Kalman.
De même l'équation d'observation de l'amer i ne dépend que de l'état du robot et de l'amer :

$$\mathbf{y_i} = h\left(\mathcal{R}, \mathcal{L}_i, \mathbf{v}\right)$$

où $\mathbf{v}$ est un bruit additif gaussien centré, et qui est encore une fois linéarisé.
Le filtre de Kalman fournit alors les équations permettant d'estimer $\mathcal{X}$ et sa matrice de covariance à travers les étapes de prédiction et de correction.

À chaque fois qu'un capteur fournit une nouvelle donnée permettant l'observation d'amers, le filtre exécute l'étape de prédiction afin d'estimer l'état $\mathcal{X}$ à l'instant où la donnée a été acquise. L'étape de correction permet ensuite d'intégrer l'information apportée par l'observation des amers.

3 Implémentation logicielle

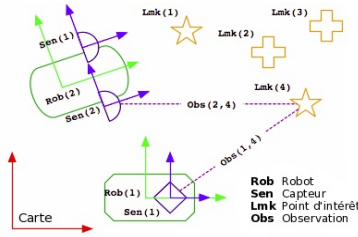
Limitées par le nombre de ressources disponibles sur une carte embarquée (mémoire, puissance de calcul...), les fonctions de traitement se doivent d'être les plus optimisés possible.

3.1 SLAM Toolbox

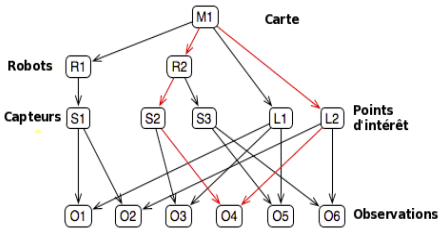
J.Sola a d'abord prototypé tous ces algorithmes dans une *SLAM Toolbox* développée sous MATLAB [15] qui permet de tester des méthodes d'EKF-SLAM pour un ou plusieurs robots qui naviguent dans un environnement, découvrant et cartographiant en même temps des primitives visuelles, à l'aide de leurs capteurs embarqués. Plusieurs robots de différents types peuvent être présents, portant chacun de multiples capteurs, qui eux-mêmes permetent d'observer des primitives de différents types (points, droites), ou décrites par différentes paramétrisations. Cette variété des objets à manipuler pour un tel problème comme illustré en figure 1(a), est gérée de manière transparente par la toolbox. Cette Toolbox nous a en particulier permis de vérifier la consistance de chaque fonction présente dans les autres implémentations.

3.2 RT-SLAM

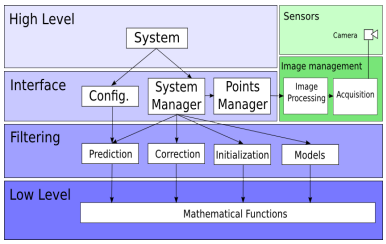
Récemment, une implémentation logicielle réalisée en C++ a été directement découlée de la SLAM Toolbox [13]. Cette version fait tourner un EKF-SLAM temps-réel, pour des données venant d'une caméra visible à 60Hz et d'une centrale inertielle à 100Hz via un couplage serré. Elle permet des extensions vers le multi-robot, les amers hétérogénes, le multi-capteurs... La figure 1(b) présente la structure de cette version logicielle, et est disponible à l'adresse : http ://www.openrobots.org/wiki/rtslam.
Cette version intègre les dernières avancées en termes d'EKF-SLAM visuel. Par exemple, nous pouvons noter l'utilisation de la paramétrisation des amers en *Anchored Homogeneous Point (AHP)* [14] et de leurs reparamétrisation en Euclidien lors de leurs convergence [12], la gestion des points inconsistants (*Outliers*) par la méthode de *One-Point Ransac* [2], l'*active search* [4] avec prédiction de l'apparence d'un point d'intérêt depuis le point de vue courant par déformation d'imagettes, ainsi que la prise en compte du SLAM-*bicam* qui exploite deux caméras [16]
Certaines optimisations ont été réalisées pour améliorer l'étape de traitement d'image, en utilisant par exemple des *images intégrales* [20] pour accélérer à la



(a) Illustration d'un SLAM multi-robot et multi-capteur pouvant être géré par la SLAM-Toolbox.



(b) Hierarchie des éléments dans l'implémentation RT-SLAM.



(c) Architecture de l'implémentation C-SLAM, divisée en 4 couches.

FIG. 1 – Présentation des architectures gérées par les différentes implémentations logicielles.

fois le détecteur de points de Harris et une corrélation basée sur un score ZNCC calculé de manière hiérarchique [17].

3.3 C-Slam

Puis, une implémentation logicielle bridée a été développée en C [7]. Cette nouvelle version du SLAM a deux buts. D'une part, elle doit être embarquée à bord d'un système embarqué doté de circuits FPGA équipé soit d'un processeur hard PowerPC (FPGA de la famille Virtex5) soit d'un processeur soft MicroBlaze (FPGA de la famille Virtex6). D'autre part, cette version doit respecter les normes strictes de codage en C (C-ANSI) ainsi que celles des systèmes embarqués dans l'aéronautique (normes DAL, pour Design Assurance Level), interdisant par exemple l'allocation dynamique de mémoire, les appels récursifs, ou l'utilisation de librairies externes.

Dans cette implémentation plus légère, nous nous concentrons sur la problématique de la localisation d'un véhicule embarquant une ou plusieurs caméras monoculaire visible et/ou infrarouge, un GPS et une centrale inertielle.

Cette version a deux atouts majeurs : sa modularité et son architecture hiérarchique présentée en figure 1(c). En effet, elle se doit d'être facilement manipulable, comme pouvoir l'interfacer avec des fonctions *OpenCv* afin de pouvoir visualiser son comportement ou traiter des jeux de données. Cette modularité vient directement de sa structure organisée en 4 couches comportant des fonctions de haut à plus bas niveau. Ainsi, l'interfaçage avec OpenCv se fait aisément via les fonctions de haut niveau que l'on peut désactiver facilement lors du portage sur carte embarquée. Les couches les plus basses correspondent aux fonctions mathématiques et pourront ensuite être portées sur les Éléments Logiques du FPGA dans le cadre d'optimisations.

Comme le nombre de ressources est limité, nous avons choisi une paramétrisation de type IDP des points lors de leurs initialisations, ce qui définit un amer sur 6 composantes au lieu de 7 pour une paramétrisation AHP.

D'autres algorithmes n'ont pas été portés, tel que le One-Point Ransac et la transformation des imagettes lors de la corrélation.

Cette version est en évolution permanente, et est disponible en *Open Source* sur le dépôt : http ://c-slam-laas.googlecode.com/svn/trunk/.

4 Implémentation du SLAM dans un système embarqué

4.1 Co-design pour le SLAM

Un FPGA permet d'implémenter des architectures dédiées pour un algorithme donné, en tenant compte de manière optimale des contraintes sévères de temps réel, compacité, coût... Ces architectures sont créées via des outils de développement (ISE pour *Xilinx*) et via un langage HDL (Hardware Description Language), VHDL, Verilog, System Verilog ou System C. Un programme HDL indique comment créer une fonction en reliant entre eux, des Éléments Logiques (LEs). Ces derniers permettent de réaliser des opérations combinatoires simples, mais ont l'importante capacité de pouvoir paralléliser plusieurs milliers d'opérations de ce type.

Sur les FPGAs de dernière génération, l'utilisateur peut porter une partie de son application sur une architecture matérielle dédiée, et une autre partie directement en logiciel développé en C ou C++, compilé pour un processeur hard associé au FPGA (PowerPC pour un Virtex 5) ou programmé sur les composants du FPGA (MicroBlaze pour *Xilinx*).

Le but du co-design va être de partager au mieux les tâches entre le processeur et les LEs de manière à obtenir la meilleure performance possible. Les opérateurs de traitement d'image étant parallélisables et pouvant travailler à la fréquence pixel sont développés sur une architecture dédiée ; les opérations plus calculatoire (filtrage EKF) sont embarquées de manière logicielle sur le processeur de la carte, éventuellement augmenté par des instructions spécifiques (par exemple, inversion de matrices)

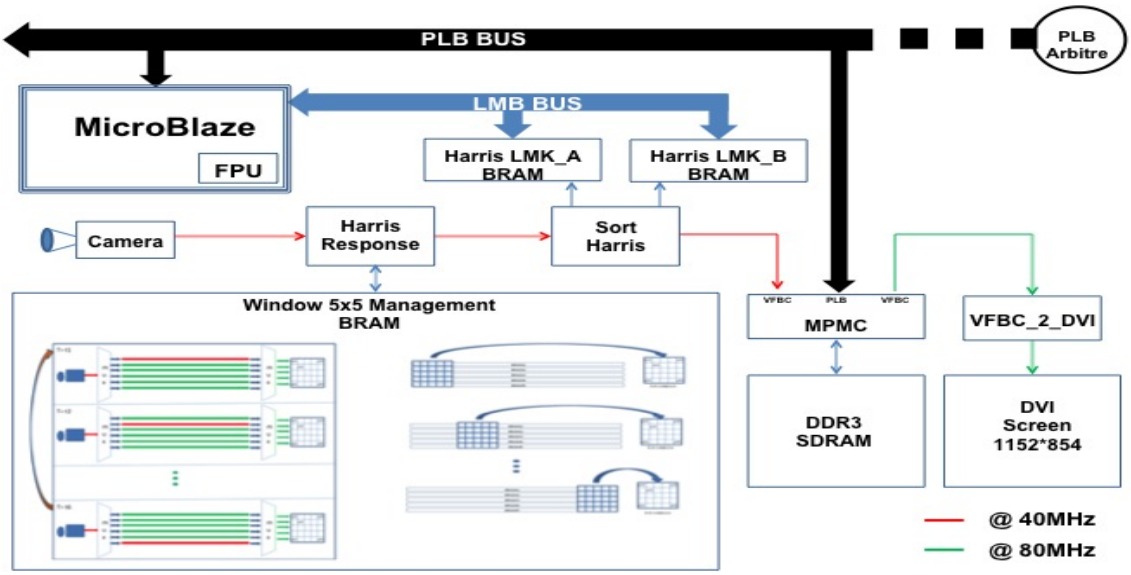


FIG. 2 – Architecture créée sur un FPGA Virtex6 pour l'implémentation du SLAM.

4.2 Interface matériel/logiciel

L'algorithme SLAM visuel décrit en section 2 a été porté sur un kit équipé d'un FPGA de la famille *Xilinx Virtex 6* via l'architecture présentée en figure 2. Chaque image, acquise par une caméra *Jai* interfacée en *Camera Link*, est traitée par un extracteur de points de Harris à la fréquence pixel. Ces points sont ensuite stockés sur une mémoire *BRAM*, accessibles pour une visualisation, et par un processeur MicroBlaze. C'est sur ce dernier qu'est portée des fonctions du code C-Slam décrit en section 3.3.

Nous avons également évaluée une implémentation du SLAM sur un Power-PC hard associé à un FPGA de la famille *Virtex 5*, processeur cadencé à 400MHz. Dans les deux architectures un bus PLB est utilisé afin de faire communiquer le processeur avec les différents périphériques du système, en particulier afin d'accéder à des mémoires Dual Port, accédées à la fois par le FPGA et par le processeur.

4.3 Architecture SLAM sur FPGA

Le principal intérêt des FPGAs est de pouvoir traiter des données à la volée. Ici, nous utilisons cette fonctionnalité pour extraire des points de Harris à la fréquence pixel lors de l'arrivée des images sur la mémoire interne. Le coeur du système est un processeur Microblaze. Ce processeur dispose d'une unité pour les opérations flottantes (FPU). Le processeur interagit avec le système via les bus PLB et LMB. Le bus PLB permet au processeur d'accéder aux différents périphériques et aux pixels de l'image, tandis que le bus LMB lui donne accès aux amers qui ont été calculés par le module «Sort Harris». En rouge est affiché le pipeline de traitement des images qui s'exécute à la fréquence pixel.

L'extraction des points de Harris nécessite un filtrage gaussien de l'image, puis le calcul de dérivées partielles en u et v. Ces opérations sont réalisées sur des fenêtres 5x5. De ce fait les pixels reçus de la caméra sont d'abord stockés dans un buffer circulaire pour mémoriser les 5 dernières lignes : le module «Harris réponse» peut ainsi lire en parallèle dans ce buffer, les 25 intensités des pixels d'une fenêtre 5x5 centrés sur le pixel (uv) traité, avec un retard de 6 lignes par rapport à l'acquisition de l'image.

Sur cette fenêtre 5x5, 13 opérations sont exécutées en pipeline à la fréquence pixel pour appliquer le filtrage, pour calculer les dérivées, et enfin pour calculer la réponse du détecteur pour un point de l'image. Des approximations permettent de fonder toute l'architecture sur des multiplieurs en virgule fixe, puis sur des décalages de registres ; un exemple est donné en figure 3 pour l'application d'un filtre gaussien. Une réponse de Harris est finalement générée pour chaque pixel (uv) .

Ensuite le module "Sort Harris" permet de répartir les points au maximum dans l'image. En effet, pour éviter les regroupements de points dans des zones particulières, un découpage de l'image (ou *tesselation*) est réalisé : uniquement le point ayant la meilleure réponse de Harris et supérieure à un seuil est sélectionné dans chaque zone. Ces meilleurs points ont une forte probabilité d'être retrouvés dans l'image suivante. Les coordonnées et le score de ces meilleurs points de Harris sont stockés dans deux mémoires internes type DUAL PORT RAM (Harris_LMK_A et Harris_LMK_B). Ces mémoires sont gérées comme des buffers ping pong : à un instant donné, une des mémoires contient les données calculées pour l'image précédente, tandis que l'autre est mise à jour par le traitement sur l'image courante. Finalement le module "Sort Harris"

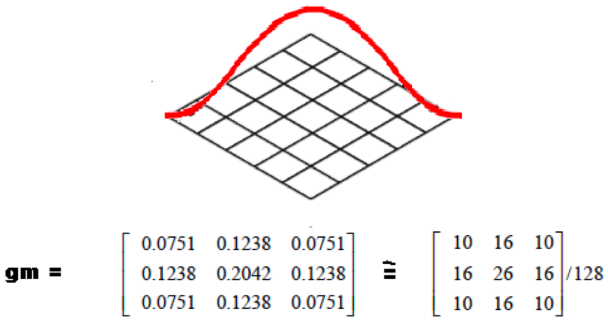


FIG. 3 – Application d'un filtrage gaussien sur FPGA.

écrit l'image sur une mémoire externe type DDR3 en indiquant chaque point extrait d'un croix. En Vert est affiché le pipeline de visualisation des images sur un écran au format DVI.

4.4 Stratégie de Recherche active

Parmi les optimisations principales de nos algorithmes, la stratégie de Recherche Active consiste à ne rechercher l'observation d'un amer que dans la zone où le filtre s'attend avec une probabilité non négligeable de le retrouver. Après l'étape de prédiction du filtre de Kalman, la fonction d'observation permet de prédire où l'on doit retrouver les amers ; la jacobienne sur la position de l'amer, permet de calculer une zone de recherche où il doit être observé. Cette zone de recherche prend la forme d'une ellipse en raison de l'hypothèse de bruit gaussien du filtre de Kalman, et est aussi appelée *ellipse d'incertitude* d'observation. Les amers sont donc projetés dans l'image, appariés, puis corrigés, voyant leur zone de recherche diminuer progressivement au fur et à mesure que l'incertitude de localisation du robot diminue.
Cette technique présente deux avantages. D'une part elle limite énormément les traitements à effectuer sur les données brutes du capteur en ne considérant que la sous-partie intéressante de ces données, nous permettant d'atteindre des fréquence de traitement très importantes (par exemple 60 Hz en logiciel pour des résolutions VGA). D'autre part elle permet de pouvoir choisir le nombre d'amers corrigés à chaque image, ce qui apporte un aspect *temps réel dur* à l'algorithme ; on choisit de corriger en priorité les amers qui ont les ellipses d'incertitude d'observation les plus grosses, car ce sont ceux dont l'observation apporte le plus d'information.
L'algorithme de recherche active sur l'implémentation matérielle diffère légèrement. En effet, nous n'avons pas la contrainte de temps sur la recherche des points d'intérêts car ceux-ci sont extraits à la fréquence pixel lors de l'arrivée de l'image. Dans cette implémentation-ci, comme les points de Harris sont déja calculés, le point le plus proche dans la zone d'incertitude de l'obervation est sélectionné.

5 Résultats

Nous allons maintenant comparer les différentes implémentations logicielles entre-elles, et évaluer leurs performances à la fois en terme de précision et de vitesse d'exécution. Nous examinerons ensuite les performances matérielles d'un SLAM embarqué.

5.1 Performances logicielles

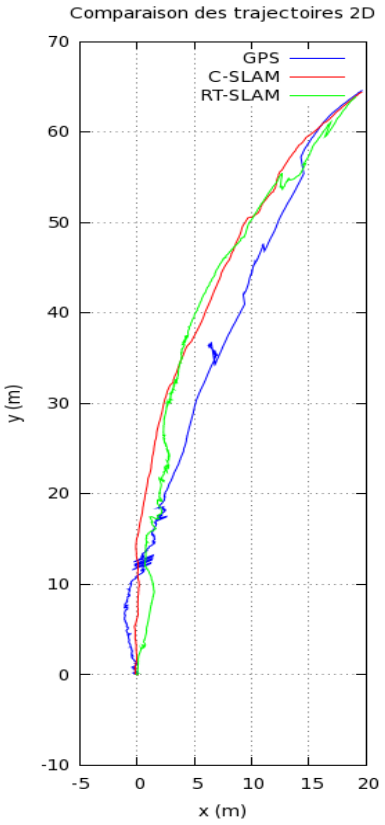


FIG. 4 – Comparaison des différentes trajectoires.

En terme de précision, la figure 4 montre les résultats du SLAM obtenus sur une séquence prise à bord d'un robot mobile. Dans ces résultats, le modèle de robot de type *Vitesse Constante* a été choisi. Bien que les 2 implémentations logicielles restent proche de la vérité terrain, on peut noter que ce type de robot ne nous permet pas d'observer la distance et l'orientation de départ, qui ont dû être recalés. Malgré cela, la version RT-SLAM est plus précise car elle implémente des algorithmes non présents dans la version C, tel que le One-Point Ransac et la transformation des imagette lors de la corrélation.
En terme de vitesse d'exécution, la version RT-SLAM a été développée dans le but de fonctionner avec des caméras à 60Hz sur des machines modernes. L'analyse de l'incidence du nombre de corrections sur la fréquence est présentée en figure 5. Ces résultats ont été obtenus sur une machine moderne équipé d'un *Core2*

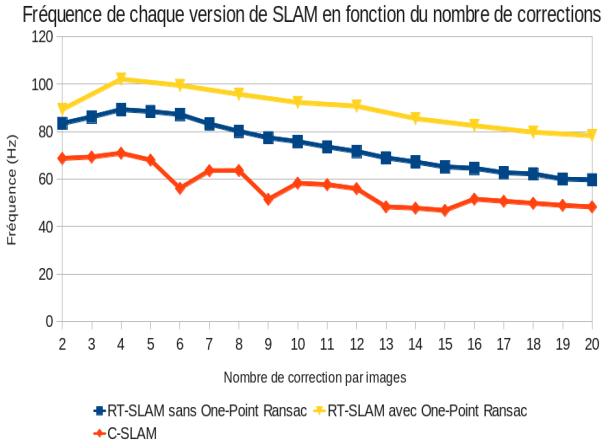


FIG. 5 – Comparaison des fréquences des différentes versions logicielles du SLAM.

Duo fréquencé à 2,8 GHz, où seulement un seul processeur était utilisé. A chaque pas de temps, 5 amers sont initialisés, jusqu'à la limite de 20. On modifie ensuite le nombre de corrections réalisées.
3 versions sont ici comparées : la version C, la version RT-SLAM avec correction des amers en utilisant uniquement l'Active Search, et la version RT-SLAM où une correction sur 2 se fait par la méthode *One-Point Ransac*.
Dans les 3 versions, on peut noter une augmentation de la fréquence lorque l'on passe de une à 4 corrections, expliqué par le fait que l'on réduit fortement les incertitudes, et donc que l'on réduit les zones sur lesquelles on réalise la recherche active. Ensuite, pour un même nombre d'amers dans la carte la fréquence a tendance à diminuer suivant un logarithme.
La version C-SLAM est moins rapide que la version RT-SLAM toujours pour la même raison : certains algorithmes n'ont pas été développés car ils seront exécutés sur FPGA.

5.2 Performances matérielles

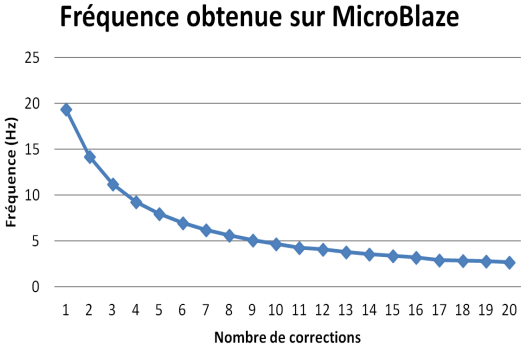


FIG. 6 – Fréquence obtenue avec l'utilisation d'un MicroBlaze.

Nous allons maintenant nous intéresser aux performances de la version C du SLAM embarquée sur deux architectures légèrement différentes.
La première consiste au portage du SLAM sur l'architecture décrite en figure 2. Il est important de noter qu'un MicroBlaze est fréquencé à 200 MHz. Cependant, plusieurs traitements sont effectués par les Éléments Logiques, tels que l'extraction des points de Harris, et la stratégie de correction décrite en 4.4. La figure 6 illustre la fréquence obtenue pour un nombre différent de corrections réalisées. Pour évaluer ces performances fréquentielles, nous avons simulé des points venant d'une caméra 640×480. A chaque test, ces points sont générés aléatoirement, et comme pour l'évaluation logicielle, 5 amers sont initialisés à chaque pas de temps jusqu'à la limite de 20.

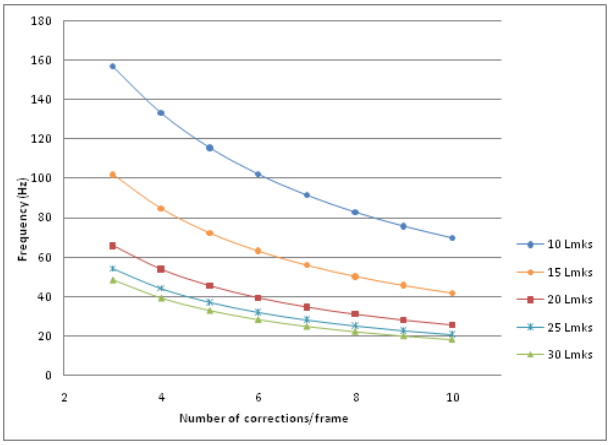


FIG. 7 – Pour un nombre différent d'amers IDP dans la carte, nous évaluons la fréquence obtenue en fonction du nombre de corrections pour chaque image.

Pour le projet final, nous utilisons en collaboration la carte *Virtex6* pour les traitements à la fréquence pixel, et une carte *Virtex5* munie d'un PowerPC fréquencé à 400MHz sur lequel est porté la même version C du SLAM. Pour évaluer les performances fréquentielles illustrées en figure 7, les mêmes conditions expérimentales que précédemment ont été appliquées. De plus nous analysons l'incidence du nombre de points dans la carte. En plus des optimisations restant à réaliser, nous pouvons noter qu'une approche temps-réel pour une application EKF-SLAM comportant un faible nombre d'amers est tout à fait possible, mais à des fréquences bien plus basses que celles des machines modernes.

6 Conclusions et perspectives

Nous avons décrit dans cet article une architecture conçue pour embarquer sur un véhicule, et pour exécuter en temps réel un algorithme de SLAM monoculaire. La méthode proposée est issue des travaux de J.Sola [14] ; une implémentation purement logicielle RT-SLAM a été décrite dans [13]. Nous avons montré

quelles modifications ont été nécessaires pour implémenter ce même algorithme sur FPGA, et comment les performances sont impactées.
Il est maintenant connu que les méthodes fondées sur des techniques d'estimation divergent dans le cas de prédictions trop imprécises sur les mouvements du véhicule, ou dans le cas de données sensorielles trop bruitées ; dans ces situations, les approches incrémentales de *Bundle Adjustment*, fondées sur des méthodes d'optimisation, arrivent à de meilleurs résultats [18]. L'implémentation réalisée sur un iPhone décrite dans [10] a des performances temps réel.
Néanmoins nous poursuivons nos travaux sur le développements d'un EKF-SLAM, en intégrant d'autres fonctions et plusieurs optimisations. Un module de détection d'obstacle par *Inverse Perspective Mapping* a déjà été réalisé sur FPGA [6]. Une extension de l'architecture sur FPGA pour le SLAM est également en cours avec l'implémentation des calculs en virgule fixe.

Références

[1] V. Bonato, E. Marques, and G. Constantinides. A floating-point extended kalman filter implementation for autonomous mobile robots. *Journal of Signal Processing Systems*, 56 :41–50, 2009.

[2] J. Civera, O.G. Grasa, A.J. Davison, and J.M.M. Montiel. 1-point ransac for ekf-based structure from motion. In *Proc. IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS)*, pages 3498 –3504, 2009.

[3] A.J. Davison. Real-Time Simultaneous Localisation and Mapping with a Single Camera. In *Proc. IEEE Int. Conf. on Computer Vision (ICCV)*, October 2003.

[4] A.J. Davison, I.D. Reid, N.D. Molton, and O. Stasse. Monoslam : Real-time single camera slam. *IEEE Trans. Pattern Anal. Mach. Intell.*, 29 :1052–1067, June 2007.

[5] J. Folkesson and H. Christensen. Graphical SLAM : a self-correcting map. In *Proc. IEEE Int. Conf. on Robotics and Automation (ICRA)*, 2004.

[6] D.A. Botero Galeano, M. Devy, J.L. Boizard, and W. Filali. Real-time architecture on fpga for obstacle detection using inverse perspective mapping. In *Proc. IEEE Int. Conf. on Electronics, Circuits and Systems (ICECS)*, December 2011.

[7] A. Gonzalez, J.M. Codol, and M. Devy. A c-embedded algorithm for real-time monocular slam. In *Proc. IEEE Int. Conf. on Electronics, Circuits and Systems (ICECS)*, December 2011.

[8] A. Gonzalez, M. Devy, and J.Solà. Slam visuel monoculaire par caméra infrarouge. In *Proc. Congrès des jeunes chercheurs en vision par ordinateur (ORASIS)*, June 2011.

[9] M. Idris, N.M. Noor, E.M. Tamil, Z. Razak, and H. Arof. Parallel matrix multiplication design for monocular slam. In *4th Asia Int. Conf. on Mathematical/Analytical Modelling and Computer Simulation (AMS)*, pages 492 –497, may 2010.

[10] G. Klein and D. Murray. Parallel tracking and mapping for small AR workspaces. In *Proc. 6th IEEE and ACM Int. Symp. on Mixed and Augmented Reality*. IEEE Computer Society, 2007.

[11] M.I.A. Lourakis and A.A. Argyros. The design and implementation of a generic sparse bundle adjustment software package based on the levenberg-marquardt algorithm. 2004.

[12] J. M. M. Montiel. Unified inverse depth parametrization for monocular slam. In *Proc. Robotics : Science and Systems (RSS)*, 2006.

[13] C. Roussillon, A. Gonzalez, J. Solà, J.M. Codol, N. Mansard, S. Lacroix, and M. Devy. Rt-slam : a generic and real-time visual slam implementation. In *Proc. 8th Int. Conf. on Computer Vision Systems (ICVS)*, September 2011.

[14] J. Solà. Consistency of the monocular ekf-slam algorithm for three different landmark parametrizations. In *Proc. IEEE. Int. Conf. Robotics and Automation (ICRA)*, May 2010.

[15] J. Solà, D. Marquez, JM Codol, and T. Vidal-Calleja. An EKF-SLAM toolbox for MATLAB. http ://homepages.laas.fr/jsola/JoanSola/eng/toolbox.html.

[16] J. Solà, A. Monin, and M. Devy. Bicamslam : Two times mono is more than stereo. In *Proc. IEEE Int. Conf. on Robotics and Automation (ICRA)*, pages 4795 –4800, 2007.

[17] L. Di Stefano, S. Mattoccia, and F. Tombari. Zncc-based template matching using bounded partial correlation. *Pattern Recognition Letters*, 26(14) :2129 – 2134, 2005.

[18] H. Strasdat, J.M.M. Montiel, and A.J. Davison. Real-time monocular slam : Why filter ? In *Proc. IEEE Int. Conf. Robotics and Automation (ICRA)*, May 2010.

[19] S. Thrun, W. Burgard, , and D. Fox. Probabilistic robotics. *MIT Press, Cambridge, MA*, 2005.

[20] P. Viola and M. Jones. Rapid object detection using a boosted cascade of simple features. In *in Proc. IEEE Conf. Comput. Vis. Pattern Recog (CVPR)*, pages 511–518, 2001.

[21] B. Xue, W. De-ming, T. Jun-jun, and Z. Bu-hui. The hardware design and simulation of kalman filter based on ip core and time-sharing multiplex. In *Int. Conf. on Intelligent Computation Technology and Automation (ICICTA)*, volume 2, pages 266 –269, march 2011.