Projet BrasRobotique Handicap

POLYTECH GRENOBLE

UNIVERSITE JOSEPH FOURIER SCIENCES TECHNOLOGIE MEDECINE

El Hadji Malick FALL

Adji Ndèye Ndaté SAMBE

Sommaire

INTRODUCTION

Déjà traité en 2011-2012, le projet BrasRobotique est un projet qui consiste à développer à long terme un logiciel de contrôle de bras manipulateur pour l'assistance de personnes handicapées et dans lequel deux groupes de RICM4 sont impliqués. En effet, la conception de ce projet a été divisée en deux sous-projets: les aspects électroniques et techniques, ainsi que la communication entre l'utilisateur et le robot étaient du ressort de l'autre groupe, alors que nous étions en charge de tous les aspects multimédia et détection de marqueur.

Les références du robot mis à notre disposition sont les suivantes :
RB-Lyn-322 : Kit Bras Robotique AL5D à 4 Degrés de Liberté Lynxmotion

1- Contexte et domaine de l'application ciblée

Il y a de nos jours un intérêt particulièrement accordé à l'autonomie des personnes handicapées. Cet intérêt est de plus en plus grandissant à cause de facteurs tels que le phénomène du vieillissement de la population. Les pays les plus touchés par cette réalité telle que le Japon ont développé une ambitieuse politique de recherche et de développement dans ce domaine. Dans ce contexte, la robotique d'assistance est actuellement l'un des secteurs les plus investis avec de nombreuses recherches avec l'objectif permanent d'améliorer sans cesse l'autonomie des personnes dépendantes.

Il existe de nombreux produits commerciaux de bras tels que Jaco mais malheureusement celle-ci sont trop onéreuses et ne dispose pas de systèmes de commande de haut niveaux.

Notre robot, le **Kit combo bras robotique à 4 degrés de liberté AL5D de Lynxmotion** fournit un mouvement rapide, précis et répétitif. Il dispose de la rotation de la base, du mouvement de l'épaule, du coude et du poignet dans un seul plan, d'une pince fonctionnelle et d'un poignet rotatif optionnel. Ce bras robotique AL5D est un système abordable, basé sur une conception solide et une longévité. Tous les éléments nécessaires pour mettre en place des logiciels de contrôle et des systèmes de commande haut-niveau sont réunis.

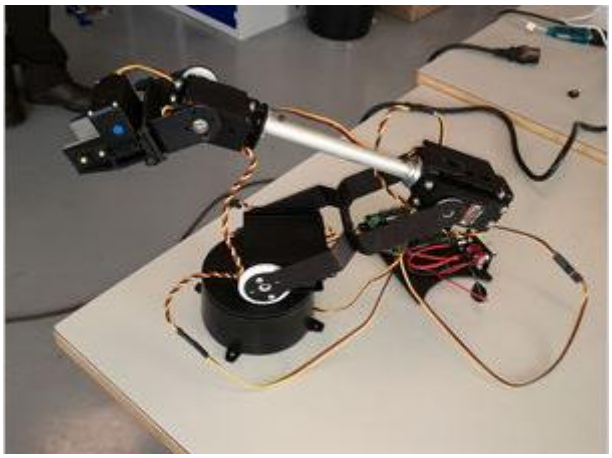


Figure 1 : Bras Robotique fourni

2- Cahier des charges résumé

Comme indiqué en introduction, le but à long terme de ce projet est de développer des logiciels de contrôle de bras manipulateur pour l'assistance des personnes handicapés.

Il doit ainsi permettre à ces personnes à mobilité réduite de pouvoir récupérer des objets tagués par un marqueur particulier. Le robot doit ainsi en mesure d'amorcer une séquence de commandes qui lui permettent d'atteindre ce marqueur.

La seule contrainte majeure qui nous a été posée pour ce projet est que le code source devait au maximum possible être en Python.

3- Implémentation

3-1- Architecture globale du projet

L'architecture est résumée par le schéma suivant :

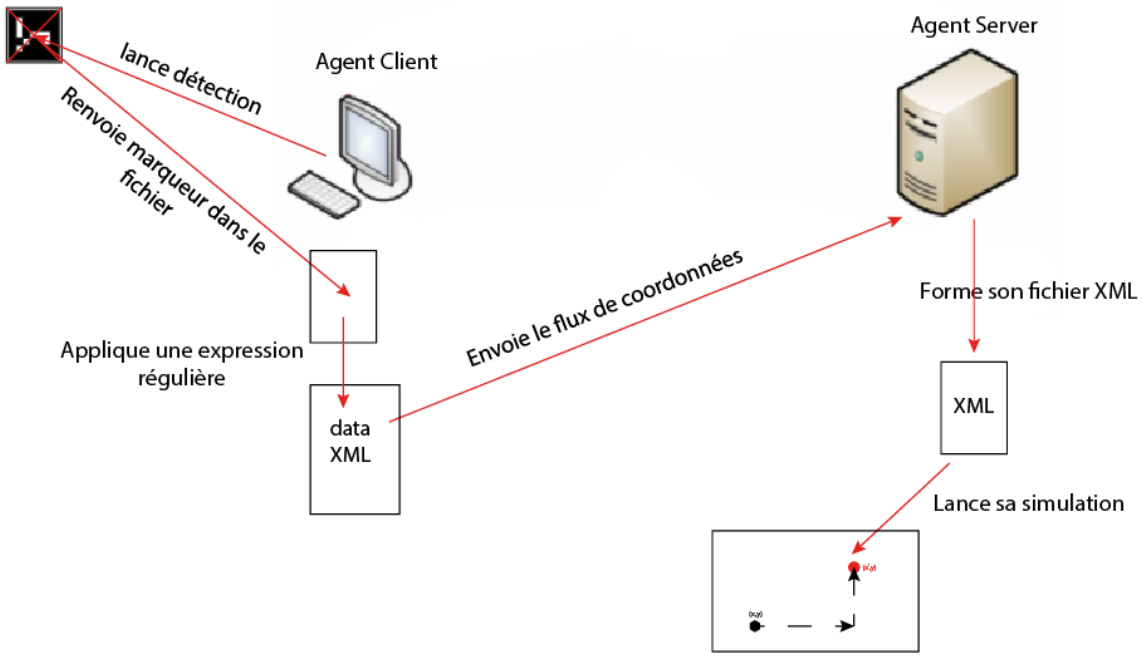


Figure 2: Architecture globale du projet

L'agent Client lance tout d'abord une détection avec ArUco et obtient un vecteur de marqueur. Il applique une expression régulière qui lui permet d'extraire les coordonnées du vecteur de marqueur et les transforme en balises XML qu'il transmet successivement à l'agent Server qui les réceptionne pour former le fichier qui lui permettra d'amorcer sa simulation.

3-2- Un choix technique particulier

Détecter un seul marqueur peut échouer pour différentes raisons telles que les mauvaises conditions, les mouvements de caméra rapide, les occlusions, etc... Pour surmonter ce problème, nous choisissons d'utiliser une matrice (grille) de marqueurs avec ArUco pour la détection. Nous choisissons également de ne récupérer que le premier marqueur vu dans cette grille.

Un marqueur étant défini soit par un rectangle ou un carré, notre choix a été de récupérer les quatre (ou au moins deux) points qui le forment. Sachant que dans un parallélogramme les diagonales se coupent en leur milieu, il devenait facile de déterminer les coordonnées du centre.

Une fois ces coordonnées obtenues, le robot effectue une translation vers ce point comme dans le schéma ci-dessous :

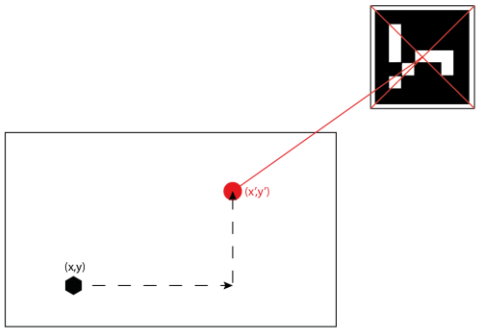


Figure3: Projection du marqueur sur une interface

3-3- La détection de marqueurs : ArUco

ArUco est une bibliothèque minimale pour les applications de réalité augmentée basée sur OpenCV. Une de ses caractéristiques de ArUco est de détecter des marqueurs avec peu de lignes de code C++. Comme nous avons pu le remarquer dans le site de source forge, chaque marqueur a un codage unique indiqué par ses couleurs noir et blanc et ce codage est défini par 5 mots de 5 bits chacun. La codification utilisée est une légère modification du Code de Hamming. Au total, chaque mot a seulement 2 bits d'information sur les 5 bits utilisés. Les 3 autres sont employés pour la détection d'erreur. En conséquence, il est possible d'avoir jusqu'à 1024 identifiants différents.

La librairie détecte les bordures et analyse dans les régions rectangulaires celles qui sont susceptibles d'être des marqueurs.

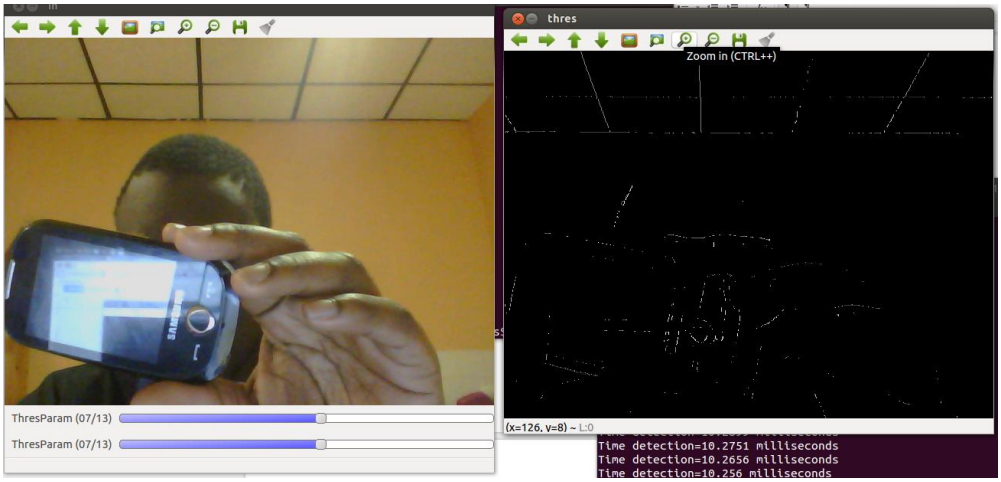


Figure4: filtre appliqué par ArUco

Ensuite, un décodage est effectué et si le code est valide, le rectangle est considéré comme un marqueur. Chaque marqueur a un codage interne

A chaque détection, ArUco renvoie des vecteurs de marqueurs. Chaque marqueur est composé d'un id et des coordonnées x, y des quatre points qui le composent.

```
177=(453.834,211.398) (510.759,209.481) (512.237,264.701)
(456.04,266.176)
```

Pour faire une détection nous exécutons le script ***aruco_test_board*** qui détecte dans un flux vidéo les marqueurs présents grâce à un fichier ***intrinsics.yml*** qui contient les informations intrinsèques à la caméra dans le format OpenCV et ***board_pix.yml*** qui contient la configuration du panneau d'affichage de la vidéo. Il contient également les identifiants des marqueurs dans le panneau d'affichage ainsi que leur agencement en pixels.

3-4- La transmission de données : les agents SPADE

Pour cette partie, suite aux TPs dont nous avons bénéficié en Applications Réparties, nous avons choisi d'utiliser le système multi-agent. Cette solution présente en effet de nombreux avantages :

- ne pas avoir à se soucier des adresses et ports d'écoutes
- facilité à créer des comportements sans se soucier des threads;
- facilité à envoyer des chaînes de caractères dans de l'ACL et de récupérer ces chaînes. On peut aussi envoyer des données sérialisées, du XML, etc
- le déploiement des agents et le référencement dans le DF (annuaire des agents) est automatique. Lorsque l'agent disparaît il est retiré du DF.

Un système multi-agent (SMA) est un système composé d'un ensemble d'agents, situés dans un certain environnement et interagissant selon certaines relations. Un agent est une entité caractérisée par le fait qu'elle est, au moins partiellement, autonome. Il peut être codé dans la plupart des langages de communication connus. Le modèle AAA (Agent Anytime Anywhere) programmé en Java est un des plus connus.

Cependant pour rester conforme à une des exigences du cahier des charges qui était d'utiliser le langage Python, nous avons mené une série de recherches qui nous ont emmenés à faire la découverte de SPADE.

SPADE est une plate-forme de développement d'infrastructure écrit en Python et basée sur le paradigme des systèmes multi-agent. SPADE est basé sur le protocole XMPP/Jabber. Il est conforme aux normes FIPA/ACL et de ce fait un agent Python peut très bien communiquer avec un agent C++ ou Java.

L'agent est donc une entité pouvant prendre des décisions toute seule ou en groupe. À chaque agent une tâche et attribuée (*Behavior, Comportement*) qui est en fait une sorte de *thread*. On pourrait par exemple imaginer un agent ayant un behavior pour envoyer des ordres à d'autres agents, ainsi qu'un behavior afin de récupérer les résultats.

On distingue différents types de behavior :

- ❏ cycliques et périodiques pour les tâches répétitives;
- ❏ One-Shot et Time-Out pour effectuer des tâches occasionnelles;
- ❏ à états (terminal), pour des comportements plus complexes;

- orienté événement. Répond à certains événements reçus par l'agent.

Lorsqu'un agent A1 envoie un message à A2, A1 doit indiquer quel behavior devra intercepter le message. En effet, chaque behavior d'un agent a sa propre liste d'attente (boîte aux lettres). Cependant l'envoi de messages est réalisé par l'agent lui-même (et pas par le behavior). C'est pour cette raison qu'on verra que la méthode d'envoi s'utilise de cette manière: *self.myAgent.send(ACLmessage)*. Il faut passer par l'attribut *myAgent* de la classe.

Dans le cas de notre implémentation, nous avons choisi le mode One-Shot et mis en place un système Client-Serveur qui communique sous une plateforme localhost via le port 8008 qui sert de canal de communication.

```
elhadjimalick@elhadjimalick-Aspire-7250: ~/Bureau/agents/spade-master
Fichier  Édition  Affichage  Rechercher  Terminal  Aide
elhadjimalick@elhadjimalick-Aspire-7250:~/Bureau/agents/spade-master$ python con
figure.py localhost
Translating localhost DNS to IP (127.0.0.1).
elhadjimalick@elhadjimalick-Aspire-7250:~/Bureau/agents/spade-master$ python run
spade.py
SPADE  2.2  <gusarba@gmail.com> - http://spade2.googlecode.com
Starting SPADE...... [done]

  [info] WebUserInterface serving at port 8008
```

L'agent Client récupère dans le fichier contenant le vecteur de marqueur ArUco les coordonnées des quatre points du marqueur qu'il envoie successivement à l'agent Server qui une fois qu'il les a reçu, les utilise pour former son fichier xml nécessaire à la simulation graphique.

3-5- La simulation graphique : modules xml.dom et Tkinter de Python

Pour simuler le comportement du robot en cas de détection, nous avons choisi de le représenter lui ainsi que le marqueur dans une simple interface graphique sous forme de cercles de différentes couleurs. Une fois que le serveur parse le fichier XML, il détermine position du marqueur qui peut être affiché à l'écran avec le bouton GetMarkerPosition. Le robot peut alors être déplacé avec les touches du claviers : ←,↑,↓,→. Dès qu'il atteint le marqueur une fenêtre pop-up est générée.

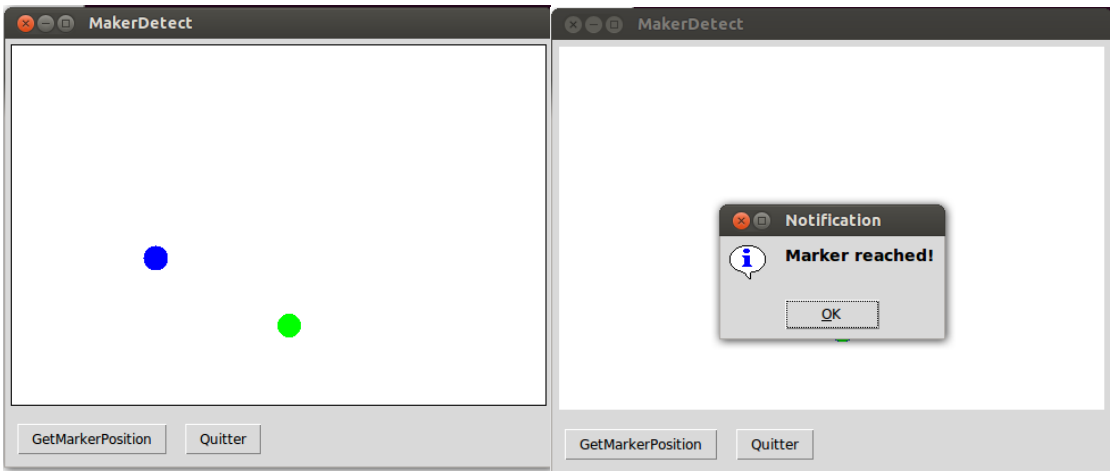


Figure5 : Interface de simulation

Pour réaliser cette interface nous avons utilisé la librairie ***Tkinter*** de Python. Pour récupérer les coordonnées nécessaires à l'affichage, nous avons utilisé la bibliothèque ***xml.dom***.

4- Métrique et Performances

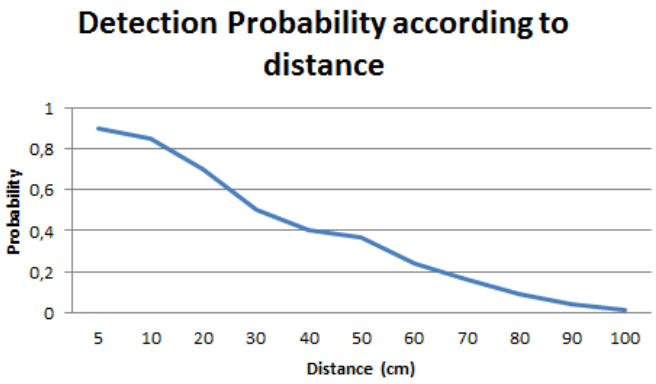
4-1- Métrique

Pour déterminer le nombre de lignes que nous avons écrit, nous avons utilisé le package SLOC. Les lignes de code fournies par les libraires ne sont pas comptabilisées.

Fichier	Langage	Commentaires	Ligne de code
simulation.py	Python	47	56
server.py	Python	13	33
client.py	Python	30	60
aruco_test_board.cpp	C++	68	142 (132 de base +10)

4-2- Performances

Durant l'étape intermédiaire, nous avons effectué une série de mesures avec la webcam intégré de notre ordinateur de résolution 1600x900, afin d'évaluer la performance de la détection. Nous avons simulé la détection dans des conditions assez médiocres (photographie des marqueurs sur un téléphone de résolution moyenne). Nous avons obtenu la courbe suivante :



Nous constatons que plus la distance est proche, plus on a de chances de détecter un marqueur. La distance optimale dans ces conditions est estimée à 14 cm et moins. Mais dans des conditions plus favorables (lumière bien calibrée, marqueur bien visible, photographie de meilleure résolution), nous pouvons souligner que cette distance optimale passe à 40 cm.

Nous avons également effectué des tests sur le nombre de marqueurs nécessaires. Nous avons généré plusieurs matrices de marqueurs dont nous avons testé la détection. Voici les résultats obtenus :

1	2	3	4	5	6
2					
3					
4					
5					
6					

Success Failure

Nous pouvons constater que l'idéal serait d'avoir une taille minimum de 4x5 pour garantir l'optimalité de la détection. C'est à dire d'être qu'un marqueur sera détecté.

Dans sa globalité, la performance du système est limitée par le fait qu'il y a des étapes qui ne se font pas en temps constant, tant du côté du client, (extraction de coordonnées dans le vecteur du marqueur et transformation en XML), que du côté serveur (lecture du contenu de chaque ligne du fichier XML). Ces étapes sont dues au format renvoyé par ArUco, qui ne permet pas l'instantanéité et un transfert automatique.

Le transfert via les agents se fait par contre très rapidement.

5- Gestion du projet

Après avoir établi la répartition pour les deux grandes parties du projet, nous nous sommes entièrement consacrés aux aspects que nous devions développer tout en organisant de temps de temps des rencontres avec l'autre groupe pour s'accorder sur les solutions proposées. Cependant nous avons décidé d'implémenter une solution indépendante de la leur, qui prouve le bon fonctionnement de notre programme.

La fiche de suivi de notre projet est disponible sur le lien suivant :
http://air.imag.fr/index.php/Proj-2013-2014-BrasRobot-Handicap-1

6- Problèmes rencontrés et solutions élaborées

Durant notre implémentation, nous avons rencontré plusieurs difficultés. La première était de savoir quelle démarche adopter puis une fois cette étape franchie, de comprendre les librairies que nous avons dû utiliser, ArUco et SPADE, dont les documentations sont très peu disponibles et qui sont programmés dans des langages que nous n'avions pas l'habitude d'utiliser, à savoir C++ et Python.

Par la suite nous nous sommes heurtés à des problèmes techniques comme la structure de fichier renvoyé par ArUco. Comment extraire les coordonnées du vecteur de marqueur ci-dessous ?

```
76=(432.647,306.132)  (435.541,252.122)  (487.151,254.896)  (483.938,307.729)
Txyz=-999999 -999999 -999999 Rxyz=-999999 -999999 -999999
```

Pour se faire, nous avons implémenté une solution basée sur une expression régulière : [0-9.]* (qui renvoie tous les nombres séparés ou non par une virgule présents dans le vecteur sous forme de tableau) grâce au module ***re*** de Python.

Ce problème résolu il s'agissait de transférer de transformer le flux de coordonnées en format XML. Mais la validité de ce format était corrompue par un message intrinsèque au server de SPADE qui transitait dans le réseau :

```
<presence xmlns="jabber:client" to="agent_server@127.0.0.1"
type="subscribed" id="1" from="ams.127.0.0.1" />
```

En faisant une comparaison sur le début de la chaîne du message, nous avons pu empêcher l'écriture de ce message dans le fichier à transmettre.

7- Perspectives de développement

La solution que nous avons proposée est loin d'être optimale. Sachant que nous utilisons un vecteur de marqueur, rien ne garantit que le marqueur trouvé soit le plus intéressant à atteindre. Une amélioration possible serait de recentrer la caméra vers le marqueur "central" de la matrice.

Un autre fait est que ArUco renvoie que des coordonnées (x,y). La possibilité d'obtenir une coordonnée z rendrait la simulation plus réaliste.

Concernant la transmission de données, transmettre directement le fichier XML au lieu de le faire ligne par ligne sous forme de messages, rendrait notre programme plus performant et instantané.

CONCLUSION

Pour conclure, notre programme apporte une nouvelle fonctionnalité haut-niveau dans l'utilisation du bras robotique, mais aussi tout type de robot qui utilise un système de commande. Nous sommes actuellement en mesure de détecter un marqueur et de renvoyer ses coordonnées. En outre, grâce au travail du deuxième groupe, qui a développé une interface de simulation avancée, il sera possible de guider le robot vers la position à atteindre.

Basé sur la solution du système multi-agent, notre solution est facilement transférable, quel que soit le langage ou la plate-forme utilisée par les différentes entités du projet.