Rapport de stage d'option

Comportement réactif pour l'évitement d'obstacles en Robotique : application à un bras manipulateur

Mai Nguyen
X2004
Ecole Polytechnique

Directeur de Stage : Pr. Oussama Khatib
Maitre de Stage:Vincent Padois
Lieu :Manipulation Group, AI Lab., Stanford University

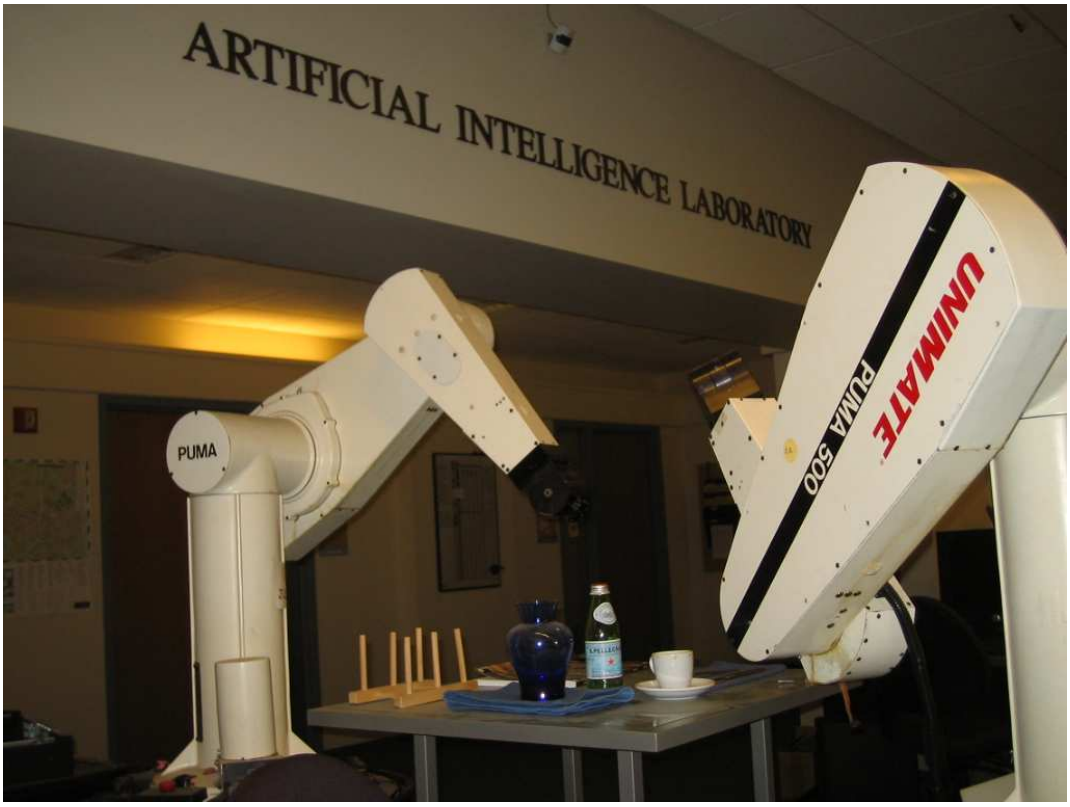

Table des Matières

Introduction

Les usines font un usage massif des robots qui leur sont de plus en plus utiles et irremplaçables tant au niveau de la puissance que de la précision ou de la vitesse. Cependant, les robots restent inexistants en dehors de cet environnement industriel. En effet, les robots humanoïdes évoluent encore difficilement dans l'environnement humain. Contrairement à la robotique industrielle où la position de l'obstacle est connue à l'avance, l'espace dans lequel nous vivons n'est pas tout prédéterminé et balisé comme dans les usines, mais contient au contraire beaucoup d'obstacles mobiles qu'il lui faut gérer en temps réel. Le robot doit être sur et être capable de contourner les obstacles et ne pas développer un comportement dangereux lors d'une collision inévitable. Le robot doit donc répondre à cette double exigence tout en effectuant sa tâche (voir Figure1).

Figure 1: Exemple de robot humanoïde en train d'effectuer une tache tout en evitant un obstacle

Dans l'optique de pouvoir intégrer un robot dans un environnement humain de la vie de tous les jours, l'objet du stage est donc d'étudier comment commander un robot pour effectuer certaines taches tout en évitant les obstacles mobiles en temps réel. Pour cela, nous utiliserons un bras articule, le PUMA, qui constitue un système moins complexe qu'un humain ou un robot humanoïde. Avant de gérer un robot humanoïde comme Asimo de Honda qui a 22 degrés de liberté (ce qui reste moins complexe que la cinquantaine de degré de liberté chez l'être humain) et qui aurait à effectuer des taches complexes sous la contrainte d'équilibre, nous nous sommes limités à un système à 6 degrés de libertés, ce qui nous permet de faire des calculs en temps réel.

Dans ce rapport, nous présenterons d'abord le matériel dont le système de détection de l'obstacle et le PUMA et la méthode d’évitement de l’obstacle choisise parmi celles généralement adoptées, avant de voir la commande du robot implementée, et de discuter de la pertinence des résultats obtenus.

1 Matériel et détection de l'obstacle

1.1 Le PUMA

Pour notre étude, nous utilisons un bras articule : le PUMA ou Programmable Universal Machine for Assembly (voir Figure 2).

Les robots de la gamme PUMA sont probablement les robots les plus répandus dans les universités.

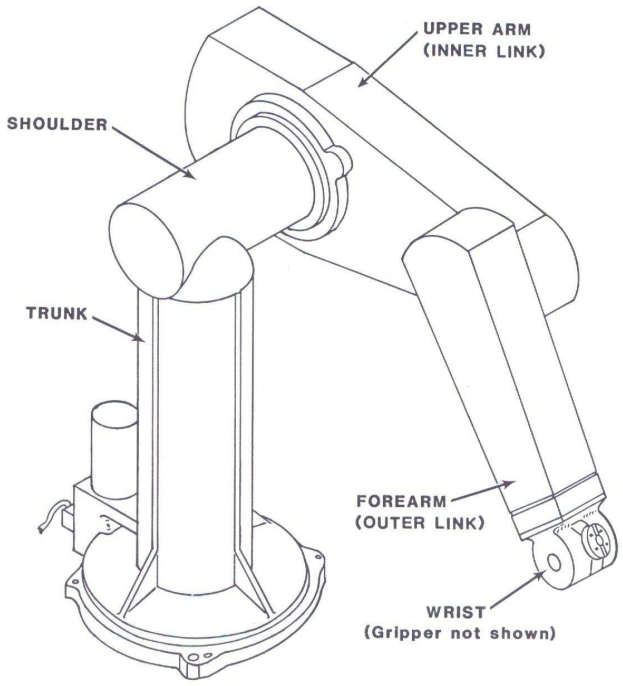

Figure 2: Schema du PUMA : bras articulé

Le PUMA (Programmable Universal Machine for Assembly) a été originellement conçu par Vic Schienman et finance par General Motors et The Massachussets Institute of Technology en 1969, et fut produit pendant de nombreuses années par Unimation (société qui fut rachetée plus tard par Westinghouse, avant d'être revendue a Staubli, une grande société Suisse de robotique). Le bras manipulateur est le composant mécanique du système et comporte six axes de rotation, chacun contrôlés par un servomoteur à courant continu :

- 1 pour rotation du tronc
- 1 pour rotation de l'épaule

- 1 pour le coude
- 3 pour le poignet

La rotation de chaque articulation du bras manipulateur est assurée par un servomoteur à courant continu et à aimant permanent, par l'intermédiaire d'un réducteur. Chaque moteur est associé à un codeur incrémental et à un potentiomètre, ainsi qu'à un réducteur de rapport 116/1. Le bon fonctionnement du PUMA nécessite un contrôle de la position et de la vitesse de rotation de chaque articulation du robot.
Les changements de position de chaque articulation sont fournis par les codeurs, alors que les informations de vitesse de rotation sont calculées par l'ordinateur du robot.

Les servomoteurs pour les trois axes principaux (axes 1, 2 et 3) sont équipés de freins électromagnétiques. Ces freins sont activés lorsque l'alimentation de ces moteurs est coupée, et maintiennent donc le bras du robot en position fixe. Ceci est une sécurité destinée à éviter les risques de blessures ou de casse sur le robot lorsque l'alimentation est coupée accidentellement (coupure de courant, etc...).
L'alimentation électrique est fournie par le contrôleur (ordinateur de commande), par l'intermédiaire d'un câble par lequel passent aussi toutes les informations de position et de vitesse de déplacement.

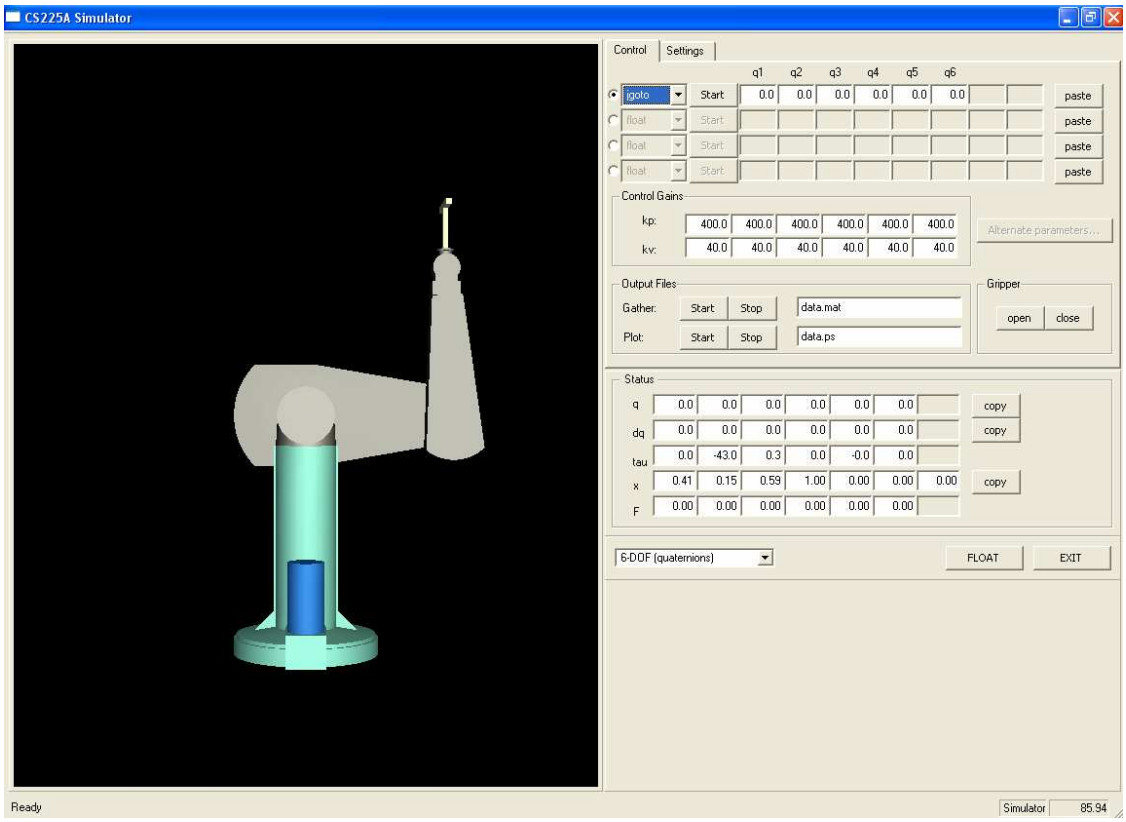

Figure 3: Simulateur du PUMA

Le contrôle effectif du robot se fait à travers QNX, système opérationnel temps réel fait pour des applications embarquées, et largement utilisé en robotique. Il permet d'envoyer directement le couple de commande à chaque articulation du robot. Cette méthode de commande directe et précise est celle adopté par le groupe de recherche, et se démarque des contrôles en position ou en vitesse.

Pour le développement de la commande d'évitement d'obstacle, nous avons par contre utilisé un simulateur du robot (figure 3). Celui-ci dispose d'une interface graphique complète et d'un modèle du robot complet. Il s'agissait de rajouter au simulateur les commandes du robot, que cela soit pour effectuer des taches ou pour gérer les obstacles.

1.2 Le système Atracsys

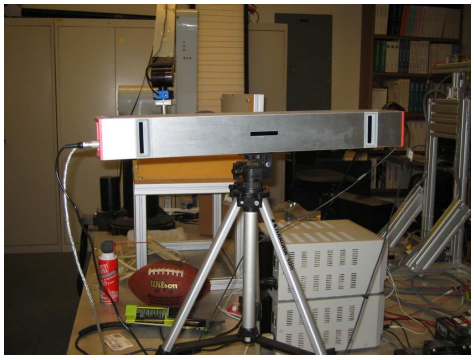

Figure 4: détecteur Atracsys avec 3 CCD

Figure 5: marqueur muni de 4 leds

Pour simuler l'obstacle et avoir un moyen simple d'évaluer le contrôleur développé pour l'évitement d'obstacles, nous avons choisi d'utiliser un système optique appelé Atracsys. Atracsys-X2, développé par le Groupe de Réalité Virtuelle et Interfaces Actives (VRAI Group) de l'EPFL, est un système de mesure optique 3D de haute précision. A l'aide de 3 caméras et de capteurs appropriés, cet appareil permet de situer un marqueur et son orientation dans un espace de travail en temps réel. Il effectue 30 000

mesures par seconde avec une précision inférieure à 0.3 mm et est sujet à un délai inférieur a 10ms. Les marqueurs du système, composés de 4 leds infra rouges (figure 5) sont détectés par 3 CCD (Charge Coupled Device) linéaires (figure 4). Leur position et orientation sont calculées dans l'espace 3D par triangulation.

Le choix d'utiliser ce système a été effectué pour la simplicité d'utilisation du système, son utilisation en temps réel et pour que le laboratoire puisse utiliser cet outil dans le futur.

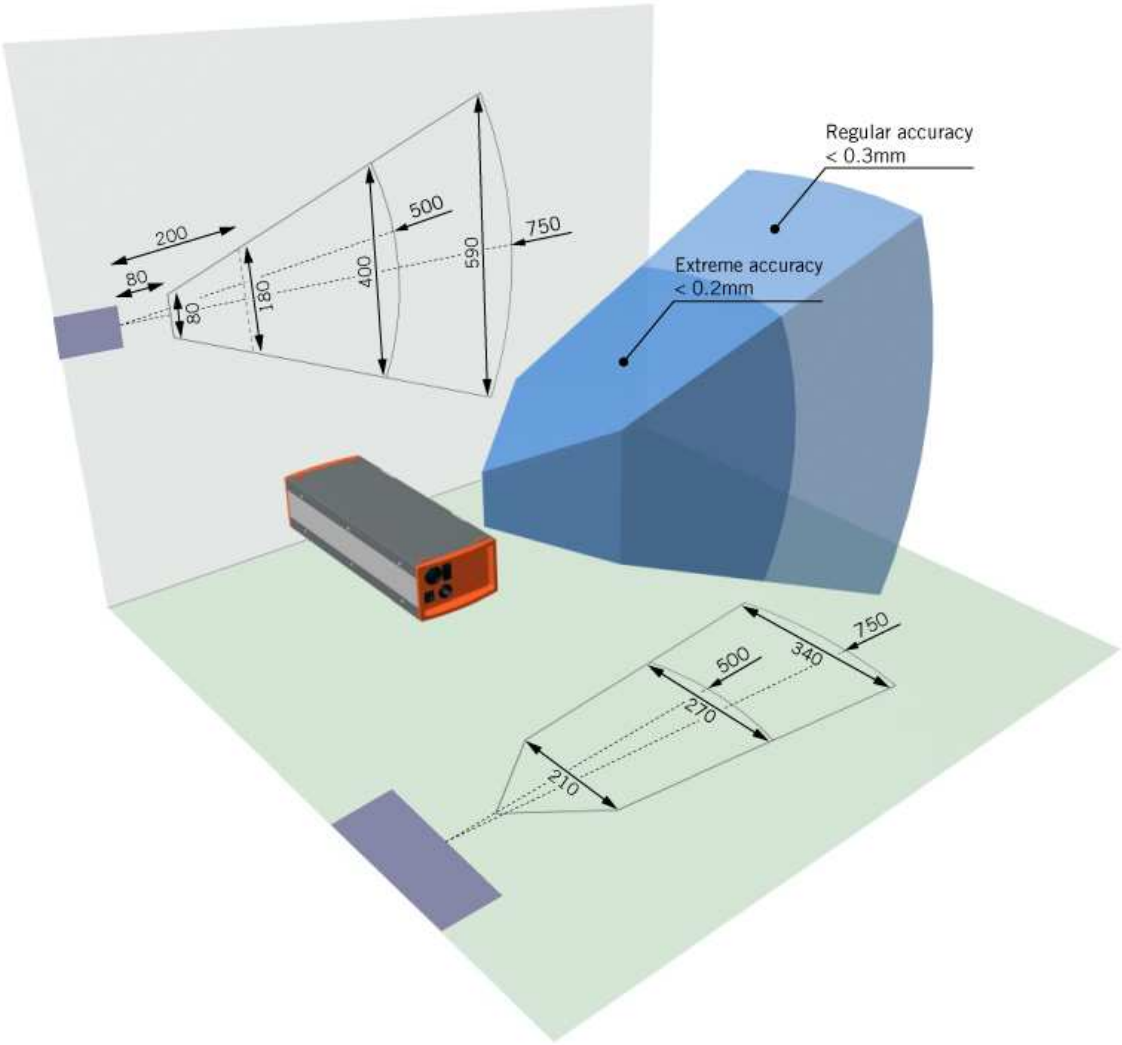

Figure 6: espace de travail de l'Atracsys

Le matériel est fourni avec driver et une librairie documentée qui permet d'initialiser l'appareil et les marqueurs. Il a donc fallu utiliser cette librairie pour récupérer les informations du marqueur pour les envoyer au robot (voir Annexe 1).

1.3 Définition du projet

Pour éviter les obstacles, deux méthodes sont généralement appliquées :

- la planification de trajectoire qui cherche à aller d'un point A à un point B, en cherchant grosso modo à répondre à la question : est-ce que deux points peuvent être connectés par un chemin continu étant donnés les contraintes ? Les techniques développées sont très souvent lentes et laborieuses. Et les plus rapides sont généralement spécifiques à des situations données.
- l'évitement réactif qui permet d'éviter rapidement les obstacles, en déviant sa trajectoire à la vue de l'obstacle. Cette méthode présente l'inconvénient de bloquer le robot en cas de situation complexe à plusieurs obstacles dans un minimum local. Mais il permet une réactivité plus rapide et s'applique à des situations plus générales.

C'est la deuxième méthode que nous avons choisie d'adopter pour notre projet, car elle est la plus facilement transposable dans toutes les situations et car elle présente un temps de réponse plus rapide.

L'évitement de l'obstacle en lui-même est d'abord réduit à l'évitement de l'obstacle par l'effecteur, puis par l'avant-bras modélisé par un ensemble de 2 segments effecteur-poignée et poignée-coude.

Même si le bras lui-même ne se déplace pas, il reste soumis à des contraintes de mobilité. Pour intégrer au mieux cette contrainte de mobilité et d'autonomie et construire une commande du robot indépendante du système de détection de l'obstacle, et parce que le détecteur Atracsys est utilise sous Windows et non QNX, nous avons sépare le système de commande du bras et la commande du robot. La transmission de données se fait via une communication UDP.

Par ailleurs, le PUMA est certes un robot facilement manipulable, mais il peut développer des couples très importants. Ajouté à cela son poids important, il peut donc s'avérer dangereux en cas d'instabilité. C'est pourquoi nous choisissons d'utiliser un simulateur avant toute mise en pratique sur le robot réel. Ce simulateur est utilisé non sous QNX mais sous Windows. Ainsi, même si nous utilisons Windows dans notre projet avec le simulateur, les deux parties du programme communiquent via des sockets UDP.

L'obstacle, quant à lui est modélisé par une sphère de taille réglable, dont le centre se trouve au centre du marqueur.

2 La commande du Puma

2.1 Définition des taches standard à effectuer

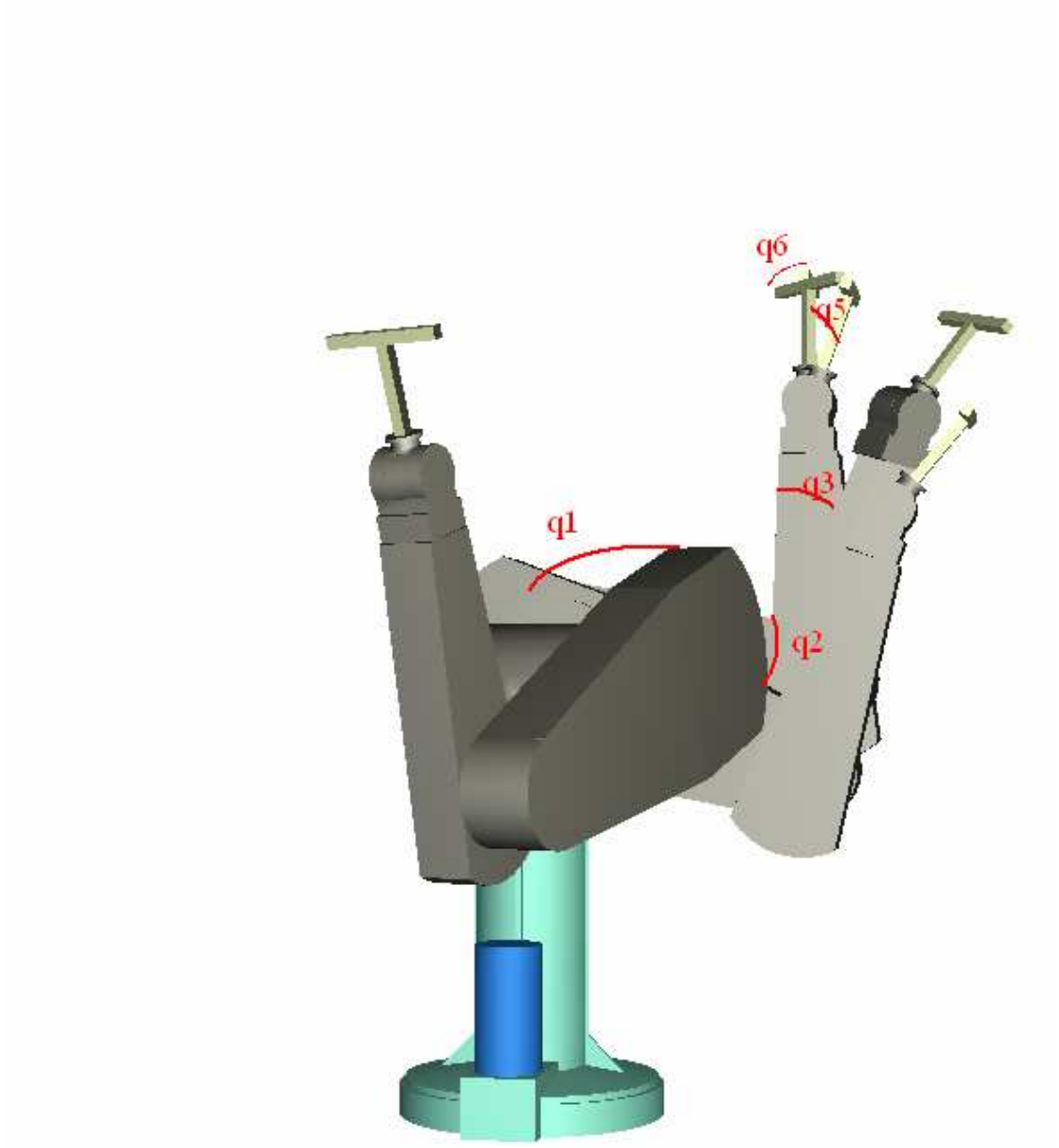

Figure 7: Les 6 paramètres articulaires du PUMA sont les angles de ses articulations q=(q1,q2,q3,q4,q5,q6)

Le puma est composé de servomoteurs qui commandent les couples à chaque articulation. Il est donc naturel de commander directement chacune des articulations en couple de force. Cette approche en coordonnées articulaires, où les paramètres cinématiques sont les angles ou longueurs des articulations (figure 7), présente

l'avantage d'être simple et d'accéder à toutes les configurations permises par le robot. Par contre, cette approche est peu adaptée pour suivre une trajectoire particulière ou pour interagir avec le monde extérieur où on a besoin de connaître les coordonnées du robot. Il n'y a aucune garantie que nous puissions pour toute position désirée résoudre la cinématique inverse pour trouver les angles des articulations correspondantes, et encore moins trouver une configuration unique ou une trajectoire continue (figure 8).

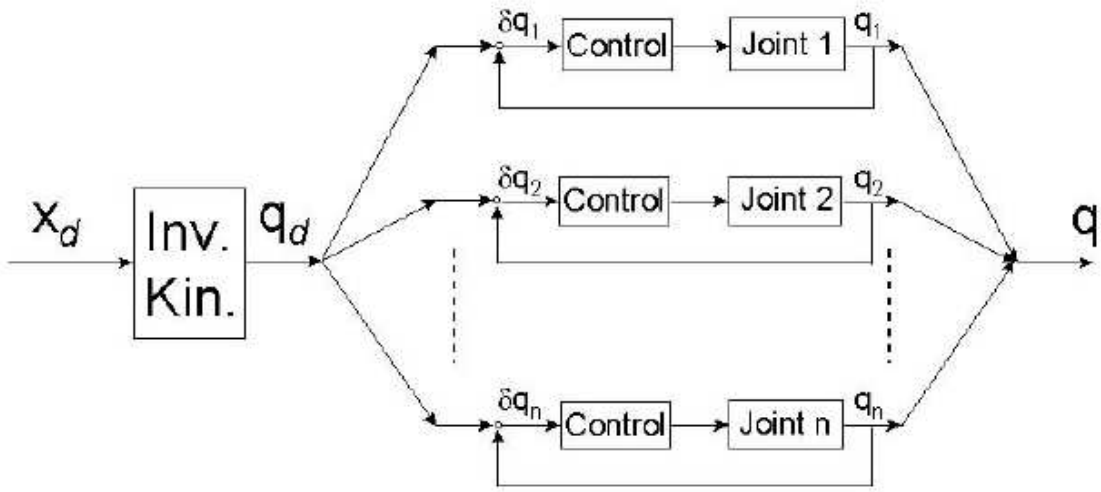

Figure 8: Approche en coordonnées articulaire (joint space control) avec q les paramètres du robot et x le vecteur positon et orientation du robot

L'approche en coordonnées opérationnelles, qui consiste uniquement à repérer la position et l'orientation de l'effecteur, se rapproche plus des exigences d'une trajectoire prédéfinie de manière précise. Néanmoins, pour effectivement appliquer les couples adéquats aux joints, il est nécessaire de disposer d'un modèle pour remonter d'un déplacement infinitésimal δx à un changement infinitésimal δq sur les articulations. Il faut donc résoudre la cinématique inverse en tout point de la trajectoire pour les déplacements infinitésimaux. Ces calculs doivent être effectués à chaque boucle de contrôle des moteurs, ce qui peut s'avérer fort coûteux (voir la figure 9).

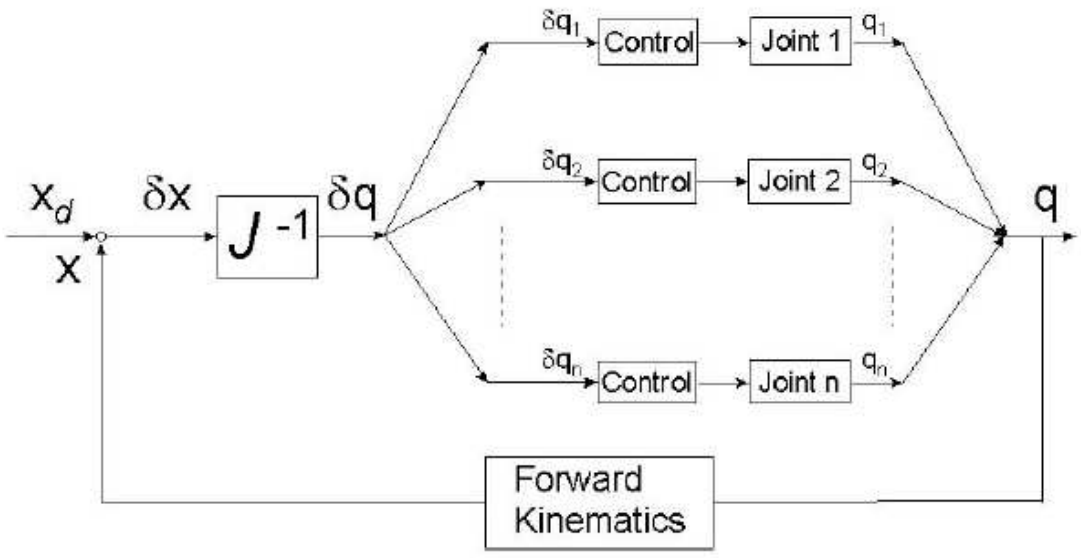

Figure 9: Approche en coordonnées opérationnelles (operational space control) avec q les paramètres du robot et x le vecteur positon et orientation du robot

Devant les deux approches équivalentes qui présentent toutes deux leurs avantages et désavantages suivant les objectifs de tache, nous avons décidé d'implémenter des contrôles pour les deux approches. C'est pourquoi nous avons réuni tous les paramètres cinématiques du robot sous l'objet GlobalVariables& gv (voir annexe 2) et nous avons construit les fonctionnalités suivantes :

Fonctionnalité	Fonction dans l'approche articulaire (joint space control)	Fonction dans l'approche opérationnelle (operational space control)
Compensation de la gravité : le robot est souple et ne fait que compenser son poids	Float	Float
Tenir sa position : quelle que soit les perturbations introduites, le robot revient à sa position initiale	Jhold	Hold
Aller a une position désirée sans saturation de vitesse (de manière linéaire, avec un contrôle Proportionnel Dérivé)	Jmove	Move
Aller a une position désirée avec saturation de vitesse (de manière linéaire, avec un contrôle Proportionnel Dérivé)	Jgoto	Goto
Aller a une position désirée en suivant une trajectoire cubique (dans le temps pour avoir un démarrage et un arrêt moins abrupts et un mouvement plus souple, et avec un contrôle PD)	Jtrack	Track
Maintenir une orientation verticale pour l'effecteur (pour porter un plateau horizontal par exemple)		Pfmove

2.2 Couple à appliquer pour l'approche articulaire

2.2.1 Couple à appliquer à une articulation pour atteindre un angle donné dans un modèle statique

Dans le cas de l'approche articulaire, le couple à appliquer en vue d'une configuration donnée est simple à implémenter. A un angle q et une accélération angulaire dq, il faut pour attendre l'angle qd, appliquer le couple PD(Proportionnel Dérivé):

tau = - kp*(q- qd)-(kv * dq)+ G

où qd et dq sont les vecteurs parametres du robot réel, désiré et son dérivé (vitesse)

kp est un coefficient de proportionalité

G est couple pour compenser la gravité

L'implementation complète de cette fonction se trouve dans void njmoveControl (GlobalVariables &gv) de controlDLL.cpp dans l'annexe 3.

2.2.2 Couple à appliquer à une articulation pour atteindre un angle donné dans un modèle dynamique

Cette première approche ne tient pas de la dynamique du robot et notamment des forces de Coriolis et de d'entraînement qui font fortement dévier la position finale en cas de grande vitesse, et peuvent même introduire des instabilités. D'après l'équation du mouvement de Lagrange,

$$\frac{d}{dt}(\frac{\partial L}{\partial \dot{\mathbf{q}}}) - \frac{\partial L}{\partial q} = \tau$$

où

$$L = K - V$$

est la différence entre l'énergie cinétique et l'énergie potentielle,

et q est le vecteur des paramètres du robot (ici les angles des articulations),

le couple à appliquer pour compenser ces effets dynamiques est :

tau =A*(-kp*(q-qd)-kv *dq)+v+G

où q, qd et dq sont les vecteurs parametres du robot réel, désiré et son dérivé

A(q) est la matrice masse du robot

G(q) est le couple pour compenser la gravite

kp et kv sont deux vecteurs coefficients de proportionnalité

et v(q) est la force de Coriolis et d'entraînement facilement calculable a partir le la matrice masse du robot :

$$\mathbf{v}(\mathbf{q}, \dot{\mathbf{q}}) = \dot{M}\dot{\mathbf{q}} - \frac{1}{2}\begin{bmatrix} \dot{\mathbf{q}}^T \frac{\partial M}{\partial q_1}\dot{\mathbf{q}} \\ \vdots \\ \dot{\mathbf{q}}^T \frac{\partial M}{\partial q_n}\dot{\mathbf{q}} \end{bmatrix}$$

L'implémentation complète de cette fonction se trouve dans void jmoveControl (GlobalVariables &gv) de controlDLL.cpp dans l'annexe 3.

2.2.3 Saturation de vitesse pour chaque articulation

Il faut maintenant pour assurer une sécurité minimale, limiter la vitesse du robot. On introduit alors une vitesse de saturation dq_{max}. Pour limiter la vitesse, considérons le couple appliqué sous la forme

tau =-A*kv*(dq –(-kp/kv*(q-qd))+v+G

On voit une vitesse désirée apparaître sous la forme

vit = -A*(kp*(q-qd))/ (A*kv)

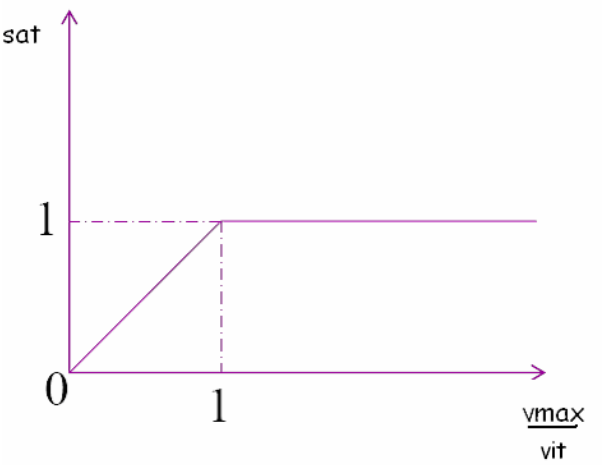

Figure 10: Fonction de saturation de la vitesse du PUMA

D'où, avec sat la fonction représentée en figure 10), le couple final :

tau =-A*kv*(dq – sat)+v+G

L'implémentation complète de cette fonction se trouve dans void jgotoControl (GlobalVariables &gv) de controlDLL.cpp dans l'annexe 3.

2.3 Couple à appliquer pour l'approche opérationnelle

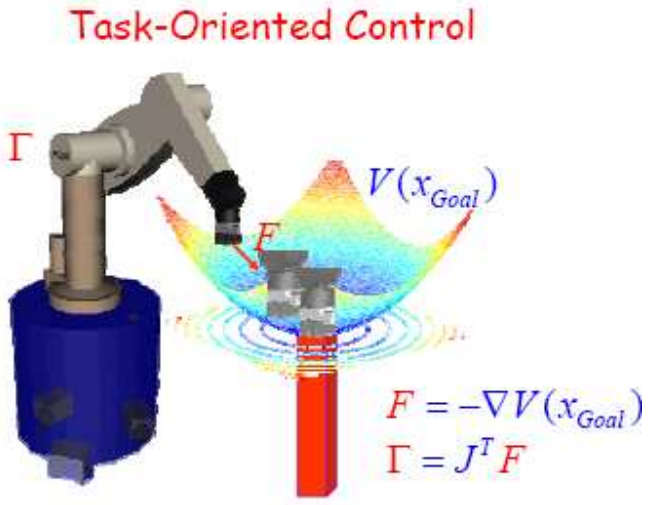

Figure 11: Potentiel attractif artificiel subi par le robot

2.3.1 Commande du robot pour atteindre une position donnée dans un modèle statique

Dans le cas de l'approche opérationnelle, au lieu de faire un double calcul de cinématique inverse pour calculer les paramètres q et qd correspondant aux positions de l'effecteur x actuel et xd désiré, nous simulons un potentiel attractif dont le minimum se trouve a la position xd désirée (figure 11):

$$V(x) = \frac{1}{2}k_p(x - x_d)^2$$

ce qui correspond a la force

$$f = -\nabla V(x) = -\frac{\partial V}{\partial x}$$

Le couple correspondant est

$$\tau = J^T \mathbf{F}$$

où J est la Jacobienne du robot et le caractérise. Elle est définie par

$$\dot{\mathbf{x}}_{(m\times 1)} = J(\mathbf{q})_{m\times n}\dot{\mathbf{q}}_{(n\times 1)}$$

Donc le plus intuitif pour obtenir un control direct en position est d'appliquer le couple

$$f = -k_p(x - x_d) - k_v\dot{x}$$

tau= Jtranspose*F+ G;

où kp et kv sont des coefficients de proportionnalité

Jtranspose est la matrice transposée de la Jacobienne

G est le couple correspondant à la gravité

L'implémentation complète de cette fonction se trouve dans void ngotoControl (GlobalVariables &gv) de controlDLL.cpp dans l'annexe 3.

2.3.2 Commande du robot pour atteindre une position donnée dans un modèle dynamique

Encore une fois, cette première solution est placée dans un cadre statique et ne tient pas des forces d'inertie du robot en mouvement. La mise en équation complète donne la correction suivante :

tau= Jtranspose*(Lambda*F)+ B+ G;

où kp et kv sont des coefficients de proportionnalité

Jtranspose est la matrice transposée de la Jacobienne

Lambda est la matrice masse du robot exprimée dans les paramètres de l'espace opérationnel

B est le couple correspondant aux forces d'inertie (Coriolis et centrifuge)

G est le couple correspondant à la gravité

2.3.3 Saturation de vitesse

De même que pour l'approche articulaire, nous cherchons à limiter la vitesse du PUMA pour des raisons de sécurité, et essayons de saturer la vitesse qui apparaît dans l'expression de la force par une vitesse maximale vmax :

$$v = \frac{k_p}{k_v}\left(x_g - x\right)$$

Ceci nous permet de comparer la norme du vecteur v et vmax et de prendre comme force:

$$F^* = -k_v\left(\dot{x} - sat\, v\right)$$

où sat est le scalaire défini par la figure 10.
L'implémentation complète de cette fonction se trouve dans void gotoControl (GlobalVariables &gv) de controlDLL.cpp dans l'annexe 3.

2.4 Evitement d'obstacle sous contraintes

2.4.1 Evitement d'un obstacle par un point ponctuel

En la présence d'un obstacle, il faut pouvoir repérer le robot par rapport au monde extérieur. La meilleure approche est par conséquent celui des coordonnées opérationnelles qui permettent de repérer directement les relations relatives entre le robot et l'obstacle. Mais cette fois-ci, au lieu de s'approcher d'un point désirer, il s'agit de s'éloigner de d'une sphère. Au lieu de simuler un potentiel attractif, nous pouvons introduire un potentiel répulsif autour de l'obstacle. Pour rendre le comportement du robot indépendant de l'obstacle quand il en est loin, nous devons l'annuler au delà d'une certaine distance. Ainsi nous pouvons prendre (voir figure 12)

$$U_{\mathcal{O}}(\mathbf{x}) = \begin{cases} \frac{1}{2}\eta(\frac{1}{\rho} - \frac{1}{\rho_0})^2 & \text{if } \rho \leq \rho_0 \text{ ;} \\ 0 & \text{if} \rho > \rho_0. \end{cases}$$

où ρ est la distance entre le robot et l'obstacle,
et ρ_0 est la distance au-delà de laquelle le robot ne « voit » pas l'obstacle dans sa planification de mouvement.

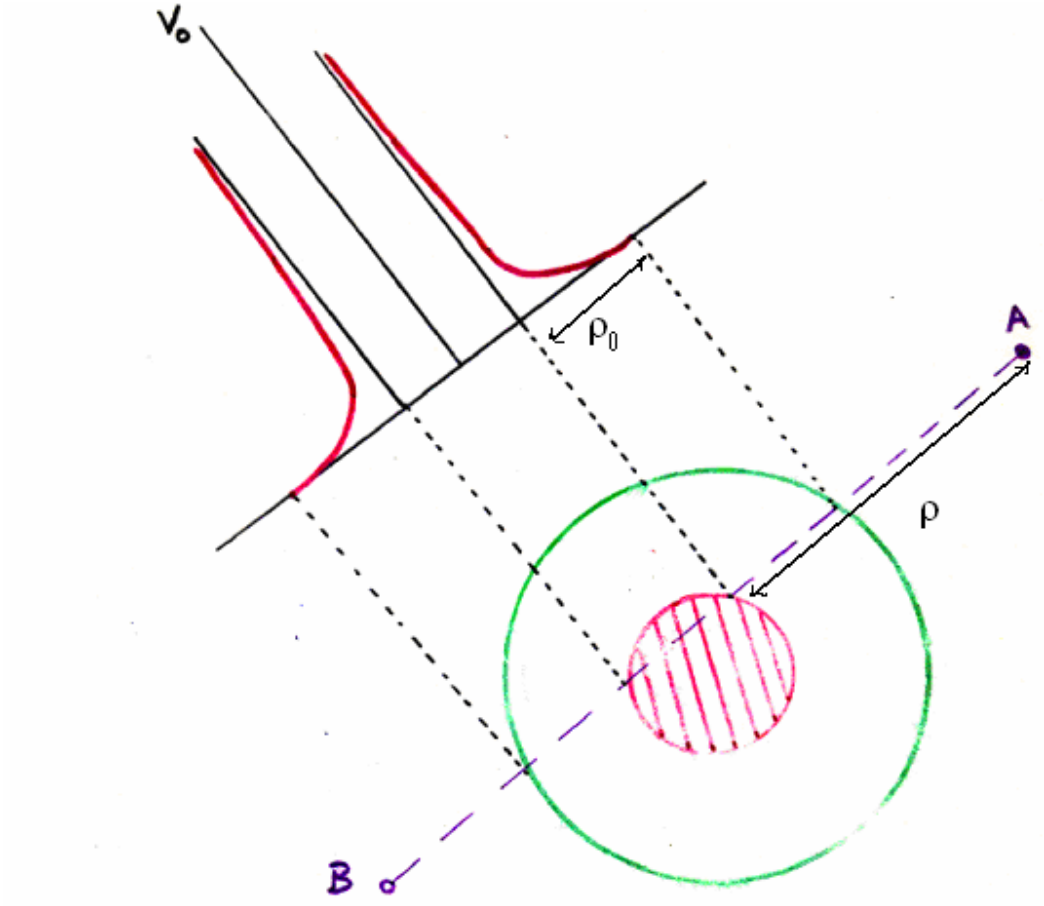

Figure 12: Potentiel répulsif subi en présence de l'obstacle

Donc la force introduite est

$$F = \frac{\eta}{2}\left(\frac{1}{p} - \frac{1}{p0}\right)\frac{1}{p^3}(\mathbf{PosObstacle} - \mathbf{PosEffecteur})$$

2.4.2 Evitement d'obstacle par un squelette

La formule précédente permet simplement à l'effecteur d'éviter les obstacles. Pour l'étendre au squelette du bras (voir la fonction PrVector ObstacleAvoidance (GlobalVariables& gv, PrVector3 &point1, PrVector3 &point2) de controlDLL.cpp dans l'annexe 3), il faut d'abord :

- modéliser le squelette par un ensemble de segments. Ainsi pour l'instant nous considérons que notre bras est un ensemble de 2 segments effecteur-poignet et poignet-coude (312)

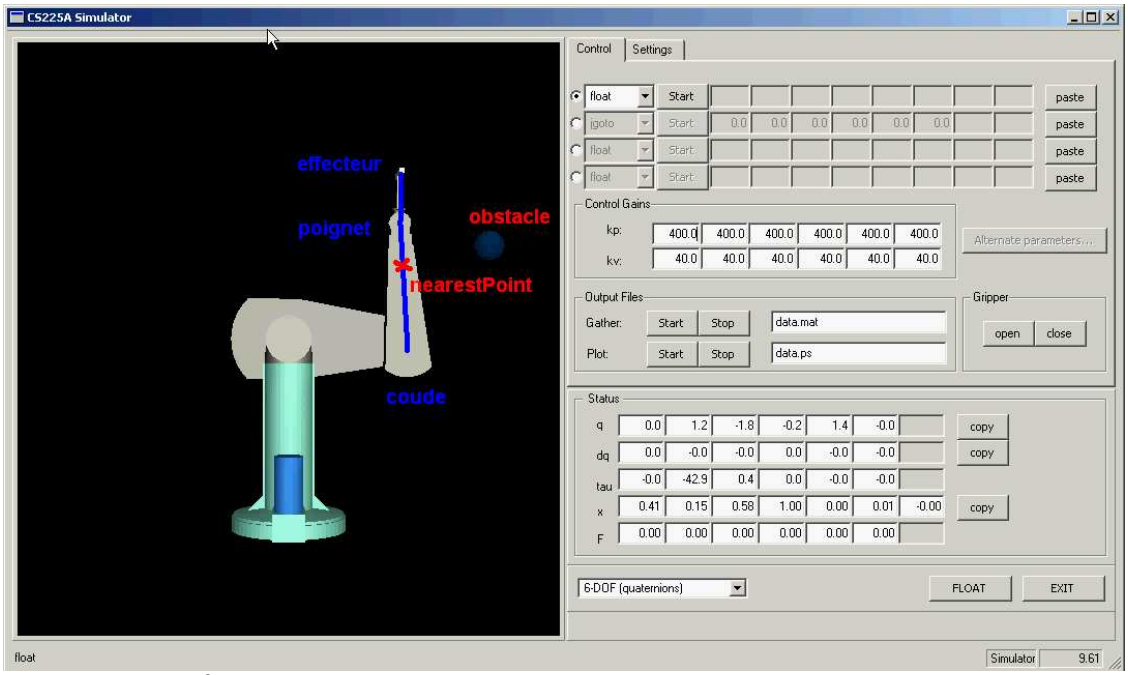

Figure 13: évitement de l'obstacle par le squelette du robot

- trouver pour chaque segment le point nearestPoint le plus proche de l'obstacle par projection de l'obstacle sur le segment
- trouver la force qu'il faut appliquer en ce point. Il suit la formule précédente :

$$F = \frac{\eta}{2}\left(\frac{1}{p} - \frac{1}{p0}\right)\frac{1}{p^3}(\mathbf{PosObstacle} - \mathbf{nearestPoint})$$

- trouver la force et le couple correspondants à appliquer à l'effecteur, car la force que nous savons appliquer au robot correspond à celui que nous appliquons à l'effecteur. Il s'agit en fait de la même force **F**, et le couple correspondant est : **F**x(**x-nearestPoint**) , où x est la position de l'effecteur.

2.4.3 Effectuer une tache prioritaire en évitant les obstacles

Nous avons maintenant une fonction qui permet au robot d'éviter parfaitement les obstacles. Cependant, il faut savoir intégrer cette fonctionnalité dans un contexte d'une tache prioritaire. Il faut donc pour que cette sous tache n'influence pas la tache prioritaire, projeter la force que nous venons de calculer sur l'espace des états possibles de la tache prioritaire. D'où

```
joint_obstacle = kj *(Jtranspose*forceProjetee);
forceProjetee= (nullSpaceTask.transpose()*force);
```

où Kj est un coefficient de proportionnalité

Jtranspose est la transposée de la Jacobienne

nullSpaceTask.transpose est la transposée de la matrice de projection sur l'espace définie par la tache

et force est la force précédemment calculée pour éviter l'obstacle.

Le PUMA essaie désormais d'éviter l'obstacle tout en effectuant sa tache principale et prioritaire. Le contrôle principal du PUMA est maintenant complet, et prêt à être intégré dans la solution complète.

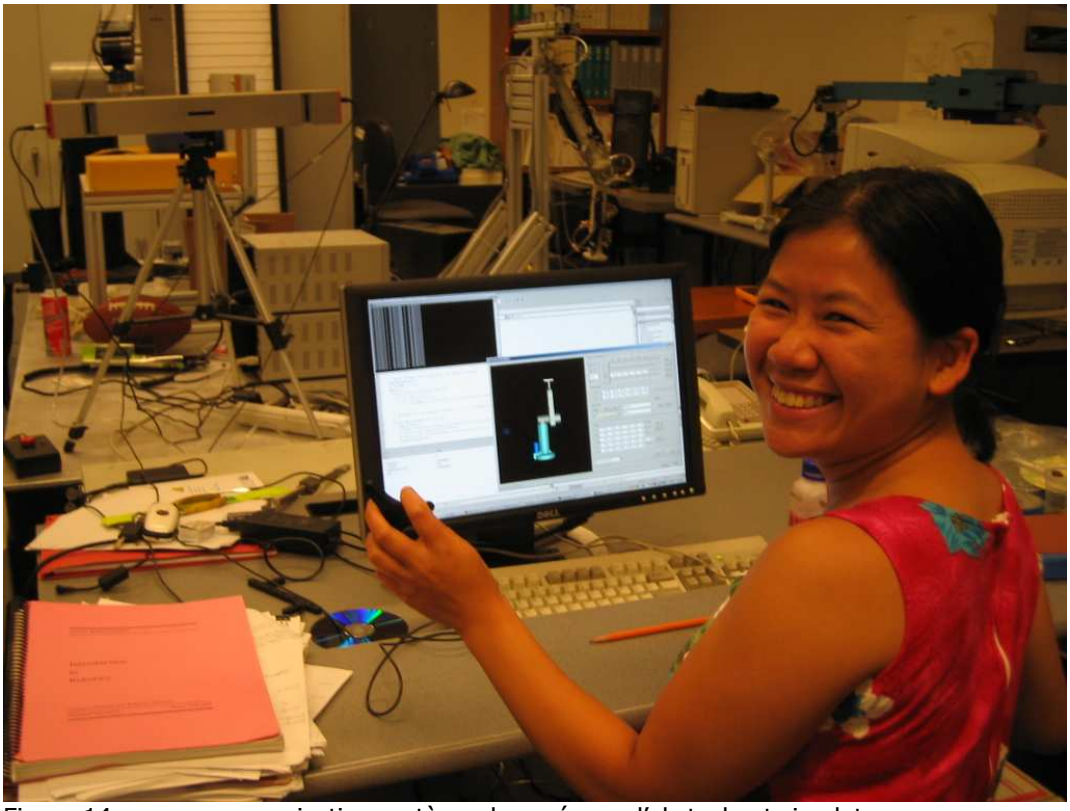

Figure 14: communication système de repérage d'obstacle et simulateur

3 Résultats obtenus

3.1 Projet global

La solution en elle-même comporte trois projets principaux (voir figure 14 et 15):

- un thread qui gère la détection de collision et envoie la position de l'obstacle au PUMA, via une connexion UDP
- un thread gérant l'interface graphique du simulateur. Celui-ci doit à la fois récupérer les données sur la configuration du robot en temps réel et renvoyer au système les ordres de l'utilisateur
- la boucle principale de contrôle qui a intervalle régulier : récupère les ordres de l'interface utilisateur, récupère les locations de l'obstacle et la configuration du robot, envoie le couple de contrôle principal que nous venons d'implémenter, puis borne ce couple avant de l'appliquer.

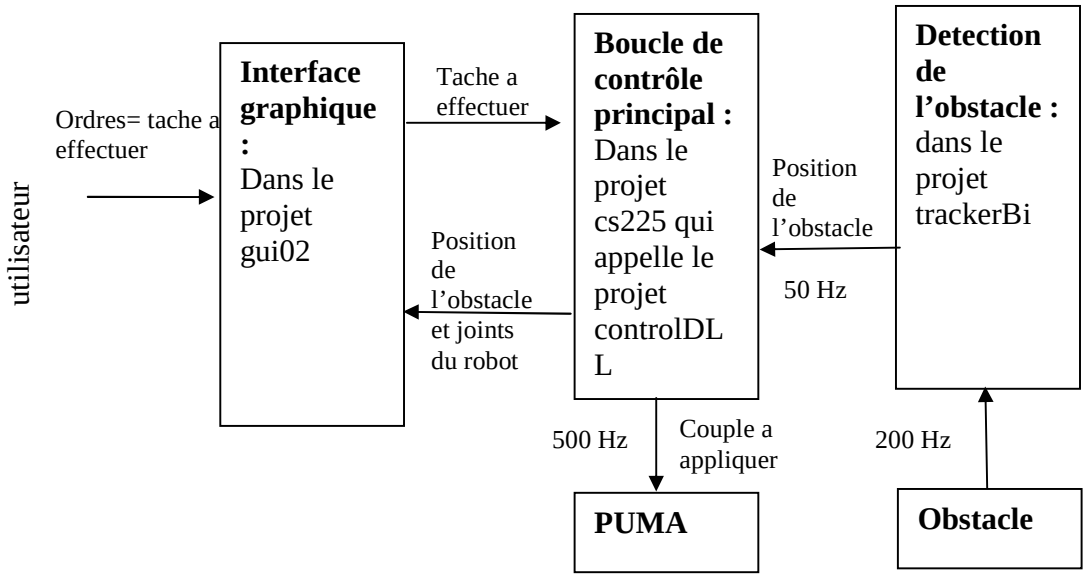

Figure 15: Structure simplifiée de la solution

3.2 Performances obtenues

Le simulateur obtenu permet à l'avant-bras d'éviter un obstacle mobile en temps réel. Les contraintes temporelles sont respectées et permettent d'effectuer la boucle d'asservissement à une fréquence de 500 fois par seconde.

Pour satisfaire à cette contrainte, comme la communication par UDP est lente, et surtout dans l'hypothèse que celle-ci doit se déplacer a vitesse inférieure au PUMA pour que celui-ci puisse bien l'éviter, la réactualisation de la position de l'obstacle n'est faite que toutes les 10 boucles d'asservissement.
Ce choix nous permet d'obtenir un mouvement fluide non hache et en temps réel.

3.3 Suites du projet

Le simulateur est certes opérationnel, mais des améliorations restent à faire. La première étape serait de passer au vrai robot. Pour cela, il faudrait intégrer le projet dans l'environnement qnx. Ceci réutilise les mêmes fichiers. La plus grande modification à apporter serait dans le Makefile, pour lier les différents projets entre eux.
Une autre suite du projet serait de prendre en compte la géométrie et le volume du bras, et non plus seulement l'assimiler à un segment. Ceci requiert d'avoir des données plus précises sur la géométrie du robot et de regarder plus en détail les calculs qui prendront bien plus de temps a être effectues.

Pour aller plus loin, la prise en compte de l'environnement extérieur implique aussi toutes les études de contact du robot avec le monde extérieur grâce a l'intégration de capteur d'effort, d'interface haptique a retour d'effort, de peau artificielle ou de capteurs de pression.

Conclusion

Le produit final permet sur simulateur à un bras articulé d'éviter un obstacle. Il m'a ainsi permis d'étudier les méthodes de gestion d'obstacle, surtout la méthode d'évitement réactif.

Ce projet m'a plus généralement appris à commander un robot assez complexe, en introduisant des outils d'algèbre linéaire et de l'informatique. Il permet de voir une commande de robot directement par les couples appliquées aux moteurs, et de résoudre les différentes taches classiques via l'introduction de l'approche cartésienne et de potentiel artificiel de force. Il a su mobiliser des compétences informatiques pour lancer des threads, pour établir une communication réseau et pour gérer une solution très complexe de 6 projets différents avec des variables publiques et privées. Il m'a aussi appris à commander un robot pour atteindre une position précise ou l'éviter, et voir les contraintes qui sont impliquées tels les frottements, les instabilités et forces d'inertie.

Plus globalement, le stage a aussi été une première approche au milieu de la recherche et au milieu de la robotique. Il m'a permis d'appréhender les techniques de travail d'un laboratoire, mais surtout m'a mise en contact avec des robots et les différentes problématiques de la robotique, que cela concerne la robotique industrielle, la biomimétique, la robotique télé opérationnelle, les robots bipèdes ou la robotique humanoïde.

Références

- Introduction to robotics, by Oussama Khatib and Krasimir Kolarov, Stanford University, winter 2006
- Advanced Robotics, by Oussama Khatib, Stanford University, spring 2007
- Oussama Khatib, Oliver Brock, K.C. Chang, Diego Ruspini, Luis Sentis, Sriram Viji. Human-Centered Robotics and Interactive Haptic Simulation. *International Journal of Robotics Research* 23(2):167-178, 2004.
- http://ai.stanford.edu/~lsentis/

Remerciements a:
Oussama Khatib , Vincent Padois, Francois Conti, Jaehung Park, Jinsung Kwong, Emel Dermican

Annexe 1 : Le système Altracsys

1/ trackerBi.h

```
#ifndef __REFAIRE_H__
#define __REFAIRE_H__

#include <stdio.h>
#include <windows.h>                                    // To use time
#include <math.h>
#include "apiTracking.hpp"                      // easyTrack library
//#include "stdafx.h"

#define ETK_MARKER1   0x000f
#define ETK_MARKER2   0x00f0
#define NB_FRAMES 50

static double mDataMarker[2][3];
static double limsup[3];
static double liminf[3];

class TrackerBi
{
public:
        TrackerBi();
        ~TrackerBi();
        void traiterInformation(void * marqueur);
        void init();
        void start();
        void process(void * marqueur);
        void close();


private:
        static void monMarkerCallback(glbDevice *pDevice, unsigned uMarkerID, double *adMatrix33, double
*adVector3, double *pdError, unsigned uIsExtended);
        //! Handler on easyTrack library
        glbHandle myhandle;
        //! easyTrack device
        glbDevice *pDevice;
        glbDevice *pDevice2;

};

//---------------------------------------------------------------------------
//extern TrackerBi *pETK;
//---------------------------------------------------------------------------
#endif
```

2/ principal.cpp : programme qui gérer tout le système de repérage de l'obstacle

```
#include "trackerBi.h"
#include "trackerBiSocket.h"
#include "stdafx.h"
#include <iostream>
using namespace std;
```

```
int _tmain(int argc, _TCHAR* argv[])
{
		int compteur=0;
		printf("Hello \n");
		TrackerBi *pointeur= new TrackerBi();
		pointeur->init();
		pointeur->start();
		//network initialization
		initialize_UDP_connection();

		while(1){
			pointeur->process((void*)ETK_MARKER1);
			Sleep(5);
			//send information to PUMA
			netWrite(pointeur->tableMarker[0][0],pointeur->tableMarker[0][1],pointeur->tableMarker[0][2]);
		}

		pointeur->close();
//		network shutdown
		end_sockets();
		return 0;
}
```

3/ trackerBi.cpp

```
#include <iostream>
#include "trackerBi.h"
//#include "stdafx.h"
using namespace std;

TrackerBi::TrackerBi()
{
		mDataMarker[0][0] = 0.0;
		mDataMarker[0][1] = 0.0;
		mDataMarker[0][2] = 0.0;
		mDataMarker[1][0] = 1.0;
		mDataMarker[1][1] = 1.0;
		mDataMarker[1][2] = 1.0;
		liminf[0]= -350;
		liminf[1]= -342;
		liminf[2]= 0;
		limsup[0]= 350;
		limsup[1]= 500;
		limsup[2]= 2600;
}

TrackerBi::~TrackerBi()
{
}


void TrackerBi::traiterInformation(void * marqueur)
{
		if (marqueur == (void *) ETK_MARKER1){
			tableMarker[0][0]=5*(mDataMarker[0][2]-1000)/(limsup[2]-liminf[2]); // axis are not the same on
puma and altracsys
			tableMarker[0][1]=5*mDataMarker[0][0]/(limsup[0]-liminf[0]);
			tableMarker[0][2]=5*mDataMarker[0][1]/(limsup[1]-liminf[1]);
//			printf (" Position1 (%lf , %lf )\n", tableMarker[0][0],  tableMarker[0][1]);
		}
		else if (marqueur == (void*) ETK_MARKER2)
			for (int i=0 ; i<3 ; i++){
				tableMarker [1][i]= mDataMarker[1][i];
```

```
                }
}

void TrackerBi::monMarkerCallback(glbDevice *pDevice, unsigned uMarkerID, double *adMatrix33, double *adVector3,
double *pdError, unsigned uIsExtended){
        if (adVector3) // If LED is not valid, vector is NULL
        {
            if (uMarkerID == 0){
                mDataMarker[0][0] = adVector3[0];
                mDataMarker[0][1] = adVector3[1];
                mDataMarker[0][2] = adVector3[2];
            }
            else if (uMarkerID == 1){
                mDataMarker[1][0] = adVector3[0];
                mDataMarker[1][1] = adVector3[1];
                mDataMarker[1][2] = adVector3[2];
            }

        }
}

void TrackerBi::init()
{
        myhandle =    apiInitialize (        NULL );    // Open library
        apiSetMarkerCallback    (myhandle, monMarkerCallback);          // set myCallback
        pDevice = apiGetDevice  (myhandle, ETK_MASTER      , 0);  //get the device
}

void TrackerBi::start(){
        apiSetOption (pDevice, "LEDMASK", (void*)ETK_MARKER1);
        apiStartAll(myhandle);
}

void TrackerBi::process (void * marqueur)
{
        apiSetOption (pDevice, "LEDMASK", marqueur);
        apiProcessAll(myhandle,NB_FRAMES);  //process next NB_FRAMES data frames
        traiterInformation(marqueur);
}

void TrackerBi::close()
{
        apiStop(pDevice);
        apiClose(myhandle);
}
```

4/ trackerBiSocket.cpp : partie servant à envoyer la position de l'obstacle au PUMA

```
/////// network functions for communication
#include <windows.h>
#include "trackerBiSocket.h"
#include <iostream>
using namespace std;

// ======================================== NetWork Variables
==========================

#define MY_IP   "127.0.0.1"
#define MY_PORT 5000
#define MY_IP   "127.0.0.1"
#define PUMA_PORT 5001
#define PUMA_IP "127.0.0.1"
```

```
#define MAXDATASIZE 300
// WSAData for socket initialization
WSADATA tracker_WSAData;
SOCKET tracker_pumaSock;
SOCKADDR_IN tracker_puma_addr;
SOCKADDR_IN tracker_my_addr;

////////////////////////////////////////// functions to connect UDP ports
void  initialize_UDP_connection()
{
		// Initialize Winsock
		if (WSAStartup (MAKEWORD(1,1), &tracker_WSAData) != 0)   {
			cout << "Initialization failed ! Error: " << WSAGetLastError () << endl;
			exit (1);
		}
		cout<<"now starting to initialize the connection"<<endl;

		// Create socket
		if ((tracker_pumaSock = socket(AF_INET, SOCK_DGRAM, 0)) == INVALID_SOCKET) {
			cout << "Creation of sending socket failed !  Error: " << WSAGetLastError () << endl;
			exit(1);
		}

		tracker_my_addr.sin_family = AF_INET;          // host byte order
		tracker_my_addr.sin_port = htons(MY_PORT);      // short, network byte order
		tracker_my_addr.sin_addr.s_addr = inet_addr(MY_IP); //INADDR_ANY
		memset(&(tracker_my_addr.sin_zero), '\0', 8);  // zero the rest of the struct

		if (bind(tracker_pumaSock, (struct sockaddr FAR *)&tracker_my_addr, sizeof(tracker_my_addr))
==SOCKET_ERROR){
			cout << "Binding sending socket failed ! Error: " << WSAGetLastError () << endl;
			closesocket (tracker_pumaSock);
			exit(1);
		}


		tracker_puma_addr.sin_family = AF_INET;          // host byte order
		tracker_puma_addr.sin_port = htons(PUMA_PORT);   // short, network byte order
		tracker_puma_addr.sin_addr.s_addr = inet_addr(PUMA_IP);
		memset(&(tracker_puma_addr.sin_zero), '\0', 8);  // zero the rest of the struct

		if (int iResult=connect(tracker_pumaSock, (struct sockaddr FAR *)&tracker_puma_addr,
sizeof(tracker_puma_addr))==SOCKET_ERROR){
			cout << "Binding sending socket failed! Error: " << WSAGetLastError () << endl;
			closesocket (tracker_pumaSock);
			exit(1);
		}

		cout << "Sockets connected ! " << endl;
//		non_blocking_recv_set();
}


void  end_sockets(void)
{
		closesocket(tracker_pumaSock);
		if (WSACleanup() != 0)   {
				cout << "Cleaning up WSA failed ! Error: " << WSAGetLastError () << endl;
				exit (1);
		}

}

void  netStart() {

		static char buf[MAXDATASIZE];
```

```
		sprintf(buf,"zero");

	if ((sendto(tracker_pumaSock, buf, MAXDATASIZE, 0,
			(struct sockaddr *)&tracker_puma_addr, sizeof(struct sockaddr)))==-1){
	perror("sendto");
	printf("%d\n",errno);
	}
		cout<<"sent starting command to PUMA"<<endl;

}

void  netStop() {

		static char buf[MAXDATASIZE];

		sprintf(buf,"end");

	if ((sendto(tracker_pumaSock, buf, MAXDATASIZE, 0,
			(struct sockaddr *)&tracker_puma_addr, sizeof(struct sockaddr)))==-1){
	perror("sendto");
	printf("%d\n",errno);
	}

		cout<<"sent stop command to PUMA"<<endl;

}

int  netWrite(double sendData1, double sendData2,double sendData3){
		static char buf[MAXDATASIZE];

		/*buf[0]=sendData1;
		buf[1]=sendData2;*/
		sprintf(buf,"%lf a %lf a %lf\n", sendData1,sendData2,sendData3);
   if ((sendto(tracker_pumaSock, buf, MAXDATASIZE, 0,
			(struct sockaddr *)&tracker_puma_addr, sizeof(struct sockaddr)))==-1){
     perror("sendto");
     printf("%d \n",errno);
   }
		else printf(buf);

return 0;
}

int  netRead() {
		char recvbuf[300];
   int recvbuflen = 300;
		int iResult = recv(tracker_pumaSock, recvbuf, recvbuflen, 0);

		if ( iResult > 0 ){
			printf( recvbuf);
			printf("\n");
		}
		else if ( iResult == 0 )
			printf("Connection closed\n");
		else
			printf("recv failed: %d\n", WSAGetLastError());
		return 0;
}
```

Annexe 2 : GlobalVariables& gv : les variables du PUMA, ses descripteurs

State variables

gv.dof : Degrees of freedom
gv.curTime : Current simulator time
gv.tau : Vector of joint torques [in newton-meters]
gv.q : Vector of current joint space positions [rad]
gv.dq : Vector of current joint space velocities [rad / sec]
gv.kp : Vector of position gains (kp)
gv.kv : Vector of velocity gains (kv)
gv.qd : Vector of desired joint positions [rad]
gv.dqd : Vector of desired joint velocities [rad / sec]
gv.ddqd : Vector of desired joint accelerations [rad / sec2]
gv.x : Vector of current operational space positions
gv.dx : Vector of current operational space velocities
gv.xd : Vector of desired operational space positions
gv.dxd : Vector of desired operational space velocities
gv.ddxd : Vector of desired operational space accelerations
gv.elbow : Desired elbow configuration for track control mode
gv.T : Linear transformation for end-effector position/orientation
gv.Td : Linear transformation for desired end-effector position/orientation

Kinematics & Dynamics Variables

gv.J : Jacobian
gv.Jtranspose : Jacobian transpose
gv.A : Mass matrix. Also called "M" in some robotics classes.
gv.B : Centrifugal/coriolis vector
gv.G : Gravity vector
gv.Lambda : Mass matrix in operational space
gv.mu : Centrifugal/coriolis vector in operational space
gv.p : Gravity vector in operational space
gv.singularities : Bitmap of singularities
gv.E : Matrix converting linear/angular velocity to configuration parameters
gv.Einverse : Inverse of g E matrix

Limit Variables

gv.qmin : Minimum joint positions allowance [rad]
gv.qmax : Maximum joint positions allowance [rad]
gv.dqmax : Maximum joint velocities allowance [rad / sec]
gv.ddqmax : Maximum joint accelerations allowance [rad / sec2]
gv.taumax : Maximum joint torques allowance [newton-meter]
gv.xmin : Vector of minimum operational-space coordinates
gv.xmax : Vector of maximum operational-space coordinates
gv.dxmax : Maximum operational space velocity allowance (scalar)
gv.ddxmax : Maximum operational space acceleration allowance (scalar)
gv.wmax : Maximum angular velocity in operational space (scalar)
Potential-field Variables
gv.jlimit : (simulator only) Joint Limits potential field flag
gv.q0 : Vector of minimum distances to apply joint limit potential fields
gv.kj : Vector of gains for the joint limit potential field
gv.rho0 : Minimum distance to apply potential field controller
gv.eta : Gain for potential field controller
gv.sbound : Boundary of singularity [rad]
gv.line : Structure holding a line for the Line Trajector controller
gv.numObstacles : Number of obstacles, for the potential field controller
gv.obstacles : Array of obstacles, for the potential field controller

Annexe2 : Control.cpp

```
// controlDLL.cpp : Defines the entry point for the DLL application.
//
//#ifdef WIN32
#include "stdafx.h"
BOOL APIENTRY DllMain( HANDLE hModule,
                       DWORD  ul_reason_for_call,
                       LPVOID lpReserved
                                                    )
{
   return TRUE;
}
#include <stdio.h>

//#else //#ifdef WIN32
#include "servo.h"

//#endif //#ifdef WIN32

#include "param.h"
#include "control.h"
#include "PrVector.h"
#include "PrMatrix.h"
#include "Utils.h" // misc. utility functions, such as toRad, toDeg, etc.
#include <math.h>
#include <algorithm>

using std::min;
using std::max;




static PrVector splineParam0;  // params used for track splines
static PrVector splineParam2;
static PrVector splineParam3;
static Float splineStartTime, splineDuration;
Float timeF;


// ***********************************************************************
// Initialization functions
// ***********************************************************************

void InitControl(GlobalVariables& gv)
{
  // This code runs before the first servo loop
}

void PreprocessControl(GlobalVariables& gv)
{
                PrMatrix3 rotationEffector;
                PrMatrix3 rotationElbow, rotationTemp;
                PrVector3 dElbowWRist; dElbowWRist.values(-L3,0,0);//coordinates of vector Elbow-Wrist in reference 3
                gv.T.rotation().rotationMatrix(rotationEffector);
                gv.elbowPos.values(L2,L1,0);
                gv.wristPos.values(0,0,-L6);   //coordonnees dans le repere du end-effector, repere6
                gv.wristPos= gv.x.subvector(0,3)+ rotationEffector*gv.wristPos;//position of the wrist in reference 0
                rotationElbow.setRotation(PrVector3(0,0,1),gv.q[0]); //
                gv.elbowPos= rotationEffector*gv.elbowPos;  //position of the test in reference 0
```

```
}


void PostprocessControl(GlobalVariables& gv)
{
		PrVector joint_limit =PrVector(gv.dof);  //torques
		PrVector joint_line =PrVector(gv.dof);
		PrVector joint_obstacles =PrVector(gv.dof);
		PrVector deltaQ =PrVector (gv.dof);	//difference avoiding position and current position
		PrVector3 deltaX = PrVector3();
		PrVector3 force = PrVector3();
		static const float EPSILON = .1;
		const float MAX_DIST = 0.3;

		if (gv.jlimit == true) {   //soft joint limit
			for (int i=0; i<gv.dof; i++){
				deltaQ[i]= abs(gv.q[i]-gv.qmax[i]);
				if (deltaQ[i]<EPSILON) //deltaQ[i]=EPSILON;
					joint_limit[i] = (gv.kj[i]/2) * pow( (1/deltaQ[i] -1/abs(gv.q0[i]) ), 2);
				if (joint_limit[i]> gv.taumax[i]) joint_limit[i] = gv.taumax[i];
			else if (joint_limit[i] < - gv.taumax[i]) joint_limit[i] = -gv.taumax [i];
				else joint_limit[i]=0;
			}
		}

		if (gv.line.enabled()){ // move towards a line with a potential field
			//Line
			PrMatrix3 i3;
			i3.identity();
			PrVector vect_line= gv.line.unitVec();
			PrVector3 vectorAM = (i3-vect_line.multiplyTransposed(vect_line))*(gv.x.subvector(0,3) -
gv.line.point(0));
			for (int i=0; i < 2; i++)
			{
				vectorAM[i] = -100000*gv.kj[i]*vectorAM[i];
			}
			vectorAM[2] = -1000*vectorAM[2];
			joint_line = (gv.J.submatrix(0,0,3,3).transpose())*(vectorAM);
		}

		if (gv.numObstacles >0 && gv.calibration>=2){
			for (int i=0; i<3;i++)
			joint_obstacles[i] = ObstacleAvoidance(gv,(PrVector3)gv.x.subvector(0,3),gv.wristPos)[i]+
ObstacleAvoidance(gv,gv.elbowPos,gv.wristPos)[i];
			if (joint_obstacles[i]> gv.taumax[i]) joint_obstacles[i] = gv.taumax[i]/2;
			else if (joint_obstacles[i] < - gv.taumax[i]) joint_obstacles[i] = -gv.taumax [i]/2;
		}

		if (gv.nullSpaceTask.row() ==gv.dof)
			gv.tau += joint_obstacles+ joint_limit +joint_line;


}

PrVector3 ProximumPunctum(PrVector3 point1, PrVector3 point2, PrVector3 point3) //nearest point of point3 on line
(point1,point2)
{
		PrVector3 punctum,vecteurDroite;
		Float norme;
		vecteurDroite = (point2 -point1).normalize();
		norme = vecteurDroite.dot(point3 -point1);
		if (norme <0) punctum=point1;
		else if(norme> (point2-point1).magnitude()) punctum =point2;
		else punctum =point1 + norme* vecteurDroite;
		return punctum;
}
```

```
PrVector damping( GlobalVariables & gv)
{
		PrVector F = PrVector(6);
		PrVector x_d =PrVector(gv.x.size());
		x_d=gv.x;
		F = -gv.kp*(gv.Einverse*(gv.x-x_d))-gv.kv*(gv.dx);
		return F;
}

PrVector ObstacleAvoidance(GlobalVariables& gv, PrVector3 &point1,PrVector3 &point2){
		PrVector joint_obstacle= PrVector(gv.dof);
		PrVector3 deltaX = PrVector3();
		PrVector3 nearestPoint =PrVector3();
		PrVector force = PrVector(6);
		PrVector forceProjetee = PrVector(6);
		PrVector3 moment =PrVector3(); // moment des forces en nearestPoint
		Float gradient =0;
		const float MAX_DIST = 0.3;

		for (int j= 0; j < gv.numObstacles/*MAX_OBSTACLES*/;j++)
		{
			nearestPoint = ProximumPunctum(point1,point2,gv.obstacles[j].coord);
			deltaX = -(nearestPoint -gv.obstacles[j].coord);
			if (deltaX.magnitude()< MAX_DIST){
				gradient = 300*gv.eta *(1/deltaX.magnitude() -
1/gv.obstacles[j].radius)/pow(deltaX.magnitude(),3);
				for (int i=0; i<3; i++){
				//		repulsive potential field = gv.eta/10 * pow(1/deltaX[i] -
1/(gv.obstacles[i].radius),2);
				force[i] += gradient * deltaX[i];
				}
				moment = (force.subvector(3,3)).cross(nearestPoint - (PrVector3)gv.x.subvector(0,3));
				for (int i=0; i<3;i++) force [3+i] =moment[i];
				forceProjetee = (gv.nullSpaceTask.transpose()*force);
			}
			else{
				forceProjetee = gv.nullSpaceTask.transpose()*damping(gv);
			}
			joint_obstacle += (gv.kj) *((gv.Jtranspose)*forceProjetee);
		}
		return joint_obstacle;

}



void initFloatControl(GlobalVariables& gv)
{
   // Control Initialization Code Here
}

void initOpenControl(GlobalVariables& gv)
{
		// Control Initialization Code Here
}

void initNjholdControl(GlobalVariables& gv)
{
		//set desired position to original
		gv.qd = gv.q;
		//set desired velocity to zero
		gv.dqd.zero();
}

void initJholdControl(GlobalVariables& gv)
```

```
{
		//set desired position to original
		gv.qd = gv.q;
		//set desired velocity to zero
		gv.dqd.zero();
}

void initNjmoveControl(GlobalVariables& gv)
{
		//set desired position
		//gv.qd = gv.qd;
		//set desired velocity to zero
		gv.dqd.zero();
}

void initJmoveControl(GlobalVariables& gv)
{
		//set desired position
		//gv.qd = gv.qd;
		//set desired velocity to zero
		gv.dqd.zero();
}


void initNjgotoControl(GlobalVariables& gv)
{/*
		// set desired position
		bool flag = false;
		for(int i = 0; i < gv.qd.size(); i++)
		{
			if( (gv.qd[i] > gv.qmax[i]) || (gv.qd[i] < gv.qmin[i]))
				flag = true;
		}

		if(flag)
			gv.qd = gv.q;

//		else gv.qd = gv.qd

		gv.dqd.zero();*/
}

void initJgotoControl(GlobalVariables& gv)
{/*
		// set desired position
		bool flag = false;
		for(int i = 0; i < gv.qd.size(); i++)
		{
			if( (gv.qd[i] > gv.qmax[i]) || (gv.qd[i] < gv.qmin[i]))
				flag = true;
		}

		if(flag)
			gv.qd = gv.q;
		//else gv.qd = gv.qd

		gv.dqd.zero();*/
}

void initNjtrackControl(GlobalVariables& gv)
{
			// Control Initialization Code Here
		splineStartTime=gv.curTime;
		timeF = 0;
		Float t_min;
		PrVector q0= gv.q;
		PrVector dq0=gv.dq;
```

```
		for(int i=0; i<gv.dof;i++){
			t_min=sqrt((6/(gv.ddqmax[i]))*abs(gv.qd[i]-q0[i]));
			if (t_min > timeF) timeF=t_min;
			t_min=(3/(2*gv.dqmax[i]))*abs(gv.qd[i]-q0[i]);
			if (t_min > timeF) timeF=t_min;
		}
		splineParam0=q0;
		splineParam2=(3/(timeF*timeF))*(gv.qd-q0);
		splineParam3= (-2/(timeF*timeF*timeF))*(gv.qd-q0);

}

void initJtrackControl(GlobalVariables& gv)
{
				// Control Initialization Code Here
		splineStartTime=gv.curTime;
		timeF = 0;
		Float t_min;
		PrVector q0= gv.q;
		PrVector dq0=gv.dq;

		for(int i=0; i<gv.dof;i++){
			t_min=sqrt((6/(gv.ddqmax[i]))*abs(gv.qd[i]-q0[i]));
			if (t_min > timeF) timeF=t_min;
			t_min=(3/(2*gv.dqmax[i]))*abs(gv.qd[i]-q0[i]);
			if (t_min > timeF) timeF=t_min;
		}
		splineParam0=q0;
		splineParam2=(3/(timeF*timeF))*(gv.qd-q0);
		splineParam3= (-2/(timeF*timeF*timeF))*(gv.qd-q0);

}
void FindCubicSpline(GlobalVariables& gv)
{
			// Control Initialization Code Here
		splineStartTime=gv.curTime;
		timeF = 0;
		Float t_min;
		PrVector q0= gv.q;
		PrVector dq0=gv.dq;

		for(int i=0; i<gv.dof;i++){
			t_min=sqrt((6/(gv.ddqmax[i]))*abs(gv.qd[i]-q0[i]));
			if (t_min > timeF) timeF=t_min;
			t_min=(3/(2*gv.dqmax[i]))*abs(gv.qd[i]-q0[i]);
			if (t_min > timeF) timeF=t_min;
		}
		splineParam0=q0;
		splineParam2=(3/(timeF*timeF))*(gv.qd-q0);
		splineParam3= (-2/(timeF*timeF*timeF))*(gv.qd-q0);

}

void initNxtrackControl(GlobalVariables& gv)
{
		inv_kin(gv.Td.translation(),gv.Td.rotation().matrix(),gv.elbow,gv.qd,gv);
		initNjtrackControl(gv);
}

float asinb( float x )
{
		if (abs(x) <= 1)
			return asin(x);
		else if (x > 1)
			return asin(1.);
		else
```

```
                return asin(-1.);
}

float acosb( float x )
{
        if (abs(x) <= 1)
                return acos(x);
        else if (x > 1)
                return acos(1.);
        else
                return acos(-1.);
}

void OpDynamics( const PrVector& force, GlobalVariables& gv)
{
        // Operational Space Dynamic control law including gravity compensation
        PrMatrix J_r;
        //Jr.setSize(6,6,true);

        PrMatrix S;
        PrMatrix Lambda_r;
        PrMatrix Identity;
        PrVector6 escapeTorque;
        PrMatrix Ainverse= PrMatrix(gv.A.column(),gv.A.row());


        //not sure what size should be here
        Identity.setSize(6,6);
        Identity.identity();

        if( gv.singularities == 0)
                gv.tau = gv.Jtranspose*(gv.Lambda*force+gv.mu+gv.p);
        else {
                gv.A.inverse(Ainverse);

                //getNonSingularSelection does not throw up an error
                getNonsingularSelection(S,gv);
                J_r = S*gv.J;
                Lambda_r = (J_r*Ainverse*J_r.transpose());

                Lambda_r.inverse(Lambda_r);

                //getEscapeTorque gives no error
                getEscapeTorque(force,escapeTorque,gv);

                gv.tau = J_r.transpose()*Lambda_r*S*force + (Identity-
J_r.transpose()*Lambda_r*J_r*Ainverse)*escapeTorque + gv.G;

        }
}

void OpNonDynamics( const PrVector& force, GlobalVariables& gv)
{
                gv.tau = gv.Jtranspose*force + gv.G;
}

bool inv_kin( const PrVector3& POS, const PrMatrix3& ROT, int elbow,PrVector& qOut, GlobalVariables& gv )
{
        PrVector3 p06;
        PrVector3 pE6;
        PrVector3 x03;
        PrVector3 z03;
        PrVector3 x05;
        PrVector3 y04;
        PrVector3 z04;
        PrVector3 x0E;
        PrVector3 y0E;
```

```
		PrVector3 z0E;
		PrVector6 temp;

		float deltaTheta1;
		float deltaTheta2;
		float lPrime;
		float xWrist;
		float yWrist;
		float zWrist;

		const float EPSILON = .001;

		qOut[0] = 0;
		qOut[1] = 0;
		qOut[2] = 0;
		qOut[3] = 0;
		qOut[4] = 0;
		qOut[5] = 0;

		temp[0] = qOut[0];
		temp[1] = qOut[1];
		temp[2] = qOut[2];
		temp[3] = qOut[3];
		temp[4] = qOut[4];
		temp[5] = qOut[5];

		pE6[0] = 0;	// Initialization
		pE6[1] = 0;
		pE6[2] = -L6;

		p06 = POS + ROT*pE6;
		ROT.getColumn(1,x0E);
		ROT.getColumn(1,y0E);
		ROT.getColumn(2,z0E);

		xWrist = p06[0];
		yWrist = p06[1];
		zWrist = p06[2];

		lPrime = sqrt(pow(xWrist,2) + pow(yWrist,2) + pow(zWrist,2) - pow(L1,2));

// first find teta 1
		deltaTheta1 = asinb((-L1)/sqrt(pow(xWrist,2)+pow(yWrist,2)));
		if ((elbow & 0x01) == 0)
			qOut[0] = atan2(yWrist,xWrist) + deltaTheta1;
		else
		{
			if (atan2(yWrist,xWrist) > 0)
				qOut[0] = atan2(yWrist,xWrist) - M_PI - deltaTheta1;
			else
				qOut[0] = atan2(yWrist,xWrist) + M_PI - deltaTheta1;
		}

		temp[0] = qOut[0];

// find teta 3
		if ((((elbow&0x01) == 0) && ((elbow&0x02) == 0)) || (((elbow&0x01) == 0x01) && ((elbow&0x02) ==
0x02)))
		{
			qOut[2] = M_PI/2-acosb(1/(2*L2*L3)*(pow(xWrist,2)+pow(yWrist,2)+pow(zWrist,2)-
(L1*L1+L2*L2+L3*L3)));
		}
		else
		{
			qOut[2] = M_PI-(M_PI/2-acosb(1/(2*L2*L3)*(pow(xWrist,2)+pow(yWrist,2)+pow(zWrist,2)-
(L1*L1+L2*L2+L3*L3))));
		}
```

```
		temp[2] = qOut[2];

		// find teta 2
		if ((elbow & 0x01) == 0)
		{
			deltaTheta2 = asinb(L3*sin(M_PI/2 - qOut[2])/lPrime);
			qOut[1] = -asinb(zWrist/lPrime) + deltaTheta2;
		}
		else
		{
			deltaTheta2 = asinb(L3*sin(M_PI/2 - qOut[2])/lPrime);
			qOut[1] = -M_PI + asinb(zWrist/lPrime) + deltaTheta2;
		}

		temp[1] = qOut[1];

// find teta 5
		z04[0] = cos(qOut[0]) * sin(qOut[1] + qOut[2]);
		z04[1] = sin(qOut[0]) * sin(qOut[1] + qOut[2]);
		z04[2] = cos(qOut[1] + qOut[2]);
		qOut[4] = acosb(z04.dot(z0E));
		temp[4] = qOut[4];

// find teta 4
		x03[0] = cos(qOut[0]) * cos(qOut[1] + qOut[2]);
		x03[1] = sin(qOut[0]) * cos(qOut[1] + qOut[2]);
		x03[2] = -sin(qOut[1] + qOut[2]);

		z03[0] = -sin(qOut[0]);
		z03[1] = cos(qOut[0]);
		z03[2] = 0;

		if (abs(qOut[4]) < EPSILON)
			qOut[3] = 0;
		else
		{
			qOut[3] = acosb(z03.dot(z04.cross(z0E).normalize()));
		}

		temp[3] = qOut[3];

		if ( acosb(x03.dot(z04.cross(z0E).normalize())) < M_PI/2 )
		{
			qOut[4] = -qOut[4];
			qOut[3] = M_PI - qOut[3];
		}

		temp[3] = qOut[3];
		temp[4] = qOut[4];

		if ((elbow & 0x04) == 0x04)
		{
			qOut[3] = qOut[3] - M_PI;
			qOut[4] = -qOut[4];
		}

		temp[3] = qOut[3];
		temp[4] = qOut[4];
		temp[5] = qOut[5];

//find teta 6

		y04[0] = -cos(qOut[0]) * cos(qOut[1] + qOut[2]) * sin(qOut[3]) - sin(qOut[0]) * cos(qOut[3]);
   y04[1] = -sin(qOut[0]) * cos(qOut[1] + qOut[2]) * sin(qOut[3]) + cos(qOut[0]) * cos(qOut[3]);
   y04[2] = (sin(qOut[1] + qOut[2])) * sin(qOut[3]);
```

```
		x05 = y04.cross(z0E);

		if (abs(qOut[4]) < EPSILON)
		{
			qOut[5] = acosb(y0E.dot(y04));
		}
		else
		{
			qOut[5] = acosb(y0E.dot(y04));
		}

		if (acosb(y0E.dot(x05)) < M_PI/2)
		{
			qOut[5] = -qOut[5];
		}
		temp[5] = qOut[5];
		return true;
}

void initXtrackControl(GlobalVariables& gv)
{
		inv_kin(gv.Td.translation(),gv.Td.rotation().matrix(),gv.elbow,gv.qd,gv);
		initJtrackControl(gv);

		// Control Initialization Code Here
}

void initNholdControl(GlobalVariables& gv)
{
		gv.xd=gv.x;
}

void initHoldControl(GlobalVariables& gv)
{
		gv.xd=gv.x;
		// Control Initialization Code Here
}

void initNgotoControl(GlobalVariables& gv)
{
		gv.dxd.zero();
}

void initGotoControl(GlobalVariables& gv)
{
		// Control Initialization Code Here
}

void initNtrackControl(GlobalVariables& gv)
{
		splineStartTime=gv.curTime;
		timeF = 0;
		Float t_min;
		PrVector x0= gv.x;
		PrVector dx0=gv.dx;

		for(int i=0; i<gv.dof;i++){
			t_min=sqrt((6/(gv.ddxmax))*abs(gv.xd[i]-x0[i]));
			if (t_min > timeF) timeF=t_min;
			t_min=(3/(2*gv.dxmax))*abs(gv.xd[i]-x0[i]);
			if (t_min > timeF) timeF=t_min;
		}
		splineParam0=x0;
		splineParam2=(3/(timeF*timeF))*(gv.xd-x0);
		splineParam3= (-2/(timeF*timeF*timeF))*(gv.xd-x0);
```

```
}

void initTrackControl(GlobalVariables& gv)
{
		// Control Initialization Code Here
}

void initPfmoveControl(GlobalVariables& gv)
{
		gv.xd[3]=1;
		gv.xd[4]=gv.xd[5]=gv.xd[6]=0;
		gv.nullSpaceTask.identity();
		gv.nullSpaceTask(3,0) =gv.nullSpaceTask(3,1) =gv.nullSpaceTask(3,2) = 1;
}

void initLineControl(GlobalVariables& gv)
{
		// Control Initialization Code Here
}

void initProj1Control(GlobalVariables& gv)
{
		// Control Initialization Code Here
}

void initProj2Control(GlobalVariables& gv)
{
		// Control Initialization Code Here
}

void initProj3Control(GlobalVariables& gv)
{
		// Control Initialization Code Here
}


// ********************************************************************
// Control laws
// ********************************************************************

void noControl(GlobalVariables& gv)
{
}

void floatControl(GlobalVariables& gv)
{
		gv.tau = gv.G;
}

void openControl(GlobalVariables& gv)
{
		if ( abs(gv.x[0]-0.5)<EPSILONOBSTACLE && abs(gv.x[1]-0)<EPSILONOBSTACLE && abs(gv.x[2]-
0)<EPSILONOBSTACLE ) {
			gv.trackerMatrix(0,0)=2*gv.obstacles[0].coord[0];
			gv.trackerMatrix(1,0)=2*gv.obstacles[0].coord[1];
			gv.trackerMatrix(2,0)=2*gv.obstacles[0].coord[2];
			gv.calibration=1;
		}
		else if ( abs(gv.x[0]-0)<EPSILONOBSTACLE && abs(gv.x[1]-0.5)<EPSILONOBSTACLE && abs(gv.x[2]-
0)<EPSILONOBSTACLE ) {
			gv.trackerMatrix(0,1)=2*gv.obstacles[0].coord[0];
			gv.trackerMatrix(1,1)=2*gv.obstacles[0].coord[1];
			gv.trackerMatrix(2,1)=2*gv.obstacles[0].coord[2];
		}
		else if  (abs(gv.x[0]-0)<EPSILONOBSTACLE && abs(gv.x[1]-0)<EPSILONOBSTACLE && abs(gv.x[2]-
0.5)<EPSILONOBSTACLE ) {
			gv.trackerMatrix(0,2)=2*gv.obstacles[0].coord[0];
```

```
                gv.trackerMatrix(1,2)=2*gv.obstacles[0].coord[1];
                gv.trackerMatrix(2,2)=2*gv.obstacles[0].coord[2];
                gv.calibration=3;
                gv.obstacles[0].radius=0.2;
                gv.trackerMatrix.inverse(gv.trackerMatrixInverse);
        }
}

void njholdControl(GlobalVariables& gv)
{
  gv.tau = -gv.kp*(gv.q-gv.qd) - gv.kv*(gv.dq-gv.dqd);
}

void jholdControl(GlobalVariables& gv)
{
 gv.tau = gv.A*(-gv.kp*(gv.q-gv.qd)-gv.kv*(gv.dq-gv.dqd)) + gv.B + gv.G;
}

void njmoveControl(GlobalVariables& gv)
{
  gv.tau = -gv.kp*(gv.q-gv.qd) - gv.kv*(gv.dq-gv.dqd)+gv.G;
}

void jmoveControl(GlobalVariables& gv)
{
  gv.tau = gv.A*(-gv.kp*(gv.q-gv.qd)-gv.kv*(gv.dq-gv.dqd)) + gv.B + gv.G;
}


void njgotoControl(GlobalVariables& gv)
{
        for( int i=0; i< gv.dof; i++ ){  //check joint limits
                if (gv.dq[i]   >= gv.dqmax[i]||gv.q[i]   >= gv.qmax[i]||gv.q[i]   <= gv.qmin[i]){
                        floatControl(gv);
                        return;
                }
        }

        for( int i=0; i< gv.dof; i++ ){                 //velocity saturation
                if( gv.kv[i]!=0 && gv.kp[i]/gv.kv[i]*(gv.qd[i]-gv.q[i])>gv.dqmax[i]){
                        gv.tau[i] = -gv.kv[i] *(gv.dq[i]-gv.dqmax[i])+gv.G[i];
                }
                else if ( gv.kv[i]!=0 && gv.kp[i]/gv.kv[i]*(gv.qd[i]-gv.q[i])<-gv.dqmax[i]){
                        gv.tau[i] = -gv.kv[i] *(gv.dq[i]+gv.dqmax[i])+gv.G[i];
                }
                else{
                        gv.tau[i] = -gv.kp[i]*(gv.q[i]-gv.qd[i])-(gv.kv[i] *gv.dq[i])+gv.G[i];
                }
        }

}

void jgotoControl(GlobalVariables& gv)
{

        for( int i=0; i< gv.dof; i++ ){   //joint limits
                if (gv.dq[i]   >= gv.dqmax[i]||gv.q[i]   >= gv.qmax[i]||gv.q[i]   <= gv.qmin[i]){
                        floatControl(gv);
                        return;
                }
        }

        PrVector deltaQ= -gv.A*(gv.kp*(gv.q-gv.qd));     //velocity saturation
        for( int i=0; i< gv.dof; i++ ){
                if(gv.kv[i]<0.0001) gv.kv[i]=0.0001;
                deltaQ[i]=deltaQ[i]/(gv.A*gv.kv)[i];
                if( deltaQ[i]>gv.dqmax[i]){
```

```
                        gv.tau[i] =-(gv.A*gv.kv)[i] *(gv.dq[i]-gv.dqmax[i]);
                }
                else if ( deltaQ[i]<-gv.dqmax[i]){
                        gv.tau[i] =-(gv.A*gv.kv)[i] *(gv.dq[i]+gv.dqmax[i]);
                }
                else{
                        gv.tau[i] =-(gv.A*gv.kv)[i] *(gv.dq[i]-deltaQ[i]);  //gv.tau[i] =gv.A*(-gv.kp*(gv.q-gv.qd)-
(gv.kv *gv.dq))[i]+gv.B[i]+gv.G[i];
                }
        }

}


void njtrackControl(GlobalVariables& gv)
{

        for( int i=0; i< gv.dof; i++ ){                  //joint limits
                if (gv.dq[i]   >= gv.dqmax[i]||gv.q[i]   >= gv.qmax[i]||gv.q[i]   <= gv.qmin[i]){
                        floatControl(gv);
                        return;
                }
        }
                Float t=gv.curTime-splineStartTime;         //cubic spline trajectory
                PrVector qd_t;
                PrVector dqd_t;
                        if( t< timeF){
                                qd_t = splineParam0+splineParam2*t*t+splineParam3*t*t*t;
                                dqd_t = 2*splineParam2*t+3*splineParam3*t*t;
                                gv.tau = -gv.kp*(gv.q-qd_t)-gv.kv*(gv.dq-dqd_t)+gv.G;
                                t=gv.curTime-splineStartTime;
                        }
                        if (t>timeF) {
                                gv.tau=-gv.kp*(gv.q-gv.qd)-gv.kv*gv.dq + gv.G;
                        }

}

void jtrackControl(GlobalVariables& gv)
{
                for( int i=0; i< gv.dof; i++ ){    //joint limits
                if (gv.dq[i]   >= gv.dqmax[i]||gv.q[i]   >= gv.qmax[i]||gv.q[i]   <= gv.qmin[i]){
                        floatControl(gv);
                        return;
                }
        }

                Float t=gv.curTime-splineStartTime;       //cubic spline trajectory
                PrVector qd_t;
                PrVector dqd_t;

                if(t< timeF){
                        qd_t = splineParam0+splineParam2*t*t+splineParam3*t*t*t;
                        dqd_t = 2*splineParam2*t+3*splineParam3*t*t;
                }
                if (t>timeF) {
                        qd_t=gv.qd;
                        dqd_t=gv.dqd;
                }

        PrVector deltaQ= dqd_t - gv.A*(gv.kp*(gv.q-qd_t));  //velocity saturation
        for( int i=0; i< gv.dof; i++ ){
                if(gv.kv[i]<0.0001) gv.kv[i]=0.0001;
                deltaQ[i]=deltaQ[i]/(gv.A*gv.kv)[i];
                if( deltaQ[i]>gv.dqmax[i]){
                        gv.tau[i] =-(gv.A*gv.kv)[i] *(gv.dq[i]-gv.dqmax[i]);
                }
```

```
                else if ( deltaQ[i]<-gv.dqmax[i]){
                        gv.tau[i] =-(gv.A*gv.kv)[i] *(gv.dq[i]+gv.dqmax[i]);
                }
                else{
                        gv.tau[i] =-(gv.A*gv.kv)[i] *(gv.dq[i]-deltaQ[i]);  //gv.tau[i] =gv.A*(-gv.kp*(gv.q-gv.qd)-
(gv.kv *gv.dq))[i]+gv.B[i]+gv.G[i];
                }
        }

}



void nxtrackControl(GlobalVariables& gv)
{
        njtrackControl(gv);
        }

void xtrackControl(GlobalVariables& gv)
{
  jtrackControl(gv);

}
void nholdControl(GlobalVariables& gv)
{
        PrVector F = PrVector(6);
        F = -gv.kp*(gv.Einverse*(gv.x-gv.xd))-gv.kv*(gv.dx);
        gv.tau=gv.Jtranspose*F+gv.G;

}

void holdControl(GlobalVariables& gv)
{
        PrVector F = PrVector(6);
        F = -gv.kp*(gv.Einverse*(gv.x-gv.xd))-gv.kv*(gv.dx);
        gv.tau=gv.Jtranspose*(gv.Lambda*F)+gv.B+gv.G;
}

void ngotoControl(GlobalVariables& gv)
{

        PrVector force = PrVector(6);
        PrVector deltaX = -gv.kp*(gv.Einverse*(gv.x-gv.xd));

        for(int i = 0; i<deltaX.size(); i++)
        {
              if (gv.kv[i] < 0.0001)
                        gv.kv[i] = 0.0001;
              deltaX[i] = (1/gv.kv[i])*(deltaX[i]);
        }

        //calculate velocity
        double temp_lin_mag = sqrt(pow(deltaX[0],2)+pow(deltaX[1],2)+pow(deltaX[2],2));
        double temp_ang_mag = sqrt(pow(deltaX[3],2)+pow(deltaX[4],2)+pow(deltaX[5],2));
        //velocity saturation
        if (temp_lin_mag > gv.dxmax)
        {
              for(int i=0; i<=2; i++)
                        deltaX[i] *= (gv.dxmax/temp_lin_mag);
        }
        if (temp_ang_mag > gv.wmax)
        {
              for(int i=3; i<=5; i++)
                        deltaX[i] *= (gv.wmax/temp_ang_mag);
        }

        force = -gv.kv*(gv.dx-deltaX);
```

```
		OpNonDynamics(force,gv);
		//gv.tau = gv.Jtranspose * (force) + gv.G;
}

void gotoControl(GlobalVariables& gv)   ////***DO NOT MODIFY IT WORKS !!!!!!!!*****
{
		PrVector force = PrVector(6);
		PrVector deltaX = -gv.kp*(gv.Einverse*(gv.x-gv.xd));

for(int i = 0; i<deltaX.size(); i++)
		{
			if (gv.kv[i] < 0.0001)
				gv.kv[i] = 0.0001;
			deltaX[i] = (1/gv.kv[i])*(deltaX[i]);
		}

   //calculate velocity
		double temp_lin_mag = sqrt(pow(deltaX[0],2)+pow(deltaX[1],2)+pow(deltaX[2],2));
		double temp_ang_mag = sqrt(pow(deltaX[3],2)+pow(deltaX[4],2)+pow(deltaX[5],2));
		//velocity saturation
		if (temp_lin_mag > gv.dxmax)
		{
			for(int i=0; i<=2; i++)
				deltaX[i] *= (gv.dxmax/temp_lin_mag);
		}
		if (temp_ang_mag > gv.wmax)
		{
			for(int i=3; i<=5; i++)
				deltaX[i] *= (gv.wmax/temp_ang_mag);
		}

		force = -gv.kv*(gv.dx-deltaX);

		OpDynamics(force,gv);

}

void ntrackControl(GlobalVariables& gv)
{
		Float t=gv.curTime-splineStartTime;
		PrVector xd_t;
		PrVector dxd_t;

		if( t< timeF){
			for(int i=0; i<3;i++){
				if (2*splineParam2[i]+6*splineParam3[i]*t > gv.ddxmax)  splineParam3[i]=(gv.ddxmax-2
*splineParam2[i])/t;
			}
			xd_t = splineParam0+splineParam2*t*t+splineParam3*t*t*t;
			dxd_t = gv.Einverse*(2*splineParam2*t+3*splineParam3*t*t);
		}
		if (t>timeF) {
			xd_t = gv.xd;
			dxd_t = gv.dxd;
		}

		PrVector force = PrVector(6);
		PrVector deltaX = dxd_t-gv.kp*(gv.Einverse*(gv.x-xd_t));
//		PrVector Fnonborne= -gv.kp*(gv.Einverse*(gv.x-xd_t))-gv.kv*(gv.dx-dxd_t);

			for(int i = 0; i<deltaX.size(); i++)
		{
			if (gv.kv[i] < 0.0001)
				gv.kv[i] = 0.0001;
			deltaX[i] = (1/gv.kv[i])*(deltaX[i]);
		}
```

```
   //calculate velocity
                double temp_lin_mag = sqrt(pow(deltaX[0],2)+pow(deltaX[1],2)+pow(deltaX[2],2));
                double temp_ang_mag = sqrt(pow(deltaX[3],2)+pow(deltaX[4],2)+pow(deltaX[5],2));
                //velocity saturation
                if (temp_lin_mag > gv.dxmax)
                {
                      for(int i=0; i<=2; i++)
                                deltaX[i] *= (gv.dxmax/temp_lin_mag);
                }
                if (temp_ang_mag > gv.wmax)
                {
                      for(int i=3; i<=5; i++)
                                deltaX[i] *= (gv.wmax/temp_ang_mag);
                }

                force = -gv.kv*(gv.dx-deltaX);

                gv.tau=gv.Jtranspose*(force+gv.p);
}

void trackControl(GlobalVariables& gv)
{

  if(gv.singularities !=0) ntrackControl(gv);

else {
                Float t=gv.curTime-splineStartTime;
                PrVector xd_t;
                PrVector dxd_t;

                if( t< timeF){
                      for(int i=0; i<3;i++){
                                if (2*splineParam2[i]+6*splineParam3[i]*t > gv.ddxmax)  splineParam3[i]=(gv.ddxmax-2
*splineParam2[i])/t;
                      }
                      xd_t = splineParam0+splineParam2*t*t+splineParam3*t*t*t;
                      dxd_t = gv.Einverse*(2*splineParam2*t+3*splineParam3*t*t);
                }
                if (t>timeF) {
                      xd_t = gv.xd;
                      dxd_t = gv.dxd;
                }

                PrVector force = PrVector(6);
                PrVector deltaX = dxd_t-gv.kp*(gv.Einverse*(gv.x-xd_t));

                      for(int i = 0; i<deltaX.size(); i++)
                {
                      if (gv.kv[i] < 0.0001)
                                gv.kv[i] = 0.0001;
                      deltaX[i] = (1/gv.kv[i])*(deltaX[i]);
                }

   //calculate velocity
                double temp_lin_mag = sqrt(pow(deltaX[0],2)+pow(deltaX[1],2)+pow(deltaX[2],2));
                double temp_ang_mag = sqrt(pow(deltaX[3],2)+pow(deltaX[4],2)+pow(deltaX[5],2));
                //velocity saturation
                if (temp_lin_mag > gv.dxmax)
                {
                      for(int i=0; i<=2; i++)
                                deltaX[i] *= (gv.dxmax/temp_lin_mag);
                }
                if (temp_ang_mag > gv.wmax)
                {
                      for(int i=3; i<=5; i++)
                                deltaX[i] *= (gv.wmax/temp_ang_mag);
                }
```

```
		force = -gv.kv*(gv.dx-deltaX);
		OpDynamics(force,gv);


}
}

void pfmoveControl(GlobalVariables& gv)
{
  PrVector force = PrVector(6);
		PrVector deltaX = -gv.kp*(gv.Einverse*(gv.x-gv.xd));

for(int i = 3; i<deltaX.size(); i++)
		{
			if (gv.kv[i] < 0.0001)
				gv.kv[i] = 0.0001;
			deltaX[i] = (1/gv.kv[i])*(deltaX[i]);
		}

    //calculate velocity - angular velocity saturation
		double temp_ang_mag = sqrt(pow(deltaX[3],2)+pow(deltaX[4],2)+pow(deltaX[5],2));
		if (temp_ang_mag > gv.wmax)
		{
			for(int i=3; i<deltaX.size(); i++)
				deltaX[i] *= (gv.wmax/temp_ang_mag);
		}
		for(int i = 3; i<deltaX.size(); i++){
		force[i] = -gv.kv[i]*(gv.dx[i]-deltaX[i]);
		}
		OpDynamics(force,gv);
}

void lineControl(GlobalVariables& gv)
{
  floatControl(gv);  // Remove this line when you implement openControl
}

void proj1Control(GlobalVariables& gv)
{
  floatControl(gv);  // Remove this line when you implement proj1Control
}

void proj2Control(GlobalVariables& gv)
{
  floatControl(gv);  // Remove this line when you implement proj2Control
}

void proj3Control(GlobalVariables& gv)
{
  floatControl(gv);  // Remove this line when you implement proj3Control
}

void getNonsingularSelection(PrMatrix& selectionMatrix, GlobalVariables& gv){
		PrMatrix3 transform=PrMatrix3(); transform.identity();
		PrMatrix3 transform2=PrMatrix3();
		PrVector v[6];
		selectionMatrix.setSize(5,6);
		PrMatrix selectionTranspose =PrMatrix(6,5);
		for (int i=0;i<6;i++){
			v[i]=PrVector(6);
			v[i][i]=1;
		}

		if(gv.singularities ==HEAD_LOCK ){ //singular direction is y1
			//rotation of y0 by q[0] about z0=z1
			transform.setRotation( PrVector3(v[2].subvector(0,3)) ,gv.q[0] );
```

```
		v[0].subvector(0,3).transfer( transform*(v[0].subvector(0,3)) );
		v[2].subvector(0,3).transfer( transform*(v[2].subvector(0,3)) );

		selectionTranspose.setColumn(0,v[0]);
		selectionTranspose.setColumn(1,v[2]);
		selectionTranspose.setColumn(2,v[3]);
		selectionTranspose.setColumn(3,v[4]);
		selectionTranspose.setColumn(4,v[5]);
	}
	else if(gv.singularities == ELBOW_LOCK) { //singular direction y3=z4
		// rotation of y0 by q[0] about z0 and q[1] about y1 and q[2] about z2
		transform.setRotation( PrVector3(v[2].subvector(0,3)) ,gv.q[0] );
		v[0].subvector(0,3).transfer( transform*(v[0].subvector(0,3)) );
		v[1].subvector(0,3).transfer( transform*(v[1].subvector(0,3)) );
		v[2].subvector(0,3).transfer( transform*(v[2].subvector(0,3)) );

		transform.setRotation( PrVector3(v[1].subvector(0,3)) ,gv.q[1] );
		v[0].subvector(0,3).transfer( transform*(v[0].subvector(0,3)) );
		v[1].subvector(0,3).transfer( transform*(v[1].subvector(0,3)) );
		v[2].subvector(0,3).transfer( transform*(v[2].subvector(0,3)) );

		transform.setRotation( PrVector3(v[2].subvector(0,3)) ,gv.q[2] );
		v[0].subvector(0,3).transfer( transform*(v[0].subvector(0,3)) );
		v[1].subvector(0,3).transfer( transform*(v[1].subvector(0,3)) );
		v[2].subvector(0,3).transfer( transform*(v[2].subvector(0,3)) );

		selectionTranspose.setColumn(0,v[0]);
		selectionTranspose.setColumn(1,v[2]);
		selectionTranspose.setColumn(2,v[3]);
		selectionTranspose.setColumn(3,v[4]);
		selectionTranspose.setColumn(4,v[5]);
	}
	else if(gv.singularities == WRIST_LOCK)   {// singular direction is rotation about x4
		// rotation of y0 by q[0] about z0, and q[1] about y1, and q[2] about z2, and q[3] about -y3=z4
			transform.identity();
			transform2.setRotation( (PrVector3(v[2].subvector(0,3))) ,gv.q[0] ); //frame 1
			for (int i=0;i<3;i++){		//compute x1,y1...
				v[i].subvector(0,3).transfer( transform2*(v[i].subvector(0,3)) );
			}
			transform=transform2*transform;
			transform2.setRotation( (PrVector3(v[1].subvector(0,3))) ,gv.q[1] ); //frame 2
			for (int i=0;i<3;i++){		//compute x2,y2...
				v[i].subvector(0,3).transfer( transform2*(v[i].subvector(0,3)) );
			}
			transform=transform2*transform;
			transform2.setRotation( (PrVector3(v[2].subvector(0,3))) ,gv.q[2] );  //frame 3
			for (int i=0;i<3;i++){		//compute x3,y3...
				v[i].subvector(0,3).transfer( transform2*(v[i].subvector(0,3)) );
			}
			transform=transform2*transform;
			transform2.setRotation( (PrVector3(-v[1].subvector(0,3))) ,gv.q[3] ); //frame4
			for (int i=0;i<3;i++){		//compute x4,y4...
				v[i].subvector(0,3).transfer( transform2*(v[i].subvector(0,3)) );
			}
			transform=transform2*transform;
			for (int i=3;i<6;i++){
				v[i].subvector(2,3).transfer( transform*(v[i].subvector(2,3)) );
			}

		selectionTranspose.setColumn(0,v[0]);
		selectionTranspose.setColumn(1,v[2]);
		selectionTranspose.setColumn(2,v[3]);
		selectionTranspose.setColumn(3,v[4]);
		selectionTranspose.setColumn(4,v[5]);
	}
	selectionTranspose.transpose(selectionMatrix);
}
```

```
void getEscapeTorque (const PrVector& fPrime, PrVector& escapeTorque, GlobalVariables& gv)
{
		PrVector deltaX =PrVector(gv.x.size());
		PrVector deltaQ= PrVector(gv.dof);
		float magnitude=0, angle=0;  //used for HEAD_LOCK
		float radial_displacement = deltaX.subvector(0,3).dot(gv.x.subvector(0,3)); // used for ELBOW_LOCK
		PrMatrix3 transform  = PrMatrix3();
		PrMatrix3 transform2 = PrMatrix3();
		PrVector v[3];  //x,y,z
		double delta_xx,delta_xy;  // projection of deltaX on (x1,y1) plane
		for (int i=0;i<3;i++){  //x,y,z,w1,w2,w4
			v[i]=PrVector(6);
			v[i][i]=1;
		}


		for (int i=0;i<6;i++){		//approximation of desired displacement
			deltaX[i]=fPrime[i]/gv.kp[i];
		}

		switch (gv.singularities){

			case HEAD_LOCK:
				transform.setRotation( PrVector3(v[2].subvector(0,3)) ,gv.q[0] );
				v[0].subvector(0,3).transfer( transform*(v[0].subvector(0,3)) ); //x1
				v[1].subvector(0,3).transfer( transform*(v[1].subvector(0,3)) ); //y1
				delta_xx = deltaX.subvector(0,3).dot(v[0].subvector(0,3));
				delta_xy = deltaX.subvector(0,3).dot(v[1].subvector(0,3));;
				magnitude=sqrt(pow(delta_xx,2)+pow(delta_xy,2));
				if (delta_xx == 0) angle = 0;
				else angle = atan(delta_xy/delta_xx);
				deltaQ[0] = magnitude*angle;
				break;

			case ELBOW_LOCK :
				if (radial_displacement > 0) {
					deltaQ[3]= (gv.q[3]-3.14159/2)*radial_displacement;
					deltaQ[2]= - deltaQ[3];
				}
				else if (radial_displacement <0 ){
					deltaQ[2]= radial_displacement;
					deltaQ[1]= - deltaQ[2];
				}
				break;

			case WRIST_LOCK :
				transform.identity();
				transform2.setRotation( (PrVector3(v[2].subvector(0,3))) ,gv.q[0] ); //frame 1
				for (int i=0;i<3;i++){		//compute x1,y1...
					v[i].subvector(0,3).transfer( transform2*(v[i].subvector(0,3)) );
				}
				transform=transform2*transform;
				transform2.setRotation( (PrVector3(v[1].subvector(0,3))) ,gv.q[1] ); //frame 2
				for (int i=0;i<3;i++){		//compute x2,y2...
					v[i].subvector(0,3).transfer( transform2*(v[i].subvector(0,3)) );
				}
				transform=transform2*transform;
				PrVector3  z2= PrVector3((v[2].subvector(0,3)));
				transform2.setRotation( z2 ,gv.q[2] );  //frame 3
				for (int i=0;i<3;i++){		//compute x3,y3...
					v[i].subvector(0,3).transfer( transform2*(v[i].subvector(0,3)) );
				}
				transform=transform2*transform;
				transform2.setRotation( (PrVector3(-v[1].subvector(0,3))) ,gv.q[3] ); //frame4
				for (int i=0;i<3;i++){		//compute x4,y4...
					v[i].subvector(0,3).transfer( transform2*(v[i].subvector(0,3)) );
```

```
                    }
                    transform=transform2*transform;
                    delta_xx = deltaX.subvector(2,3).dot(v[0].subvector(0,3));
                    delta_xy = deltaX.subvector(2,3).dot(v[1].subvector(0,3));
                    magnitude = sqrt(pow(delta_xx,2)+pow(delta_xy,2));
                    if (delta_xx == 0) angle = 0;
                    else angle = atan(delta_xy/delta_xx);
                    deltaQ[3] = magnitude * angle;
                    deltaQ[5] = - deltaQ[3];
                    break;
            }


}
```