

CINEMATIQUE INVERSE APPLIQUEE AU BRAS
DE ROBOT POPPY-HUMANOÏDE

Par

Xavier MARIOT

Étude entrant dans le cadre d'un stage effectué
entre la 2^{ème} et 3^{ème} année du cursus :

FITE de l'ENSA
(formation ingénieur généraliste)

Arts Et Métiers ParisTech – CER Bordeaux-Talence

Été 2015

Tuteur de stage :

Jean-Luc CHARLES

En partenariat avec :

l'INRIA - Bordeaux

TABLE DES MATIERES

Table des matières	i
Introduction.....	ii
Glossaire et notations.....	iii
Chapitre 1	1
Présentation du robot Poppy-humanoïde	1
Poppy-humanoïde	1
Paramétrage du bras (en cours de rédaction)	3
Analyse des Problématiques	4
Choisir le modèle cinématique directe du système.....	4
Trouver les paramètres de la position finale.....	5
Piloter le système de sa position réelle initiale à la position finale	5
Stratégie globale de résolution	6
Modélisation préliminaire de l'architecture cinématique du système	6
Obtention des paramètres inverses.....	7
Pilotage réel du robot vers la position finale.....	9
Chapitre 2 (En cours de rédaction)	10
Application au bras de robot Poppy-humanoïde	10
Modèle cinématique directe	10
Mapping des positions théoriques initiales	11
Détermination d'une position initiale optimale pour démarrer l'algorithme.....	11
Caractérisation d'une trajectoire élémentaire de résolution	11
Résolution par itération de la méthode de Jacobi.....	12
BIBLIOGRAPHIE.....	a
ANNEXES.....	b

INTRODUCTION

Cette étude vise à résoudre le problème de cinématique inverse dans le pilotage des mouvements d'un robot qui est un problème couramment rencontré dans le domaine de la robotique. Le problème revient à déterminer les valeurs des paramètres (angles ou translations de servomoteurs, etc.) à imposer pour un modèle cinématique donné du robot afin que le système ou une partie du système soit dans une position voulue.

La difficulté de la résolution de ce problème est en général la non unicité de la solution permettant de donner cette position au système ou à une partie du système c.à.d. qu'il y a une infinité de valeurs possibles pour les paramètres de commande qui vont toutes amener le robot à la position voulue. Il arrive également que aucune solution n'existe car on impose trop de contraintes sur la pose que doit prendre le système, ce problème n'est pas traité dans cette étude mais se résout de manière similaire par la méthode des moindres carrés.

Ce document présente dans un premier temps un découpage des différentes problématiques et une stratégie globale de résolution de ce problème de cinématique inverse. Est ensuite exposé l'application de cette méthode au bras de robot Poppy avec les différents algorithmes et outils mathématiques utilisés permettant d'implémenter numériquement cette stratégie.

Pour donner un aspect pratique, la résolution du problème est appliquée au bras de robot humanoïde Poppy tout au long de l'étude mais peut aisément être adapté à tout autre système (jambe, buste, bras articulé 6 axes, etc.).

GLOSSAIRE ET NOTATIONS

Vecteurs et fonctions vectorielles : Tout au long de ce document, les vecteurs et les fonctions vectorielles seront notés en gras. Ex. « la fonction \mathbf{f} », ou encore « l'ensemble des paramètres $\mathbf{q} = (q_1, \dots, q_n)$ »

$D\mathbf{f}$ représente la différentielle de \mathbf{f} .

Le lecteur ne se laissera pas perturber par le fait que pour une application linéaire, notée ϕ par exemple, on parle indifféremment de l'application linéaire ou de la matrice associée sans nécessairement changer de notation.

Pour un vecteur $\mathbf{u} = (u_1, \dots, u_n)$ et un repère orthonormé $R0 = (\mathbf{x}_1, \dots, \mathbf{x}_n)$, on note $M_{R0}^{\mathbf{u}} \in \mathbb{R}_n$ la matrice représentative des coordonnées de \mathbf{u} dans $R0$.

Pour deux repères orthonormés $R1$ et $R0$, on note M_{R0}^{R1} la matrice de $R1$ dans $R0$ ou matrice de changement de base telle que pour tout vecteur \mathbf{u} :

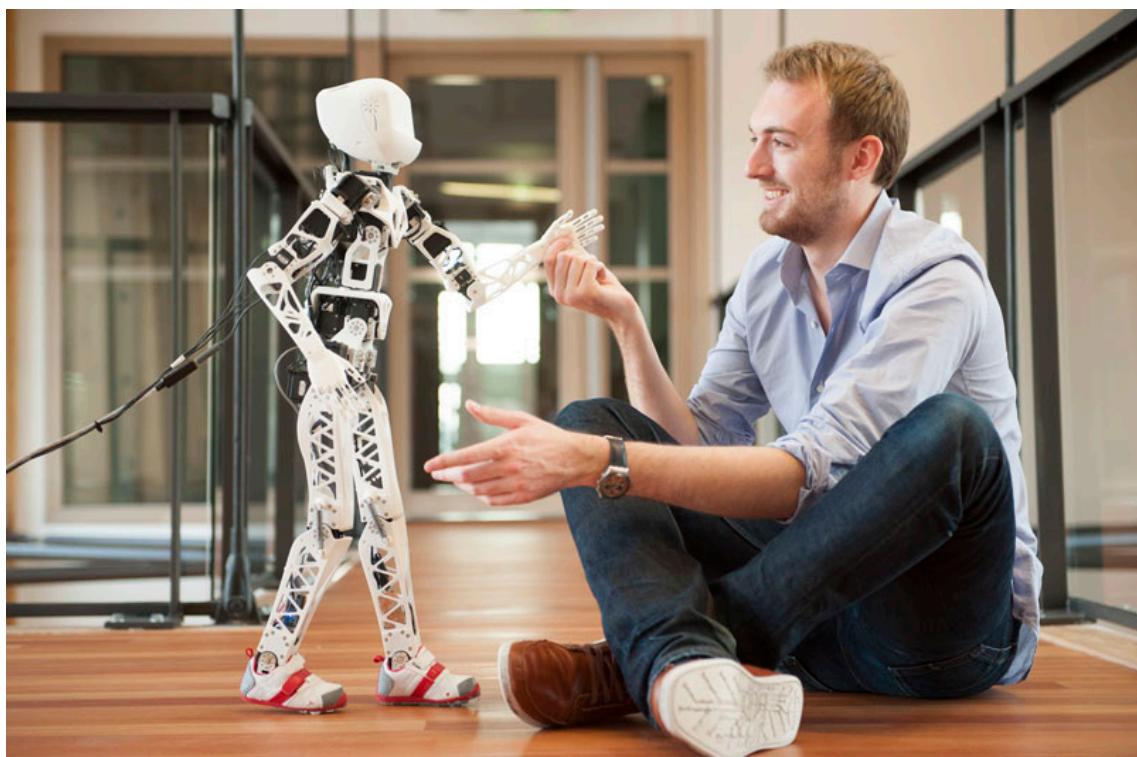
$$M_{R0}^{\mathbf{u}} = M_{R0}^{R1} * M_{R1}^{\mathbf{u}}$$

Chapitre 1

PRESENTATION DU ROBOT POPPY-HUMANOÏDE

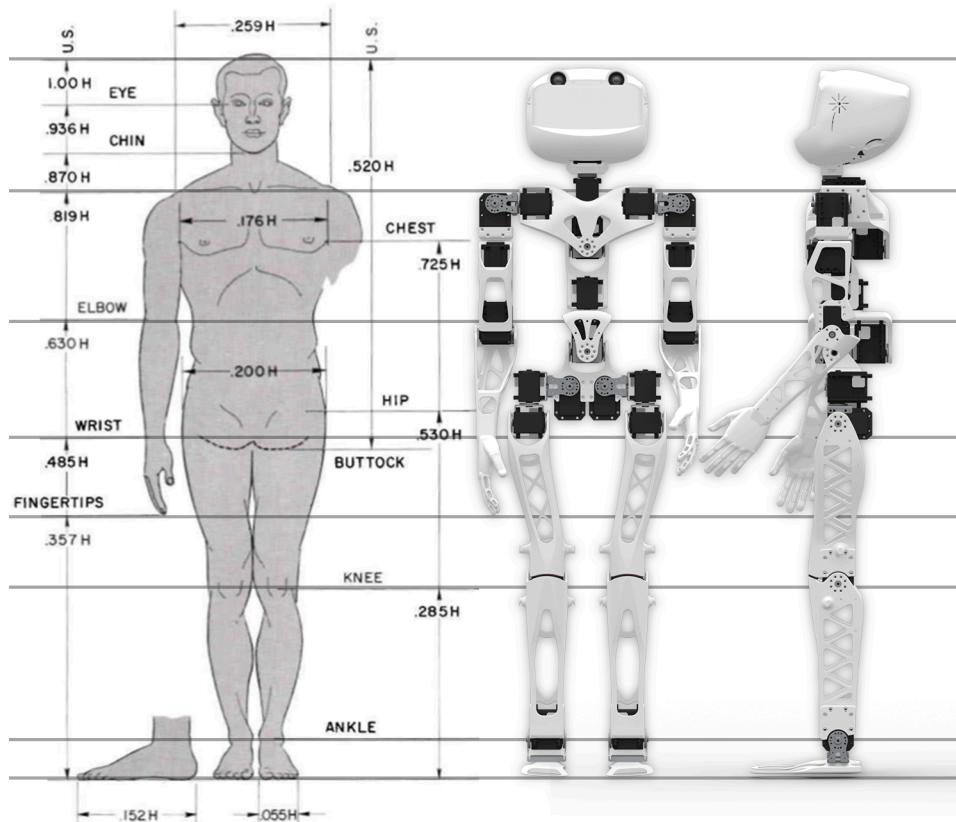
Poppy-humanoïde

Comme indiqué dans l'introduction, cette étude est appliquée au bras du robot Poppy-Humanoïde développé par le laboratoire de recherche de l'INRIA-Bordeaux.

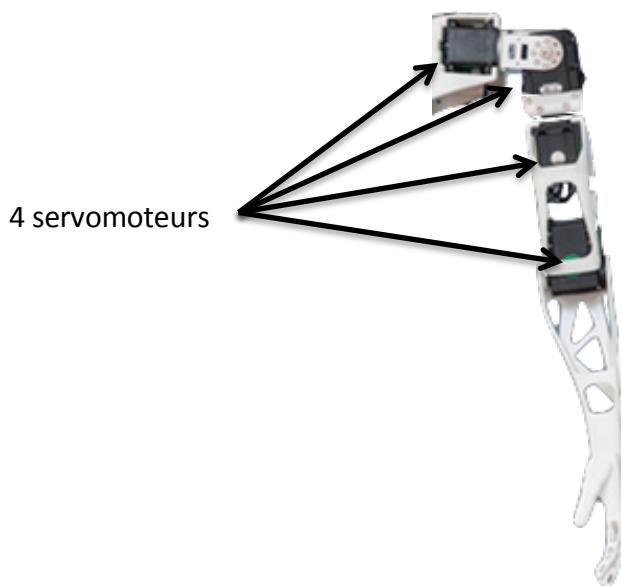


Poppy et M. LAPEYRE membre de l'équipe FLOWERS au laboratoire de l'INRIA

Son profil et ses proportions sont présentés sur le schéma suivant :



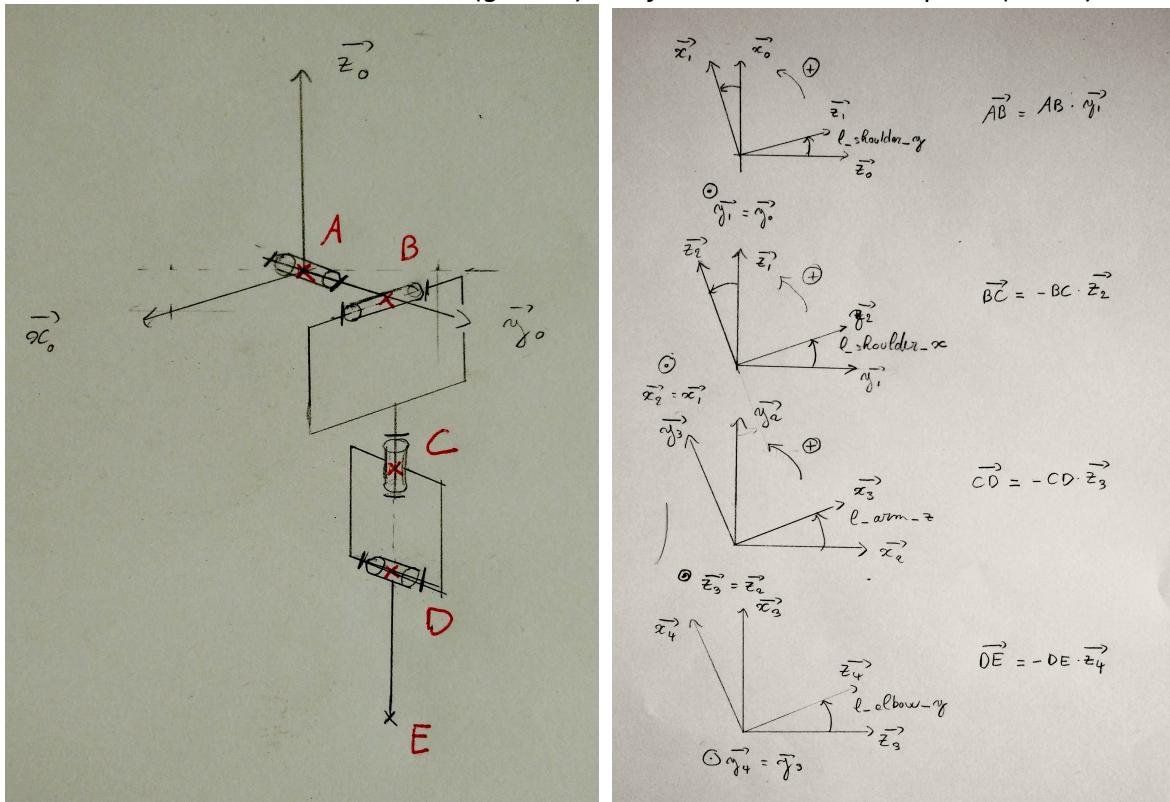
Pour mettre en place notre algorithme, nous l'appliquons au bras de ce robot humanoïde. Cela permet dans un cas simple (bras à 4 degrés de liberté) de vérifier concrètement le fonctionnement de l'algorithme en cherchant par exemple à positionner la main à une position $M(x, y, z)$ dans le repère du buste.



Paramétrage du bras (en cours de rédaction)

Afin que le modèle qui est mis en place soit compatible avec les différents développements déjà réalisés sur Poppy-humanoïde, nous allons utiliser les différents repères définis à partir des schémas suivants. Ainsi les « zéros » des différents moteurs et les sens positifs de rotation seront respectés par rapport à la configuration « habituelle » de Poppy.

Position « zéro » du bras (gauche) – Définition relative des repères (droite)



ANALYSE DES PROBLEMATIQUES

Dans cette étude de cinématique inverse, il est indispensable d'identifier et de séparer les différentes problématiques que l'on cherche à résoudre. Nous pouvons distinguer trois parties majeures indépendantes.

Le premier problème est de choisir une modélisation cinématique adéquate de notre système par rapport aux objectifs de contrôle de la position. Le second problème est d'obtenir à partir de la modélisation choisie la relation de cinématique inverse. Enfin le dernier problème est de piloter le système de la position actuelle où il se trouve à la position finale obtenue par l'algorithme de cinématique inverse.

Choisir le modèle cinématique directe du système

La problématique ici est la mise en équation de la structure du robot et le choix du modèle mathématique qui représente la cinématique directe du système. En effet selon le pilotage que l'on veut pouvoir effectuer (position (x, y, z) dans un repère, positions angulaires, etc.) certains modèles mathématiques sont plus adaptés que d'autres.

Prenons le cas du bras de robot Poppy-humanoïde. Dans un cas on veut piloter la position de la main de Poppy, par exemple le bout de l'index représenté par un point M de coordonnées (x, y, z) dans le repère du buste. Dans ce cas un modèle vectoriel simple c.à.d. un vecteur **OM** qui représente la position de l'index en fonction des quatre angles des servomoteurs et des dimensions du bras suffit comme modèle cinématique directe. Par contre dans un autre cas où l'on voudrait par exemple faire garder une position horizontale à l'avant bras de Poppy et déplacer sa main, ce modèle n'est plus adapté car il ne rend pas compte de la position angulaire du repère du bras par rapport au repère du buste.

A l'issue de cette étape nous devrons avoir un modèle mathématique exploitable par l'algorithme de cinématique inverse qui suit.

Trouver les paramètres de la position finale

Il s'agit ici de trouver l'algorithme de cinématique inverse qui va retourner les valeurs des paramètres pilotables directement sur le système pour que le robot soit dans une position voulue. Attention il ne s'agit pas ici de piloter le robot, ce n'est pas le rôle de cet algorithme ! Il doit simplement répondre à la question : « Si je veux que mon système soit dans telle position, quelles sont les valeurs qu'auraient les différents paramètres d'entrée de mon système ? ».

Reprendons le cas de l'index du bras de Poppy, la question posée est : Quelles sont les valeurs des quatre angles des servomoteurs pour que l'index soit à la position (x_1, y_1, z_1) à partir du modèle cinématique direct mis en place ? L'algorithme doit renvoyer les quatre valeurs qu'ont les servomoteurs lorsque l'index est à cette position.

La difficulté majeure est qu'il peut y avoir aucune, une seule ou une infinité de solutions. Comme indiqué dans l'introduction, nous traiterons dans cette étude uniquement le cas où au moins une solution existe c.à.d. une seule ou une infinité.

Piloter le système de sa position réelle initiale à la position finale

Le dernier problème à résoudre est maintenant de déplacer physiquement le système de la position où il se trouve actuellement vers la position définie à l'issue de l'algorithme précédent. Les questions qui se posent sont de savoir comment nous faisons varier les différents paramètres du système pour arriver aux paramètres finaux. Un problème qui peut se poser est le passage du système par des positions interdites, par exemple le bras de Poppy qui traverserait son corps ce qui est techniquement impossible.

STRATEGIE GLOBALE DE RESOLUTION

Les problématiques présentées précédemment induisent naturellement la stratégie de résolution qui suit.

Modélisation préliminaire de l'architecture cinématique du système

Des modèles optimisés existent, comme les paramètres de Denavit-Hartenberg, et permettent d'incorporer un maximum d'informations sur la structure et la position des repères de chaque élément du système en un minimum de paramètres. Pour des raisons de temps, ceux-ci n'ont pas été implémentés dans cette étude. Nous utilisons un modèle vectoriel simple mais tout autre modèle peut être utilisé avec les algorithmes qui suivent.

Dans le choix du modèle cinématique direct on définit donc une fonction f comme suit caractéristique de la cinématique du robot :

$$f : \begin{cases} \mathbb{R}^n \rightarrow \mathbb{R}^m \\ q = (q_1, \dots, q_n) \rightarrow x = (x_1, \dots, x_m) = f(q) \end{cases}$$

Où q représente l'ensemble des paramètres pilotés directement sur le robot, dans le cas du bras de Poppy les quatre angles des servomoteurs

x représente l'ensemble des paramètres que l'on veut imposer sur l'état du système, par exemple les coordonnées (x, y, z) de la main de Poppy dans le repère du buste.

Obtention des paramètres inverses

Dans cet algorithme on veut obtenir la fonction inverse de \mathbf{f} :

$$\mathbf{f}^{-1} : \begin{cases} \mathbb{R}^m \rightarrow \mathbb{R}^n \\ \mathbf{x} = (x_1, \dots, x_m) \rightarrow \mathbf{q} = (q_1, \dots, q_n) = \mathbf{f}^{-1}(\mathbf{x}) \end{cases}$$

Cependant comme mentionné précédemment, cette fonction n'existe pas dans la majorité des cas car il y a une infinité de solutions. Une première approche consiste à vouloir imposer des conditions supplémentaires sur la position du système afin qu'il ne reste qu'une unique solution. Cependant cette méthode n'est pas assez générale, elle induit une gestion « cas par cas » trop contraignante.

La méthode que nous utilisons consiste à définir une position initiale théorique du système \mathbf{q}_{init} qui va permettre de définir une variation minimale du vecteur \mathbf{q} entre cette position initiale et la position finale $\mathbf{q}_{\text{final}}$ que l'on recherche telle que $\mathbf{f}(\mathbf{q}_{\text{final}}) = \mathbf{x}_{\text{final}}$. En effet il existe une seule solution dans la grande majorité des cas qui minimise la norme :

$$\|\mathbf{q}_{\text{final}} - \mathbf{q}_{\text{init}}\|$$

Attention, il faut bien garder en tête que \mathbf{q}_{init} n'a rien à voir avec la position réelle du système, c'est une position imaginaire utilisée pour obtenir numériquement dans un algorithme une solution $\mathbf{q}_{\text{final}}$ qui vérifie $\mathbf{f}(\mathbf{q}_{\text{final}}) = \mathbf{x}_{\text{final}}$.

On note $\Delta\mathbf{q} = \mathbf{q}_{\text{final}} - \mathbf{q}_{\text{init}}$ et $\Delta\mathbf{x} = \mathbf{x}_{\text{final}} - \mathbf{x}_{\text{init}}$

Où $\mathbf{f}(\mathbf{q}_{\text{final}}) = \mathbf{x}_{\text{final}}$ et $\mathbf{f}(\mathbf{q}_{\text{init}}) = \mathbf{x}_{\text{init}}$

La seule inconnue est $\mathbf{q}_{\text{final}}$, tous les autres éléments sont connus (NB : \mathbf{x}_{init} est déterminé par le modèle directe à partir de \mathbf{q}_{init}).

Dans l'hypothèse de petits déplacements on peut appliquer la méthode de Jacobi :

$$\mathbf{x}_{\text{final}} = \mathbf{f}(\mathbf{q}_{\text{final}}) = \mathbf{f}(\mathbf{q}_{\text{init}} + \Delta\mathbf{q}) \approx \mathbf{f}(\mathbf{q}_{\text{init}}) + \mathbf{D}\mathbf{f}(\mathbf{q}_{\text{init}}) * \Delta\mathbf{q} = \mathbf{x}_{\text{init}} + \mathbf{D}\mathbf{f}(\mathbf{q}_{\text{init}}) * \Delta\mathbf{q}$$

Donc en notant \mathbf{J} la jacobienne de \mathbf{f} en \mathbf{q}_{init} :

$$\Delta\mathbf{x} \approx \mathbf{J} * \Delta\mathbf{q}$$

En utilisant les pseudo-inverses de Moore-Penrose, on obtient la solution qui minimise $\Delta\mathbf{q}$ par l'équation :

$$\Delta\mathbf{q} \approx \mathbf{J}^+ * \Delta\mathbf{x}$$

Où \mathbf{J}^+ est la matrice pseudo-inverse de \mathbf{J} .

On obtient ainsi $\mathbf{q}_{\text{final}}$:

$$\mathbf{q}_{final} \approx \mathbf{q}_{init} + J^+ * \Delta \mathbf{x}$$

Le calcul pratique de J^+ se fait assez simplement dans notre cas. J est représentative d'un système de n équations à m inconnues avec $n \leq m$ où, sauf cas particulier, les équations sont deux à deux libres, donc le rang de J est égal au nombre de lignes de J . Alors J^+ se calcule par la formule :

$$J^+ = J^t * (J * J^t)^{-1}$$

Où J^t est la transposée de J .

Dans le cas où $(J * J^t)$ n'est pas inversible, cela signifie que deux des lignes de J sont localement liées, il suffit alors de déplacer légèrement la position initiale d'une valeur aléatoire δq telle que $\mathbf{q}_{init, bis} = \mathbf{q}_{init} + \delta \mathbf{q}$ et de recalculer la jacobienne de f en $\mathbf{q}_{init, bis}$.

Il peut arriver qu'une seule itération du calcul donne un résultat numérique \mathbf{q}_{final} qui pour une tolérance IT donnée ne vérifie pas $||\mathbf{x}_{final} - f(\mathbf{q}_{final})|| \leq IT$. Cela est dû à une mauvaise vérification de l'hypothèse des « petits déplacements ». Quelques itérations de l'algorithme (moins de dix) en prenant comme nouvelle position initiale $\mathbf{q}_{init, k} = \mathbf{q}_{final, k-1}$ permet d'atteindre la tolérance voulue pour de faibles déplacements.

Dans le **cas d'un déplacement quelconque** on met en place deux nouveaux outils.

Premièrement on définit un « Mapping » d'une dizaine de positions initiales théoriques qui associent des positions \mathbf{x}_k à des valeurs connues des paramètres \mathbf{q}_k vérifiant la relation du modèle directe $\mathbf{x}_k = f(\mathbf{q}_k)$. Ainsi on choisira parmi les positions du Mapping la valeur initiale de départ \mathbf{q}_i telle que la distance $||\mathbf{x}_{final} - \mathbf{x}_i||$ soit minimale.

Par ailleurs si le tableau de Mapping des positions théoriques de départ est correctement choisi par rapport à l'architecture réelle du système, celui-ci permet d'éviter des valeurs \mathbf{q}_{final} interdites pour le système (cf. collisions entre pièces).

Une fois la position initiale de départ déterminée on découpe le trajet $\mathbf{x}_{initial} - \mathbf{x}_{final}$ en petits déplacements intermédiaires $\Delta \mathbf{x}_k = \mathbf{x}_{final, k} - \mathbf{x}_{init, k}$ pour lesquels on applique l'algorithme précédent.

A l'issu de cet algorithme on obtient donc l'ensemble des paramètres \mathbf{q}_{final} tel que :

$$||\mathbf{x}_{final} - f(\mathbf{q}_{final})|| \leq IT$$

Pilotage réel du robot vers la position finale

Cet algorithme définit l'évolution de la position initiale réelle \mathbf{q}_{init} du système vers la position $\mathbf{q}_{\text{final}}$ obtenue en sortie de l'algorithme précédent. Il est extrêmement sommaire, et ne règle pas les problèmes de collision au sein du système de façon exhaustive.

On peut découper le mouvement en n étapes et définir un déplacement élémentaire $\delta \mathbf{q}$ tel que :

$$\delta \mathbf{q} = \frac{\mathbf{q}_{\text{init}} - \mathbf{q}_{\text{final}}}{n} \quad (\#)$$

Ainsi à l'étape k : $\mathbf{q}_k = k * \delta \mathbf{q}$. Cependant cette méthode a de fortes chances d'engendrer des positions interdites pour le système lors du déplacement.

L'algorithme de pilotage doit également éviter les collisions majeures des différentes parties du système entre elles. Ce problème n'est pas étudié dans le détail mais une première approche peut consister à utiliser le Mapping défini pour l'algorithme précédent.

Dans un premier temps on amène le système de sa position initiale à la position du Mapping la plus proche selon le principe de l'équation (#). Avec un mapping bien défini (propre à chaque système), les chances de collisions sont faibles.

Ensuite on définit des trajectoires entre les différentes positions du Mapping qui n'induisent pas de collision. On amène le système grâce à ces trajectoires jusqu'à la position initiale qui a été utilisée pour obtenir $\mathbf{q}_{\text{final}}$ lors de l'algorithme de cinématique inverse. Puis on applique à nouveau un découpage homogène du mouvement en utilisant (#) entre la position où se trouve maintenant le bras et la position $\mathbf{q}_{\text{final}}$.

APPLICATION AU BRAS DE ROBOT POPPY-HUMANOÏDE

L'application au bras de robot de poppy humanoïde de la méthode présentée précédemment est faite en python. Le fichier kinematics.py mis en annexe contient l'ensemble des algorithme. Les sections suivantes viennent compléter et expliquer certaines formules du fichier kinematics.py.

Modèle cinématique direct

Nous définissons ici la fonction vectorielle \mathbf{f} représentative de la position de la main de poppy dans le repère du buste avec pour origine le centre du moteur « l_shoulder_y. »

$$\mathbf{f} : \left\{ \begin{array}{l} \mathbb{R}^4 \rightarrow \mathbb{R}^3 \\ \mathbf{q} = (q_1, q_2, q_3, q_4) \rightarrow \mathbf{x} = (x_0, y_0, z_0) = \mathbf{f}(\mathbf{q}) \end{array} \right.$$

où $q_1 = l_shoulder_y$; $q_2 = l_shoulder_x$; $q_3 = l_arm_z$; $q_4 = l_elbow_y$

Pour déterminer cette fonction on calcule les matrices de changement de repère suivantes et on note les distances conformément au schéma donné en page 3.

$$M_{R_0}^{R_1} = \begin{bmatrix} \cos(q_1) & 0 & \sin(q_1) \\ 0 & 1 & 0 \\ -\sin(q_1) & 0 & \cos(q_1) \end{bmatrix}$$

$$M_{R_1}^{R_2} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(q_2) & -\sin(q_2) \\ 0 & \sin(q_2) & \cos(q_2) \end{bmatrix}$$

$$M_{R_2}^{R_3} = \begin{bmatrix} \cos(q_3) & -\sin(q_3) & 0 \\ \sin(q_3) & \cos(q_3) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$M_{R_3}^{R_4} = \begin{bmatrix} \cos(q_4) & 0 & \sin(q_4) \\ 0 & 1 & 0 \\ -\sin(q_4) & 0 & \cos(q_4) \end{bmatrix}$$

Les vecteurs **AB**, **BC**, **CD** et **DE** s'écrivent simplement dans leurs repères respectifs R_1 , R_2 , R_3 et R_4 .

$$M_{R_1}^{AB} = \begin{bmatrix} 0 \\ ab \\ 0 \end{bmatrix} ; M_{R_2}^{BC} = \begin{bmatrix} 0 \\ 0 \\ -bc \end{bmatrix} ; M_{R_3}^{CD} = \begin{bmatrix} 0 \\ 0 \\ -cd \end{bmatrix} ; M_{R_4}^{DE} = \begin{bmatrix} 0 \\ 0 \\ -de \end{bmatrix}$$

Ainsi :

$$M_{R_0}^{f(q)} = M_{R_0}^{AB} = M_{R_0}^{AB} + M_{R_0}^{BC} + M_{R_0}^{CD} + M_{R_0}^{DE}$$

Or :

$$M_{R_0}^{AB} = M_{R_0}^{R_1} * M_{R_1}^{AB}$$

$$M_{R_0}^{BC} = M_{R_0}^{R_1} * M_{R_1}^{R_2} * M_{R_2}^{BC}$$

$$M_{R_0}^{CD} = M_{R_0}^{R_1} * M_{R_1}^{R_2} * M_{R_2}^{R_3} * M_{R_3}^{CD}$$

$$M_{R_0}^{DE} = M_{R_0}^{R_1} * M_{R_1}^{R_2} * M_{R_2}^{R_3} * M_{R_3}^{R_4} * M_{R_4}^{DE}$$

Avec un logiciel de calcul formel (ou en calculant soigneusement à la main) on a alors :

$$M_{R_0}^{f(q)} =$$

Les valeurs numériques en mm relevées sur le bras de poppy-humanoïde sont les suivantes :

$$ab = 30 ; bc = 45 ; cd = 105 ; de = 120 + 90/2 \text{ (avant bras + milieux de la main)}$$

Voir la fonction dk_l_hand du fichier kinematics.py qui correspond à cette fonction.

Mapping des positions théoriques initiales

On détermine une dizaine de position du bras pour couvrir les différentes zones de l'espace que peut atteindre le bras de robot. Un ensemble sommaire de ces positions est défini dans la variable globale mapping_l_hand du fichier kinematics.py.

Détermination d'une position initiale optimale pour démarrer l'algorithme

Il s'agit ici de faire une boucle qui parcourt l'ensemble des positions contenues dans le mapping et d'utiliser comme position initiale celle qui est la plus proche de la position finale.

Le code correspondant est indiqué dans la fonction ik_num du fichier kinematics.py.

Caractérisation d'une trajectoire élémentaire de résolution

Lors de cette étape on découpe la trajectoire de la position initiale à la position finale en étapes élémentaires qui pourront être traitées successivement par la méthode de jacobi pour les petits déplacements. Le code correspondant est indiqué dans le fonction ik_num du fichier kinematics.py

Résolution par itération de la méthode de Jacobi

On applique la méthode décrite dans le cas général. On rappelle que la Jacobienne dans le cas de notre fonction \mathbf{f} se calcule de la manière suivante :

$$J = \begin{bmatrix} \frac{df}{dq_1} & \frac{df}{dq_2} & \frac{df}{dq_3} & \frac{df}{dq_4} \end{bmatrix} = \begin{bmatrix} \frac{\partial f_x}{\partial q_1} & \frac{\partial f_x}{\partial q_2} & \frac{\partial f_x}{\partial q_3} & \frac{\partial f_x}{\partial q_4} \\ \frac{\partial f_y}{\partial q_1} & \frac{\partial f_y}{\partial q_2} & \frac{\partial f_y}{\partial q_3} & \frac{\partial f_y}{\partial q_4} \\ \frac{\partial f_z}{\partial q_1} & \frac{\partial f_z}{\partial q_2} & \frac{\partial f_z}{\partial q_3} & \frac{\partial f_z}{\partial q_4} \end{bmatrix}$$

En théorie cette jacobienne peut être calculée analytiquement, cependant elle sera calculée numériquement (par la méthode conventionnelle des accroissements finis) à chaque étape de calcul. La fonction jacobi effectue cela dans le fichier kinematics.py.

Voir la fonction ik_num_base dans le fichier annexe kinematics.py pour l'implémentation en python de cet algorithme.

BIBLIOGRAPHIE

- **Wikipédia.** *Inverse Kinematic.* https://en.wikipedia.org/wiki/Inverse_kinematics, 2015-07-28.
- **Wikipédia.** *Moore-Penrose pseudoinverse.* https://en.wikipedia.org/wiki/Moore-Penrose_pseudoinverse, 2015-07-28.
- **Wikipédia.** *Jacobian matrix and determinant.* https://en.wikipedia.org/wiki/Jacobian_matrix_and_determinant, 2015-07-28.
- **Wikipédia.** *Jacobian inverse technic.* https://en.wikipedia.org/wiki/Inverse_kinematics#The_Jacobian_inverse_technique, 2015-07-28.

ANNEXES

```

# -*- coding: utf-8 -*-
"""
Created on Mon Aug  3 12:42:24 2015

@author: Xavier MARIOT
@contact: xavier.mariot@gadz.org
"""

import numpy as np
import math

"""
    --- Description jacobian ---

jacobian calcul la jacobienne de la fonction vectorielle f au point q.
nb_q est le nombre de paramètres que prend f c.à.d. la longueur de la liste q
On peut régler le pas utilisé pour calculer la dérivée grâce au paramètre h.
La dérivée est calculée par la méthode des accroissements finis.
jacobian retourne une matrice de type numpy.matrix
    --- Fin description ---
"""

def jacobian(f,nb_q,q,h=0.001):
    """
        === Description des arguments ===
    f      --->  Fonction de cinématique directe qui retourne un numpy.array
    nb_q   --->  'int' qui indique le nombre de paramètres de
                  f contenu dans q
    q      --->  valeurs numériques du point auquel est calculé la
                  jacobienne de f, q est au format numpy.array
    h      --->  Valeur du pas de calcul de la dérivée avec la formule
                  des accroissements finis

        === Fin description ===
    """

    # On vérifie que q est bien un numpy.array sous forme
    # de liste et non de tableau
    if len(q) != nb_q:
        return False

    # On définit le vecteur infinitésimal dh qui vont servir au calcul de J
    dh = np.array([])
    for i in range(nb_q) :
        dh = np.append(dh,0)

    # On calcule la jacobienne colonne par colonne
    # c.à.d. chaque vecteur df/dqi en q
    # NB : pour des raisons pratiques on enregistre ces vecteurs en ligne,
    # on transpose ensuite pour obtenir j
    j = []
    for i in range(nb_q):
        dh[i] = h # On donne la valeur dh à la ième composante du vecteur dh

        # On calcule df/dqi par la formule des accroissements finis
        j.append(1/h*(f(q+dh)-f(q)))
        dh[i] = 0 # On remet la ième composante de dh à zéro pour la suite

    j = np.matrix(j) # On transforme j en matrice, pour l'instant on a en

```

```

# stocké la transposée de la jacobienne de f en q
return j.T

"""
    === Description ik_num_base ===

Cette fonction est l'algorithme de cinématique inverse pour des petits
déplacements autour d'une position initiale. La position initiale est donnée
par le paramètre qi.
Le petit déplacement par rapport à la position xi=f(qi) est noté dx.
La position visée vaut xf = xi + dx

"""

def ik_num_base(f, nb_q, qi, dx, it=1, imax=10):
    """
        === Descriptions des arguments ===

        f      --> 'arg: nd.array 1D; return: nd.array 1D' fonction
                  vectorielle représentative du modèle de cinématique directe
        nb_q   --> 'int' qui représente le nombre de paramètres d'entrée
                  que prend la fonction f, c.à.d. la taille de la liste q.
        qi     --> 'numpy.array 1D' position initiale du système
        dx     --> 'numpy.array 1D' déplacement infinitésimal du système
                  par rapport à sa position initiale
        it     --> 'float' tolérance admise entre le résultat retourné et
                  la position finale souhaitée (norme de la différence)
        imax   --> 'int' nombre maximal d'itérations à partir duquel
                  l'algorithme de Jacobi s'arrête.

        === Fin de la description ===
    """

    """
        --- Initialisation ---
        q = qi # on initialise la valeur des paramètres q
        x = f(q) # on enregistre la position initiale du système

        xf = x + dx # on calcule la position finale à atteindre
        dx = dx
        # on initialise la variation à effectuer pour se rapprocher
        # de la position finale
    """

    """
        --- Recurrence ---
        for i in range(imax):
            # Calcul du nouveau dx au format 'numpy.array 1D'
            dx = xf-x

            # vérification de la condition d'arrêt
            if np.linalg.norm(dx) <= it :
                break
            else:
                # mise en forme de dx à l'itération considérée
                dx = np.matrix(dx)
                dx = dx.T # dx est maintenant un vecteur colonne qui va pouvoir
                # être multiplié par une matrice
    """

```

```

# Calcul numérique de la jacobienne j au point considéré
j = jacobian(f,nb_q,q,1e-6)

# incrémentation de q en gardant le type 'numpy.array 1D'
# NB : j.I retourne la pseudo inverse de j
q = q + (j.I*dx).T.getA()[0]

# mise à jour de x
x = f(q)

# si la condition d'arrêt n'est pas vérifiée,
# on itère le calcul jusqu'à imax fois

# A la sortie de la méthode de jacobi, q vérifie ||xfinal - f(q)|| <= it
return q

"""
    --- Description ik_num ---

ik_num pour 'inverse kinematic numeric' est la fonction de cinématique inverse
numérique qui retourne la liste de valeurs des nb_q paramètres d'entrée de f
qui permettent d'obtenir la position xfinal en sortie.

    --- Fin de la description ---
"""

def ik_num(f, nb_q, map_qinit, xfinal, it=1, imax=10):
    """
        === Descriptions des arguments ===

        f          ---> 'arg: nd.array 1D; return: nd.array 1D' fonction
                    vectorielle représentative du modèle de cinématique direct
        nb_q       ---> 'int' qui représente le nombre de paramètres d'entrée
                        que prend la fonction f, c.à.d. la taille de la liste q.
        map_qinit  ---> 'numpy.array 2D' mapping de l'ensemble des positions
                        initiales utilisées pour démarrer l'algorithme
        xfinal     ---> 'numpy.array 1D' résultat final visé pour le modèle de
                        cinématique directe
        it         ---> 'float' tolérance admise entre le résultat retourné
                        et la position finale souhaitée (norme de la différence)
        imax      ---> 'int' nombre maximal d'itérations à partir duquel
                        l'algorithme s'arrête.

        === Fin de la description ===
    """

    """
        détermination du mapping xinit """
    map_xinit = map(f,map_qinit)

    """
        détermination du point de départ de l'algorithme """
    # index de la position x qui engendre une distance minimale ||x-xfinal||
    imin = 0
    dxmin = np.linalg.norm(xfinal-map_xinit[0])

```

```

# On parcourt l'ensemble des position du mapping
for i in range(len(map_xinit)):
    dx = np.linalg.norm(xfinal-map_xinit[i])

    # On enregistre l'index minimal et la distance minimale si on trouve
    # une distance inférieure à celles enregistrées précédemment.
    if dx < dxmin :
        dxmin = dx
        imin = i

# On fixe la position initiale à partir de la plus petite distance trouvée.
q = map_qinit[imin]
x = map_xinit[imin]

""" Décoposition du trajet en plusieurs petits déplacements """
delta_x = xfinal-x # calcul du vecteur déplacement total.

# L'unité de découpage est arbitrairement 5*it
nb_div = int(np.floor(np.linalg.norm(delta_x)/(10*it)) + 1)

if nb_div > 1:
    dx = (1/float(nb_div))*delta_x

xf=[] # xf sera la liste des points intermédiaires à atteindre.
for i in range(nb_div + 1): # on crée la liste des points intermédiaires
    xf.append(x + i*dx)

# On s'assure que la dernière valeur ne sera pas altérée par
# des erreurs de calcul
xf[-1] = xfinal
xf.pop(0) # On retire la position initiale

""" Application de la méthode de jacobi pour chaque trajet intermédiaire"""
for xi in xf:
    # le petit déplacement vaut la différence entre la position finale
    # intermédiaire visée et la position théorique actuelle
    dx = xi-f(q)

    # Calcul de la nouvelle valeur de q par la méthode de Jacobi
    # pour les petits déplacements
    q = ik_num_base(f, nb_q, q, dx, it, imax)

# A la sortie de la méthode de jacobi, q vérifie ||xfinal - f(q)|| <= it
return q

""" --- Définition de dk_l_hand --- """

dk_l_hand pour 'direct kinematic left hand' représente la fonction de
cinématique directe qui retourne sous la forme d'un 'numpy.array 1D' la
position x,y,z de la main gauche de poppy avec pour origine (0,0,0) le centre

```

du servomoteur de l'épaule (l_shoulder_y) cf. schema de la doc elle prend comme argument un 'numpy.array 1D' noté q qui contient les angles des 4 servomoteurs du bras gauche dans l'ordre suivant :
 $q = [l_shoulder_y, l_shoulder_x, l_arm_z, l_elbow_y]$.

Les zéros de chaque angles et les sens positifs sont définis à partir de la config par défaut de poppy.
 voir schéma cinématique du bras gauche dans la doc pour leur représentation.
 voir 'cinematique directe.nb' pour le calcul de la formule de x, y et z

"""

```
def dk_l_hand(q=np.array([0,0,0,0])):
    ab = 30.
    bc = 45.
    cd = 105.
    de = 120. + 90./2.
    # 120 mm pour l'avant bras et 45 pour la moitiée de la main

    x = -bc*np.cos(q[1])*np.sin(q[0]) - cd*np.cos(q[1])*np.sin(q[0]) - \
        de*(np.cos(q[1])*np.cos(q[3])*np.sin(q[0]) + (np.cos(q[0])*np.cos(q[2]) + \
        np.sin(q[0])*np.sin(q[1])*np.sin(q[2]))*np.sin(q[3]))

    y = ab + bc*np.sin(q[1]) + cd*np.sin(q[1]) - \
        de*(-np.cos(q[3])*np.sin(q[1]) + np.cos(q[1])*np.sin(q[2])*np.sin(q[3]))

    z = -bc*np.cos(q[0])*np.cos(q[1]) - cd*np.cos(q[0])*np.cos(q[1]) - \
        de*(np.cos(q[0])*np.cos(q[1])*np.cos(q[3]) + (-np.cos(q[2])*np.sin(q[0]) + \
        np.cos(q[0])*np.sin(q[1])*np.sin(q[2]))*np.sin(q[3]))

    return np.array([x,y,z])

mapping_dk_l_hand = np.array([
    map(math.radians, [-60, 30, -60, -100]),
    map(math.radians, [-120, 20, -45, -70]),
    map(math.radians, [15, 10, -20, -30]),
    map(math.radians, [-45, 45, -80, -40]),
    map(math.radians, [-80, 70, -80, -60]),
    map(math.radians, [-120, 45, -65, -30])
])

def ik_num_l_hand(x, it=1, imax=10):
    ab = 30.
    bc = 45.
    cd = 105.
    de = 120. + 90./2.
    # 120 mm pour l'avant bras et 45 pour la moitiée de la main

    xf = x
    # On vérifie que la consigne demandée est réalisable.
    if np.linalg.norm(xf-np.array([0,ab,0])) > (bc+cd+de-5):
        # si non, on la met au plus proche
        xf = (bc+cd+de-5)/np.linalg.norm(xf)*xf

    return ik_num(dk_l_hand, 4, mapping_dk_l_hand, xf, it, imax)
```

```

if __name__=='__main__':
    print "--- Module test de la cinématique inverse de Poppy --- \n\n"
    while True:
        print "Pour lancer la simulation, ouvrir V-rep, \
\ncpuis appuyer sur o<enter> pour lancer la simulation \
\nAppuyer sur q<enter> à tout moment pour quitter.\n"
        s=raw_input("==>")
        if s == 'o':
            #Simulation V-rep
            print "\n--- Début de la simulation --- \n"
            # Import des bibliothèques nécessaires
            import pypot
            from poppy.creatures import PoppyHumanoid
            import time

            # démarre la simulation poppy dans vrep.
            # Il faut lancer V-rep avant d'executer le code.
            poppy = PoppyHumanoid(simulator='vrep')

            # Mise en position initiale de Poppy
            poppy.l_shoulder_y.goal_position = 20.
            poppy.l_shoulder_x.goal_position = 30.
            poppy.l_arm_z.goal_position = -20
            poppy.l_elbow_y.goal_position = -50

            print "Poppy se met en position, patientez s.v.p. (~4s)\n"
            time.sleep(4)

            # Boucle de pilotage
            while True:
                q = np.array([0.,0.,0.,0.])

                # On récupère la position actuelle de poppy dans V-rep
                q[0] = math.radians(poppy.l_shoulder_y.present_position)
                q[1] = math.radians(poppy.l_shoulder_x.present_position)
                q[2] = math.radians(poppy.l_arm_z.present_position)
                q[3] = math.radians(poppy.l_elbow_y.present_position)

                print "q initial = ",q

                x = dk_l_hand(q)
                print "x initial = ",x,"\\n"

                dx = np.array([0.,0.,0.])
                print "Entrer une chaine de caractère, chaque caractère aura \"\
+ l'effet suivant :\n\
e ---> Avance le bras de 5mm\\n\
d ---> Recule le bras de 5mm\\n\
s ---> Bouge le bras vers la gauche de 5mm\\n\
r ---> Bouge le bras vers la droite de 5mm\\n\
t ---> Torsion du bras de 5deg\\n\
q ---> Quitter le programme\\n"

```

```

f ---> Bouge le bras vers la droite de 5mm\n\
z ---> Monte la main de 5mm \n\
r ---> Descend le bras de 5mm \n\
q ---> Quitte le programme\n\n\
NB : Les caractères peuvent être entrés plusieurs fois pour \
additionner les effets.

s=raw_input("==>")

# Gestion des entrées clavier
for c in s :
    if c == 'e':
        dx[0] += 5
    if c == 'd':
        dx[0] += -5
    if c == 's':
        dx[1] += 5
    if c == 'f':
        dx[1] += -5
    if c == 'z':
        dx[2] += 5
    if c == 'r':
        dx[2] += -5
    if c == 'q':
        break
print "dx = ",dx

# calcul de la cinématique inverse
q = ik_num_1_hand(x+dx)

# Pilotage de poppy (simple)
poppy.l_shoulder_y.goal_position = math.degrees(q[0])
poppy.l_shoulder_x.goal_position = math.degrees(q[1])
poppy.l_arm_z.goal_position = math.degrees(q[2])
poppy.l_elbow_y.goal_position = math.degrees(q[3])

# On Laisse le temps à poppy d'effectuer le déplacement
# avant de relancer la boucle
print "Temporisation de 4s pour laisser poppy bouger"
time.sleep(4)

if c == 'q':
    break

else :
    print "L'entrée ne correspond à aucune action.\n"

# Arrêter complètement la simulation poppy
poppy.stop_simulation()
pypot.vrep.close_all_connections()

break

elif s == 'q':
    break

```

```
else :  
    print "L'entrée ne correspond à aucune action."  
  
print "\n----- Fin du programme -----"
```