Promotion 2002

JARDÉ
Sébastien

Mémoire de stage de 3ème année
Diplôme d'Ingénieurs de l'E.N.S.P.S.

Mise en œuvre de commandes avancées sur le robot BIP

INRIA Rhônes-Alpes
Équipe de projet BIP
ZIRST - 655 avenue de l'Europe
38330 Montbonnot

Mars-Août 2002
Tuteur de stage :
Pierre-Brice WIEBER

Remerciements

Je tiens vivement à remercier B. ESPIAU directeur de l'INRIA Rhône-Alpes de m'avoir accueilli et d'avoir mis à ma disposition tous les moyens nécessaires au bon déroulement de ce projet de fin d'étude.

Je remercie également P.B Wieber qui a suivi et encadré ce travail. Je lui suis profondément reconnaissant pour le temps qu'il y a consacré ainsi que pour ses conseils, sa rigueur et sa confiance.

Je remercie également toute l'équipe BIP et l'ensemble des membres des Moyens Robotiques, notamment E. Rutten, A. Giraud, O. Testa, G. Baille, J.F Cuniberto et S. Arias pour leur accueil , leur gentillesse et leur disponibilité.

Enfin, je remercie pour leur sympathie et leur bonne humeur mes compagnons de stage Thibault, Leonard, Mathieu, Ludovic, Cécile et Hamoudi.

Mise en œuvre de Commandes avancées sur le robot BIP

Résumé

L'INRIA Rhône-Alpes et le Laboratoire de Mécanique des Solides de l'université de Poitiers ont développé dans le cadre d'un projet commun, un robot anthropomorphe à quinze degrés de liberté capable de marcher dynamiquement sur sol plat et d'assurer la montée et la descente d'escaliers.

P.B Wieber, chercheur de l'équipe du projet BIP, a proposé une loi de commande par l'approche de fonctions de tâche. On se fixe pour objectif d'implanter cette loi de commande sur la plateforme expérimentale. Mais en l'état actuel, l'architecture matérielle et logicielle du calculateur du robot ne permet pas de faire face à la quantité de calcul imposée par la loi de commande.

La première partie de ce rapport a pour objectif d'évaluer, en terme de temps CPU, les procédures de calcul relatives à la loi de commande à mettre en œuvre. Elle décrit ensuite un nouveau changement de variable élaboré sur le simulateur du robot, nous pemettant de réduire les temps de calcul de la loi de commande. Enfin, elle décrit l'observateur développé et testé en simulation, dans le but de reconstruire à tout instant la position et l'orientation du robot dans l'espace.

La seconde partie propose tout d'abord une étude qui vise à déterminer les modifications à entreprendre, au niveau de l'architecture matérielle et logicielle du calculateur embarqué, pour l'implantation de la nouvelle loi de commande. On décide alors de déporter le calcul de celle-ci sur un serveur distant en mode connecté avec le calculateur du robot actuel. La suite décrit le développement de l'application client/serveur correspondante.

La fin de ce document est consacrée aux manipulations expérimentales effectuées sur le robot. Ces manipulations ont permis de valider différents suivis de trajectoire et le bon fonctionnement général du robot, mais elles mettent aussi en évidence des problèmes liés aux moteurs de la cheville droite, du genou gauche et de la hanche gauche.

Implementation of advanced control laws on the BIP robot

Abstract

INRIA Rhône-Alpes and LMS have developed in a common project, an anthropomorphic walking robot with fifteen degrees of freedom which is able to walk dynamicaly on flat ground and to pass over upstairs.

P.B Wieber, a member of the BIP project, has proposed a control law through the task functions approach. We have decided to insert this in the computer of the biped. But, hardware and software were not adapted to compute it.

The first part of tis report estimates, in CPU time, the computations of the control law. Then, it describes an adaptation of the task function developed on the BIP simulator ; It allows to reduce the computation time. Eventually, it describes the observer developed and tested on the simulator in order to rebuild position and orientation of the biped in the space.

The second part proposes firstly a study which gives us the hardware and software possiblities to insert the new control law in the biped computer. We have decided to deport the computation of the control law on a distant server connected to the present robot computer. The following describes the developement of client/server applications.

The end of this document is dedicated to the experimentations performed on the biped. These experimentations have enabled to validate different motions and the good functionning of the biped. But they have shown problems on the motors of the right ankle, the left knee and the left hip.

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction

1.1 Présetation de l'INRIA

Crée en 1967 à Rocquencourt près de Paris, l'Institut National de Recherche en Informatique et Automatique (INRIA), est un établissement public à caractère scientifique et technologique (EPST) placé sous la double tutelle du ministère de la recherche et du ministère de l'économie, des finances et de l'industrie. Cette institut mène des recherches avancées dans le domaine des sciences et technologies de l'information et de la communication. Ce domaine inclut l'informatique et l'automatique, mais aussi les télécommunications et le multimédia, la robotique, le traitement du signal et le calcul scientifique. L'INRIA est décentralisée en 5 unités de recherche en France.

1.1.1 Quelques chiffres

Les chiffres énoncés datent de décembre 2000.

- Resources budgétaires
  - Dotation de l'état : 442 MF HT
  - Resources propres : 174 MF HT
- Resources humaines
  - 724 titulaires INRIA
  - 256 post-doctorants et Contractuels
  - 550 doctorants
  - 230 chercheurs et enseignants d'autres organismes
  - 430 conseillers, collaborateurs divers et invités
- Indicateurs
  - Contrats de recettes actifs : plus de 600
  - Contrats de recettes signés dans l'année : plus de 200
  - Une cinquantaine de sociétés sont issue de l'INRIA
  - 7 brevets déposés en 1999.

1.1.2 L'INRIA Rhône-Alpes

L'INRIA Rhône-Alpes est une des cinq entités de l'INRIA. Crée en 1992, c'est la plus récente des cinq unités existentes. Plus de 330 personnes dont 220 chercheurs travaillent sur le site de Montbonnot. Cette unité accueille également 130 doctorants, ingénieurs et stagières.

Elle mène ses activités en étroite collaboration avec les laboratoires de recherche publics et privés, nationaux et internationaux, et elle entretient des liens privilégiés avec l'institut d'Informatique et Mathématiques Appliquées de Grenoble (IMAG). Ces activités sont organisées autour de quatre pôles de recherche divisés en plusieurs équipes autonomes :

1. Maîtriser les systèmes et réseaux informatiques (Réseaux, parallélisme et systèmes répartis).
   Équipes : APACHE, ARENAIRE, ARES, COMPSYS, PLANETE, ReMaP, RESO, SARDES, TRIO, VASY.
2. Aider à la conception et à la création (Bases de connaissances, documents multimédia, modèles cognitifs).
   Équipes : EXMO, HELIX, I3D, OPERA
3. Percevoir, simuler et agir (Synthèse d'images, réalité virtuelle, vision par ordinateur et robotique).
   Équipes : iMAGIS, MOVI, SHARP.
4. Modéliser les phénomènes complexes (Automatique, simulation et calcul scientifique).
   Équipes : BIP, IDOPT, IS2, NUMOPT, OPALE.

1.2 Le projet BIP

Dans la classe des systèmes mobiles, les robots marcheurs, par exemple hexapodes, présentent des avantages certains sur leurs homologues à roues dès que le sol n'est plus plan ou libre : le franchissement des obstacles est plus aisé, l'emprise au sol plus faible, l'adaptabilité meilleure. Ceci concerne les grands domaines de la robotique non-manufacturière : exploration, maintenance, intervention, service. Cependant, dès lors que l'environnement de travail du système est conçu pour l'homme, la technologie multipode doit en général laisser place à la bipédie si l'on désire conserver de bonne possibilités de déplacement et d'action sans modifier l'environnement. D'où l'interêt assez récent (quelques années) que porte la communauté mondiale de Recherche et Développement en robotique aux systèmes dits humanoïdes, destinés à accompagner l'homme dans certaines de ces activités personnelles ou professionnelles. Par exemple, une certaine forme d'assistance à domicile de personnes à mobilité réduite, pour des tâches très routinières, pourrait être assurée par des robots bipèdes, car ceux-ci ont la faculté de se déplacer sans adaptation particulière de l'environnement. Si la faisabilité de tels systèmes reste largement hors d'atteinte pour ce qui est de robots aux capacités d'autonomie décisionnelle élevées, le niveau actuel de la technologie permet par contre d'envisager

la réalisation de machines capables de se déplacer en marchant dans des conditions bien déterminées et d'exécuter quelques actions très simple.

Depuis 1994 et sous l'impulsion de Bernard Espiau, Directeur de Recherche à l'INRIA et à l'origine du projet BIP, l'INRIA Rhônes-Alpes et le Laboratoire de Mécanique des solides (LMS) de l'Université de Poitiers ont développé dans le cadre d'un projet commun, un robot anthropomorphe à quinze degrés de liberté. Le LMS a pris en charge la conception mécanique du prototype, le service Moyens Robotiques a réalisé pour sa part la partie electronique et informatique et l'équipe de recherche BIP a eu la responsabilité de la partie contrôle-commande du système. Ce travail commun a permis d'aboutir à une plate-forme expérimentale robuste pour l'étude de la marche bipède robotisée.

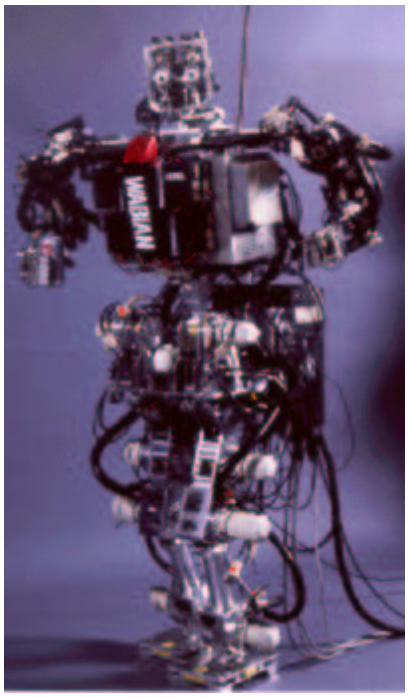
L'objectif, ici, n'est pas de reproduire la morphologie des jambes humaines qui est une machine extraordinaire comprenant quelques 55 os, 90 muscles et 16 nerfs dont une réalisation mécanique précise peut largement dépasser la technologie courante. Le but est plutôt d'appliquer le comportement anthropormorphe à une version simplifiée des jambes.

1.3 Etat de l'art

Les robots Marcheurs font l'objet d'études sérieuses depuis le milieu des années 70. Deux décénies de recherche intense (principalement au Japon et aux Etats-Unis) ont permis de voir naître une multitude de robots ayant des loi de commande et des capacités fonctionnelles différentes. En même temps une attention particulière fut concentrée sur la compréhension des mécanismes de locomotion. Mais par rapport aux robots quadrupèdes, hexapodes ou octopodes, les robots bipèdes ont longtemps été beaucoup plus rares, car ils sont d'une mise en œuvre plus difficile du fait d'une plus grande instabilité. Si les projets de robots bipèdes se multiplient aujourd'hui (HONDA, SONY, KAWADA ...), seuls quelques-uns atteignent toutefois un degré d'achèvement satisfaisant, tant du point de vue de la conception mécanique que de la mise au point de lois de commandes adaptées (figure 1.1).

On distinguent plus particulièrement, les laboratoires Humanoid Robotics Institute de la Waseda University, et le Leg Laboratory du Massachusetts Institute of Technology dans lesquels de nombreuses générations de robots marcheurs ont déjà été développées. Leurs tous derniers prototypes, WABIAN et M2 (figure 1.1), bénéficient de leur longue expérience et font incontestablement partie des projets les plus aboutis. La firme HONDA a su également créer l'événement en présentant après plus de dix ans de recherches tenues secrètes une série de prototypes, dont le dernier modèle, ASIMO (figure 1.1), représente sans aucun doute et à tous points de vue une des réalisations les plus aboutie en la matière.

Le robot BIP, développé conjointement par le LMS et l'INRIA Rhône-Alpes, se place d'emblée comme un concurrent sérieux de par la qualité de sa conception mécanique et de sa réalisation, mais la mise au point de lois de commande adaptées ne fait que commencer.



Waseda University : WABIAN

MIT : M2

HONDA : ASIMO

SONY : SDR 4X

FIG. 1.1 – Quelques-uns des projets les plus aboutis.

1.4 Introduction à la marche

La marche est un mode de locomotion pendant lequel le sujet possède toujours au moins un pied au sol, ce qui donne naissance à une alternance de phases de *simple support* (un seul pied au sol) et de phases de *double support* (deux pieds au sol). Cette alternance n'implique pas pour autant qu'un mouvement de marche soit répétitif, et l'on peut même noter qu'il l'est d'autant moins que le terrain est accidenté.

Si l'on observe un mouvement de marche dans trois plans différents de l'espace (figure 1.2), le déplacement du sujet a lieu principalement dans le plan sagittal, mais de nombreux mouvements apparaissent également dans chacun des autres plans, déhanchement, déplacement latéral et rotation du bassin, qui ne doivent pas être négligés car ils améliorent considérablement la fluidité du mouvement.

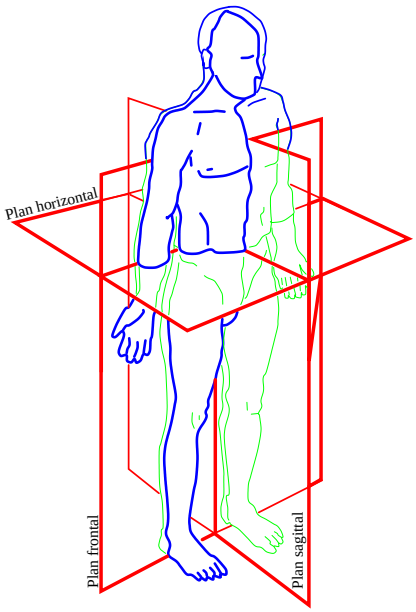


FIG. 1.2 – Plans de coupe du corps humain

1.5 Présentation du sujet de stage

Ce stage a pour but la "Mise en œuvre de commandes avancées sur le robot BIP".
Voici la définition du sujet proposée par P.B Wieber :

Un certain nombre de lois de commande ont été proposées à ce jour pour permettre d'assurer l'équilibre du robot BIP pendant des mouvements de marche dynamique 3D. Les principales approches développées par l'équipe BIP sont la commande prédictive et la fonction de tâche [15]. C'est en passant par la mise en œuvre de l'approche par fonction de tâche que ce stage se fixe pour objectif de mener à bien des expériences de marche dynamique 3D stable sur sol plat avec le robot BIP. Du fait de la complexité de cette plate-forme expérimentale, mettant en jeu 21 degrés de liberté interagissant de façon fortement non-linéaire, la mise en œuvre de ce type de loi de commande implique de pouvoir réaliser en temps-réel une quantité de calcul très importante. En l'état actuel, l'architecture matérielle et logicielle du calculateur du robot ne permet pas d'y faire face : à côté de simplifications éventuelles de ces calculs dont l'influence reste à évaluer, notamment en terme de temps CPU, il est donc nécessaire de remodeler en profondeur l'architecture du calculateur, pour en arriver très probablement à un déport d'une partie de la puissance de calcul. En conséquence, une étude des possibilités hardwares et softwares qui s'offrent à nous s'avère nécessaire, dans le but de définir au mieux les modifications qui nous permettront de faire face au calcul de la loi de commande pour une marche dynamique stable. Une implémentation stable et robuste (au sens informatique de ces termes) d'une loi de commande par fonction de tâche devrait alors fournir la dernière brique nécessaire à la mise en place d'expérimentations de mouvements dynamiques 3D, la marche sur sol plat étant l'objectif principal au terme de ce stage.

1.6 Organisation du travail

Le travail effectué durant ce stage, d'une durée de six mois, regroupe plusieurs disciplines. Il s'organise en deux parties :

1. *Études et Développements sur le modèle du robot BIP*
2. *Études et Manipulations sur la plateforme expérimentale*

La première partie du stage a essentiellement porté sur l'étude et la modélisation du robot BIP. En parallèle, l'étude et le développement du simulateur m'ont permis d'appréhender la loi de commande à implanter sur le robot BIP. Ensuite, mon travail s'est orienté vers la plateforme expérimentale avec la prise en main du matériel informatique (le contrôleur de robot ORCCAD) et le robot proprement dit. Une fois ces fondamentaux maitrisés, la réalisation d'un audit concernant le choix d'un calculateur a pu être menée à bien. La fin de ce stage concerna essentiellement les tests expérimentaux sur le bipède. Le planning donné figure (1.3) décrit chronologiquement l'évolution de mon travail. Voici la description des briques numérotées de un à six :

1. Développement d'un changement de variable $Q(q)$, sur le simulateur du robot. Celui-ci exprime l'ensemble des coordonnées articulaires du robot dans un espace de sortie choisi, afin de découpler les variables articulaires q_1 et q_2 (position et orientation) dans l'expression de la loi de commande.

2. Construction et test d'un observateur de q_2 sur simulateur. Ceci permet de reconstruire à chaque pas d'échantillonnage la position et l'orientation du robot dans l'espace.

3. Evaluation des temps de calcul CPU des différentes lois de commande élaborées pour différents calculateurs et notamment pour le calculateur embarqué du robot.

4. Audit mené auprès des Moyens Informatiques (MI) et Robotiques (MR) de l'INRIA dans le but d'évaluer les possibilités qui s'offre à nous pour le changement du calculateur du robot, puis rédaction de celui-ci.

5. Etude et Développement d'application client/serveur.

6. Développement d'une application nous permettant d'effectuer le calcul de la loi de commande à chaque pas d'échantillonnage sur un serveur distant et de renvoyer les consignes de couple au calculateur du robot.

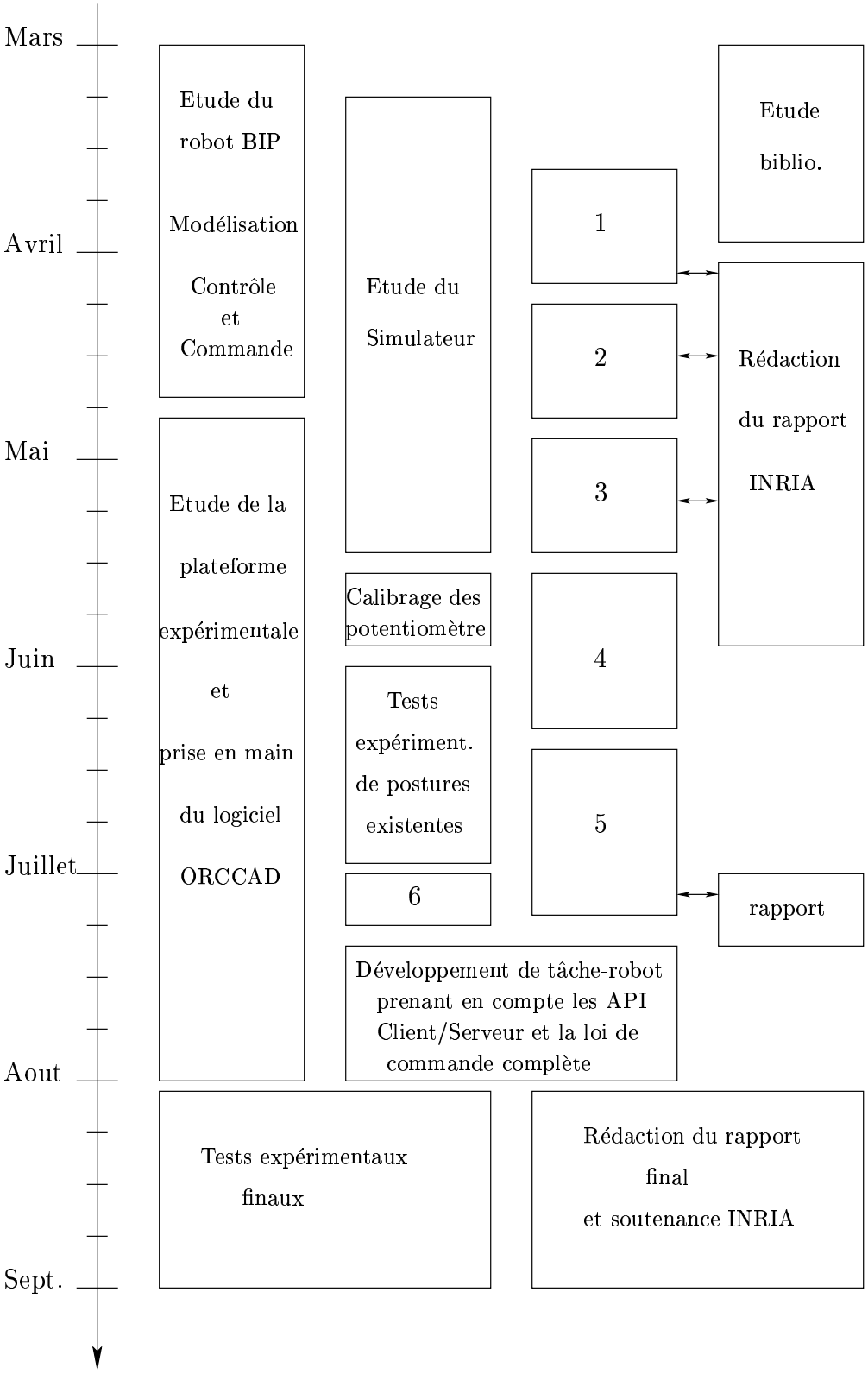


FIG. 1.3 – Organisation du travail.

Première partie

Étude et Développement Théorique

Chapitre 2

Le Robot BIP

2.1 Présentation générale

L'objectif initial du projet BIP fut la réalisation d'un robot bipède à 17 degrés de liberté (ddl), aux dimensions et masses des membres inférieurs voisines de celles de l'homme [16], [21] et [22]. Ce robot a été conçu pour reproduire une marche dynamique 3D stable sur sol plat et assurer la montée et la descente d'escaliers.

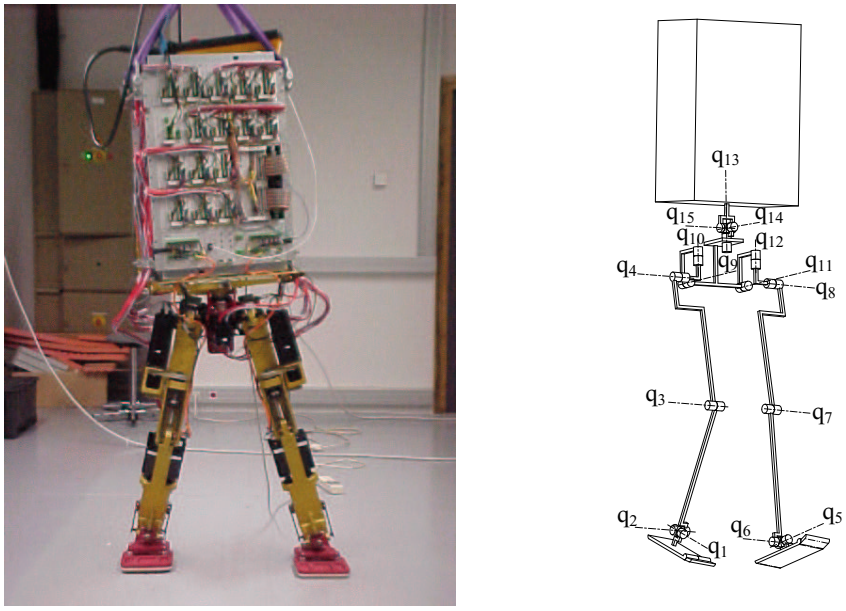


FIG. 2.1 – Le robot bipède BIP et ses degrés de liberté

Dans sa première version (Juillet 1999) le système ne comportait que deux jambes (8 ddl) associées à un chariot à quatre roues (6 ddl plan). Le prototype suivant (Janvier 2000) vit l'apparition d'un tronc intégrant l'électronique de commande. Cette dernière version totalise ainsi 15 ddl dont une rotule complète (3 ddl) au niveau du tronc (Cf figure 2.1). Il mesure 180 cm pour 105 kg.

Le robot peut se déplacer dans le plan grâce à la rotation de ses chevilles, genoux et hanches qui permettent la flexion/extension dans le plan sagittal (figure 2.2).

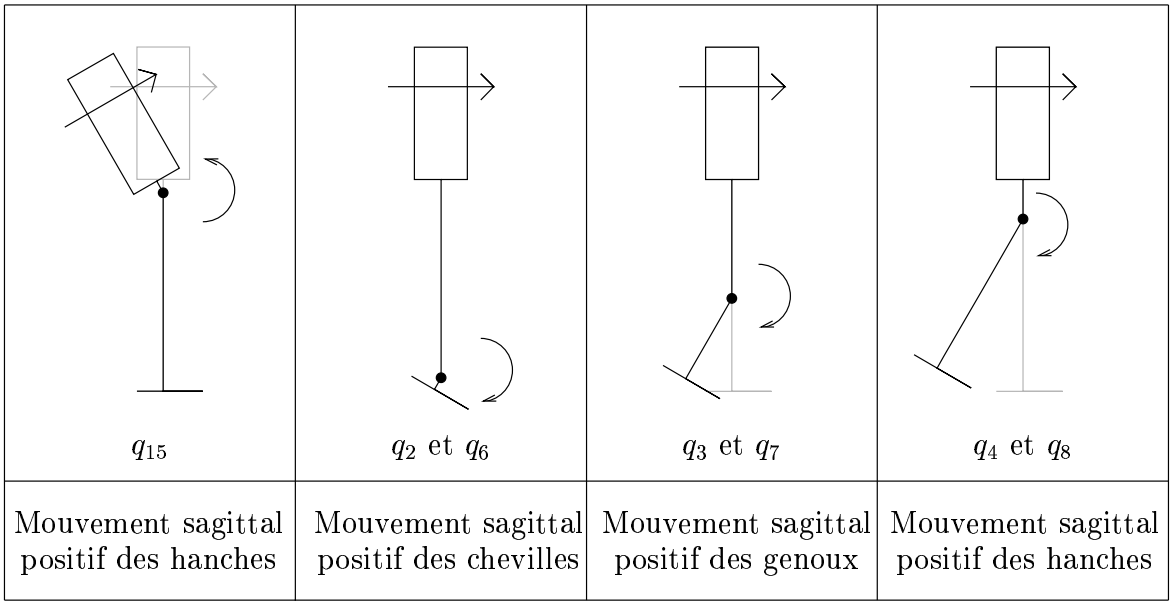


FIG. 2.2 – Mouvements du robot dans le plan sagittal

Les changements de direction sont possible par la rotation interne/externe du tronc, du bassin et des hanches (figure 2.3). Les rotation des chevilles, des hanches et des vertèbres lombaires permettent l'abduction/adduction dans le plan frontal (figure 2.4) et maintiennent ainsi l'équilibre latéral du robot. Un degré de liberté supplémentaire entre le tronc et le bassin rend les systèmes de déplacement et d'équilibre indépendants.

Le lecteur pourra se reporter à l'annexe C pour y consulter les différentes conventions utilisées pour décrire les mouvements, les articulations et le moteurs du robot.

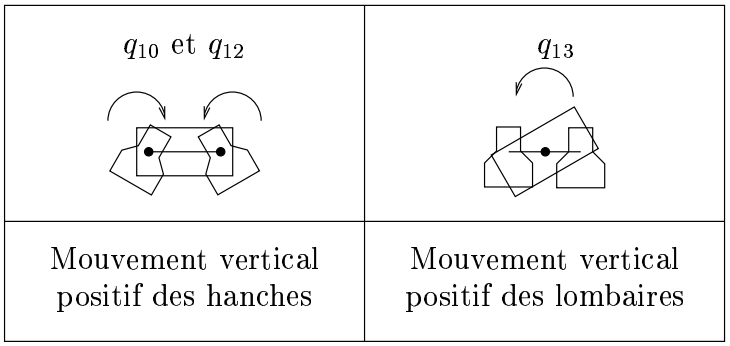


FIG. 2.3 – Mouvements du robot dans le plan horizontal

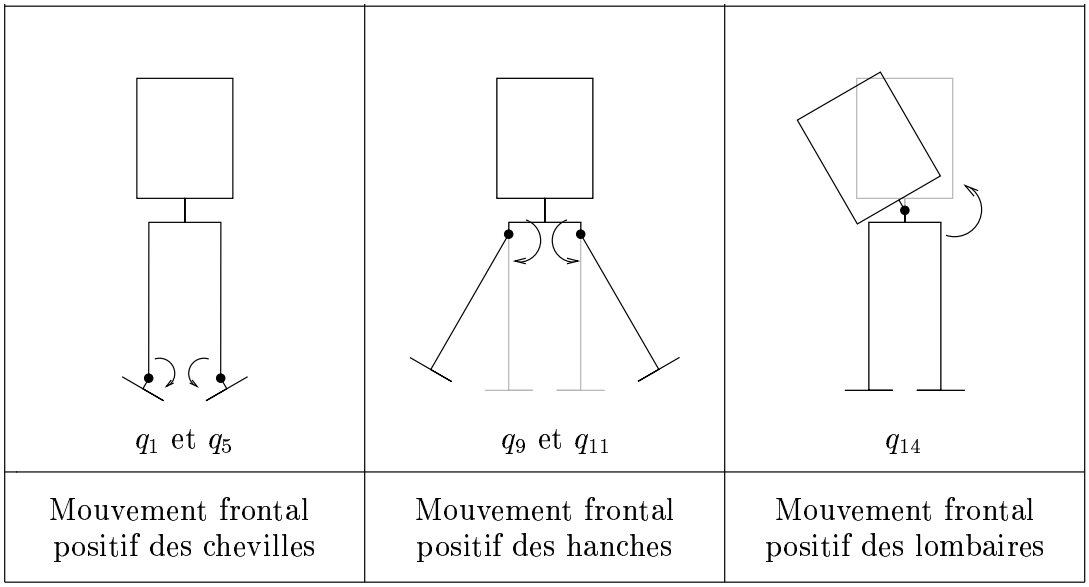


FIG. 2.4 – Mouvements du robot dans le plan frontal

2.2 Modélisation du système

Pour étudier les mouvements d'un robot, il est indispensable de pouvoir décrire la position dans laquelle il se trouve à chaque instant. Un robot est constitué d'un ensemble de solides, et décrire sa position nous amène à détailler la position et l'orientation de chacun d'eux. Mais pour décrire entièrement la position d'un bipède il faut ajouter la notion de *position* et d'*orientation* de celui-ci dans l'espace. La posture décrit ainsi la disposition des solides les uns par rapport aux autres, tandis que la position et l'orientation permettent de les situer correctement par rapport à l'environnement.

Les articulations du robot BIP ne permettent que des rotations autour de certains axes (figure 2.1). Ces articulations conditionnent donc la disposition des solides les uns par rapport aux autres, et il suffit de répertorier les positions dans lesquelles elles se trouvent pour pouvoir décrire complètement la posture du robot.

Une fois la posture du robot déterminée, il suffit de connaître la position et l'orientation de l'un de ses solides pour pouvoir en déduire la position et l'orientation du robot au complet. On peut alors réunir l'ensemble des positions articulaires ainsi que la position et l'orientation de ce solide pour constituer un vecteur $q \in \mathbb{R}^n$ qui décrit entièrement la position du robot.

Le robot BIP compte actuellement 15 degrés de liberté. Sa posture est donc déterminée par un vecteur q de dimension 15 auquel il faut ajouter 6 variables suplémentaires pour exprimer la position et l'orientation du robot dans l'espace. On obtient ainsi un système totalisant 21 degrés de liberté et pour lequel un vecteur q de dimension 21 décrit totalement sa position.

2.2.1 La formalisation de Khalil-Kleinfinger

La formalisation dite de Khalil-Kleinfinger [10] est une modification de la représentation de Denavit-Hartenberg afin de faciliter son application aux chaînes cinématiques fermées. Ce paramétrage décrit le passage du repère $i-1$ au repère i avec les conventions de la figure 2.5.

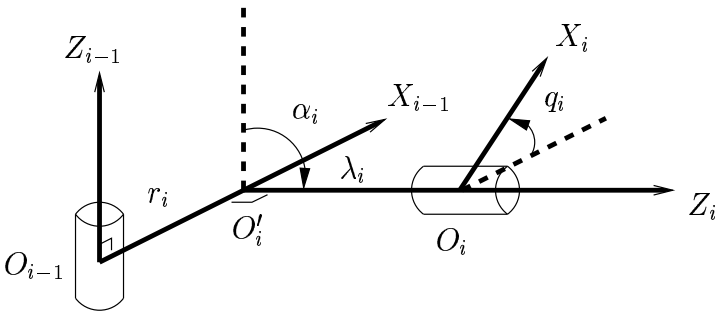


FIG. 2.5 – Description du paramétrage de Khalil-Kleinfinger

A chaque solide i on associe un repère (O_i, X_i, Y_i, Z_i). La représentation classique de Khalil-

Kleinfinger pour une chaîne de n segments successifs est donnée par une liste de vecteurs de la forme $(r_i, \lambda_i, \alpha_i, q_i)_{i=1..n}$. Pour définir la position et l'orientation du solide i par rapport au solide $i-1$ ces paramètres sont défini de la manière suivante :

$$\begin{cases} r_i = \overrightarrow{O_{i-1}O'_i} \cdot \overrightarrow{X_{i-1}} \\ \lambda_i = \overrightarrow{O'_iO_i} \cdot \overrightarrow{Z_i} \\ \alpha_i = (\widehat{\overrightarrow{Z_{i-1}}, \overrightarrow{Z_i}}) \cdot \overrightarrow{X_{i-1}} \\ q_i = (\widehat{\overrightarrow{X_{i-1}}, \overrightarrow{X_i}}) \cdot \overrightarrow{Z_i} \end{cases} \tag{2.1}$$

Ainsi, la matrice qui permet de passer d'un repère à l'autre prend la forme (en coordonnées homogènes) :

$$\begin{bmatrix} \cos q_i & -\sin q_i & 0 & r_i \\ \cos\alpha_i \sin q_i & \cos\alpha_i \cos q_i & -\sin\alpha_i & -\lambda_i \sin\alpha_i \\ \sin\alpha_i \sin q_i & \sin\alpha_i \cos q_i & \cos\alpha_i & \lambda_i \cos\alpha_i \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Le détail de la modélisation géométrique du bipède est disponible dans le rapport technique de P. Sardain [16].

2.2.2 Modélisation Dynamique

La dynamique du robot BIP peut être établie suivant un modèle Lagrangien. En utilisant la formulation d'Euler-Lagrange on obtient la dynamique suivante :

$$M(q)\,\ddot{q} + N(q,\dot{q})\,\dot{q} + G(q) = T(q)\,u \tag{2.2}$$

Où $M(q)$ représente la matrice d'inertie du système et $N(q,\dot{q})\,\dot{q}$ rassemble pour sa part des effets non-linéaires tels que les effets centrifuges, gyroscopiques et Coriolis. $G(q)$ représente le vecteur gravité du système et les efforts généralisés sont de la forme $T(q)\,u$, où $T(q)$ est une matrice représentant les caractéristiques électromécaniques des moteurs et des transmissions, et u le vecteur des consignes envoyées aux moteurs.

D'un point de vue mécanique, le robot, en phase de locomotion, est un système mécanique de corps rigide soumis à des contraintes unilatérales. Ces contraintes sont des inégalités qui traduisent la non-interpénétrabilité des corps rigides en présence de frottements secs. Ces inégalités sont de la formes :

$$\varphi_n(q) \geq 0$$

Où φ_n désigne la position de différents points du robot. On note φ_n^* l'ensemble des points du robot qui sont en contact avec le sol, autrement dit l'ensemble des contraintes actives à l'instant t. On peut alors écrire la relation suivante :

$$\varphi_n^*(q) = 0 \tag{2.3}$$

Afin de s'assurer que ces points ne pénètre pas dans le sol à l'instant t on impose une vitesse et une accélération nulles à cet instant. Ce qui se traduit par les deux conditions suivantes :

$$C_n(q)\,\dot{q} = 0 \tag{2.4}$$

$$C_n(q)\,\ddot{q} + s_n(q,\dot{q}) \geq 0 \tag{2.5}$$

Où $C_n(q) = \partial\varphi_n^*/\partial q$ est la matrice jacobienne de φ_n^* et $s_n(q,\dot{q})$ le hessiens de φ_n^*.

De même lorsqu'un robot est en contact avec le sol, il peut éventuellement glisser dessus, mais des forces de frottement entrent alors en jeu, et s'y opposent. Les points soumis au glissement se déplace alors parallèlement au sol.
On se place ici dans le cas où le robot ne glisse pas sur le sol. Dès lors ceci se traduit par un ensemble de contraintes portant sur les positions des points posés au sol, ce que l'on peut exprimer par les égalités suivantes :

$$\varphi_t^*(q) = 0 \tag{2.6}$$

φ_t^* désigne ainsi les contraintes de non-glissement à l'instant t. Il en découle également un ensemble de contraintes sur la vitesse et l'accélération du système, que nous obtenons par dérivations successives :

$$C_t(q)\,\dot{q} = 0 \tag{2.7}$$

$$C_t(q)\,\ddot{q} + s_t(q,\dot{q}) = 0 \tag{2.8}$$

Où $C_t(q) = \partial\varphi_t^*/\partial q$ est le jacobien des contraintes de non-glissement et $s_t(q,\dot{q})$ le hessiens de φ_t^*.

On peut alors adjoindre à la dynamique (2.2) du robot les contraintes linéaires (2.5) et (2.8). On obtient alors une nouvelle formulation de la dynamique :

$$M(q)\,\ddot{q} + N(q,\dot{q})\,\dot{q} + G(q) = T(q)\,u + C(q)^T\lambda \tag{2.9}$$

$$C_n(q)\,\ddot{q} + s_n(q,\dot{q}) \geq 0 \tag{2.10}$$

$$C_t(q)\,\ddot{q} + s_t(q,\dot{q}) = 0 \tag{2.11}$$

$$\mathcal{A}(\lambda) \geq 0 \tag{2.12}$$

Avec $C(q) = \begin{bmatrix} C_n(q) \\ C_t(q) \end{bmatrix}$ et $\lambda = \begin{bmatrix} \lambda_n \\ \lambda_t \end{bmatrix}$ Le développement mathématique qui conduit à cette formulation de la dynamique est détaillé dans la référence [18]. La dynamique 2.9 fait ainsi intervenir les multiplicateurs de Lagrange λ_n et λ_t qui traduisent l'amplitude des efforts introduit par les contraintes de non-pénétration et de non-glissement.

Nous sommes donc en présence de forces de contact normales et tangentielles. On sait qu'un ensemble de forces normales s'opposent à toute pénétration dans le sol tandis qu'un ensemble de forces tangentielles s'opposent aux glissements. Or, selon le modèle de frottements secs

d'Amontons-Coulomb, lorsqu'un contact entre deux objets est soumis à une force normale f_n, la force de frottement tangentielle f_t ne peut prendre ses valeurs que dans un cône de frottement, on introduit alors l'inégalité (2.12) qui impose aux forces de contact de bien rester à l'intérieur du cône de frottement.

2.2.3 Interprétation de la dynamique

Nous avons vu que les actionneurs du robot et leurs transmissions ont pour but de produire un ensemble de couples τ sur chacune des articulations. La dynamique (2.9) peut alors s'écrire de la manière suivante :

$$M(q)\,\ddot{q} + N(q,\dot{q})\,\dot{q} + G(q) = \begin{bmatrix} \tau \\ 0 \end{bmatrix} + C(q)^T \lambda \tag{2.13}$$

Ceci met en évidence le fait que les moteurs agissent sur une partie seulement de la position du robot. Plus précicément, en agissant sur les articulations du robot, c'est sur sa posture qu'ils agissent.

Afin de préciser l'effet de ces couples sur la dynamique du robot, reprenons la description de la position du robot que nous avons entrepris au début de cette section, et considérons séparément un vecteur q_1 constitué des positions articulaires, et un vecteur q_2 décrivant la position et l'orientation d'un de ses solides. Le vecteur q, qui décrit la position du robot, rassemble ces deux vecteurs et présente donc la structure suivante :

$$q = \begin{bmatrix} q_1 \\ q_2 \end{bmatrix} \tag{2.14}$$

On montre alors [18] qu'il est possible de découper chacun des éléments de cette dynamique selon le même schéma que les efforts moteurs. La dynamique possède alors la structure suivante :

$$\begin{bmatrix} M_1(q) \\ M_2(q) \end{bmatrix} \ddot{q} + \begin{bmatrix} N_1(q,\dot{q}) \\ N_2(q,\dot{q}) \end{bmatrix} \dot{q} + \begin{bmatrix} G_1(q) \\ G_2(q) \end{bmatrix} = \begin{bmatrix} \tau \\ 0 \end{bmatrix} + \begin{bmatrix} C_1(q)^T \\ C_2(q)^T \end{bmatrix} \lambda \tag{2.15}$$

P.B. Wieber [18] a montré que la deuxième partie de cette expression de la dynamique est composée d'une équation de Newton et d'une équation d'Euler qui mettent respectivement en jeu l'accélération du centre de masse du robot et son moment dynamique de rotation.
L'équation de Newton montre ainsi que les déplacements du robot sont exclusivement influencés par la gravité et par les forces de contact. En particulier, ces déplacements apparaissent indépendants des changements de posture et des rotations du robot.
L'équation d'Euler, pour sa part, fait apparaître de façon indissociable les rotations et les changements de posture du robot.

Ainsi, c'est uniquement par l'intermédiaire des forces de contact, en prenant appui sur le sol, que le robot peut être en mesure d'influer sur ses déplacements. Mais ces forces de contact établissent en fait, une forte corrélation entre les changements de posture et les déplacements du robot. En effet, tout changement de posture est en mesure de produire un déplacement ,et réciproquement, si le bipède est en appui sur le sol.

Chapitre 3

Stabilité et Loi de commande

Ce chapitre a pour but d'évoquer la stabilité d'un robot bipède et de décrire la loi de commande retenue pour notre robot. En outre, il répertorie les différentes approximations possibles dans le calcul de la loi de commande et le temps CPU qu'il faut pour les générer. Les tests de temps de calcul sont réalisés dans un premier temps sur un calculateur de type Pentium, puis sur le calculateur embarqué du robot BIP. On détermine ainsi les besoins en puissance de calcul nécessaire à la mise en oeuvre d'une marche dynamique 3D stable sur sol plat.

3.1 Stabilité d'un bipède

Cette section commence par introduire quelques définitions, pour aboutir à une définition convenable de la stabilité d'un bipède.

On appelle **polygone de sustentation** l'enveloppe convexe contenant tous les points de contact du bipède avec le sol (Cf figure 3.1).

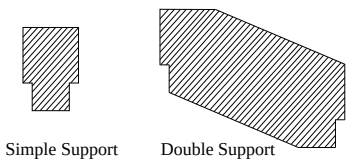


FIG. 3.1 – Polygone de sustentation

Un bipède est dit **statiquement stable** si et seulement si la projection de son centre de masse sur le sol se trouve à l'intérieur du polygone de sustentation.

La **démarche** d'un bipède est dite **statiquement stable** si la stabilité statique du robot est vérifiée à chaque instant de son déplacement. On notera que ce type de démarche fait intervenir de faibles vitesses et de faibles accélérations lors des déplacements.

La **démarche** d'un bipède est dite **dynamiquement stable** lorsque la trajectoire suivie est constituée d'une succession de mouvement en déséquilibre menant pourtant à un marche stable. A cette démarche, on associe souvent le critère du ZMP (Zéro Moment Point). Le ZMP est défini comme le centre de pression des forces de contact. Si ce point reste à l'intérieur du polygone de sustentation alors la démarche est dite dynamiquement stable.

Cependant, ces deux critères de stabilités ne sont pas satisfaisants. Ils ne prennent pas en compte les contraintes de non-glissement évoquées dans le chapitre précédent, de plus le ZMP n'est pas défini, et le critère du polygone de sustentation n'est valide, que si tous les points de contact sont sur un même plan. Cette analyse n'est donc plus valable dès que le robot monte ou descend une marche ou dès qu'il prend appui sur des plans d'inclinaisons différentes : elle n'est valable que si le robot se déplace indéfiniment sur un même plan, ce qui est évidemment plutôt réducteur.

Pour notre étude, nous utiliserons l'approche proposée par P.B Wieber dans [18], basée sur la notion de mouvement réalisable. La faisabilité d'un mouvement correspond alors en quelque sorte à un critère de stabilité instantanée. Cette approche s'appuie directement sur l'écriture du modèle dynamique du robot tel qu'il a été défini à la fin du chapitre précédent. Considérons alors les deux partie de la dynamique (2.15) et l'équation (2.12).

Soient q, $\dot{q}$, $\ddot{q}$ donnés à l'instant t.
S'il existe des forces de contact vérifiant les équations (2.15) et (2.12) pour un couple articlaire τ imposé à l'instant t, alors le mouvement est réalisable.

Si l'on considère maintenant une trajectoire de marche qui fixe q, $\dot{q}$ et $\ddot{q}$ et qu'à chaque instant cette trajectoire est réalisable, au sens où on l'a défini ci-dessus, alors le mouvement global sera réalisable.
C'est en ce sens que l'on considèrera la stabilité d'une trajectoire.

3.2 Les Trajectoires de Références

La mise en œuvre d'une démarche anthropomorphe fait intervenir deux notions fortement corrélées. La génération de **trajectoires de référence** et la **loi de commande** proprement dite, qui permettra d'ailleurs de suivre celle-ci à tout instant. C'est la première de ces deux notions que nous abordons ici.

Définir une trajectoire de référence, c'est donner l'évolution de la valeur des coordonnées articulaires du robot au cours du temps. Bien sûr, une telle trajectoire doit satisfaire le critère de stabilité énoncé à la section précédente.

La méthode la plus élémentaire mise en œuvre consiste à imposer des contraintes cinématiques arbitraires sur certaines variables caractéristiques du robot. Ces contraintes tentent de faire ressembler l'allure générale de la trajectoire de marche du robot à une trajecoire humaine (par exemple contrainte de verticalité du tronc et trajectoire parabolique du pied de

vol). Cependant la trajectoire ainsi formée ne s'appuie pas sur des contraintes réelles, mais sur des contraintes arbitrairement choisies qui manquent de signification physique.

D'un point de vue pratique, la génération de trajectoire de référence se fait actuellement, par interpolation polynômiale de degré 5. Quelques positions stable, au sens d'un mouvement réalisable, sont établis numériquement puis le reste de la trajectoire est déterminé par interpolation. Une étude plus approfondie sur la génération de trajectoire est abordée dans le chapitre suivant.

Pour assurer un démarche stable, le seul objectif considéré est de stabiliser asymptotiquement une unique trajectoire de référence. Si le robot est suffisament proche de cette trajectoire à un moment donné, une telle stabilité asymptotique permet d'assurer son équilibre par la suite.

3.3 La loi de commande

Le premier objectif de la loi de commande est donc de stabiliser asymptotiquement une trajectoire de référence $q_d(t)$, et nous supposerons pour commencer que cette trajectoire satisfait les contraintes de non-pénétration et de non-glissement présentées dans le chapitre précédent. Ne sachant maîtriser la stabilité des mouvements d'un robot en présence de prises de contact et de pertes de contact, nous supposerons également qu'aucune perte de contact ne survient de façon impromptue, et que les prises de contact ne perturbent pas cette stabilité. Plus particulièrement, nous supposerons qu'après chaque prise de contact, les contraintes actives normales et tangentielles sont celles prévues par la trajectoire de référence.

On ne sait donc garantir la stabilité d'une trajectoire que si le robot maintient un contact avec le sol conforme à ce que cette trajectoire prévoit, et pour assurer ce dernier point, nous avons décidé d'appliquer l'approche par **fonction de tâche** [15].

Cette fonction exprime l'erreur de positionnement du robot par rapport à une trajectoire de référence Q_d où Q désigne un changement de variable faisant apparaître entre autres les contraintes de contact actives :

$$Q(q) = \begin{bmatrix} Q_n(q) = \varphi_n^*(q) \\ Q_t(q) = \varphi_t^*(q) \\ \vdots \end{bmatrix} \tag{3.1}$$

En éliminant au besoin les contraintes dépendantes, nous supposerons que ce changement de variables constitue un difféomorphisme de classe C^2 entre un ensemble Ω et son image $Q(\Omega)$. Nous supposerons que la position du robot se trouve toujours à l'intérieur de cet ensemble, que la trajectoire $q_d(t, p(t))$ reste toujours à l'intérieur de cet ensemble, qu'elle reste à une certaine distance des limites de cet ensemble, et qu'elle est bornée : dans ces conditions, il devient totalement indifférent de considérer la stabilité asymptotique de cette trajectoire avant ou après ce changement de variables.
Deux exemples complets de changement de variables sont présentés dans le chapitre suivant.

Intéressons-nous à cette fonction de tâche et à ses dérivées successives :

$$\begin{aligned} e(q) &= Q(q) - Q_d \\ \dot{e}(q,\dot{q}) &= H(q)\,\dot{q} - \dot{Q}_d \\ \ddot{e}(q,\dot{q},\ddot{q} &= H(q)\,\ddot{q} + h(q,\dot{q}) - \ddot{Q}_d \end{aligned} \tag{3.2}$$

avec $H(q) = \partial Q/\partial q$ la matrice jacobienne du changement de variables, $h(q,\dot{q})$ les autres termes apparaissant lors de la dérivation.

En s'appuyant sur la dynamique (2.13) du robot, l'approche par fonction de tâche consiste à faire réaliser par le robot des efforts moteurs τ calculés à partir de la formule suivante :

$$\begin{bmatrix} \tau \\ 0 \end{bmatrix} + C(q)^T\lambda = M(q)\,H(q)^{-1}\big[v - h(q,\dot{q}) + \ddot{Q}_d)\big] + N(q,\dot{q})\,\dot{q} + G(q) \tag{3.3}$$

où le vecteur v reste à définir, ce qui permet d'obtenir une dynamique de l'erreur $e(q,t)$ découplée et linéarisée :

$$\ddot{e}(q,\dot{q},\ddot{q},t) = v$$

Ce vecteur v nous permet donc de contrôler directement l'évolution de $e(q)$. On choisit alors v, tel que la loi de commande soit un simple proportionnel-dérivé.

$$v = -K_p\big(Q(q) - Q_d(q)\big) - K_v\big(\dot{Q}(q) - \dot{Q}_d(q)\big) \tag{3.4}$$

ce qui nous amène à la dynamique :

$$\ddot{e}(q,\dot{q},\ddot{q},t) = -K_p e(q) - K_v \dot{e}(q,\dot{q})$$

où K_p et K_v sont respectivement les gains proportionnels et dérivé.

3.4 Quelques simplifications

Le temps nécessaire pour le calcul d'une loi de commande détermine la fréquence à laquelle on peut l'échantillonner, ce qui peut avoir une influence décisive sur ses performances : toute simplification de ce calcul est donc à même d'avoir une influence bénéfique sur l'équilibre du robot.

La loi de commande précédente, fournit l'ensemble des couples articulaires à appliquer à chaque pas d'échantillonnage. Ces couples sont déterminés à partir de l'équation (3.3). On dispose cependant d'un calculateur embarqué aux capacités de calcul limitées. Afin de maintenir une période d'échantillonnage suffisament faible (de l'ordre de 10 ms), il nous faut évaluer le temps de calcul des différentes lois de commande.

On distingue trois types de loi de commande, toutes basées sur l'équation (3.3) à différents niveaux de dégradations :

$$T(q)\,u + C(q)^T\lambda = v + G(q) \tag{3.5}$$

$$T(q)\,u + C(q)^T\lambda = M(q)\,\big(g + v\big) \tag{3.6}$$

$$T(q)\,u + C(q)^T\lambda = M(q)\Big(\,g + H(q)^{-1}\big(v - h(q,\dot{q}) + \ddot{Q}_d\big)\Big) \tag{3.7}$$

Où g est un vecteur constant tel que $M(q)g = G(q)$

L'équation (3.5) défini une loi de commande proportionnelle-dérivé avec compensation de la gravité, tandis que l'équation (3.6) défini une loi de commande dite "Computed Torque".

On utilise en simulation une loi de commande plus sophistiquée que les précédentes. Celle-ci est donnée par le problème linéaire suivant :

$$\begin{cases} \min\limits_{u,\,\lambda,\,w} \; w \\ T(q)\,u + C(q)^T\lambda = M(q)\Big(\,g + H(q)^{-1}\big(v - h(q,\dot{q}) + \ddot{Q}_d\big)\Big) \\ \mathcal{A}(\lambda) + w \geq 0 \end{cases} \tag{3.8}$$

Ce problème consite à chercher des forces de contact les plus éloignées des bords du cône de frottement, permettant ainsi d'assurer au mieux le contact avec le sol.

Cependant il est peu probable que l'on puisse utiliser cette procédure sur le robot BIP tant elle est coûteuse en temps de calcul. Le lecteur pourra se reporter à la section 3.6 pour de plus amples informations sur les temps de calcul.

Dans tous les cas, On notera que ces différentes loi de commande, dégradées ou non, ne tiennent pas compte du vecteur $N(q,\dot{q})\,\dot{q}$ qui rassemble les effets non-linéaire tels que les effets centrifuges, gyroscopyques et Coriolis. On décide de négliger leurs influences dans le calcul de la loi de commande.

3.5 Présentation des procédures C et Scilab

Considérons l'ensemble des lois de commandes précédentes. On distingue plusieurs éléments de calcul, notamment $C(q)$ le jacobien des points de contact, $M(q)$ la matrice d'inertie du robot, le vecteur gravité $G(q)$ et les matrices du changement de variable telles que $H(q)$, $h(q,\dot{q})$ définies plus haut.

Le modèle du robot BIP a été préalablement généré en langage C à l'aide du logiciel de calcul symbolique Maple. Les différentes procédures de calcul utilisé pour la loi de commande sont données tableau (3.1).

`gravite(q)`	$G(q)$, vecteur des effets gravitationnels
`inertie(q)`	$M(q)$, matrice d'inertie du robot
`jacobien_contact(q)`	$C(q)$, jacobien des coordonnées des points de contact
`sortie(q)`	procédure de changement de variable $Q(q)$
`jacobien_sortie(q)`	$H(q)$, jacobien du changement de variable $Q(q)$
`hessiens_sortie(q,q̇)`	$h(q, \dot{q})$, terme apparaissant lors de la seconde dérivation du changement de variable $Q(q)$

TAB. 3.1 – Ensemble des procédures C utilisées pour le calcul de la commande

3.6 Evaluation des temps de calcul

On se propose maintenant, d'évaluer les temps de calcul de la loi de commande pour les différents niveaux de dégradations explicités à la section 3.4. On utilise pour ces tests, les procédures de calcul Scilab sur un calculateur Pentium 4 à 1,5 GHz et les procédures C pour le calculateur embarqué (processeur MC68040 à 32 MHz) sur le robot BIP. Les résultats de ces tests sont répertorié dans le tableau (3.2).

On notera que les temps de calcul relatif aux procédures Scilab sont ralenti par le logicel lui-même. Le test des procédures C sur Pentium 4 prendrait moins de temps.

3.7 Conclusions

Notre objectif est d'implanter sur la plateforme expérimentale une des lois de commande définies par les équations 3.6 et 3.7 et si possible la dernière.

La période d'échantillonnage est actuellement fixée sur la plateforme expérimentale à 10 ms. Elle comprend l'acquisition des données capteurs, le calcul des ratios de transmissions du robot et la loi de commande.

La loi de commande implantée actuellement sur le robot, n'utilise pas de changement de variable, ne prend pas en compte les efforts $C(q)^T\lambda$ et ne comporte qu'un correcteur proportionnel-dérivé, une estimation de la gravité et une compensation de frottement, le tout évalués à environ 4ms [2] (Cf chapitre 8).

On se rend bien compte qu'avec un calculateur puisssant de type Pentium le calcul de la

PROCEDURES	PENTIUM	MC68040
`gravite(q)`	0,11 ms	0,6 ms
`inertie(q)`	0,24 ms	9,3 ms
`jacobien_contact(q)`	0,15 ms	3,3 ms
`sortie(q)`	0,1 ms	0,6 ms
`jacobien_sortie(q)`	0,13 ms	2,8 ms
`hessiens_sortie(q,q̇)`	0,25 ms	8,5 ms
$\mathtt{T(q)\,u} + \mathtt{C(q)}^{\mathtt{T}}\lambda = \mathtt{v} + \mathtt{G(q)}$	0,28 ms	5,5 ms
$\mathtt{T(q)\,u} + \mathtt{C(q)}^{\mathtt{T}}\lambda = \mathtt{M(q)}\left(\mathtt{g} + \mathtt{v}\right)$	0,53 ms	14,7 ms
$\mathtt{T(q)\,u} + \mathtt{C(q)}^{\mathtt{T}}\lambda = \mathtt{M(q)}\left(\mathtt{g} + \mathtt{H(q)}^{-1}\left(\mathtt{v} - \mathtt{h(q,q)} + \ddot{\mathtt{Q}}_{\mathtt{d}}\right)\right)$	1,41 ms	30,0 ms
`Problème linéaire (3.8)`	7,97 ms	——

TAB. 3.2 – Evaluation des temps de calcul des procédures C et Scilab.

loi de commande sous sa forme complète (équation 3.7) ne présenterait pas de problème. Par contre le calculateur embarqué sur le robot est fortement limité. Au vue des temps de calcul CPU sur celui-ci, les lois de commande définies par les équations 3.6, 3.7 et 3.8 ne sont pas implantables.

En conséquence, il est pour nous important de déterminer la solution la plus appropriée pour pouvoir au moins implanter la loi de commande Computed Torque sur la plateforme expérimentale. Le chapitre 6 fait l'inventaire des possibilités qui s'offrent à nous et conclut sur les mesures à prendre.

Chapitre 4

Développement sur le simulateur du robot BIP

Le calcul d'un modèle de robot est un problème classique que de nombreux outils tels que Symoro ou Robotica permettent de résoudre. Mais le modèle du robot BIP comporte 21 degrés de libertés, ce qui nous amène très loin des robots manipulateurs habituels qui en comptent rarement plus de 6, et la taille du modèle généré, ainsi que le temps de calcul et les besoins en mémoire explosent rapidement si les algorithmes employés ne sont pas adaptés.

4.1 Le modèle du robot BIP

Le Laboratoire de Mécanique des Solides (LMS) de l'Université de Poitiers s'est inspiré des données cinématiques et des capacités dynamiques anthropomorphes pour la conception du robot. En s'appuyant sur ces travaux, l'équipe du projet BIP a développé une application spécifique qui s'appuie sur les capacités de calcul symbolique du logiciel Maple, et a généré le modèle du robot BIP. Ce modèle est composé de 5 fichiers :

Fichier *cinematique.maple*

Ce fichier modèlise la position du robot BIP en fonction du vecteur q par l'intermédiaire des paramètres de Khalil-Kleinfinger (Cf figure 4.2).

Fichier *complement.maple*

Ce fichier caractérise certains points important comme les points de contacts du robot BIP (Cf figure 4.1).

Fichier *dynamique.maple*

Ce fichier défini les paramètres dynamiques du robot (masse, matrice d'inertie et centre de masse de chaque solide qui constitue le robot) [16].

Fichier *robodyn2000.maple*

Ce fichier, écrit par F.Genot et modifié par P.B.Wieber, contient toutes les fonctions qui permettent de générer, sous forme de procédures C et Scilab, le modèle cinématique et dynamique du robot BIP (Cf tableau 4.1).

Fichier *commande.maple*

Ce fichier contient la définition du changement de variable utilisé pour le calcul de la loi de commande. Il calcule aussi le jacobien et le hessien de ce changement de variable.

4.2 Organisation du simulateur

Le simulateur du robot BIP est un outil générique de simulation de systèmes dynamiques lagrangiens avec contraintes unilatérales en présence de frottement sec. Ce simulateur s'appuie sur les capacités de calcul scientifique du logiciel `Scilab`. Il utilise le modèle du robot généré par Maple en langage C et au format Scilab et est constitué de 9 procédures, définies dans les fichiers :

- *dynamique.sci*
- *impact.sci*
- *jacadi.sci*
- *invsortie.sci*
- *trajectoire.sci*
- *commande.sci*
- *tension.sci*
- *simulation.sci*
- *visu.sci*

Ces procédures sont décrites ci-dessous et supposent que le modèle dynamique du système est lui-même disponible sous la forme des procédures décrite précédemment.

Fonction *[qddot, lambda] = dynamique(q, qdot, couple, contacts_actifs)*

1. Cette procédure permet de calculer la dynamique du robot BIP.
2. Elle prend en paramètre q, $\dot{q}$, les efforts $T(q)\,u$ généré par la commande et les contraintes de contact actives φ_n^*.
3. Elle retourne l'accélération $\ddot{q}$ du système et les forces de contact λ obtenues.

Fonction *[qdot_plus, Lambda] = impact(q, qdot_moins, contacts_actifs)*

1. Cette procédure permet de déterminer la vitesse après impact et la force impulsive qui en résulte.
2. Elle prend en paramètre q, la vitesse avant impact $\dot{q}_-$ et l'ensemble des contacts actifs.

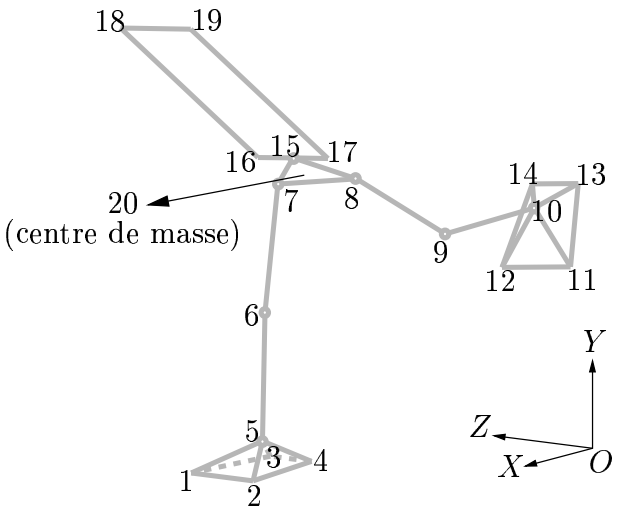


FIG. 4.1 – Quelques points caractéristiques.

```
# Coordonnées de l'origine du repère de référence
base := vector([q[16], q[17], q[18]]):

# Repère 1 : pied droit
ref_1    := 20:
r_1      := 0:
lambda_1 := 0:
alpha_1  := 0:
theta_1  := Pi/2:

# Repère 2 : cardan droit
ref_2    := 1:
r_2      := 0.083:
lambda_2 := -0.170:
alpha_2  := Pi/2:
theta_2  := q[1]:

# Repère 3 : tibia droit
ref_3    := 2:
r_3      := 0:
lambda_3 := 0:
alpha_3  := -Pi/2:
theta_3  := q[2]:

# Repère 4 : cuisse droite
ref_4    := 3:
r_4      := 0.410:
lambda_4 := 0:
alpha_4  := 0:
theta_4  := q[3]:

# Repère 5 : levier hanche droite
ref_5    := 4:
r_5      := 0.410:
lambda_5 := 0:
alpha_5  := 0:
theta_5  := Pi/2+q[4]:
.
.
.
.
.
# Repère 16 : cardan armoire
ref_16    := 15:
r_16      := 0:
lambda_16 := 0:
alpha_16  := Pi/2:
theta_16  := Pi/2+q[14]:

# Repère 17 : support armoire + armoire
ref_17    := 16:
r_17      := 0:
lambda_17 := 0:
alpha_17  := -Pi/2:
theta_17  := q[15]:

# Repère 18 : rotation Y du repère de référence
ref_18    := 0:
r_18      := 0:
lambda_18 := 0:
alpha_18  := -Pi/2:
theta_18  := Pi/2+q[20]:

# Repère 19 : rotation X du repère de référence
ref_19    := 18:
r_19      := 0:
lambda_19 := 0:
alpha_19  := Pi/2:
theta_19  := Pi/2+q[19]:

# Repère 20 : rotation Z du repère de référence
ref_20    := 19:
r_20      := 0:
lambda_20 := 0:
alpha_20  := -Pi/2:
theta_20  := -Pi/2+q[21]:
```

FIG. 4.2 – Extrait du fichier cinematique.maple

`matrice_passage(k)`	matrice de passage du repère k vers le repère ref_k
`matrice_repere(k)`	matrice de passage du repère k vers le repère d'origine
`coord_cdm(k)`	coordonnées du centre de masse du solide k exprimées dans le repère absolu
`coord_cdm_robot()`	coordonnées du centre de masse du robot
`vecteur_gravite()`	$G(q)$, vecteur des effets gravitationnels
`jac_rotation(k)`	$J_{Rk}(q)$, jacobien de rotation du solide k
`mat_huygens(G)`	moment d'inertie d'une masse de 1 kg positionnée au point G (pour le théorème de Huygens)
`matrice_inertie()`	$M(q)$, matrice d'inertie du robot
`matrice_coriolis()`	$N(q, \dot{q})$, matrice des effets non-linéaires
`coord_tag(k)`	coordonnées du tag k dans le repère d'origine
`matrice_tag()`	matrice réunissant les coordonnées de l'ensemble des tags ainsi que celles du centre de masse du robot
`matrice_contact()`	matrice réunissant les coordonnées des points de contact utilisée pour générer les contraintes $\varphi(q)$
`jacobien_contact()`	$C(q)$, jacobien des coordonnées des points de contact
`hessiens_contact()`	$s(q, \dot{q})$, terme apparaissant lors de la seconde dérivation des coordonnées des points de contact

TAB. 4.1 – Ensemble des procédures de l'application robotdyn2000.

3. Elle retourne la vitesse après impact $\dot{q}_+$ et la force impulsive λ qui en résulte.

Fonction *xdot = equadiff(t, x)*

1. Cette procédure permet de mettre la dynamique du système sous la forme $\dot{x} = f(t, x)$ avec $x = \begin{bmatrix} q \\ \dot{q} \end{bmatrix}$.
2. Elle prend en paramètre l'instant t et le vecteur x
3. Elle retourne le vecteur $\dot{x} = \begin{bmatrix} \dot{q} \\ \ddot{q} \end{bmatrix}$.

Fonction *detect = detection(t, x)*

1. Cette procédure permet de comparer $\varphi_n(q)$ avec les niveaux de détection des impacts et des décollages.
2. Elle prend en paramètre l'instant t et le vecteur d'état x.
3. Elle retourne un vecteur *detect* qui rend compte de la detection d'un impact ou d'un décollage suivant une altitude fixé à 2.10^-8.

Fonction *[q]=jacadi(selection, sortie_desiree, appuis)*

1. Cette procédure permet de déterminer la position q qui minimise la consommation d'énergie du robot à l'arrêt tout en satisfaisant les limites articulaires et une selection de fonctions de sortie désirée, les appuis du robot étant précisés.
2. Elle prend en paramètre une selection de fonctions de sortie désirée, la valeur affecté à ces sorties et les appuis préconisés.
3. Elle retourne les positions q correspondant à ces contraintes .

Fonction *[q, d] = invsortie(sortie_desirée)*

1. Cette procédure permet de calculer la position q telle que sortie(q) soit le plus proche possible de sortie_desiree, dans les limites des butees articulaires. (d est la distance obtenue).
2. Elle prend en paramètre un vecteur de dimension 21 sous la forme du changement de variable ennoncée plus haut.
3. Elle retourne le vecteur q et la distance obtenu d.

Fonction *[pos, vit, acc, cont] = trajectoire(t, pos_ref, cont_ref)*

1. Cette procédure permet de construire une trajectoire de reférence par interpolation polynômiale de degré 6, à partir de positions de référence ateintes aux instants t et avec des contacts précisé.
2. Elle prend en paramètre les positions de passages pos_ref aux instants t avec les contacts $cont_ref$ correspondant.

3. Elle retourne la positions, la vitesse, l'accélération et les informations sur les contacts à l'instant t.

Fonction *[couple] = commande(t, q, qdot, lambda)*

1. Cette procédure fait le calcul de la loi de commande par l'approche de fonction de tâche.
2. Elle prend en paramètre q, $\dot{q}$ et les forces de contact à l'instant t .
3. Elle retourne la consigne sous forme de couple articulaire.

Fonction *[u] = tension(q, couple)*

1. Cette procédure permet de convertir les couples articulaires en tension moteur en appliquant les transformations dû aux systèmes de transmissions sur le robot BIP.
2. Elle prend en paramètre q et les couples articulaires correspondant.
3. Elle retourne la tension u en Volt.

Fonction [Q, ...] = simulation(t_ini, t_fin, delta_t, q_ini, qdot_ini, commande)

1. Procédure principale de simulation de la dynamique du système.
2. Elle prend en paramètre les instants de début et de fin de simulation t_{ini} et t_{fin} en la loi de commande échantillonné tous les δt.
3. Elle retourne les positions Q, les vitesses $\dot{Q}$, les forces de contact λ, les consignes de couples τ réalisées aux instants d'échantillonnage T, les accélération $\ddot{Q}$ et les consignes de tension.

Fonction visu(Q, LAMBDA)

1. Cette procédure permet une visualisation 3D du système avec, selon le cas, les forces de contact ou la projection d'un point du système sur le plan $y = 0$.
2. Elle prend en paramètre l'ensemble des positions articulaires et des forces de contact pour une trajectoire donnée

Le simulateur s'organise comme le montre la figure (4.3)

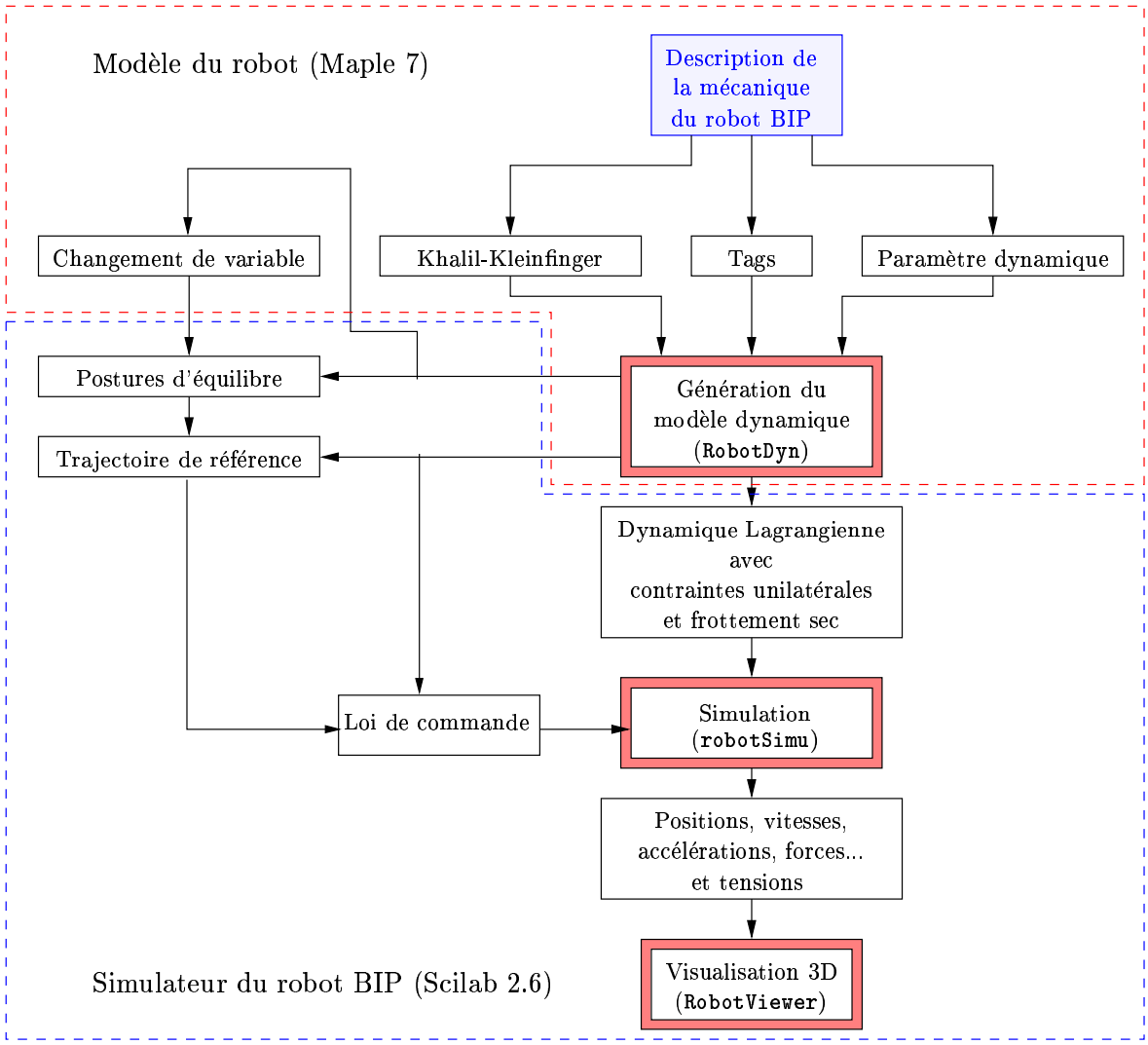


FIG. 4.3 – Organisation du simulateur du robot BIP.

4.3 Changement de variable

Nous avons vu précédemment que la position du robot est décrite entièrement par le vecteur q. Il précise l'ensemble des positions articulaires, la position et l'orientation du robot dans l'espace. On sait aussi que l'on ne peut garantir la stabilité d'une trajectoire que si le robot maintient un contact avec le sol conforme à ce que cette trajectoire prévoit, et pour assurer ceci, on utilise une loi de commande par fonction de tâche.

Pour réaliser une telle loi de commande, on préfère intoduire un changement de variable $Q(q)$ qui fasse apparaître les contraintes de contact actives $\varphi^*(q)$ et les coordonnées du centre de masse du robot, élément essentiel à la stabilité statique du robot.

Le tableau 4.2 présente le changement de variable développé par P.B Wieber et utilisé pour le calcul de la loi de commande dans le simulateur du robot BIP. Il y exprime les coordonnées du centre de masse, des chevilles et de l'orientation du tronc dans un repère absolu (figure 4.1). Les coordonnées exprimées dans le repère absolu sont dotées de l'indice A.

$Q_{1...3}$	$\{x_G^A,\ y_G^A,\ z_G^A\}$	position du centre de masse du robot
$Q_{4...6}$	$\{x_5^A,\ y_5^A,\ z_5^A\}$	position de la cheville droite
$Q_{7...9}$	$\{x_{10}^A,\ y_{10}^A,\ z_{10}^A\}$	position de la cheville gauche
$Q_{10...12}$	$\{y_{17}^A - y_{16}^A,\ x_{17}^A - x_{16}^A,\ x_{18}^A - x_{16}^A\}$	orientation du tronc
$Q_{13...15}$	$\{y_1^A - y_2^A,\ x_1^A - x_2^A,\ y_4^A - y_2^A\}$	orientation du pied droit
$Q_{16...18}$	$\{y_{11}^A - y_{12}^A,\ x_{11}^A - x_{12}^A,\ y_{14}^A - y_{12}^A\}$	orientation du pied gauche
$Q_{19...21}$	$q_{13...15}$	

TAB. 4.2 – Définition d'un changement de variables

Ce changement de variable a donc pour but de rendre possible le contrôle du centre de masse du robot et de faire apparaître les contraintes de contact actives $\varphi^*(q)$ afin de s'assurer que les contacts du robot sur le sol sont bien ceux imposés par la trajectoire de référence. En effet, on remarque que les éléments $Q_{4...6}$ et $Q_{13...15}$ de ce changement de variables représentent la position et l'orientation du pied droit, qui participent d'un bloc aux contraintes de contact actives lorsque celui-ci est posé bien à plat sur le sol, les éléments $Q_{7...9}$ et $Q_{16...18}$ remplissant

le même rôle pour le pied gauche.

Néanmoins, il ne présente pas que des avantages. En effet, si la trajectoire de référence prévoit un appui en simple support sur le sol, la loi de commande, telle qu'elle a été conçue, impose à ce contact de ne plus bouger. Dès lors, si le pied d'appui est mal positionné dans l'espace, le fait d'exprimer le centre de masse et les contraintes de contact actives dans le repère absolu engendre un déséquilibre qui peut conduire à l'instabilité. Pour palier à cet inconvénient, il nous faudrait alors connaitre la position relative du centre masse par rapport au pied gauche et au pied droit. De cette façon, même si le pied d'appui est mal positionné dans l'espace, la loi de commande prend pour référence le pied gauche ou le pied droit et limite le déséquilibre. On définit alors un nouveau changement de variable qui prend en considération les arguments précédents. Le tableau (4.3) rend compte de ces changements. Les coordonnées exprimées dans le repère lié au pelvis sont dotées de l'indice P.

Ce changement de variable a été élaboré à l'aide du logiciel de calcul formel Maple. Les procédures de calcul existentes ne permettaient pas de faire ce changement de variable. Il s'est avéré nécessaire de recréer les fonctions qui permettent d'exprimer les coordonées du centre de masse, de la cheville droite et de la cheville gauche dans le repère lié au pelvis. De même, qu'il a été nécessaire de recréer les fonctions qui permettent d'exprimer l'orientation du tronc dans les repères lié au pied gauche et au pied droit.

On désire exprimer la position relative du centre de masse par rapport au pied gauche et au pied droit est exprimé dans le repère lié au pelvis. Or ce dernier ne possède pas la même orientation que le repère absolu. Il est donc important de réorienter les éléments de ce changement de variable suivant les axes du repère absolu.

Ce problème intervient aussi dans l'expression de l'orientation du tronc dans les repères liés au pied gauche et au pied droit. Néanmoins, cette réorientation est plus intuitive que la précédente. Une copie du fichier *commande.maple*, disponible en annexe A, rend compte des modifications apportées.

On remarquera, en outre, que ce changement de variable a pour effet de rendre les coordonnées articulaires $Q_{1...15}$ indépendantes des éléments $q_{16...21}$ donc indépendantes de la position et de l'orientation du robot dans l'espace. Dans sa version actuelle, le robot BIP ne permet pas d'avoir accès à la mesure des éléments $q_{16...21}$ du vecteur q. Il est donc intéressant pour nous de s'affranchir de ces éléments dans le calcul de la loi de commande. De plus, ceci réduit considérablement le temps de calcul de la commande à chaque pas d'échantillonnage.

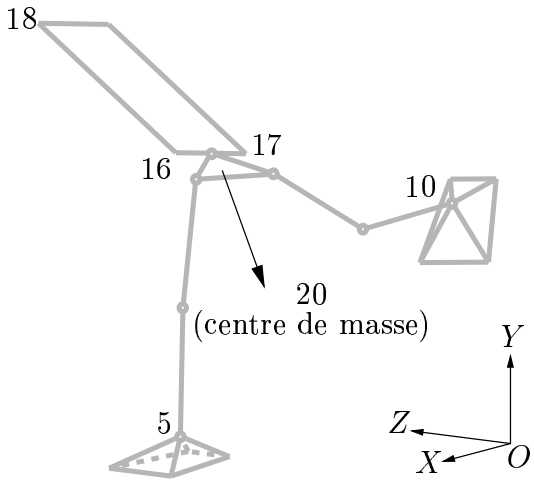


FIG. 4.4 – Quelques points caractéristiques.

$Q_{1...3}$	$\{z_G^P - z_5^P,\ y_5^P - y_G^P,\ x_G^P - x_5^P\}$	position du centre de masse du robot par rapport à la position du pied droit exprimée dans le repère lié au pelvis
$Q_{4...6}$	$\{z_G^P - z_{10}^P,\ y_{10}^P - y_G^P,\ x_G^P - x_{10}^P\}$	position du centre de masse du robot par rapport à la position du pied gauche exprimée dans le repère lié au pelvis
$Q_{7...9}$	$\{x_{16}^P - x_{17}^P,\ z_{17}^P - z_{16}^P,\ z_{18}^P - z_{16}^P\}$	orientation du tronc exprimée dans le repère lié au pied droit
$Q_{10...12}$	$\{x_{16}^P - x_{17}^P,\ z_{17}^P - z_{16}^P,\ z_{18}^P - z_{16}^P\}$	orientation du tronc exprimée dans le repère lié au pied gauche
$Q_{13...15}$	$q_{13...15}$	
$Q_{16...21}$	$q_{16...21}$	

TAB. 4.3 – Définition du nouveau changement de variables.

4.4 Construction d'un Observateur de q_2

On reprendra, ici, les notations imposées par le découpage de l'équation de la dynamique (2.15) ; soit q_1 l'ensemble des 15 variables articulaires et q_2 l'ensemble des 6 variables précisant la position et l'orientation du robot.

Le robot BIP dans sa version actuelle ne dispose pas de capteur lui permettant de déterminer sa position et son orientation dans l'espace. On se propose donc de construire un observateur de q_2 et de le tester en simulation. On notera que le simulateur du robot BIP utilisait jusque là, le vecteur q dans sa totalité. On supposait, en effet, que la mesure de q_2 était toujours accessible. On utilisera maintenant, en simulation, le vecteur $\hat{q}$ construit à partir du vecteur q_1 et de l'observateur $\hat{q}_2$.

$$\hat{q} = \begin{bmatrix} q_1 \\ \hat{q}_2 \end{bmatrix}$$

4.4.1 Une Méthode

Il est en effet possible de reconstruire q_2 à chaque pas d'échantillonnage à partir des positions articulaires q_1, et des contacts actifs du robot sur le sol. Ces contacts représentent l'ensemble des points du pied gauche et/ou du pied droit dont les forces normales de pression sont positives. On cherche ainsi un vecteur q_2 qui minimise la distance entre les contraintes de contact active $\varphi^*(q)$ à l'instant t et la positions des contacts de référence (défini à chaque impact). On défini alors la fonction de coût suivante :

$$f = \frac{1}{2}\left(\varphi^*(q) - \varphi^*_{ref}\right)^2$$

Où $\varphi^*(q)$ désigne l'ensemble des contraintes actives à l'instant t, donné en fonction de q. φ^*_{ref} désigne la position des contacts actifs de références du robot. On notera que la position des contacts est rafraîchie à chaque fois qu'il y a impact ou décollage d'un pied.
On utilise, ensuite sous `Scilab`, une procédure d'optimisation non-linéaire avec pour objectif de minimiser la fonction f. Ce problème prend, alors, la forme suivante :

$$\hat{q}_2 = \min_{q_2} \ \frac{1}{2}\left(\varphi^*(q) - \varphi^*_{ref}\right)^2 \tag{4.1}$$

Cette procédure d'optimisation utilise la fonction f, le gradient ∇f et la valeur initiale de q_2. Elle retourne la valeur q_{2opt}, où q_{2opt} n'est autre que l'observateur $\hat{q}_2$. La figure (4.5) décrit l'organisation algorithmique de la procédure Observateur.

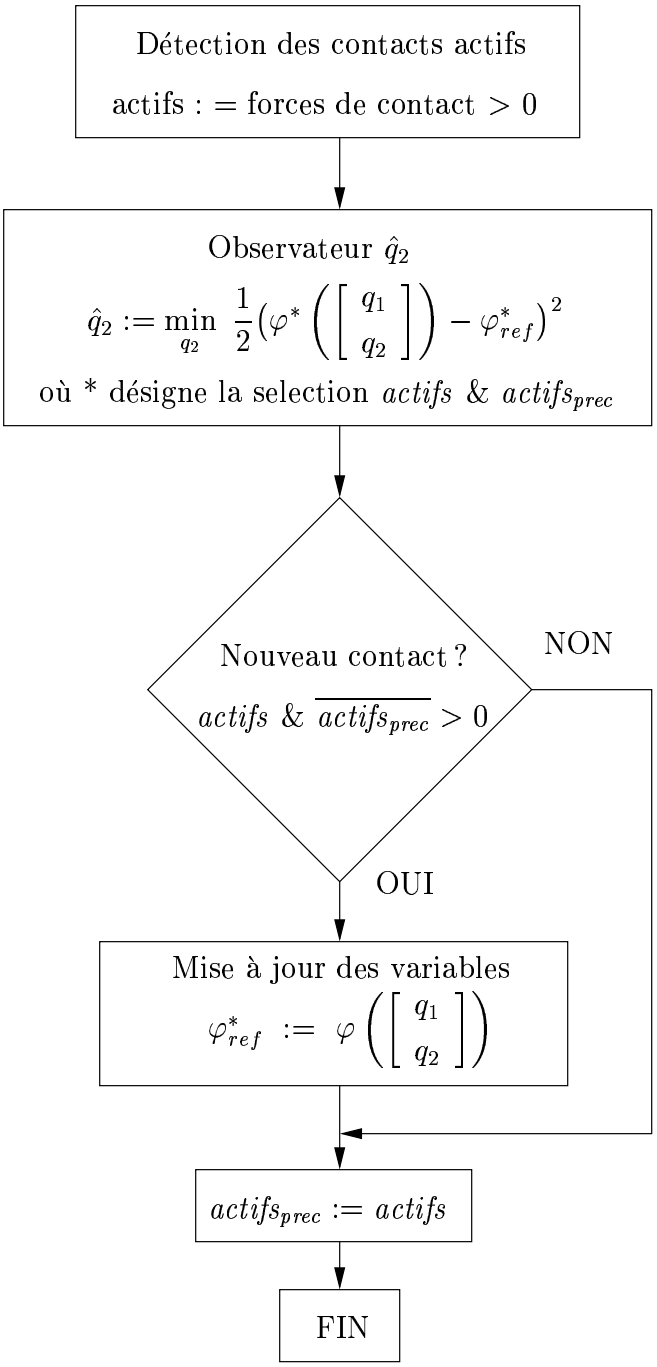


FIG. 4.5 – Description algorithmique du calcul de l'observateur $\hat{q}_2$

On teste, puis on valide cette procédure sur le simulateur du robot BIP. Le calcul de la commande se fait alors non plus par rapport au vecteur q mais par rapport au vecteur $\hat{q}$.
Le lecteur pourra se reporter au fichier *observateur.sci* et *commande.sci*, disponible en annexe A.

Cependant, le calcul de cet observateur utilise une procédure d'optimisation non-linéaire qui prend un temps de calcul non négligeable dans notre cas. En effet, cette procédure est itérative, elle part d'une valeur initiale et converge vers la valeur optimale souhaitée lorsque le gradient de la fonction de coût tend vers zéros. Ceci est effectué à chaque pas d'échantillonnage (la période est de l'ordre de 10 ms), on voit alors la necessité de diminuer ces temps de calcul.

4.4.2 Une autre possibilité

On désire maintenant contrôler le nombre d'itération pour le calcul de l'observateur de q_2. On reprend pour cela le problème non-linéaire 4.1, mais on travaille maintenant avec un vecteur $q_2 + \delta q_2$, où q_2 est fixé. Ce problème s'écrit alors de la façon suivante :

$$\hat{q}_2 = \min_{\delta q_2} \frac{1}{2} \left(\varphi^* \left(\begin{bmatrix} q_1 \\ q_2 + \delta q_2 \end{bmatrix} \right) - \varphi^*_{ref} \right)^2$$

On réalise ensuite le développement limité au premier ordre de la fonction φ^* :

$$\varphi^* \left(\begin{bmatrix} q_1 \\ q_2 + \delta q_2 \end{bmatrix} \right) = \varphi^* \left(\begin{bmatrix} q_1 \\ q_2 \end{bmatrix} \right) + \frac{\partial \varphi^*}{\partial q_2} \left(\begin{bmatrix} q_1 \\ q_2 \end{bmatrix} \right) \cdot \delta q_2 + o(\|\delta q_2\|)^2 \tag{4.2}$$

Afin d'alléger les notations on notera J^T le jacobien $\dfrac{\partial \varphi^*}{\partial q_2} \left(\begin{bmatrix} q_1 \\ q_2 \end{bmatrix} \right)$
On remplace, pour finir, φ^* par l'expression 4.2. On obtient alors le problème quadratique linéaire suivant :

$$\hat{q}_2 = \min_{\delta q_2} \frac{1}{2} \left(\varphi^* \left(\begin{bmatrix} q_1 \\ q_2 \end{bmatrix} \right) + J^T \delta q_2 - \varphi^*_{ref} \right)^2$$

On peut résoudre ce probème de façon analytique. Développons l'expression précédente puis dérivons là par rapport à δq_2. On obtient alors :

$$J^T J \delta q_2 + J^T \left(\varphi^* \left(\begin{bmatrix} q_1 \\ q_2 \end{bmatrix} \right) - \varphi^*_{ref} \right) = 0$$

$$\Leftrightarrow \delta q_2 = -(J^T J)^{-1} J^T \left(\varphi^* \left(\begin{bmatrix} q_1 \\ q_2 \end{bmatrix} \right) - \varphi^*_{ref} \right)$$

Bien entendu on considère δq_2 petit, il faut donc choisir q_2 proche de la solution du problème. C'est pourquoi on choisit comme vecteur q_2, le vecteur $\hat{q}_2$ précédent.

$$\boxed{\hat{q}_2 = \hat{q}_{2p} - (J^T J)^{-1} J^T \left(\varphi^* \left(\begin{bmatrix} q_1 \\ \hat{q}_2 \end{bmatrix} \right) - \varphi^*_{ref} \right)}$$

Où $\hat{q}_{2p}$ est la valeur optimale calculée lors de l'itération précédente.

Cette démarche nous permet de déterminer un observateur de q_2, mais pour obtenir une valeur optimale ou du moins proche de cette valeur, il est nécessaire de faire plusieurs itérations. Ce n'est qu'après quelques essais en simulation que l'on choisira le nombre d'itérations. On notera que la valeur de $\hat{q}_2$ précédent est rafraîchie à chaque appel de la procédure observateur et que l'utilisation de cette nouvelle procédure est identique à la précédente. Le lecteur pourra se reporter aux fichier *observateur2.sci* et *commande.sci*, disponible en annexe A.

4.4.3 Comparaison des différentes méthodes

On choisit de tester trois méthodes de calcul pour notre observateur. Ces trois méthodes sont les suivantes :

- Observateur 1 : Méthode d'optimisation non-linéaire.
- Observateur 2 : Méthode développée à la section 4.4.2 sur une itération.
- Observateur 3 : Méthode développée à la section 4.4.2 sur deux itérations.

On teste ces trois procédures de calcul sur le simulateur du robot BIP. On choisit de tester la procédure *Observateur 3* de deux manières différentes. On teste cette procédure en faisant d'abord deux itérations. Le calcul du jacobien des points de contact actifs J^T se fait alors à chaque itération. On teste ensuite cette même procédure sur deux itérations mais en ne calculant le jacobien des points de contact actifs qu'une seule fois lors de la première itération. On notera *Observateur 3* et *Observateur 3bis* ces deux méthodes de calcul. Notons aussi que la simulation se fait sur un temps simulé de 8 secondes et possède une période d'échantillonnage de 20 ms soit 400 pas d'échantillonnage. On choisit de s'intéresser aux temps de calcul et aux nombres d'itérations de chacune des procédures. On verifiera aussi la norme $\|q_2 - \hat{q}_2\|$. Les résultats de simulation sont disponible dans le tableau (4.4).

On remarque que l'Observateur 1, 3 et 3bis atteignent la même précision si l'on se réfère à l'erreur maximal de la norme $\|q_2 - \hat{q}_2\|$ relevée pour les 400 pas d'échantillonnage. Cependant, le temps de calcul est bien plus intéressant en utilisant la procédure Observateur 3bis. C'est cette procédure de calcul que l'on retiendra pour reconstruire q_2 à chaque pas d'échantillonnage. Les figures (4.6) et (4.6) permettent de voir l'évolution de la norme $\|q_2 - \hat{q}_2\|$ au cours de la simulation. On remarque, pour toutes les procédures sauf la seconde, que la norme $\|q_2 - \hat{q}_2\|$ évolue de la même manière, seul le temps de calcul les différencie.

On peut observer sur la première courbe les phases de changements de support. Le passage du double support au simple support se manifeste par une discontinuité et le passage du simple

	Temps de calcul	nb d'itérations	$\max \|q_2 - \hat{q}_2\|$
Observateur 1	$11\,ms$	~ 28	$1,4.10^{-7}$
Observateur 2	$0,5\,ms$	1	$9,8.10^{-5}$
Observateur 3	$1\,ms$	2	$1,4.10^{-7}$
Observateur 3bis	$0,7\,ms$	2	$1,4.10^{-7}$

TAB. 4.4 – Performance numérique des procédures de calculs de l'observateur $\hat{q}_2$.

support au double support se manifeste par un plateau. On peut aussi observer une certaine dérive qui augmente avec le temps.

On teste la loi de commande incluant le nouveau changement de variable en simulation. Le début de la simulation se passe pour le mieux, mais au moment de lever le pied gauche pour effectuer un pas, on constate que l'avant du pied reste en contact avec le sol et ne permet pas de faire ce pas. Ce phénomène est dû à la simulation proprement dite. Les évennements tels que les prises et les pertes de contact sont discret. Il est alors difficile de gérer une dynamique continue avec de tels évennements discrets. On se rend bien compte des limites du simulateur que l'on utilise actuellement. On se propose d'ailleurs pour la suite de travailler en collaboration avec le Laboratoire de Mécanique et Génie Civil (LMGC) de Montpellier pour l'utilisation du simulateur générique LMGC90.

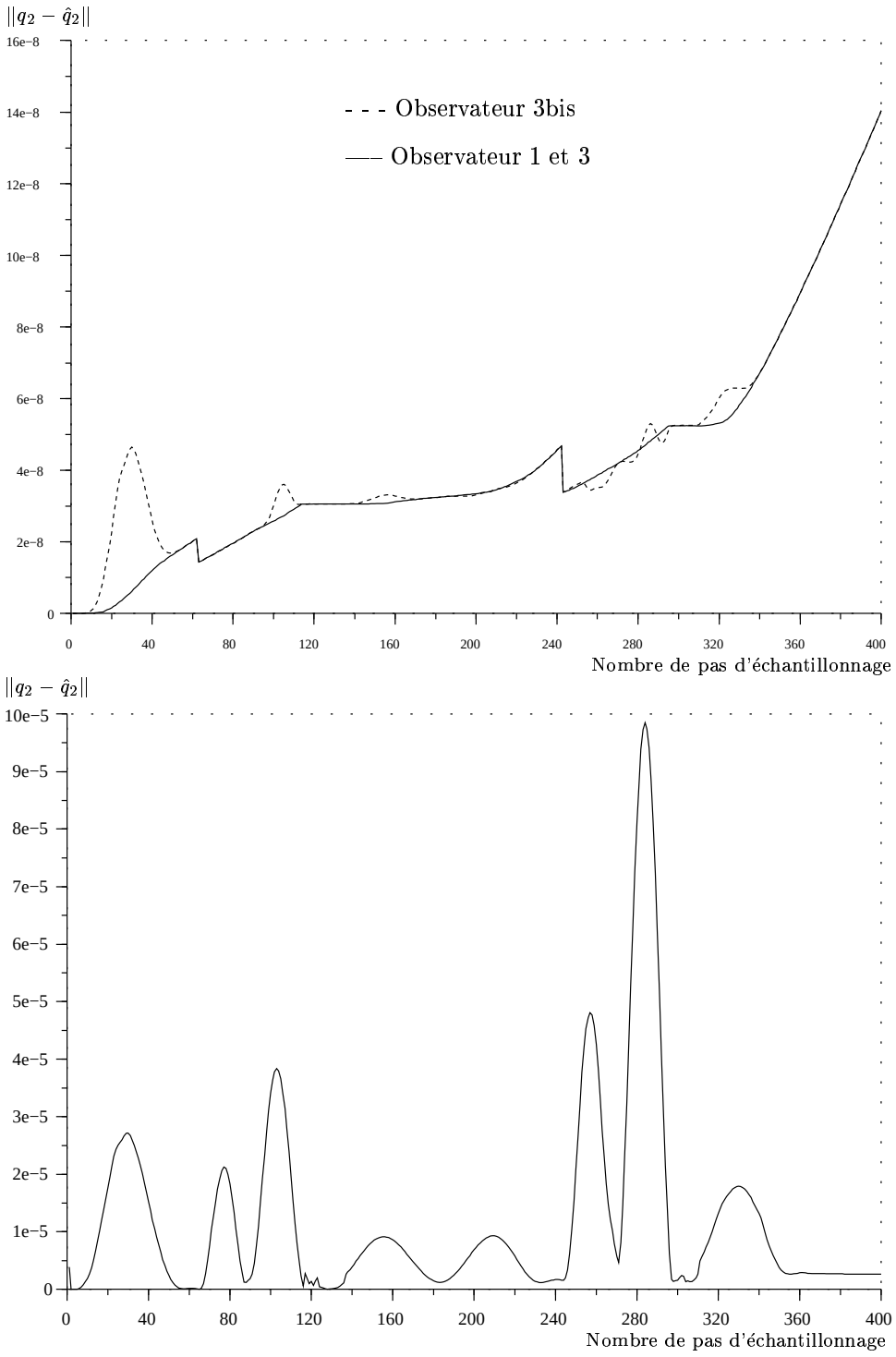


FIG. 4.6 – Evolution de la norme $\|q_2 - \hat{q}_2\|$ pour l'observateur 1, 2, 3 et 3bis.

Deuxième partie

Étude et Développement Expérimental

Chapitre 5

La plateforme expérimentale

5.1 Architecture Générale

Ce chapitre s'appuie sur les références [2], [3], [13], [21], [22] [23] et [24].
La plateforme expérimentale que constitue le robot BIP est composée de trois parties :

- la partie locomotion (jambes et pelvis) du robot
- l'armoire de commande
- le contrôleur de robot

Dans sa version actuelle, l'armoire de commande est fixée sur le pelvis et tient lieu de tronc (Cf figure 5.1).

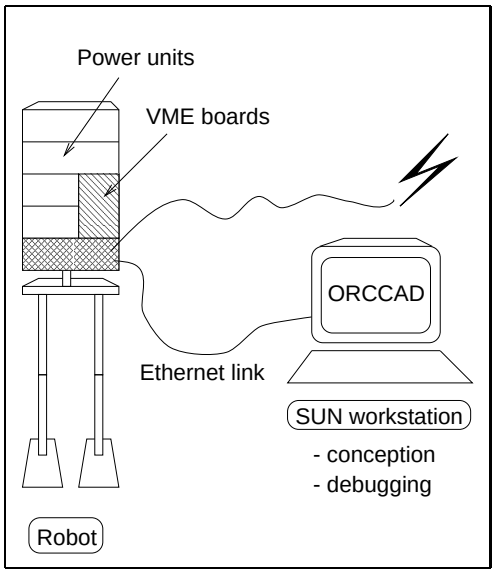


FIG. 5.1 – Architecture du robot Bip2000

5.2 La Structure mécanique

La structure mécanique du robot peut se décomposer en deux parties : tout d'abord les jambes avec chacune quatre articulations (celle de la hanche dans le plan sagital, celle du genou et les deux de la cheville), puis le pelvis supportant l'armoire de commande et sur lequel sont fixées les jambes. Le pelvis est doté de 7 degrés de liberté : les articulations complétant les rotules des hanches et la rotule du tronc (3 ddl).

La structure mécanique du robot bipède est réalisée en aluminium. Elle comprend environ 280 pièces ; elles ont toutes été conçues, et en grande partie réalisées, par le LMS (Laboratoire de Mécanique des Solides) à l'Université de Poitiers.

5.3 La Chaîne électromécanique

Les actionneurs

Le robot bipède possède 15 articulations entraînées par 15 moteurs à courant continu sans balai (*brushless*) et pilotés par 15 variateurs (SBS). Les ensembles moteur/variateur sont appairés et fournis par la société Parvex. Les études biomécaniques sur les couples mis en jeu dans les articulations humaines sont à la base du calcul des caractéristiques de ces moteurs [3].

Les variateurs

Les variateurs (ou module SBS) sont des servoamplificateurs de vitesse pour moteur synchrone auto-piloté avec utilisation d'un resolver comme capteur de position et de vitesse. Il assure la régulation en courant et en vitesse, la commande de puissance et les fonctions de sécurité. Actuellement, les variateurs sont configurés en régulation de courant seule (la régulation de vitesse est désactivée).

En outre, tous les variateurs sont équipés de l'option carte d'émulation codeur. Ces cartes d'émulation codeur sont configurées en résolution maximale de 1024 tops par tour. Cette résolution est multipliée par 4, soit 4096 tops par tour en comptabilisant les transitions sur 2 voies. Ils fournissent donc une position angulaire relative des axes moteurs. Connaissant la position initiale, on peut alors en déduire la position absolue.

Les transmissions

Le robot BIP compte cinq articulations équipées de réducteurs *harmonic-drives* (rotations z_6, z_7, z_8, z_9, z_{15} figure 2.1).

Les autres transmissions sont des systèmes de vis-écrou à rouleaux satellites avec biellettes (figure 5.2). Ces transmissions ont fait l'objet d'un brevet.

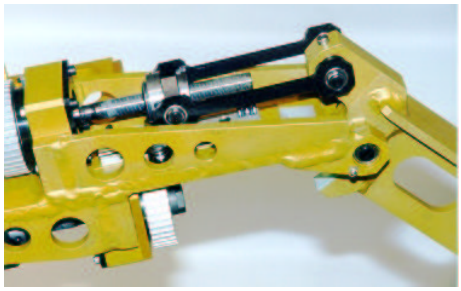


FIG. 5.2 – Système de transmission au niveau du genou

Les chevilles (figure 5.3) du robot et la liaison tronc/bassin (figure 5.3) sont constituées de deux ensembles moteur-transmission en parallèle. Ces ensembles bougent simultanément pour donner un mouvement sagittal ou/et frontal.

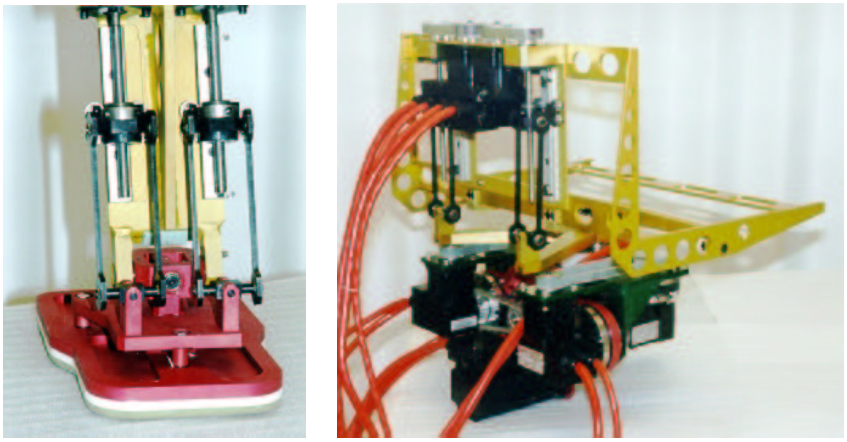


FIG. 5.3 – Robots parallèles de la cheville et du tronc

Les capteurs

Trois capteurs d'effort placés sur chaque pied permettent de mesurer la composante verticale de la force de réaction du sol (force de pression), les deux composantes du moment de cette force dans le plan horizontal et la position du centre de pression. On utilise aussi des potentiomètres qui nous permettent de déterminer la position absolue du robot à sa mise sous tension et donc d'utiliser les informations codeur en provenance des variateurs.

Ces deux types de capteurs feront l'objet d'une étude plus approfondie dans la suite de ce document.

5.4 L'armoire de commande

Le but de cette armoire est d'embarquer toute l'électronique de puissance et de commande. Elle contient :

1. l'électronique de puissance (15 variateurs SBS et leur alimentation ABS).
2. le calculateur et son alimentation.
3. les modules d'entrées/sorties analogique, digitale et quadrature permettant d'interfacer le calculateur et les variateurs.
4. les modules d'entrées/sorties analogiques et digitales pour la prise en compte des capteurs proprioceptifs (potentiomètres associés aux axes et capteurs de force) avec la connectique associée
5. l'électronique de gestion de l'arrêt d'urgence et des sécurités.

L'armoire sert aussi d'interface avec l'extérieur. En effet, elle est relié via un cordon ombilical comprenant la liaison ethernet avec la station hôte, l'alimentation secteur 220 V monophasée et l'alimentation 135V triphasée. L'architecture electronique de l'armoire est donnée figure 5.4

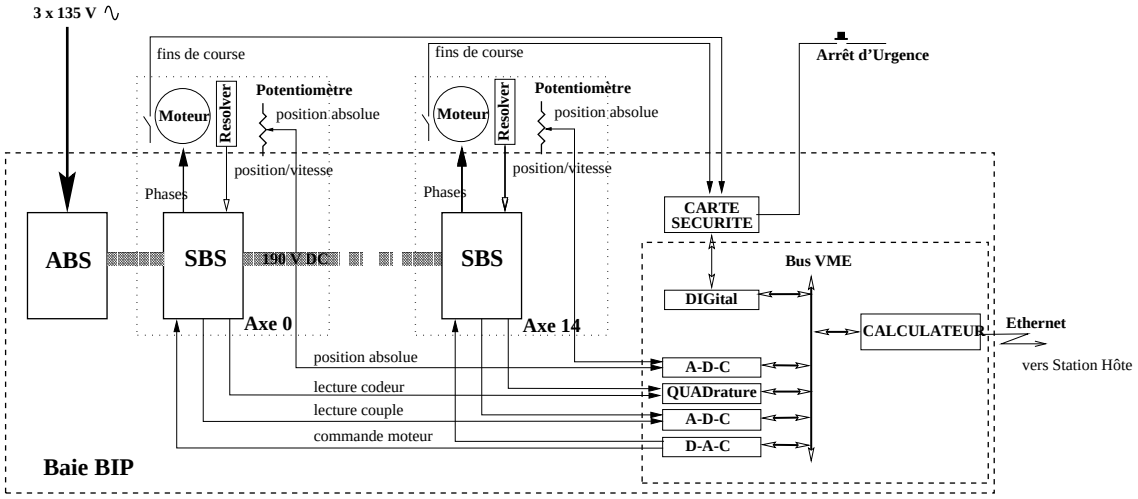


FIG. 5.4 – Schéma synoptique de l'architecture électronique de la baie

5.5 Architecture Informatiques et logicielles

Le robot bipède utilise une carte CPU embarquée MOTOROLA MVME162 avec le système d'exploitation VxWorks. Les programmes sont d'abord compilés de façon croisée sur stations SUN solaris puis une commande de VxWorks télécharge le code sur la carte cible MVME162.

Pour des raisons d'origine de création ou de maintenabilité, différentes couches logicielles interviennent dans les programmes développés sur le robot bipède. De manière incrémentale, on trouve :

1. les librairies livrées avec les modules d'entrée/sortie
2. une librairie permettant d'utiliser l'ensemble des entrées/sorties de la baie, en masquant les différentes origines ou la multiplicité des modules utilisés. Il s'agit ici d'une factorisation et d'une homogénéisation de différents codes.
3. une librairie permettant d'utiliser les ressources du robot bipède. A ce niveau les fonctions sont utilisables pour commander les différents axes du robot bipède.

Concrètement, voici les différents modules VxWorks à télécharger pour commencer à travailler :

```
ld < /home/mouflon/jarde/BIP/lib/t501drv.vxo
ld < /home/mouflon/jarde/BIP/lib/libRdacsu.vxo
ld < /home/mouflon/jarde/BIP/lib/libRquadrature.vxo
ld < /home/mouflon/jarde/BIP/lib/libRunidig.vxo
ld < /home/mouflon/jarde/BIP/lib/libR16adc.vxo
ld < /home/mouflon/jarde/BIP/lib/utilBip15.vxo
ld < /home/mouflon/jarde/BIP/lib/drvBip.vxo
```

Cependant, la programmation de tâche-robots complexes se fait au travers de l'environnement de programmation ORCCAD depuis une station de travail (fig.5.1). Ces Tâches Robots sont des boucles d'asservissement qui permettent au robot bipède de se maintenir debout, de faire un pas, etc ... A l'initialisation, les différents programmes élaborés sur ORCCAD sont transférés, via la liaison Ethernet, sur le processeur de la carte embarquée sur le tronc, la station ne sert ensuite qu'à la lecture des signaux entrée-sortie du robot.

5.6 L'environnement de programmation Orccad

Orccad[24] (**O**pen **R**obot **C**ontroller **C**omputer **A**ided **D**esign) est une architecture de type contrôle-commande dont l'objectif est de faciliter l'implémentation de lois de commande et la programmation de missions robotiques complexes notamment dans des environnements peu structurés. Il permet également de spécifier et valider des missions robotiques à réaliser par le système. Cet outil a été développé à l'INRIA [17].

5.6.1 Méthodologie

La méthodologie d'Orccad est basée sur deux niveaux d'abstraction :
- le niveau fonctionnel
- le niveau contrôle

La communication entre ces deux niveaux d'abstraction se fait à l'aide d'événements discrets qui permettent de lancer des actions ou de les interrompre, et qui informent de l'état de l'exécution des actions.

Niveau fonctionnel

Ce niveau manipule une entité appelée *Tâche Robot* (*TR*). Une *TR* est la spécification complète et paramétrable d'une *Ressource Physique*, d'une *Loi de Commande*, et d'un *Comportement Logique*.

La Ressource Physique

Elle représente le système mécanique que l'on souhaite commander. Elle contient les informations provenant du robot. Elle constitue l'interface entre la loi de commande et le robot. Elle permet l'envoi de consignes aux actionneurs et fournit les informations issues des capteurs.

La Loi de Commande

Elle détermine la consigne à envoyer à chaque pas d'échantillonnage aux actionneurs du robot en fonction des informations provenant de la *Ressource Physique*. La spécification de la *Loi de commande* se fait grâce à l'enchaînement de *Modules Algorithmiques* qui communiquent à travers des *Ports de Données*.

Le Comportement Logique

Il gère le contrôle de la TR et utilise trois types de signaux :

1. les *Préconditions*, qui doivent être vérifiées avant que la TR ne démarre.
2. les *Postconditions* qui représentent les événements émis par la TR lorsqu'elle se termine normalement
3. les *Exceptions*.

Les *Exceptions* sont aussi au nombre de trois :

1. *T1* : elles sont générées par la TR elle-même et signifient à la tâche courante qu'elle doit changer son mode de fonctionnement.
2. *T2* : elles dictes à la tâche courante qu'elle doit arrêter son exécution. Le contrôle est alors transféré au niveau supérieur (niveau contrôle).
3. *T3* : elles entraînent l'arrêt immédiat du système.

Niveau contrôle

Ce niveau manipule des entités appelées *Procédures Robot* (ou *PR*) qui permettent de composer logiquement et hiérarchiquement des *Tâches Robot* et d'autres *PR* pour spécifier des actions et des applications. Ce niveau manipule des événements discrets. Ce sont ces événements (signaux) qui permettent la communication entre les niveaux fonctionnel et de contrôle. La spécification sous forme de Procédure Robot se fait dans le langage Esterel.

5.6.2 Environnement de programmation

On distingue quatre étapes dans le cycle de programmation d'une application robotique :

1. la *Spécification*,
2. la *Vérification*,
3. la *Simulation*,
4. l'*Exécution*.

ORCCAD propose pour chacune de ces phases une interface graphique qui intègre des outils dédiés. On ne décrit ici que les étapes de spécification et d'execution. Le lecteur pourra se reporter aux références [17, 2] pour plus de renseignements.

La Spécification

Pour la spécification on distingue trois types de modules :
- les *Modules ressources physiques*
- les *Modules algorithmiques*
- les *Modules automates*

Ces modules se présentent sous la forme de schéma-bloc avec une architecture interne bien défini. A chaque types de modules on associe des fichiers C.

Modules ressources physiques

Les fichiers de code associés à un module ressource physique sont :
- *inc.h* fichier include : contient les déclarations externes et les *include*.
- *var.c* fichier variables : contient la déclaration des variables internes utilisées dans les autres codes.
- *init.c* fichier initialisation : ce programme initialise les moteurs ou les capteurs.
- *reinit.c* fichier de réinitialisation : ce fichier réinitialise les moteurs et capteurs (envoi de nouveaux paramètres).
- *end.c* fichier de fin : ce programme coupe les moteurs ou les capteurs.

Modules algorithmiques

Les fichiers de code associés à un module algorithmique sont :
- *inc.h* fichier include : contient les déclarations externes et les *include*.
- *var.c* fichier variables : contient la déclaration des variables internes utilisées dans les autres codes.
- *init.c* fichier initialisation : ce programme initialise les variables.
- *compute.c* fichier de calcul : programme l'algorithme qui produit les variables de sortie ou les évènements à partir des variables d'entrée.
- *end.c* fichier de fin : libère les allocations mémoire.

Modules automates

Ce module spécifie le comportement logique de la TR. Il gère les exceptions défini auparavant.

Ainsi, la spécification de TR se fait graphiquement en ajençant les différents schémas-blocs requis (*Ressource Physique, Module Algorithmique, Comportement Logique*). Ces blocs communiquent grâce à des liaisons entre leurs *Ports*.

Pour la programmation en Esterel des PR, Orccad propose à l'utilisateur un éditeur qui facilite la programmation des applications. Cet éditeur permet l'insertion automatique du code généré pour chaque TR et chaque PR.

Exécution

L'*Éxécution* correspond à la traduction de la spécification de l'application en code C++ temps-réel. Orccad dispose d'une interface qui permet de définir les informations nécessaires à l'exécution.

Chapitre 6

Choix d'un calculateur

Le robot BIP dispose d'une carte CPU embarquée permettant de réaliser les calculs de la loi de commande. Cette carte fut dimensionnée pour la version du robot BIP à 8 DDL. La loi de commande associée a permis au robot d'effectuer une marche statiquement stable 2D sur sol plat. On se place maintenant dans le cas où le robot possède 15 DDL et on désire appliquer une loi de commande permettant au robot d'effectuer une marche dynamique 3D stable sur sol plat. Le calculateur embarqué dispose d'une puissance trop limitée pour la loi de commande développée en simulation. On décide donc de remplacer cette carte par un autre calculateur. Néanmoins le choix d'un tel calculateur passe par une étude approfondie des possibilités qui s'offrent à nous. Il s'agira dans un premier temps de déterminer la solution adéquate pour nos besoins en temps de calcul puis de mettre en œuvre la solution apportée. Néanmoins, notre objectifs principal reste de faire marcher le robot BIP dans les plus brefs délais. On donnera donc une solution à court terme puis une solution à long terme nous permettant de faire évoluer la plate-forme expérimentale du robot.

6.1 Configuration actuelle

La version actuelle du robot BIP dispose d'une carte CPU MVME162-522A au format industriel (6U), munie d'un processeur Motorola MC68040 [25] à 32 MHz et de 8 Mo de RAM. Cette carte fonctionne avec le bus VME et dispose de quatres slots pour modules IP.

A cause de contraintes temps réels importantes, on utilise le système d'exploitation temps réel VxWorks [26]. Le développement logiciel du code embarqué se fait à l'aide de TORNADO, outils logiciels de WIND RIVER Systems [19, 20]. On utilise néanmoins ORCCAD pour le développement des applications logicielles du robot.

6.2 ORCCAD

Nous avons vu au chapitre précédent que ORCCAD permettait de définir des actions élémentaires (Tâche Robot) à travers une interface graphique de schéma-bloc (Module) et de les séquencer (Procédure Robot) en utilisant le langage ESTEREL. Il permet aussi, de générer le code exécutable temps réels C++ associé. Ce code est mono-cadence et mono-processeur. En d'autre terme, le code algorithmique d'une action élémentaire est exécuté dans une seule tâche temps réel périodique sur un seul processeur.

ORCCAD autorise la génération de code exécutable temps réels C++ pour les cibles suivantes :

- VxWorks
- Solaris 2.6
- Linux

La portabilité de l'environnement de développement ORCCAD sur la cible temps réel RTAI est actuellement en cours de réalisation. Olivier TESTA, ingénieur de l'équipe BIP, ajuste actuellement la génération de code C++ temps réel pour cette cible sur processeur INTEL.

Il a été question, un moment donné, de se passer d'ORCCAD pour la programation temps réel de tache robot. Il a été proposé d'écrire, à la main, le code C++ temps réel nécessaire pour la commande du robot. Mais l'aisance que procure ORCCAD lors de la programmation et de la génération automatique de code C++ temps réel fait de ce logiciel un outil indispensable pour l'implantation de loi de commande sur le robot BIP. On décide ainsi de ne pas se passer de ORCCAD.

6.3 Etendue des possibilités

6.3.1 Choix du materiel informatique

En l'état actuel des choses, nous disposons de deux alternatives. La première consiste à conserver un calculateur embarqué sur le robot BIP avec toute les contraintes hardware que cela impose (format industriel 6U, bus VME, module IP...). Les Moyens Robotiques disposent d'une carte CPU VSBC 6862 au format industriel 6U, munie d'un processeur PowerPC à 200 MHz et de 32 Mo de RAM. Cette carte fonctionne avec le bus VME et dispose de quatres slots de modules IP. Cette carte est fournie avec le système d'exploitation Linux.

La seconde alternative consiste à utiliser un calculateur déporté via une liaison Ethernet. En ce qui concerne le type de calculateur, le choix n'est pas arreté. Peu importe pour l'instant son architecture pourvu qu'il satisfasse le cahier des charges imposé (temps de calcul de la

loi de commande) et une durée de communications Ethernet bien inférieure à la période d'échantillonnage.

On décide, pour des raisons de coût, de ne pas acheter un autre calculateur embarqué plus puissant. On ne retiendra donc que les deux configurations matérielles citées plus haut.

6.3.2 Choix d'un système d'exploitation

Il nous faut maintenant déterminer le système d'exploitation que l'on pourrait utiliser sur l'une ou l'autre des configurations matérielles énoncées plus haut. Il apparaît que quelque soit la configuration hardware que l'on choisira, nous serons amenés à choisir un système d'exploitation parmi les trois suivant :

- VxWorks
- RTAI ou RTLinux
- Linux

En effet, on a décidé de ne pas se passer de ORCCAD. Le choix est donc restreint aux cibles temps réels compatibles avec ORCCAD.

Il paraît évident que l'OS temps réel VxWorks est le plus adapté à nos besoins. Or, cet OS temps réel a un coût non négligeable. Le système de développement de base multi-utilisateur (Tornado + chaine de compilation + TCP/IP) et la licence VxWorks s'élève à environ 25000 €. Ainsi pour des raisons de coût, on préfère se passer de VxWorks et se tourner vers un OS temps réel tel que RTAI où les droit d'exploitation sont libre.

Cependant, le portage de ORCCAD sur RTAI est encore en cours de réalisation. Nous sommes donc contraint d'attendre la réalisation de ce travail pour pouvoir utiliser RTAI sur l'une ou l'autre des configurations matérielles. De plus, utiliser cet OS temps réel sur un processeur type PowerPC, conduira à des modifications supplémentaires dans la génération de code, notamment au niveau de l'addressage hardware.

Une alternative à ce délai serait, dans l'immédiat, de tester une loi de commande en temps réel mou sous Linux. En effet, nous avons vu précédamment qu'ORCCAD était utilisable sur Linux.

6.4 Conclusion

La solution à long terme la plus probante sera le choix d'un calculateur déporté via une liaison Ethernet avec comme système d'exploitation RTAI Linux. On préfèrera le calculateur déporté plutôt que la carte embarquée pour des raisons de coût et d'évolutivité.

La solution à court terme sera de déporter le calcul de la loi de commande sur un serveur distant tout en gardant l'architecture informatique actuelle. Cette option nécessite de faire

des tests suplémentaires afin de s'assurer que les temps de communications réseau sont aussi courts que possible. On désire en effet garder une période d'échantillonnage de l'ordre de 10 ms, il nous faut donc des temps de communications courts et prendre en compte les temps de calculs CPU de la loi de commande (Cf tableau 3.2). Ce sera l'objet du chapitre suivant.

Chapitre 7

Tests de communications entre sockets

Nous avons vu dans le chapitre 4, que l'on ne pouvait pas assurer le calcul de la loi de commande, sur une période d'échantillonnage, avec le calculateur embarqué actuel. On a donc décidé d'utiliser un calculateur déporté relié au calculateur du robot via une liaison Ethernet. On se propose, ici, de tester les temps de communications entre une socket client, attachée au calculateur du robot et une socket serveur attachée à une machine déportée.

7.1 Le modèle CLIENT-SERVEUR

Les protocoles de communications comme TCP/IP permettent la communication point à point entre deux applications s'exécutant éventuellement sur deux machines différentes.

Le détail du transfert effectif des données entre deux applications est spécifié par le protocole de communication de la couche transport, mais le moment et la façon dont les applications interagissent entre elles sont laissés à la charge du programmeur. L'architecture client/serveur est devenue la méthode incontournable pour la communication point à point au niveau applicatif, quand le protocole utilisé est de la famille TCP/IP.

Ce modèle est motivé par le fait que TCP/IP ne fournit aucun mécanisme permettant l'exécution automatique d'un programme à l'arrivée d'un message si bien que dans une communication point à point, l'une des applications doit attendre l'initiative de la communication de la part de l'autre application. Les applications peuvent être classées en deux catégories :

- Les applications **clientes** : elles prennent l'initiative du lancement de la communication, c'est à dire demandent l'ouverture d'une connexion, envoient une requête, attendent la réponse à la requête et reprennent l'exécution du programme.
- Les applications **serveurs** : elles attendent une demande d'ouverture de connexion, attendent la réception d'une requête et renvoient une réponse.

7.2 Introduction aux sockets

La notion de socket a été introduite dans les distributions Berkeley. Son but est de fournir un modèle général de communication inter processus (IPC - Inter Processus Communication) afin de permettre à divers processus de communiquer aussi bien sur une même machine qu'à travers un réseau TCP/IP. Cela veut dire qu'une socket est utilisé pour permettre aux processus de communiquer entre eux de la même manière que le téléphone ou le service postal nous permet de communiquer entre nous. Cette analogie sera utilisée pour éclaircir la notion de socket.

On peut distinguer deux types principaux de communication : le téléphone et le courrier. Ces modes de communication sont caractérisés tous deux par le fait que l'une des parties prend l'initiative d'établir la communication (ceci implique bien sûr que l'autre lui a fourni un moyen de la joindre : Adresse ou numéro de téléphone). Par contre, ils fonctionnent sur des principes radicalement différents :

- Pour le téléphone, après une phase de connexion, l'interlocuteur reçoit la conversation dans l'ordre et séquentiellement.
- Pour le courrier, on envoie la lettre d'un bloc (le destinataire ne voit pas arriver le message caractère par caractère) ; De plus, deux lettres envoyées à des moments différents peuvent se croiser en chemin, voire même se doubler.

On distingue donc clairement deux types de communication :

- Le mode **connecté** (comparable à une communication téléphonique), utilisant le **protocole TCP**. Dans ce mode de communication, une connexion durable est établie entre les deux processus, de telle façon que la socket de destination n'est pas nécessaire à chaque envoi de données.
- Le mode **non connecté** (analogue à une communication par courrier), utilisant le **protocole UDP**. Ce mode nécessite l'adresse de destination à chaque envoi, et aucun accusé de réception n'est donné.

Les sockets sont généralement implémentées en langage C, et utilisent des fonctions et des structures disponibles dans la librairie <sys/socket.h>.

Les sockets sont utilisables juste au-dessus de la couche **transport** du modèle **OSI**.

7.3 Déroulement d'une communication

Comme dans le cas de l'ouverture d'un fichier, la communication par socket utilise un descripteur pour désigner la connexion sur laquelle on envoie ou reçoit les données. Ainsi la première opération à effectuer consiste à appeler une fonction créant une socket et retournant un descripteur (un entier) identifiant de manière unique la connexion. Ainsi ce descripteur est passé en paramètres des fonctions permettant d'envoyer ou de recevoir des informations

à travers la socket.

L'ouverture d'une socket se fait en deux étapes :

- La création d'une socket et de son descripteur par la fonction *socket()*. Cela permet de spécifier la famille de protocole utilisé (AF_INET pour TCP/IP ou AF_UNIX pour les communications UNIX en local sur une même machine) et le type de service désiré (SOCK_STREAM pour une communication par flot de données (protocole TCP) et SOCK_DGRAM pour une communication par blocs de données (protocole UDP)).
- La fonction *bind()* permet de lier la socket à un point de communication défini par une adresse et un port.

On ne décrit ici que le cas d'une communication en mode connecté (TCP), ce qui sera le cas lors des communications entre robot (client) et calculateur déporté (serveur).

Un serveur doit être à l'écoute de messages éventuels. Toutefois, l'écoute se fait différemment selon que la socket est en mode connecté (TCP) ou non (UDP). En mode connecté le message est reçu d'un seul bloc. La fonction *listen()* permet de placer la socket en mode passif (à l'écoute des messages). En cas de message entrant, la connexion peut être acceptée grâce à la fonction *accept()*. Lorsque la connexion a été acceptée, le serveur reçoit les données grâce à la fonction *recv()*. La fin de la connexion se fait grâce à la fonction *close()*. Le schéma d'une communication client/serveur en mode connecté est donnée figure (7.1).

Pour obtenir une description plus détaillé sur l'API socket et ces fonctions (*socket(), bind(), listen(), accept(), connect(), write(), read(), ...*) le lecteur pourra se reporter à l'ouvrage de référence [14].

7.4 Mise en œuvre d'applications client/serveur

7.4.1 Objectifs

On souhaite établir une communication entre client (le calculateur du robot) et serveur (un calculateur déporté quelconque). Cette comunication aura pour but de soulager le calculateur du robot. En effet, le calcul de la loi de commande ne sera plus fait par celui-ci mais par un autre calculateur, déporté et relié au robot via une liaison Ethernet.

A terme, on se propose d'établir une connexion avec un serveur distant, de lui envoyer les vecteurs q, $\dot{q}$ et λ qui décrivent respectivement l'ensemble des positions articulaires du robot, la vitesse de rotation de chacune d'entre-elles et l'ensemble des forces de contacts à chaque pas d'échantillonnage. Le serveur récupère alors ces vecteurs et se charge du calcul de la commande. Il renvoie, pour finir, au robot les consignes de couples à appliquer sur chacune des articulations du robot. Ensuite le calculateur du robot reprend la main et se charge de transformer ces consignes de couples en tensions et de les acheminer aux variateurs du robot. Une fois la tâche-robot effectuée, on ferme la communication entre le robot et le serveur.

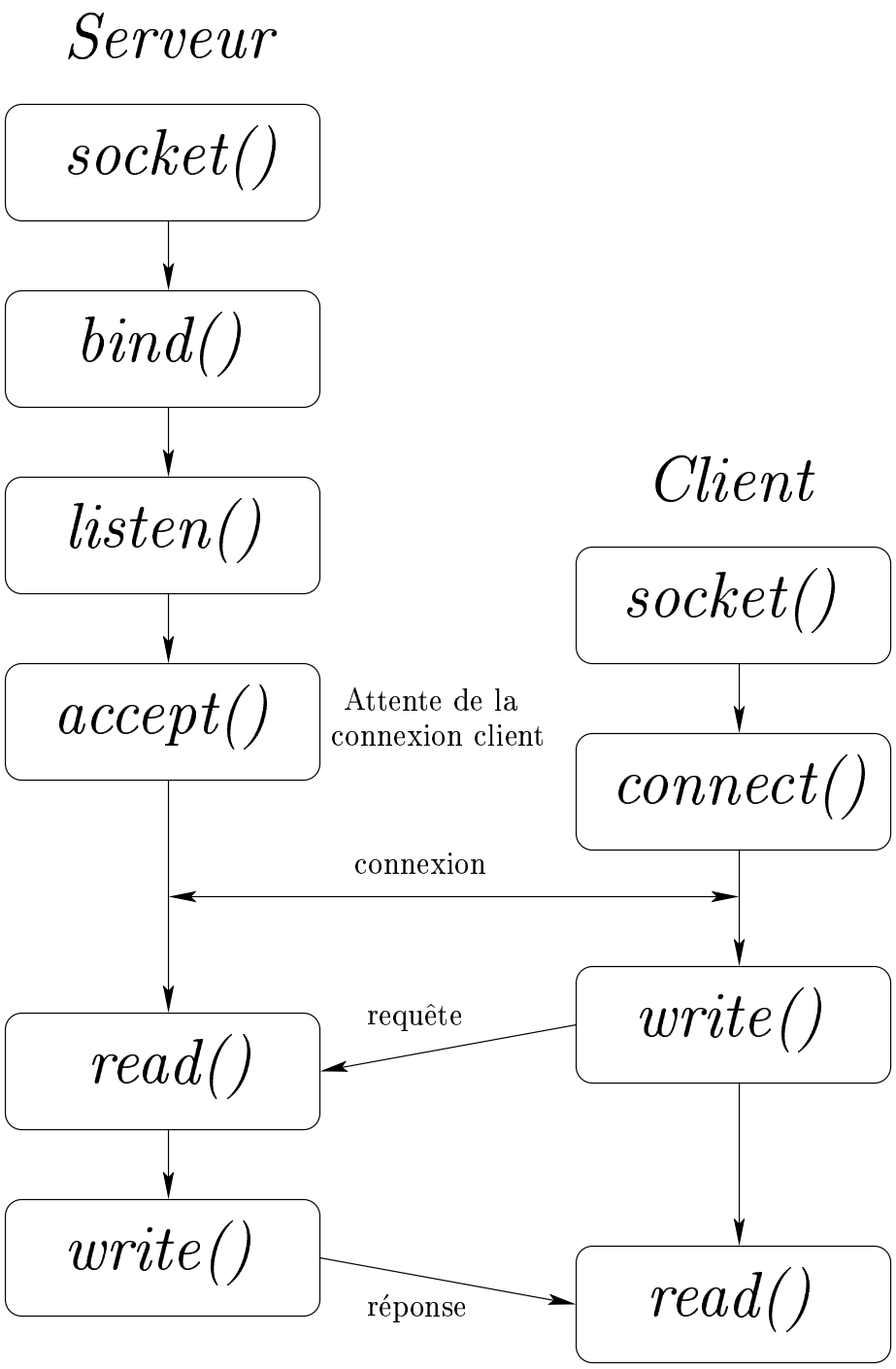


FIG. 7.1 – schéma d'une communication client/serveur en mode connecté

7.4.2 Les fontions de base d'une application client/serveur

Pour mettre en œuvre les objectifs cités précédemment il est obligatoire de créer les fonctions qui permettront de suivre le schéma de communication décrit figure (7.1).

Roger Pissard, Ingénieur des Moyens Robotiques de l'INRIA a développé les fonctions de base décrite ci-dessous. Pour suivre un schéma de communication client/serveur en mode connecté, on utilise la structure SOCK_SERVER définissant ainsi la structure mémoire de l'ensemble des sockets :

```
typedef struct sockServer {
        int s_generic;
        int s_client;
        int port_nb;
        char hostnumber[100];
} SOCK_SERVER;
```

A la création d'une socket serveur, on utilise la fonction *socket()*. Elle renvoie un entier qui correspond à un descripteur de la socket nouvellement créé et qui sera passé en paramètre à la fonction *bind()*. Ce descripteur de socket sera stocké dans le champ `s_generic` de la structure SOCK_SERVER.

De même, la fonction *accept()* qui autorise la connexion en acceptant un appel retourne un identificateur du socket de réponse. Cette identificateur sera stocké dans le champ `s_client` de la structure SOCK_SERVER.

les champs `port_nb` et `hostnumber[100]` permettent respectivement de stocker le numéro de port de la socket et le nom du serveur auquel se connecter. Voici le descriptif des fonctions développées par Roger Pissard :

Fonction *sockClientOpen*

1. Cette procédure permet d'ouvrir la partie client d'une connection point à point TCP/IP. Elle s'organise comme le schéma de la figure (7.1) pour la partie client et utilise les primitives élémentaire *socket(), connect()*.
2. Elle prend en paramètre la structure SOCK_SERVER définie plus haut, le serveur auquel se connecter et le numéro de port de la socket correspondante.
3. Enfin, elle retourne la valeur OK si tout s'est déroulé normalement et ERROR si le numéro de port spécifié du serveur n'est pas valide ou bien si le nom du serveur n'est pas valide.

Fonction *sockServerOpen*

1. Cette procédure permet d'ouvrir la partie serveur d'une connection point à point TCP-/IP. Elle s'organise comme le schéma de la figure (7.1) pour la partie serveur et utilise les primitives élémentaires *socket()*, *bind()*, *listen()* et *accept()*.
2. Elle prend en paramètre la structure SOCK_SERVER et le numéro de port de la socket correspondante.
3. Elle retourne la valeur OK si tout s'est déroulé normalement et ERROR si la socket cliente n'est pas valide.

Fonction *sockClose*

1. Cette procédure permet de fermer une connection TCP/IP aussi bien cliente que serveur. Elle utilise la fonction élémentaire *close()*.
2. Elle prend en paramètre la structure SOCK_SERVER.
3. Elle retourne la valeur -1 si la fermeture échoue.

Fonction *sockSend*

1. Cette procédure permet d'envoyer une chaîne de caractère sur une socket. Elle utilise la primitive *send()* plutot que *write()*.
2. Elle prend en paramètre la structure SOCK_SERVER, le message sous forme d'un *char* et la taille du message transmis.
3. Elle retourne la valeur ERROR si une erreur survient lors de la transmission et la taille du message transmis.

Fonction *sockRec*

1. Cette procédure permet de lire une chaîne de caractère reçue sur une socket, en prenant en compte l'en-tête du message. Elle utilise la primitive *recv()* plutot que *read()*.
2. Elle prend en paramètre la structure SOCK_SERVER, un tableau de type *char* pour y stocker le message reçu et la taille maximum à ne pas dépasser.
3. Elle retourne la taille du message reçu et la valeur ERROR si la taille maximum a ne pas dépasser est inférieur à la taille du message reçu.

Fonction *sockIsAlive*

1. Cette procédure permet de tester si une socket est encore "en vie".
2. Elle prend en paramètre la structure SOCK_SERVER.
3. Elle retourne ERROR si la socket a été fermée, 0 si la socket est encore en vie mais avec aucun octets valide reçu et >0 si la taille des octets reçus est valide.

7.4.3 La communication Bip/Serveur

Après avoir crée les fonctions de base nécessaires à la communication client/serveur, il est impératif de les ajuster pour pouvoir remplir les objectifs fixés.

Le calculateur du robot dispose à chaque pas d'échantillonnage des vecteurs q, $\dot{q}$ et λ. Pour effectuer le calcul de la loi de commande, le serveur doit pouvoir recevoir ces trois vecteurs, et renvoyer le résulat du calcul dans les plus brefs délais.

Soraya Arias, Ingénieur informatique des Moyens Robotiques de l'INRIA a développé les briques de base nécessaire à ce travail. Je me suis appuyé sur ces travaux et ceux de R. Pissard pour développer et adapter les fonctions qui nous permettent de remplir nos objectifs.

Le robot BIP possède un calculateur embarqué de type 68040 fonctionnant avec le bus VME et disposant de l'OS temps-réel VxWorks 5.3.1. Il nous faut donc ouvrir une socket cliente sur ce calculateur. Le développement en langage C se trouve alors contraint par les librairies systèmes de cette cible temps-réel et nécessite quelques ajustements.

Voici la structure et les fonctions que j'ai reprise et développé pour nos besoins :

```
typedef struct _COMBIP_DESC
{
   int dimrow;
   double *q;
   double *qdot;
   double *lambda;
   double *commande;
   int status;
 } COMBIP_DESC;
```

Cette structure définit la taille des vecteurs transmis d'une socket à l'autre, et quatre pointeurs : q, $\dot{q}$, λ et le résultat du calcul de la *commande*. C'est donc par cette structure que communique le robot et le serveur.

Fonction *combipStructCreate*

1. Cette procédure permet de spécifier la taille des vecteurs transmis entre robot et serveur.
2. Elle prend en paramètre la structure COMBIP_DESC et la taille des vecteurs.
3. Elle retourne OK si tout s'est déroulé normalement.

Fonction *convertCombipVect2String*

1. Cette procédure permet de convertir un vecteur de la structure COMBIP_DESC en chaîne de caractère.

2. Elle prend en paramètre la structure COMBIP_DESC et la taille des vecteurs.
3. Elle retourne OK si tout s'est déroulé normalement.

Fonction *convertString2CombipVect*

1. Cette procédure permet de convertir une chaîne de caractère en vecteur de type COMBIP_DESC.
2. Elle prend en paramètre la structure COMBIP_DESC et la taille des vecteurs.
3. Elle retourne OK si tout s'est déroulé normalement.

Fonction *combipSockSendVect*

1. Cette procédure permet d'envoyer un vecteur de type COMBIP_DESC sous forme de chaîne de caractère.
2. Elle prend en paramètre la structure COMBIP_DESC et le nom du vecteur à envoyer.
3. Elle retourne OK si tout s'est déroulé normalement et ERROR sinon.

Fonction *combipSockRecVect*

1. Cette procédure permet de recevoir un vecteur de type COMBIP_DESC sous forme de chaîne de caractère.
2. Elle prend en paramètre la structure COMBIP_DESC et le nom du vecteur à recevoir.
3. Elle retourne OK si tout s'est déroulé normalement et ERROR sinon.

Fonction *combipOpen*

1. Cette procédure ouvre une socket serveur en utilisant naturellement les fonctions développées par R. Pissard *sockServerOpen()* et *sockIsAlive()*.
2. Elle retourne OK si tout s'est déroulé normalement et ERROR sinon.

Fonction *combipOpenClient*

1. Cette procédure ouvre une socket client en utilisant naturellement les fonctions développées par R. Pissard *sockClientOpen()* et *sockIsAlive()*.
2. Elle prend en paramètre la chaîne de caractère désignant le nom du serveur.
3. Elle retourne OK si tout s'est déroulé normalement et ERROR sinon.

Fonction *combipClose*

1. Cette procédure permet de fermer une socket. Elle utilise la fonction *sockClose()*.
2. Elle retourne OK si tout s'est déroulé normalement.

Le détail des fonctions citées jusque là, se trouve en annexe III.

7.5 Mesure des temps de communications

Il nous faut maintenant évaluer les performances temporelles de cette liaison Ethernet. Pour cela, j'ai utilisé les outils de développement de WindRiver Systems [20, 19]. Les fonctions mises à notre disposition permettent de créer un timer qui permettra d'évaluer le temps entre l'envoi des vecteurs q, $\dot{q}$ et λ, la réception de ceux-ci sur le serveur, le calcul de la loi de commande, le renvoi et la reception du résultat.

On choisi de tester les temps de communications pour deux types de loi de commande : la "Computed Torque" et la loi de commande complète (équation 3.6 et 3.7).

Les temps de communication peuvent être variable, c'est pourquoi le calcul de la loi de commande est fait en boucle près de mille fois. On relève, sur ces milles itérations, la valeur minimum et maximum du temps de communication aller et retour incluant le temps de calcul de la loi de commande. On réalise ensuite la moyenne des temps sur les mille itérations.

Les résultats de ces temps sont disponibles tableau (7.1). Deux tests ont été effectués ; le premier test utilise le calculateur du robot BIP comme client et Affligem (station Sun ultra 5, OS Solaris) comme serveur. Le second utilise Lucifer (station Sun ultra 5, OS Solaris) comme client et Affligem comme serveur et sert de référence pour nos mesures.

PROCEDURES	Bip	Lucifer
$\mathtt{T(q)\,u} + \mathtt{C(q)}^{\mathtt{T}}\lambda = \mathtt{M(q)}\left(\mathtt{g} + \mathtt{v}\right)$	7 ms 14,0 ms 9,3 ms	2,3 ms 12,8 ms 3,5 ms
$\mathtt{T(q)\,u} + \mathtt{C(q)}^{\mathtt{T}}\lambda = \mathtt{M(q)}\left(\mathtt{g} + \mathtt{H(q)}^{-1}\left(\mathtt{v} - \mathtt{h(q,q)} + \ddot{\mathtt{Q}}_{\mathtt{d}}\right)\right)$	10,0 ms 30,0 ms 14,0 ms	3,5 ms 27,0 ms 5,9 ms

TAB. 7.1 – Evaluation des temps de communication client/serveur.

On peut observer sur la figure 7.2 les temps de communications et de calcul de la loi de commande complète sur mille itérations. On constate des perturbations réseaux et notamment pour la communication Lucifer/Affligem où de grandes valeurs de temps de communication réseau interviennent. Tout délai de communication trop important (supérieur à la période d'échantillonnage) pourrait avoir des conséquences facheuses sur les manipulations expérimentales. L'ideal serait de ne jamais dépasser 5-6ms pour un aller-retour BIP/Affligem

comprenant les temps de communications aller-retour et le calcul de la loi de commande. Les ingénieurs systèmes des Moyens Informatiques de l'INRIA Rhône-Alpes sont catégoriques, les pointes de temps mesurées ne sont pas dûes à un encombrement réseau mais sembleraient plutôt provenir de daemons réseau (inetd), de la priorité donnée à la tâche d'ouverture de la socket serveur et des délais imposés par le protocole TCP.

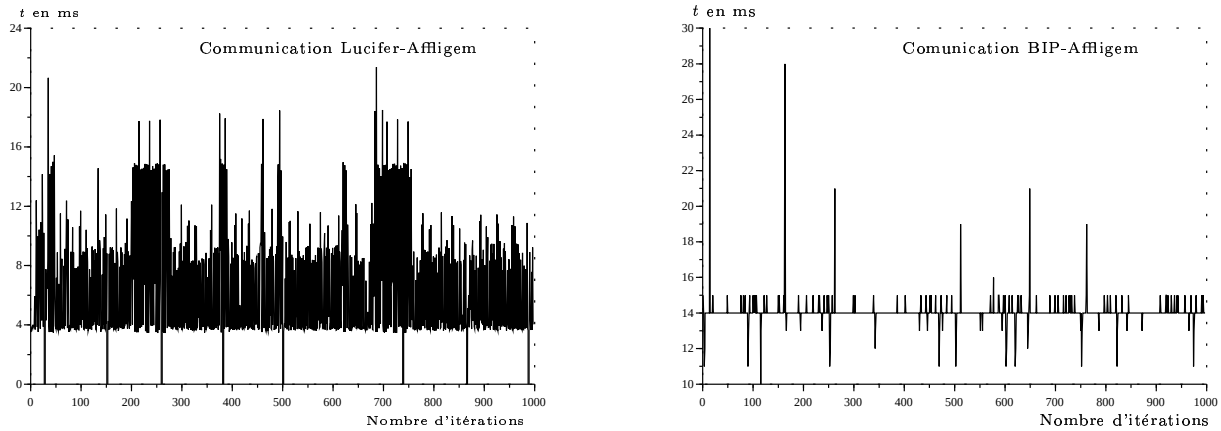


FIG. 7.2 – temps de communications lucifer/Affligem et BIP/Affligem.

Concernant la différence de temps qu'il peut y avoir entre les deux tests, ceci est probablement dû au fait que la carte réseau utilisée sur le robot BIP est à 10 Mbits/s et le débit entre deux machines comme Lucifer et Affligem est de 100Mbits/s. On pourrai alors changer la carte réseau du robot BIP mais il s'avère qu'elle fait partie intégrante du calculateur. On devra donc pour l'instant se contenter de ces performances

Si l'on se réfère au temps de calcul des différentes loi de commande du chapitre 3, on constate que les performances sont honorables et exploitables pour la suite.

Chapitre 8

Manipulations expérimentales

8.1 Objectifs

Ce stage se donne pour objectifs d'implanter la loi de commande par l'approche de fonction de tâche telle qu'elle a été définie au chapitre 3. Néanmoins, il apparaît important, avant de modifier en profondeur l'architecture hardware et software du robot, de refaire les manipulations exécutées par le passé. Il est en effet important de vérifier que l'architecture actuelle fonctionne correctement afin d'éviter toute surprise lors du développement de la nouvelle tâche robot incluant le déport du calcul de la loi de commande.

On répète ainsi toutes les procédures mises en place jusque maintenant pour assurer le suivi de trajectoires élaboré sous l'environnement ORCCAD. Nous présenterons donc dans ce chapitre la procédure d'étalonnage des potentiomètres qui nous permettront de donner la position articulaire absolue du robot en vue de son initialisation, le suivi de quelques trajectoires de référence et l'interprétation des données obtenues.

8.2 Les potentiomètres

On désire obtenir la position absolue des articulations à la mise sous tension de manière à initialiser la position du robot bipède. Pour des raisons de coût, et aussi d'encombrement, une technologie de potentiomètre a été choisie pour cette phase d'initialisation. La commande du robot utilise ensuite les valeurs délivrées par les codeurs incrémentaux dont la pécision est bien supérieure à celle des potentiomètres.

8.2.1 Généralités

Le robot BIP dispose de potentiomètres monotours sans butée. Ils sont fixés sur les articulations et alimentés par une tension de 10 volts. La lecture de la tension du point milieu permet d'avoir l'angle voulu. La tension d'alimentation est imposée par les convertisseurs

analogique/numérique dont l'entrée est comprise entre -10 V et $+10$ V. Nous obtenons cette tension par diviseur potentiométrique (Cf figure 8.1) à partir du $+15$ V (tensions disponibles : $+5$ V et ± 15 V).

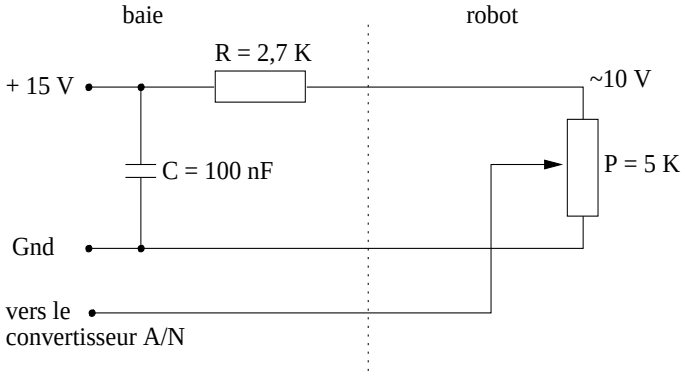


FIG. 8.1 – Schéma de câblage des potentiomètres

Pour des raisons d'encombrement au niveau de l'architecture mécanique le robot est équipé de deux types de potentiomètres. On distingue le modèle MÉGATRON et VISHAY. (Cf figure 8.2 et 8.3)

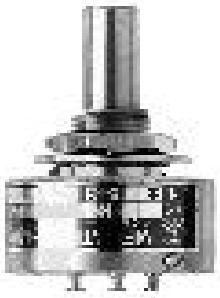


FIG. 8.2 – Vue du potentiomètre MÉGATRON

Le premier modèle (MÉGATRON) est fixé par collage de l'axe du potentiomètre dans un trou situé sur l'axe de chaque articulation et l'immobilisation de son corps se fait par une vis implanté dans la structure du robot. Les valeurs fournies par le constructeur donnent une résistance totale de 5 KΩ, un angle électrique de 340 °, une résolution de 0,01 ° et une linéarité standard indépendante de $\pm 0,25$ %. Le potentiomètre est installé de manière à ce que la course de l'articulation opère dans l'angle électrique. Le facteur d'échelle s'établit à environ 25 mV par degré lorsque la tension aux bornes du potentiomètre est de 10 V.

Le second modèle (VISHAY) a du être spécifié puis fabriqué en sous-traitance. Les valeurs fournies par le constructeur donnent une résistance totale de 5 KΩ et un angle électrique de

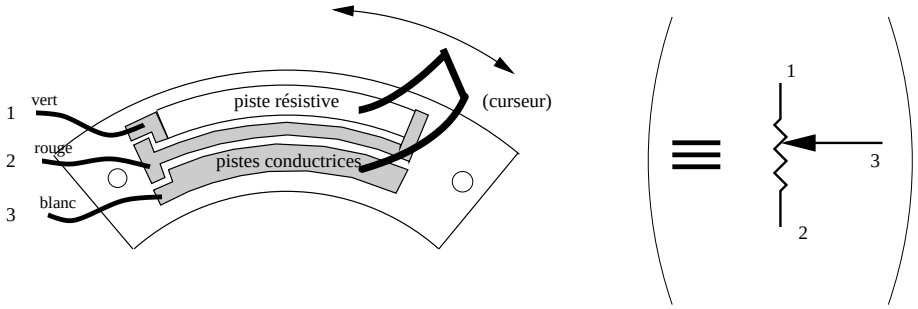


FIG. 8.3 – Potentiomètre Vishay

45 °. Il se compose de deux parties, un curseur (fixé sur la partie dite mobile) et un support pour la piste résistive (fixé sur l'autre partie de l'articulation dite fixe).

8.2.2 Problématique

Les tests réalisés in situ en vue de l'utilisation des potentiomètres pour l'initialisation n'ont pas été satisfaisant. En effet, on observe un cycle d'hystérésis sur la caractéristique des potentiomètres MÉGATRON. Cet hystérésis est d'origine mécanique (jeu entre les axes, efficacité du collage) et électrique (changement de sens de rotation du curseur). Il a été constaté, en outre, que la mesure de la tension aux bornes des potentiomètres est fortement bruitée. Tout ceci dégrade fortement la précision de la mesure angulaire initiale.

Cependant, après vérification des alimentations stabilisées disponibles sur la baie du robot, on s'est aperçu qu'il y avait des fluctuations de tensions. Ces fluctuations étant liées à un emplacements défectueux du module IP contenant les convertisseurs analogique-numérique des potentiomètres, sur la carte VIPC 712. On a donc changé l'emplacement de ce module IP et les problèmes de bruit de mesure ont été fortement réduit.

8.2.3 Etalonnage des potentiomètres du robot BIP

On utilise les potentiomètres dans le but de déterminer la position absolue du robot à sa mise sous tension. Bien sûr, une telle mesure nécessite d'utiliser une position de référence dite "position 0" (Cf figure 8.4). Il nous faut donc étalonner les potentiomètres pour cette position de référence.

Procédure d'étalonnage

La mise en position au zéro mécanique se fait manuellement et s'appuie sur les propriétés géométriques du bipède. Le matériel utilisé à cet effet se compose essentiellement d'un réglet, d'une équerre , d'un niveau et d'un fil à plomb. Les outils utilisés sont, certes, très rudimentaires mais s'avèrent efficaces et suffisants. Le lecteur pourra se rapporter à la référence [5], où

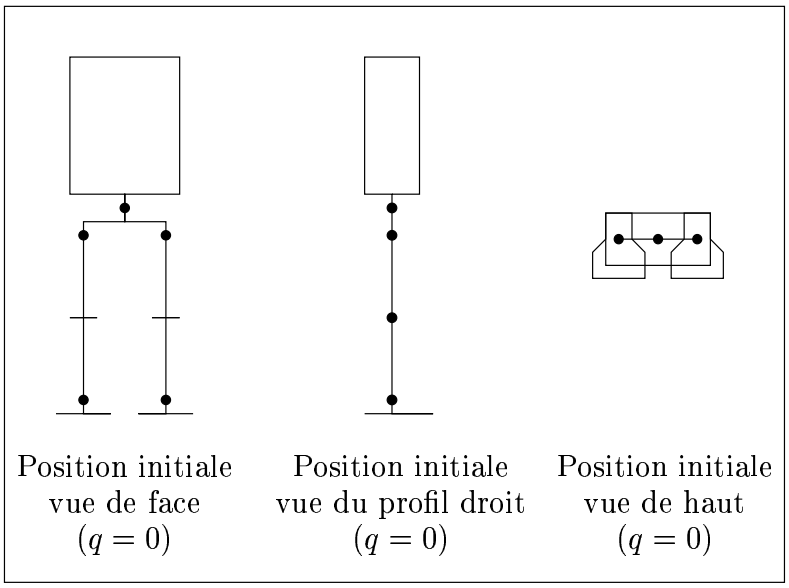


FIG. 8.4 – Position de référence.

une méthodologie rigoureuse est proposée pour la mise en position au zéro mécanique. Une fois établie, cette position de référence est repérée à l'aide de marque blanche (Tipex) sur chaque articulations. Cette procédure a dû être reconduite car elle n'était pas satisfaisante.

L'étalonnage proprement dit se fait à partir des tensions relevées au point milieu des potentiomètres et des angles délivrés par les codeurs incrémentaux. Ces codeurs permettent une mesure relative des angles moteurs. On s'arrange donc pour que le bipède soit au zéro mécanique. De cette manière, à la mise sous tension, les codeurs sont mis à zéro, et permettent donc de connaître après un déplacement, les angles moteurs relatifs par rapport au zéro mécanique.

V. Bozonnet, stagiaire des Moyens Robotiques (MR) en 2001, a développé les fonctions C permettant de relever, à la fois, les tensions des potentiomètres et les angles articulaires mesurés à partir des codeurs (Cf référence [5]).

Pour étalonner les potentiomètres il nous faut relever les valeurs de tensions pour toute la zone angulaire de l'articulation considérée. A partir du zéro mécanique, on effectue plusieurs allers-retours pour chaque articulation et on relève toutes les 10 ms la tension point milieu du potentiomètre considérée et l'angle articulaire correspondant. On obtient alors deux vecteurs de mesure. A l'aide de `Scilab`, on trace la caractéristique de chaque potentiomètre. Les relevés de caractéristiques effectués sur le genou droit (potentiomètre de type MÉGATRON) et l'articulation verticale du tronc (potentiomètre de type VISHAY) sont disponnible figure 8.5.

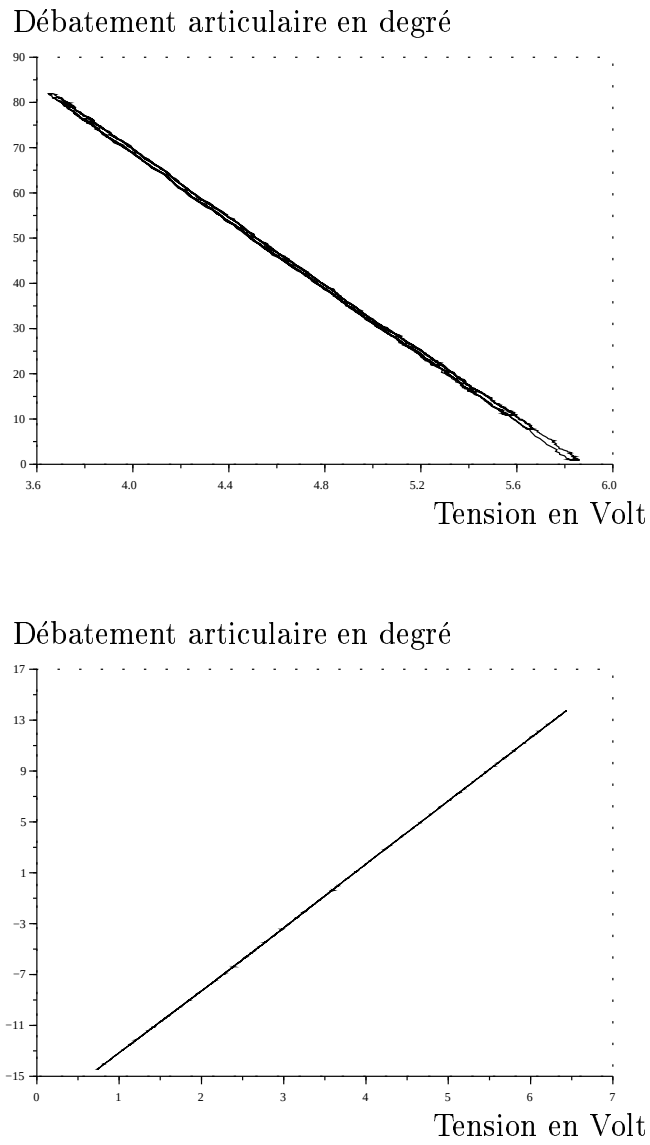


FIG. 8.5 – Exemples de potentiomètres MÉGATRON et VISHAY.

Ces caractéristiques sont linéaire et de la forme : $y_k = ax_k + b$, où y_k est une mesure de l'angle articulaire considéré en radiant et x_k une mesure de la tension correspondante. On déduit alors le coefficient directeur a et l'ordonnée à l'origine b par la méthode des moindres carrés. On résout alors le problème suivant :

$$\min_{a,b} \sum_{k=1}^{N} \left(y_k - ax_k - b\right)^2$$

Ce problème conduit à l'équations matricielle suivante :

$$\begin{pmatrix} \sum_{k=1}^{N} x_k^2 & \sum_{k=1}^{N} x_k \\ \sum_{k=1}^{N} x_k & N \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} \sum_{k=1}^{N} x_k y_k \\ \sum_{k=1}^{N} y_k \end{pmatrix}$$

On résoud cette équation à l'aide de `Scilab`. Ces deux paramètres sont maintenant identifiés, il est alors simple de retrouver l'angle articulaire à partir de la tension lue au point milieu des potentiomètres. On utilise ensuite une procédure `Scilab` nous permettant de générer automatiquement le fichier *BipPotars.h* nécessaire au routine d'initialisation du bipède. Ce fichier contient les coefficients de chacune des articulations du robot. Le fichier de calcul *CalibPotars.sce* et de génération des coefficients a et b est disponible en annexe C.

Exploitation des résultats

Une fois les potentiomètres calibrés, on peut évaluer la qualité de mesure de chacun, en utilisant les informations des codeurs incrémentaux. On rappellle que ces codeurs ont une précision de 4096 pas par tour. Un tour moteur représente, selon les articulations, un déplacement angulaire de 2 à 4 °. On peut alors tracer l'erreur de mesure $(q_i - ax_i - b)$ du potentiomètre i en fonction de l'angle articulaire q_i délivré par le codeur de l'articulation considérée. On notera que x_i est la tension du point milieu délivrée par le potentiomètre i. Le relevé de l'erreur de mesure de chaque potentiomètre est disponible en annexe C figure D.1, D.2, D.3. On peut remarquer les cycles d'hystéresis sur les potentiomètres de type MÉGATRON.

8.3 La phase d'initialisation

L'objectif de l'initialisation, outre le démarrage du système et des différents modules, c'est de déterminer la position absolue du robot à sa mise sous tension. Deux solutions sont possibles, l'utilisation des potentiomètres pour une initialisation automatique ou le positionnement du robot dans sa position dite "0" pour une initialisation manuelle.

Il existe pour l'initialisation automatique du système, une tâche-robot (TR) Orccad nommée *bipInit* (Cf référence [2]). On teste puis on valide cette tâche-robot. Cependant, nous avons préféré utilisé une simple procédure C nommée *bipInitAuto* développée par V.Bozonnet [5]. Celle-ci a été compilée sur station Sun pour la cible temps réelle VxWorks et chargée sur le processeur de la carte.

A l'exécution de cette procédure C, on obtient pour une initialisation automatique les informations suivantes :

```
VIO Console Create succeeded!
Std I/O set here !
device t501 ok
IP_ADC POTENTIOMETRE ok
IP_UNIDIG ok
IP_ADC CURRENT non pris en compte
IP_DACSU ok
IP_QUADRATURE 1 ok
IP_QUADRATURE 2 ok
IP_QUADRATURE 3 ok
IP_QUADRATURE 4 ok
IP_ADC SENSOR ok
Open Bipede driver... ok
Init automatique (o/n) ?oo

Joints Right = 0.03138 0.0490249 0.10979 -0.0214935
Joints Left = -0.0050617 -0.0195781 0.125823 0.00545531
Joints Pelvis = -0.00024393 0.00397796 -0.00122236
                -0.00069012 -0.000490698
Joints trunc = -0.000297708 -0.011723

Foot Right = -4.43596 1.69091 -3.81139
Foot Left = -8.65256 3.51282 -3.36353
```

FIG. 8.6 – Résultat de la procédure d'initialisation *bipInitAuto*

On trouve bien entendu l'initialisation hardware de tous les modules IP. On peut observer le relevé des positions articulaires établi à partir des tensions point milieu de chaque potentiomètre et la valeur des capteurs de pression. A noter que les positions articulaires sont déterminées à partir du fichier de calibrage des potentiomètres *bipPotars.h* établi auparavant.

8.4 Poursuite de trajectoire

Le suivi de trajectoire se fait normalement au travers de la procédure robot *ProcMove* développée par C. Azevedo [2]. Cette procédure met en jeu deux tâches-robot *bipInit* et *bipMove* qui s'enchaînent. J'ai repris et modifié cette procédure. Elle se nomme à présent *ProcTest* et met en jeu la seule TR *bipTest* (Cf figure 8.8). L'exécution de cette tâche se fait en cinq étapes (Cf figure 8.7) :

1. le robot est suspendu et atteint la posture de départ du mouvement.
2. le robot reste immobile pendant un temps fixé à l'avance le temps d'être posé sur le sol.
3. le robot effectue le mouvement désiré.
4. le robot reste immobile pendant un temps fixé à l'avance le temps d'être à nouveau suspendu.
5. le robot revient dans sa position initiale.

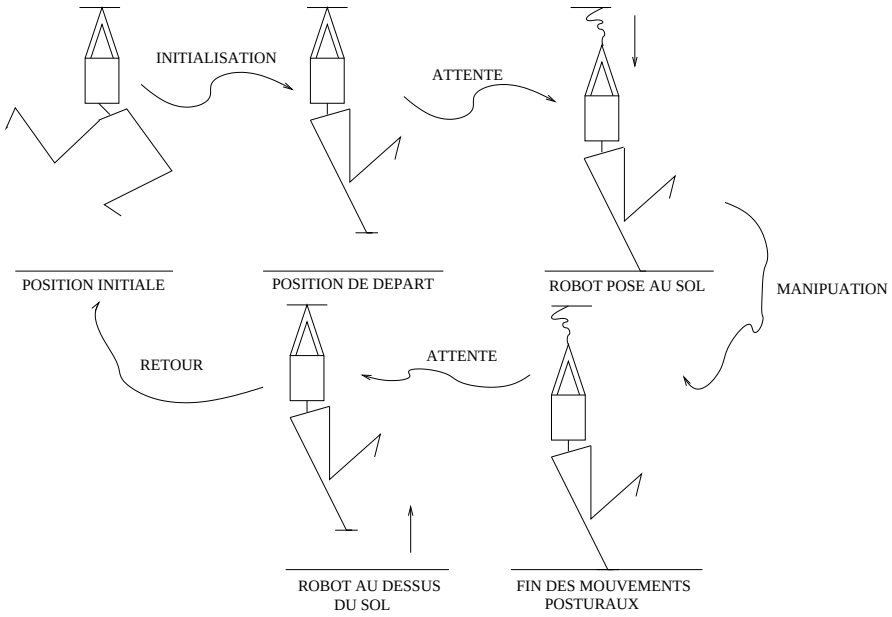


FIG. 8.7 – Les Différentes étapes de la TR *bipMove*

Il est important de noter que toutes les trajectoires actuellement jouées se font en simple support droit. Aucune phase de simple support gauche ou de double support n'est mis en jeu dans les expérimentations menées.

On note trois états fondamentaux sur les cinq étapes de la TR *bipTest* :

1. aucun pied au sol (robot suspendu),
2. robot en cours de dépose ou de levage (transitions),

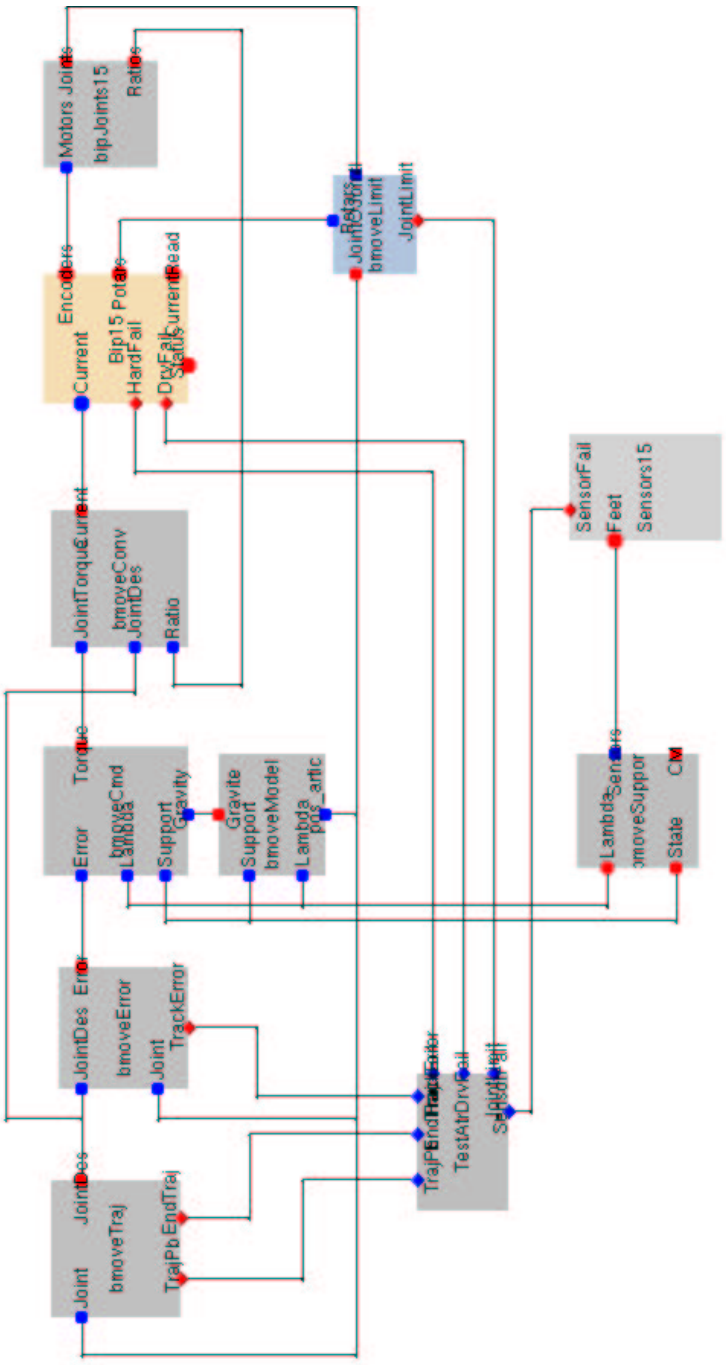


FIG. 8.8 – Tâche robot Orccad *bipTest*

3. simple support droit.

Ces trois états ont une grande influence sur la loi de commande à mettre en œuvre. On se rend bien compte que l'estimation de la gravité ou la valeur des gains proportionnel-dérivé sont différents selon que le robot est suspendu, en transition ou au sol. Il est alors primordial pour commander au mieux le robot de détecter ces différents états.

8.4.1 La détection de support

La détection de support s'appuie sur les informations fournies par les capteurs d'éffort par l'intermédiaire de la ressource physique *Sensors15*.
Les phases de support considérées sont les trois décrites précédemment. Les transitions correspondent aux cas ou la somme des force de pression est inférieure au poids total du robot, c'est à dire lorsque le robot n'est que partiellement posé sur sol (en cours de pose ou de levage).

Le robot pèse 105 kg, la force exercée lorsqu'il est posé au sol est donc d'environ mille Newtons. Le module algorithmique *bmoveSupport* est responsable de la détection de support. Il teste la somme cumulée des forces de pression F_T sur le pied droit. On distingue trois cas :

1. Si $0 < F_T < 200$ (en Newton), alors le robot est suspendu.
2. Si $200 < F_T < 900$, alors le robot est en transition.
3. Si $F_T > 900$, alors le robot est posé au sol.

Ce module utilise aussi une fonction de détection d'impacts F_d pour chaque pied. Cette fonction correspond à la somme cumulée de la force de pression F_T sur un pied donné, corrigée du bruit moyen.

$$F_d = F_T - \frac{\Delta\mu}{2,0}$$

Lorsque le pied est au dessus du sol, les informations des capteurs sont très bruitées, en revanche la somme cumulée de la force de pression est une droite peu perturbée. On observe cette droite au cours du temps, si l'on détecte une modification de la pente, et si cette modification est significative on conclut à un impact.

Cependant, la detection d'impact ne nous intéresse pas ici. On rappelle que seul le simple support droit est considéré.

En cours de pose ou de levage, on fait l'approximation que le poids du robot se déplace de façon linéaire sur le pied droit. On défini alors la valeur λ_0 correspondant au rapport entre le poids supporté par le pied droit et le poids total du robot et $\lambda_1 = 1 - \lambda_0$.

Le module algorithmique *bmoveSupport* génère ainsi les informations relatives à la phase de support et aux valeurs λ_0 et λ_1 correspondantes (Cf figure 8.8). Le détail de ce module est décrit plus précisément dans [2].

8.4.2 Contrôle-commande du robot

Le robot BIP dispose d'une puissance de calcul limitée. Afin de maintenir une période d'échantillonnage suffisamment faible (10 ms), on utilise une loi de commande de type proportionnel-dérivé associé à une compensation de gravité et une compensation de frottement. On calcule dans un premier temps la partie PD+Gravité pour déterminer les couples articulaires Γ :

$$\Gamma = K_p(q_d - q) + K_v(\dot{q}_d - \dot{q}) + \hat{G}(q) \tag{8.1}$$

avec K_p et K_v les gains proportionnel et dérivé, q_d la trajectoire désirée que l'on poursuit et $\hat{G}$ une estimation du vecteur gravité.

On remarque que la loi de commande que l'on utilise sur la plateforme expérimentale diffère fortement de celle utilisée dans le simulateur. Elle ne prend pas en compte les contraintes de non-pénétration et de non-glissement. C'est pourquoi on utilise un jeu de gain proportionnel-dérivé adapté à la phase de support considérée. On distingue ainsi trois jeux de gains pour le robot suspendu, en support pied droit et en transition. On utilise λ_0 et λ_1 pour passer du jeu de gain correspondant au robot suspendu au jeu de gain correspondant au robot en simple support pied droit.

Le module algorithmique *bmoveCmd* est responsable du calcul de la loi de commande. Il prend en entrée les informations relative à l'état du robot, le vecteur (λ_0, λ_1) issu du module *bmoveSupport*, le vecteur gravité et l'erreur $q_d - q$ calculée dans le module *bmoveError*. Il renvoie en sortie le vecteur des consignes de couples à appliquer sur chaque articulation (Cf figure 8.8).

Le vecteur Γ est ensuite transformé en vecteur de tensions U' grâce aux modèles de transmissions moteurs [12] implantés dans le module *bmoveConv*. Puis, on introduit pour finir une compensation de frottements (module *bmoveConv*) définie de la manière suivante :

$$U = U' + F.sign(\dot{\theta}) \tag{8.2}$$

où F est le vecteur des constantes de frottements et $\dot{\theta}$ le vecteur des vitesses moteurs.

8.4.3 Estimation de la gravité

Comme on l'a vu précédemment, la loi de commande dépend de la phase de support considérée. Il en va de même pour l'estimation de la gravité. On dispose de deux procédures pour le calcul de la gravité :

1. *gravite_pendu.c*
2. *gravite.c*

On utilise ainsi deux procédures suivant l'état du système pour estimer le vecteur gravité $\hat{G}$ du robot.

- $\hat{G}$=*gravite*, dans le cas du simple support droit.

- $\hat{G}$=*gravite_pendu*, dans le cas du robot suspendu.

Pour les phases de transition, on utilise l'approximation linéaire définie auparavant et l'on obtient :

$$\hat{G} = \lambda_0 * G0 + \lambda_1 * G1$$

Le module algorithmique *bmoveModel* assure le calcul de la gravité en fonction de l'état du robot et du vecteur (λ_0, λ_1) (Cf figure 8.8).

8.4.4 La génération de trajectoire

La génération de trajectoire se fait au travers d'un fichier pour lequel le format est prédéfini à l'avance. Ce fichier contient des points de passages de la trajectoire à suivre. Pour la phase de mise en position initiale et de retour en position initiale la trajectoire est interpolée par polynôme de degré 5 pour trouver les positions à chaque instant. Dans ce fichier on distingue les champs suivant :

- **Format** : déclare le format de fichier.
- **NbArts** : déclare le nombre d'articulations du robot considéré.
- **NbPts** : déclare le nombre de points de passages définis dans le fichier.
- **Cycle** : déclare le nombre de fois que la trajectoire sera rejouée.
- **Rebouclage** : spécifie sur quel point de passage on reboucle la trajectoire pour un nouveau cycle lorsque le point de passage final a été atteint.
- **Temps** : déclare le temps alloué au robot pour atteindre le premier point de passage à partir de sa position initiale (phase de mise en position initiale et de retour en en position initiale).

On a ensuite une liste de valeurs rangées dans l'ordre de numérotation des articulations et regroupées par 3 : position articulaire, vitesse articulaire, accélération articulaire désirées. Vitesse et accélération ne sont pas utilisées dans la version actuellement implémentée.

Le module algorithmique *bmoveTraj* est conçu pour suivre les cinq phases de la figure 8.7. Il permet aussi de charger les fichiers définies ci-dessus.

8.4.5 Les trajectoires de référence

Comme nous l'avons dit au début de cette section, le robot n'effectue pour l'instant que des suivis de trajectoire en simple support droit. Les manipulations se font dans un premier

temps, robot suspendu, puis une fois validées on les rejoue, robot au sol.
Les postures disponibles sont les suivantes :

p0.traj	posture initiale
p1.traj	posture pied légèrement sur le coté
p2.traj	posture pied sur le coté
p5.traj	posture pied en avant
p6.traj	posture flamand rose

TAB. 8.1 – Postures disponibles

Ces cinq postures sont fixes et donc sans mouvement. Les points de passage contenu dans le fichier de génération sont donc constant. Néanmoins, on dispose de mouvements posturaux en reliant les différentes postures ci-dessus. Les mouvements posturaux disponibles sont les suivants :

p01.traj	petit balayage horizontal
p02.traj	grand balayage horizontal
p65.traj	

TAB. 8.2 – Mouvements posturaux disponibles

Un aperçu des postures disponibles se trouve en annexe D.

8.5 Résultats

Le contrôleur de robot Orccad nous permet de faire une acquisition de données durant toute la phase de manipulation. Il suffit juste de choisir les grandeurs à relever. Les grandeurs intéressantes sont q, l'erreur de suivi $(q - q_d)$, les consignes de couple Γ, la phase de support à l'instant t, l'évolution du vecteur (λ_0, λ_1) et les forces de pression sur chaque pied. L'extraction de ces grandeurs se fait à l'aide d'un shell script nommé *datfilter*, celui-ci est disponible en annexe D.

8.5.1 Robot suspendu

On rejoue dans cette section toutes les trajectoires citées dans le paragraphe précédent, mais le robot est suspendu durant toute la phase de manipulation. Intéressons-nous plus particulièrement au suivi de trajectoire *p0.traj* (Cf annexe D figure E.1). La figure 8.9 donne l'évolution, en radian, du vecteur articulaire q, l'erreur de suivi $(q - q_d)$ et les consignes en

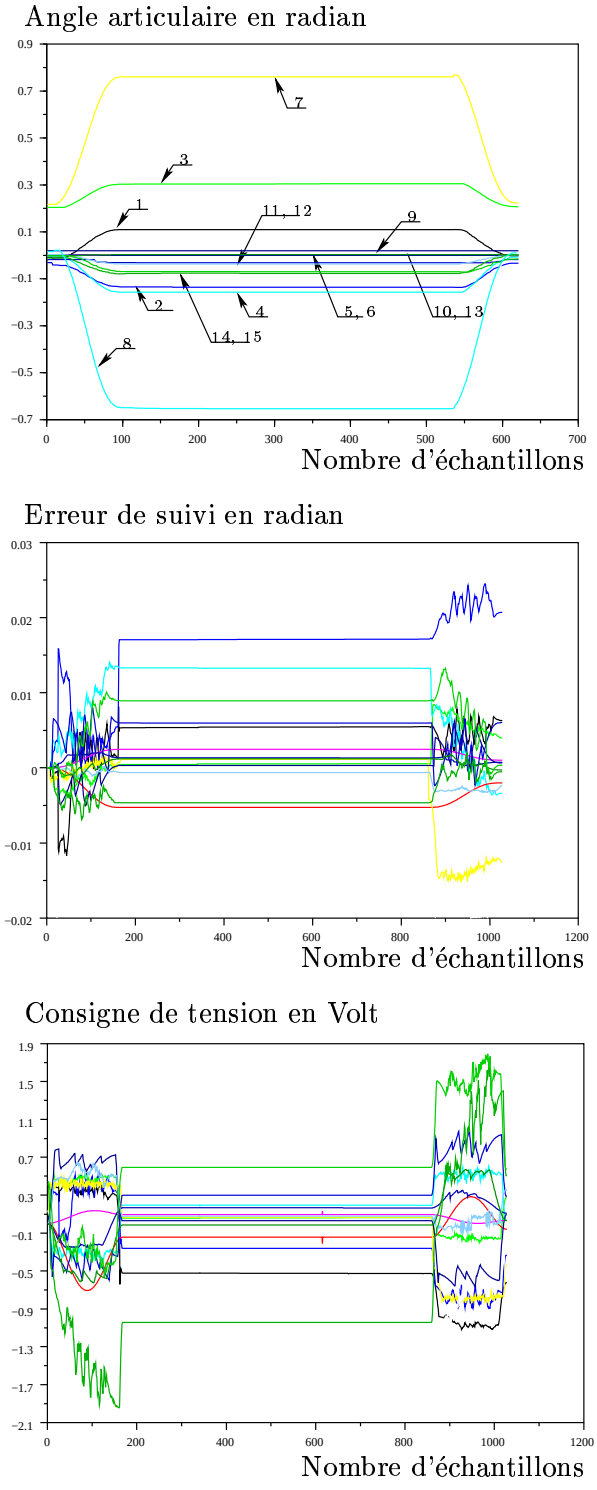


FIG. 8.9 – Évolution du vecteur q, de l'erreur de suivi et des consignes de tension.

tension U envoyées aux variateurs des moteurs Le lecteur pourra se reporter à l'annexe D où la figure E.6 montre l'évolution du vecteur articulaire q pour les autres trajectoires.

Pour tous les tests, robot suspendu, on remarque un mauvais retour en position initiale. En effet, on constate sur le robot un écart important (de l'ordre du cm) au niveau des pieds. Les articulations du genou gauche et de la hanche sagitale gauche ne reviennent pas parfaitement en position initiale. On a d'abord modifié le gain proportionnel de ces deux articulations mais sans succès. On a ensuite réétalonné les potentiomètres avec une grande précision mais le problème ne semblait pas venir de là. Ce problème ne fut pas identifié tout de suite, il a fallu vérifier et parfois refaire le code relatif à la TR *bipMove*. La résolution de ce problème est abordée plus loin dans ce chapitre.

On continue ensuite les manipulations mais on suit maintenant la procédure définie en début de la section 8.4.

8.5.2 Robot au sol

Les manipulations se font maintenant au sol. Il est important ici de bien vérifier la détection de support et les étapes de transitions lors de la pose et du levage du robot. On dispose pour cela du vecteur (λ_0, λ_1) et de la variable *State* laquelle nous informe sur l'état du robot.

On s'intéresse ici uniquement au suivi de la trajectoire *p0.traj*, le principe restant le même pour les autres trajectoires.

On peut observer sur la figure 8.10 les états par lesquel passe le robot. On se rend compte alors qu'après la phase de transition l'état du robot oscille entre 1 et 4, soit entre l'état *posé au sol* et *transition*. Ceci est dû au fait que le robot ne tient pas bien en équilibre et qu'il bouge sur son pied d'appui (droit).

Les capteurs d'effort placés sur les pieds du bipède sont de type piézo-résistif et travaillent en traction-compression. Cela se traduit par une valeur négative en compression et positive en traction. Le lecteur pourra se reporter en annexe E ou aux référence [16, 3] pour en savoir plus sur les capteurs de force.
Les forces de contact relevées figure 8.10 montre que le robot a tendance à pencher vers l'avant gauche. En effet, le capteur arrière délivre une valeur positive ce qui signifie qu'il est en traction symbolisant bien le fait qu'il se penche en avant. De même, on observe une valeur plus grande sur le capteur avant gauche comparé au capteur avant droit. Le robot a donc tendance à pencher vers la gauche.

Si l'on observe pour finir la courbe 3 de la figure 8.10 qui décrit l'évolution du vecteur (λ_0, λ_1) durant la phase de transition, on voit bien comment s'applique le transfert linéaire du poid du robot de la phase robot suspendu au sol.

On s'aperçoit que le robot ne tient pas sur son pied d'appui droit. Le bipède penche vers l'avant gauche et l'on désire s'expliquer cela. On calcule pour cela la position du centre de pression sur le pied droit à tout instant. Ces calculs sont disponibles en annexe E. La figure 8.11 rend compte de cette évolution. On trace aussi la projection du centre de masse dans

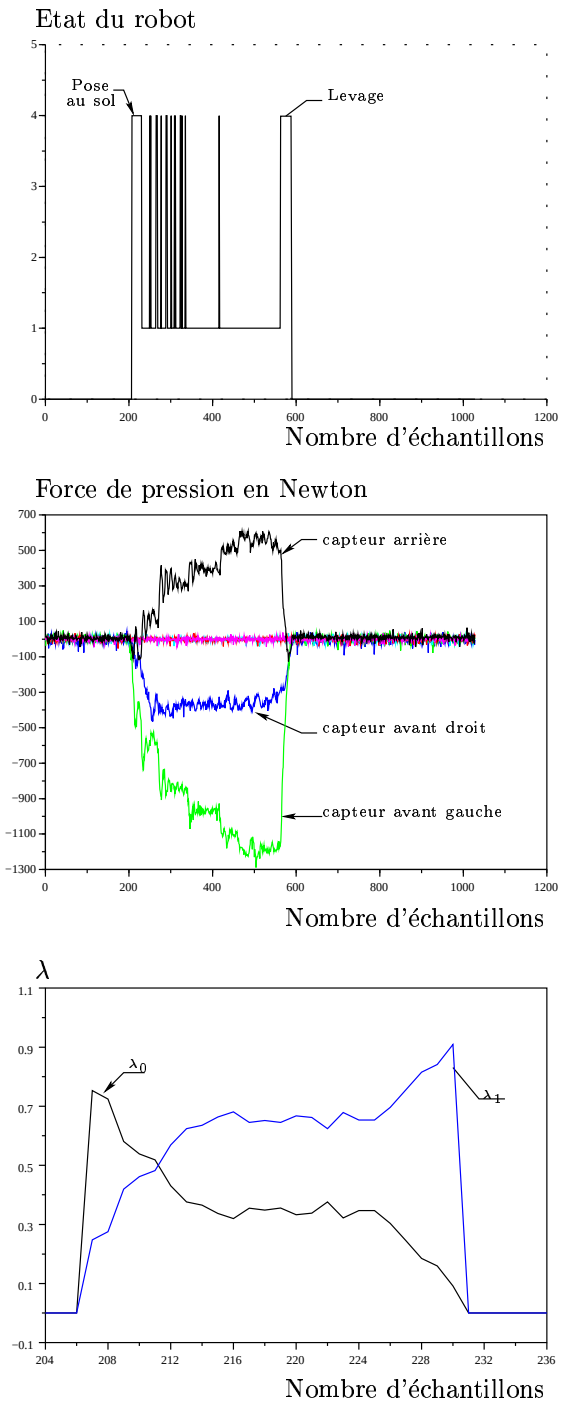


FIG. 8.10 – Évolution de l'état du système, du vecteur (λ_0, λ_1) et des forces de pression.

le plan (O, x, y) du pied droit. On constate que le centre de masse et de pression reste à l'intérieur du polygone de sustentation mais ils dérivent lentement vers l'avant gauche. Ainsi,

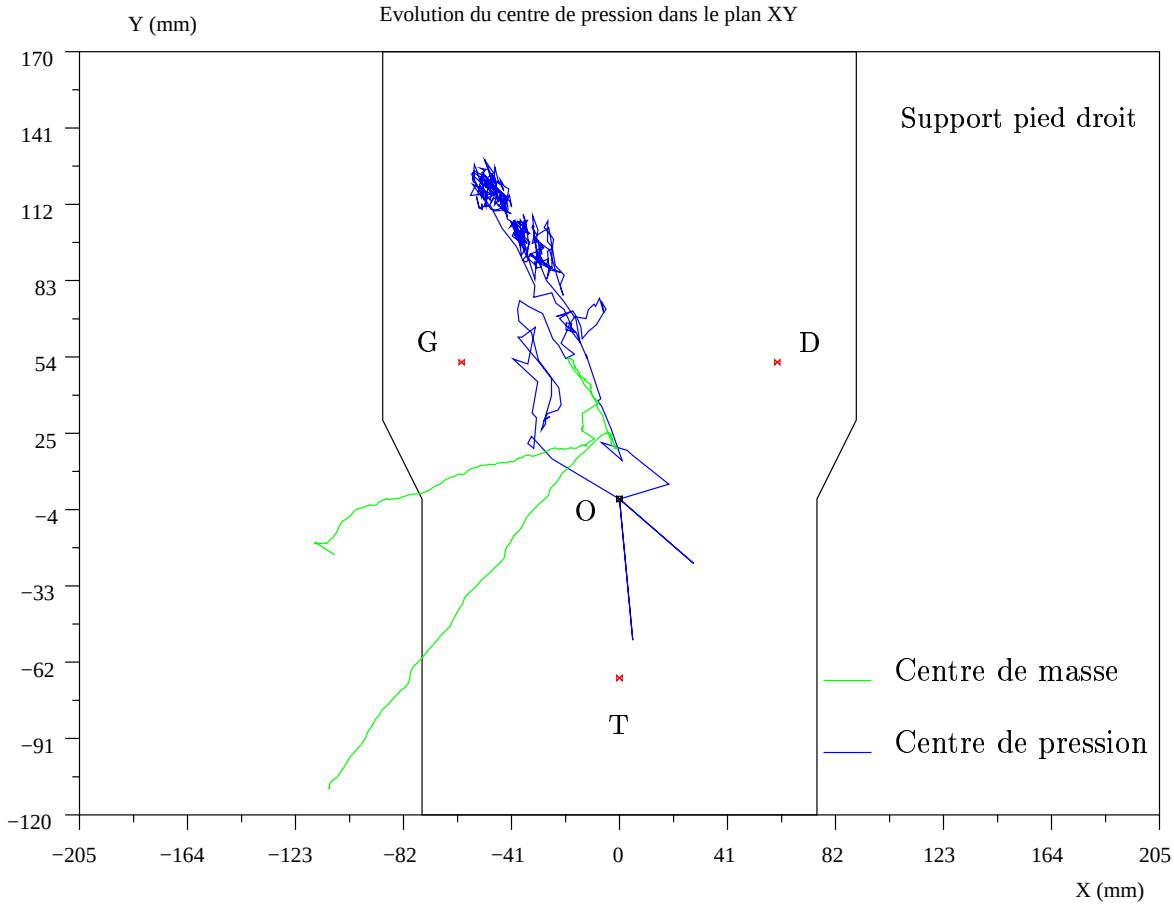


FIG. 8.11 – Évolution du centre de pression et du centre de masse dans le plan (O, x, y).

les capteurs d'effort placés sur les pieds du robot délivrent des informations cohérentes compte tenu des résultats énoncés.

Le robot ne tient cependant pas correctement sur le pied droit. Pour résoudre ce problème on décide de modifier la valeur des gains proportionnel-dérivé lorsque le robot est sol.

8.6 Réglages des gains proportionnel-dérivé

Nous avons vu dans la section 8.4.2 que l'on dispose d'un jeu de gain pour les états : robot suspendu, en transition et au sol. On augmente ainsi le gain proportionnel des moteurs

de la cheville droite. On espère de cette façon rendre plus rigide ces articulations. Mais on se rend compte que la solution n'est pas là. On peut observer sur la figure 8.12 l'effet d'une augmentation du gain sur les deux moteurs de la cheville droite. On constate que le système devient instable.

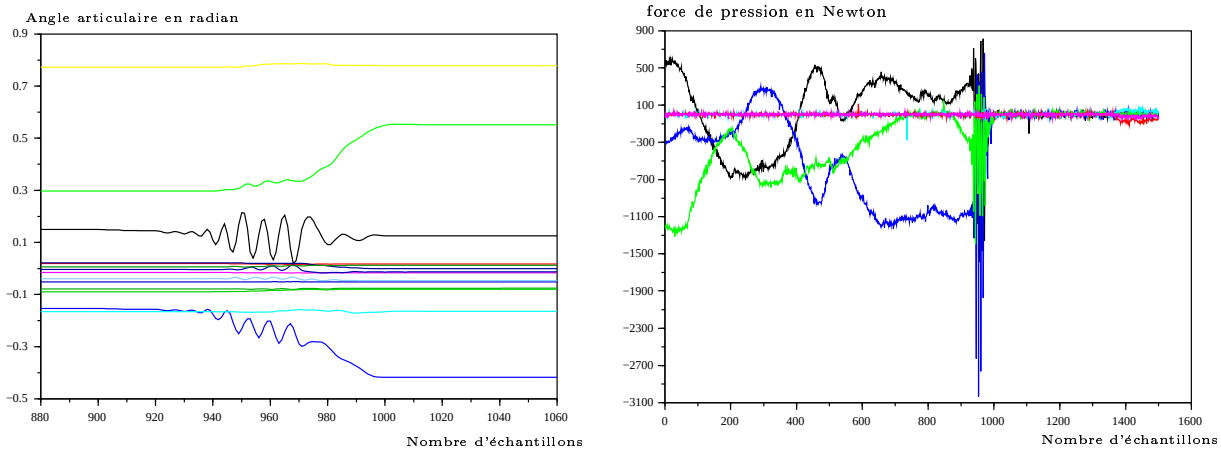


FIG. 8.12 – Instabilité des articulations frontale et sagittale.

Il ne nous reste plus qu'à vérifier maintenant les coefficients de frottement sec des moteurs.

8.7 Les frottements secs

Comme nous l'avons vue à la section 8.4.2, la TR *bipTest* applique une compensation de frottement. Nous réévaluons ici les coefficients de frottement sec pour chaque moteur et pour les deux sens de fonctionnement. Pour réaliser ceci, j'ai créé une application C qui permet de relever le coefficient de frottement sec du moteur spécifié. Cette procédure envoie une consigne de tension au variateur du moteur considéré et s'incrémente avec un pas de 30 mV jusqu'à ce que le moteur tourne d'un nombre de pas désirés. On obtient finalement la valeur de la tension à injecter à chaque moteur pour que ceux-ci compensent les frottements secs. Le détail de cette procédure nommée *frottement.c* est disponible en annexe F. De la même manière que la procédure bipInitAuto, elle est compilé pour VxWorks sur station Sun puis charger sur le processeur de la carte embarquée.
Les mesures de ces coefficients sont aussi disponibles en annexe F.

Lors de l'évaluation de ces frottements secs, un fait marquant est apparu. Les moteurs interne et externe de la cheville droite et ceux du genou et de la hanche sagittale gauche ont un problème. Lorsque l'on exécute la procédure C qui mesure le coefficient frottement sec de ces moteurs dans le sens négatif, ils partent dans le sens positifs et qui plus est pour une

valeur de consigne égale à 30 mV ! Et ce n'est que pour une forte valeur de tension que ces moteurs fonctionnent correctement.

Ce problème semble directement lié au problème de retour en position initiale que l'on observait au début des manipulations sur les articulations de la cheville droite et celles du genou et de la hanche sagitale gauche. Il en va de même pour l'équilibre du robot lorsqu'on le pose au sol. Les moteurs de la cheville droite montrent un faible couple articulaire.

On montre ainsi que la plateforme expérimentale est opérationnelle si ce n'est l'utilisation de ces quatres moteurs. Il convient de repérer l'anomalie et d'agir en conséquence. La prochaine étape consiste après ceci, à implanter la loi de commande développée dans le simulateur en utilisant les sockets développées au chapitre 7.

Perspectives & Conclusions

Ce stage se fixait pour objectifs d'implanter sur la plateforme expérimentale la loi de commande établie par P.B Wieber dans [18]. Elle devait permettre au robot bipède d'évoluer de façon stable et dynamique sur sol plat.

Cette nouvelle loi de commande comporte 21 degrés de liberté contre 15 ddl pour celle actuellement utilisée sur le bipède. Ces six variables supplémentaires décrivent la position et l'orientation du robot dans l'espace. Or, le robot ne dispose pas de capteur extéroceptifs lui permettant de mesurer sa position et son orientation dans l'espace. De plus, l'architecture matérielle et logicielle du calculateur du robot ne permet pas de faire face à la quantité de calcul générée par cette loi de commande.

La première partie de ce stage fut théorique et s'appuya sur le simulateur du robot. On se fixait pour objectifs d'optimiser les temps de calcul de la loi de commande et de reconstruire la position et l'orientation du robot dans l'espace par l'intermédiaire d'un observateur.

Pour ce faire nous avons d'abord développé un nouveau changement de variable pour notre fonction de tâche, permettant ainsi d'améliorer la stabilité du Bipède et de découpler les variables articulaires des variables de position et d'orientation, ce qui a pour effet de simplifier les calculs générés par la loi de commande et donc de diminuer les temps de calculs.

L'observateur de q_2 (variables de position et d'orientation du bipède dans l'espace) fut testé puis validé sur le simulateur. Il permettra d'utiliser la loi de commande telle qu'elle a été établie, avec 21 degrés de liberté.

Une fois ce travail d'adaptation terminé, nous en sommes venus à la plateforme expérimentale. Après l'étude menée auprès des différents acteurs du projet BIP, nous avons décidé de déporter le calcul de la loi de commande sur un serveur distant en mode connecté (TCP/IP). Les tests de communication entre calculateur et station Sun s'avèrent encourageants. Mais l'objectif final n'a pas été atteint, du moins, pas complètement.

Durant les phases d'expérimentations, nous avons constaté plusieurs anomalies, notamment au niveau de l'alimentation des potentiomètres et surtout au niveau des moteurs de la cheville droite, du genou gauche et de la hanche droite. Ces problèmes moteurs provoquent un mauvais retour en position initiale du bipède et ne permettent pas au robot de se maintenir en équilibre latéral. Tant que ces moteurs ne seront pas opérationnels, aucune manipulation au sol ne pourra être entreprise.

Ceci-dit, ces tests expérimentaux, menés avec méthode et minutie, ont permis de valider

le bon fonctionnement général de la plateforme, tant d'un point de vue électromécanique que d'un point de vue informatique. Nous avons aussi développé en chemin les applications client/serveur nécessaires à la mise en place de la nouvelle loi de commande.

La prochaine étape consiste à adapter l'application client/serveur développée entre le calculateur du robot BIP et le calculateur déporté sous l'environnement de programmation Orccad, puis d'effectuer les expérimentations finales.

Sur un plan plus personnel, ce stage d'une durée de six mois, au sein de l'équipe du projet BIP m'a permis de participer à un projet de grande envergure et d'affirmer mon goût pour l'ingénierie dans la recherche. Un tel projet exige la collaboration de plusieurs équipes spécialisées dans des domaines aussi différents que la mécanique, l'électronique, l'informatique temps réels et l'automatique. L'intéraction de ces différentes disciplines constitue une expérience enrichissante, à la fois sur un plan scientifique et humain avec la nécessité d'un travail en équipe. Je retiendrais aussi l'impressionnante quantité d'informations existantes et l'inertie qu'un tel projet peut provoquer. Ce projet a démaré voilà maintenant 8 ans. Les prérequis indispensable pour commencer à travailler sont important, traite de domaine différent pour une compléxité variable. Je n'oublie pas non plus les nombreuses procédures de mise en route de la plateforme expérimentale.

Troisième partie

Annexes

Annexe A

Développement du simulateur

fichier *Commande.maple*

```
#####################################################################
#calcul la matrice de passage du repere ref_k vers le repere k#
#####################################################################
inverse_matrice_passage := proc(k) option remember;

        RETURN(map(simplify, inverse(matrice_passage(k)))):
end:

#############################################
#calcule la matrice de passage du repere k #
#vers le repere 6 (lie au pelvis)          #
#############################################
matrice_repere_j := proc(k, j) option remember;

        if k <> j
        then
           RETURN(evalm(matrice_repere_j(eval(cat(ref_,k)), j) &* matrice_passage(k))):
        else
           RETURN(evalm(matrix(array(1..4, 1..4, identity)))):
        fi:
end:

#############################################
# Calcule les coordonnees du centre de masse #
# du solide (k) dans le repere 6.           #
#############################################
coord_cdm_6 := proc(k) option remember;
local matrice, trans, rot;

        matrice := matrice_repere_j(k, 6):
        trans := subvector(matrice, 1..3, 4):
        rot := submatrix(matrice, 1..3, 1..3):
        RETURN(evalm(trans + rot &* eval(cat(G_,k)))):
end:

#############################################
# Calcule les coordonnees du centre de masse #
# du robot dans le repere 6.                #
#############################################
```

```
coord_cdm_robot_6 := proc() option remember;
local cdm, masse, k;

        print(`CENTRE DE MASSE (repere 6)`);
        masse := masse_robot():
        cdm := vector(3, 0):
        for k from 1 to NSOL
        do
                if eval(cat(m_,k)) > 0 then
                    cdm := evalm(cdm + eval(cat(m_,k))/masse * coord_cdm_6(k)):
                fi:
        od:
        RETURN(evalm(cdm)):
end:

############################################################
#calcule la matrice de passage du repere 23 (lie au tronc)#
#vers le repere 12 (lie au pied droit)                     #
############################################################
matrice_12_23 := proc() option remember;
local matrice_12_6, m_6_23,  i;

        matrice_12_6 := inverse_matrice_passage(7):
        for i from 8 to 12
        do
            matrice_12_6 := evalm( inverse_matrice_passage(i) &* matrice_12_6 ):
        od:
        m_6_23 :=  matrice_repere_j(23, 6):
        RETURN(evalm( matrice_12_6 &* m_6_23 )):
end:

############################################################
#calcule la matrice de passage du repere 23 (lie au tronc)#
#vers le repere 19 (lie au pied gauche)                    #
############################################################
matrice_19_23 := proc() option remember;
local matrice_19_6, m_6_23, i;

        matrice_19_6 := inverse_matrice_passage(14):
        for i from 15 to 19
        do
            matrice_19_6 := evalm( inverse_matrice_passage(i) &* matrice_19_6 ):
        od:
        m_6_23 :=  matrice_repere_j(23, 6):
        RETURN(evalm( matrice_19_6 &* m_6_23 )):
end:

#####################################################
#calcul les coordonnees du tag (k) dans le repere 6 #
#####################################################
coord_tag_6 := proc(k) option remember;
local matrice, trans, rot;

        matrice := matrice_repere_j(eval(cat(reftag_, k)), 6):
        trans := subvector( matrice , 1..3, 4 ):
        rot := submatrix( matrice, 1..3,  1..3 ):
        RETURN(evalm(trans + rot &*  eval(cat(tag_,k)))):
end:
```

```
#####################################################################
#calcul les coordonnees des tags lies au tronc (tags 16, 17, 18)#
#dans le repere lie au pied droit(12) ou au pied gauche(19)     #
#sachant que passage_matrice doivent prendre les valeurs        #
#matrice_12_23() ou matrice_19_23() et k :16, 17, 18.           #
#####################################################################
coord_tronc := proc(k, passage_matrice) option remember;
local trans, rot;

        trans := subvector( passage_matrice , 1..3, 4 ):
        rot := submatrix( passage_matrice, 1..3,  1..3 ):
        RETURN(evalm(trans + rot &*  eval(cat(tag_,k)))):
end:

##########################
# changement de variable #
##########################
fonction_sortie := proc() option remember;
local cdm, tronc_16_19, tronc_17_19, tronc_18_19,
           tronc_16_12, tronc_17_12, tronc_18_12,
           m_12_23, m_19_23,
           sortie;

        cdm := coord_cdm_robot_6():
        print(`FONCTION DE SORTIE`);
        m_12_23 := matrice_12_23():
        m_19_23 := matrice_19_23():
        tronc_16_19 := coord_tronc(16, m_19_23):
        tronc_17_19 := coord_tronc(17, m_19_23):
        tronc_18_19 := coord_tronc(18, m_19_23):
        tronc_16_12 := coord_tronc(16, m_12_23):
        tronc_17_12 := coord_tronc(17, m_12_23):
        tronc_18_12 := coord_tronc(18, m_12_23):
        sortie := vector(NDDL, 0);
        sortie[1] := cdm[3]-coord_tag_6(5)[3]:
        sortie[2] := -cdm[2]+coord_tag_6(5)[2]:
        sortie[3] := cdm[1]-coord_tag_6(5)[1]:
        sortie[4] := cdm[3]-coord_tag_6(10)[3]:
        sortie[5] := -cdm[2]+coord_tag_6(10)[2]:
        sortie[6] := cdm[1]-coord_tag_6(10)[1]:
        sortie[7] := -tronc_17_12[1]+tronc_16_12[1]:
        sortie[8] := tronc_17_12[3]-tronc_16_12[3]:
        sortie[9] := tronc_18_12[3]-tronc_16_12[3]:
        sortie[10] := -tronc_17_19[1]+tronc_16_19[1]:
        sortie[11] := tronc_17_19[3]-tronc_16_19[3]:
        sortie[12] := tronc_18_19[3]-tronc_16_19[3]:
        sortie[13] := q[13]:
        sortie[14] := q[14]:
        sortie[15] := q[15]:
        sortie[16] := q[16]:
        sortie[17] := q[17]:
        sortie[18] := q[18]:
        sortie[19] := q[19]:
        sortie[20] := q[20]:
        sortie[21] := q[21]:
        RETURN(evalm(sortie)):
end:

jacobien_sortie := proc() option remember;
local sortie;

        sortie := fonction_sortie():
```

```
        print(`JACOBIEN DE SORTIE`);
        RETURN(jacobian(sortie, q)):
end:

hessiens_sortie := proc() option remember;
local sortie;

        sortie := fonction_sortie():
        print(`HESSIENS DE SORTIE`);
        RETURN(map(x->evalm(transpose(qdot) &* hessian(x, q) &* qdot), sortie)):
end:

#######################################
# changement de variable pour jacadi #
#######################################
sortie_jacadi := proc() option remember;
local cdm, sortie;

        cdm := coord_cdm_robot():
        print(`FONCTION DE SORTIE JACADI`);
        sortie := vector(NDDL, 0);
        sortie[1] := cdm[1]:
        sortie[2] := cdm[2]:
        sortie[3] := cdm[3]:
        sortie[4] := coord_tag(5)[1]:
        sortie[5] := coord_tag(5)[2]:
        sortie[6] := coord_tag(5)[3]:
        sortie[7] := coord_tag(10)[1]:
        sortie[8] := coord_tag(10)[2]:
        sortie[9] := coord_tag(10)[3]:
        sortie[10] := coord_tag(17)[2]-coord_tag(16)[2]:
        sortie[11] := coord_tag(17)[1]-coord_tag(16)[1]:
        sortie[12] := coord_tag(18)[1]-coord_tag(16)[1]:
        sortie[13] := coord_tag(1)[2]-coord_tag(2)[2]:
        sortie[14] := coord_tag(1)[1]-coord_tag(2)[1]:
        sortie[15] := coord_tag(4)[2]-coord_tag(2)[2]:
        sortie[16] := coord_tag(11)[2]-coord_tag(12)[2]:
        sortie[17] := coord_tag(11)[1]-coord_tag(12)[1]:
        sortie[18] := coord_tag(14)[2]-coord_tag(12)[2]:
        sortie[19] := q[13]:
        sortie[20] := q[14]:
        sortie[21] := q[15]:
        RETURN(evalm(sortie)):
end:

jacobien_sortie_jacadi := proc() option remember;
local sortie;

        sortie := sortie_jacadi():
        print(`JACOBIEN DE SORTIE JACADI`);
        RETURN(jacobian(sortie, q)):
end:

#####################################################################
# Exporte les fichiers "sortie.*", "jacobien.*", "hessien.*".#
#####################################################################
ecriture_commande := proc()
local T, J, H, TJ, JJ, fd,
      sortie_commande, jacobien_sortie_commande,
      hessiens_sortie_commande, jacadi_commande,
      jacobien_jacadi_commande ;
```

```
global q, qdot;

        T := matrix(NDDL, 1, fonction_sortie()):
        J := jacobien_sortie():
        H := matrix(NDDL, 1, hessiens_sortie()):
        TJ := matrix(NDDL, 1, sortie_jacadi()):
        JJ := jacobien_sortie_jacadi():

        printf("Ecriture de ""sortie.sci""\n"):
        fd := fopen("sortie.sci", WRITE):
        fprintf(fd, "function [var]=sortie(q)\n"):
        fprintf(fd, "var=fort('sortie_commande',q,2,'d','out',[%d,%d],1,'d')\n", rowdim(T), coldim(T)):
        fclose(fd):
        printf("Ecriture de ""sortie.c""\n"):
        T := convert(transpose(T), vector):
        sortie_commande := makeproc(T, [T, q::array(1..vectdim(q))]):
        C(sortie_commande, optimized, filename="sortie.c"):

        printf("Ecriture de ""jacobien_sortie.sci""\n"):
        fd := fopen("jacobien_sortie.sci", WRITE):
        fprintf(fd, "function [var]=jacobien_sortie(q)\n"):
        fprintf(fd, "var=fort('jacobien_sortie_commande',q,2,'d','out',[%d,%d],1,'d')\n",
                        rowdim(J), coldim(J)):
        fclose(fd):
        printf("Ecriture de ""jacobien_sortie.c""\n"):
        J := convert(transpose(J), vector):
        jacobien_sortie_commande := makeproc(J, [J, q::array(1..vectdim(q))]):
        C(jacobien_sortie_commande, optimized, filename="jacobien_sortie.c"):

        printf("Ecriture de ""hessiens_sortie.sci""\n"):
        fd := fopen("hessiens_sortie.sci", WRITE):
        fprintf(fd, "function [var]=hessiens_sortie(q, qdot)\n"):
        fprintf(fd, "var=fort('hessiens_sortie_commande',q,2,'d',qdot,3,'d','out',[%d,%d],1,'d')\n",
                        rowdim(H), coldim(H)):
        fclose(fd):
        printf("Ecriture de ""hessiens_sortie.c""\n"):
        H := convert(transpose(H), vector):
        hessiens_sortie_commande := makeproc(H,[H, q::array(1..vectdim(q)),qdot::array(1..vectdim(qdot))]):
        C(hessiens_sortie_commande, optimized, filename="hessiens_sortie.c"):

        printf("Ecriture de ""sortie_jacadi.sci""\n"):
        fd := fopen("sortie_jacadi.sci", WRITE):
        fprintf(fd, "function [var]=sortie_jacadi(q)\n"):
        fprintf(fd, "var=fort('jacadi_commande',q,2,'d','out',[%d,%d],1,'d')\n", rowdim(TJ), coldim(TJ)):
        fclose(fd):
        printf("Ecriture de ""sortie_jacadi.c""\n"):
        TJ := convert(transpose(TJ), vector):
        jacadi_commande := makeproc(TJ, [TJ, q::array(1..vectdim(q))]):
        C(jacadi_commande, optimized, filename="sortie_jacadi.c"):

        printf("Ecriture de ""jacobien_sortie_jacadi.sci""\n"):
        fd := fopen("jacobien_sortie_jacadi.sci", WRITE):
        fprintf(fd, "function [var]=jacobien_sortie_jacadi(q)\n"):
        fprintf(fd, "var=fort('jacobien_jacadi_commande',q,2,'d','out',[%d,%d],1,'d')\n",
                        rowdim(JJ), coldim(JJ)):
        fclose(fd):
        printf("Ecriture de ""jacobien_sortie_jacadi.c""\n"):
        JJ := convert(transpose(JJ), vector):
        jacobien_jacadi_commande := makeproc(JJ,[JJ, q::array(1..vectdim(q))]):
        C(jacobien_jacadi_commande, optimized, filename="jacobien_sortie_jacadi.c"):
end:
```

fichier *observateur1.sci*

```
//////////////////////////////////////////////////
// definitions de la fonction de cout a minimiser //
//////////////////////////////////////////////////
function [diff] = cout_obs(q2_test, q1, contacts_actifs, positions_contacts)

q_obs = [ q1; q2_test ];
C = positions_contacts([contacts_actifs; contacts_actifs; contacts_actifs;], :);

c_obs = contact(q_obs);
C_obs = c_obs([contacts_actifs; contacts_actifs; contacts_actifs;], :);

diff = ( C_obs - C );
diff = 0.5 * diff'*diff;

////////////////////////////////////
// gradient de la fonction de cout //
////////////////////////////////////
function [grad_diff] = gradient_obs(q2_test, q1, contacts_actifs, positions_contacts)

q_obs = [ q1; q2_test ];
C = positions_contacts([contacts_actifs; contacts_actifs; contacts_actifs;], :);

c_obs = contact(q_obs);
C_obs = c_obs([contacts_actifs; contacts_actifs; contacts_actifs;], :);

j_obs = jacobien_contact(q_obs);
J_obs = j_obs([contacts_actifs; contacts_actifs; contacts_actifs;], :);

grad_diff = J_obs' * ( C_obs - C );
grad_diff = grad_diff(16:21);

////////////////////////////////////
// definition d'un observateur de q2 //
////////////////////////////////////
function [q2_obs] = observateur (q1, q2_obs_prec, contacts_actifs, positions_contacts)

// definition d'une fonction de cout qui retourne la fonction a
//minimiser et son gradient.

deff('[d, gradient, ind] = costf(q2_test, ind)',
     'd = cout_obs(q2_test, q1, contacts_actifs, positions_contacts),
     gradient = gradient_obs(q2_test, q1, contacts_actifs,
     positions_contacts)');

[d,q2_test,gradient_opt]=optim(costf,q2_obs_prec,'gc');

q2_obs = q2_test;
q2_obs_prec = q2_obs;
q2_obs_prec = resume(q2_obs_prec);
```

fichier *observateur2.sci*

```
////////////////////////////////////////////////
// f(q2) = contact([q1,q2]) - positions_contacts //
////////////////////////////////////////////////
function [diff] = cout(q2, q1, contacts_actifs, positions_contacts)

q_obs = [ q1; q2 ];
C = positions_contacts([contacts_actifs; contacts_actifs; contacts_actifs;], :);

c_obs = contact(q_obs);
C_obs = c_obs([contacts_actifs; contacts_actifs; contacts_actifs;], :);

diff = ( C_obs - C );

////////////////////////////
// jacobien de q2 : J(q2)//
////////////////////////////
function [J_q2] = jacobien_q2(q2, q1, contacts_actifs)

q_obs = [ q1; q2 ];

j_obs = jacobien_contact(q_obs);
J_obs = j_obs([contacts_actifs; contacts_actifs; contacts_actifs;], :);

J_q2 = J_obs(:, 16:21);

//////////////////////////////////////////////////////////////////
// construction d'un observateur de q2 par la methode precedente //
//////////////////////////////////////////////////////////////////
function [q2_obs] = observateur (q1, q2_obs_prec, contacts_actifs, positions_contacts)

f_q2 = cout(q2_obs_prec, q1, contacts_actifs, positions_contacts);
J_q2 = jacobien_q2(q2_obs_prec, q1, contacts_actifs);

q2_obs = q2_obs_prec - inv(J_q2'*J_q2)*J_q2'*f_q2;

q2_obs_prec = q2_obs;
q2_obs_prec = resume(q2_obs_prec);
```

fichier *commande.sci*

```
//////////////////////////////////////////
// construction de l'observateur q2 (extrait
// du fichier commande.sci)
// premiere version
//////////////////////////////////////////
  q1 = q(1:15);
  positions_contacts = contact(q);
  q2_obs = observateur(q1, lambda($/3+1:2*$/3)>0, positions_contacts);
  q = [q1;q2_obs];

//////////////////////
// seconde version //
//////////////////////
   contacts_actifs = lambda($/3+1:2*$/3)>0;
   contacts_actifs = ((sum(contacts_actifs&pied_gauche)==4)&pied_gauche) |
                     ((sum(contacts_actifs&pied_droit)==4)&pied_droit)
   q2_obs = observateur(q(1:15), q2_obs_prec, contacts_actifs &
                        contacts_actifs_prec, positions_contacts);
//  test sur impact
  if sum(contacts_actifs & ~contacts_actifs_prec) > 0  then
        positions_contacts = contact([q(1:15);q2_obs]);
  end;
  contacts_actifs_prec = contacts_actifs;
  q = [q(1:15);q2_obs];

//////////////////////
// troisieme version //
//////////////////////
   contacts_actifs = lambda($/3+1:2*$/3)>0;
   contacts_actifs = ((sum(contacts_actifs&pied_gauche)==4)&pied_gauche) |
                     ((sum(contacts_actifs&pied_droit)==4)&pied_droit)
   for i=1:2,
        q2_obs = observateur(q(1:15), q2_obs_prec, contacts_actifs &
                             contacts_actifs_prec, positions_contacts);
   end;
// test sur impact
  if sum(contacts_actifs & ~contacts_actifs_prec) > 0  then
        positions_contacts = contact([q(1:15);q2_obs]);
  end;
  disp(norm(q2_obs-q(16:21)));
  contacts_actifs_prec = contacts_actifs;
  q = [q(1:15);q2_obs];

  [contacts_actifs_prec,positions_contacts] = resume(contacts_actifs_prec,positions_contacts);
```

Annexe B

Développement d'applications Client/Serveur

Socket.h

```
/*************************************************************************
**                                                                      **
** $Id: Rsock.h,v 1.1.1.1 1998/09/22 07:38:10 pissard Exp $             **
**                                                                      **
** Modifications:                                                       **
**  8 February 1993     (jls)   creation                                **
**                                                                      **
** Description:                                                         **
** Message exchanges under TCP/IP                                       **
**                                                                      **
**                                                                      **
*************************************************************************/
#ifndef socketLib_h
#define socketLib_h

#include <Rmi.h>

/* La structure de memoire de l'ensemble des sockets */
typedef struct sockServer {
   int s_generic;
   int s_client;
   int port_nb;
   char hostnumber[100];
} SOCK_SERVER;

#ifdef __cplusplus
extern "C" {
#endif
#ifdef __STDC__
extern int sockClientOpen(SOCK_SERVER *pSock,char *host,int port_nb);
extern int sockServerOpen(SOCK_SERVER *socket_server,int port_nb);
extern int sockClose (SOCK_SERVER *pSock);
extern int sockSendString(SOCK_SERVER *pSock,char *msg);
extern int sockSend(SOCK_SERVER *pSock,char *msg,unsigned int size);
extern int sockRec(SOCK_SERVER *pSock,char *msg,unsigned int sizemax);
extern int sockSendImage(SOCK_SERVER *pSock,MI_IMAGE image);
```

```
extern int sockRecImage(SOCK_SERVER *pSock,MI_IMAGE *image);
extern int sockIsAlive(SOCK_SERVER *pSock);
extern unsigned int sockMsgLength(SOCK_SERVER *pSock);

#ifdef VXWORKS
extern SOCK_SERVER *sockGetStructure(int clientId, SEM_ID *sem_mutex);
extern int sockServerOpenMultiClients(int port_nb, FUNCPTR fonction);
#endif

#else
extern int sockClientOpen();
extern int sockClose ();
extern int sockSendImage();
extern int sockRecImage();
extern int sockSendString();
extern int sockSend();
extern int sockRec();
extern unsigned int sockMsgLength();
extern int sockIsAlive();
extern int sockKernelBufferSize();
extern int sockServerOpen();
#endif

#ifdef __cplusplus
}
#endif

#endif   /* ! socketLib_h */
```

ComBip.c

```
/*****************************  Bipede   *****************************
**                                                                   **
** File           : $HOME/BIP/utils_vxworks/Socket/comBip.c          **
** Author         : RPG, Soraya Arias and Sébastien Jardé            **
** Version        : 1.1                                              **
** Creation       : 1 Juillet 2002                                   **
**                                                                   **
****************************** Description **************************
**                                                                   **
**  Gestion par string de la communication pour le calcul de         **
**  la loi de commande du robot BIP                                  **
**                                                                   **
***************************** Modifications ************************
**                                                                   **
** PBPBPB : affectation de la structure combip desc !!               **
**                                                                   **
***********************************************************************
** (c) Copyright 2000, INRIA, all rights reserved                    **
***********************************************************************/
#ifndef __combip_c
#define __combip_c

#include <stdio.h>
#include <string.h>
#include "comBip.h"

#define PORT 5001
#define MAX_SIZE_MESSAGE 1024
#define SIZE_STR 1024

//dimension du vecteur des contacts actifs
#define DIM_C 24

#define Q_VECTTYPE 0
#define QDOT_VECTTYPE 1
#define LAMBDA_VECTTYPE 2
#define COM_VECTTYPE 3

#define COMBIPSTATUS_NEW 0
#define COMBIPSTATUS_OLD 1

/***************************/
/* Combip Structure created */
/***************************/
static COMBIP_DESC bipVect;

/*****************/
/* Socket created */
/*****************/
static SOCK_SERVER s;
SOCK_SERVER *combipSocket()
{
  return &s;
}

/*****************************/
/* Create a combip structure */
/*****************************/
int combipStructCreate(COMBIP_DESC *combipStruct, int dim)
{
```

```
 combipStruct->dimrow = dim;
 combipStruct->lambda = (double *)malloc(DIM_C*sizeof(double));
 combipStruct->q = (double *)malloc(dim*sizeof(double));
 combipStruct->qdot = (double *)malloc(dim*sizeof(double));
 combipStruct->commande = (double *)malloc(dim*sizeof(double));
 combipStruct->status = COMBIPSTATUS_OLD;

  return OK;
}

/*********************************************************/
/* Return a Commande vector pointer from a combip structure */
/*********************************************************/
void *combipGetComPointer(COMBIP_DESC *combipStruc)
{
  if (combipStruc == NULL) {
    fprintf(stderr, "combipGetPointer***: arg combipStruc NULL !\n");
    return(NULL);
  }
  return(combipStruc->commande);
}

/*****************************************************/
/* Return a q vector pointer from a combip structure */
/*****************************************************/
void *combipGetqPointer(COMBIP_DESC *combipStruc)
{
  if (combipStruc == NULL) {
    fprintf(stderr, "combipGetPointer***: arg combipStruc NULL !\n");
    return(NULL);
  }
  return(combipStruc->q);
}

/***********************************************************/
/* Convert a vector from the combip structure into a string  */
/***********************************************************/
int convertCombipVect2String(double *combipVect,char *vect2Str,int dim)
{
  int i;
  char *pvect = vect2Str;

  for(i = 0; i < dim; i++) {
    sprintf(pvect,"%lf:", combipVect[i]);
    pvect = vect2Str + strlen(vect2Str);
  }

  return OK;
}

/*********************************************************************/
/* Convert a string received through the socket into a combip vector */
/*********************************************************************/
int convertString2CombipVect(double *combipVect,char *str2Vect,int dim)
{
  int i;
  char *pstr = (char *)malloc((sizeof(char))*strlen(str2Vect)+1);

  for(i = 0; i < dim ; i++) {
    if (i == 0) {
      pstr = strtok(str2Vect,":");
    }
```

```
    else {
      pstr = strtok(NULL,":");
    }

    sscanf(pstr,"%lf", &(combipVect[i]));
  }
  return OK;
}

/*********************************************************/
/* Send combip vector as message string through the socket */
/*********************************************************/
int combipSockSendVect(COMBIP_DESC *bipVect, int vectType)
{
  char vect2Str[SIZE_STR];

  if (bipVect != NULL) {
    switch (vectType) {
    case Q_VECTTYPE: {
      convertCombipVect2String(bipVect->q, vect2Str, bipVect->dimrow);
      break;
    }
    case QDOT_VECTTYPE: {
      convertCombipVect2String(bipVect->qdot,vect2Str,bipVect->dimrow);
      break;
    }
    case LAMBDA_VECTTYPE: {
      convertCombipVect2String(bipVect->lambda, vect2Str, DIM_C);
      break;
    }
    case COM_VECTTYPE: {
      convertCombipVect2String(bipVect->commande, vect2Str, bipVect->dimrow);
      break;
    }
    default: {
      printf("Vector Type not supported \n");
      return ERROR;
    }
  }

    return(sockSendString(combipSocket(), vect2Str)) ;
  }
  else {
    printf("combipSockSend combip structure error ! \n");
    return ERROR;
  }
  return OK;
}

/*********************************************************/
/* Receive combip vector as message string through the socket */
/*********************************************************/
int combipSockRecVect(COMBIP_DESC *bipVect, int vectType)
{
  char str2Vect[SIZE_STR];

    if (sockRec(combipSocket(), str2Vect, MAX_SIZE_MESSAGE) == ERROR)
    {
      return ERROR;
    }

  if (bipVect != NULL) {
```

```
    switch (vectType) {
    case Q_VECTTYPE:
      convertString2CombipVect(bipVect->q, str2Vect, bipVect->dimrow);
      break;
    case QDOT_VECTTYPE:
      convertString2CombipVect(bipVect->qdot,str2Vect,bipVect->dimrow);
      break;
    case LAMBDA_VECTTYPE:
      convertString2CombipVect(bipVect->lambda, str2Vect, DIM_C);
      break;
    case COM_VECTTYPE:
      convertString2CombipVect(bipVect->commande, str2Vect, bipVect->dimrow);
      break;
    default:
      printf("Vector Type not supported \n");
      return ERROR;
    }
  }
  else {
    printf("combipSockRec combip structure error ! \n");
    return ERROR;
  }
  return OK;
}

/*********************************/
/* Send q Vector through socket */
/*********************************/
int combipSockSendq(COMBIP_DESC *bipVect)
{
  if(combipSockSendVect(bipVect, Q_VECTTYPE) == ERROR) {
    printf("combipSockSendq -> Error sending q vector through socket, error = %d, OK= %d \n", ERROR, OK);
    return ERROR;
  }

  return OK;
}

/***********************************/
/* Send qdot Vector through socket */
/***********************************/
int combipSockSendqdot(COMBIP_DESC *bipVect)
{
  if(combipSockSendVect(bipVect, QDOT_VECTTYPE) == ERROR) {
    printf("combipSockSendqdot -> Error sending qdot through socket , error = %d, OK= %d \n", ERROR, OK);
    return ERROR;
  }

  return OK;
}

/*************************************/
/* Send Lambda Vector through socket */
/*************************************/
int combipSockSendLambda(COMBIP_DESC *bipVect)
{
  if(combipSockSendVect(bipVect, LAMBDA_VECTTYPE) == ERROR) {
    printf("combipSockSendLambda -> Error sending Lambda vector through socket \n");
    return ERROR;
  }

  return OK;
```

```
}

/*************************************/
/* Send commande Vector through socket */
/*************************************/
int combipSockSendCom(COMBIP_DESC *bipVect)
{
  if(combipSockSendVect(bipVect, COM_VECTTYPE) == ERROR) {
    printf("combipSockSendCom -> Error sending Commande vector through socket \n");
    return ERROR;
  }

  return OK;
}

/*********************************/
/* Receive q Vector through socket */
/*********************************/
int combipSockRecq(COMBIP_DESC *bipVect)
{
  if(combipSockRecVect(bipVect, Q_VECTTYPE) == ERROR) {
    printf("combipSockSendq -> Error receiving q vector through socket \n");
    return ERROR;
  }

  return OK;
}
/*************************************/
/* Receive qdot Vector through socket */
/*************************************/
int combipSockRecqdot(COMBIP_DESC *bipVect)
{
  if(combipSockRecVect(bipVect, QDOT_VECTTYPE) == ERROR) {
    printf("combipSockSendqdot -> Error receiving qdot vector through socket \n");
    return ERROR;
  }

  return OK;
}

/***************************************/
/* Receive Lambda Vector through socket */
/***************************************/
int combipSockRecLambda(COMBIP_DESC *bipVect)
{
  if(combipSockRecVect(bipVect, LAMBDA_VECTTYPE) == ERROR) {
    printf("combipSockRecLambda -> Error receiving lambda vector through socket \n");
    return ERROR;
  }

  /* Change combip structure status */
  bipVect->status = COMBIPSTATUS_NEW;

  return OK;
}

/*****************************************/
/* Receive Commande Vector through socket */
/*****************************************/
int combipSockRecCom(COMBIP_DESC *bipVect)
{
  if(combipSockRecVect(bipVect, COM_VECTTYPE) == ERROR) {
```

```
    printf("combipSockRecCom -> Error receiving Commande vector through socket \n");
    return ERROR;
  }

  /* Change combip structure status */
  bipVect->status = COMBIPSTATUS_NEW;

  return OK;
}

/************************/
/* Ouvre 1 socket client */
/************************/
int combipOpenClient(char *target)
{
 int tempo=0;
 char message[MAX_SIZE_MESSAGE];

 while (sockClientOpen(&s, target, PORT) == ERROR) {
    sleep(2); /* time for vxworks to run the sockServer */
    tempo++;
    /* Temporisation */
    if (tempo > 10) {
      fprintf(stderr,"sock: Give up...\n");
       return(ERROR);
    } else {
      fprintf(stderr,"sock: Retrying...\n");
    }
  }
#ifdef DEBUG
  printf("connection ok\n");
#endif

  if (sockIsAlive(&s) == ERROR) {
    printf("problem socket...closing\n");
    sockClose(&s);
    return ERROR;
  } else {
    sockRec(&s, message, MAX_SIZE_MESSAGE);
    printf("message received <%s>\n", message);
  }
  return OK;
}

/*****************/
/* socket closed */
/*****************/
int combipClose()
{
  sockClose(&s);
  return OK;
}

#endif
```

ComBip.h

```
/*****************************  Bipede   *****************************
**                                                                   **
** File           : $HOME/BIP/utils_vxworks/Socket/comBip.c          **
** Author         : RPG, Soraya Arias and Sébastien Jardé            **
** Version        : 1.1                                              **
** Creation       : 1 Juillet 2002                                   **
**                                                                   **
***************************** Description ***************************
**                                                                   **
**  Gestion par string de la communication pour le calcul de         **
**  la loi de commande du robot BIP                                  **
**                                                                   **
***************************** Modifications *************************
**                                                                   **
** PBPBPB : affectation de la structure combip desc !!               **
**                                                                   **
***********************************************************************
** (c) Copyright 2000, INRIA, all rights reserved                    **
***********************************************************************/
#ifndef __combip_h
#define __combip_h

#include "Rsock.h"

#define PORT 5001
#define MAX_SIZE_MESSAGE 1024
#define SIZE_STR 1024

typedef struct _COMBIP_DESC
{
  int dimrow;
  double *q;
  double *qdot;
  double *lambda;
  double *commande;
  int status;
} COMBIP_DESC;


SOCK_SERVER *combipSocket();

/* Create combip structure */
int combipStructCreate(COMBIP_DESC *combipStruct, int dim);
/* Send q Vector throug socket */
int combipSockSendq(COMBIP_DESC *bipVect);
/* Send qdot Vector throug socket */
int combipSockSendqdot(COMBIP_DESC *bipVect);
/* Send Lambda Vector throug socket */
int combipSockSendLambda(COMBIP_DESC *bipVect);
/* Send Commande Vector throug socket */
int combipSockSendCom(COMBIP_DESC *bipVect);

/* Receive q Vector throug socket */
int combipSockRecq(COMBIP_DESC *bipVect);
/* Receive qdot Vector throug socket */
int combipSockRecqdot(COMBIP_DESC *bipVect);
/* Receive Lambda Vector throug socket */
int combipSockRecLambda(COMBIP_DESC *bipVect);
/* Receive Commande Vector throug socket */
int combipSockRecCom(COMBIP_DESC *bipVect);
```

```
/*******************************************************/
/* Ouvre 1 socket Serveur, lance 1 tache qui boucle avec */
/* 1 delay et regulierement fait 1 demande de calcul     */
/*******************************************************/
int combipOpen();

/*****************************************************/
/* Ouvre 1 socket Client, attend 1 requete de calcul, */
/* fait le calcul, et renvoie la valeur  du calcul    */
/*****************************************************/
int combipOpenClient(char *target);

int combipClose();

#endif
```

combipservtest.c

```
#define _POSIX_C_SOURCE 199506

#include <pthread.h>
#include <limits.h>
#include <stdio.h>
#include <unistd.h>

#include "comBip.h"

#define NBLOOP 1000

typedef void (*FUNCPTR)(void *);
//special type for cast in thr_create()
typedef void * (*SOSO)  (void *);

void modele_gravite(double G[21],double q[21]);
void jacobien_contact(double q[21], double J[21][24]);
void inertie(double q[21], double M[21][21]);
void contact(double q[21], double C[1][24]);
void sortie(double q[21], double T[1][21]);
void jacobien_sortie(double q[21], double J[21][21]);
void hessiens_sortie(double q[21], double qdot[21], double H[1][21]);

COMBIP_DESC bipVect;

int orcSpawn(pthread_t *tid, FUNCPTR funcptr, void * arg)
{
  int status;
  pthread_attr_t attributes;

  // Allocation and initialization of the attribute structure
  pthread_attr_init(&attributes);

  //create pthread
  status =  pthread_create(tid, &attributes, (SOSO)funcptr, arg);
  printf("pthread_create error = %d \n", status);

  return status;
}

/*****************************************/
/* Server Program wait for vectors from  */
/* the client and comput the control law */
/*****************************************/
int combipServ()
{
  int k,i,l,j;
  double q[21], qdot[21], J_C[21][24], M[21][21],
                          H_SH[21], M_PD[21], C_LAMBDA[21],
                          T[1][21], J_S[21][21], H[1][21],
                          s_hs[21], s_T[21];

  /* Create the Bip Struct -> vector dimension = 21 */
  combipStructCreate(&bipVect,21);

  /* Server opens a socket */
  if (combipOpen() != 0) {
    printf("Server can not open socket \n");
    return ;
  }
```

```
/* compute with q, qdot and lambda in input and Commande in output */
for ( k = 0 ; k < NBLOOP ; k++)
{
  printf("receive q, qdot and Lambda vector...!\n");
  combipSockRecq(&bipVect);
  combipSockRecqdot(&bipVect);
  combipSockRecLambda(&bipVect);

  /* Computation */
      printf("compute...!\n");

  /* somme sortie(q) - hessiens(q) */
      sortie(bipVect.q, T);
      hessiens_sortie(bipVect.q, bipVect.qdot, H);
      for(l=0;l<bipVect.dimrow;l++) s_hs[l] = T[1][l] - H[1][l] ;

  /* multiplication H(q)*[sortie(q) - hessiens(q)] */
      jacobien_sortie(bipVect.q, J_S);
      for(l=0;l<21;l++)H_SH[l]=0;
      for(i=0;i<bipVect.dimrow;i++)
      {
              for(j=0;j<bipVect.dimrow;j++)
              {
                      H_SH[i] = H_SH[i] +J_S[i][j]*s_hs[j];
              }
      }
  /* multiplication M(q)*H(q)*[sortie(q) - hessiens(q)] */
      for(l=0;l<21;l++)M_PD[l]=0;
      inertie(bipVect.q, M);
      for(i=0;i<bipVect.dimrow;i++)
      {
              for(j=0;j<bipVect.dimrow;j++)
              {
                      M_PD[i] = M_PD[i] +M[i][j]*H_SH[j];
              }
      }


  /* multiplication C(q)*lambda */
  for(l=0;l<21;l++)C_LAMBDA[l]=0;
  jacobien_contact(bipVect.q, J_C);
  for(i=0;i<bipVect.dimrow;i++)
  {
      for(j=0;j<24;j++)
      {
              C_LAMBDA[i] = C_LAMBDA[i] +J_C[i][j]*bipVect.lambda[j];
      }
  }

  /* M(q)*H(q)*[sortie(q) - hessiens(q)] + C(q)*lambda */
  for(l=0;l<bipVect.dimrow;l++)
  {
      s_T[l] = M_PD[l] + C_LAMBDA[l];
      bipVect.commande[l] = s_T[l];
  }

  /* Send Message including result G to Client */
  printf("send Commande vector...!\n");
  combipSockSendCom(&bipVect);
}
```

```
  printf("closing socket\n");
  combipClose();
  return(0);
}


int main(int argc, char **argv)
{
  int n;
  pthread_t ServManageId;

  if (orcSpawn(&ServManageId,(FUNCPTR) combipServ, NULL) == ERROR)
  {
    printf("ERROR:Can't create thread from the server management \n");
    return(1);
  }

  printf("orcSpawn called, entering pthread_join \n");

  /* Suspend the main thread on the terminaison of the ServManageId thread */
  if (n = pthread_join(ServManageId, NULL)) {
    fprintf(stderr, "pthread_join: %s\n", strerror(n));
    exit(1);
  }

  printf("Main combipservtest end. \n");

  return(0);
}
```

VwSockClienTest.c

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <time.h>
#include <sysLib.h>

#include "comBip.h"

#define CHEMIN "/home/mouflon/jarde/BIP/utils_vxworks/Socket/test_sock_bip_client"
#define NBLOOP 1000
#define DIM_CONTACTS 24

int count=0;

/***********************/
/* creation d'un timer */
/***********************/
int clk()
{
count++;
return(count);
}

/*****************************************/
/* fonction de sauvegarde des temps de com */
/*****************************************/
int combipSaveTime(char *filename, int tableau[NBLOOP])
{
  int i;
  FILE *fp;

  /* open the file */
  fp = fopen(filename,"w");
  if (!fp) return ERROR;

  /* enregistrement des valeurs */
  for (i=0; i<NBLOOP;i++)
    {
      fprintf(fp, "%d\n", tableau[i]);
    }

  /* fermeture du fichier */
  fclose(fp);

  return(OK);
}

/*********************/
/* Init combipStruct */
/*********************/
int combipAffectTest(COMBIP_DESC *combipStruc)
{
  int i;

  for(i = 0; i < combipStruc->dimrow ; i++) {
    combipStruc->q[i] = 0;
    combipStruc->qdot[i] = 0;
    combipStruc->commande[i] = 0;
```

```
  }

  for(i = 0; i < DIM_CONTACTS ; i++) {
    combipStruc->lambda[i] = 0;
  }

  return OK;
}

/*********************************************/
/* Client program send a request for the server */
/* and wait for the message of the server       */
/*********************************************/
int sockclientest()
{
  COMBIP_DESC bipVect;
  char target[20];
  char vect2Str[SIZE_STR];
  int i, k, clk_send, clk_receive;
  int tab[NBLOOP];
  char name[64];

  /* mise en place de l'interruption timer */
  /* interruption toutes les 1 ms (1000/frequence) */
  sysAuxClkRateSet(1000);
  sysAuxClkConnect(clk, 0);

  /* Create the Bip Struct -> vector dimension = 21 */
  combipStructCreate(&bipVect,21);

  /* Client opens connection */
  printf("entree le nom du serveur cible:");
  scanf("%s", target);
  printf("Le nom du serveur cible est:%s\n",target);

  combipOpenClient(target);

  /* Create q, qdot and lambda vector message to send */
  combipAffectTest(&bipVect);

  /* launch clock */
  sysAuxClkEnable();
  printf("clk_initiale = %d\n", count);

  for (k = 0 ; k < NBLOOP ; k++)
  {
        /* Send q, qdot and lambda as input */
        clk_send = count;
        combipSockSendq(&bipVect);
        combipSockSendqdot(&bipVect);
        combipSockSendLambda(&bipVect);

        /* Wait for the message from the server to get the G value */
        combipSockRecCom(&bipVect);

        /* get time and eval it between the sent and the reception*/
        clk_receive = count;
        printf("timeval = %d ms\n", clk_receive - clk_send);

        /* fill tab with the comunication time */
        tab[k] = clk_receive - clk_send;
```

```
        /* Display commande */
        for(i = 0 ;  i < bipVect.dimrow ; i ++)
        {
                printf("com[%d] = %lf \n", i, bipVect.commande[i]);
        }

        /* reinitialize combipStruct */
        combipAffectTest(&bipVect);
  }

  /* stop clock */
  sysAuxClkDisable();

  /* save the time result in a file */
  sprintf(name, "%s/time.data", CHEMIN);
  combipSaveTime(name, tab);

  /* fermeture de la socket */
  printf("closing socket\n");
  combipClose();
  return(0);
}
```

Annexe C

Conventions

Mouvements, Articulations et Moteurs

Le tableau suivant récapitule les différents types de mouvements dont est capable le robot ainsi que l'appellation des articulations qui permettent ces mouvements.

Mouvement	**Articulation associée**
Rotation interne-externe du pied par rapport à la jambe	Chevilles axe frontal
Flexion et extension du pied par rapport à la jambe	Chevilles axe sagittal
Flexion et extension du tibia par rapport au fémur	Genou axe sagittal
Flexion et extension du fémur par rapport au tronc	Hanche axe sagittal
Abduction et adduction du fémur	Hanche axe vertical
Rotation interne et externe de la hanche	Hanche axe frontal
Rotation interne externe des lombaires	Tronc axe vertical
Inflexion latérale des lombaires	Tronc axe frontal
Flexion et extension des lombaires	Tronc axe sagittal

TAB. C.1 – Relation entre mouvements et articulations

Remarque : Lorsqu'on utilisera des logiciels tels que SCILAB la numérotation des éléments du vecteur se fera de 1 à 15 ! Il y a un moteur par articulation, sauf dans le cas des chevilles et du tronc où les mouvements frontal/sagittal sont dus aux effets du mouvement combiné de deux moteurs montés en parallèle.

	Articulation	Situation	Position
0	Cheville axe frontal	jambe droite	q[0]
1	Cheville axe sagittal	jambe droite	q[1]
2	Genou axe sagittal	jambe droite	q[2]
3	Hanche axe sagittal	jambe droite	q[3]
4	Cheville axe frontal	jambe gauche	q[4]
5	Cheville axe sagittal	jambe gauche	q[5]
6	Genou axe sagittal	jambe gauche	q[6]
7	Hanche axe sagittal	jambe gauche	q[7]
8	Hanche axe vertical	jambe droite	q[8]
9	Hanche axe frontal	jambe droite	q[9]
10	Hanche axe vertical	jambe gauche	q[10]
11	Hanche axe frontal	jambe gauche	q[11]
12	Tronc axe vertical	bassin	q[12]
13	Tronc axe frontal	lombaires	q[13]
14	Tronc axe sagittal	lombaires	q[14]

TAB. C.2 – Numérotation des articulations et notation des positions articulaires associées

	Moteur	Situation	Position
0	Cheville droite	extérieur	th[0]
1	Cheville droite	intérieur	th[1]
2	Genou droit		th[2]
3	Hanche droit sagittale		th[3]
4	Cheville gauche	intérieur	th[4]
5	Cheville gauche	extérieur	th[5]
6	Genou gauche		th[6]
7	Hanche gauche sagittale		th[7]
8	Hanche droite frontale		th[8]
9	Hanche droite verticale		th[9]
10	Hanche gauche frontale		th[10]
11	Hanche gauche verticale		th[11]
12	Bassin vertical		th[12]
13	Tronc	droit	th[13]
14	Tronc	gauche	th[14]

TAB. C.3 – Numérotation des moteurs et notation pour la position des moteurs

Annexe D

Étalonnage des potentiomètres

Fichier *CalibPotars.sce*

```
Calib = zeros(2, 15);

for k = 0:14,
  q = fscanfMat('lectures_potars/q'+string(k)+'.data');
  pot = fscanfMat('lectures_potars/potars'+string(k)+'.data');
  A = [sum(pot.^2), sum(pot); sum(pot), size(pot, 1)]\[sum(pot.*q); sum(q)];

  xset('window', k);
  xname('Erreur en degres sur l''etalonnage du potentiometre no. '+string(k));
  xbasc();
  plot(pot,q*180/%pi);
  plot2d(q*180/%pi, (q-A(1)*pot-A(2))*180/%pi);

  Calib(:, k+1) = A;
end;

template_BipPotars_h = "..
/*********************************************************\n..
Fichier d''etalonnage des 15 potentiometres du robot BIP,\n..
genere automatiquement par CalibPotars.sce.\n\n..
Ces constantes relient la position articulaire q (en radians)\n..
a la lecture en sortie des potentiometres potars (en Volt)\n..
de la facon suivante:\n\n..
         q = Potxxxx_x * potars + Offxxxx_x\n\n..
*********************************************************/\n..
#define PotCheD_F %f\n..
#define OffCheD_F %f\n\n..
#define PotCheD_S %f\n..
#define OffCheD_S %f\n\n..
#define PotGenD_S %f\n..
#define OffGenD_S %f\n\n..
#define PotHanD_S %f\n..
#define OffHanD_S %f\n\n..
#define PotCheG_F %f\n..
#define OffCheG_F %f\n\n..
#define PotCheG_S %f\n..
#define OffCheG_S %f\n\n..
#define PotGenG_S %f\n..
#define OffGenG_S %f\n\n..
```

```
#define PotHanG_S %f\n..
#define OffHanG_S %f\n\n..
#define PotHanD_F %f\n..
#define OffHanD_F %f\n\n..
#define PotHanD_V %f\n..
#define OffHanD_V %f\n\n..
#define PotHanG_F %f\n..
#define OffHanG_F %f\n\n..
#define PotHanG_V %f\n..
#define OffHanG_V %f\n\n..
#define PotLomb_V %f\n..
#define OffLomb_V %f\n\n..
#define PotLomb_F %f\n..
#define OffLomb_F %f\n\n..
#define PotLomb_S %f\n..
#define OffLomb_S %f\n\n";

unix("rm -f BipPotars.h");
fprintf("BipPotars.h", template_BipPotars_h, Calib(1), Calib(2),..
Calib(3), Calib(4), Calib(5), Calib(6), Calib(7), Calib(8),..
Calib(9), Calib(10), Calib(11), Calib(12), Calib(13), Calib(14),..
Calib(15), Calib(16), Calib(17), Calib(18), Calib(19), Calib(20),..
Calib(21), Calib(22), Calib(23), Calib(24), Calib(25), Calib(26),..
Calib(27), Calib(28), Calib(29), Calib(30));
```

Fichier *BipPotars.h*

```
/*******************************************************
Fichier d'etalonnage des 15 potentiometres du robot BIP.
*******************************************************/
#define PotCheD_F 0.629684
#define OffCheD_F -3.580310
#define PotCheD_S -0.630323
#define OffCheD_S 2.571803
#define PotGenD_S -0.647265
#define OffGenD_S 3.794288
#define PotHanD_S 0.613690
#define OffHanD_S -2.859404
#define PotCheG_F -0.614876
#define OffCheG_F 3.564100
#define PotCheG_S 0.611579
#define OffCheG_S -3.442484
#define PotGenG_S 0.613117
#define OffGenG_S -2.210423
#define PotHanG_S -0.621639
#define OffHanG_S 2.543111
#define PotHanD_F -0.085811
#define OffHanD_F 0.382532
#define PotHanD_V 0.086198
#define OffHanD_V -0.325520
#define PotHanG_F 0.086652
#define OffHanG_F -0.412984
#define PotHanG_V -0.087146
#define OffHanG_V 0.310868
#define PotLomb_V 0.086611
#define OffLomb_V -0.317327
#define PotLomb_F 0.651830
#define OffLomb_F -1.497027
#define PotLomb_S 0.655513
#define OffLomb_S -5.741643
```

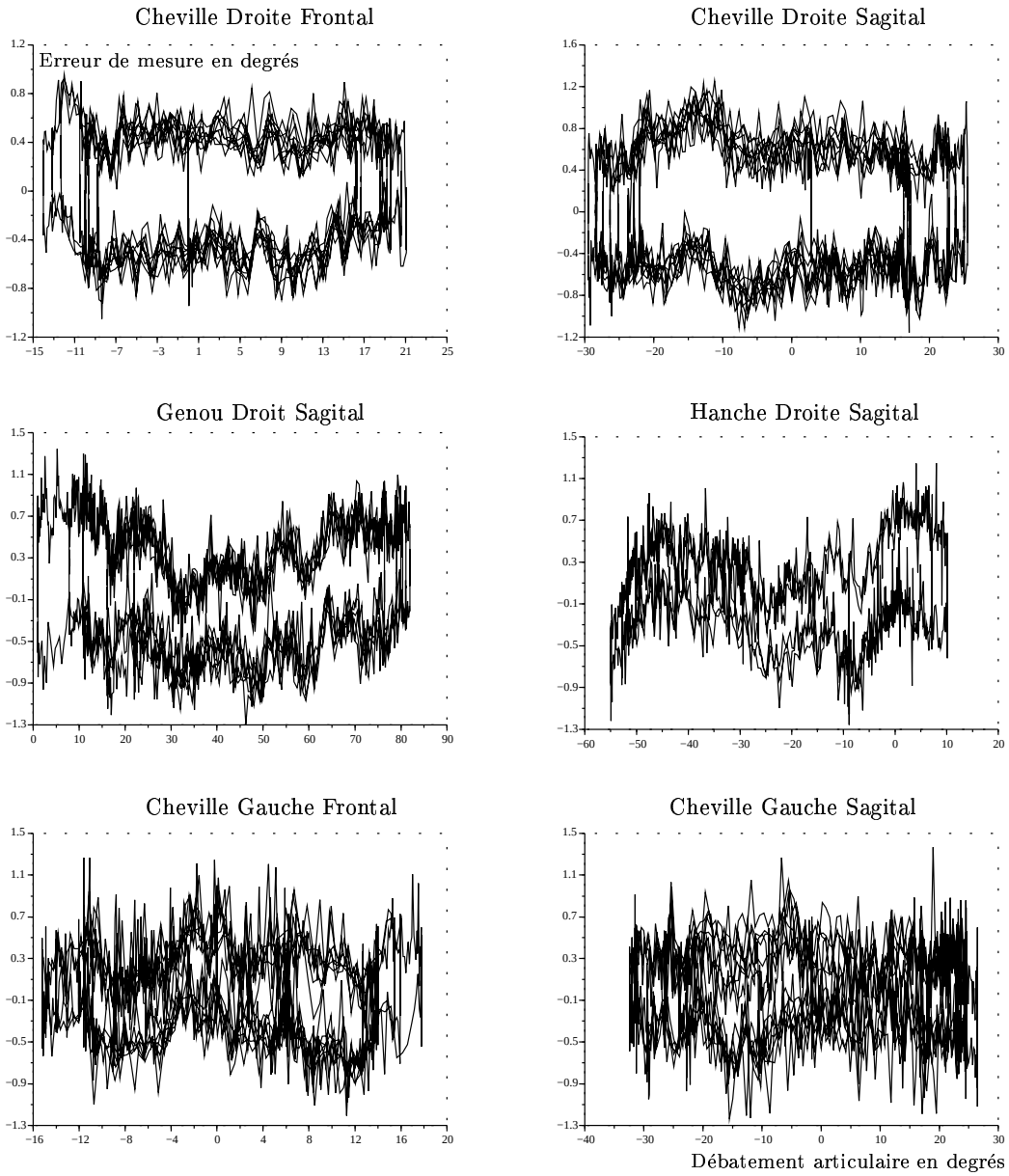


FIG. D.1 – Evaluation de l'erreur de mesure des potentiomètres.

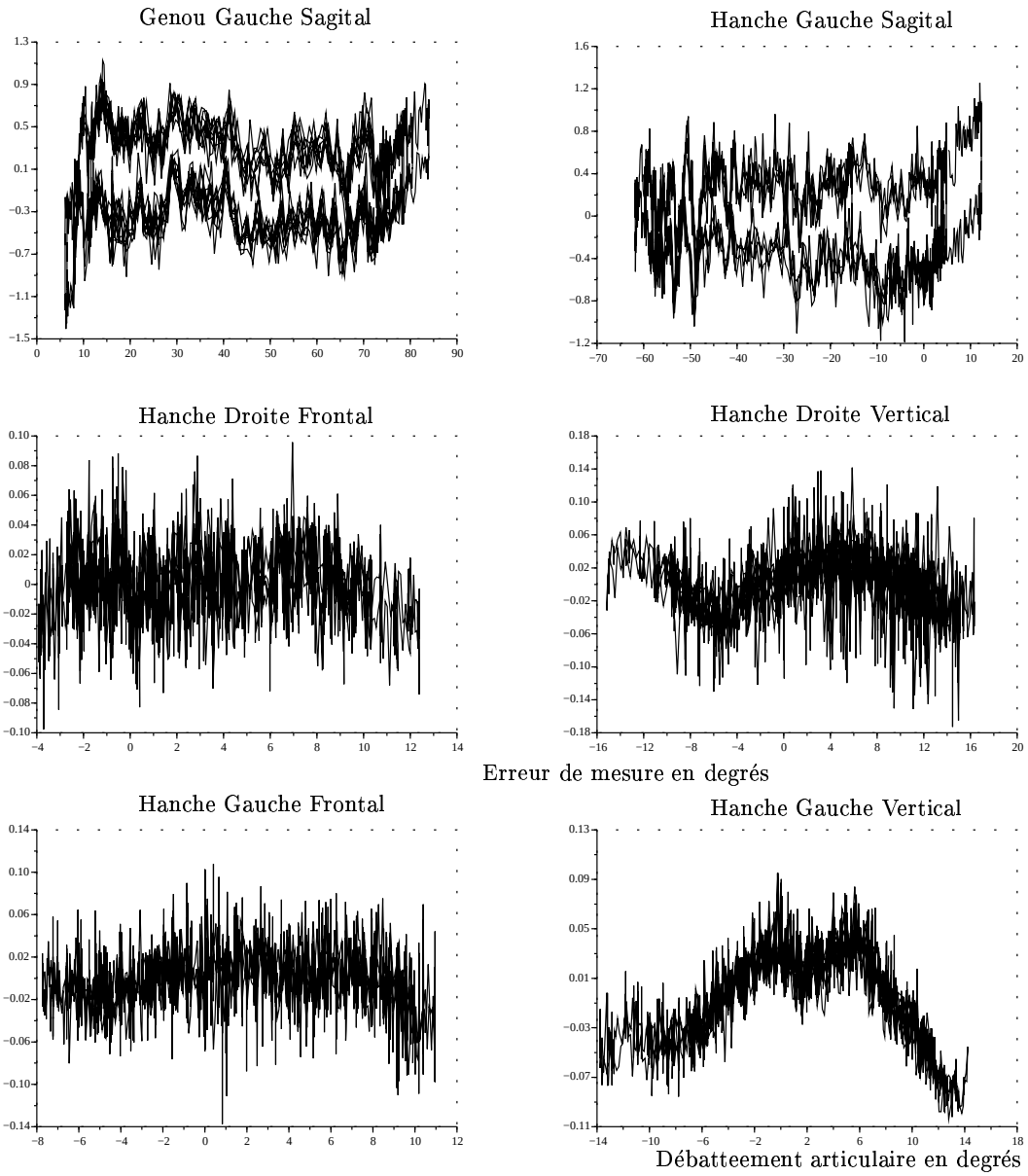


FIG. D.2 – Evaluation de l'erreur de mesure des potentiomètres.

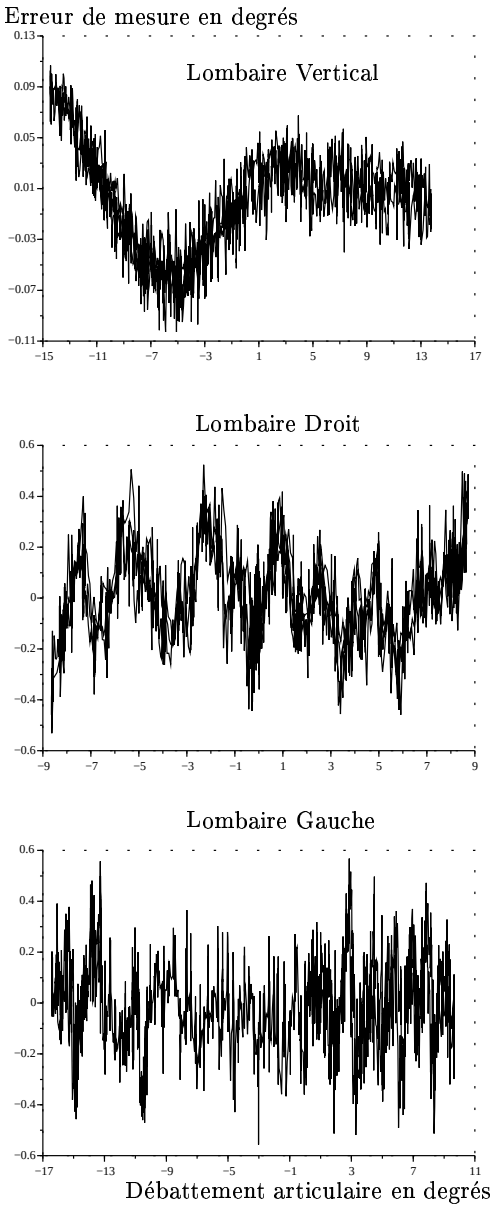


FIG. D.3 – Evaluation de l'erreur de mesure des potentiomètres.

Annexe E

Manipulations expérimentales

Les postures du robot BIP

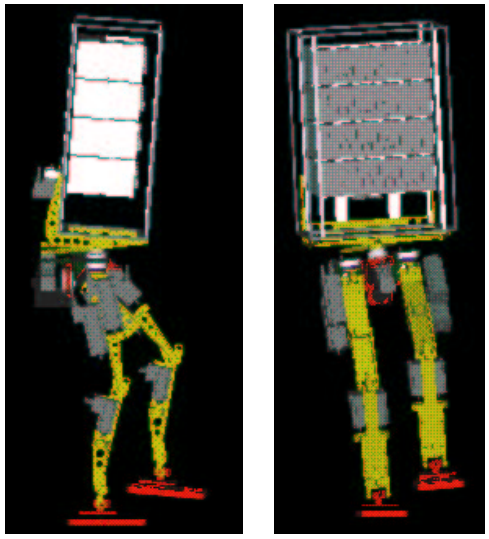


FIG. E.1 – Position p0 : position initiale

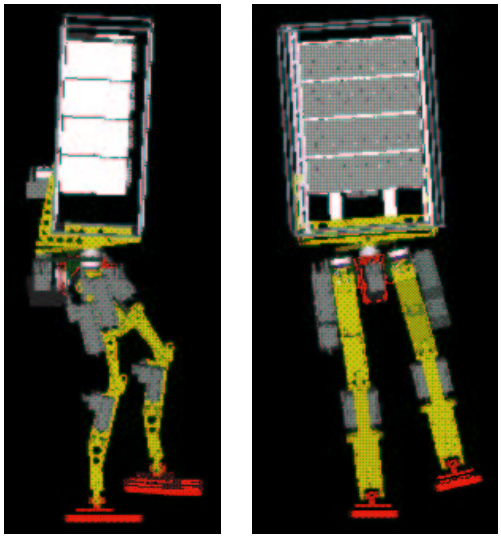


FIG. E.2 – Position p1 : pied petit écart côté

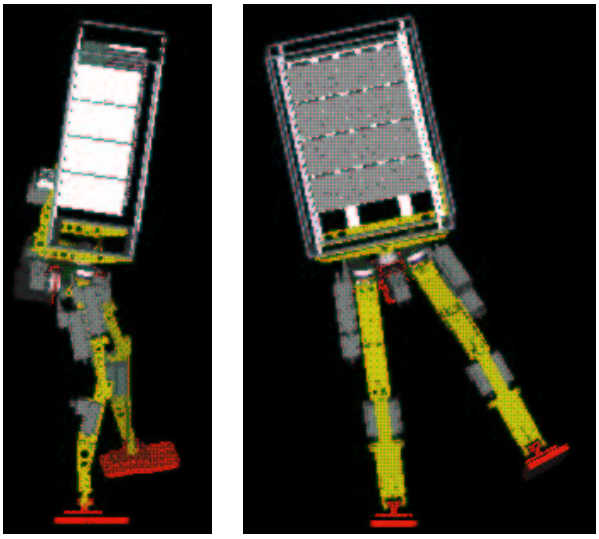


FIG. E.3 – Position p2 : pied grand écart côté

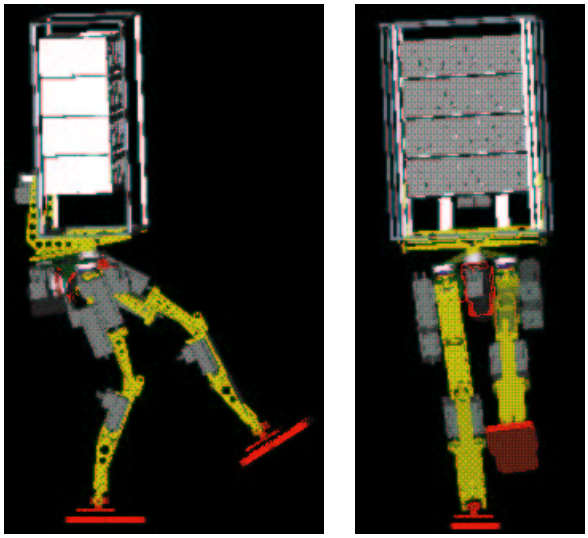


FIG. E.4 – Position p5 : pied en avant

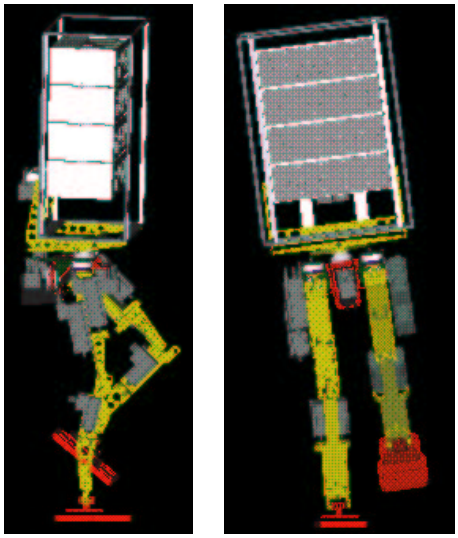


FIG. E.5 – Position p6 : pied flamand rose

Fichier *Datfilter*

```
#!/usr/local/bin/tcsh
setenv INPUT_PATH /home/mouflon/jarde/BIP/orccad/user15/Exec/Sessions/current
setenv OUTPUT_PATH /home/mouflon/jarde/BIP/datas

ls -l $INPUT_PATH | cut -f 15- -d' '

echo ''
date
echo ''

echo 'Extraction qd'
cut -f 3-17 -d ' ' $INPUT_PATH/bmoveTraj_1_JointDes > $OUTPUT_PATH/qd

echo 'Extraction torque'
cut -f 3-17 -d ' ' $INPUT_PATH/bmoveCmd_3_Torque > $OUTPUT_PATH/torque

echo 'Extraction tension'
cut -f 3-17 -d ' ' $INPUT_PATH/bmoveConv_4_Current > $OUTPUT_PATH/tension

#echo 'Extraction gravite'
#cut -f 3-17 -d ' ' $INPUT_PATH/bmoveModel_9_Gravite > $OUTPUT_PATH/gravite

echo 'Extraction state'
cut -f 3 -d ' ' $INPUT_PATH/bmoveSupport_14_State > $OUTPUT_PATH/state

echo 'Extraction feet'
cut -f 3-8 -d ' ' $INPUT_PATH/Sensors15_Feet > $OUTPUT_PATH/feet

#echo 'Extraction Lambda'
#cut -f 3-4 -d ' ' $INPUT_PATH/bmoveSupport_14_Lambda > $OUTPUT_PATH/Lambda

echo 'Extraction q'
cut -f 3-17 -d ' ' $INPUT_PATH/bipJoints15_5_Joints > $OUTPUT_PATH/q

echo 'Extraction error'
cut -f 3-17 -d ' ' $INPUT_PATH/bmoveError_2_Error > $OUTPUT_PATH/error
```

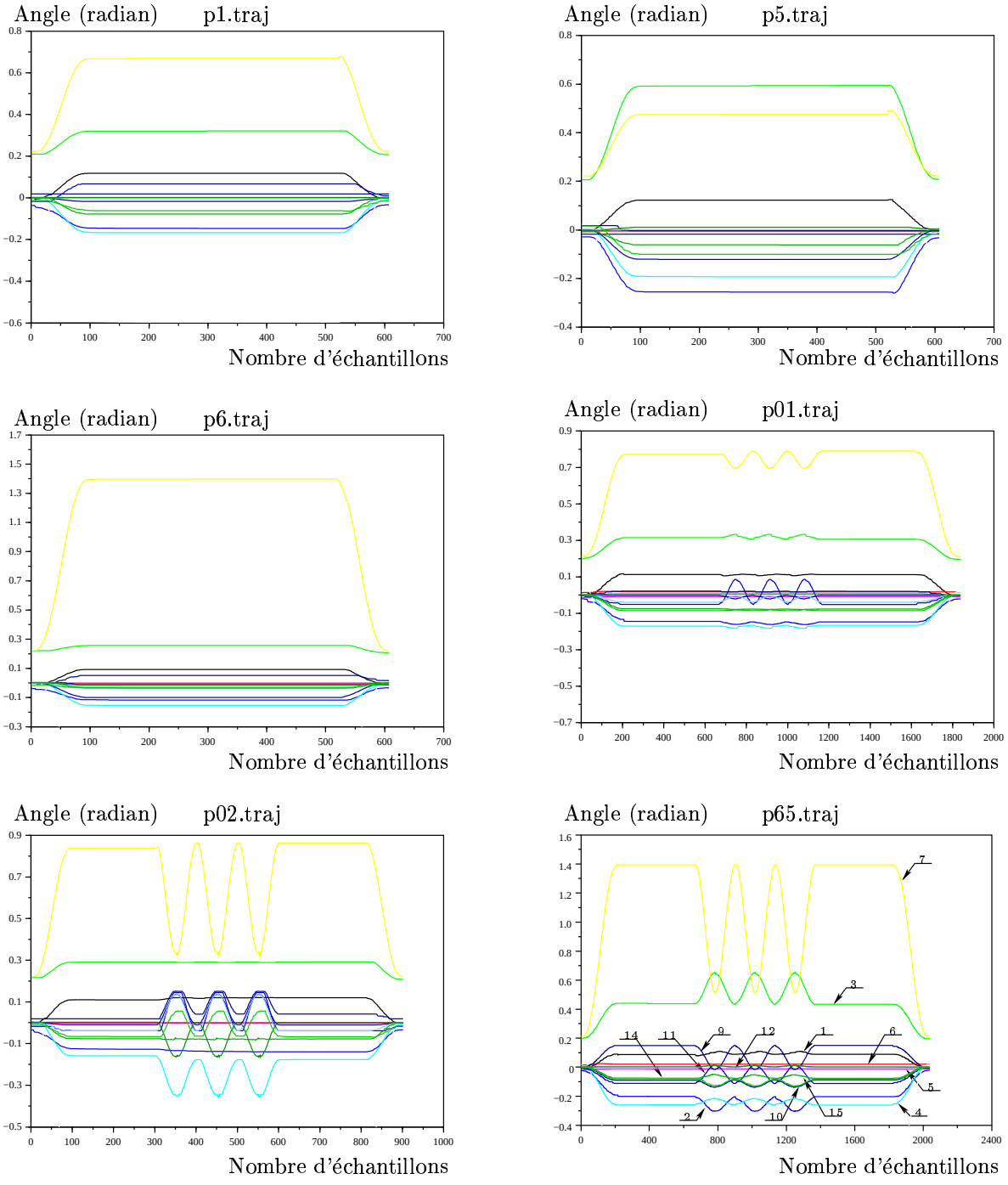


FIG. E.6 – Évolution du vecteur q pour les différents suivis.

Annexe F

Les capteurs de force

Définitions

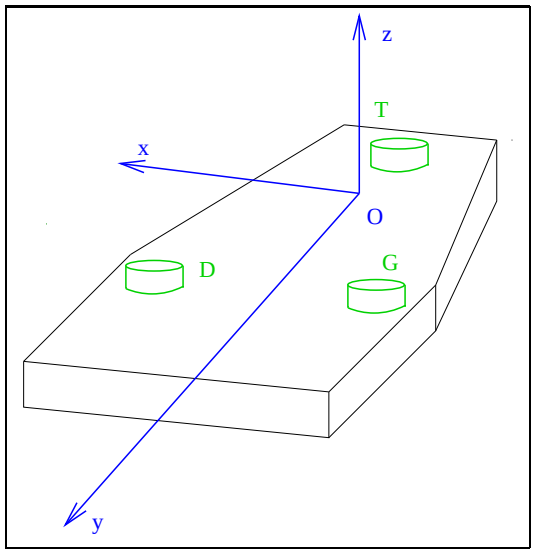


Fig. F.1 – Représentation d'un pied du robot

Ce chapitre reprend les calculs présentés dans le document [16].

Repère R

Le repère $R = (O, x, y, z)$ lié à la semelle du pied (figure F.1) se définit comme suit :
- O : projection de la cheville sur la semelle,
- x : axe latéral, vers la droite,

- y : axe latéral, vers l'avant,
- z : axe normal à la semelle, vers le haut.

Position des capteurs dans R

- Capteur du talon $\vec{OT} = (x_T, y_T, z_T)$
- Capteur gauche $\vec{OG} = (x_G, y_G, z_G)$
- Capteur droit $\vec{OD} = (x_D, y_D, z_D)$

Masse et gravité

- Masse de la semelle : m,
- Accélération de la pesanteur : g,
- Centre de masse M dans le repère R : $\vec{OM} = (x_M, y_M, z_M)$,
- Verticale ascendante exprimée dans le repère R : $\vec{VA} = (VA_x, VA_y, VA_z)$,

Informations des capteurs

- F positive en traction : 2500N pour 5V,
- F négative en compression : -2500N pour -5V,

Bilan des forces exercées sur la semelle

Action des forces de pression

$\vec{F_p} = (0, 0, F_p)$
$\vec{M_p} = (M_{px}, M_{py}, 0)$ exprimé au point O

Action des forces tangentielles de frottement

$\vec{F_t} = (F_{tx}, F_{ty}, 0)$
$\vec{M_t} = (0, 0, M_{tz})$ exprimé au point O

Poids

$\vec{F_M} = (-mgVA_x, -mgVA_y, -mgVA_z)$
$\vec{M_M} = (0, 0, 0)$ exprimé au point M

Actions du tibia sur les capteurs

Les actions du tibia sur les trois capteurs sont les forces et moments suivants :

$$\begin{array}{ll}\vec{F_T} = (F_{Tx}, F_{Ty}, F_T) & \\ \vec{M_T} = (0,0,0) & \text{exprimé au point } T \\ \vec{F_G} = (F_{Gx}, F_{Gy}, F_G) & \\ \vec{M_G} = (0,0,0) & \text{exprimé au point } G \\ \vec{F_D} = (F_{Dx}, F_{Dy}, F_D) & \\ \vec{M_D} = (0,0,0) & \text{exprimé au point } D \end{array}$$

F_{Tx}, F_{Ty}, F_{Gx}, F_{Gy}, F_{Dx} et F_{Dy} non mesurées

Équations d'équilibre

Somme des forces nulle

$$\vec{F_p} + \vec{F_t} + \vec{F_M} + \vec{F_T} + \vec{F_G} + \vec{F_D} = \vec{0} \tag{1}$$

Somme des moments en O nulle

$$\vec{M_p} + \vec{M_t} + \vec{OM}\vec{F_M} + \vec{OT}\vec{F_T} + \vec{OG}\vec{F_G} + \vec{OD}\vec{F_D} = \vec{0} \tag{2}$$

On projette (1) sur l'axe z et (2) sur les axes x et y.

$$F_p - mgVA_z + F_T + F_G + F_D = 0$$

$$\begin{aligned} M_{px} - mg(y_M VA_z - z_M VA_y) + (y_T F_T - z_T F_{Ty}) + \\ (y_G F_G) + (y_D F_D - z_D F_{Dy}) = 0 \end{aligned}$$

$$\begin{aligned} M_{py} - mg(z_M VA_x - x_M VA_z) + (z_T F_{Tx} - x_T F_T) + \\ (z_G F_{Gx} - x_G F_G) + (z_D F_{Dx} - x_D F_D) = 0 \end{aligned}$$

En faisant l'hypothèse que les produits zF_x et zF_y sont négligeables, les actions de pression sont représentées par le torseur suivant exprimé au point O :

$$\begin{aligned} F_p &= mgVA_z - F_T - F_G - F_D \\ M_{px} &= mg(y_M VA_z - z_M VA_y) - y_T F_T - y_G F_G - y_D F_D \\ M_{py} &= mg(z_M VA_x - x_M VA_z) + x_T F_T + x_G F_G + x_D F_D \end{aligned}$$

Centre de pression

Soit $C = (x_C, y_C, 0)$ le centre de pression sur le pied considéré :

$$x_C = -\frac{M_{Py}}{F_p}$$
$$y_C = \frac{M_{Px}}{F_p}$$

Cas particulier de la semelle horizontale

$VA = (0, 0, 1)$,

$$F_p = mg - F_T - F_G - F_D$$
$$M_{px} = mgy_M - y_T F_T - y_G F_G - y_D F_D$$
$$M_{py} = -mgx_M + x_T F_T + x_G F_G + x_D F_D$$

Valeurs numériques

Positions des capteurs

$$x_T = 0$$
$$y_T = -68$$
$$z_T = 32$$
$$x_G = -60$$
$$y_G = 52$$
$$z_G = 32$$
$$x_D = 60$$
$$y_D = 52$$
$$z_D = 32$$

Masse et centre de masse

$$m = 1.9kg$$
$$x_M = 0$$
$$y_M = 32$$
$$z_m = 14$$
$$g = 9.81m.s^{-2}$$

Annexe G

Évaluation des frottements secs

fichier *frottement.c*

```
#include <stdio.h>
#include <math.h>
#include <taskLib.h>

#include "hardBip.h"
#include "drvBip.h"

/*************************************************************/
/* nombre d'essai pour le calcul de frottement sec, la valeur */
/* max sur les moteurs est donc NB_ESSAI * INCR_CONSIGNE      */
/*************************************************************/
#define NB_ESSAI 80 /*20*/

/* temps d'attente au demarrage apres consigne moteur en tick */
#define T_D_MOTEUR 30

/* en tick, temps d'attente a l'arret apres consigne moteur */
#define T_F_MOTEUR 20

/* nombre de pas codeurs pour valider que le moteur a bouge */
#define DELTA_CODEUR 1.5

/* increment sur la consigne de commande */
#define INCR_CONSIGNE 0.03

/* variable gerant l'initialisation */
static int bipTestInitFlag = ERROR;

#define BIP_DOF 15

int bipTestInit()
{
  if (bipTestInitFlag == ERROR)
    {
      bipHardInit();
      bipTestInitFlag = OK;
    }
  return(OK);
}
```

```
static int bipTestFrottement(int sens,int index)
{
  int c, cc;
  float consigne;
  int fin_test = 0;
  double current[BIP_DOF];
  double joint_init[BIP_DOF];
  double joint[BIP_DOF];

  if ((sens != 1) && (sens != -1))
    {
      printf("Pb Argument\n");
      return(-1);
    }

  /* gestion de l'init */
  if (bipTestInitFlag == ERROR)
    {
      bipHardInit();
      bipTestInitFlag = OK;
    }

  /* init */
  printf("init variables\n");
  for(c=0;c<BIP_DOF;c++)
        {
        current[c]=0.0;
        }
  bipPutCurrent(current);
  bipGetEncoders(joint_init);


  printf("Moteur No %d\n", index);
  fin_test = 0;
  /* increment de sortie: jusqu'a ce que ca bouge */
  for (c=1;(c<=NB_ESSAI) && (fin_test==0);c++)
        {
          current[index]=INCR_CONSIGNE*sens*c;
          printf("consigne=%.2f\n", current[index]);
          bipPutCurrent(current);

          /* Demarage du moteur */
          taskDelay(T_D_MOTEUR);

          /* arret moteur, les donnees sont lues */
          current[index]=0.0;
          bipPutCurrent(current);

          /* prise en compte de l'inertie moteur pour l'arret */
          taskDelay(T_F_MOTEUR);

          /* les donnees sont lues */
          bipGetEncoders(joint);
          printf("codeur %d : %f\n", index, joint[index] - joint_init[index]);

          if (fabs(joint[index] - joint_init[index]) > DELTA_CODEUR)
                {
                  fin_test = 1;
                  consigne = INCR_CONSIGNE*sens*c;
                }
          for(cc=0;cc<BIP_DOF;cc++)
                {
```

```
                joint_init[cc] = joint[cc];
              }
        }

  printf("Moteur No %d : fs= %.2f\n",index, consigne);

  return(0);
}


int bipTestFrotPos(int index)
{
  return(bipTestFrottement(1,index));
}


int bipTestFrotNeg(int index)
{
  return(bipTestFrottement(-1,index));
}
```

Moteur	**Tension mesurée** en volts (sens positif)	**Tension mesurée** en volts (sens négatif)
MotCheD_E	0,57	-0,03 sens positif
MotCheD_I	0,48	-0,03 sens positif
MotGenD_S	0,45	-0,21
MothanD_S	0,54	-0,45
MotCheG_I	1,77	-1,59
MotCheG_E	0,57	-0,51
MotGenG_S	0,48	-0,03 sens positif
MothanG_S	0,51	-0,03 sens positif
MothanD_F	0,99	-0,69
MotHanD_V	0,6	-0,45
MotHanG_F	1,05	-0,48
MotHanG_V	0,66	-0,12
MotLomb_V	0,96	-0,66
MotLomb_D	1,32	-1,32
MotLomb_G	1,83	-0,72

TAB. G.1 – Évaluation des frottements.

Bibliographie

[1] C. Azevedo and N. Andreff. Étude expérimentale des premières démarches du robot BIP2000. Rapport de recherche, INRIA, 2000.

[2] C. Azevedo and R. Pissard-Gibollet. Le contrôleur du robot BIP2000. Rapport de recherche, INRIA, 2000.

[3] G. Baille, P. Di Giacomo, H. Mathieu, and R. Pissard-Gibollet. L'armoire de commande du robot bipède BIP2000. Rapport technique, INRIA, 2000.

[4] J.M. Bourgeot. Planification et génération de trajectoires d'un robot bipède en environnement non structuré. Rapport de stage, Université de Versailles, 2001.

[5] V. Bozonnet. Programmation d'un robot bipède. Rapport de stage, Institue des Sciences et Techniques de Grenoble (ISTG), 2001.

[6] B. Brogliato. *Nonsmooth Impact Mechanics.* Springer-Verlag, 1996.

[7] B. El Ali. *Contribution à la Commande du Centre de masse d'un robot bipède.* Thèse de doctorat, INPG, 1999.

[8] B. Espiau and the Bip team. *BIP : A Joint Project for the Development of an Anthropomorphic Biped Robot.* Proc. Int. Conf. on Advanced Robotics, July 1997.

[9] F. Génot. *Contributions à la modélisation et à la commande des systèmes mécaniques de corps rigides avec contraintes unilatérales.* PhD Thesis-INPG, France, 1998.

[10] W. Khalil and J.F. Kleinfinger. *A new geometric notation for open and closed loop robots.* IEEE International Conference on Robotics & Automation, 1986.

[11] F. Lydoire. Le simulateur du robot BIP2000. Rapport technique, INRIA, 2001.

[12] J.J. Parmentier. Contribution à la commande d'un robot bipède. Rapport de stage, Ecole Polytechnique, 1999.

[13] R. Pissard-Gibollet and K. Kapellos. **O**pen **R**obot **C**ontoller **C**omputer **A**ided **D**esign. orccad version 3.0β. User manual, INRIA, December 1998.

[14] J.M. Rifflet. *La communication sous Unix, Applications Réparties.* Ediscience International, Paris, 1996.

[15] C. Samson, M. Le Borgne, and B. Espiau. *Robot Control : The Task Function Approach.* Oxford Science Publications, 1991.

[16] P. Sardain. Paramètres de BIP2000. Rapport technique, Laboratoire de Mécanique des Solides (LMS) de Poitier, 2000.

[17] The Orccad Team. *An Integrated and Modular Approach for the Specification, the Validation and the Implementation of Complex Robotics Missions*, volume 17-(4). International Journal of Robotics Research, April 1998. Special Issue on Integreated Architectures for Robot Control and Programming.

[18] P.B. Wieber. *Modélisation et commande d'un robot marcheur anthropomorphe*. Thèse de doctorat, Ecole des Mines de Paris, 2000.

[19] WindRiver Systems. *VxWorks 5.3.1 Programmer's Guide*, 1 edition, Mar 1997.

[20] WindRiver Systems. *VxWorks 5.3.1 Reference Manual*, 1 edition, Feb 1997.

[21] http ://www.inrialpes.fr/iramr/bip/ : site public sur la réalisation du bipède.

[22] http ://www-lms.univ-poitiers.fr/robot/ : site public du LMS de Poitiers.

[23] http ://www.inrialpes.fr/iramr/ : sevice robotique de l'INRIA Rhônes-Alpes.

[24] http ://www.inrialpes.fr/iramr/Orccad/ : le logiciel ORCCAD.

[25] http ://www.mcg.mot.com : carte MVME162-522A de motorola.

[26] http ://www.wrs.com/products/html/vxworks.html : Real Time Operating System de WindRiver Systems.