Master Systèmes Dynamiques et Signaux

Mémoire

---

# Génération automatique des équations du modèle dynamique d'un robot polyarticulaire

---

*Auteur :*
M. Quentin Canovas

*Jury :*
Pr. Laurent Hardouin
Dr. Nicolas Delanoue
Pr. Jean-Baptiste Fasquel
Pr. David Rousseau

# Remerciements

Je remercie tout d'abord Pr. Laurent Hardouin qui a su m'accepter dans le cursus Master Systèmes Dynamiques et Signaux.

Je remercie Dr. Nicolas Delanoue et Dr. Rémy Guyonneau qui ont été mes tuteurs du master et de ce stage, qui ont su m'orienter et m'aider pendant toutes les heures de projet.

Je remercie aussi M. Nicolas Testard et M. Franck Mercier qui m'ont donné leur résultat qui m'a permis de tester mes propres résultats.

# Table des matières


**1 Introduction** **1**

**2 Les méthodes de Lagrange et de Newton-Euler sur un robot2R** **5**

2.1 La méthode de Lagrange . . . . . . . . . . 6

2.1.1 Le principe . . . . . . . . . . 6

2.1.2 Robot2R . . . . . . . . . . 7

2.2 La méthode Newton-Euler . . . . . . . . . . 10

2.2.1 Le principe . . . . . . . . . . 10

2.2.2 Robot2R . . . . . . . . . . 11

**3 URDF2DynamicModel : Générateur des équations du modèle dynamique d'un robot polyarticulaire** **17**

3.1 Présentation de URDF2DynamicModel . . . . . . . . . . 17

3.2 Qu'est-ce l'URDF ? . . . . . . . . . . 18

3.3 Explication du code . . . . . . . . . . 24

**4 Résultats** **31**

4.1 Robot2R . . . . . . . . . . 31

4.2 Structure humanoïde : robot 3P33R . . . . . . . . . . 35

**5 Conclusion** **39**

**A Diagramme UML de URDF2DynamicModel** **41**

# Table des figures

1.1 Diagramme vu générale du fonctionnement du même que URDF2DynamicModel 1
1.2 Schéma cinématique du robot 2R . . . . . . . . . . . . . . . . . . . . . . . . . 3
1.3 Diagramme de l'obtention des forces et des couples via URDF2DynamicModel . 3

2.1 Position du repère monde . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
2.2 Robot2R et le repère monde . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
2.3 Représentation de chaque repère . . . . . . . . . . . . . . . . . . . . . . . . 12
2.4 Représentation des composantes de l'équation (2.33) . . . . . . . . . . . . . . . 14
2.5 Efforts appliqués au robot 2R . . . . . . . . . . . . . . . . . . . . . . . . . . 15

3.1 Diagramme UML de URDF2DynamicModel . . . . . . . . . . . . . . . . . . . . . 18
3.2 Repère des articulations dans un fichier URDF . . . . . . . . . . . . . . . . . . 21
3.3 Schéma du robot 2R avec les nominations des éléments . . . . . . . . . . . . . . 22
3.4 Représentation des repères des segments sur le robot 2R . . . . . . . . . . . . . 23

4.1 Schéma du corps avec la positon des segments et des articulations . . . . . . . . 36
4.2 Position en $y$ de l'articulation LLJ . . . . . . . . . . . . . . . . . . . . . . . . 37
4.3 Position en $y$ de l'articulation LNJ . . . . . . . . . . . . . . . . . . . . . . . . 38
4.4 Position en $y$ de l'articulation ULJ . . . . . . . . . . . . . . . . . . . . . . . . 38

# List of acronyms

| | |
|---|---|
| **URDF** | *Unified Robot Description Format* |
| **NE** | *Newton Euler* |
| **DoF** | *Degrees of Freedom* |
| **ddl** | Degrées de liberté |
| **CPU** | *Central Processing Unit* |
| **ROS** | *Robot Operating System* |
| **XML** | *Extensible Markup Language* |
| **HuMoD** | *Human Motion Dynamics* |

# Chapitre 1

# Introduction

Le contrôle et la simulation d'un robot requièrent plusieurs modèles différents. Certains objectifs n'ont pas forcément besoin de modèles très évolués mais dès qu'on s'intéresse à déplacer un robot, le besoin d'avoir un modèle dynamique est présent. Ce document présente mes recherches, mes travaux autour de la modélisation dynamique et de la création d'un outil. Le but de ce stage de master est de pouvoir proposer un outil qui permet d'obtenir *automatiquement* un modèle dynamique d'un manipulateur en chaîne ouverte ayant potentiellement plusieurs bras. Le nom de l'outil est URDF2DynamicModel.

L'origine de URDF2DynamicModel vient du fait que peu d'outils permettant d'obtenir un modèle sont disponibles. Et ceux qui le sont comme des "boîtes noires" où seules les entrées et sorties sont accessibles. Le modèle dynamique utilisé reste hors d'accès de l'utilisateur qui peut être problématique. Surtout si la comparaison des résultats entre des calculs fait à la main et par l'outil montre des différences. Savoir quels résultats sont les plus pertinents devient plus compliqué.

Toutefois, on peut noter l'existence du logiciel **Symoro**+ [3], programme créé au sein du laboratoire des Sciences du Numérique de Nantes, **LS2N**, supervisé par le Pr. Khalil. Cette application est *open-source* et crée le modéle dynamique d'un robot à partir de sa description géométrique et mécanique via les paramètres de Denavit-Hartenberg (modifiés). Malheureuseusement, le logiciel bloque la personnalisation du robot au niveau de la structure.

On peut aussi retrouver SD/Fast [5], ROBOTRAN [6], METAPOD [7], RobCoGen [4]. Tous ces outils fonctionnent a peu près pareil, on leur donne en entrée le robot, la structure et l'outil donne son modèle dynamique. C'est pourquoi l'outil voulu à la fin doit être simple d'utilisation, et accessible sans trop de connaissances en robotique. Le code doit être aussi

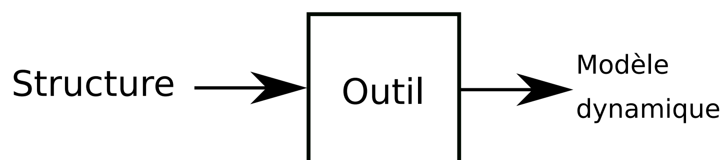



FIGURE 1.1 – Diagramme vu générale du fonctionnement du même que URDF2DynamicModel

accessible à l'utilisateur et qu'il puisse le modifier.

Le projet, avant le stage, était d'apprendre, comprendre les méthodes de génération des modèles dynamiques pour un robot poly-articulé en chaîne ouverte. L'objectif du stage était d'obtenir un outil qui permet de générer les modèles dynamiques voulus dans un langage de programmation : le Python. L'évolution de URDF2DynamicModel serait de proposer les scripts de modèles dynamique sous différent langage comme le C ou même MatLab. L'outil doit pouvoir générer un robot simple comme à 2 articulations jusqu'à un "robot" complexe comme à 36 articulations, comme le corps humain 4.

Ce document va rappeler deux méthodes de modélisations les plus utilisées.

**La méthode de Lagrange.** Les notions mathématiques sont simples d'accès. Elle s'appuie sur l'énergie cinétique et l'énergie potentielle de chaque segment. Finalement, on obtient les couples et les froces pour chaque actionneur. Cette méthode a un désavantage : son temps de calcul. Dès que le nombre de **DoF** du robot dépasse 4, les temps de calcul augmentent d'une façon trop importante.

**La méthode de Newton-Euler.** Elle est une alternative à la méthode précédente car son temps de calcul est plus petit. La méthode est un algorithme récursif. Ceci permet de gagner du temps **CPU** et mémoire.

La modélisation dynamique d'un robot permet de pouvoir contrôler son déplacement. Elle est traduit par des équations qui permettent d'obtenir les couples et forces pour chaque variable articulaire $q_i$. On retrouve dans ces équations les composantes des segments ainsi que les forces qui leur sont appliquées.
L'expression générale du couple, comme le montre M. Khalil [2] :

$$\Gamma = f(q, \dot{q}, \ddot{q}, \mathcal{F}) \tag{1.1}$$

- $\Gamma$ : couple appliqué au segment,
- $q$ : vecteur position de l'articulation,
- $\dot{q}$ : vecteur vitesse de l'articulation,
- $\ddot{q}$ : vecteur accélération de l'articulation,
- $\mathcal{F}$ : vecteur des forces et des moments appliqués au segment dû à l'environnement du robot.

Pour expliciter ces méthodes, on va s'appuyer sur un robot 2R (figure 1.2). Il possède 2 articulations pivots en série. L'espace de travail du robot est le plan $xOy$ car les pivots ont leur axes de mouvements parallèles à l'axe $z$. La position du repère sera indiqué dans la suite du document.

Pour rappel et classiquement, les articulations utilisées en robotique sont à un **ddl** et sont divisées en 2 catégories :

- **Rotoïde**, l'articulation est une rotation sur un seul axe.
- **Prismatique**, l'articulation est une translation sur un seul axe.

Donc les articulations qui ont plusieurs *ddl* seront décomposées. Exemple, le pivot glissant possédant une translation et une rotation est décrit comme étant une articulation prismatique et une articulation rotoïde.

L'outil est expliqué dans le 3$^{\text{ème}}$ chapitre du document et met en relation la méthode de

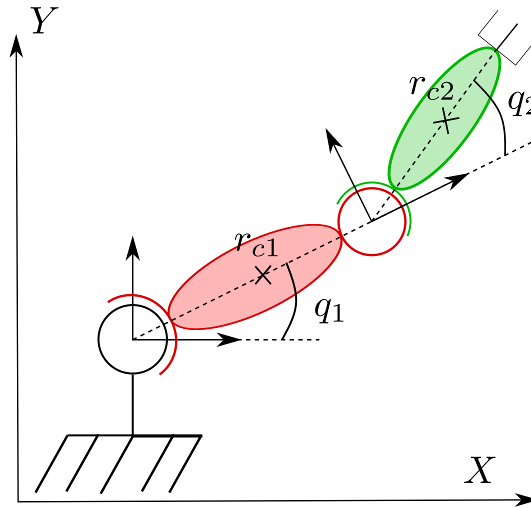



FIGURE 1.2 – Schéma cinématique du robot 2R

Newton-Euler avec l'automatisation des créations des équations. Dans cette partie, on explique aussi la structure d'un fichier URDF et les règles pour décrire le robot dans ce type de format. Le code est explicité mais il n'est pas dans le document car la taille est trop imposante.

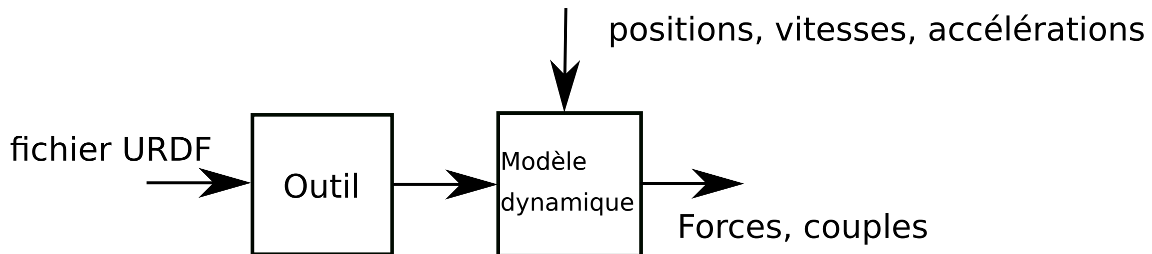



FIGURE 1.3 – Diagramme de l'obtention des forces et des couples via URDF2DynamicModel

Avant la conclusion du document, on présente 2 types de résultats :
- La retranscription de la méthode Newton-Euler d'un robot 2R en code grâce à l'outil
- La comparaison des résultats des équations d'un modèle dynamique d'un corps humain et d'une base de données [9] réalisées par une équipe de l'université de technologie de Darmstadt.

Le site **HuMoD** [9], *Human Motion Dynamics*, présente les données mesurées pendant le déplacement à plusieurs vitesses, d'un homme et d'une femme. Les données sont multiples et présentées dans la suite du document 4.2. Le site propose les positions, vitesses et accélérations de chaque articulation ainsi que leurs coordonnées. On peut alors tester le générateur de modèles pour voir s'il fonctionne et tester sa vitesse d'exécution sur une structure complexe.

# Chapitre 2

# Les méthodes de Lagrange et de Newton-Euler sur un robot2R

La modélisation dynamique a 2 formes, il y a :

**le modèle dynamique direct** : il permet de calculer les accélérations des segments et des articulations grâce aux positions, aux vitesses et aux efforts données de ces éléments,

**le modèle dynamique inverse** : celui-ci permet de connaître les efforts des articulations et des segments grâce aux positions, aux vitesses et aux accélérations données.

Dans notre cas, seul le modèle dynamique inverse est utilisé. Pour obtenir ces équations, on émet quelques hypothèses simplificatrices :

— les corps sont des éléments solides rigides, leur déformation n'est pas prise en compte dans les équations,

— les articulations sont des pivots ou des glissières (articulation rotoïde ou prismatique),

— les manipulateurs sont des chaîne ouvertes. La structure ne doit pas posséder de boucles mécaniques.

Le modèle dynamique inverse qui va être calculé peut être représenté par l'équation (2.1) :

$$\Gamma(t) = H(q(t))\ddot{q}(t) + C(q(t), \dot{q}(t)) \tag{2.1}$$

Nous retrouvons l'accélération multipliée par la matrice d'inertie $H$, le tout additionné avec un vecteur force $C$ qui contient la force de Coriolis, de centrifuge, la force gravitationnelle et d'autres forces. C'est encore une autre formulation de la 2$^{\text{ème}}$ loi de Newton. Cette formulation est une équation différentielle car elle lie les fonctions $t \to \Gamma(t)$, $t \to q(t)$ ainsi que ces dérivées. Et cela peut nous permettre d'avoir les positions, les vitesses et les accélérations avec le modèle inverse grâce à des méthode comme celle de *Euler*, *Runge-Kutta* ou d'autres. Ces méthodes peuvent servir à tester les résultats du modèle dynamique inverse. Si les efforts calculés sont correctes, alors le résultats de ces méthodes devraient être égaux aux entrées du modèle dynamique inverse.

Dans cette partie, le modèle dynamique inverse est explicité sur un robot 2R. Afi d'obtenir ce modèle, les approches les plus souvent utilisés qui sont la méthode de *Lagrange* et la méthode de *Newton-Euler*. Les notations utilisées dans les équations sont :

— ${}^{i}r_{cj}$ : vecteur position du centre de masse du segment $j$ par rapport au référentiel $R_i$,

— ${}^{i}r_j$ : vecteur position de l'extrémité du segment $j$, opposée au repère $R_j$, par rapport au référentiel $R_i$,
— ${}^{0}V_{cj}$ : vecteur vitesse du centre de masse du segment $j$ par rapport à $R_0$,
— ${}^{0}V_j$ : vecteur vitesse de l'extrémité du segment $j$, opposée au repère $R_j$, par rapport au référentiel $R_0$,
— ${}^{i}\omega_j$ : vecteur vitesse angulaire du segment $j$ dans le référentiel $R_i$,
— ${}^{0}\dot{V}_{cj}$ : vecteur accélération du centre de masse du segment $j$ par rapport à $R_0$,
— ${}^{0}\dot{V}_j$ : vecteur accélération de l'extrémité du segment $j$, opposée au repère $R_j$, par rapport au référentiel $R_0$,
— ${}^{i}\dot{\omega}_j$ : vecteur accélération angulaire du segment $j$ dans le référentiel $R_i$,
— ${}^{i}Rot_j$ : matrice rotation du repère $R_i$ au repère $R_j$
— $\Gamma_j$ : force appliquée par l'articulation $j$,
— $q_j$, $\dot{q}_j$, $\ddot{q}_j$ : variables généralisées,
— $m_j$ : masse du segment $j$,
— $I_j$ : matrice d'inertie du segment $j$,
— $g$ : vecteur d'accélération de la gravité.

Le repère monde est positionné au niveau du support du robot2R. Pour simplifier les calculs,

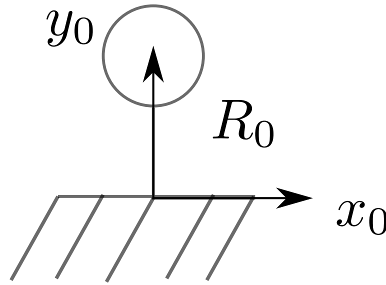



FIGURE 2.1 – Position du repère monde

la 1ère articulation est positionnée aux cordonnées $[0\,0\,0]^T$.

## 2.1 La méthode de Lagrange

### 2.1.1 Le principe

Cette méthode est relativement simple pour obtenir le modèle dynamique. Son principale défaut est la complexité algorithmique. En effet le nombre d'opérations, en calcul formel, peut être très important. C'est pourquoi ce formalisme est préféré pour de petite architecture.

Formellement, cette approche ne dépend que des 2 équations suivantes :

$$\Gamma_1 = \frac{d}{dt}\frac{\partial \mathcal{L}}{\partial \dot{q}_j} - \frac{\partial \mathcal{L}}{\partial q_j} \tag{2.2}$$

$$\mathcal{L} = Ec - Ep \tag{2.3}$$

— $\mathcal{L}$ : est l'équation de lagrange,

— $Ec$ : énergie cinétique du robot,
— $Ep$ : énergie potentielle.

Le procédé est de calculer la position et la vitesse de chaque élèment. Puis on utilise l'équation lagrangienne (2.3) pour après effectuer les dernières opérations pour résoudre l'équation (2.2). Les équations potentielles et cinétiques utilisées sont :

$$Ec_j = \frac{1}{2} \cdot (\omega_j^T \cdot I_j \cdot \omega_j + m_j \cdot {}^0V_j^T \cdot {}^0V_j) \tag{2.4}$$

$$Ep_j = m_j \cdot g \cdot {}^0Vy_j \tag{2.5}$$

### 2.1.2 Robot2R

Pour rappel : la structure du robot et la position du repère monde (repère 0) :

FIGURE 2.2 – Robot2R et le repère monde

Tout d'abord, il faut trouver les équations des positions et des vitesses de chaque élément voulu. Ici, on les calcule pour les centres de gravité. On va calculer aussi les énergies cinétiques et potentielles individuellement. Finalement, on fait la sommes de ces énergies pour $N$ segments :

$$\begin{aligned} Ec &= \sum_{j=1}^{N} Ec_j \\ Ep &= \sum_{j=1}^{N} Ep_j \end{aligned} \tag{2.6}$$

En robotique, pour la modélisation d'un système, on peut approximer un segment par une ellipsoïde ce qui permet de simplifier la matrice d'inertie car $I_{xy} = I_{yx} = I_{zx} = I_{xz} = I_{zy} = I_{yz} = 0$ pour cette forme. Dans notre exemple, les corps sont homogènes donc les centres de gravité sont des segments. Les constantes pour cet exemple sont les suivantes :

— $l_j$ : longueur du segment $j$,
— $A_j$ : matrice d'inertie du segment $j$,
— $m_j$ : la masse du segment $j$.

**Segment 1**

Voici les composantes de position pour le segment 1 :

$$^0r_{c1} = \begin{bmatrix} \frac{l_1}{2} \cdot \cos(q_1) \\ \frac{l_1}{2} \cdot \sin(q_1) \\ 0 \end{bmatrix} \tag{2.7}$$

Pour calculer la vitesse, nous pouvons utiliser la matrice jacobienne :

$$\begin{aligned} ^0V_{cj} &= \mathcal{J}(r_{cj}) \cdot [\dot{(q)}_1, ..., \dot{(q)}_j]^T \\ ^0V_{c1} &= \mathcal{J}(r_{c1}) \cdot [\dot{(q)}_1]^T \end{aligned} \tag{2.8}$$

Ce qui donne :

$$^0V_{cj} = \begin{bmatrix} -\dot{q}_1 \cdot \frac{l_1}{2} \cdot \sin(q_1) \\ \dot{q}_1 \cdot \frac{l_1}{2} \cdot \cos(q_1) \\ 0 \end{bmatrix} \tag{2.9}$$

Ici, la vitesse angulaire est $\omega_1 = [0\,0\,q_1]^T$ et la matrice d'inertie est une matrice diagonale. Avec ces résultats nous pouvons calculer l'énergie cinétique et potentielle du segment 1 :

$$\begin{aligned} Ec_1 &= \frac{1}{2}([0\,0\,(q_1 Izz_1)] \cdot [0\,0\,q_1]^T + m_1 \cdot (-\dot{q}_1 \cdot \frac{l_1}{2} \cdot \sin(q_1))^2 + (\dot{q}_1 \cdot \frac{l_1}{2} \cdot \cos(q_1))^2) \\ Ec_1 &= \frac{1}{2}(q_1^2 \cdot Izz_1 + m_1 \cdot l_1^2 \cdot \dot{q}_1^2 \cdot (\sin(q_1)^2 + \cos(q_1))^2) \\ Ec_1 &= \frac{1}{2}(q_1^2 \cdot Izz_1 + m_1 \cdot l_1^2 \cdot \dot{q}_1^2) \end{aligned} \tag{2.10}$$

$$Ep_1 = m_j \cdot g \cdot \dot{q}_1 \cdot \frac{l1}{2} \cdot \cos(q_1) \tag{2.11}$$

Voici la 1$^{\text{ère}}$ étape à propos de la procédure de Lagrange. Il faut faire cela avec tous les corps où il y a besoin de connaître les efforts.

**Segment 2**

Voici les composantes de position pour le segment 2 et son vecteur vitesse grâce à la matrice jacobienne :

$$^0r_{c1} = \begin{bmatrix} l_1 \cdot \cos(q_1) + \frac{l_2}{2} \cdot \cos(q_2 + q_1) \\ l_1 \cdot \sin(q_1) + \frac{l_2}{2} \cdot \sin(q_1 + q_2) \\ 0 \end{bmatrix} \tag{2.12}$$

$$\mathcal{J}(r_{c2}) = \begin{bmatrix} \frac{\partial rx_{c2}}{\partial q_1} & \frac{\partial rx_{c2}}{\partial q_2} \\ \frac{\partial ry_{c2}}{\partial q_1} & \frac{\partial ry_{c2}}{\partial q_2} \\ 0 & 0 \end{bmatrix} \tag{2.13}$$

$$\begin{aligned} ^0V_{c2} &= \mathcal{J}(r_{c2}) \cdot [\dot{q}_1, \dot{q}_2]^T \\ ^0V_{c2} &= \begin{bmatrix} \dot{q}_1 \cdot (-l_1 \cdot \sin(q_1) - \frac{l_2}{2} \cdot \sin(q_1 + q_2)) - \frac{l_2}{2} \cdot \sin(q_1 + q_2) \cdot \dot{q}_2 \\ \dot{q}_1 \cdot (l_1 \cdot \cos(q_1) + \frac{l_2}{2} \cdot \cos(q_1 + q_2)) + \frac{l_2}{2} \cdot \cos(q_1 + q_2) \cdot \dot{q}_2 \end{bmatrix} \end{aligned} \tag{2.14}$$

Enfin l'énergie potentielle et l'énergie cinétique qui n'est pas écrite avec les valeurs précédentes car elle est trop imposante :

$$Ep_2 = m_2 \cdot g \cdot {}^0Vy_{c2} \tag{2.15}$$

$$Ec_2 = \frac{1}{2} \cdot (\omega_2^T \cdot I_2 \cdot \omega_2 + m_2 \cdot {}^0V_2^T \cdot {}^0V_2) \quad (2.16)$$

avec $\omega_2 = [0\,0\,(q_1 + q_2)]^T$

L'équation de l'énergie cinétique est imposante dès le 2ème segment. Même si trouver cette équation par un ordinateur prendra très peu de temps, il faut penser quand il faudra trouver l'énergie cinétique pour le 8ème segment. Cette équation sera très grande et donc beaucoup d'octets à manipuler et à garder en mémoire. Et pour un outil devant trouver le modèle dynamique pour des structures complexes, prendre du temps juste pour l'énergie cinétique, qui n'est pas la dernière étape, ce n'est pas la meilleure chose. On se rend compte que cette méthode n'est pas adéquate pour le but final.

### L'équation de Lagrange

Maintenant, il faut rassembler toutes les énergies et enfin calculer les efforts de chaque segment. Les équations suivantes ne sont pas décrites complètement, seules des variables globales sont écrites pour garder un minimum de lisibilité. Les opérations compliquées sont décrites jusqu'à un certain point.

$$\mathcal{L} = Ec_1 + Ec_2 - (Ep_1 + Ep_2) \quad (2.17)$$

Pour obtenir les couples de chaque segment, on doit dériver partiellement l'équation (2.17) par leur variable généralisée propre.

$$\begin{aligned} \Gamma_1 &= \frac{d}{dt}\frac{\partial \mathcal{L}}{\partial \dot{q}_1} - \frac{\partial \mathcal{L}}{\partial q_1} \\ \Gamma_2 &= \frac{d}{dt}\frac{\partial \mathcal{L}}{\partial \dot{q}_2} - \frac{\partial \mathcal{L}}{\partial q_2} \end{aligned} \quad (2.18)$$

On peut remplacer par :

$$\begin{aligned} \Gamma_1 &= \mathcal{J}(\frac{\partial \mathcal{L}}{\partial \dot{q}_1}) \cdot [\dot{q}_1, \dot{q}_2, \ddot{q}_1, \ddot{q}_2]^T - \frac{\partial \mathcal{L}}{\partial q_1} \\ \Gamma_2 &= \mathcal{J}(\frac{\partial \mathcal{L}}{\partial \dot{q}_2}) \cdot [\dot{q}_1, \dot{q}_2, \ddot{q}_1, \ddot{q}_2]^T - \frac{\partial \mathcal{L}}{\partial q_2} \end{aligned} \quad (2.19)$$

Et nous obtenons finalement :

$$\begin{aligned} \Gamma_1 = {} & (l_1 \cdot l_2 \cdot m_2 \cdot \cos(q_2) + l_2^2 \cdot m_2) \cdot \ddot{q}_2 - l_1 \cdot l_2 \cdot m_2 \cdot \sin(q_2) \cdot \dot{q}_2^2 \\ & -2 \cdot l_1 \cdot l_2 \cdot m_2 \cdot \dot{q}_1 \cdot \sin(q_2) \cdot \dot{q}_2 + g \cdot l_2 \cdot m_2 \cdot \cos(q_2 + q_1) \\ & +2 \cdot l_1 \cdot l_2 \cdot m_2 \cdot \ddot{q}_1 \cdot \cos(q_2) - Izz_2 \cdot q_2 + ((l_2^2 + l_1^2) \cdot m_2 + l_1^2 \cdot m_1) \cdot \ddot{q}_1 \\ & +(g \cdot l_1 \cdot m_2 + g \cdot l_1 \cdot m_1) \cdot \cos(q_1) + (-Izz_2 - Izz_1) \cdot q_1 \\ \Gamma_2 = {} & l_2^2 \cdot m_2 \cdot \ddot{q}_2 + g \cdot l_2 \cdot m_2 \cdot cos(q_2 + q_1) + l_1 \cdot l_2 \cdot m_2 \cdot \dot{q}_1^2 \cdot sin(q_2) \\ & +l_1 \cdot l_2 \cdot m_2 \cdot \ddot{q}_1 \cdot cos(q_2) - Izz_2 \cdot q_2 + l_2^2 \cdot m_2 \cdot \ddot{q}_1 - Izz_2 \cdot q_1 \end{aligned} \quad (2.20)$$

Au premier abord, le nombre d'opérations peut être perçu comme étant faible alors que c'est le contraire dû aux dérivations. La dérivation est une multitude d'opérations augmentant aux nombres de variables généralisées. Le nombre d'opérations augmente d'une façon très importante quand la structure obtient une articulation en plus. Et la dérivation se trouve dans plusieurs étapes telles que la recherche de la vitesse par exemple.

En second temps, il y a aussi un problème de mémoire. Si les équations doivent être trouvées par ordinateur, alors celui-ci doit stocker les équations formelles et les retravailler quand on doit leur appliquer des dérivations. Comme les équations deviennent très imposantes dès lors que la structure possède 4 articulations, la mémoire arrive vite à être remplie. Ajouter au fait de nombreuses opérations à faire dessus, le temps de calcul devient trop long.

Donc pour l'outil, ce temps pour trouver les équations est trop long et l'utilisateur doit pas attendre plusieurs minutes pour avoir ses résultats voulus. Le test a été fait pour trouver les équations pour un structure à 4 articulations et le temps dépassait déjà les 4 secondes. C'est pourquoi le formalisme de Newton-Euler sera préféré.

## 2.2 La méthode Newton-Euler

### 2.2.1 Le principe

La méthode de Newton-Euler est un algorithme de calcul qui permet d'obtenir le modèle dynamique d'un robot polyarticulé sans boucle mécanique. Cette méthode créée par Luh. Walker et Paul est considérée comme une avancé importante vers la possibilité d'évaluer en ligne le modèle dynamique des robots. Elle utilise les équations (2.21) , ci-dessous, et est fondée sur une double récurrence. La récurrence avant, de la base du robot vers l'effecteur, calcule successivement les vitesses et accélérations des corps, puis les torseurs dynamiques. Une récurrence arrière, de l'effecteur vers la base, permet le calcul des efforts en exprimant pour chaque corps le bilan des forces.

L'avantage de cette méthode pour notre objectif sont que les équations sont "simples" à trouver, dans le sens, ce n'est pas dérivation. Aucune dérivation formelle est requise. Ce sont des séquences logiques qui sont mises bout à bout et qui donnent finalement le modèle dynamique. De plus au contraire du formalisme de Lagrange, on insère directement les valeurs numérique dès le début. On ne garde pas en mémoire les équations formelles. Cela réduit grandement la quantité d'informations stockées dans la mémoire.

Le modèle dynamique peut être représenté par les équations généralisées suivantes (Khalil [2]) :

$$\begin{aligned} F_j &= M_j \cdot \dot{V}_{G_j} \\ M_{G_j} &= I_{G_j} \cdot \dot{\omega}_j + \omega_j \wedge (I_{G_j} \cdot \omega_j) \end{aligned} \tag{2.21}$$

Ces équations expriment le torseur dynamique en $G_j$ des efforts extérieurs sur segment $j$. Voici les éléments qui sont présents dans ces équations :

- $F_j$ : force extérieur exercée sur le segment $j$,
- $G_j$ : centre de masse du segment $j$,
- $\dot{V}_{G_j}$ : accélération du centre de gravité du segment $j$,
- $M_{G_j}$ : moment des efforts extérieurs exercés sur le segment $j$ autour de $G_j$,
- $I_{G_j}$ : matrice d'inertie du segment $j$ exprimée dans le repère $R_j$ et d'origine $G_j$,
- $\omega_j$ : vitesse de rotation du segment $j$,
- $\dot{\omega}_j$ : accélération de rotation du segment $j$.

L'intérêt de calculer explicitement le moment des efforts extérieurs est d'avoir ce paramètre

pour configurer un actionneur qui permet une articulation rotoïde. L'algorithme de Newton-Euler fondé sur la double récurrence s'effectue en 3 étapes :

1. Choisir quels référentiels utilisés et les caractérisés.
2. Calculer $\omega_j$, $\dot{\omega}_j$ et $\dot{V}_{G_j}$ en parcourant la structure du robot de la base à l'effecteur. C'est la récurrence avant, en anglais *Forward.*
3. Calculer $F_j$ et $M_{G_j}$ en faisant le parcours inverse par rapport à l'étape précédente. C'est la récurrence arrière, en anglais *Backard.*

Nous allons représenter chaque vecteur position dans son propre repère et le repère $R_0$. Les vecteurs vitesses et accélérations vont être représentés dans le repère $R_0$ directement. Les équations qui sont dans la suite du document se reposent sur les suivantes (2.22) :

$$\begin{array}{lcl} {}^i r_j & = & {}^i r_{j-1} + {}^i Rot_j \cdot {}^j r_j \\ {}^i V_j & = & {}^i V_{j-1} + {}^i \omega_j \wedge {}^i r_j \\ {}^i \dot{V}_j & = & {}^i \dot{V}_{j-1} + {}^i \dot{\omega}_j \wedge {}^i r_j + {}^i \omega_i \wedge {}^i V_j \end{array} \tag{2.22}$$

Ces équations sont générales pour les articulations de type révolutive.

### 2.2.2 Robot2R

#### Choix de référentiel

Le choix du référentiel est crucial et influence les calculs qui vont être utilisés. Le choix le plus approprié pour placer l'origine du référentiel est son articulation. Cela permet de simplifier la représentation de sa position. Il faut toujours faire attention à son orientation. La convention de *Denavit-Hartenberg modifiée* est pratique dans le sens où elle permet de dire que l'axe $z$ est toujours l'axe où s'applique le mouvement, axe de translation ou de rotation. Mais ce n'est pas obligatoire.

Dans notre cas, nous allons choisir 3 repères :

1. un référentiel monde $R_0$ positionné au support. On suppose que la distance entre la première articulation et le support est nulle, pour une simplification des calculs,
2. un référentiel $R_1$ placé sur l'articulation 1. Son orientation varie avec l'angle $q_1$.
3. un référentiel $R_2$ placé sur l'articulation 2. Son orientation varie avec les angles $q_1$ et $q_2$

Voici la représentation des repères sur le schéma suivant (2.3) :

### Récurrence avant : *Forward*

#### Segment 1

Les coordonnées dans le repère $R_1$ de ${}^1 r_{c1}$ et de ${}^1 r_1$ :

$${}^1 r_{c1} = \begin{bmatrix} l_1/2 \\ 0 \\ 0 \end{bmatrix}, \; {}^1 r_1 = \begin{bmatrix} l_1 \\ 0 \\ 0 \end{bmatrix} \tag{2.23}$$

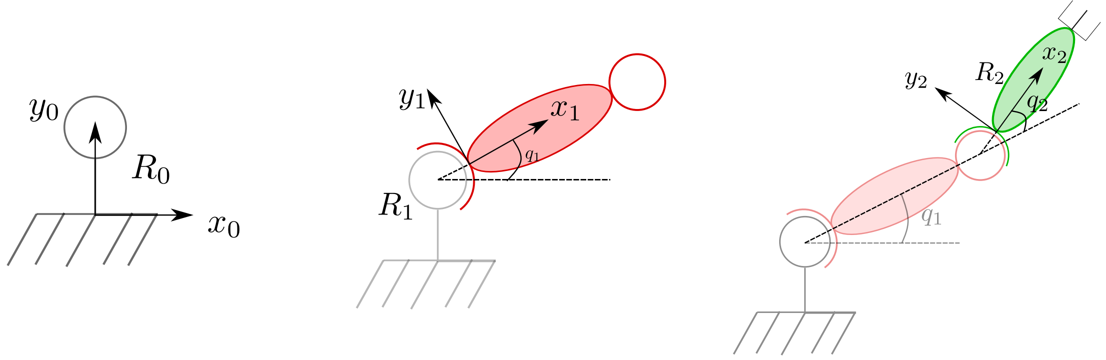



FIGURE 2.3 – Représentation de chaque repère

Dans le repère $R_0$, les coordonnées des positions :

$$\begin{gathered}
{}^0r_{c1} = {}^0Rot_1 \cdot \begin{bmatrix} l_1/2 \\ 0 \\ 0 \end{bmatrix}, \; {}^0r_1 = {}^0Rot_1 \cdot \begin{bmatrix} l_1 \\ 0 \\ 0 \end{bmatrix}, \; {}^0Rot_1 = \begin{bmatrix} \cos q_1 & -\sin q_1 & 0 \\ \sin q_1 & \cos q_1 & 0 \\ 0 & 0 & 1 \end{bmatrix}. \\
{}^0r_{c1} = \begin{bmatrix} l_1/2 \cdot \cos q_1 \\ l_1/2 \cdot \sin q_1 \\ 0 \end{bmatrix}, \; {}^0r_1 = \begin{bmatrix} l_1 \cdot \cos q_1 \\ l_1 \cdot \sin q_1 \\ 0 \end{bmatrix}.
\end{gathered} \tag{2.24}$$

Pour les coordonnées des vitesses et des accélérations, nous allons utiliser la vitesse et l'accélération angulaire dans le repère $R_0$ qui sont :

$${}^0\omega_1 = \begin{bmatrix} 0 \\ 0 \\ \dot{q}_1 \end{bmatrix}, \; {}^0\dot{\omega}_1 = \begin{bmatrix} 0 \\ 0 \\ \ddot{q}_1 \end{bmatrix} \tag{2.25}$$

Les coordonnées des positions calculées précédemment sont utilisées pour obtenir la vitesse et l'accélération :

$${}^0V_{c1} = \begin{bmatrix} -\frac{l_1}{2} \cdot \dot{q}_1 \cdot \sin q_1 \\ \frac{l_1}{2} \cdot \dot{q}_1 \cdot \cos q_1 \\ 0 \end{bmatrix}, \; {}^0V_1 = \begin{bmatrix} -l_1 \cdot \dot{q}_1 \cdot \sin q_1 \\ l_1 \cdot \dot{q}_1 \cdot \cos q_1 \\ 0 \end{bmatrix} \tag{2.26}$$

Une astuce qui est proposée dans le livre de M. Siciliano [1], est d'ajouter la gravité dans le calcul de l'accélération du 1$^{\text{er}}$ segment, considérée comme étant l'accélération précédente. L'intérêt est de ne pas avoir ce paramètre dans le calcul des forces dans le *Backward*. Donc

nous calculons les accélérations :

$$
\begin{aligned}
{}^{0}\dot{V}_1 &={}^{0}\dot{V}_0 + {}^{0}\dot{\omega}_1 \wedge {}^{0}r_1 + {}^{0}\omega_1 \wedge {}^{0}V_1 \\
{}^{0}\dot{V}_1 &= \begin{bmatrix}0\\g\\0\end{bmatrix} + \begin{bmatrix}0\\0\\\ddot{q}_1\end{bmatrix} \wedge \begin{bmatrix}l_1\cdot\cos q_1\\l_1\cdot\sin q_1\\0\end{bmatrix} + \begin{bmatrix}0\\0\\\dot{q}_1\end{bmatrix} \wedge \begin{bmatrix}-l_1\cdot\dot{q}_1\cdot\sin q_1\\l_1\cdot\dot{q}_1\cdot\cos q_1\\0\end{bmatrix} \\
{}^{0}\dot{V}_1 &= \begin{bmatrix}-l_1\cdot\dot{q}_1^2\cdot\cos q_1 - l_1\cdot\sin q_1\cdot\ddot{q}_1\\-l_1\cdot\dot{q}_1^2\cdot\sin q_1 + l_1\cdot\cos q_1\cdot\ddot{q}_1 + g\\0\end{bmatrix} \\
{}^{0}\dot{V}_{c1} &= \begin{bmatrix}-\frac{l_1}{2}\cdot\dot{q}_1^2\cdot\cos q_1 - \frac{l_1}{2}\cdot\sin q_1\cdot\ddot{q}_1\\-\frac{l_1}{2}\cdot\dot{q}_1^2\cdot\sin q_1 + \frac{l_1}{2}\cdot\cos q_1\cdot\ddot{q}_1 + g\\0\end{bmatrix}
\end{aligned}
\tag{2.27}
$$

### Segment 2

Les coordonnées du segment 2 dans $R_2$ :

$$
{}^{2}r_{c2} = \begin{bmatrix}l_2/2\\0\\0\end{bmatrix},\ {}^{2}r_2 = \begin{bmatrix}l_2\\0\\0\end{bmatrix}
\tag{2.28}
$$

Dans $R_0$, avec ${}^{0}Rot_2 = {}^{0}Rot_1 \cdot {}^{1}Rot_2$ :

$$
\begin{aligned}
{}^{0}r_2 &= {}^{0}Rot_2 \cdot {}^{2}r_2 + {}^{0}r_1 = \begin{bmatrix}l_1\cdot\cos q_1 + l_2\cdot\cos(q_1+q_2)\\l_1\cdot\sin q_1 + l_2\cdot\sin(q_1+q_2)\\0\end{bmatrix} \\
{}^{0}r_{c2} &= {}^{0}Rot_2 \cdot {}^{2}r_{c2} + {}^{0}r_1 = \begin{bmatrix}l_1\cdot\cos q_1 + \frac{l_2}{2}\cdot\cos(q_1+q_2)\\l_1\cdot\sin q_1 + \frac{l_2}{2}\cdot\sin(q_1+q_2)\\0\end{bmatrix}
\end{aligned}
\tag{2.29}
$$

Les coordonnées des vitesses calculées grâce aux équations (2.22) et ensuite les coordonnées des accélérations :

$$
\begin{aligned}
{}^{0}V_{c2} &= {}^{0}V_1 + \begin{bmatrix}0\\0\\\dot{q}_1+\dot{q}_2\end{bmatrix} \wedge {}^{0}r_{c2} = \begin{bmatrix}-l_1\cdot\dot{q}_1\cdot\sin q_1 - \frac{l_2}{2}\cdot\dot{q}_1\cdot\sin(q_1+q_2) - \frac{l_2}{2}\cdot\dot{q}_2\cdot\sin(q_1+q_2)\\l_1\cdot\dot{q}_1\cdot\cos q_1 + \frac{l_2}{2}\cdot\dot{q}_1\cdot\cos(q_1+q_2) + \frac{l_2}{2}\cdot\dot{q}_2\cdot\cos(q_1+q_2)\\0\end{bmatrix} \\
{}^{0}V_2 &= {}^{0}V_1 + \begin{bmatrix}0\\0\\\dot{q}_1+\dot{q}_2\end{bmatrix} \wedge {}^{0}r_2 = \begin{bmatrix}-l_1\cdot\dot{q}_1\cdot\sin q_1 - l_2\cdot\dot{q}_1\cdot\sin(q_1+q_2) - l_2\cdot\dot{q}_2\cdot\sin(q_1+q_2)\\l_1\cdot\dot{q}_1\cdot\cos q_1 + l_2\cdot\dot{q}_1\cdot\cos(q_1+q_2) + l_2\cdot\dot{q}_2\cdot\cos(q_1+q_2)\\0\end{bmatrix}
\end{aligned}
\tag{2.30}
$$

$$
\begin{aligned}
{}^{0}\dot{V}x_2 &= -\frac{l_2}{2}\cdot\sin(q_1+q_2)\cdot\ddot{q}_2 - \dot{q}_1^2\cdot\frac{l_2}{2}\cdot\cos(q_1+q_2) - l_1\cdot\cos q_1\cdot\dot{q}_1^2 - \\
&\quad \dot{q}_1\cdot\dot{q}_2\cdot l_2\cdot\cos(q_1+q_2) - \dot{q}_2^2\cdot\frac{l_2}{2}\cdot\sin(q_1+q_2) + \ddot{q}_1\cdot l_1\sin q_1 \\
{}^{0}\dot{V}y_2 &= \frac{l_2}{2}\cdot\cos(q_1+q_2)\cdot\ddot{q}_2 - \dot{q}_1^2\cdot\frac{l_2}{2}\cdot\sin(q_1+q_2) - l_1\cdot\sin q_1\cdot\dot{q}_1^2 - \\
&\quad \dot{q}_1\cdot\dot{q}_2\cdot l_2\sin(q_1+q_2) - \dot{q}_2^2\cdot\frac{l_2}{2}\cdot\sin(q_1+q_2) + \ddot{q}_1\cdot\frac{l_2}{2}\cdot\cos(q_1+q_2) \\
&\quad +\ddot{q}_1\cdot l_1\cdot\cos q_1 + g \\
{}^{0}\dot{V}z_2 &= 0
\end{aligned}
\tag{2.31}
$$

### Récurrence arrière : *Backward*

Le calcul des forces se formalisent comme l'équation (2.32) ci-dessous :

$$^{j}F_{j-1} + {}^{j+1}F_j = m_j \cdot {}^{0}\dot{V}_j \tag{2.32}$$

Comme les articulations sont des rotations, il faut calculer les moments, décrites par l'équation (2.33) :

$$^{j}\tau_{j-1} = {}^{j+1}\tau_j - {}^{j}F_{j-1} \wedge a + {}^{j+1}F_j \wedge b + I_j \cdot \dot{\omega}_j + \omega_j \wedge (I_j \cdot \omega_j) \tag{2.33}$$

Le moment est une force qui a l'aptitude de faire tourner un système mécanique autour d'un point, d'un axe. Donc pour un mouvement prismatique, le moment n'est pas important pour son contrôle à l'inverse d'un mouvement rotatoire. Voici ce que représente les composantes $a$ et $b$ dans la figure (2.4) ci-dessous :

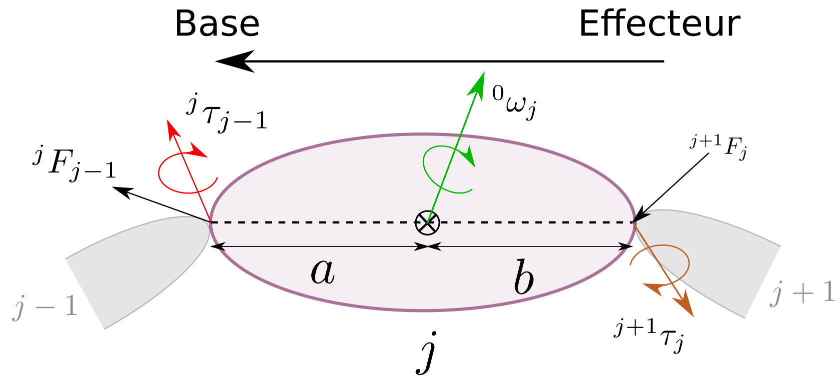



Figure 2.4 – Représentation des composantes de l'équation (2.33)

Avec la notation mise en place au début de la procédure : $a = ({}^{0}r_{cj} - {}^{0}r_j)$ et $b = (-{}^{0}r_{j+1} + {}^{0}r_{cj})$ et voici les efforts appliqués sur le robot :

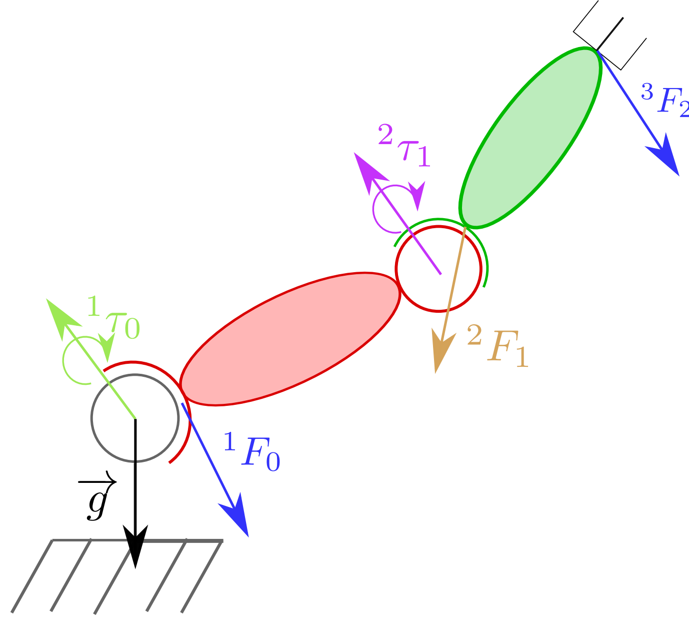



FIGURE 2.5 – Efforts appliqués au robot 2R

Dans la suite du document, les termes ne seront pas décrites intégralement car les équations seraient trop illisibles.

**Segment 2**

Comme il n'y a pas d'application de forces sur l'effecteur, $^3F_2 = \widehat{0}$, donc :

$$^2F_1 = m_2 \cdot {}^0\dot{V}_2 \tag{2.34}$$

$$\begin{aligned} ^2\tau_1 &= -{}^2F_1 \wedge ({}^0r_{c2} - {}^0 r_2) + I_2 \cdot {}^0\dot{\omega}_2 + {}^0\omega_2 \wedge (I_2 \cdot {}^0\omega_2) \\ ^2\tau_1 &= \begin{bmatrix} {}^2F_{z1} \cdot ({}^0r_{yc2} - {}^0 r_{y2}) \\ -{}^2F_{z1} \cdot ({}^0r_{xc2} - {}^0 r_{x2}) \\ -{}^2F_{x1} \cdot ({}^0r_{yc2} - {}^0 r_{y2}) + {}^2F_{y1} \cdot ({}^0r_{xc2} - {}^0 r_{x2}) \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ Izz_2 \cdot (\ddot{q}_1 + \ddot{q}_2) \end{bmatrix} \end{aligned} \tag{2.35}$$

**Segment 1**

Donc on reprend l'équation (2.32) pour le segment 1.

$$^1F_0 = m_1 \cdot {}^0\dot{V}_1 + {}^2F_1 \tag{2.36}$$

On observe que l'équation reprend la force appliquée par l'articulation suivante supportant le reste de la branche. Donc cela montre l'utilité de la *descente* de l'algorithme. Toutes les termes demandés ont été calculés. Maintenant, on calcule le moment produit sur l'articulation 1.

$$^1\tau_0 = {}^2\tau_1 - {}^1F_0 \wedge {}^0r_{c1} + {}^2F_1 \wedge (-{}^0r_2 + {}^0r_{c1}) + I_1 \cdot {}^0\dot{\omega}_1 + {}^0\omega_1 \wedge (I_1 \cdot {}^0\omega_1) \tag{2.37}$$

On retrouve les équations formelles (2.20) mais on ne va pas les réécrire car du point vu de la méthode de Newton-Euler, elles ne sont pas écrites explicitement car dès les équation (2.37) et (2.35) nous avons le résultat numérique. Il ne faut pas oublier que dans ce formalisme, on travaille qu'avec des valeurs numériques. Pour l'explication, on se devait avoir ces écritures mais du point de vu de l'outil et de son code, seules les équations "générales" de chaque terme sont utilisées. Par exemple, on ne va pas retrouver l'imposante équation permettant d'avoir l'accélération du segment 2. Ce qui n'aurait pas été le cas si cette équation devait être utilisée dans le formalisme de Lagrange car elle aurait pu être utilisée pour de la dérivation par exemple.

C'est l'avantage de cette méthode. Ce transport de valeurs numériques entre chaque segment permet que la mémoire ne soit pas saturée. De plus, les équations écrites dans le code sont courtes et simples ce qui rend l'appel d'instruction assez rapide. Un autre avantage est la modification des équations. Dans la méthode de Lagrange, s'il y a eu une erreur ou bien s'il faut ajouter un élément au début, il faut refaire toute la procédure qui est quand même longue et fastidieuse. Alors qu'avec le formalisme de Newton-Euler, les modifications sont rapides à faire, il suffit juste aller à l'étape erronée, corrigé et relancé le code. Pas besoin de retrouver chaque équation. C'est pourquoi cette méthode a été choisie pour l'outil.

# Chapitre 3

# URDF2DynamicModel : Générateur des équations du modèle dynamique d'un robot polyarticulaire

## 3.1 Présentation de URDF2DynamicModel

L'objectif du stage de Master est d'avoir un outil qui permet d'obtenir un programme, dans un langage de programmation choisi par l'utilisateur, qui calcule le modèle dynamique d'un robot poly-articulé en chaîne ouverte. La structure du robot sera donnée en entrée de l'outil sous la forme d'un URDF, ***U**nified **R**obot **D**escription **F**ormat*.

URDF2DynamicModel est développé en langage Python. Le langage du programme en sortie sera choisi par l'utilisateur. L'objectif est de pouvoir proposer du Python, du C++ ainsi que du MatLab, et potentiellement d'autres langages qui ne sont pas encore définis à ce jour.

La structure de l'outil a 3 parties distinctes :

— le parser de l'URDF : la partie qui va lire le fichier donné en entrée,

— la structure du robot : c'est la création du robot en un objet *Robot* constitué d'objets *Link*, les liens et d'objets *Joint*, les articulations,

— la création du modèle dynamique,

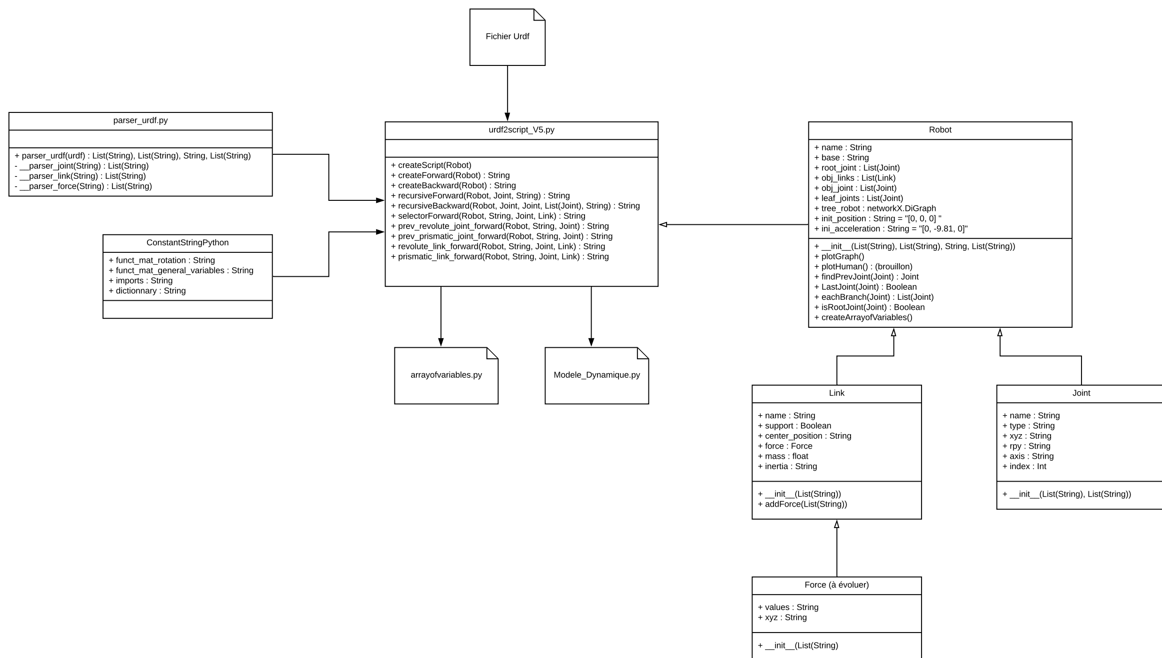



Figure 3.1 – Diagramme UML de URDF2DynamicModel

Le diagramme en plus grande version en annexe A.

Le `parser` va permettre de récupérer les données utiles et faire un premier tri dans les informations. Ces informations sont données au constructeur de l'objet `Robot` qui architecture les données permettant d'avoir un accès facile à celles-ci. L'objet est articulé autour de 2 objets : `Link` et `Joint`. ils favorisent aussi l'organisation des données. Ensuite, cet objet `Robot` est donné au script central allant produire le code calculant le modèle dynamique du robot. L'outil a deux sortie :

— un script du modèle dynamique de la structure donnée à l'entrée,
— un morceau de script permettant à l'utilisateur de pouvoir correctement initialiser ses entrées. Ce morceau de code est aussi à insérer dans le script avant l'appel de l'instruction du modèle dynamique.

## 3.2 Qu'est-ce l'URDF ?

L'URDF est le format unifié de la description d'un robot. Ce format a été créé par l'entreprise qui a créé ROS, ***R**obot **O**perating **S**ystem*. ROS est un méta système d'exploitation, quelque chose entre le système d'exploitation et le middleware. Avant ROS, chaque concepteur de robot devait créer leur propre système. Grâce à ROS, le concepteur peut l'ajouter à son robot et le faire fonctionner. Les avantages sont multiples :

— l'accès au domaine de la robotique plus facilement,
— une communauté qui apporte des outils permettant de personnaliser le système de son propre robot plus facilement,

— un système qui fonctionne et qui va pouvoir être corriger par une entreprise.

C'est comme nos ordinateurs et les systèmes d'exploitations. L'utilisateur peut créer la carcasse comme il le veut. Et pour le faire fonctionner lui met un système d'exploitation. Il n'a pas besoin de programmer son ordinateur, ou robot à partir de zéro. Puis le système d'exploitation et sa communauté propose des applications en plus pour que l'objet inanimé au départ puisse remplir son objectif. Souvent ROS est destiné au robot dit de service. Mais il est de plus en plus utilisé par les chercheurs.

Pour que le système d'exploitation "comprenne" comment le robot doit fonctionner, il faut donner la structure du robot. C'est le rôle de l'URDF. L'utilisateur crée un fichier avec comme extension urdf où il décrit son robot comme indiqué dans la documentation de ROS. D'après la documentation le robot peut être constitué de 6 éléments :

<**sensor**> : élément qui décrit un capteur, ses spécifications,

<**link**> : élément décrivant un segment du robot, son visuel, ses attributs de collision et son inertie,

<**joint**> : élément détaillant une articulation au niveau fonctionnel (coordonnée, liaison, type d'articulation ...),

<**transmission**> : description de l'actionneur par des options décrites dans le wiki ROS.

<**gazebo**> : élément comportant des informations permettant au simulateur Gazebo de fonctionner

<**model_state**> : élément qui décrit le status d'une articulation à un temps donné (position, vitesse, effort)

Dans notre cas, seuls les éléments <**joint**> et <**link**> sont utilisés pour obtenir le modèle dynamique d'un robot décrit en URDF. Voici un exemple d'un fichier URDF qui décrit un robot ayant 2 articulations pivots :

```
<robot name="robot2R">
        <joint name="joint2" type="continuous">
                <parent link="link1"/>
                <child link="link2"/>
                <origin xyz="5 0 0" rpy="0 0 0"/>
                <axis xyz="0 0 1"/>
        </joint>
        <link name="link1">
                <inertial>
                        <origin xyz="2.5 0 0" rpy="0 0 0"/>
                        <mass value="3"/>
                        <inertia ixx="1.2" ixy="0" ixz="0" iyy="15.6" iyz="0" izz="15.6"/>
                </inertial>
        </link>
        <joint name="joint1" type="continuous">
                <parent link="world"/>
                <child link="link1"/>
                <origin xyz="0 0 0" rpy="0 0 0"/>
                <axis xyz="0 0 1"/>
```

```
        </joint>
        <link name="link2">
                <inertial>
                        <origin xyz="1.5 0 0" rpy="0 0 0"/>
                        <mass value="5"/>
                        <inertia ixx="2" ixy="0" ixz="0" iyy="10" iyz="0" izz="10"/>
                </inertial>
        </link>
</robot>
```

### Formalisme URDF

L'URDF comme dit précédemment est composé de 6 éléments. Pour décrire un robot quelques règles. Tout d'abord, dans le fichier URDF, il faut déclarer le robot :

```
<robot name="robot">
```

Et quand le fichier est fini d'être complété, il faut finir le document par `</robot>`.

— **Pour une articulation :**

Pour déclarer une articulation, il faut lui donner un nom et un type. Le nom permet de retrouver plus facilement le composant dans le fichier. Le type donne le nombre de degrés de liberté de la jointure. Voici les types différents :
- **revolute** définit le pivot avec des contraintes dans ces mouvement,
- **continuous** définit aussi le pivot mais sans contraintes,
- **prismatic** définit l'articulation glissante limitée dans l'espace,
- **fixed** définit que la jointure à 0 degrés de liberté.
- **floating** qui est l'inverse du fixed, il a 6 degrès de liberté,
- **planor** donne un mouvement dans le plan perpendiculaire à l'axe.

Les types utilisés par URDF2DynamicModel sont **continuous** et **prismatic** sans prendre en compte la limitation dans l'espace. L'écriture du modèle ne prend pas en compte ces limitations. C'est à l'utilisateur, quand il aura le modèle dynamique, d'envoyer des valeurs qui seront dans les limites de l'articulation. Voici un exemple d'un pivot :

```
<joint name="pivot" type="continuous">
```

Il faut déclarer sa position grâce à l'attribut `<origin>`. Ses coordonnées sont relatives par rapport au repère précédent. Si c'est la 1$^{\text{ère}}$ articulation, son repère est celui du repère monde.

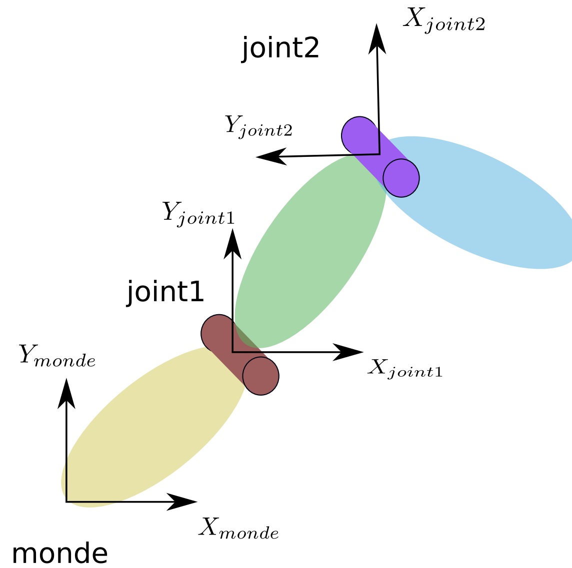



FIGURE 3.2 – Repère des articulations dans un fichier URDF

Dans l'exemple ci-dessus (figure 3.2), voici comment on indique la position et l'orientation dans un fichier URDF. Le `joint1` a une orientation identique au repère monde.

```
<joint name="joint1" type="continuous">
    <origin xyz="1.5 1.5 0" rpy="0 0 0"/>
    ...
```

Pour le `joint2` :

```
<joint name="joint2" type="continuous">
    <origin xyz="1 2 0" rpy="0 0 1.57"/>
    ...
```

Le `joint2` est à 1 unité en $X$ et 2 unités en $Y$ dans le repère de `joint1`. Dans le wiki de ros, il n'y a pas d'indications particulières à propos de l'unité de distance, le choix a été fait d'utiliser le mètre. L'unité la rotation donnée par le paramètre `rpy` est en radian. *Roll-Pitch-Yaw* est la signification de `rpy`. *Roll* indique la rotation du repère par rapport à l'axe $X$, *Pitch* indique la rotation du repère par rapport à l'axe $Y$, enfin *Yaw* indique la rotation du repère par rapport à l'axe $Z$. Sur la figure 3.2, le `joint2` à son repère qui a tourné de 90 degrés soit 1.57 radians sur l'axe $Z$, sur la composante *Yaw*.

Il faut déclarer l'axe où le mouvement s'effectue. Cet axe se fait dans le repère de l'articulation et non du repère monde. Donc si votre jointure a une rotation de 90 degrés sur l'axe $Z$ et que vous déclarez que le mouvement sur l'axe $X$, alors dans le repère monde, le mouvement s'effectuera sur l'axe $Y$. Dans la figure 3.2, les jointures ont leur axe sur $Z$ :

```
<joint name="joint1" type="continuous">
    <origin xyz="1.5 1.5 0" rpy="0 0 0"/>
```

```
        <axis xyz="0 0 1"/>
        ...
    </joint>
    <joint name="joint2" type="continuous">
        <origin xyz="1 2 0" rpy="0 0 1.57"/>
        <axis xyz="0 0 1"/>
        ...
```

Il faut déclarer aussi avec quels segments, l'articulation fait la liaison.

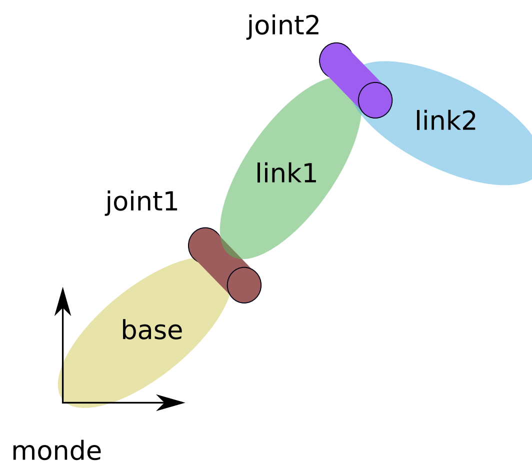



FIGURE 3.3 – Schéma du robot 2R avec les nominations des éléments

Voici la déclaration pour la figure 3.3 :

```
    <joint name="joint1" type="continuous">
        <origin xyz="1.5 1.5 0" rpy="0 0 0"/>
        <axis xyz="0 0 1"/>
        <parent link="base"/>
        <child link="link1"/>
    </joint>
    <joint name="joint2" type="continuous">
        <origin xyz="1 2 0" rpy="0 0 1.57"/>
        <axis xyz="0 0 1"/>
        <parent link="link1"/>
        <child link="link2"/>
    </joint>
```

Comme l'indique l'exemple ci-dessus, on ajoute 2 lignes pour indiquer le segment qui le précède et qui le suit. Pour les jointures, seules ces informations sont utilisées par l'outil. Normalement, pour le type d'articulation *prismatic*, il faut d'autres spécifications mais

URDF2DynamicModel ne les utilise pas. Il y aussi quelques affectations telles que la limite de l'articulation, alors vous pouvez consulter leur wiki pour en savoir plus : `http://wiki.ros.org/urdf/XML/joint` . Et quand toutes les propriétés ont été écrites, il faut fermer le balisage par `</joint>`.

— **Pour les segments :**

C'est le dernier élément primaire pour structurer un robot. Il y d'autres éléments mais ils ne seront pas présentés dans ce documents.

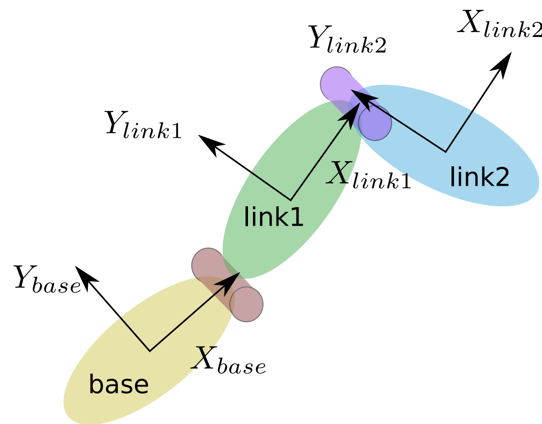



FIGURE 3.4 – Représentation des repères des segments sur le robot 2R

Les segments fonctionne de la même idée que les articulations, avec des repères relatifs. Leur repère est toujours relatif à l'articulation précédente ou au *monde* s'il n'y en a pas. Mais les segments peuvent avoir 3 repères différents. Sur la figure 3.4, les repères sont pour identifier où se positionne le centre de masse. Il y a 3 types d'attributs pour le segment :

**Inertial** : cet attribut indique toutes les composantes d'inertie. Il indique la masse de l'objet, sa matrice d'inertie et la position de son centre de masse,

**Visual** : celui désigne la géométrie visuelle du robot. Cet composante est surtout pour représenter visuellement le robot,

**Collision** : il fonctionne exactement comme l'attribut **visual**, mais il permet d'indiquer une géométrie plus simple du lien pour des temps de simulations moins grandes, par exemple.

Chaque attribut a son propre repère. Les informations données par les caractéristiques **visual** et **collision** ne sont pas utilisées par l'outil. Donc on utilise que la propriété **inertial**. Voici le format d'un segment attendu dans le fichier URDF pour faire fonctionner l'outil :

```
<link name="base">
    <inertial>
        <origin xyz="0.5 0.5 0.5" rpy="0 0 0"/>
        <mass value="3"/>
        <inertia ixx="2" ixy="0" ixz="0" iyy="10" iyz="0" izz="10"/>
    </inertial>
</link>
```

Rappel, la matrice d'inertie est symétrique, donc il n'y a besoin de décrire toutes les composantes de la matrice. Particularité, pour les segments qui ne servent pas dans le modèle dynamique. Exemple, le robot contient un articulation à 3 degrés de liberté de type révolutive. La déclaration de cette jointure va se faire, dans le fichier URDF, par 3 `joint` de type `continuous`. Pour regrouper les 3 articulations, il faut les positionner au même point. Et pour expliciter qu'elles appartiennent à la même branche, il faut déclarer des liens entre chaque fixation. C'est pour cela qu'on peut déclarer des liens "*vide*" comme :

```
<link name="link"/>
```

Cela permet de déclarer des liens *support* dans le fichier.

#### Pourquoi l'utilisation de l'URDF ?

Comme ROS est de plus en plus utilisé dans le monde, le format URDF également. Cela permet de ne pas perdre de temps à créer un template pour acquérir la structure du robot. De plus, un fichier URDF est écrit en langage XML, un langage de type balisage, qui permet de récupérer les informations souhaitées très facilement. Cela facilite la tâche de la création d'un parser. Grâce aux règles dictées par le formalisme de l'URDF et de XML, les fichiers sont très compréhensibles et la recherche des informations est simplifiée. De plus, les informations utiles pour créer un modèle dynamique du robot sont données dans les balises <**link**> et <**joint**>. L'avantage aussi d'utiliser ce format est de pouvoir le réutiliser avec ROS pour de la simulation par exemple.

## 3.3 Explication du code

URDF2DynamicModel n'a pas d'interface car ce n'était pas la priorité. Celle-ci était plutôt d'avoir quelque chose qui fonctionne et avec des résultats assez cohérents. Le code se doit être modifiable par les personnes qui prendront potentiellement la suite dessus. Donc le code est commenté et les noms des variables et des fonctions sont claires. La convention de nommage n'est pas respectée dans ce document car à ce moment le code ne le respecte pas encore, mais cela sera corrigée plus tard. Le code `urdf2script.py` est le code principale mais à l'avenir, cela ne sera plus le cas quand d'autres propositions de langages de programmation seront présentes.

#### Le parser

Le fichier `parser_urdf_v2.py` permet de récupérer les informations du fichier URDF. Comme ce format de fichier est du même type que le **XML**, *Extensible Markup Language*, on peut utiliser la librairie `lxml` [8].Cette librairie est un analyseur de fichier XML. Grâce au nom des balises, on peut retrouver n'importe quelles informations. Cela a été utile pour ne pas obliger une certaine norme d'indentation dans le fichier.

Car la version 1 de `parser_urdf.py` était un analyseur refait totalement à la main en utilisant les expressions régulières, le *regex*, pour trouver les données. Ce qui est compliqué avec les expressions régulières est l'indentation à reconnaître. Par exemple, chercher `<joint> name..` et que l'utilisateur écrit `<joint>name..`, ce n'est plus la même donnée pour l'ordinateur, alors que pour nous, c'est aussi ce qui est recherché. Pour éviter ce genre de problème, une norme était imposée. Il y a sans doute une solution sans utiliser la norme mais l'expression régulière n'est pas simple à prendre en main et est chronophage entre la recherche de solution et les tests. Telle était la cause de la mise en place de cette norme d'indentation au début. Et le parser a évolué en utilisant la librairie lxml.

Le code de ce fichier est découpé en fonction :

— `parser_urdf()` : la fonction publique principale qui a pour paramètre le chemin d'accès du fichier. C'est cette fonction qui doit être appelée de l'extérieur pour avoir les informations du fichier. Elle donne en sortie autant d'objets `List(str)` qu'il y a de fonction appelée dans celle-ci. Elle appelle d'autres fonctions privées qui analyse les informations pour les articulations, ou les segments. Elle donne aussi le nom du robot donné dans le fichier URDF,

— `__parser_joint()` : cette fonction récupère les attributs de la balise `<joint>` et les stocke dans un tableau qui est renvoyé à la fin. Le double tiret du bas avec le nom de la fonction signifie que c'est une fonction privée,

— `__parser_link()` : celle-ci récupère les attributs de la balise `<link>` et renvoie ces informations dans un tableau,

— `__parser_force()` : c'est une balise qui n'existe pas dans ROS mais qui a été créée pour permettre d'appliquer des forces extérieurs sur la structure. Cette fonction se comporte comme les 2 précédentes.

Le but d'avoir cette fonction principale qui en appelle d'autres et qui recherche l'information, est de permettre d'ajouter d'autres balises dans le fichier URDF pour avoir des précisions sur la structure, comme par exemple les frottements.

Voici la fonction `__parser_joint()`, les autres fonctions sont assez ressemblantes dans l'architecture du code à part la fonction principale qui fait juste des appels.

```
1  def __parser_joint(file_urdf):
2      joints = []
3      for joint in file_urdf.xpath("/robot/joint"):
4          j = []
5          j.append(joint.get("name"))
6          j.append(joint.get("type"))
7          j.append(joint.find("parent").get("link"))
8          j.append(joint.find("child").get("link"))
9          j.append(joint.find("origin").get("xyz"))
10         j.append(joint.find("origin").get("rpy"))
11         j.append(joint.find("axis").get("xyz"))
12         joints.append(j)
13     return joints
```

On parcours le fichier texte à la recherche de balise. Puis on ajoute dans un tableau temporaire chaque attribut de la balise trouvée. Puis celui-ci est mis dans un tableau plus global et ainsi on recommence avec d'autres balises. Et ce tableau global est renvoyé à la fonction `paser_urdf()`.

### Les objets

#### Robot

Cette classe permet de structurer les données reçues pour que celles-ci soient utilisables pour le reste du code. Ses attributs sont des tableaux d'objets :

- un tableau contenant les objets `Link`,
- un autre contenant les objets `Joint`,
- un tableau ayant seulement les articulations accrochées au segment racine du robot,
- un autre tableau qui contient les jointures accrochées aux "feuilles" du robot, les segments étant à l'opposé total de la racine,
- il y un objet de la librairie `newtworkX` qui permet de créer un graphe du robot, pour avoir un aperçu visuel de celui-ci.

Cet objet permet d'avoir tout ces attributs à l'initialisation de celui-ci grâce aux données renvoyées par le parser. Les tableaux ne sont pas triés en interne mais comme les objets `Joint` connaissent le segment précédent et suivant, alors on peut parcourir la structure comme une liste chaînée. Et les fonctions de `Robot` permettent ce parcours.

Il y aussi une fonction qui créée un 1$^{er}$ fichier : `createArrayofVariables()`. Le modèle dynamique a besoin des variables généralisées de position, vitesse et accélération de chaque articulation. Il faut pouvoir attribuer les variables de l'utilisateur à celles correspondantes dans le modèle dynamique. Pour que celui ne se trompe pas à l'affectation, cette fonction écrit un morceau de code qui reprend chaque variable généralisée avec comme nom, celui de leur articulation écrite dans le fichier URDF, comme ceci par exemple :

```
1 gv_position_joint1 =
2 gv_velocity_joint1 =
3 gv_acceleration_joint1 =
```

On retrouve `gv_+attribut_+nom` :

- `gv_` : pour *general variables*, en français variables généralisées,
- `position_`, `velocity_`, `acceleration_` : indique le type de variable généralisée attendue,
- `name` ou `joint1` : le nom correspondant à l'articulation.

Ce processus est fait pour articulation et ensuite toutes ces variables sont insérées dans un tableau. Et ce tableau est à mettre directement comme paramètre de la fonction du modèle dynamique.

### Link et Joint

Ce sont les classes contenant les informations utiles. Celles-ci transforment aussi les paramètres sous le bon format pour écrire le modèle dynamique. Les variables récupérées dans l'URDF ne sont pas forcement réutilisable telles quelles. De plus il faut penser que les variables ne vont pas être directement calculées mais écrites dans un autre script. Donc leur format doit être sous **str**, un string, et qui a la forme d'une variable Python. Exemple, les coordonnées dans la balise <origin> à l'attribut `xyz`, quand elles sont récupérées, on obtient `"4 1 5"`. S'il cette information est écrite comme cela, le code ne va pas fonctionner. Il manque les séparateurs entre les composante, une virgule sous Python, et ce qu'il définie en temps que tableau, les crochets. Donc les objets `Link` et `Joint` permettent de les retranscrire. Et on obtient `"[0.0, 1.0, 2.0]"`. Ces objets mettent aussi les valeurs sous le bon format, des **float**.

### Newton-Euler en Python

Le code principale se situe dans `urd2script_V5.py`. Le but de ce fichier est de créer le fichier de sortie qui contient le modèle dynamique. Pour avoir le modèle, il faut la fonction `createScript(Robot)`. Elle prend en entrée l'objet `Robot` initialisé avec les variables renvoyées par le parser. Et cette fonction principale va pouvoir appeler les fonctions utiles. Il y a d'abord 2 grandes fonctions :

- `createForward()` : cette fonction va permettre de créer la partie de la récursion avant, en anglais *Forward*. Par rapport à la partie 2, c'est le moment du calcul des positons, vitesses et accélérations des composantes utiles. Dans la version 5 de ce fichier, cette fonction prépare aussi la récursion arrière, en anglais le *Backward*,
- `createBackward()` : celle-ci fini d'écrire dans le fichier finale, les équations du *Backward*.

Le *Forward* est la partie plus longue en terme de code car les équations dépendent beaucoup du type de l'articulation présente et précédente. Donc il y a des fonctions pour chaque architecture :

- pour le calcul de la position de l'articulation :
  - si l'articulation précédente est une jointure prismatic : `prev_prismatic_joint_forward()`
  - si l'articulation précédente est une jointure continuous : `prev_revolute_joint_forward()`
- pour le calcul de la position du centre de gravité du segment :
  - si l'articulation est une jointure prismatic : `prismatic_link_forward()`
  - si l'articulation est une jointure continuous : `revolute_link_forward()`

Ces fonctions permet d'écrire l'équation adéquate. Voici comme exemple, la fonction `prev_prismatic_joint_f`

```
1 def prev_revolute_joint_forward(robot, actual_indentation,actual_joint):
2     prev_joint = robot.findPrevJoint(actual_joint)
3     script = "#Joint : {0} {1}\n".format(actual_joint.name, actual_joint.type)
4     #^{i-1}p_j = ^{i-1}p_j + Rotation_{j-1} \cdot ^{i}p_j
```

```
5       script += "{0}values_robot[\"position_{1}\"] =
        ↪  values_robot[\"position_{2}\"] +
        ↪  dot(rotation_{2},{3})\n".format(actual_indentation,
        ↪  actual_joint.name,prev_joint.name, actual_joint.xyz)
6       #^{i-1}v_j = ^{i-1}v_j + \omega_{j-1} \wedge ^{i-1}p_j
7       script += "{0}values_robot[\"velocity_{1}\"] =
        ↪  values_robot[\"velocity_{2}\"] + cross(omega_rev_{2},
        ↪  values_robot[\"position_{1}\"])\n".format(actual_indentation,
        ↪  actual_joint.name, prev_joint.name)
8       #^{i-1}a_j = ^{i-1}a_j + \dot{\omega}_{j-1} \wedge ^{i-1}p_j +
        ↪  \omega_{j-1} \wedge ^{i-1}v_j
9       script += "{0}values_robot[\"acceleration_{1}\"] =
        ↪  values_robot[\"acceleration_{2}\"] + cross(omegap_rev_{2},
        ↪  values_robot[\"position_{1}\"]) + cross(omega_rev_{2},
        ↪  values_robot[\"velocity_{1}\"])\n".format(actual_indentation,
        ↪  actual_joint.name, prev_joint.name)
10      #Tips to compute the rotation's matrix in the part of link's computation
11      script += "{0}rotation_{1} =
        ↪  dot(matrixRotationGeneralVariables(general_variables_{1}[0]),
        ↪  rotation_{2})\n".format(actual_indentation,actual_joint.name,prev_joint.name)
12      if actual_joint.type=="continuous":
13          script += "{0}omega_rev_{1} = omega_rev_{2} +
            ↪  general_variables_{1}[1]\n".format(actual_indentation,
            ↪  actual_joint.name,prev_joint.name)
14          script += "{0}omegap_rev_{1} = omegap_rev_{2} +
            ↪  general_variables_{1}[2]\n".format(actual_indentation,
            ↪  actual_joint.name,prev_joint.name)
15      return script
```

Les autres méthodes fonctionnent avec le même principe. On récupère les données de l'articulation bougeant le composant observé puis on écrit les équations décrites dans le chapitre 2. Les chaînes de caractères sont écrites avec des noms de variables, des indications, etc qui sont toujours constantes et on leur ajoute les valeurs qui ne sont pas constantes grâce à la méthode `.format()` qui remplace les caractères `{...}` par les valeurs mis en paramètres de cette méthode. On observe une variable récurrente dans les chaînes de caractères : `values_robot["..."]`. C'est un dictionnaire permettant à l'utilisateur de récupérer les données qu'il a besoin.

Pour sélectionner la fonction pertinente, c'est la fonction `selectorForward()` qui est appelée dans la fonction `createForward()`. `selectorForward()` permet aussi d'initialiser les variables comme le vecteur de gravité, les articulations à la racine et d'autres variables utiles pour les équations. Puis elle appelle une fonction récursive qui parcours la structure du robot et permet d'avoir les équations pour chaque articulation et centre de gravité des segments.

Quand le parcours est fini dans la partie *Forward*, la fonction `createBackward()` est appe-

lée qui initialise quelques variables pour la partie *Backward*. Puis une fonction récursive allant des "feuilles" de la structure jusqu'à la racine écrit dans le fichier les équations pour les efforts. Comme les équations ne dépendent moins de la structure de l'articulation, tout est fait dans la fonction récursive `recursiveBackward()`.

# Chapitre 4

# Résultats

Dans cette partie, il y a 2 types de résultats :

1. dans la section 4.1, nous allons avoir le résultat produit par le fichier `urdf2script.py`. Ce qui est produit est comparé aux équations du formalisme de Newton-Euler,
2. dans la section 4.2, nous allons regarder les résultats produits pour une structure humanoïde grâce aux données du site HuMoD [9]. Il y a encore des problèmes de code au niveau des noeuds qu'on peut trouver dans une structure et donc les forces sont fausses pour l'instant avec le modèle dynamique produit par URDF2DynamicModel.

La comparaison est faite par rapport un robot 2R utilisé dans le chapitre 2. Le site HuMoD regroupe des données mesurées sur 2 types de personne :

1. une femme de 27 ans mesurant 161 cm pesant 57.3kg,
2. un homme de 32 ans mesurant 179 cm pesant 84.8kg.

Les données mesurées sont :

— les variables généralisées de position, de vitesse et d'accélérations des articulations,
— la position des articulations dans le repère global,
— l'activité des muscles,
— la force de réaction du sol.

Elles ont été mesurées grâce à des capteurs positionnées sur le corps du sujet à des endroits choisis préalablement. Les sujets ont marché et couru sur un tapis de course qui tournait à différentes vitesses. les résultats dans la section 4.2 viennent des données des variables généralisées du sujet femme quand elle courait et que le tapis de course fonctionnait à 3.0 m/s pendant une durée de 110s. Il y a 20s de mise en place au début, 60s où la personne court d'une manière plutôt régulière et 30s où la personne commence à se relâcher et s'arrête. Nous utilisons les données de quelques secondes pendant les 60s de course régulière. Et les résultats sont comparés aux positions des articulations données par le site.

## 4.1 Robot2R

Pour ces résultats, nous allons utilisées le script `urdf2script_V5.py`. Voici le fichier urdf utilisé :

```
<robot name="robot_2R">
        <joint name="joint2" type="continuous">
                <parent link="link1"/>
                <child link="link2"/>
                <origin xyz="2 0 0" rpy="0 0 0"/>
                <axis xyz="0 0 1"/>
        </joint>
        <link name="link1">
                <inertial>
                        <origin xyz="1 0 0" rpy="0 0 0"/>
                        <mass value="3"/>
                        <inertia ixx="1.2" ixy="0" ixz="0" iyy="15.6" iyz="0" izz="15.6"/>
                </inertial>
        </link>
        <joint name="joint1" type="continuous">
                <parent link="world"/>
                <child link="link1"/>
                <origin xyz="0 0 0" rpy="0 0 0"/>
                <axis xyz="0 0 1"/>
        </joint>
        <link name="link2">
                <inertial>
                        <origin xyz="1 0 0" rpy="0 0 0"/>
                        <mass value="5"/>
                        <inertia ixx="2" ixy="0" ixz="0" iyy="10" iyz="0" izz="10"/>
                </inertial>
        </link>
        <link name="world"/>
</robot>
```

Tout d'abord, il faut procéder à la partie *Forward*.

```
1  #Part Forward : compute position, velocity and acceleration of all joints and
   ↪ links
2  general_variables_joint1 = np.array([p*np.array([0., 0., 1.]) for p in
   ↪ np.array(array_of_variables[1])])
3  #Root Joint : joint1
4  values_robot["position_joint1"]= np.array([0,0,0]) +np.array([0., 0., 0.])
5  values_robot["velocity_joint1"] = np.array([0.0,0.0,0.0])
6  values_robot["acceleration_joint1"] =np.array([ 0.  ,-9.81, 0.  ])
7  rotation_joint1 = mat([[1.0,0.0,0.0],[0.0,1,0.0],[0.0,0.0,1]])
8  omega_rev_joint1 = general_variables_joint1[1]
9  omegap_rev_joint1 = general_variables_joint1[2]
```

```
10  rotation_joint1 =
    ↪ dot(matrixRotationGeneralVariables(general_variables_joint1[0]),
    ↪ rotation_joint1)
11
12  #Link : link1
13  rotation_joint1 = dot(matrixRotation([0., 0., 0.]), rotation_joint1)
14  values_robot["position_link1"] = values_robot["position_joint1"] +
    ↪ dot(rotation_joint1,[1., 0., 0.])
15  values_robot["velocity_link1"] = values_robot["velocity_joint1"] +
    ↪ cross(omega_rev_joint1, values_robot["position_link1"])
16  values_robot["acceleration_link1"] = values_robot["acceleration_joint1"] +
    ↪ cross(omegap_rev_joint1, values_robot["position_link1"]) +
    ↪ cross(omega_rev_joint1, values_robot["velocity_link1"])
17  #Forces and torques in local (joint1 and link1) without extern forces
18  values_robot["forces_joint1"] = dot(values_robot["acceleration_link1"],3.0)
19  values_robot["torques_joint1"] = 0
20  values_robot["torques_joint1"] += dot(mat([[ 1.2, 0. , 0. ], [ 0. ,15.6, 0. ],
    ↪ [ 0. , 0. ,15.6]]),omegap_rev_joint1) + cross(omega_rev_joint1, dot(mat([[
    ↪ 1.2, 0. , 0. ], [ 0. ,15.6, 0. ], [ 0. , 0. ,15.6]]),omegap_rev_joint1))
21
22  general_variables_joint2 = np.array([p*np.array([0., 0., 1.]) for p in
    ↪ np.array(array_of_variables[0])])
23  #Joint : joint2 continuous
24  values_robot["position_joint2"] = values_robot["position_joint1"] +
    ↪ dot(rotation_joint1,[2., 0., 0.])
25  values_robot["velocity_joint2"] = values_robot["velocity_joint1"] +
    ↪ cross(omega_rev_joint1, values_robot["position_joint2"])
26  values_robot["acceleration_joint2"] = values_robot["acceleration_joint1"] +
    ↪ cross(omegap_rev_joint1, values_robot["position_joint2"]) +
    ↪ cross(omega_rev_joint1, values_robot["velocity_joint2"])
27  rotation_joint2 =
    ↪ dot(matrixRotationGeneralVariables(general_variables_joint2[0]),rotation_joint1)
28  omega_rev_joint2 = omega_rev_joint1 + general_variables_joint2[1]
29  omegap_rev_joint2 = omegap_rev_joint1 + general_variables_joint2[2]
30
31  #Link : link2
32  rotation_joint2 = dot(matrixRotation([0., 0., 0.]), rotation_joint2)
33  values_robot["position_link2"] = values_robot["position_joint2"] +
    ↪ dot(rotation_joint2,[1., 0., 0.])
34  values_robot["velocity_link2"] = values_robot["velocity_joint2"] +
    ↪ cross(omega_rev_joint2, values_robot["position_link2"])
35  values_robot["acceleration_link2"] = values_robot["acceleration_joint2"] +
    ↪ cross(omegap_rev_joint2, values_robot["position_link2"]) +
    ↪ cross(omega_rev_joint2, values_robot["velocity_link2"])
```

```
36 #Forces and torques in local (joint2 and link2) without extern forces
37 values_robot["forces_joint2"] = dot(values_robot["acceleration_link2"],5.0)
38 values_robot["torques_joint2"] = 0
39 values_robot["torques_joint2"] += dot(mat([[ 2., 0., 0.], [ 0.,10., 0.], [ 0.,
   ↪ 0.,10.]]),omegap_rev_joint2) + cross(omega_rev_joint2, dot(mat([[ 2., 0.,
   ↪ 0.], [ 0.,10., 0.], [ 0., 0.,10.]]),omegap_rev_joint2))
```

On peut observer juste avant les calculs des composantes : `general_variables_joint1 = np.array([p*np.array([0., 0., 1.])for p in np.array(array_of_variables[1])])`, aux lignes 2 et 22. Comme les variables généralisées sont des scalaires en entrée alors qu'il faut des vecteurs, cette instruction permet ce passage en multipliant ce scalaire au vecteur de l'axe du mouvement de l'articulation. Donc on obtient pour l'articulation 1 $[0\,0\,q1]^T$.

Puis de la linge 4 à 10, nous avons l'initialisation et le calcul des composantes de l'articulation 1. On retrouve les spécificités disant que la jointure 1 est positionnée au niveau du repère monde aux coordonnées $[0\,0\,0]^T$ et les lignes suivantes sont les déclarations de la vitesse et de l'accélération de cette articulation. Le calcul de la matrice rotation est faite à la ligne 7 et poursuivi à ligne 10. La ligne 7 crée d'abord la matrice de rotation du repère global. Puis comme c'est une articulation rotoïde, on multiplie cette matrice par la matrice de rotation produite avec $q1$ et on obtient le vecteur angulaire du prochain segment. On déclare aussi $\omega_1$ et $\dot{\omega}_1$ aux lignes 8 et 9.

De la ligne 13 à 16, on calcul les composantes du centre de gravité du segment 1 comme avec les équations (2.24), (2.26) et (2.27).

On observe de la ligne 18 à 20 qu'il y a des calculs de forces et de moments. Normalement dans la méthode de Newton-Euler cela est fait dans la partie *Backward*. C'est pour simplifier celui-ci. On ne calcule qu'avec les termes qu'on connaît de l'équation. On reprend qu'une partie de l'équation (2.32) et (2.33) étant $I_j \cdot \dot{\omega}_j + \omega_j \wedge (I_j \cdot \omega_j)$ et ${}^0\dot{V}_j \cdot m_j$. Cette simplification permet de rencontrer moins de problème pour la récursion arrière.

Puis on recommence à l'articulation 2 de la ligne 14 à 29 et pour le segment 2 de la ligne 32 à la ligne 39 comparables aux équations (2.29), (2.30) et (2.31).

Et maintenant la partie *Backward* :

```
1 # Part Backward : compute of all forces and torques
2 #Values of joint2
3 values_robot["torques_joint2"] += -
  ↪ cross(values_robot["forces_joint2"],(values_robot["position_link2"] -
  ↪ values_robot["position_joint2"]))
4
5 #Values of joint1
6 values_robot["forces_joint1"] += values_robot["forces_joint2"]
7 values_robot["torques_joint1"] += values_robot["torques_joint2"]
```

```
8  values_robot["torques_joint1"] +=
   ↪ cross(values_robot["forces_joint2"],(values_robot["position_joint1"] -
   ↪ values_robot["position_joint2"]))
9  values_robot["torques_joint1"] += -
   ↪ cross(values_robot["forces_joint1"],(values_robot["position_link1"] -
   ↪ values_robot["position_joint1"]))
```

On finit les équations des efforts en commençant par la "feuille" donc la 2ème articulation pour aller à la 1ère jointure. Comme il n'y a pas de forces qui s'ajoutent, on retrouve bien l'équation (2.34) calculée à ligne 37 du *Forward*, où seul l'accélération du segment multipliée à sa masse est prise en compte. Puis on finit de calculer le moment en ajoutant $-{}^{2}F_1 \wedge ({}^{0}r_{c2} -{}^{0} r_2)$. Comme la partie inertielle a été ajoutée précédemment, on retombe sur l'équation (2.35).

Vient au tour de la partie 1 du robot. On ajoute la force ${}^{2}F_1$ à ${}^{1}F_0$ à la ligne 6 du *Backward*, et on retombe sur l'équation (2.36). Puis on ajoute les parties manquantes au moment de cette partie du robot. On ajoute ${}^{2}\Gamma_1$ à ${}^{1}\Gamma_0$ à la ligne 7, ${}^{2}F_1 \wedge (-{}^{0}r_2 + {}^{0}r_{c1})$ à ligne 8 et $-{}^{1}F_0 \wedge {}^{0}r_{c1}$ à la ligne 9 . Et à cet instant, le moment de la jointure 1 correspond à l'équation (2.37).

URDFD2DynamicModel additionne logiquement chaque terme utile sous forme de chaîne de caractères au script de sortie en respectant les équations pouvant être trouvé avec la méthode de Newton-Euler. Il y a la légère différence où on calcule une partie des équations des efforts dans la partie de la récursion avant.

## 4.2 Structure humanoïde : robot 3P33R

Le corps contient 30 degrées de liberté sans compter les doigts. Les 6 derniers ddl font références au corps qui peut bouger dans le monde sans attache. Voici le schéma du corps humain où cela montre la position des articulations :

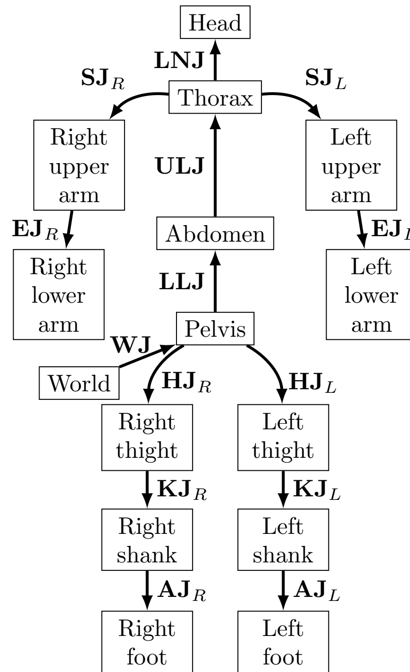
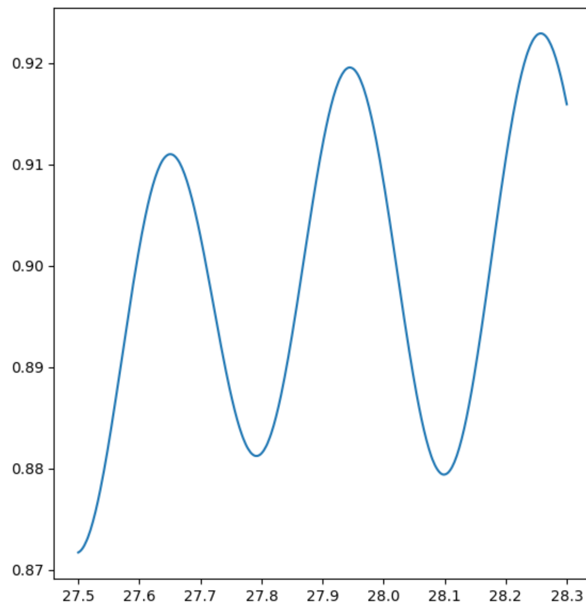



FIGURE 4.1 – Schéma du corps avec la positon des segments et des articulations

Voici la présentation de chaque articulation et comment ils sont décrits dans le fichier URDF

— $WJ$ : articulation à 6 ddl : 3 prismatiques et 3 rotoïdes,
— $LLJ$ : articulation à 2 ddl : 2 rotoïdes, représentant le bas du dos,
— $ULJ$ : articulation à 3 ddl : 3 rotoïdes, représentant la colonne vertébrale,
— $LNJ$ : articulations à 3 ddl : 3 rotoïdes, représentant le cou,
— $SJ_R$ : articulations à 3 ddl : 3 rotoïdes, représentant l'épaule droite,
— $SJ_L$ : articulations à 3 ddl : 3 rotoïdes, représentant l'épaule gauche,
— $EJ_R$ : articulations à 1 ddl : 1 rotoïde, représentant le coude droit,
— $EJ_L$ : articulations à 1 ddl : 1 rotoïde, représentant le coude gauche,
— $HJ_R$ : articulations à 3 ddl : 3 rotoïdes, représentant la hanche droite,
— $HJ_L$ : articulations à 3 ddl : 3 rotoïdes, représentant la hanche gauche,
— $KJ_R$ : articulations à 1 ddl : 1 rotoïde, représentant le genou droit,
— $KJ_L$ : articulations à 1 ddl : 1 rotoïde, représentant le genou gauche,
— $AJ_R$ : articulations à 3 ddl : 3 rotoïdes, représentant la cheville droite,
— $AJ_L$ : articulations à 3 ddl : 3 rotoïdes, représentant la cheville gauche.

En voyant le nombre de degrées de liberté, on peut se dire que la méthode de Lagrange prendrait beaucoup de temps pour trouver le modèle à cause des dérivations. Maintenant on peut tester le modèle dynamique produit par l'outil avec les données du site [9]. D'abord, on doit voir si les positions correspondent entre les articulations du produit de l'outil et ce qui estimé par le site. Sur la droite, c'est le résultat du modèle dynamique et à gauche les données de HuMoD. Les données de HuMod sont en millimètres et le modèle en mètres.



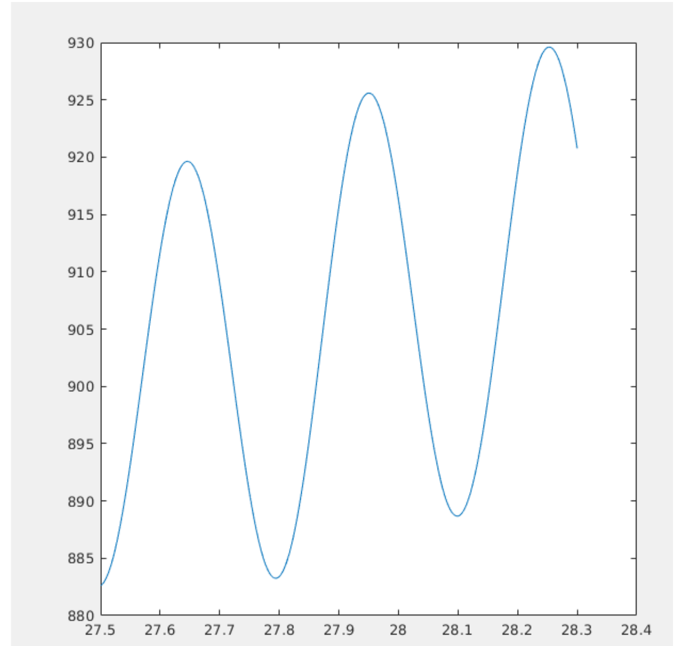
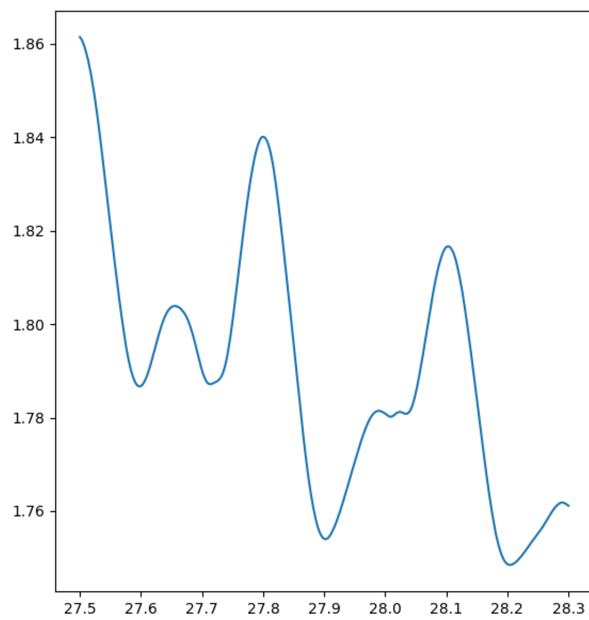




FIGURE 4.2 – Position en $y$ de l'articulation LLJ

Les différences sont de l'ordre de quelques millimètres voir de centimètres, mais les différences sont maximum autour de 2cm. Cela est dû au fichier URDF où la position de LLJ qui n'est pas exacte.

Un problème est survient au bout de la 3ème articulations, par exemple LNJ, qui ne suit pas du tout les données de HuMoD.



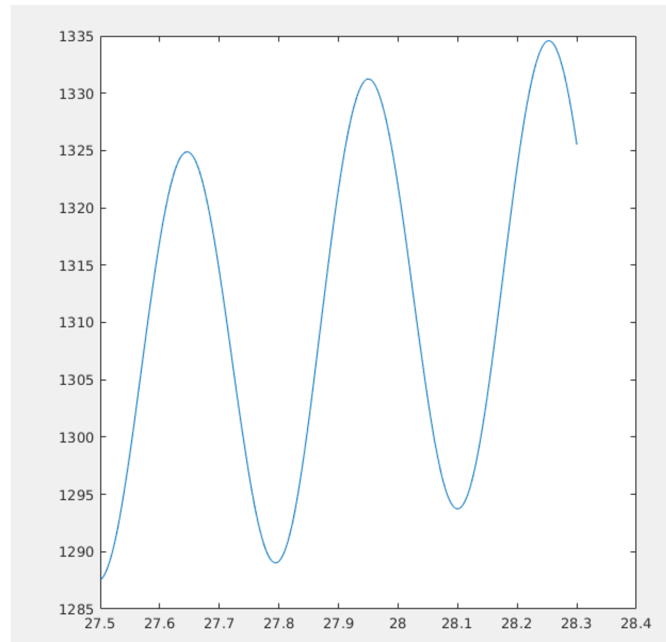
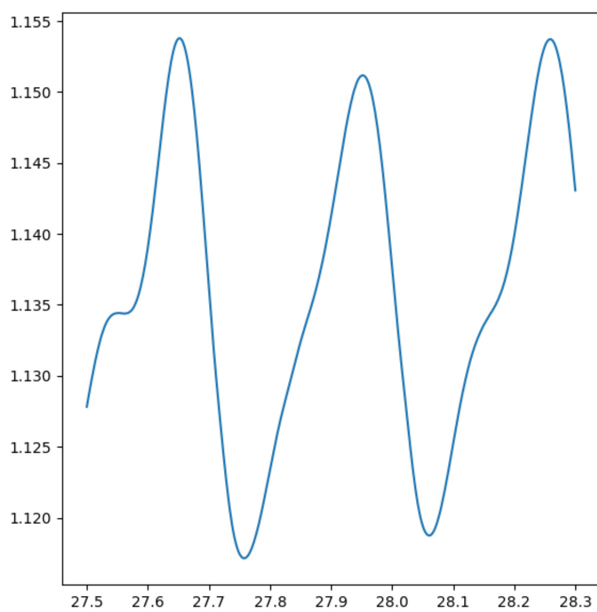




FIGURE 4.3 – Position en $y$ de l'articulation LNJ

Et pourtant l'articulation précédente est toujours au bon endroit.



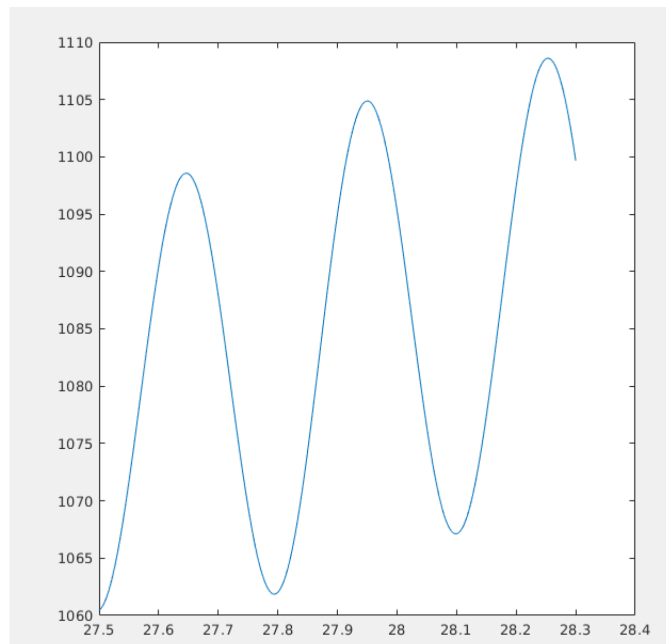




FIGURE 4.4 – Position en $y$ de l'articulation ULJ

Le problème vient potentiellement que LNJ se trouve au niveau d'un embranchement. Le thorax possède 4 articulations, et potentiellement le code ajoute des valeurs qu'il ne faut pas. Et ce problème des noeuds revient aussi dans la partie *Backward*.

# Chapitre 5

# Conclusion

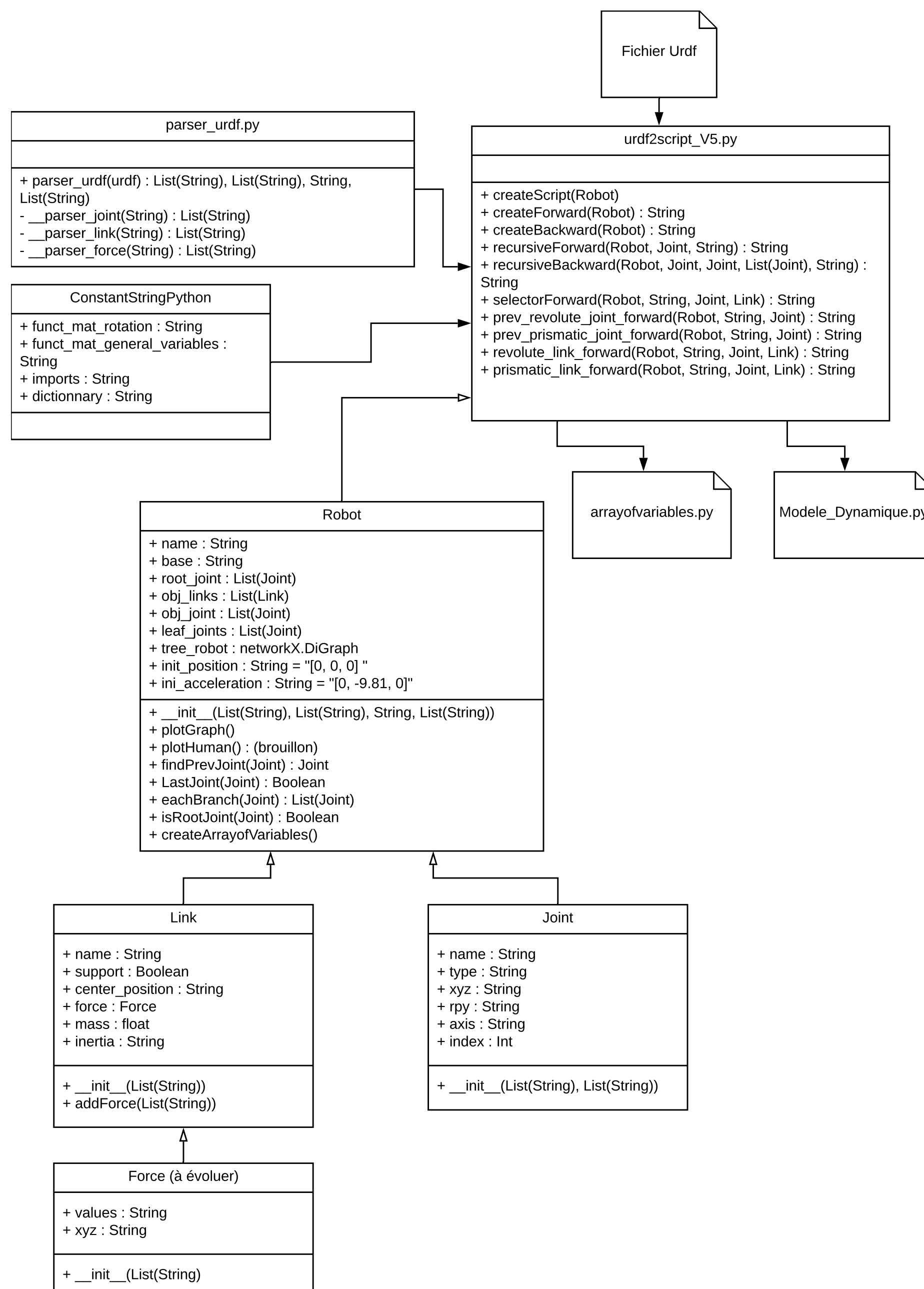
L'outil a besoin encore d'être développé. Il faut d'abord corriger les problèmes au niveaux du code. Les problèmes ne sont pas très importants et concernent surtout les structures possédant des noeuds. Une mauvaise itération dans une fonction récursive, un signe mal placé mais comme le code est imposant l'erreur peut être bien caché. De plus, pour vérifier les erreurs au niveau des résultats peut être compliqué. Heureusement que l'étude HuMoD ait été produite. Même si la structure est unique, les résultats ne peuvent pas être comparés car d'autres ne sont pas trouvables sur Internet. il faudrait les outils comme Simulink pour tester.

De plus, le fait de régler des problèmes d'équations pour la méthode de Newton-Euler est moins compliqué que pour la méthode de Lagrange. Cela est permis car les équations de Newton-Euler sont justes des "briques" d'équations formelles. On ne fait que des opérations basiques. Alors que corriger un code utilisant du Lagrange, cela serait plus compliqué car si une dérivation ne fonctionne pas, il faut tester toutes les dérivations faites pour savoir où est l'erreur. Avec Newton-Euler, juste en regardant le modèle la mauvaise place des termes peut être trouvée assez rapidement. Il faut faire attention aussi au fichier URDF. Si les données de départ ne sont pas correctes alors forcément les résultats finaux ne sont pas bons.

URDF2DynamicModel peut évoluer aussi pour qu'il soit plus simple à utiliser. Ajouter une interface graphique pour l'utilisateur par exemple. Faire un graphe de la structure avec les données du fichier URDF. Cette implémentation de graphe à commencer mais ne donne pas de résultats correctes pour l'instant. Peut-être l'évolution à suivre est de développer, après les corrections des bogues, du code permettant de produire des modèles dynamiques sous un autre langage de programmation. Le C++ est très utilisé pour ce type de travail car le code se compile d'une façon plus efficace. Le Python est très utile car c'est un langage possédant beaucoup de librairie mathématique.

# Annexe A

# Diagramme UML de URDF2DynamicModel



# Bibliographie

[1] L. S. e. L. V. B. Siciliano and G. Oriolo. *Robotics : Modelling, Planning and Control.* Springer, 2009. ISBN 9781846286414.

[2] W. K. et E. Dombre. *Modeling, Identification & Control of Robots.* Hermes Penton Ltd, Nantes, 2002. ISBN 1903996139.

[3] I. et Wasima Khalil. Github : Symoro. URL `https://github.com/symoro/symoro`.

[4] D. G. C. e. C. S. M. Frigerio, J. Buchli. *"RobCoGen :a code generator for efficient kinematics and dynamics of articulatedrobots, based on Domain Specific Languages,"Journal of SoftwareEngineering for Robotics (JOSER)*, volume 7, no. 1, pages 36–54. 2016.

[5] e. M. A. S. M. G. Hollars, D. E. Rosenthal. Sd/fast user's manual, 1991.

[6] e. P. F. N.Docquier, A.Poncelet. *"Robotran : a powerfulsymbolic gnerator of multibody models", Mechanical Sciences*, volume 4, no. 1, pages 199–219. 2013.

[7] L. B. e. R. P.-G. P.-B. Wieber, F. Billet. Metapod — template meta-programming applied to dynamics :cop-comtrajectoriesfiltering. *IEEE-RAS InternationalConference on Humanoid Robots (Humanoids)*, 2014.

[8] M. F. e. I. B. Stephen Behnel. lxml - xml and html with python. URL `https://lxml.de`.

[9] J. Wojtusch. Humod database. URL `https://www.sim.informatik.tu-darmstadt.de/res/ds/humod`.

**Résumé** —The dynamic modeling of a robot provides path equations, motor torque and other important information to move the manipulator. Either by hand or by a formal calculation's software, the time to set up the equation is long and can bring errors. The goal of this course is to create a tool that solves this problem. By the choice of the method allowing to have the dynamic model while passing by the explanations of the code, this document takes again the work of this intership. The tool must compute dynamic model for simple and hard structure without to take time with the help of URDF's file.

**Mots clés :** Modélisation Dynamique, Robot, Méthode de Lagrange, Méthode de Newton-Euler, URDF, Procédure automatique, Calcul Formel, Structure humanoïde.

---

Polytech Angers, Université d'Angers
62, avenue Notre Dame du Lac
49000 Angers