Rapport de stage
Optimisation de mouvements complexes pour les robots à pattes
L3 Informatique
Ecole normale supérieure de Paris

Axelle Piot
axelle.piot@ens.fr

Responsable de stage : Sébastien Lengagne
sebastien.lengagne@univ-bpclermont.fr
Université Blaise Pascal, Clermont Ferrand, France
ISPR - Equipe MACCS

Juin - juillet 2014

Remerciements

Je remercie Sébastien Lengagne pour m'avoir accueillie dans son équipe de recherche pendant toute la durée du stage. J'ai beaucoup apprécié son excellent encadrement, sa présence et ses explications. J'ai pu, grâce à lui, découvrir un sujet de recherche passionnant.

Je remercie également Anne Bouillard et Marc Lelarge en charge des stages à l'école normale supérieure de Paris, sans qui je n'aurais pas eu l'occasion de bénéficier d'un stage aussi intéressant.

Je tiens aussi à remercier Isabelle Delais pour s'être occupée de toutes la correspondance relative au stage, pour sa gentillesse et sa disponibilité.

Merci à Clément Fouque pour ses astuces de "vieux loup" de programmeur.

Je remercie enfin Marc Pouzet, mon tuteur, pour m'avoir soutenue et conseillée tout au long de cette année très chargée.

Table des matières

1 Présentation du stage

1.1 Lieu du stage : L'Institut Pascal

Le stage s'est déroulé au sein de l'équipe **MACCS** - *Modeling, Autonomy and Control of Complex Systems* - de l'axe de recherche **ISPR** - *ImageS, Perception systems and Robotics* - de l'Institut Pascal (UMR 6602 UBP/CNRS) à l'Université Blaise Pascal de Clermont-Ferrand sous la supervision de Sébastien Lengagne, Maître de conférences.

L'axe de recherche ISPR est composée de quatre équipes de recherche, l'équipe ComSee spécialisée dans la vision par ordinateur, l'équipe DREAM, spécialisée dans les systèmes embarqués, l'équipe PerSyst, spécialisée dans la perception des systèmes et l'équipe MACCS.

L'équipe de recherche MACCS est une unité de recherche liée au CNRS, à l'UBP - *Université Blaise Pascal* - et à l'IFMA - *Institut Français de Mécanique Avancée* - (UMR 6602 CNRS / UBP / IFMA). L'équipe MACCS s'occupe principalement de la modélisation et du contrôle de robots mobiles et manipulateurs, de la vision par ordinateur, de l'asservissement visuel et de l'anticipation des comportements pour des applications urbaines hors routes.

Lengagne travaille sur le projet **MoGS** [1] - *Motion Generation Software* - qui est un logiciel codé en C++ qui sert à simuler un ou des robots dans un environnement 3D, à optimiser les mouvements des robots ainsi que d'autres fonctionalités qui ne seront pas abordées lors ce stage. Les algorithmes de simulation et d'optimisation suivent un des deux modèles dynamiques, le direct ou l'inverse [3] : ces notions seront explicitées plus loin dans le rapport. Plusieurs vidéos de Lengagne avec le HRP2 [2] illustrant ce qu'il est possible de réaliser avec le modèle dynamique inverse sont disponibles sur Youtube [4].

1.2 L'optimisation de mouvements

La plannification des mouvements de robots à pattes, à base fixe ou flottante, en respectant de nombreuses contraintes, dans un environnement quelconque, est un problème complexe. Il y a les contraintes liées au robot lui-même, telles que les limites des articulations, la limite des angles de rotation, la limite de la vitesse de rotation et la limite des couples maximaux applicables, cela dans le cas d'une articulation rotoïde, puis les contraintes liées à l'environnement, à d'autres robots, à des obstables ou bien à des reliefs.

Pour déplacer un robot d'une posture de départ à une posture désirée, il y a souvent plusieurs solutions possibles. Si plusieurs solutions existent, il faut déterminer laquelle ou

1. `https://github.com/lengagne/MoGS`
2. `https://unit.aist.go.jp/is/cie/group/jrl_e.html`

lesquelles sont les plus optimales en fonction de critères définis.

Dans le contexte de ce stage, deux critères sont utilisés, le critère **TorqueSquare** :

$$TorqueSquare = IntegrationStep \times \sum_{t=0}^{n-1} \Gamma(t)^2 \tag{1.1}$$

Ce critère a pour but de minimiser les dépenses d'énergie fournie aux actionneurs des articulations.

Le critère **Jerk** :

$$Jerk = IntegrationStep \times \sum_{t=0}^{n-2} \left(\ddot{q}(t) - \ddot{q}(t+1)\right)^2 \tag{1.2}$$

Ce critère a pour but de rendre le mouvement fluide en minimisant les différences d'accélération d'un instant à un autre. $q(t)$ est le vecteur des variables articulaires du robot en fonction du temps et $\Gamma(t)$ le vecteur des couples appliqués aux articulations en fonction du temps. $IntegrationStep$ est le temps d'un pas de calcul. Il y a ici un abus de langage, car le terme *couple* est aussi utilisé dans le cas d'articulations prismatiques, cependant dans le cadre de ce stage, les articulations prismatiques ne sont pas utilisées.

1.3 MoGS

Le projet MoGS est composé d'une centaine de *classes* C++ qui forment plusieurs modules qui fonctionnent comme des librairies. Ce stage a nécessité l'utilisation des modules de simulation, de la **RBDL**[3] - *Rigid Body Dynamic Library* - du module de collision et des modules d'optimisation.

Le module de simulation - *le simulateur* - sert à simuler un robot dans un environnement virtuel 3D composé d'objets 3D. Le simulateur est le garant du respect des lois de la physique.

Le module collision sert à calculer les points de collision lorsque deux objets entrent en collision dans le simulateur.

La RBDL donne la représentation des robots et de l'environnement et possède aussi les fonctions qui représentent la cinématique et la dynamique des solides.

Le module d'optimisation sert à optimiser un mouvement d'un robot en fonction d'un critère.

Lengagne a implémenté dans MoGS le simulateur en suivant les travaux de la thèse de Chardonnet [1] et l'optimisation selon le modèle dynamique inverse [5]. Néanmoins, le simulateur était incomplet, en particulier, l'environnement ne pouvait être qu'un sol à l'horizontal sans reliefs, et le seul objet géométrique utilisable était la "box". Bien que les objets "sphere" et "cylinder" aient été implémentés, ils n'étaient pas utilisables car les vecteurs normaux aux faces des objets 3D n'étaient pas implémentés.

3. `http://rbdl.bitbucket.org`

Le modèle dynamique inverse permet d'obtenir les couples à appliquer aux articulations en fonction de la posture du robot, de la vitesse et de l'accélération angulaires des articulations ainsi que des forces extérieures.

$$\Gamma(t) = MDI(q(t), \dot{q}(t), \ddot{q}(t), f\_ext(t)) \tag{1.3}$$

Dans un premier temps, j'ai dû me familiariser avec les différents outils nécessaires au bon déroulement du stage et terminer le simulateur 3D en transformant les objets 3D *box*, *sphere* et *cylinder* en object 3D Mesh [4]. D'autres modifications du code ont été nécessaires pour rendre le simulateur compatible avec l'algorithme d'optimisation. Dans un second temps, j'ai implémenté l'algorithme d'optimisation selon le modèle dynamique direct, implémentation décrite dans l'article de Diehl [2]. Ce modèle permet d'obtenir les accélérations angulaires des articulations en fonction de la posture du robot, de la vitesse et des couples des articulations ainsi que des forces extérieures.

$$\ddot{q}(t) = MDD(q(t), \dot{q}(t), \Gamma(t), f\_ext(t)) \tag{1.4}$$

Nous avons alors pu effectuer quelques tests pour obtenir des résultats expérimentaux en comparant les deux algorithmes d'optimisation.

4. Un mesh ou maillage est un objet tridimensionnel constitué de sommets, d'arêtes et de faces organisés en polygones sous forme de fil de fer dans une infographie tridimensionnelle. Les faces se composent généralement de triangles. [Wikipédia]

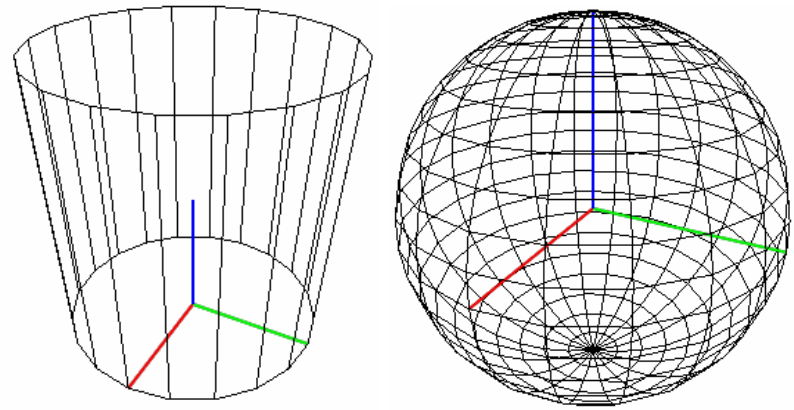
2 Le simulateur 3D

Avant de pouvoir implémenter l'optimisation, il a fallu terminer le simulateur.

2.1 Transformation des objets 3D en objets 3D Mesh

La raison pour laquelle il n'est possible d'utiliser que l'objet 3D *box* est que le vecteur normal des faces d'un object 3D est codé en dur selon l'axe z du repère *world*. Les calculs des forces liées aux collisions ne seront alors pas réalistes à moins que la collision soit selon l'axe z du repère *world* et selon le sens suivant : du haut vers le bas, soit le robot tombant sur un sol horizontal. La conséquence d'avoir un vecteur normal codé en dur selon l'axe des z est que lors des collisions, par exemple un robot qui tombe sur un sol en pente, la force de réaction du sol sur le robot est selon l'axe des z plutôt que d'être normale à la pente, par conséquent le calcul de la force de frottement est faussé et le robot peut basculer sur le sol en pente plutôt que glisser. Afin de corriger cela, il faut calculer le vecteur normal de chaque face d'un object 3D, et pour cela il faut convertir les objets 3D en object 3D Mesh. J'ai utilisé une décomposition en *triangle* des objets *box*, *cylinder* et *sphere* puis ajouté pour chaque triangle le vecteur normal correspondant par l'intermédiaire d'un produit vectoriel et d'un produit scalaire et les vecteurs normaux aux sommets du *triangle* dont le calcul est expliqué peu après.

La *box* est constituée de six rectangles, et chacun de ces six rectangles est décomposé en deux *triangles*. Le *cylinder* est constituée de deux disques et de n rectangles. Chaque disque est décomposé en n *triangles* et les n rectangles sont décomposés en deux *triangles*. La *sphere* est décomposée en $n \times n$ trapèzes, chacun de ces trapèzes est décomposé en deux *triangles*.



source : openclassrooms [1]

1. `http://fr.openclassrooms.com/informatique/cours/creez-des-programmes-en-3d-avec-opengl/les-quadriques-1`

Les vecteurs normaux aux sommets des *triangles* sont la somme des vecteurs normaux aux faces des triangles qui possèdent ce sommet, puis ils sont normalisés afin d'avoir une norme unitaire.

2.2 Gestion des templates pour les calculs de dérivées

Parmi les autres modifications du code nécessaires pour rentre le simulateur compatible avec l'algorithme d'optimisation, il y avait la gestion des templates de C++ pour les calculs de dérivées. En effet, l'optimisation doit résoudre un système d'équations non linéaires. Pour cela j'ai utilisé **IPOPT** [2] - *Interior Point OPTimizer* - qui demande le calcul de la simulation pour déplacer le robot d'une posture à une autre en un temps *IntegrationStep* donné ainsi que le gradient de ce calcul.

Certains fragments du code du simulateur utilisent le module **FADBAD** [3] - *Forward Automatic Differentiation Backward Automatic Differentiation* - qui calcule automatiquement le gradient d'un calcul donné selon un type C++ donné, en particulier le type *double*. Malheureusement le module FADBAD ne peut calculer automatiquement la dérivée d'un calcul qui utilise déjà le module FADBAD. Il faut alors supprimer tous les fragments de code qui utilisent ce module et les remplacer par un calcul analytique des dérivés. De même pour le solver d'Eigen qui calcule les valeurs propres d'une matrice, EigenSolver [4], qui n'est pas compatible avec le module FADBAD.

Une fois cela fait, et à l'instar de la RBDL, il a fallu "templatiser" le simulateur afin qu'il puisse accepter le type C++ *double* et le type FADBAD *F<double>* - *F* pour *forward* -, en fonction de s'il faut fournir le calcul de la simulation ou bien la dérivée, au module d'optimisation. Puis j'ai créé un nouvel object C++ que l'on peut appeler via la fonction *Integrate* pour obtenir le calcul de la simulation et la fonction *FIntegrate* pour obtenir le calcul du gradient de la simulation.

2.3 Ajuster le temps d'attente dans le simulateur

Pour simuler le temps réel lors de la visualisation interactive du robot, le simulateur effectue les calculs pour déplacer le robot et attend $40ms$ à l'aide de la fonction *usleep* de C++ pour incrémenter de $40ms$ le temps de simulation. De fait, le simulateur perd le temps du calcul de simulation par rapport au temps réel qui s'écoule : j'ai déduit du temps d'attente le temps de calcul si celui-ci est inférieur à $40ms$, sinon, il n'y a plus de temps d'attente.

2. `https://projects.coin-or.org/Ipopt`
3. `http://www.fadbad.com`
4. `http://eigen.tuxfamily.org/dox/classEigen_1_1EigenSolver.html`

3 Optimisation avec le modèle dynamique direct

Pour l'algorithme d'optimisation utilisant le modèle dynamique direct, j'ai utilisé IPOPT qui est un logiciel dédié à l'optimisation non linéaire de grande envergure. IPOPT est conçu pour trouver une solution localement optimale d'un problème mathématique d'optimisation de la forme :

Fonction objective :

$$f(x), x \in \mathbb{R}^n \tag{3.1}$$

Sous les contraintes :

$$g\_L \leq g(x) \leq g\_U \tag{3.2}$$

$$x\_L \leq x \leq x\_U \tag{3.3}$$

avec x le vecteur des paramétres du problème d'arité n, f la fonction objective à minimiser : $f(x) : \mathbb{R}^n \to \mathbb{R}$, g la fonction des contraintes : $g(x) : \mathbb{R}^n \to \mathbb{R}^m, n \geq m$. g_L et g_U les valeurs minimales et maximales des contraintes. Les fonctions $f(x)$ et $g(x)$ peuvent être non linéaires et non convexes mais deux fois continûment dérivables.

3.1 Implémentation avec IPOPT

IPOPT a besoin qu'on lui fournisse les paramètres du problème, la fonction qui calcule les critères, soit la fonction objective à minimiser, ainsi que la fonction qui calcule le gradient du critère. IPOPT a aussi besoin de la fonction qui calcule les contraintes et de la fonction qui calcule le gradient des contraintes.

3.1.1 Interface entre MoGS et IPOPT

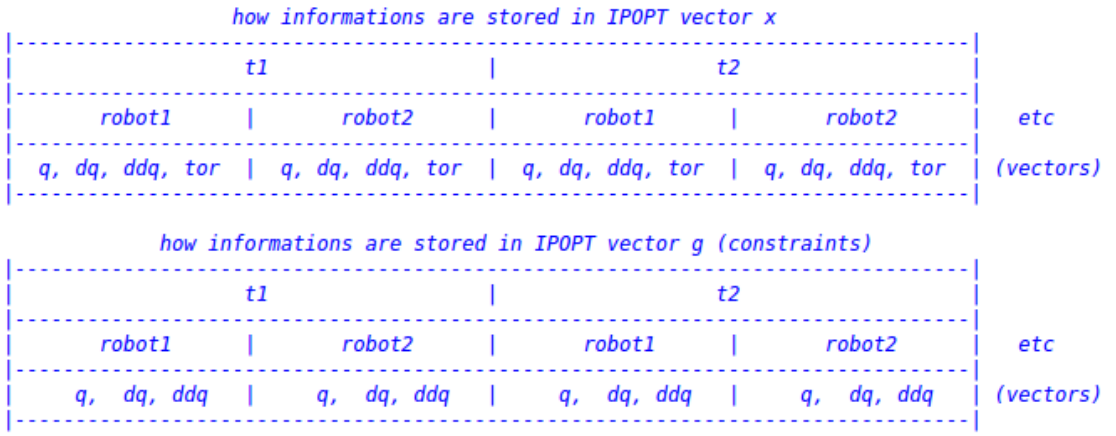
Il faut créer l'interface entre IPOPT et MoGS en fournissant les bons paramètres à IPOPT. Le vecteur x - *un tableau C++* - contient les paramètres du problème selon un arrangement que j'ai défini : x est segmenté en autant d'étape de calcul Nb_Step, chaque étape de calcul est segmentée en autant de robots considérés, et chaque robot possède tous ses paramètres q, $\dot{q}$, $\ddot{q}$, Γ.

$$Nb\_Parametre = Nb\_Step \times \sum_{r=0}^{Nb\_Robot-1} (Nb\_Articulation(r)) \times 4 \tag{3.4}$$

Pour les contraintes, il n'y en a que sur q, $\dot{q}$ et $\ddot{q}$ et la première étape de calcul est fixe, l'arrangement est similaire à celui du vecteur x, sauf qu'il n'y a pas Γ.

$$Nb\_Contrainte = (Nb\_Step - 1) \times \sum_{r=0}^{Nb\_Robot-1} (Nb\_Articulation(r)) \times 3 \tag{3.5}$$

Le schéma suivant illustre cet arrangement :



Dans les paramètres, les postures ne sont fixées que pour la première et la dernière étape du mouvement, définies par l'utilisateur. Toutes les autres valeurs sont mises à zéro.

Concernant les valeurs minimales et maximales du vecteur x, cela est en fonction des capacités du robot considéré, qui pour chaque articulation possède ses propres caractéristiques.

3.1.2 Calculs des critères

Le calcul des critères se fait selon les équations (1.1) et (1.2) et le calcul des gradients en découle.

Algorithm 1 TorqueSquare

Require: x

for s in steps **do**
  for r in robots **do**
    for a in $articulations(r)$ **do**
      $TorqueSquare = TorqueSquare + x_\Gamma(s, r, a)^2$
    end for
  end for
end for
return $TorqueSquare \times IntegrationStep$

La valeur de retour de ces algorithmes est fournie à IPOPT, qui s'en sert pour trouver un optimum local. Les calculs des gradients sont fait de façon analytique puisqu'ils sont assez simple à réaliser.

Et pour le critère Jerk :

Algorithm 2 Jerk

Require: x

for s in steps **do**
  for r in robots **do**
    for a in $articulations(r)$ **do**
      $Jerk = Jerk + (x_{\ddot{q}}(s, r, a) - x_{\ddot{q}}((s+1), r, a))^2$
    end for
  end for
end for
return $Jerk \times IntegrationStep$

Avec *steps* le nombre d'étape de calcul de la simulation à optimiser, *robots* le nombre de robot considéré, $articulations(r)$ le nombre d'articulation du robot r et $x_q(s, r, a)$ qui renvoie la valeur q du vecteur x qui correspondant à l'étape de calcul s, du robot r et de son articulation a. Comme q est un vecteur, a est la position de la valeur recherchée dans le vecteur q. Il en est de même pour $x_{\ddot{q}}(s, r, a)$ et $x_\Gamma(s, r, a)$.

3.1.3 Calculs des contraintes

Concernant les contraintes, le calcul est un peu plus compliqué. Il faut récupérer les valeurs dans x pour reconstituer les vecteurs q, $\dot{q}$, $\ddot{q}$ et Γ afin de les envoyer à la fonction d'intégration qui calcule les nouvelles valeurs de ces vecteurs en respectant les lois de la physique et en prenant en compte l'environnement et les forces extérieures pour ensuite faire la différence avec la valeur du pas d'intégration suivant.

Algorithm 3 Constraints g

Require: x

int $cpt = 0$
for s in steps **do**
  for r in robots **do**
    for a in $articulations(r)$ **do**
      $q(r, a) = x_q(s, r, a)$
      $\dot{q}(r, a) = x_{\dot{q}}(s, r, a)$
      $\ddot{q}(r, a) = x_{\ddot{q}}(s, r, a)$
      $\Gamma(r, a) = x_\Gamma(s, r, a)$
    end for
  end for
  $q, \dot{q}, \ddot{q} = Integrate(q, \dot{q}, \ddot{q}, \Gamma, IntegrationStep)$
  for r in robots **do**
    for a in $articulations(r)$ **do**
      $g(cpt{+}{+}) = q(r, a) - x_q((s+1), r, a)$
      $g(cpt{+}{+}) = \dot{q}(r, a) - x_{\dot{q}}((s+1), r, a)$
      $g(cpt{+}{+}) = \ddot{q}(r, a) - x_{\ddot{q}}((s+1), r, a)$
    end for
  end for
end for
return g

Les valeurs minimales et maximales des contraintes sont fixées à 0, IPOPT doit ajuster les valeurs des paramètres des robots afin que toutes les valeurs du vecteur g soient nulles.

La fonction *Integrate* est la fonction qui calcule les différents paramètres du robot en fonction de q, $\dot{q}$ et $\ddot{q}$ et qui a été "templatisée" lors de la première partie de ce stage. Il est alors possible d'appeler la fonction *FIntegrate - F pour forward* - qui calcule le gradient de la fonction *Integrate*. Il suffit alors de remplacer la ligne 11 de l'algorithme 3 par :

$$q, \dot{q}, \ddot{q} = FIntegrate(q, \dot{q}, \ddot{q}, \Gamma, IntegrationStep) \tag{3.6}$$

et de stocker les résultats dans le vecteur *values* de IPOPT.

3.1.4 Les autres paramètres

IPOPT requière bien évidemment plus de paramètres, telle que la fonction qui fixe les minimaux et maximaux des valeurs de x et des contraintes g, cette fonction servira notamment à fixer des points de passage d'un corps du robot. Il faut aussi écrire la fonction qui permet de gérer les calculs de gradient, et déterminer les indices des variables qui sont stockées dans les vecteurs x et g pour indiquer à IPOPT quelle variable sert à chaque calcul de gradient.

3.1.5 Les paramètres utilisateur

Après avoir créé un environnement, un ou des robots via un fichier xml, l'utilisateur peut choisir plusieurs paramètres que le robot doit respecter, des points de passages d'un corps du robot ou bien retirer ou prendre en compte une limite d'une ou de plusieurs articulations du robot. L'utilisateur doit aussi imposer une posture initiale et finale au robot - *c'est à dire le vecteur q* - et peut choisir si le mouvement est cyclique. Dans ce cas, la posture initiale est identique à la posture finale et il en est de même pour la vitesse et l'accélération des articulations. Il existe d'autres options que l'utilisateur peut régler.

Toutes ces options supplémentaires sont ajoutées dans les calculs des critères et des contraintes. En ce qui concerne l'implémentation de ces options, dans la mesure où elles découlent des explications précédentes, il est laissé le soin aux lecteurs de se référer au code du projet MoGS [1].

3.2 Premiers résultats

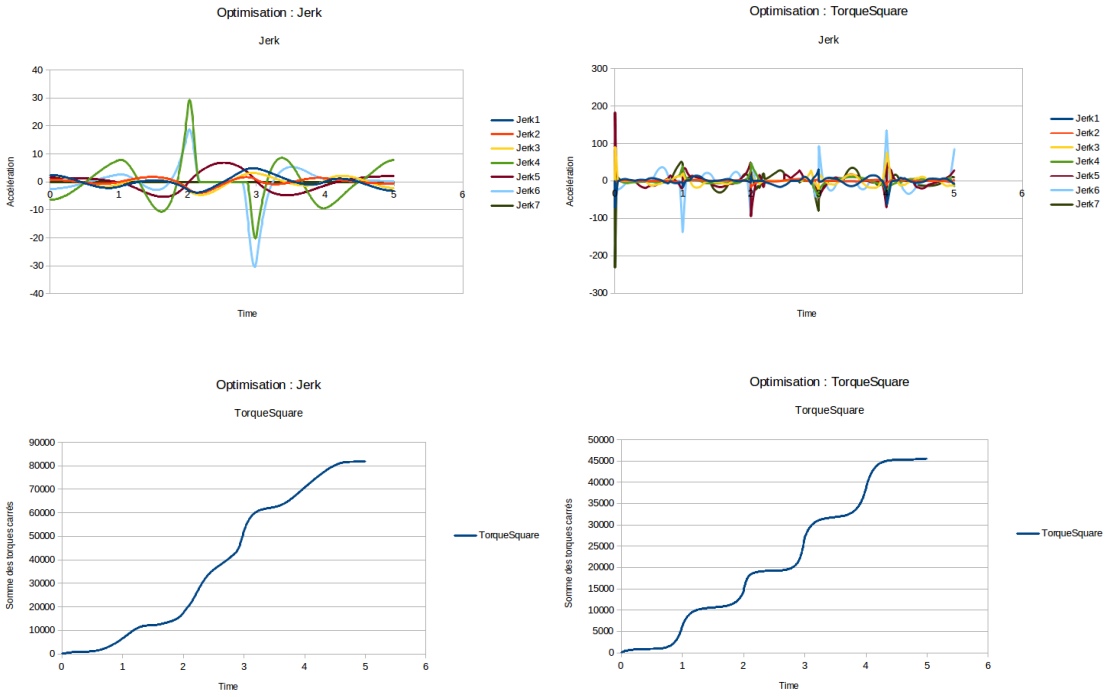
3.2.1 Vérification des critères

Une fois que l'implémentation de l'algorithme d'optimisation a été réalisée, nous avons mis en oeuvre quelques vérifications. Pour cela Lengagne a ajouté le robot Kuka [2] dans MoGS. Kuka est un robot bras à sept degrés de libertés à base fixe. Je lui ai fait faire un mouvement simple où le corps effecteur du robot doit passer par quatre positions. Le seul paramètre qui est modifié lors des essais est le critère d'optimisation, dans un premier temps, j'ai utilisé le critère Jerk, puis le critère TorqueSquare.

Nous observons dans les graphiques ci-dessous que les résultats semblent être les résultats attendus, en effet, lorsque le critère est Jerk, il n'y a pas de variations "brutales" de l'accélérations des sept articulations du robot Kuka, les courbes sont bien arrondies, et

1. `https://github.com/lengagne/MoGS`
2. `http://www.kuka-robotics.com/fr/`

la somme des couples des sept articulations est de 81790 newtons contre 45585 lorsque le critère est TorqueSquare, où nous observons que les courbes d'accélérations sont plus saccadées.



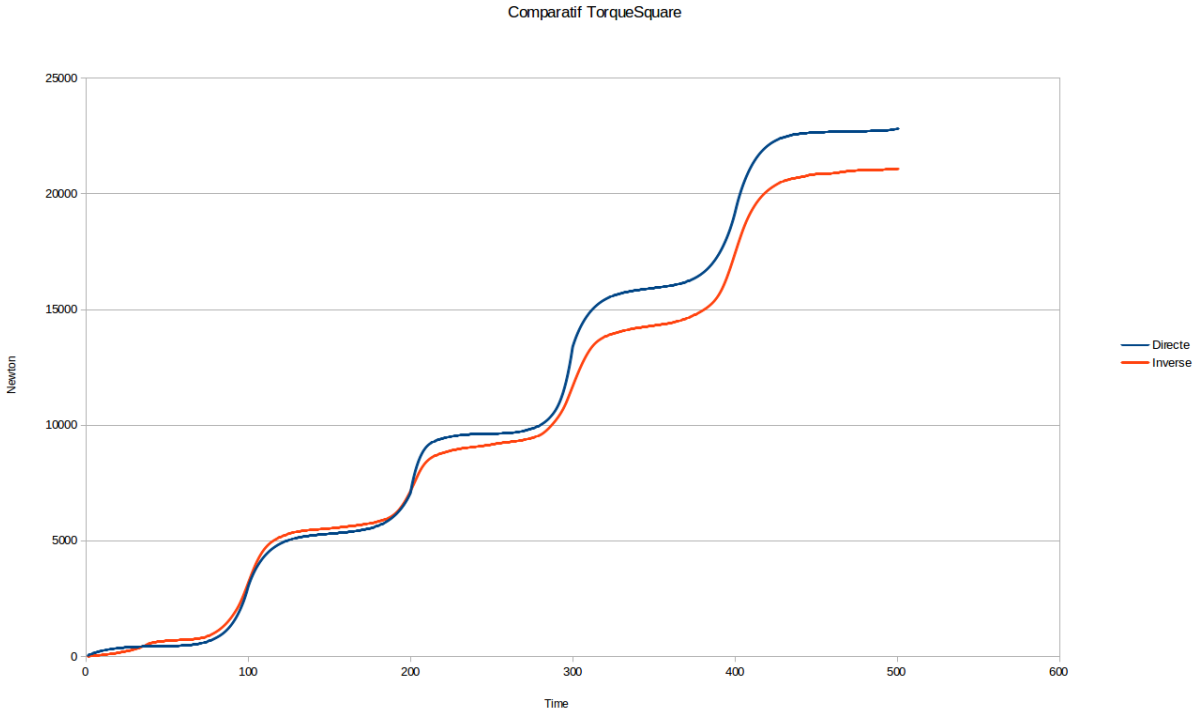
3.2.2 Premier comparatif entre les deux modèles dynamiques

Ensuite j'effectue quelques tests afin de comparer les modèles dynamiques direct et inverse sur le critère TorqueSquare. Malheureusement, nous ne pourrons pas réaliser le même test avec le critère Jerk dans la mesure où ce critère n'a pas été implémenté pour le modèle dynamique inverse.

Pour ce test, nous avons fait faire au robot Kuka exactement le même mouvement avec les mêmes paramètres.

Nous avons pu constater que contrairement à ce que prédisait la littérature [2], le modèle inverse avec un total de 21090 newtons se révèle plus performant que le modèle direct avec 22808 newtons, soit 8,14% de plus.

Cependant nous avons aussi constaté que le modèle inverse requiert 2261 - *1599s* - itérations d'IPOPT contre 577 - *662s* - pour le modèle direct. Nous observons que le calcul du modèle direct amène plus rapidement et en moins d'itération à une solution localement optimale mais le calcul d'une itération est plus long à réaliser. Pour l'implémentation du simulateur selon le modèle dynamique direct, Chardonnet [1] explique que le calcul d'une matrice Λ est coûteux d'un point de vue complexité et il propose un nouvel algorithme bien plus performant.



3.2.3 Résultat inattendu

Lengagne avait réalisé des tests avec le modèle inverse sur le robot HRP2, des vidéos sont visibles sur Youtube. Nous souhaitons alors réaliser des tests avec un robot à base flottante. Nous utilisons un petit robot “Hopping Robot” assez simple, composé de quatre corps : une tête, une jambe composée elle-même de deux corps, et un pied. J’ai programmé un mouvement simple où Hopping Robot doit aller d’un point à un autre sans entrer en collision avec son environnement et l’algorithme d’optimisation a trouvé une solution assez rapidement. Puis je l’ai obligé, en modifiant les paramètres de l’optimisation, à toucher le sol, le robot étant en position de départ au dessus du sol, il devait tomber et rester sur le sol.

Contrairement à nos attentes, l’algorithme d’optimisation n’a pas trouvé de solution et nous a indiqué que le problème est infaisable.

Avec Lengagne nous pensons que le problème est lié à la méthode par descente de gradient d’IPOPT. Au moment où Hopping Robot touche le sol, IPOPT va rechercher une solution optimale en faisant varier plusieurs paramètres de son vecteur x. Si parmis ces paramètres, la posture du robot vient à être en collision avec l’environnement, la fonction *Integrate* va immobiliser le robot car cette fonction est garante des lois de la physique, et l’optimisation ne pourra plus se terminer. Le problème réside alors dans le contact entre le ou les corps du robot avec un autre corps.

4 Conclusion

4.1 Perspectives

Nous avons été un peu déçus que notre méthode ne fonctionne pas lorsqu'il y a contact entre le robot et son environnement, nous empêchant de comparer le modèle direct et inverse dans tous les cas de figure, cependant j'ai pu implémenter le noyau qui permettra à Lengagne de comparer plus sérieusement les deux modèles dynamiques.

Je n'ai pas eu le temps d'implémenter la méthode de Chardonnet pour le calcul de la matrice Λ ce qui améliorerait la complexité de la fonction *Integrate*. Par ailleurs, le multi-threading est implémenté pour l'algorithme d'optimisation selon le modèle inverse, en modifiant le code de l'algorithme d'optimisation selon le modèle direct, il serait aussi possible d'y ajouter le multi-threading.

Evidemment, la recherche d'une nouvelle méthode qui pourrait gérer les contacts entre le robot et son environnement serait une priorité.

4.2 Ce que ce stage m'a apporté

Ce stage a été très riche pour moi, j'ai pu y découvrir un domaine qui me fascine et pour lequel j'éprouve beaucoup de curiosité depuis longtemps sans pouvoir réellement m'y impliquer. Grâce à cette expérience, je connais aujourd'hui la base de la robotique, comment sont représentés les robots en informatique et quels sont les grands algorithmes qui sont au coeur de ce domaine. Même si j'ai bien conscience de n'éffleurer que le sommet de l'iceberg de ce domaine, j'ai envie d'en connaitre plus, et ce stage me confirme que je souhaiterai, si c'est possible, avoir un travail en lien avec la robotique.

C'est aussi la première fois que je travaille dans un laboratoire de recherche, et j'ai pu, durant ce stage, me faire une première expérience de ce qu'est la recherche tout en travaillant au sein d'une équipe, même si dans cette équipe nous n'étions que deux. Cela m'a aussi obligée à utiliser Github, et j'ai découvert d'autres outils tels que Valgrind, IPOPT, le langage C++ que j'ai utilisé pour la première fois ici et BibTex qui me sert à écrire ce rapport.

Bien que cela soit particulier, le fait que la méthode utilisée ne fonctionne pas lors des contacts entre le robot et son environnement est aussi une expérience enrichissante, car dans la recherche tout ce que je serai amenée à faire ne sera pas forcément un succès. Il est dommage que ce stage ne dure que deux mois : plus de temps nous aurait permis de réfléchir à une nouvelle méthode et tenter de trouver une solution à ce problème.

Bibliographie

[1] Jean-Rémy Chardonnet. *Modèle dynamique temps-réel pour l'animation d'object poly-articulés dans les environnements contraints, prise en compte des contacts frottants et des déformations locales : application en robotique humanoïde et aux avatars virtuels.* PhD thesis, Université Montpellier II, Juin 2009.

[2] M. Diehl, H.G. Bock, H. Diedam, and P.-B. Wieber. *Fast Direct Multiple Shooting Algorithms for Optimal Robot Control*, pages 65–93. Springer Berlin Heidelberg, 2006.

[3] Wisama Khalil and Etienne Dombre. *Modeling, Identification and Control of Robots, 3rd.* Taylor and Francis, Inc. Bristol, PA, USA, 2002.

[4] Sébastien Lengagne. Youtube profile. https ://www.youtube.com/user/lengagne2583.

[5] Sébastien Lengagne, Joris Vaillant, Eiichi Yoshida, and Kheddar Abderrahmane. Generation of whole-body optimal dynamic multi-contact motions. *International Journal of Robotics Research*, 32 :1104–1119, August 2013.