Rapport de fin de stage internationnal

Optimisation du mouvement du violoniste afin de minimiser la fatigue musculaire

Étudiant :
Théophile Gousselot EI18

Tuteur pédagogique :
David Moreau

Tuteur laboratoire :
Benjamin Michaud

Directeur du laboratoire :
Mickaël Begon

Effectué du 18 Mars 2020 au 04 Août 2020

En télétravail depuis Montréal

Résumé

Le présent rapport s'intègre au sein d'une thèse portant sur l'optimisation du mouvement du violoniste afin de minimiser la fatigue musculaire. Ma contribution à ce projet fut parallèlement partagée entre le développement d'un outil générique de contrôle optimal : *BiorbdOptim*, et l'optimisation du mouvement du violoniste grâce à cet outil.*BiorbdOptim* est une interface entre un utilisateur et un solveur de problèmes de contrôle optimal. Cet outil est développé par et dans une première finalité, pour les chercheurs du laboratoire *s2mlab*. L'utilisation de *BiorbdOptim* et d'un modèle, issue de la librairie *BIORBD*, des membres supérieurs humains portant un violon et son archet, m'a permis d'écrire un problème de contrôle optimal. Ce problème s'est progressivement raffiné, en intégrant successivement : le mouvement de tiré-poussé via des couples articulaires, le maintient du contact entre l'archet et la corde, le respect du parallélisme entre l'archet et le chevalet, le maintient du même angle pour l'archet, une force de contact entre l'archet et le violon, les muscles aux alentours du bras, une modélisation de fatigue musculaire, plusieurs mouvement successifs de tiré-poussé. La finalité de cette modélisation est de parvenir à déterminer le plus fidèlement possible, et par du contrôle optimal le mouvement minimisant la fatigue musculaire dans le but d'aider à l'enseignement du violon.

Abstract

This report is part of a thesis on the optimization of the violinist's movement in order to minimize muscular fatigue. My contribution was shared between the development of a generic optimal control tool: *BiorbdOptim*, and the optimization of the violinist's movement thanks to this tool. *BiorbdOptim* is an interface between a user and a solver of optimal control problems. Further, it is developed by and also for the researchers of the s2mlab laboratory. The use of BiorbdOptim and a model of human upper limbs carrying a violin and its bow, from the *BIORBD* library, allowed me to write an optimal control problem. This problem has been progressively enhanced, successively integrating these concepts : the back and forth movement via articular couples, maintaining contact between the bow and the string, respecting parallelism between the bow and the bridge, maintaining the same angle for the bow, a contact force between the bow and the violin, the muscles around the arm, a model of muscular fatigue, several successive pull-push movements. The purpose of this modeling is to determine as accurately as possible, and through optimal control, the movement that minimizes muscle fatigue in order to assist in the teaching of the violin.

Remerciements

Mickaël BEGON Benjamin MICHAUD David MOREAU

Paul WEGIEL

Ariane DANG Kilpéric NOUVELLET Quitterie BOISSÉ Amedeo CEGLIA Bailly FRANCOIS Léa SANCHEZ

Eve CHARBONNEAU Najoua ASSILA

Clara ZIANE

Fabien DEL MASO Béatrice MOYEN-SYLVESTRE Anne Laure MÉNARD + tout le laboratoire

Klara NOVOTNA Frédérique Alexandre MICHAUD : Les évènements culturels rythmant les été de Montréal -> FAM

Bruno MONSARRAT Les familles SOLAR

Sylvie et Philippe GOUSSELOT

Camille WILHELM

Á mes amis musiciens, et à ceux qui aiment la musique...

Figure 1: Un violon muni d'une mentonnière .

La mentonnière, utilisée pour la première fois au début du XIX$^{\text{ème}}$ siècle, est une des premières adaptation du violon au corps du musicien, elle sépare la sueur du violoniste du violon afin de ne pas altérer le vernis à sa surface.

Â l'image de la mentonnière, mon travail est de perfectionner le "couple" du musicien et de son violon.

0. Sommaire

0. Glossaire

ACADOS — 17

Anaconda — 9

Biorbd — 11, 13, 14, 18, 19, 26, 33

BiorbdOptim — 9, 16, 18, 19, 23–25, 30–32, 36, 38–40, 75

BiorbdViz — 13–15, 18, 19, 21, 38–40, 75

CasADi — 13, 19, 29, 39

Eigen — 13, 39

GitHub : service de gestion de développement de logiciels individuel ou collectif — 6, 13, 14, 21

GitKraken : interface graphique pour logiciels de gestion de développement de logiciels — 6

IEEE Industrial Electronics Society — 23

Ipopt — 17, 19, 26, 28, 29, 36–39

Linux — 9

Matplotlib — 18, 37, 40, 76

MoCo — 23

NumPy — 13

Pickle — 39

pomodoro : technique de gestion du temps avec alternance de temps de travail (25 minutes) et de pauses (5 minutes) — 7

PyCharm — 9, 18

Qt — 9

SciPy — 38

Slack : plate-forme de communication collaborative — 6

Teams : application de communication collaborative — 6

Zoom : service de téléconférence — 6

Part I

Introduction

1. Stage international

Mon cursus à l'École des Mines de Saint-Étienne permet au deuxième semestre de deuxième année de réaliser un *projet industriel* sur le campus ou un *stage international*. J'ai souhaité partir à l'étranger, l'expérience associée me semblant davantage complète et engageante. La dimension professionnelle du stage, mettant en jeu des interactions sociales différentes de celles rencontrées au sein d'associations et de groupes de projets étudiants en est la première raison. Mon choix s'est porté sur le Canada et la province du Québec. Principalement motivé par le laboratoire de simulation et modélisation du mouvement (*S2M*), la culture québecoise et les splendides paysages ne m'ont que confortés dans ce choix.

Mme Camille Wilhelm, étudiante de la promotion précédente ayant réalisé un stage international au laboratoire S2M m'en a fait une présentation globale : activités de recherches, état d'esprit, encadrement et méthodes de travail. J'ai ensuite échanger avec M. Benjamin Michaud, ancien tuteur de Mme Camille Wilhelm. La discussion a porté sur ses activités de recherches, ses besoins et mes compétences. Cette série d'échange, entre autres, m'a permis de prendre conscience de l'intérêt de s'essayer à un travail de recherche et non industriel, dans l'optique d'appréhender fidèlement ces deux mondes.

2. Laboratoire simulation et modélisation du mouvement

2.1 Histoire

En 2008, M. Mickaël Begon atteint le profession de professeur adjoint, ce qui implique la création de son laboratoire nommé Simulation et Modélisation du Mouvement dont l'objectif est la recherche en biomécanique et la formation de personnel hautement qualifié en kinésiologie. D'abord installé dans les locaux du Centre de réadaptation Marie-Enfant à Montréal, le laboratoire migre à Laval en 2011, lors de l'inoguration du nouveau campus de l'université de Montréal.

2.2 Domaine d'expertise

Le laboratoire S2M est porté vers le développement de nouvelles connaissances sur la motricité humaine à partir de mesures et de modèles de simulation pour des applications en réadaptation, prévention des blessures et amélioration de la performance sportive et artistique. On peut citer des projets phares comme l'optimisation dynamique d'acrobaties, l'optimisation du geste violonistique et pianistique, ou encore la conception d'orthèses plantaires personnalisées. Les projets se rapportent tous à une thématique musicale, sportive ou ergonomique. Le laboratoire S2M fonctionne majoritairement grâce aux subventions des organismes du Québec et du Canada comme la fondation Canadienne pour l'innovation ou l'Institut de Recherche en Santé et Sécurité au travail. Il est aussi en relation avec des partenaires industriels.

Le laboratoire bénéficie d'équipement de pointe pour les mesures biomécaniques:

- Un ergomètre isocinétique.
- Un système optoélectronique de 18 caméras.
- Un système d'électromyographie (EMG intramusculaire et de surface).
- Des plateformes de force sur une piste de marche.
- Un tapis roulant pour la marche.
- Un piano acoustique instrumenté.

2.3 Effectif

Supervisé par M. Mickaël Begon, M. Fabien Del Maso et M. PhilippeDixon, le laboratoire de recherche universitaire S2M est un outil pédagogique de formation étudiante. L'effectif de recherche regroupe en moyenne 25 étudiants. Il se renouvelle très régulièrement, allant du stagiaire de quelques mois au doctorat de plusieurs années, à titre d'exemple, on peut dénombrer 6 arrivants lors de mon stage. Ce renouvellement rempli le premier objectif d'un laboratoire universitaire : proposer à un maximum d'étudiants de vivre une expérience scientifique afin d'insuffler un goût pour la recherche, amenant certains à faire suite à leur stage avec une thèse ou un doctorat.

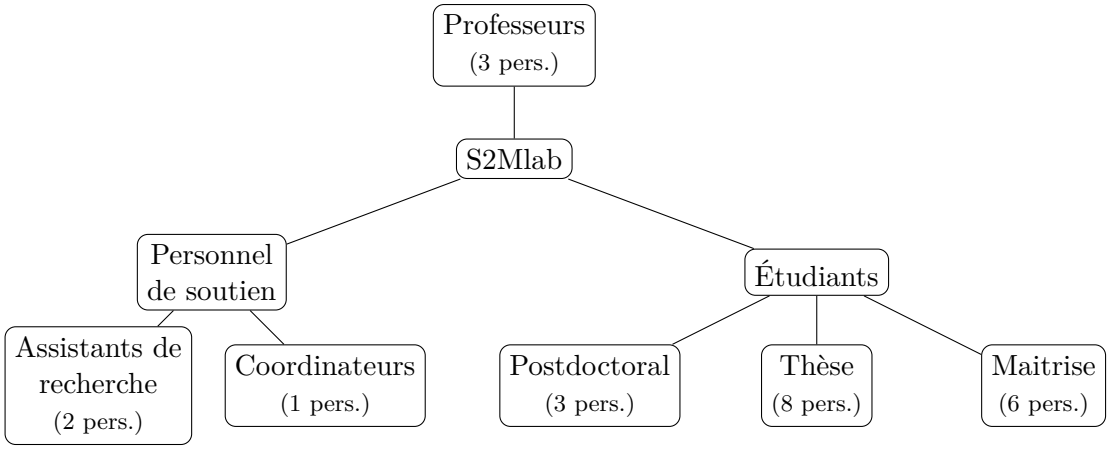


Figure 2.1: Organigramme du laboratoire au 21/07/20. On dénombre en moyenne 8 nouveaux stagiaires par trimestre.

Le laboratoire est découpé en plusieurs thématiques; en tant que stagiaire de Benjamin Michaud, j'ai rejoint l'équipe de *optimal control – commande optimale* en français. Composé d'une dizaine d'étudiants, l'équipe se réunit tous les mardi matin lors d'un tour de table où chacun expose ses avancées et ses problèmes. L'aspiration de cette réunion est double, débloquer rapidement les étudiants qui sont en difficultés et s'assurer d'une collaboration efficace : jouir de l'expérience des autres pour ne pas essayer de réinventer ce qui existe déjà. M. Mickaël Begon, professeur agrégé, participe à cette rencontre et apporte, entre autre, un sens de l'efficacité, aiansi qu'une vision de manager. Il est important de retenir que S2Mlab n'existe que par la présence d'un professeur, ici Mickaël Begon. En France, les professeurs vont majoritairement rejoindre un laboratoire existant.

2.4 Recherche scientifique

Il est important de différencier la recherche scientifique de la recherche industrielle. Si la première se veut collaborative et nécessite le concours de plusieurs laboratoire, la deuxième est par nature concurrentielle afin de commercialiser en premier des technologies et de déposer des brevets.

Un chercheur scientifique se doit donc de publier des articles afin de partager ses recherches à ses homologues. A travers ses articles, et les citations de ses articles, le professeur peut appuyer ses demandes de financement. La recherche, n'engendrant pas de profit direct, nécessite de faire appel à des acteurs extérieurs. Les organismes gouvernementaux assument en grande partie ce rôle

en finançant les projets du professeur, qui peut ainsi rémunérer ses étudiants et son personnel. Le professeur, reçoit son salaire par l'université à laquelle il est rataché.

3. Contexte

3.1 Covid-19

Lors de mon arrivé le 15/03/20, l'université de Montréal a fermé ses portes à la totalité de ses étudiants et chercheurs. Les locaux du laboratoire S2M ont ainsi été fermés, privant les chercheurs de matériel expérimental et de certaines ressources informatiques.

3.2 Télétravail

L'obligation de télétravail pour l'ensemble des membres du laboratoire a nécessité la mise en place de plusieurs outils informatiques :

Messagerie : *Slack* puis *Teams*.
Visioconférence : *Zoom*, comprenant une fonctionnalité de *remote control* : prise de contrôle à distance de l'ordinateur.

En complément de ceux existants :

Gestion de développement de logiciels : *GitHub* couplé avec *GitKraken*.
Répartition de taches : *trello*.
Partage de fichiers : *serveurs du laboratoire*.

En écartant *trello* dont l'utilisation est rendue caduc par le système d'*issue* proposé par *github*, on peut considérer que 3 logiciels de communication permettent de répondre aux besoins émanants de collaboration.

Mode	Temps	Type	Échéance	Logiciel
oral	long ($\simeq$ heure)	réflexions stratégiques/développement	long terme	*zoom*
écrit	court ($\simeq$ min)	aide rapide, mémorisation par écrit	court terme	*microsoft teams*
écrit	instantané	fichiers de programmation (**code**)	court terme	*github*, *gitkraken*
écrit	court ($\simeq$ min)	liste d'idées (**issue**)	long terme	*github*, *gitkraken*

S'il est pertinent de remarquer qu'un échange oral permet un débat profond et efficace autour d'une stratégie de développement, on peut regretter l'éphémérité de l'information. Il est possible d'y palier en inscrivant l'information essentiel sur teams, sur les issues de github ou au sein des lignes de codes.

La répartition des échanges est simple, néanmoins, elle implique, à chaque instant, de réfléchir au moyen de communication le plus adapté.

Avec du recul, il est aisé de détecter un choix de communication inadapté, par exemple une discussion sur teams avec un nombre de messages important, laisse présager qu'une vidéoconférence aurait été davantage efficace. En revanche, une vidéoconférence sans prise de note, c'est à dire n'aboutissant pas à une *issue*, un message ou un commentaire dans le code, laisse présager une perte d'information.

Enfin, il faut garder à l'esprit qu'un outil de collaboration est là pour économiser du temps et préserver l'information utile, ce qui revient à économiser du temps. L'objectif étant de se concentrer davantage sur la recherche. Il faut ainsi toujours veiller à rester vigilant et lucide face à l'utilisation faite des outils de communications.

3.3 Construction d'équipe

Le confinement étant de vigueur, les régulières activités de groupe du laboratoire ont été suspendues. Malgré cela, certaines activités ont pu se réinventer via logiciel de vidéoconférence :

- Séance de sport.
- Séance de yoga.
- Escape-game en ligne.

Au fur et à mesure de l'assouplissement du confinement, des activités extérieures de groupe ont pu voir le jour, dans le respect des distances et des gestes barrières :

- Séances de travail collectives en appliquant la technique *pomodoro* dans des parcs.
- Randonnées et séances sportives.

La finalité de ces temps partagés avec des collègues, est de casser la monotonie de télétravail depuis la maison. En télétravail, il est particulièrement complexe de dissocier les temps de travail des temps personnels. D'autant plus dans un cadre de recherche où l'avancement, et ainsi l'entrain varient. Il en résulte que mon temps de travail hebdomadaire est en moyenne supérieur à 35 heures par semaine, avec un temps de travail quotidien oscillant entre 6h et 9h.

Il s'est avéré que le simple fait de travail en visioconférence avec ses collègues, sans forcement discuter, permet au travers du bruit de fond et de leur présence de reconstituer un contexte de travail.

Part II

BiorbdOptim

4. Outils de modélisation numérique du mouvement

Avant de rentrer dans le vif du sujet, il est nécessaire de définir et décrire quelques notions et logiciels sans quoi on ne peut comprendre l'intérêt ou le fonctionnement de *BiorbdOptim*.

Le système d'exploitation *Linux* a été le premier outil utilisé par la quasi-totalité des contributeurs à BiorbdOptim, il est intéressant de noter que la couche de compatibilité *Windows Subsystem for Linux* a permis à quelques utilisateurs de travailler sur BiorbdOptim depuis le système d'exploitation *Windows*. Afin de pouvoir simplifier la gestion des paquets et faciliter le déploiement du logiciel, la distribution libre et open source Anaconda a été grandement utilisée.

Enfin, la programmation en tant que telle, a été conduite grâce grâce aux environnements de développement PyCharm pour le langage python, et Qt pour le langage c++. ces deux environnements sont complets, ils proposent entre autre, un débogueur graphique, des outils de publication de code sur github ainsi que des licences étudiantes gratuites.

4.1 Commande optimal

La commande optimale est une méthode utilisée dans le cas d'un problème de dynamique régit par des équations différentielles. L'objectif est d'emmener le système d'un état initial jusqu'à un état final tout en maximiser ou minimiser des indices de performances nommées "fonctions-objectifs" et en respectant des contraintes.

L'utilisation de commande optimal est particulièrement approprié lorsque les mouvements d'un système sont dynamiques. A titre d'exemple, on l'utilise en biomécanique pour modéliser les mouvements du corps humain. La notion dynamique, réside ici dans l'impossibilité pour le système nerveux à contrôler la position finale d'un membre, son contrôle se limite à émettre un signal au muscle par le système nerveux. Le mouvement du membre est alors régit par des contraintes tels que le fonctionnement du muscle ou la gravité.

Le principe consiste à agir sur les commandes (variables d'entrées) qui à partir des équations du système donnent les états différentiels du système dynamique. On vérifie ensuite si les contraintes sont respectées et si la valeur de la fonction objectif est plus petite qu'une limite pré-établie. Si c'est le cas, la valeur constitue la solution optimale, sinon on modifie les commandes selon un algorithme de gradient pour recommencer le processus.

Il est possible de schématiser le fonctionnement de la commande optimale comme suit :

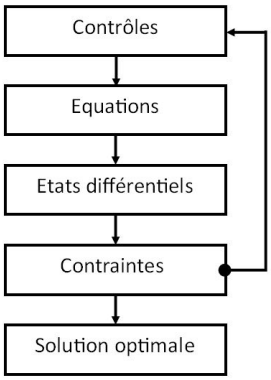


Figure 4.1: Principe de fonctionnement de la commande optimal.

La difficulté de la commande optimale réside dans la détermination de l'influence des modifications en entrée sur la fonction objectif tout en respectant les contraintes. Un problème de commande optimale possède *a priori* trois composantes :

- une fonction objectif à minimiser.
- un ensemble de contraintes à respecter *(pas nécessairement)*.
- un système dynamique.

Mathématiquement, on peut le définir génériquement comme suit :

Fonction objectif :

$$min\ J(t_F, x(t_F)) + \int_{t_0}^{t_F} \phi(t, x(t), u(t), p)dt$$

Contraintes dynamique :

$$\dot{x}(t) = f(t, x(t), u(t), p)$$

Contraintes de bornes :

$$x_{min} \leq x(t) \leq x_{max}$$

$$u_{min} \leq u(t) \leq u_{max}$$

$$x(t_0) = x_0$$

$$x(t_F) = x_F$$

avec :

J : fonction de Mayer.
ϕ **:** Lagrangien.
f : fonction décrivant la dynamique.
$\mathbf{t_0}$ **:** l'instant initial.
$\mathbf{t_F}$ **:** l'instant final.
p : les paramètres à optimiser lors de la résolution.
u : le vecteur des commandes.
x : le vecteur des états différentiels.
$\mathbf{u_{min}}$, $\mathbf{u_{max}}$ **:** les contraintes de bornes des commandes.

$\mathbf{x_{min}}$, $\mathbf{x_{max}}$: les contraintes de bornes des états différentiels.
$\mathbf{x_0}$, $\mathbf{x_F}$: états initiaux et finaux.

Le premier terme de la fonction objectif est un *terme de Mayer :* J, il traduit un objectif final, mesuré à t_F, tandis que l'intégrale, dit *terme de Lagrange :* ϕ, traduit un objectif sur l'ensemble du mouvement mesuré *à chaque instant* entre t_0 et t_F. La contrainte dynamique lie les états, commandes et paramètres aux états différentiels, en connaissant x_k et u_k avec $k \in [0, N-1]$ ainsi que p, on obtient $\dot{x}$, qui l'on peut intégrer sur $t_F - t_0$. Les contraintes de bornes c'est-à-dire les limites des variables d'états et des commandes permettent de définir le cadre du problème. Elles peuvent réduire le champ de recherche de l'optimiseur mais peuvent également rendre le problème numériquement impossible à résoudre, lorsqu'elles sont incompatibles.

L'optimiseur est le logiciel chargé d'agir sur les commandes pour aboutir à une solution. Les commandes peuvent être diverses, on peut citer, dans un contexte biomécanique et non-exhaustivement : accélérations, moments articulaires, activations musculaires, excitations neuronales.

4.2 Biorbd

La librairie Biorbd, codée au sein du laboratoire S2M en très grande partie par M. Benjamin Michaud. possède un grand nombre d'outils de dynamique et de modélisation. Le nom de cette librairie provient de l'association de :

bio : pour biomécanique, domaine d'étude du laboratoire
rbd : sigle de *rigid body dynamics*, correspond aux mécaniques mises en jeu, dans la dynamique des corps rigides. Biorbd s'appuie sur la bibliothèque de fonction de calculs RBDL (Rigid Body Dynamics Library).

La librairie gère la création et l'accès à un modèle biomécanique stocké sous le forme d'un fichier ayant un extension *.bioMod*. Les caractéristiques du modèles y sont définies avec une syntaxe précise. L'exemple ci-dessous décrit un cube dans l'espace nommé ayant :

- un *segment base : "Ground"* immobile faisant office de repère global, , il se caractérise par :
  - un *repère : "m1"* fixés sur le segment Ground aux coordonnées $(1, 0, 0)$.
  - un *repère : "m2"* fixés sur le segment Ground aux coordonnées $(2, 0, 0)$.
- un *segment : "Seg1"* représentant le cube, il se caractérise par :
  - une *translation le long de l'axe x* définie dans l'intervalle $[-0.7, 2.3]$.
  - une *translation le long de l'axe z* définie dans l'intervalle $[-1, 1]$.
  - une *rotation autour de l'axe y* définie dans l'intervalle $[-0.7, 2.3]$.
  - une *masse* de 1 kg.
  - une *matrice d'inertie* caractérisant la répartition de la masse dans le volume, moment d'inertie identique selon les 3 axes.
  - un *centre de masse* aux coordonnées $(0, 0, 0)$.
  - un *ensemble de traits* — mesh : segments représentant les arêtes du cube.

```
version 4

// Seg1
    segment Seg1
        translations     xz
        rotations        y
        rangesQ   -0.70 2.30
                  -1 1
                  -pi pi
        mass 1
        inertia
          1 0 0
          0 1 0
          0 0 1
        com 0 0 0
  mesh 0 -1 -1
  mesh 0 0 -1
  mesh 0 0 0
  mesh 0 -1 0
  mesh 0 -1 -1
  mesh 1 -1 -1
  mesh 1 0 -1
  mesh 0 0 -1
  mesh 1 0 -1
  mesh 1 0 0
  mesh 0 0 0
  mesh 1 0 0
  mesh 1 -1 0
  mesh 0 -1 0
  mesh 1 -1 0
  mesh 1 -1 -1
endsegment

// Marker on Seg1
    marker m0
        parent Seg1
        position 0 0 0
    endmarker

// Ground
    segment ground
    endsegment
```

```
43
44 // Markers on ground
45     marker m1
46         parent ground
47         position 1 0 0
48     endmarker
49
50     marker m2
51         parent ground
52         position 2 0 0
53     endmarker
```

Figure 4.2: Modèle cube.bioMod

En première approximation, un segment peut être assimilable à un membre du corps humain, et un repère à un point fixe dans le repère du segment. Un segment peut être défini dans le repère global ou à partir d'un segment existant, par exemple un avant-bras doit être défini à partir du bout du bras. La librairie propose une grande quantité d'éléments biomécaniques, on peut citer non-exhaustivement :

- des segments.
- des repères.
- des muscles.
- des forces de contact.
- des forces externes.

Biorbd comprend également une importante quantité de fonctions mécaniques calculatoires. La librairie repose, entre autre, sur *Eigen*[4] : librairie d'algèbre linéaire hautement performante, et *CasADi*[3] : un outil open-source pour l'optimisation non linéaire et la différenciation algorithmique. La différenciation algorithmique de CasADi permet un gain considérable lors de la dérivation des équations différentielles. CasADi sera ainsi utilisé dans l'exécution de commande optimale, afin de tirer profit de sa différenciation algorithmique, tandis que Eigen sera privilégiée car plus rapide dans l'utilisation de BiorbdViz.

Téléchargeable en libre accès sur GitHub[1] la librairie Biorbd se développe encore au sein du laboratoire, qui essaie de démocratiser son utilisation.

4.3 BiorbdViz

Biorbd possède également une interface de programmation écrite en Python : *BiorbdViz*. Elle permet à partir d'un fichier *.bioMod* de :

- visualiser en trois dimensions le modèle.
- visualiser un mouvement en trois dimensions du modèle à partir d'un tableau de position *NumPy*.

[1]https://github.com/pyomeca/biorbd
[3]https://web.casadi.org/
[4]https://gitlab.com/libeigen/eigen

- enregistrer une vidéo du mouvement.
- afficher graphiquement l'évolution des paramètres musculaires.

Pour chaque degré de liberté, BiorbdViz dispose de curseurs indiquant la valeur actuelle par rapport à l'intervalle autorisé. A titre d'illustration, le cube initialement à 0 sur chacun de ses degré de liberté a subit d'abord une rotation autour de l'axe de 0.74 radians puis une translation de -0.52 mètres selon l'axe z.

À l'instar de Biorbd, BiorbdViz peut être téléchargé sur GitHub[2].

[2]https://github.com/pyomeca/biorbd-viz

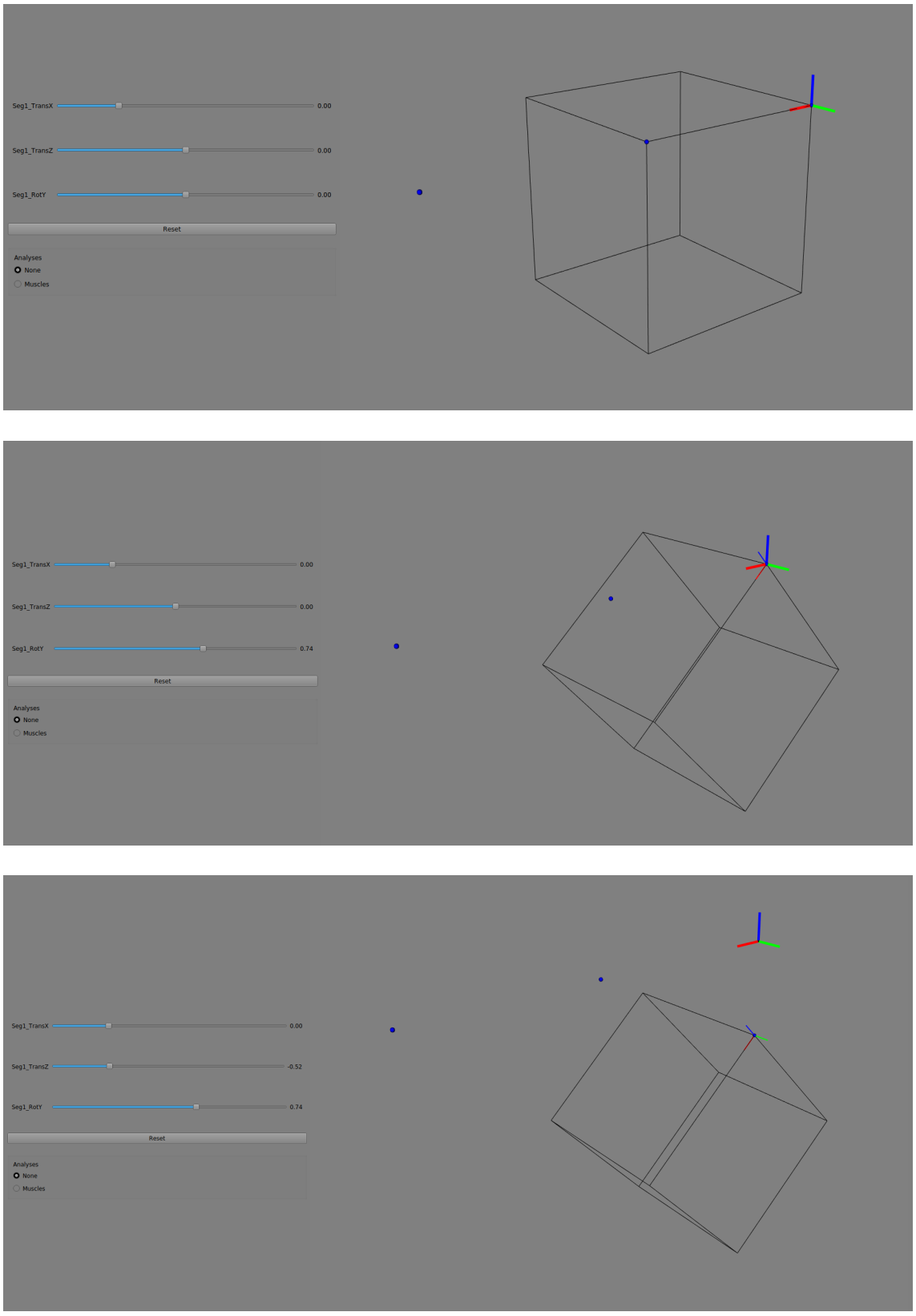


Figure 4.3: Représentation sous BiorbdViz du modèle cube.bioMod.

5. Réalisation

5.1 Nécessité collective

Comme présenté précédemment (section 2.3 p. 4), l'équipe de Commande optimale regroupe une dizaine de personne. L'épidémie du Covid-19 a contraint d'autres étudiants, privés d'expérience à mener, à rejoindre le groupe.

Chacun travaille sur une problématique biomécanique, et fait appel à la commande optimale pour la résoudre. Rapidement, un constat a émané du groupe : pour appliquer du contrôle optimal, chaque étudiant programme majoritairement les mêmes instructions que ses collègues. Face au constat qu'aucun logiciel ne permettait de résoudre correctement un problème de commande optimale semblable à celui du jeu violonistique, Benjamin Michaud a alors suggéré d'écrire une base commune permettant de mutualiser l'écriture des problèmes.

Benjamin Michaud, mon homologue Paul Wegiel et moi avons entamé la programmation de cette base au commencement de nos stages. La motivation de l'écriture de cette base est son utilisation dans le cadre de nos projets respectifs : la maximisation de la hauteur d'un saut pour Paul Wegiel, et l'optimisation du mouvement du violoniste afin de minimiser la fatigue musculaire pour moi. Il est pertinent de noter que Benjamin Michaud est le responsable de ces deux projets, la réalisation de BiorbdOptim se justifie au travers de ces deux études.

5.2 Commande optimale numérique

La solution d'un problème de commande optimale est une équation générale, obtenue comme définie précédemment (section 4.1 p. 9), à partir d'une fonction objectif, d'un ensemble de contraintes et d'un système dynamique. En pratique, la plupart des problèmes sont trop complexes pour trouver la solution exacte, on procède donc à une résolution numérique.

5.2.1 Direct multiple shooting

Pour cela, il existe plusieurs approches; au laboratoire, le direct multiple shooting est la plus utilisée. Il consiste à discrétiser la durée du problème en intervalles, puis à calculer les états en chaque point, appelé nœud. Entre ces nœuds, les commandes sont déterminées, on calcule donc l'état au prochain nœud à partir de l'état précédent et des commandes sur l'intervalle via le système dynamique et d'une intégration. Les commandes sont définies selon une fonction en escalier sur l'intervalle $[t_0, t_f]$ afin de simplifier le problème, cependant, elles pourraient suivre une fonction plus variable.

Mathématiquement, on doit définir de surcroît :

Discrétisation du temps :

$$t_0 < t_1 < ... < t_N \quad \text{avec} \quad \forall i \in [0, N] : t_i = t_0 + i \cdot \frac{t_F - t_0}{N}$$

Les états initiaux sont les points noirs (notés S_i, $i \in [0, N-1]$), les commandes sont en pointillés. L'intégration des états initiaux via les commandes permet d'obtenir les points blancs (notés X_i, $i \in [1, N]$) ainsi que le chemin pour s'y rendre.

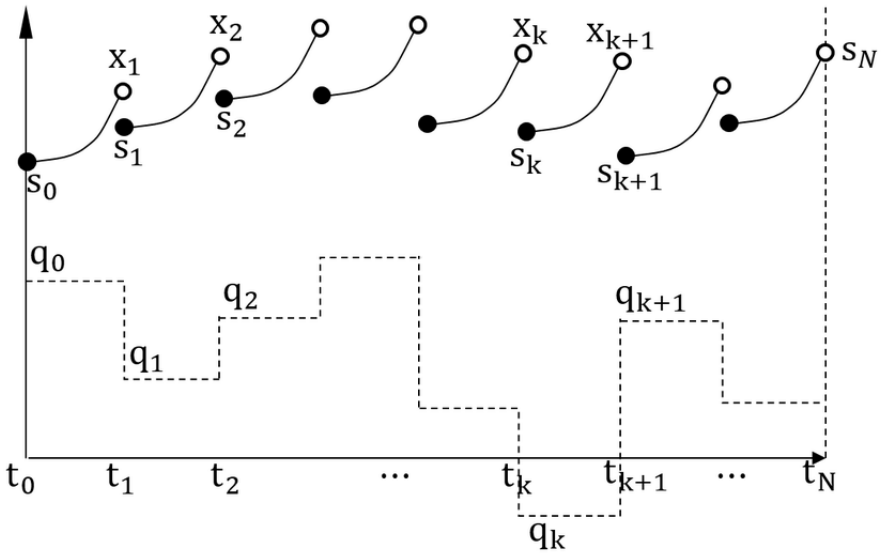


Figure 5.1: Illustration de l'approche direct multiple shooting. Ren Z. Skjetne R. Gao Z. [1]

Une remarque fondamentale est nécessaire ici, si l'on souhaite que le mouvement soit réaliste, c'est-à-dire que les positions soient continues, il faut s'assurer que :

Contrainte de continuité :

$$\forall i \in [0, N-1] : S_i = X_{i+1}$$

Il est pertinent de remarquer que le nombre de nœuds influe considérablement sur la convergence du problème : s'il est faible la solution sera approximative, à l'inverse s'il est élevé, la résolution se verra prolongée. Le travail de résolution, c'est-à-dire déterminer une solution — les commandes et les états est accompli par un logiciel appelé solveur, nous évoquerons deux d'entre eux : *Ipopt* et *ACADOS*. Afin d'appliquer un algorithme de commande optimale, il faut choisir un solveur, puis l'exécuter en lui procurant les paramètres nécessaires à la bonne définition du problème. Le solveur procédera itération après itération à des variations des commandes en fonction d'un gradient calculé à chaque itération.

5.2.2 Exemple d'implémentation

Afin de me familiariser aux outils de commande optimale, ma première mission a été d'implémenter un problème simple de commande optimale. Je l'ai codé dans le langage Python, langage très accessi-

ble, nous avons convenu avec Benjamin MICHAUD que BiorbdOptim devra être développé en Python pour simplifier et populariser son utilisation et développement. J'ai travaillé avec l'environnement de développement PyCharm, présenté précédemment (section 4 p. 9).

On considère un pendule ayant un degré de rotation autour d'un axe y, et une translation selon un axe x. Le problème consiste à amener, en cinq secondes, le pendule à partir d'une position verticale vers le bas, à une position verticale vers le haut avec une commande nulle selon la rotation. On impose également des positions initiales selon la translation identiques à 0, et des vitesses finales et initiales nulles sur les deux degrés de liberté. Concernant l'objectif, il consiste simplement à minimiser les commandes, la commande selon la rotation étant nulle, il consiste à minimiser les commandes de rotation.

L'implémentation du problème nécessite la création d'un fichier .bioMod décrivant le pendule disponible en annexe (section A p. 53), et d'un fichier appelant le solveur et décrivant le problème (section A p. 54).

Il n'est pas nécessaire d'éplucher toutes les lignes de code, il suffit de retenir qu'il est nécessaire de définir le problème comme présenté précédemment (section 4.1 p. 9). On accède aux caractéristiques du modèle .bioMod, via les fonctions *model. ....* L'utilisation du solveur ipopt impose une syntaxe, on utilise le module *opti* afin de spécifier :

les degrés de liberté et le nombre de nœuds : 2 degrés avec 31 nœuds — *lignes 27-28.*

le système dynamique : écrite avec le module casadi — *lignes 31-64.*

les fonctions objectifs : minimiser les commandes — *lignes 70-75.*

les contraintes : de continuité, de bornes, des états initiaux et finaux — *ligne 67 / lignes 77-93.*

l'appel au solveur — *ligne 97.*

La suite du fichier permet, à partir de la solution renvoyé par le solveur, d'afficher avec le module Matplotlib les valeurs des états et commandes en fonction du temps. Enfin, l'appel à BiorbdViz, l'interface trois dimension de Biorbd, avec chargement des positions permet de visualiser le mouvement.

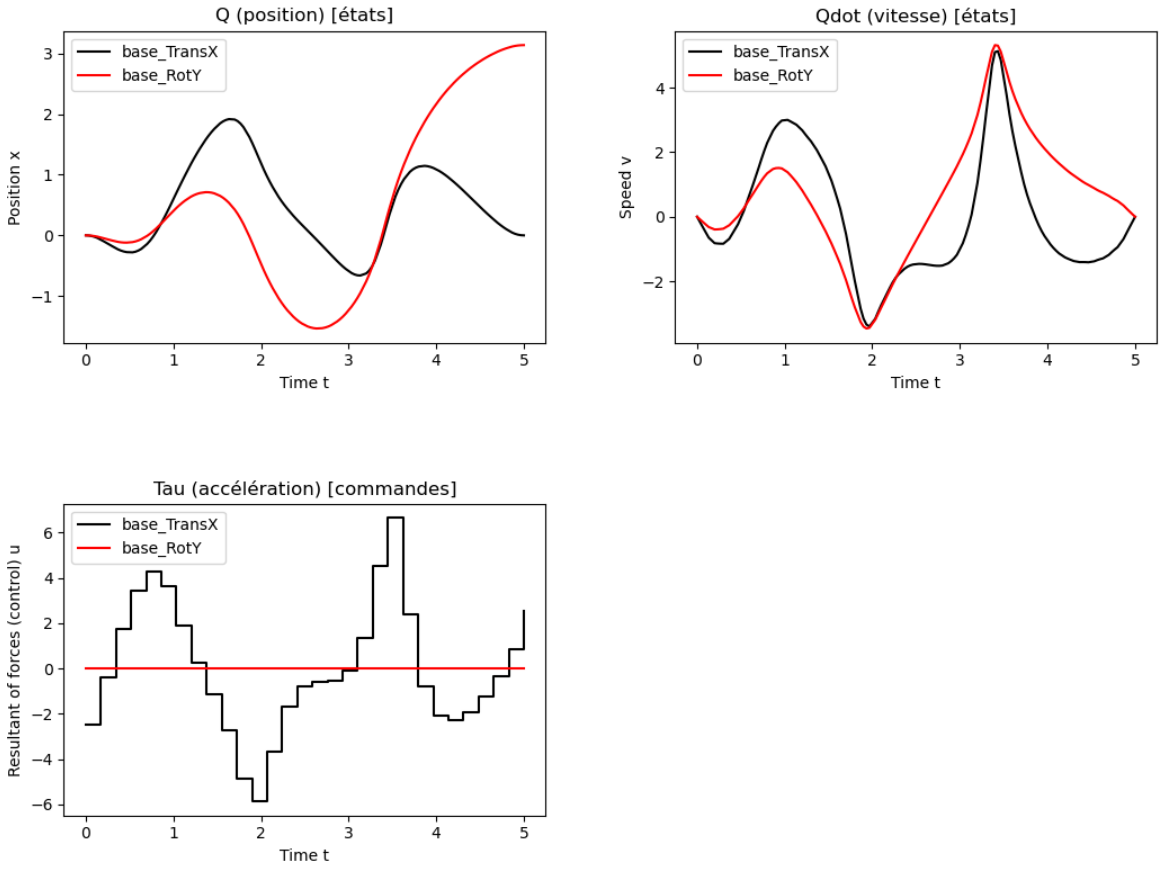


Figure 5.2: Évolution des états et commandes en fonction du temps.

La solution retournée par le solveur est correcte :

réalisation du demi tour : la position *base_RotY* passe de 0 à 3.14 radians (Figure 5.2 p. 19).

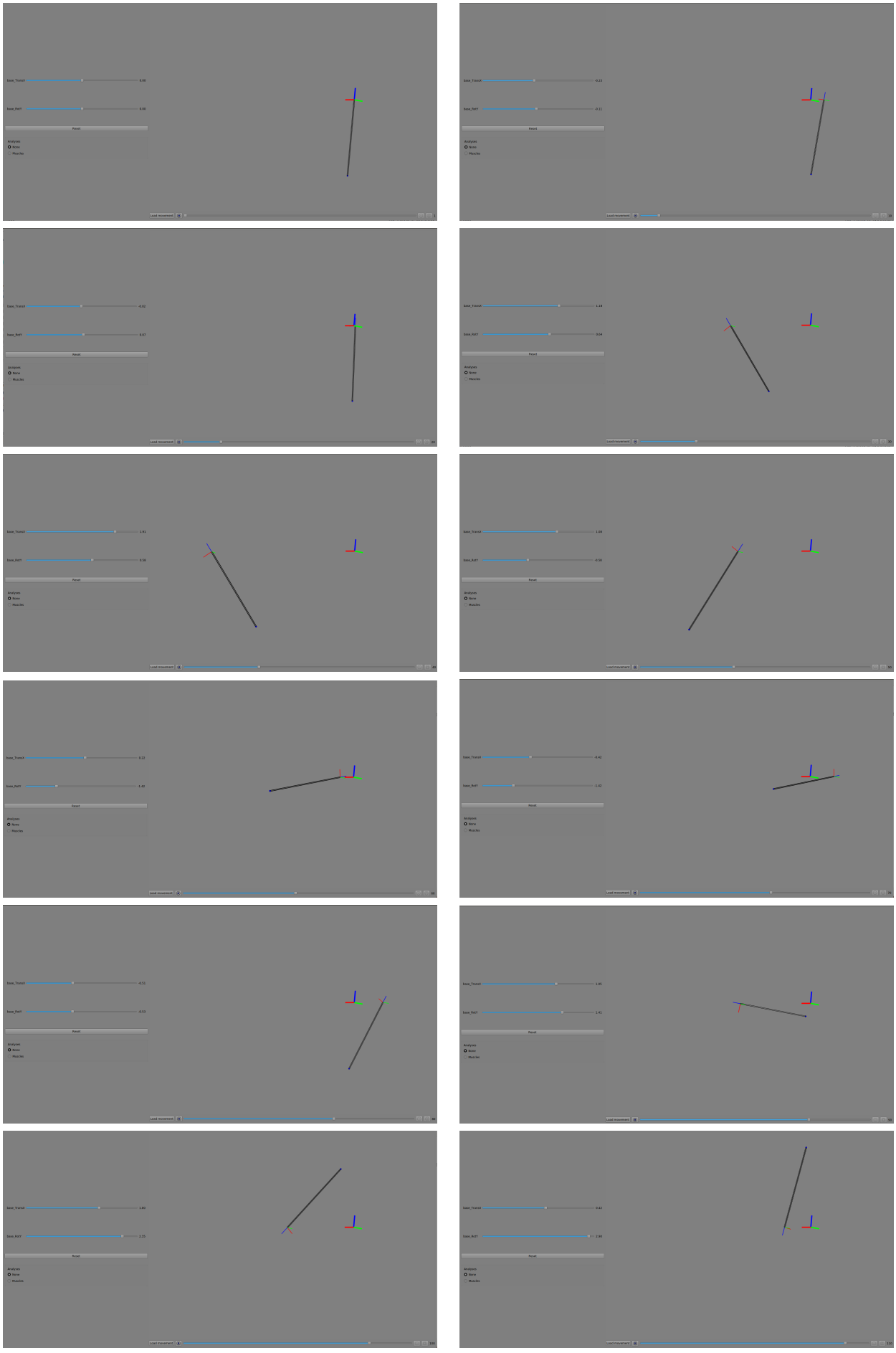
aucune commande rotation : la commande sur *base_RotY* demeure nulle (Figure 5.2 p. 19).

position initiale et finale selon la translation nulle : la position *base_TransX* débute à 0 et retombe à 0 (Figure 5.2 p. 19).

vitesses finales et initiales nulles : les vitesses débutent de 0 et finissent à 0 selon les deux degrés (Figure 5.2 p. 19).

respect de la durée de l'expérience : *5 secondes* (Figure 5.2 p. 19).

Cette première utilisation de Biorbd, CasADi, Ipopt, BiorbdViz et du module matplotlib initie le début du logiciel BiorbdOptim. Après et à partir de cet exemple, nous avons programmé le logiciel.



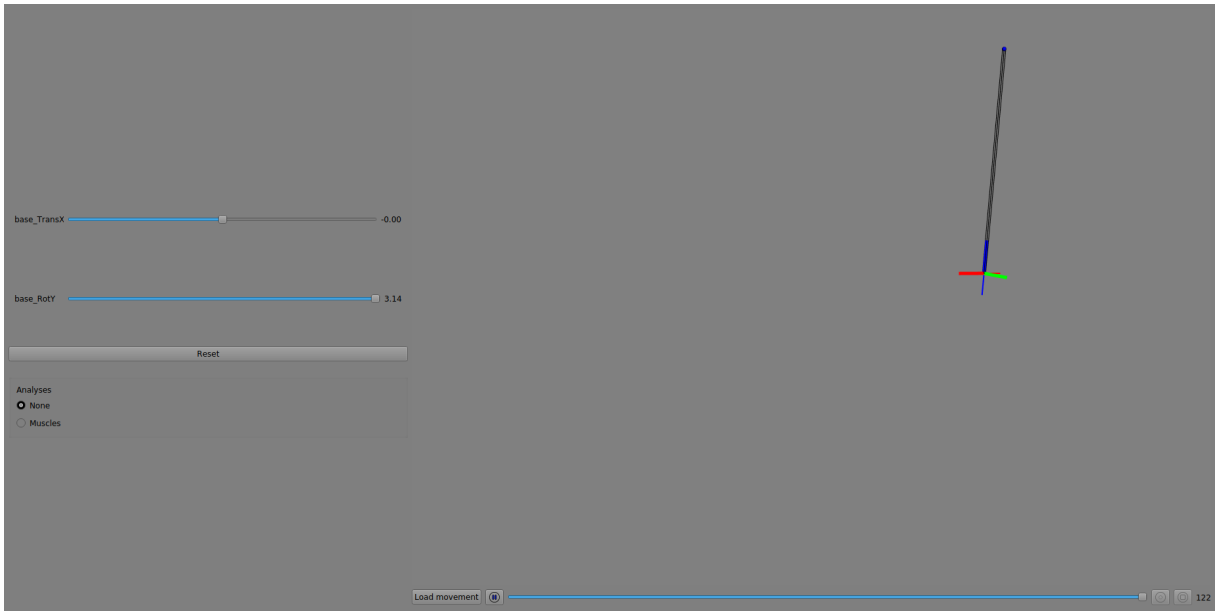


Figure 5.3: Mouvement sous BiorbdViz du modèle pendule.bioMod.

5.3 Développement collaboratif

Initialement, le logiciel a été développé par Benjamin Michaud, Paul Wegiel et moi-même. Progressivement, l'intégralité du groupe de commande optimale a contribué au développement du logiciel, *a minima* en tant qu'utilisateur.

5.3.1 Communication

Afin d'assurer la cohérence du développement, l'équipe utilise github et gitkraken présentés précédemment (section 3.2 p. 6) ainsi que Microsoft teams. L'intérêt de GitHub repose sur la possibilité d'avoir des développements parallèles opérés par différentes personnes ou non. Conceptuellement, il existe une branche principale, théoriquement toujours opérationnelle : nommée branche *master de pyomeca*. Lorsqu'un développement est souhaité, le programmeur crée une branche parallèle à celle-ci. Il programme sur son ordinateur ce que nécessaire, afin de sauvegarder ses modifications, il les regroupe et les archive sur sa branche, en anglais on parle de *commit*. Lorsque son développement est opérationnel, le programmeur fusionne sa branche avec la branche master de pyomeca, en anglais on parle de *merge*. Il est important de garder en tête que l'administrateur du projet, ici Benjamin Michaud est chargé de valider la requête de fusion, et de régler avec les développeurs les éventuels conflits : développement parallèles incompatibles et problèmes inhérents à l'implémentation du développeur. Github fonctionne en ligne de commandes, gitkraken permet de gérer les branches, et plus largement le projet graphiquement. Il est pertinent de remarquer que l'ensemble du code, des noms de commits, des issues et plus globalement de toute information publique est écrit en anglais, dans un souci de collaboration et distribution internationale.

Dans une vision à long terme, github propose d'utiliser des *issues*, tout le monde peut écrire un rapide descriptif d'une fonctionnalité à implémenter, d'un bogue à résoudre, et le partager à ses collègues via une *issue*, il peut alors s'en suivre une discussion à son propos.

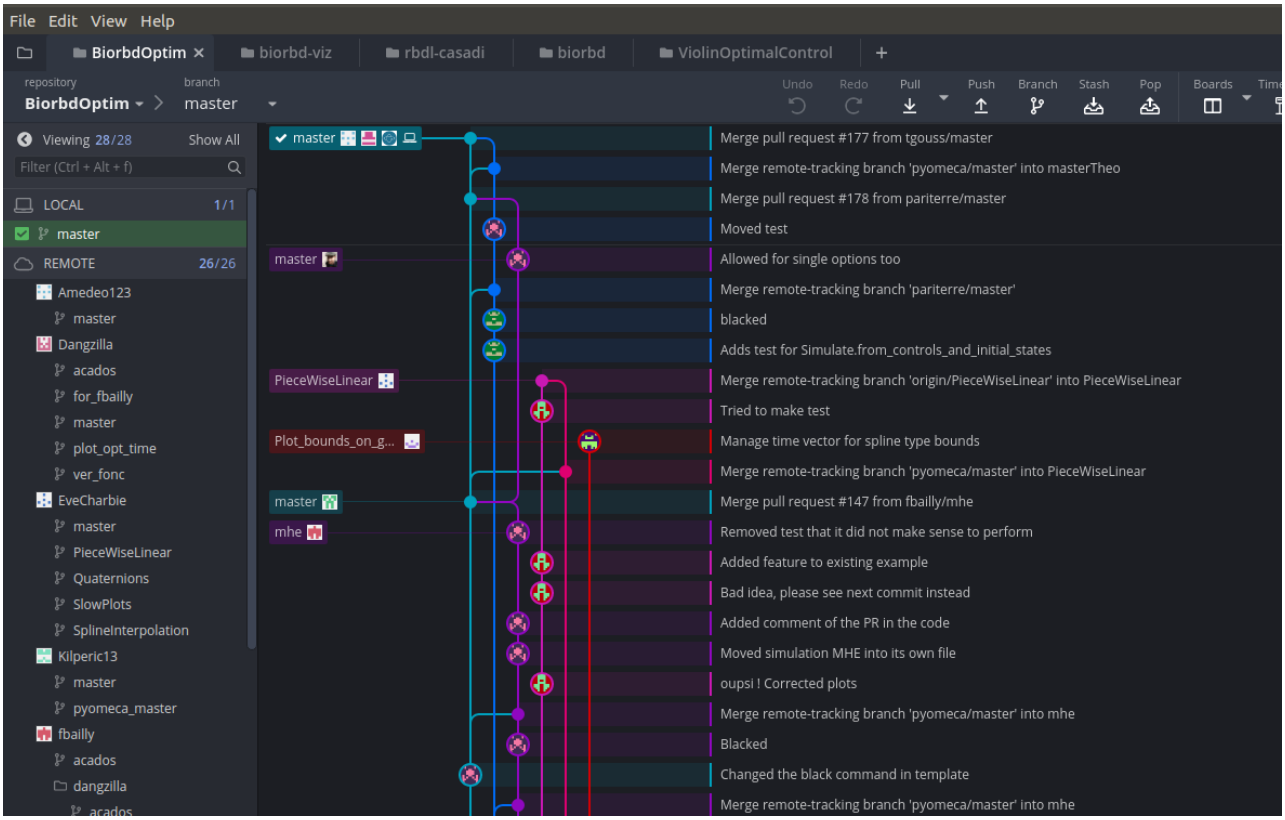


Figure 5.4: Capture d'écran du logiciel gitkraken, on y voit plusieurs branches qui fusionnent. Chaque ligne correspond à un *commit* ou un *merge* d'une des branches.

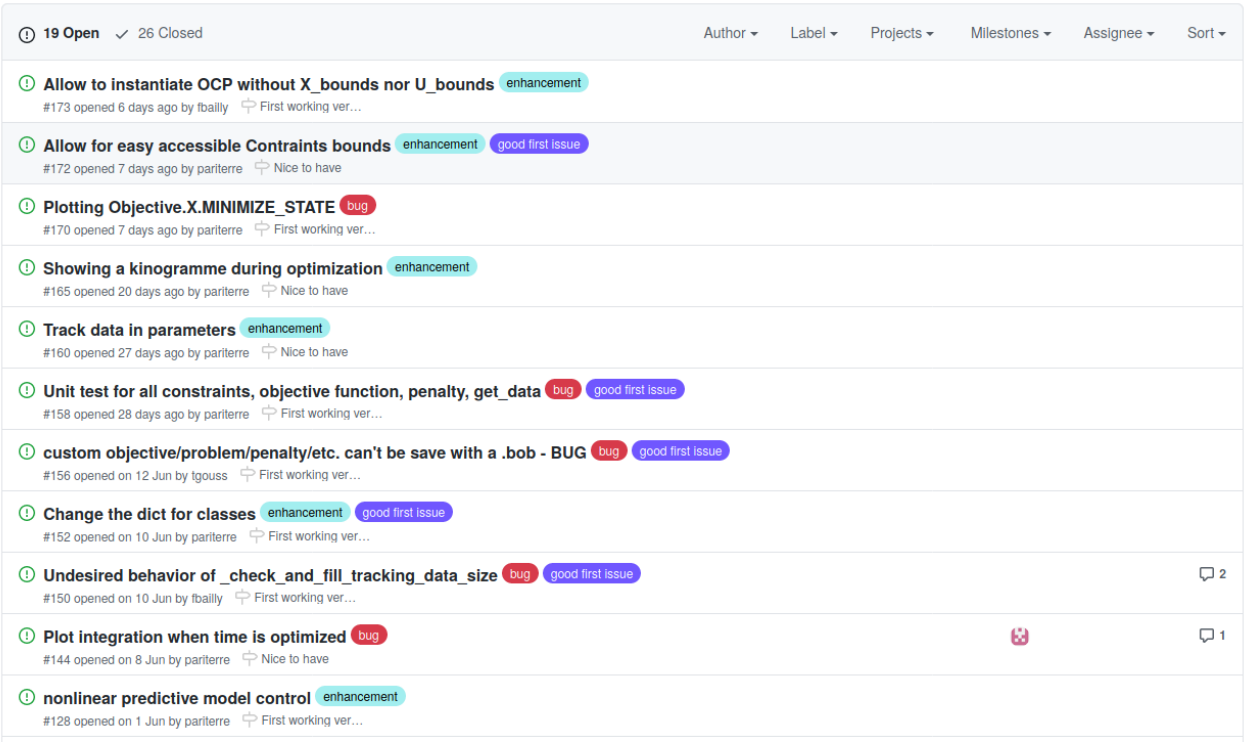


Figure 5.5: Capture d'écran de l'interface web de github, section issues. Chaque issue est décrite et étiquetée afin de renseigner sur sa criticité, son importance et sa nature.

5.3.2 Logiciel sur mesure

Puisque BiorbdOptim est en premier lieu un logiciel crée par les membres de l'équipe de commande optimale pour les membres de l'équipe, le cahier des charges n'existe qu'au travers des besoins prochains des étudiants. Il en résulte une couverture de toutes les fonctionnalités générales à l'écriture d'un problème de commande optimale. Le pendant de cette approche, est la nécessite récurrente de réécrire une structure précédemment implémentée afin de gagner en clarté et généricité. Dans la seconde section consacrée à mes travaux de recherche sur la gestuelle du violoniste, plusieurs exemples de fonctionnalités impliquant un développement sur BiorbdOptim seront développés.

5.3.3 Article scientifique

Le développement de BiorbdOptim en tant que logiciel de problème de commande optimale fait l'objet de l'écriture d'un article scientifique. Signé par les contributeurs les plus investis, l'objectif de cet article est de présenter les fonctionnalités du logiciel en comparaison à un logiciel similaire : *MoCo*. Je suis investi pour décrire certains exemples d'utilisation. La publication est envisagée d'ici le mois d'octobre 2020 dans le journal *Industrial Electronics Society — (IEEE)*[5].

[5]https://ieeexplore.ieee.orgà

6. Développement initial

6.1 Structure

Le code régissant BiorbdOptim se partitionne en deux entités : la première contient le code fonctionnel du logiciel, dossier *biorbd_optim*. La seconde recueille une série d'exemple explicitant l'utilisation de toutes les fonctionnalités du logiciel : dossier *exemples* et une série de tests : dossier *tests* Les tests écrits à l'intérieur du dossier tests permettent d'exécuter, idéalement, toutes les lignes de code

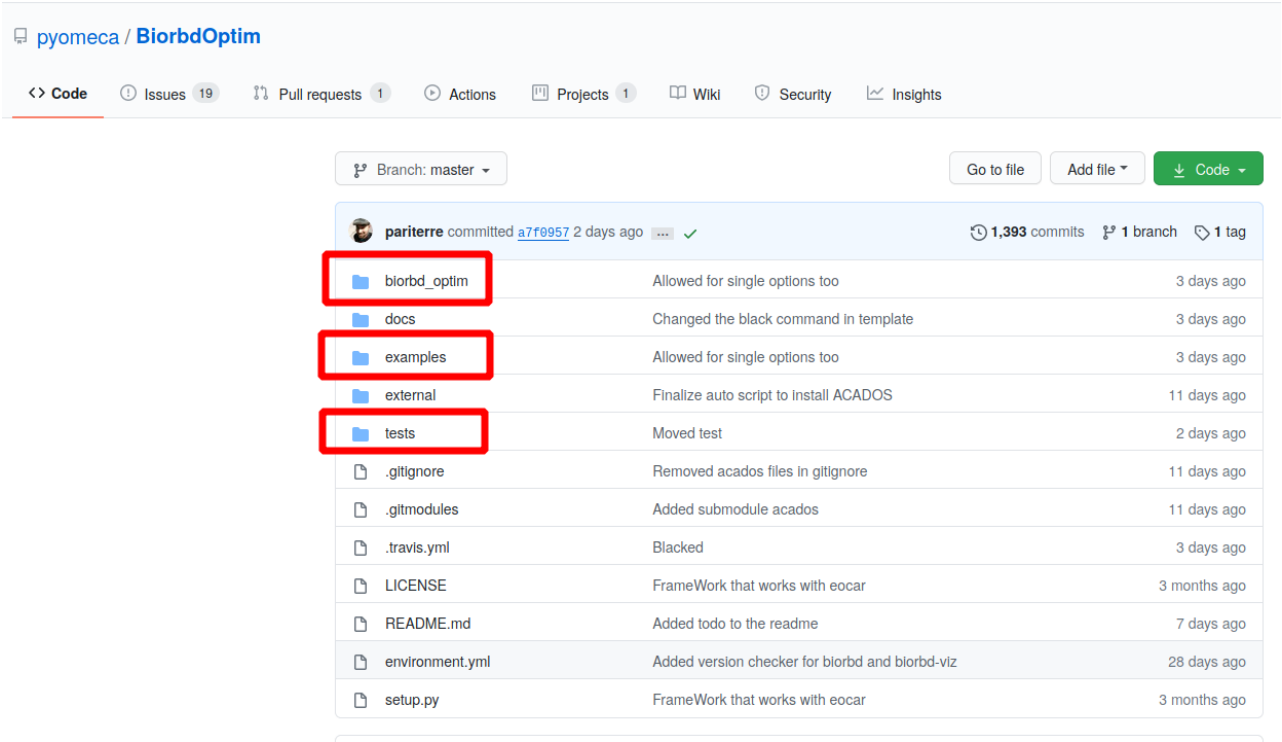


Figure 6.1: Capture d'écran de l'interface web de github, y sont répertoriés les dossiers et fichiers composant le logiciel.

du dossier biorbd_optim. Ainsi, après chaque modification ou ajout, le développeur peut s'assurer que son travail ne fausse pas celui de ses collègues. L'écriture de tests, peut paraître fastidieuse initialement, mais se révèle indispensable et permet de mettre en évidence tout dysfonctionnement.

Notons que, nous imposons à toute requête, par l'utilisation du logiciel *glstravis*, de fusion d'une branche sur la branche master de pyomeca l'acquittement de tous les tests, ainsi qu'un respect du formalise contrôlé par le module *glsblack*.

6.2 Mutualisation du code

A partir du premier fichier de commande optimale, j'ai procédé au découpage et à la mutualisation de certains parties. En reprenant l'organisation du problème du pendule décrite précédemment (section 5.2.2 p. 18). On peut dissocier :

les degrés de liberté et le nombre de nœuds : définition de problème par l'utilisateur — ici *eocar.py*.

système dynamique : écriture de la dynamique — *dynamics.py*.

fonctions objectif : écriture de toutes les fonctions objectifs — *objective_functions.py*

les contraintes : écriture de toutes les contraintes — *constraints.py*

l'appel au solveur : préparation des données pour le solveur — *__init__.py*

On obtient ainsi la structure suivante :

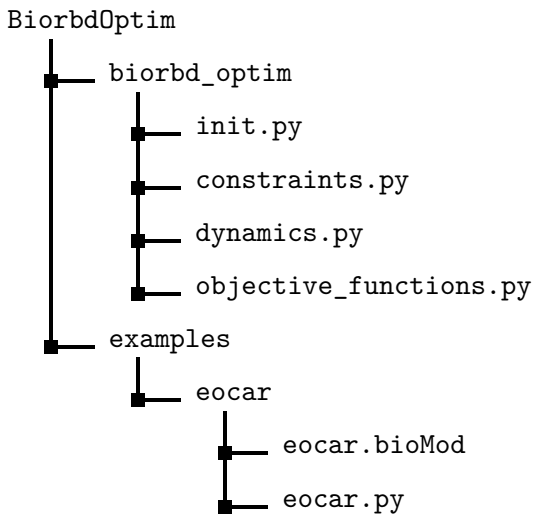


Figure 6.2: Structure initiale du logiciel BiorbdOptim

6.2.1 Définition du problème

Le problème nommé *Eocar*, utilisé comme premier exemple de BiorbdOptim consiste à déplacer un segment disposant de 3 translations et d'une rotation, d'un repère à un autre en minimisant les commandes tout au long de la simulation. On dénombre :

8 états *(X)* : 4 positions *(Q)* et 4 vitesses *(Qdot)* (3 translations et 1 rotation).
4 commandes *(U)* : 4 accélérations *(Tau)* (3 translations et 1 rotation).

L'implémentation du problème est disponible en annexe (section A p. 58).
On peut résumer les instructions à :

l'importation des fichiers de biorbd_optim — *lignes 4-7.*

paramètres du problème — *lignes 10-17.*

sélection d'une fonction objectif : parmi celles du fichiers *objective_functions.py* — *ligne 20.*

sélection de la dynamique : parmi celles du fichiers *dynamic.py* — *lignes 23-24.*

sélection des contraintes : parmi celles du fichiers *constraints.py* — *lignes 27-28.*

contraintes de bornes en position : issues du fichier *eocar.bioMod* — *lignes 34-46.*

contraintes de bornes en vitesse : vitesse intermédiaire à $\pm$ 15 m.s^{-1}, initiale et finale nulles — *lignes 48-55.*

contraintes de bornes en commande : accélération de $\pm$ 100 m.s^{-2} le long de la simulation — *lignes 57-62.*

préparation des données et appel du solveur : via le fichier *___init.py___.py* — *lignes 65-70.*

préparation des données et appel du solveur : via le fichier *___init.py___.py* — *lignes 65-70.*

affichage de la solution : comme présenté précédemment (Figure 5.2 p. 19) — *lignes 71-79.*

Il est intéressant de noter qu'avec le solveur Ipopt il est nécessaire de fournir une solution initiale, nommée *X_init* et *U_init* et définie en même temps que les contraintes de bornes. On peut facilement faire le constat que l'on pourrait automatiser l'ajout des bornes de contraintes à partir du fichier *.bioMod*, tout comme l'affichage des résultats. METTRE LIEN QUAND ON LE FAIT !

6.2.2 Dynamique

La classe *Dynamics* est amenée à posséder plusieurs fonctions dynamiques, il y a ici uniquement*forward_dynamics_torque_driven*, dynamique triviale, sans paramètre liant les accélérations *tau* aux vitesses *qdot* et aux positions *q*. On utilise directement son implémentation dans la librairie Biorbd — *ligne 11.*

```
1  from casadi import vertcat
2  import biorbd
3
4
5  class Dynamics:
6      @staticmethod
7      def forward_dynamics_torque_driven(states, controls, model):
8          q = states[: model.nbQ()]
9          qdot = states[model.nbQ() :]
10         tau = controls
11
12         qddot = biorbd.Model.ForwardDynamics(model, q, qdot, tau).to_mx()
13         return vertcat(qdot, qddot)
```

Figure 6.3: Fichier *dynamics.py* avec une seule dynamique proposée.

6.2.3 Fonctions objectif

La classe *ObjectiveFunction* est amenée à posséder plusieurs fonctions objectifs, il y a ici uniquement*minimize_torque*, fonction de Lagrange, c'est-à-dire qui s'applique tout au long de la simulation.

Elle consiste à minimiser la somme quadratique, sur chaque nœud, des commandes. Le paramètre *weight* permet lorsque plusieurs fonctions objectifs sont appliquées, de les pondérer.

```
1  import casadi
2
3
4  class ObjectiveFunction:
5      @staticmethod
6      def minimize_torque(nlp, weight=1):
7          for i in range(nlp.ns):
8              nlp.J += casadi.dot(nlp.U[i], nlp.U[i]) * nlp.dt * nlp.dt * weight
```

Figure 6.4: Fichier *objective_function.py* avec une seule fonction objectif proposée.

6.2.4 Contraintes

La classe *Constraints* est amenée à posséder plusieurs contraintes, il y a ici *___markers_to_pair*, contrainte s'assurant que, aux nœuds sélectionnés — *Instant*, la distance entre deux repères est nulle. Une seconde contrainte est disponible, *continuity_constraint*, elle assure la continuité de tous les états, entre chaque nœud et ses voisins, comme présenté précédemment (Figure 5.2.1 p. 17).

```
14
15     @staticmethod
16     class Instant(enum.Enum):
17         START = "start"
18         MID = "mid"
19         INTERMEDIATES = "intermediates"
20         END = "end"
21         ALL = "all"
22         DEFAULT = "default"
```

```
44                 x = nlp.X
45             else:
46                 continue
47
48             if elem[0] == Constraint.Type.MARKERS_TO_PAIR:
49                 Constraint.__markers_to_pair(nlp, x, elem[2])
50
51     @staticmethod
52     def __markers_to_pair(nlp, X, idx_marker):
53         for x in X:
54             marker1 = nlp.model.marker(
55                 x[: nlp.model.nbQ()], idx_marker[0]
```

```
56              ).to_mx()
57              marker2 = nlp.model.marker(
58                  x[: nlp.model.nbQ()], idx_marker[1]
59              ).to_mx()
60              nlp.g = vertcat(nlp.g, marker1 - marker2)
61              for i in range(3):
62                  nlp.g_bounds.min.append(0)
63                  nlp.g_bounds.max.append(0)
64
65      @staticmethod
66      def continuity_constraint(nlp):
67          # Loop over shooting nodes
```

Figure 6.5: Extrait du fichier *constraints.py* avec une seule contrainte proposée en complément de la contrainte de continuité.

6.2.5 Préparation des données

Avant de décrire le fichier *___init___.py*, il est nécessaire de savoir ce dont le solveur à besoin. Ipopt nécessite *a minima* :

Quatre vecteurs associés aux variables (états et commandes) :

- V contenant l'emplacement des variables, organisé comme sur la Figure 6.6.
- *V_bounds.min* contenant les bornes inférieures des éléments de V.
- *V_bounds.max* contenant les bornes supérieures des éléments de V.
- *V_bounds.init* contenant les valeurs initiales des éléments de V.

Trois vecteurs associés aux contraintes :

- g listant les expressions des contraintes.
- *g_bounds.min* contenant les bornes inférieures des éléments de g.
- *g_bounds.max* contenant les bornes supérieures des éléments de g.

Un vecteur associés aux fonctions objectifs à une dimension :

- J contenant la somme des fonctions objectifs.

Si on appelle ns le nombre de nœuds, nx le nombre d'états et nu le nombre de commandes, alors le vecteur V, et les V_bounds sont de taille $nx \cdot (ns + 1) + nu \cdot ns$. Comme exposé à la Figure 5.2.1 (p. 17), la discrétisation du temps implique *ns commandes*, chacune étant constante — entre deux nœuds, et *ns + 1 états*, ayant tous pour abscisse t_i avec $i \in [0, F]$. Il est pertinent de remarquer que les expressions des contraintes du vecteur g, valent, lorsque la contrainte est vérifiée : zéro. Ainsi les vecteurs *g_bounds.min* et *g_bounds.max* ne contiennent que des 0. Cette convention a été prise par souci de simplicité initialement, néanmoins

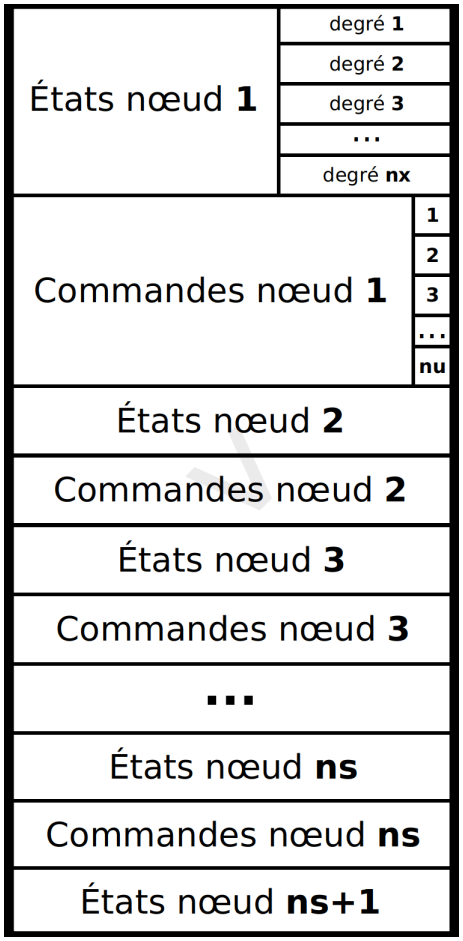


Figure 6.6: Organisation du vecteur V. Avec nx le nombre d'états, nu le nombre de commandes, et ns le nombre de nœuds

Le fichier *___init___.py*, (disponible en annexe A p. 61) organise toutes les données du problème en prévision de l'appel au solveur Ipopt. Le décrire linéairement ne présente que peu d'intérêt, il est davantage pertinent de relever ses fonctionnalités :

Vérification des bornes : — *lignes 79-82* par les classes *PathCondition* et *Bounds*, elles s'assurent que les bornes sont définies pour toutes les variables à tous les nœuds, et les complètes au besoin — *lignes 191-249*.

Dimension et organisation de V : afin de faciliter la création de V et ses dérivés, j'ai utilisé deux vecteurs : X et U, organisant respectivement des états et les commandes. La classe *variable* permet de dimensionner correctement les deux vecteurs, et de créer les variables CasADi qui leurs sont associés — *lignes 20-36*. La fonction *___define_multiple_shooting_nodes* construit le vecteur V et ses dérives à partir des vecteurs X et U. *lignes 131-163*.

Création des vecteurs J, g et ses dérivés : opérée lors de l'appel des fonctions incluses dans les fichiers *biorbd_optim*, *constraints.py* et *dynamics.py* — *lignes 94-105*. Notons l'appel de la fonction *___prepare_dynamics*, qui construit le système dynamique avec CasADi.

Appel du solveur : les vecteurs J, g ,V et leurs dérivés sont transmis au solveur Ipopt avec quelques directives — *lignes 165-188*.

6.2.6 Partage du logiciel

Cette version initiale de BiorbdOptim, utilisable par tout étudiant du laboratoire — ou chercheur à travers le monde a fait l'objet de deux présentations menées par Paul Wegiel et moi. La première (Figure A p. 67) était à l'attention des membres du groupe commande optimale, la seconde (Figure A p. 70) était à l'attention de tous membres du laboratoire.

7. Ajout de fonctionnalités

Depuis l'écriture de cette base, BiorbdOptim n'a cessé de se perfectionner selon plusieurs critères :

Fonctionnalités : capacité à décrire des problèmes complexes, minimisation du code nécessaire pour définir un problème, premières approches d'analyse des résultats per et post optimisation du solveur.

Couverture des exemples : ajout d'une panoplie d'exemples — plus de 60, illustrant l'utilisation de toutes les fonctionnalités.

Stabilité : chaque exemple induisant *a minima* un test, BiorbdOptim dispose de plus de 70 tests, couvrant 87% de toutes les lignes de codes.

Le dossier *biorbd_optim* contient à lui seul, près de trois mille lignes de codes réparties sur vingt quatre fichiers. Il convient que décrire tous les mécanismes et logiques inclus dans BiorbdOptim n'est pas le propos de ce rapport, et nécessiterait assurément plusieurs fois le nombre de page de celui-ci. Je choisis ainsi de ne présenter que certaines des fonctionnalités que j'ai implémenté, en lien avec le sujet du geste violonistique. Les contraintes, objectifs, dynamiques et modèles implémentés uniquement pour le geste violonistique seront développés dans la partie III (p. 43).

La Figure 7.1 (p. 32), affiche entre autres, le nombre de *commits* publiés au global, et individuellement. Il est pertinent de noter que malgré des fluctuations individuelles, le projet progresse à un rythme stable. Mes fluctuations sont assimilables à une alternance entre des phases de développement de BiorbdOptim et du geste violonistique.

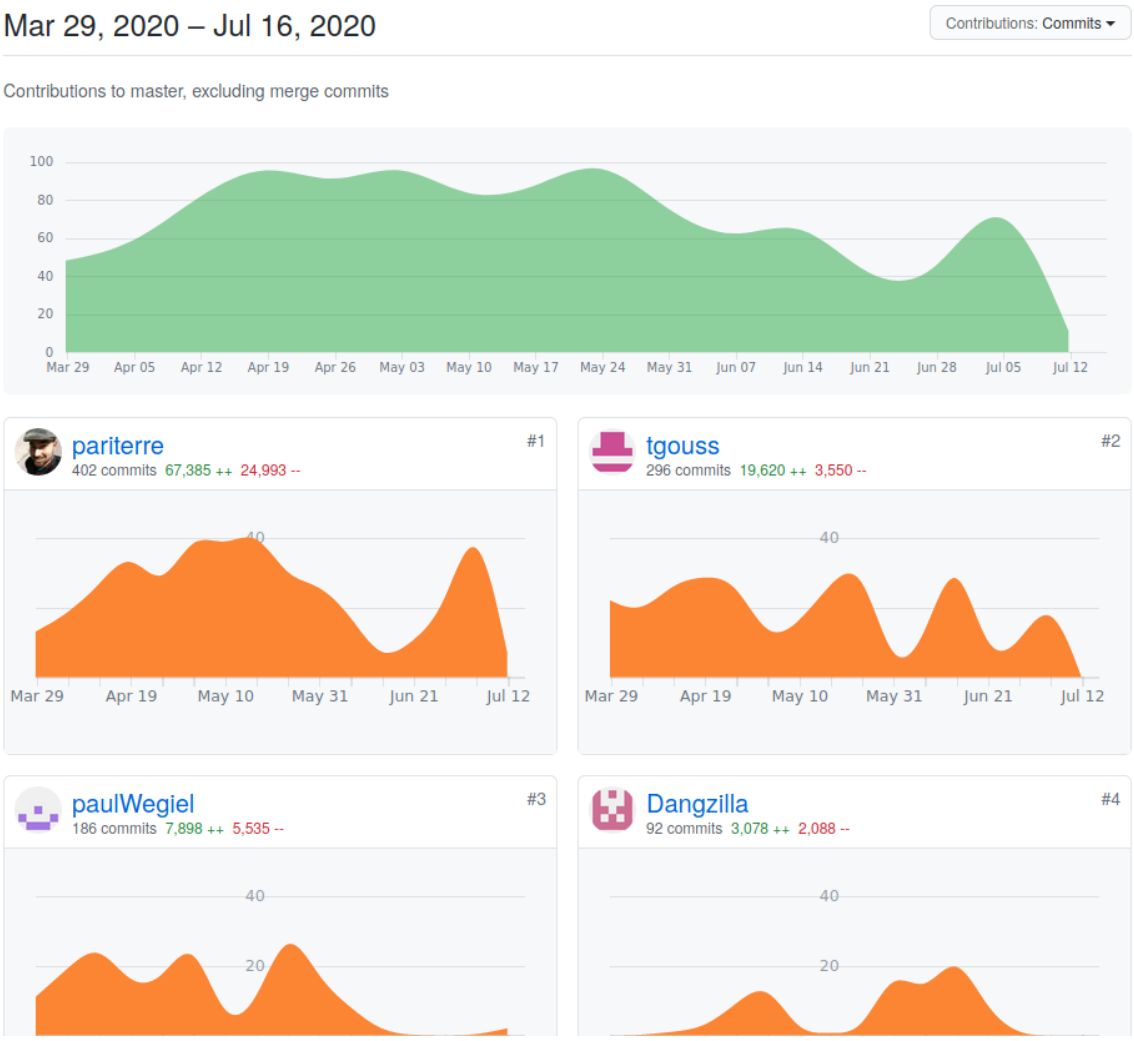


Figure 7.1: Répartition de l'écriture du code de BiorbdOptim parmi les quatre contributeurs les plus importants à date du 16/07/2020.
Benjamin MICHAUD(pariterre), Théophile GOUSSELOT(tgouss), Paul WEGIEL (paulWegiel). En vert++ et rouge– le nombre de lignes respectivement ajoutées et supprimées.

7.1 Muscles

Les variables, c'est-à-dire les états et les commandes sont identiques, des positions et des vitesses en états et des accélérations en commandes. Notons que tous ces degrés de liberté correspondent, en biomécanique, à des articulations, modélisés par des rotations. La notion de muscle, étant au cœur de mon projet violonistique, mais également d'autres projets du laboratoire, il était nécessaire de rendre l'implémentation des muscles fonctionnelle.

7.1.1 Anatomie

Les muscles sont des tissus assurant le mouvement du corps grâce une propriété essentielle, la contractilité. En effet, à la suite d'un message envoyé par le cerveau, ils ont la capacité de se contracter par eux-mêmes et ainsi entraîner les os sur lesquels ils sont attachés. Pour cela, les muscles ont une organisation bien précise. On distingue d'abord les tendons des fibres musculaires. Le tendon est la

partie qui relie les fibres musculaires aux os. Il n'a aucune capacité contractile, mais il possède des propriétés élastiques du fait du collagène qui le compose pour transmettre le mouvement du muscle vers l'os. Les fibres musculaires, elles, sont constituées de myofibrilles c'est-à-dire des tubes divisés en sarcomères qui sont le siège de la contraction grâce aux filaments d'actine et de myosine.

La force créée par cette contraction dépend des caractéristiques intrinsèques du muscle. En effet, pour une même intensité de contraction, tous les muscles ne génèrent pas la même force. Elle dépend de :

- la longueur du muscle.
- le nombre de sarcomère par myofibrille.
- le nombre de myofibrille par fibre.
- le nombre de fibre par muscle.
- l'angle de pennation — angle des fibres par rapport au muscle

7.1.2 Modélisation

La librairie Biorbd intègre une modélisation des muscles, comprenant les caractéristiques évoqués précédemment. Nous modélisons la contraction des muscles par un coefficient d'activation — compris entre 0 et 1. En fonction de ses caractéristiques, et de son activation Biorbd peut calculer les forces appliqués sur les segments.

Il suffit d'assimiler l'activation des muscles comme une commande agissant sur les états —- positions et vitesses des articulations. Il est alors nécessaire de réécrire une dynamique considérant les activations musculaires comme de nouvelles commandes.

```
@staticmethod
def forward_dynamics_torque_muscle_driven(states, controls, nlp):
    q, qdot, residual_tau = Dynamics.__dispatch_data(states, controls, nlp)

    muscles_states = biorbd.VecBiorbdMuscleStateDynamics(nlp["nbMuscle"])
    muscles_activations = controls[nlp["nbTau"] :]
    for k in range(nlp["nbMuscle"]):
        muscles_states[k].setActivation(muscles_activations[k])
    muscles_tau = (
        nlp["model"].muscularJointTorque(muscles_states, q, qdot).to_mx()
    )

    tau = muscles_tau + residual_tau
    qddot = biorbd.Model.ForwardDynamics(nlp["model"], q, qdot, tau).to_mx()
    return vertcat(qdot_reduced, qddot_reduced)
```

Figure 7.2: Extrait du fichier *dynamics.py* incluant l'activation musculaire comme commande.

Notons que pour d'autres projets de recherche, des étudiants ont souhaité dissocier la commande du cerveau de l'activation du muscle. J'ai ainsi implémenté une commande d'excitation — modélisation du message nerveux, et un état activation du muscle.

7.2 Multi-phases

La notion du multi-phase consiste à mettre bout à bout des problèmes de commande optimale ayant des influences les uns sur les autres. Il peut s'agir d'un même problème issue d'un unique modèle .bioMod avec des objectifs différents, comme de problèmes radicalement différents. On appel ainsi une *phase*, un des problèmes.

Dans le projet du geste violonistique, je modélise un tiré-poussé de l'archet, c'est-à-dire un aller retour, comme étant une phase. Dans l'optique d'une modélisation de jeu longue, la simulation sera une succession de phases, chaque phase étant un tiré-poussé de l'archet.

En principe, il suffit de remplacer toutes les entrées de l'utilisateur (Annexe A, (p. 58) par des listes de longueur égales au nombre de phase. Néanmoins, cela a imposé une réorganisation complète du fichier *___init.py___*, puisque qu'il devient nécessaire de préparer chaque phase, puis de concaténer correctement toutes les phases pour remplir correctement les vecteurs J, V, g et leurs dérivés. Toutes les phases sont regroupées dans une liste (nommée *nlp — non linear problem*), qui est l'attribut d'une classe : *OptimalControlProgram* — ses objets son majoritairement appelés *ocp*.

Afin de s'assurer de la validité des arguments fournies par l'utilisateur, des tests de types lors de l'ajout d'un paramètre à une phase ont été implémentés (Figure 7.3, p. 34). En effet, la structure de déclaration des paramètres se complexifie à cause du multi-phases (Figure 7.4, p. 35).

```
1  def __add_to_nlp(self, param_name, param, duplicate_if_size_is_one):
2      if isinstance(param, (list, tuple)):
3          if len(param) != self.nb_phases:
4              raise RuntimeError(
5                  param_name
6                  + " size("
7                  + str(len(param))
8                  + ") does not correspond to the number of phases("
9                  + str(self.nb_phases)
10                 + ")."
11             )
12         else:
13             for i in range(self.nb_phases):
14                 self.nlp[i][param_name] = param[i]
15     else:
16         if self.nb_phases == 1:
17             self.nlp[0][param_name] = param
```

Figure 7.3: Extrait de la fonction *___add_to_nlp* du fichier *___init.py___.py*. La fonction vérifie que le paramètre donné est en adéquation avec le nombre de phase, dans le cas échéant, elle ajoute à chaque phase — élément de nlp le paramètre.

```
17 def prepare_ocp(biorbd_model_path="eocar.bioMod", show_online_optim=True):
18     # --- Options --- #
19     # Model path
20     biorbd_model = (
21         biorbd.Model(biorbd_model_path),
22         biorbd.Model(biorbd_model_path),
23     )
24
25     # Problem parameters
26     number_shooting_points = (100, 1000)
27     final_time = (2, 5)
28     torque_min, torque_max, torque_init = -100, 100, 0
29
30     # Add objective functions
31     objective_functions = (
32         ((ObjectiveFunction.minimize_torque, 100),),
33         ((ObjectiveFunction.minimize_torque, 100),),
34     )
35
36     # Dynamics
37     variable_type = (ProblemType.torque_driven, ProblemType.torque_driven)
38
39     # Constraints
40     constraints = ((
41             (Constraint.Type.MARKERS_TO_PAIR, Constraint.Instant.START, (0, 1))
42             (Constraint.Type.MARKERS_TO_PAIR, Constraint.Instant.END, (0, 2)))
43         (
44             ((Constraint.Type.MARKERS_TO_PAIR, Constraint.Instant.END, (0, 1)))
45         ),
46     )
47
48     # Path constraint
49     X_bounds = [
50         QAndQDotBounds(biorbd_model[0]),
51         QAndQDotBounds(biorbd_model[0]),
52     ]
```

Figure 7.4: Extrait du fichier *eocar.py* version à *deux phases*. Tous les paramètres sont déclarés au sein de listes de deux éléments, un par phase.
A l'image de la variable *constraints*, la déclaration est complexe : ici deux contraintes sur la première phase, une seule sur la deuxième phase. L'intérêt des tests (Figure 7.3, p. 34) permet de mettre en évidence une mauvaise déclaration.

7.3 Exploitation des résultats

Le logiciel BiorbdOptim peut à ce stade décrire une large gamme de problème de commande optimale. Il retourne à l'utilisateur ce que le solveur Ipopt retourne, c'est-à-dire, en première approximation, le vecteur V complété — comprenant tous les états et commandes. Si l'objectif n'est clairement pas de proposer une solution d'analyse des résultats, il s'est avéré nécessaire de proposer à tous les utilisateurs une solution simple d'affichage des résultats.

7.3.1 Réorganisation des variables

La première tâche à accomplir, est de réorganiser le vecteur V (Figure 6.6, p. 29). J'ai crée une classe *Data*, structurant les variables de la sorte — après appel à une la fonction *Data.get_data* :

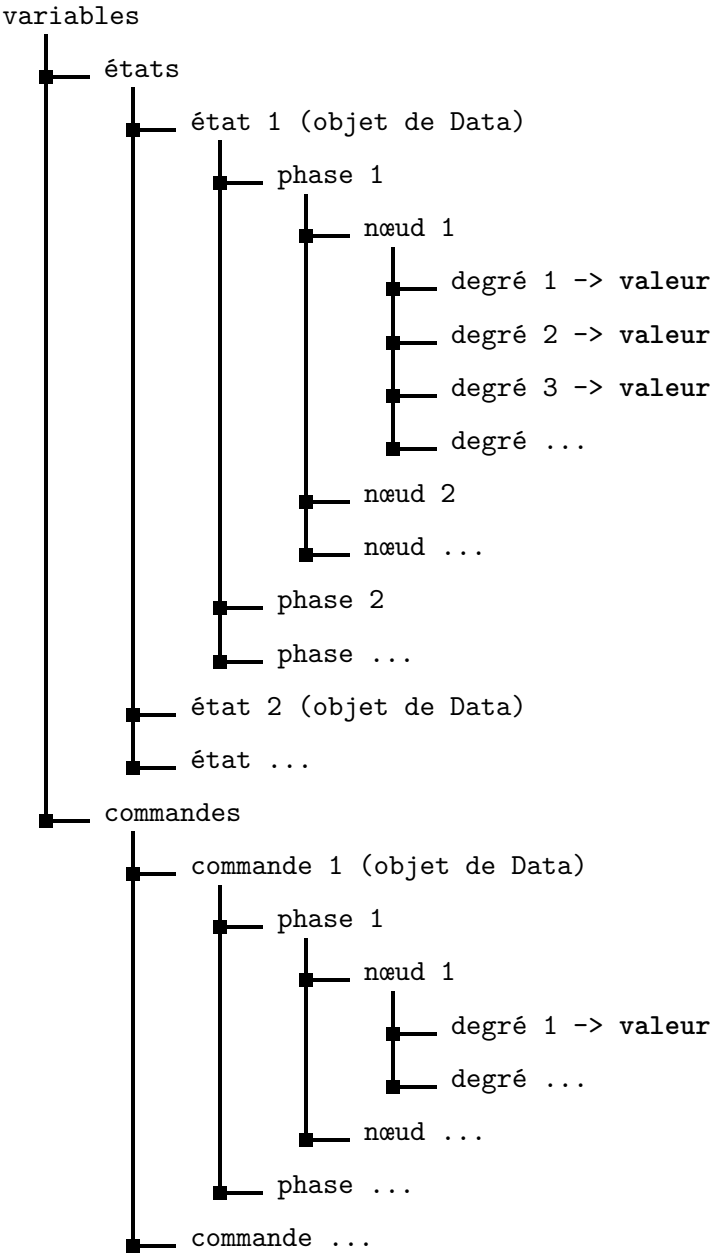


Figure 7.5: Structure retournée par l'appel à la fonction *Data.get_data* du fichier *data.py*

7.3.2 Affichage graphique

Le module Matplotlib a été utilisé pour créer et remplir les fenêtres d'affichages. J'ai crée une fenêtre par variable, c'est-à-dire, par état et commande (Figure 7.5, p. 36). Sur chaque fenêtre il y a un graphique par degré, affichant l'évolution en fonction des nœuds. Si le problème le permet on délimite les phases.

Notons, que les commandes étant considérées constantes entre deux nœuds, on trace une fonction à escalier (Figure 7.6, p. 37).

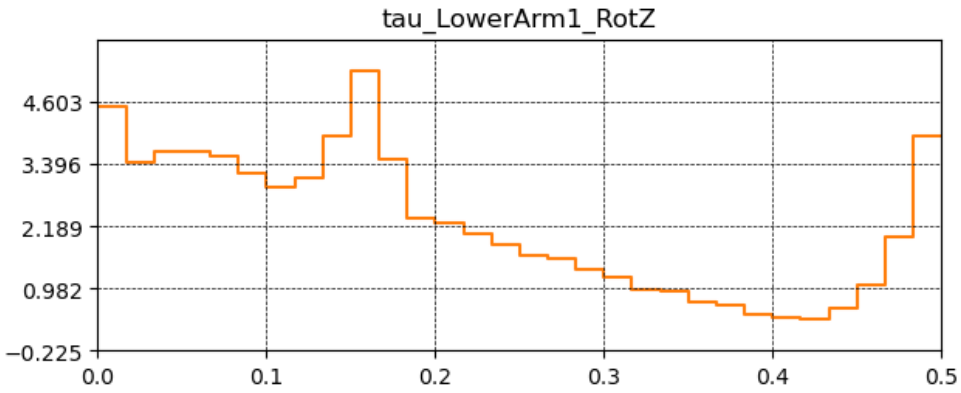


Figure 7.6: Affichage d'une commande. La commande est une fonction d'escalier.

Concernant les états, on trace autant de courbes qu'il n'y a d'intervalles entre deux nœuds. Ces courbes sont obtenus à partir de la valeur retournée par Ipopt, intégrée un certain nombre de fois. Ce type d'affichage permet de mettre en évidence des discontinuités, c'est-à-dire des non-respect de la contrainte de continuité (Figure 7.7, p. 37).

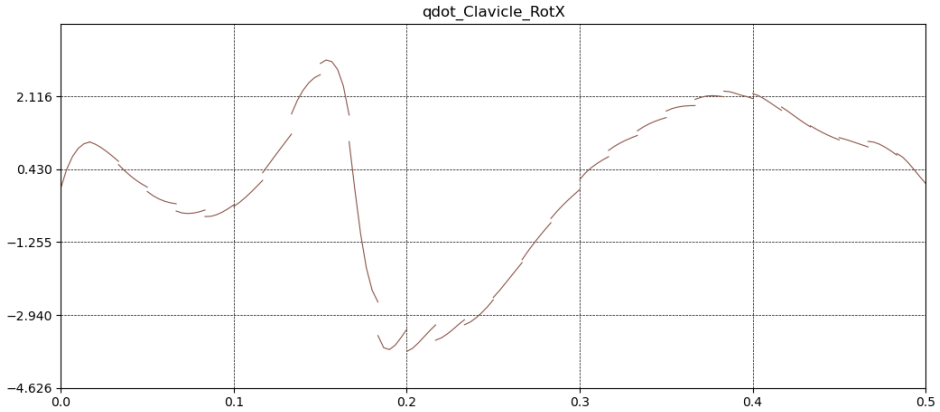


Figure 7.7: Affichage d'une vitesse. Elle présente clairement des discontinuités à certains nœuds.

Notons que l'on obtient un affichage semblable à celui exposé par Ren Z. Skjetne R. Gao Z. [1]

dans la Figure 5.1 (p. 17).

Le solveur Ipopt permet de retourner le vecteur V après chaque itération. Nous avons souhaité afficher les variables en “temps réel”, cette option a été nommée *show_online_optim* (voir Figure 7.10, p. 40). A chaque itération, on procède à la mise à jour des graphiques en fonction des solution retournée par Ipopt. En annexe est disponible une capture d'écran affichant la disposition des graphiques, et un aperçu des informations relatives aux itérations renvoyées par le solveur Ipopt (Figure A.1, p. 52).

La réalisation de tous les affichages avec :

- mise à jour à chaque itération.
- intégration des états.
- calcul de la répartition des graphiques au sein des fenêtres, et des fenêtres au sein de l'écran.

représente une importante quantité de travail, et la réécriture de plusieurs structures internes à BiorbdOptim. Néanmoins, l'affichage graphique des résultats, est disponible avec n'importe quel problème de commande optimal, indépendamment du nombre et du type d'états et de commandes.

7.3.3 Visualisation sur BiorbdViz

L'interface de programmation BiorbdViz présentée précédemment (Section A.1, p. 52), a été intégrée à BiorbdOptim. Il m'a suffit d'extraire les positions de la structure réorganisée du vecteur V (voir Figure 7.5, p. 36), et d'appeler le module BiorbdViz.

Le nombre d'images constituant l'animation du mouvement étant égal au nombre de nœuds, la vidéo pouvait être trop rapide ou lente. J'ai alors utilisé la librairie SciPy au sein de la fonction *Data.get_data* afin de retourner la structure (voir Figure 7.5, p. 36) avec un nombre de nœuds modifié par interpolation.

Le résultat est similaire à ce qu'affiché précédemment (voir Figure 5.2, p. 19), si ce n'est que le nombre d'images — en bas à droite, peut différer du nombre de nœuds par interpolation.

```
92  def get_data(
93      ocp,
94      sol_x,
95      get_states=True,
96      get_controls=True,
97      phase_idx=None,
98      integrate=False,
99      interpolate_nb_frames=-1,
100     concatenate=True,
101 ):
```

Figure 7.8: Extrait de la fonction *Data.get_data* du fichier *data.py*.
La variable *sol_x* contient le vecteur V retourné par Ipopt. On peut spécifier si la liste retournée contient les commandes, les états, si oui intégrés, interpolés ou brutes. On peut également choisir de concaténer toutes les phases.

Ces outils d'affichage des résultats ont incité la créations d'une classe *ShowResult*, construisant un object à partir d'un objet ocp de la classe OptimalControlProgram et du vecteur V retourné par Ipopt. Les object de *ShowResult* peuvent en une ligne appeler les fonctions *graphs* et *animate* qui appellent respectivement l'affichage des graphs et l'animation sur BiorbdViz.

7.4 Sauvegarde du problème et des résultats

L'exécution d'un solveur peut nécessiter un temps conséquent — plusieurs heures pour des problèmes lourds comme le geste violonistique. Il est ainsi nécessaire de pouvoir sauvegarder les résultats de la commande optimale après exécution du solveur.

J'ai utilisé la librairie Pickle afin d'écrire les fonctions de lecture/écriture de fichiers. Le choix de Pickle s'est imposé de part sa capacité à sauvegarder dans un fichier des structures complexes. J'ai nommé avec Benjamin MICHAUD l'extension de ce fichier de sauvegarde "*.bo*" en référence à BiorbdOptim. En plus de contenir le vecteur V renvoyé par le solveur, j'ai intégré une copie des arguments fournies à la classe OptimalControlProgram lors de la création de l'objet *ocp*. Cela permet de mémoriser le contexte dans lequel la solution a été obtenue, et de relancer une simulation dans un contexte identique ou proche. La copie de ocp permet également de créer un objet de la classe *ShowResult* afin d'afficher les graphiques et l'animation BiorbdViz.

J'ai crée une fonction *read_information* affichant le détail du contexte de l'objet ocp. Cela permet lorsque l'on possède bon nombre de fichier ".bo", comme ce fut mon cas avec le violon, d'identifier rapidement les particularité du contexte.

```
93  # --- Save the optimal control program and the solution --- #
94  ocp.save(sol, "pendulum.bo", sol_iterations)
95
96  # --- Load the optimal control program and the solution --- #
97  ocp_load, sol_load = OptimalControlProgram.load("pendulum.bo")
98
99  # --- Show results --- #
100 result = ShowResult(ocp_load, sol_load)
101 result.graphs()
102 result.animate()
```

Figure 7.9: Enregistrement d'un fichier ".bo" issue de l'exemple *pendulum.py*.
La variable *sol* contient le vecteur V retourné par Ipopt. On peut à partir de la sauvegarde afficher la solution.

Dans le cas d'un modèle .bioMod avec des traits — *mesh* trop nombreux, comme c'est le cas pour le modèle du violon, l'utilisation de CasADi pour ouvrir le modèle sur BiorbdViz nécessite plus de 8Go de RAM. Il devient impossible de visualiser l'animation avec CasADi, Eigen, par construction, nécessite très peu de RAM, indépendamment du nombre de traits. Le problème est que la classe OptimalControlProgram nécessite CasADi, et que l'on ne peut utiliser parallèlement CasADi et Eigen. J'ai ainsi été contraint à créer un nouveau type de fichier de sauvegarde, contenant uniquement le

résultat de la fonction *Data.get_data*. Le format étant spécialement utilisé pour la visualisation sur BiorbdViz, j'ai nommé l'extension avec Benjamin MICHAUD "*.bob*" — BiorbdOptim - BiorbdViz.

Enfin, j'ai souhaité obtenir toutes les valeurs des vecteurs V affichés en direct (Section 7.3.2, p. 38), afin de pouvoir analyser l'évolution de certains états musculaires au fil des itérations. J'ai implémenté une option, pour obtenir toutes les variables de toutes les itérations, nommée *return_iterations* (Figure 7.10, p. 40).

Il a été nécessaire d'enregistrer, à chaque itération, dans un fichier les valeurs du vecteur V. Ce fichier est supprimé une fois que l'exécution du solveur est achevée, son contenu est retourné par la fonction *solve*. Le fichier de sauvegarde temporaire doit être supprimé automatiquement, si un utilisateur le voit, c'est qu'il y a eu un dysfonctionnement. C'est pourquoi j'ai nommé son extension "*.bobo*", dans la ligné des extensions précédentes — *.bo* et *.bob*.

```
93 sol, sol_iterations = ocp.solve(
94     show_online_optim=True, return_iterations=True
95 )
```

Figure 7.10: Appel à la fonction *solve* qui lance l'exécution du solveur. Les options *show_on_optim* et *returns_iterations* permettent respectivement d'afficher les graphiques avec Matplotlib mis à jour à chaque itérations et de retourner toutes les valeurs du vecteur V pour toutes les itérations. ces valeurs sont transmises à *sol_iterations* tandis que *sol* contient en première approximation le vecteur V final.

7.5 Simulation

single shooting, comparar test manuellement sur meld-> écrire le test, il faudra l'écrire dans tous les cas

8. Enseignements

Part III

Optimisation de la gestuelle du violoniste

BiorbdOptim a pris 75% de mon temps mais a rendu le développement du Violon 4 fois plus rapide !

9. Problématique

10. Modélisation initiale

10.1 Violon

10.1.1 Jeu

10.1.2 Modélisation

10.2 Muscles

10.2.1 Fonctionnement

10.2.2 Modélisation

Travaux de Benjamin Valentin et Camille.

11. Écriture du problème

11.1 BiorbdOptim

11.2 Structure du problème

11.3 Ajout1

besoin/choix/explication/conclusion/...

11.4 Ajout2

11.5 Ajout3

11.6 Forces externes

11.7 xia

12. Résultats

torques résiduels et non-résiduels.

Part IV

Conclusion

13. Conclusion

Ces quatre mois et demi de télétravail m'ont permis de co-développer un logiciel générique de contrôle optimal que j'ai parallèlement utilisé pour modéliser et optimiser la gestuelle du violoniste.

BiorbdOptim

Le développement de *BiorbdOptim* m'a offert une expérience de travail collaboratif à distance. Progressivement, j'ai intégré le rôle et l'utilisation des divers outils de communication. J'ai compris, par la pratique et l'échec, qu'il est primordial de toujours savoir situer son travail par rapport au groupe et au projet. Sans cela, il devient inévitable de programmer des fonctionnalités déjà existantes ou incompatibles avec le reste du projet. Dans ce sens, la présence d'un manager assurant la cohérence des développements demeure irréfragable. Enfin, l'entraide étant permanente et réciproque au sein de mon groupe de travail, je ressors de ce stage convaincu par l'adage *"seul, on va plus vite, ensemble on va plus loin"*. J'en tire la conclusion qu'il est essentiel de réfléchir avec lucidité face à chaque obstacle : faut-il chercher seul, demander de l'aide et à qui, ou retarder la confrontation ? Tout cela nécessite de l'expérience, ce stage a contribué, à sa hauteur, à m'en pourvoir.

Le développement de *BiorbdOptim* m'a également offert une expérience de création de logiciel avec le langage Python, en y intégrant un logiciel écrit avec le langage C++. Outre les compétences indéniablement acquises en programmation, j'ai utilisé et joué avec un logiciel de gestion de projet informatique : *github* renforcé par *gitkraken*. Le terme "joué" me semble important, car c'est en expérimentant, que l'on acquière une maîtrise complète d'un logiciel. De même, l'utilisation quotidienne du système d'exploitation *Linux* et de *miniconda* m'a permis de manipuler ces outils basiques avec aisance. Enfin, si je ne devais retenir qu'une morale, alors je choisirais sans hésiter l'importance d'écrire de tests, dès le commencement, qui vérifient l'exactitude de toutes les lignes de code et de leurs évolutions.

BiorbdOptim s'étoffe progressivement, en intégrant continuellement plus de fonctionnalités. Á ce jour, il égal son principal concurrent : *MOCO*. Un article est, à ce jour, en écriture afin de présenter *BiorbdOptim* à la communauté scientifique de contrôle optimal appliqué à la biomécanique. Le propos de cet article est de comparer ses performances et fonctions à ceux d'une référence : *MoCo*. Je suis fier d'avoir écrit les premières lignes de ce logiciel, et d'avoir contribué, approximativement, à l'écriture d'un quart du logiciel.

Optimisation de la gestuelle du violoniste

La gestuelle de violoniste, peu étudiée par les biomécaniciens,...

modélisation, importante, 74 états 18 muscles 10 dof xia multiphase tant d'aller retour intéret de BiorbdOptim rapide à coder

démarche scientifique

difficulté de dissocier la démarche scientifique et le développement de BiorbdOptim (qu'on outil)

J'aurais souhaité...

Il reste encore à...

Expérience générale

L'expérience apportée par ces 4 mois et demi passés virtuellement au laboratoire s2m correspond à mon attente : découvrir le métier de chercheur. Je peux succinctement, citer divers apprentissages : s'il est évident de mener ses propres recherches, il demeure crucial de chercher les conclusions de ses homologues. J'ajouterais la valeur de considérer avec recul les situations de stagnation, où il parait insurmontable de parvenir à son objectif. C'est là tout l'enjeu de la recherche : expérimenter ce que personne n'a déjà tenté. Subséquemment à la frustration de la stagnation, la joie enivrante lors d'un succès alimente la volonté de perpétuer et perfectionner ses travaux. Bien que n'ayant qu'à traverser mon appartement pour me rendre sur mon lieu de travail, je l'ai toujours fait avec alacrité et excitation.

Finalement, l'expérience apportée par ces 4 mois et demi passés effectivement à Montréal me remplie de satisfaction. En dépit du contexte pandémique, j'ai pu profiter de l'accueil enthousiaste des québecois à commencer par le tutoiement qui l'illustre parfaitement. Je retiendrais des belles rencontres avec de longs échanges sur l'histoire et la double colonisation du Québec, dont la devise est : *Je me souviens, que né sous le lys, je croîs sous la rose.* Au terme de mon stage, j'ai pu jouir de la superbe nature québecoise, de ses lacs et forêts, peu apprivoisées par l'homme.

[2]

A. Annexes

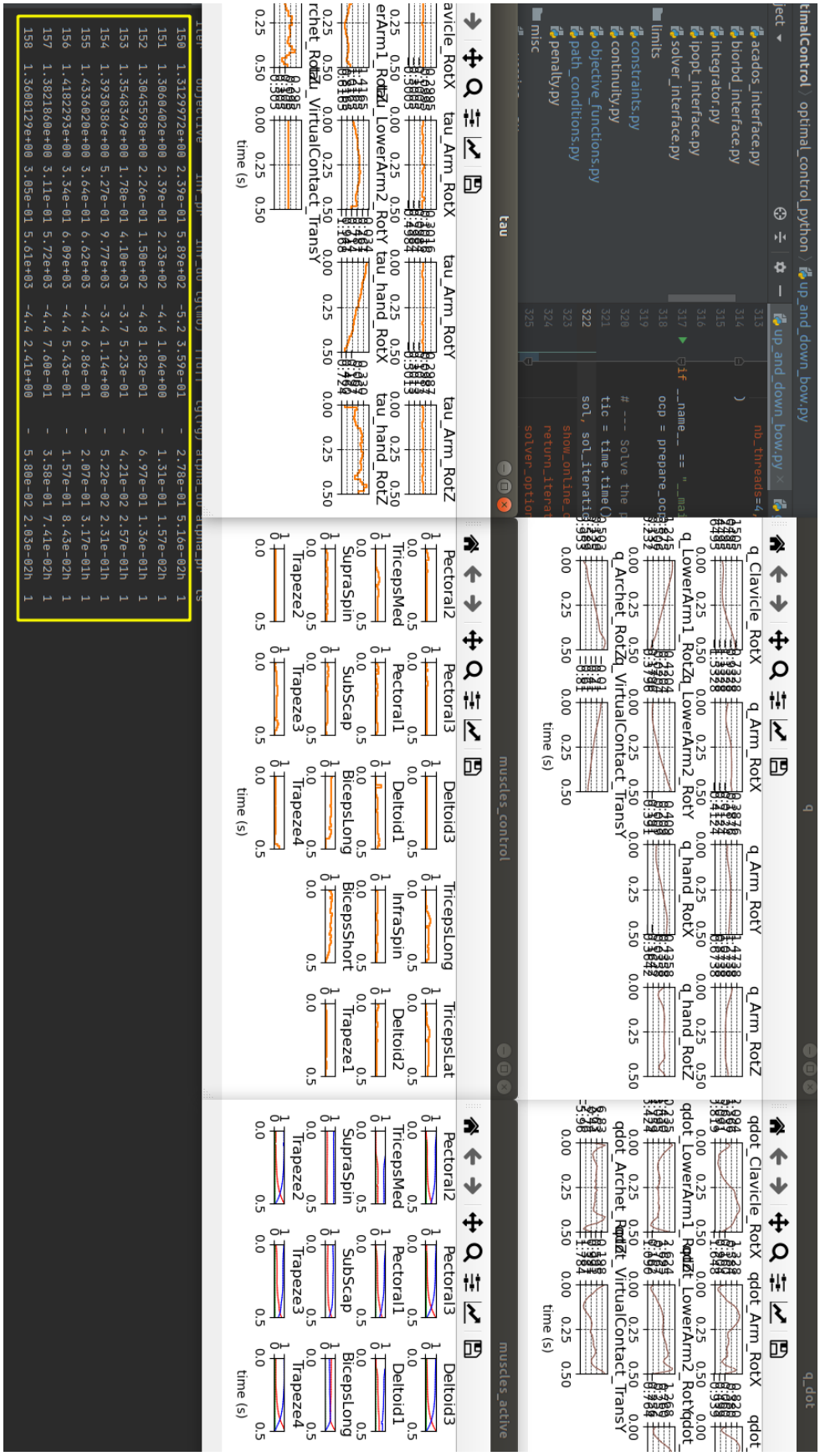


Figure A.1: Utilisation de la fonction d'affichage des variables avec mise à jour à chaque itérations. On compte une fenêtre par état/commande, excepté pour la fenêtre *muscles_actives*, qui regroupe trois états. Les itérations renvoyées par le solveur défilent dans le cadre jaune.

```
version 3

// SEGMENT DEFINITION

// Information about base segment
    // Segment
    segment base
        translations x
        rotations y
        mass 1
        com 0 0 -1
        inertia
        1 0 0
        0 1 0
        0 0 1

        mesh 0.005 0.005 0
        mesh -0.005 0.005 0
        msh -0.005 -0.005 0
        mesh 0.005 -0.005 0
        mesh 0.005 0.005 0
        mesh 0.005 0.005 -1
        mesh -0.005 0.005 -1
        mesh -0.005 0.005 0
        mesh -0.005 0.005 -1
        mesh -0.005 -0.005 -1
        mesh -0.005 -0.005 0
        mesh -0.005 -0.005 -1
        mesh 0.005 -0.005 -1
        mesh 0.005 -0.005 0
        mesh 0.005 -0.005 -1
        mesh 0.005 0.005 -1



    endsegment
// Markers
        marker  origine
                parent  base
                position        0 0 -1
        endmarker
```

Figure A.2: Modèle pendule.bioMod.

```
from casadi import MX, Opti, vertcat, Function
from matplotlib import pyplot as plt
import numpy as np
import biorbd
import BiorbdViz

model = biorbd.Model("pendule.bioMod")


# Initialisation
T = 5  # 100 secondes
N = 30  # 30 intervalles
G = 9.81

continous_integration = True
number_of_nodes = 5
nbQ = model.nbQ()
nbQdot = model.nbQdot()
nbTau = model.nbGeneralizedTorque()
nx = nbQ + nbQdot
nu = nbTau
dt = T / N  # duree d'un intervalle
colors = ["k", "r", "g", "b", "y", "m"]
captions = [s.to_string() for s in model.nameDof()]

opti = Opti()
x = opti.variable(nx, N + 1)  # position et vitesse
u = opti.variable(nu, N)  # controle


# Declaration dynamique
x_sym = MX.sym("x", nx)
u_sym = MX.sym("u", nu)
dyn = Function(
    "Dynamics",
    [x_sym, u_sym],
    [
        vertcat(
            x_sym[nbQ:, 0],
            model.ForwardDynamics(
```

```
                x_sym[:nbQ, 0], x_sym[nbQ:, 0], u_sym
            ).to_mx(),
        )
    ],
    ["x", "u"],
    ["xdot"],
).expand()


def rk_calcul(x, u, dt_kutta):
    k1 = dyn(x, u)
    k2 = dyn(x + (dt_kutta / 2) * k1, u)
    k3 = dyn(x + (dt_kutta / 2) * k2, u)
    k4 = dyn(x + dt_kutta * k3, u)
    return x + (dt_kutta / 6) * (k1 + 2 * k2 + 2 * k3 + k4)


# RK4
dt_kutta = dt / (number_of_nodes)
rk = Function("RK", [x_sym, u_sym], [rk_calcul(x_sym, u_sym, dt_kutta)])
for j in range(N):
    x_next = x[:, j]
    for p in range(number_of_nodes):
        x_next = rk(x_next, u[:, j])

    # Contrainte de continuite
    opti.subject_to(x_next == x[:, j + 1])

# Fonction objectif
obj = MX(0)
for i in range(nu):
    for j in range(N):
        obj += u[i, j] * dt * u[i, j] * dt

opti.minimize(obj)  # optimisation sur plusieurs crit res possible

# Contraintes de continuite
opti.subject_to(x[:, 0] == 0)  # vitesse et position nulles au debut
opti.subject_to(x[0, N] == 0)
opti.subject_to(x[1, N] == 3.14)
opti.subject_to(x[nbQ:, N] == 0)  # vitesses nulles     la fin

# Contraintes sur le chemin
```

```
84  for j in range(N + 1):
85      opti.subject_to(x[:nbQ, j] >= -10)
86      opti.subject_to(x[:nbQ, j] <= 20)
87      opti.subject_to(x[nbQ:, j] >= -100)
88      opti.subject_to(x[nbQ:, j] <= 100)
89
90  for j in range(N):
91      opti.subject_to(u[0, j] >= -100)
92      opti.subject_to(u[0, j] <= 100)
93      opti.subject_to(u[1, j] == 0)
94
95  # Solution
96  opti.solver("ipopt")
97  sol = opti.solve()  # appel au solveur
98
99  x_opt = sol.value(x)
100 u_opt = sol.value(u)
101
102 t_int = np.ndarray(((number_of_nodes - 1) * N + 2,))
103 q_int = np.ndarray((nbQ, (number_of_nodes - 1) * N + 2))
104 qdot_int = np.ndarray((nbQdot, (number_of_nodes - 1) * N + 2))
105 x_next = x_opt[:, 0]
106 cmp = 0
107 continous_integration = True
108 t_int[cmp] = 0
109 q_int[:, cmp] = x_next[:nbQ]
110 qdot_int[:, cmp] = x_next[nbQ:]
111 cmp += 1
112 for j in range(N):
113     if not continous_integration:
114         x_next = x_opt[:, j]
115
116     for p in range(number_of_nodes):
117         x_next = rk(x_next, u_opt[:, j])
118         t_int[cmp] = dt_kutta * j * number_of_nodes + dt_kutta * p + dt_kutta
119         q_int[:, cmp] = np.array(x_next[:nbQ, :]).reshape((nbQ,)).squeeze()
120         qdot_int[:, cmp] = (
121             np.array(x_next[nbQ:, :]).reshape((nbQdot,)).squeeze()
122         )
123         cmp += 1
124     cmp -= 1
125
126 plt.subplot(221)
```

```
for i in range(nbQ):
    plt.title("Q (position) [ tats ]")
    plt.plot(t_int, q_int[i, :], colors[i], label=captions[i])
    plt.xlabel("Time t")
    plt.ylabel("Position x")

plt.subplot(222)
for i in range(nbQdot):
    plt.title("Qdot (vitesse) [ tats ]")
    plt.plot(t_int, qdot_int[i, :], colors[i], label=captions[i])
    plt.xlabel("Time t")
    plt.ylabel("Speed v")

plt.subplot(223)
for i in range(nu):
    plt.title("Tau (acc l ration) [commandes]")
    line = plt.step(
        np.linspace(0, T, N),
        u_opt[i],
        where="post",
        color=colors[i],
        label=captions[i],
    )
    plt.xlabel("Time t")
    plt.ylabel("Resultant of forces (control) u")


plt.tight_layout()
plt.legend()
plt.show()


BiorbdViz.BiorbdViz("pendule.bioMod").load_movement(q_int[:, :61])
```

Figure A.3: Programmation du problème du pendule.

```python
import biorbd
from matplotlib import pyplot as plt

import biorbd_optim
from biorbd_optim.objective_functions import ObjectiveFunction
from biorbd_optim.constraints import Constraint
from biorbd_optim.dynamics import Dynamics

# --- Options --- #
# Model path
biorbd_model = biorbd.Model("eocar.bioMod")

# Problem parameters
number_shooting_points = 30
final_time = 2
ode_solver = biorbd_optim.OdeSolver.RK
is_cyclic_constraint, is_cyclic_objective = False, False

# Objective functions
objective_functions = ((ObjectiveFunction.minimize_torque, 100),)

# Dynamics
variable_type = biorbd_optim.Variable.variable_torque_driven
dynamics_func = Dynamics.forward_dynamics_torque_driven

# Constraints
constraints = (
    (Constraint.Type.MARKERS_TO_PAIR, Constraint.Instant.START, (0, 1)),
    (Constraint.Type.MARKERS_TO_PAIR, Constraint.Instant.END, (0, 2)),
)

# Define path constraint
X_bounds = biorbd_optim.Bounds()
X_init = biorbd_optim.InitialConditions()
ranges = []
for i in range(biorbd_model.nbSegment()):
    segRanges = biorbd_model.segment(i).ranges()
    for j in range(len(segRanges)):
        ranges.append(biorbd_model.segment(i).ranges()[j])

for i in range(biorbd_model.nbQ()):
    X_bounds.first_node_min.append(ranges[i].min())
```

```
        X_bounds.first_node_max.append(ranges[i].max())
        X_bounds.min.append(ranges[i].min())
        X_bounds.max.append(ranges[i].max())
        X_bounds.last_node_min.append(ranges[i].min())
        X_bounds.last_node_max.append(ranges[i].max())
        X_init.init.append(0)

for i in range(biorbd_model.nbQdot()):
        X_bounds.first_node_min.append(0)
        X_bounds.first_node_max.append(0)
        X_bounds.min.append(-15)
        X_bounds.max.append(15)
        X_bounds.last_node_min.append(0)
        X_bounds.last_node_max.append(0)
        X_init.init.append(0)

U_bounds = biorbd_optim.Bounds()
U_init = biorbd_optim.InitialConditions()
for i in range(biorbd_model.nbGeneralizedTorque()):
        U_bounds.min.append(-100)
        U_bounds.max.append(100)
        U_init.init.append(0)

# --- Solve the program --- #
nlp = biorbd_optim.OptimalControlProgram(
        biorbd_model,
        variable_type,
        dynamics_func,
        ode_solver,
        number_shooting_points,
        final_time,
        objective_functions,
        X_init,
        U_init,
        X_bounds,
        U_bounds,
        constraints,
        is_cyclic_constraint=is_cyclic_constraint,
        is_cyclic_objective=is_cyclic_objective,
)

sol = nlp.solve()
for idx in range(biorbd_model.nbQ()):
```

```
86      plt.figure()
87      q = sol["x"][0 * biorbd_model.nbQ() + idx :: 3 * biorbd_model.nbQ()]
88      q_dot = sol["x"][1 * biorbd_model.nbQ() + idx :: 3 * biorbd_model.nbQ()]
89      u = sol["x"][2 * biorbd_model.nbQ() + idx :: 3 * biorbd_model.nbQ()]
90      plt.plot(q)
91      plt.plot(q_dot)
92      plt.plot(u)
93  plt.show()
```

Figure A.4: Définition du problème eocar.

```python
import enum

import casadi
from casadi import MX, vertcat

from .constraints import Constraint


class OdeSolver(enum.Enum):
    """
    Four models to solve.
    RK is pretty much good balance.
    """

    COLLOCATION = 0
    RK = 1
    CVODES = 2
    NO_SOLVER = 3


class Variable:
    @staticmethod
    def variable_torque_driven(nlp):
        dof_names = nlp.model.nameDof()
        q = MX()
        q_dot = MX()
        for i in range(nlp.model.nbQ()):
            q = vertcat(q, MX.sym("Q_" + dof_names[i].to_string()))
        for i in range(nlp.model.nbQdot()):
            q_dot = vertcat(q_dot, MX.sym("Qdot_" + dof_names[i].to_string()))
        nlp.x = vertcat(q, q_dot)

        for i in range(nlp.model.nbGeneralizedTorque()):
            nlp.u = vertcat(nlp.u, MX.sym("Tau_" + dof_names[i].to_string()))

        nlp.nx = nlp.x.rows()
        nlp.nu = nlp.u.rows()


class OptimalControlProgram:
    """

```

```
43      """
44
45      def __init__(
46          self,
47          biorbd_model,
48          variable_type,
49          dynamics_func,
50          ode_solver,
51          number_shooting_points,
52          final_time,
53          objective_functions,
54          X_init,
55          U_init,
56          X_bounds,
57          U_bounds,
58          constraints,
59          is_cyclic_constraint=False,
60          is_cyclic_objective=False,
61      ):
62          """
63
64          :param biorbd_model:
65          :param variable_type:
66          :param dynamics_func:
67          :param ode_solver:
68          :param number_shooting_points:
69          :param final_time:
70          :param X_bounds:
71          :param U_bounds:
72          :param constraints:
73          """
74          self.model = biorbd_model
75
76          # Define some aliases
77          self.ns = number_shooting_points
78          self.tf = final_time
79          self.dt = final_time / max(number_shooting_points, 1)
80          self.is_cyclic_constraint = is_cyclic_constraint
81          self.is_cyclic_objective = is_cyclic_objective
82
83          # Compute problem size
84          self.x = MX()
85          self.u = MX()
```

```
86          self.nx = -1
87          self.nu = -1
88          variable_type(self)
89
90          X_init.regulation(self.nx)
91          X_bounds.regulation(self.nx)
92          U_init.regulation(self.nu)
93          U_bounds.regulation(self.nu)
94
95          # Variables and constraint for the optimization program
96          self.X = []
97          self.U = []
98          self.V = MX()
99          self.V_init = InitialConditions()
100         self.V_bounds = Bounds()
101         self.g = []
102         self.g_bounds = Bounds()
103         self.__define_multiple_shooting_nodes(
104             X_init, U_init, X_bounds, U_bounds
105         )
106
107         # Define dynamic problem
108         self.__prepare_dynamics(biorbd_model, dynamics_func, ode_solver)
109         Constraint.continuity_constraint(self)
110
111         # Constraint functions
112         self.constraints = constraints
113         Constraint.add_constraints(self)
114
115         # Objective functions
116         self.J = 0
117         for (func, weight) in objective_functions:
118             func(self, weight=weight)
119
120     def __prepare_dynamics(self, biorbd_model, dynamics_func, ode_solver):
121         states = MX.sym("x", self.nx, 1)
122         controls = MX.sym("p", self.nu, 1)
123         dynamics = casadi.Function(
124             "ForwardDyn",
125             [states, controls],
126             [dynamics_func(states, controls, biorbd_model)],
127             ["states", "controls"],
128             ["statesdot"],
```

```
        ).expand()
        ode = {"x": self.x, "p": self.u, "ode": dynamics(self.x, self.u)}

        ode_opt = {"t0": 0, "tf": self.dt}
        if ode_solver == OdeSolver.RK or ode_solver == OdeSolver.COLLOCATION:
            ode_opt["number_of_finite_elements"] = 5

        if ode_solver == OdeSolver.RK:
            self.dynamics = casadi.integrator(
                "integrator", "rk", ode, ode_opt
            )
        elif ode_solver == OdeSolver.COLLOCATION:
            self.dynamics = casadi.integrator(
                "integrator", "collocation", ode, ode_opt
            )
        elif ode_solver == OdeSolver.CVODES:
            self.dynamics = casadi.integrator(
                "integrator", "cvodes", ode, ode_opt
            )

    def __define_multiple_shooting_nodes(
        self, X_init, U_init, X_bounds, U_bounds
    ):
        nV = self.nx * (self.ns + 1) + self.nu * self.ns
        self.V = MX.sym("V", nV)
        self.V_bounds.min = [0] * nV
        self.V_bounds.max = [0] * nV
        self.V_init.init = [0] * nV

        offset = 0
        for k in range(self.ns):
            self.X.append(self.V.nz[offset : offset + self.nx])
            if k == 0:
                self.V_bounds.min[
                    offset : offset + self.nx
                ] = X_bounds.first_node_min
                self.V_bounds.max[
                    offset : offset + self.nx
                ] = X_bounds.first_node_max
            else:
                self.V_bounds.min[offset : offset + self.nx] = X_bounds.min
                self.V_bounds.max[offset : offset + self.nx] = X_bounds.max
            self.V_init.init[offset : offset + self.nx] = X_init.init
```

```
172             offset += self.nx
173
174             self.U.append(self.V.nz[offset : offset + self.nu])
175             if k == 0:
176                 self.V_bounds.min[
177                     offset : offset + self.nu
178                 ] = U_bounds.first_node_min
179                 self.V_bounds.max[
180                     offset : offset + self.nu
181                 ] = U_bounds.first_node_max
182             else:
183                 self.V_bounds.min[offset : offset + self.nu] = U_bounds.min
184                 self.V_bounds.max[offset : offset + self.nu] = U_bounds.max
185             self.V_init.init[offset : offset + self.nu] = U_init.init
186             offset += self.nu
187
188         self.X.append(self.V.nz[offset : offset + self.nx])
189         self.V_bounds.min[offset : offset + self.nx] = X_bounds.last_node_min
190         self.V_bounds.max[offset : offset + self.nx] = X_bounds.last_node_max
191         self.V_init.init[offset : offset + self.nx] = X_init.init
192
193     def solve(self):
194         # NLP
195         nlp = {"x": self.V, "f": self.J, "g": self.g}
196
197         opts = {
198             "ipopt.tol": 1e-6,
199             "ipopt.max_iter": 1000,
200             "ipopt.hessian_approximation": "exact",  # "exact", "limited-memory"
201             "ipopt.limited_memory_max_history": 50,
202             "ipopt.linear_solver": "mumps",  # "ma57", "ma86", "mumps"
203         }
204         solver = casadi.nlpsol("nlpsol", "ipopt", nlp, opts)
205
206         # Bounds and initial guess
207         arg = {
208             "lbx": self.V_bounds.min,
209             "ubx": self.V_bounds.max,
210             "lbg": self.g_bounds.min,
211             "ubg": self.g_bounds.max,
212             "x0": self.V_init.init,
213         }
214
```

```
215         # Solve the problem
216         return solver.call(arg)
217
218
219 class PathCondition:
220     @staticmethod
221     def regulation(var, nb_elements):
222         pass
223
224     @staticmethod
225     def regulation_private(var, nb_elements, type):
226         if len(var) != nb_elements:
227             raise RuntimeError(f"Invalid number of {type}")
228
229
230 class Bounds(PathCondition):
231     def __init__(self):
232         self.min = []
233         self.first_node_min = []
234         self.last_node_min = []
235
236         self.max = []
237         self.first_node_max = []
238         self.last_node_max = []
239
240     def regulation(self, nb_elements):
241         self.regulation_private(self.min, nb_elements, "Bound min")
242         self.regulation_private(self.max, nb_elements, "Bound max")
243
244         if len(self.first_node_min) == 0:
245             self.first_node_min = self.min
246         if len(self.last_node_min) == 0:
247             self.last_node_min = self.min
248
249         if len(self.first_node_max) == 0:
250             self.first_node_max = self.max
251         if len(self.last_node_max) == 0:
252             self.last_node_max = self.max
253
254         self.regulation_private(
255             self.first_node_min, nb_elements, "Bound first node min"
256         )
257         self.regulation_private(
```

```
258             self.first_node_max, nb_elements, "Bound first node max"
259         )
260         self.regulation_private(
261             self.last_node_min, nb_elements, "Bound last node min"
262         )
263         self.regulation_private(
264             self.last_node_max, nb_elements, "Bound last node max"
265         )
266
267
268 class InitialConditions(PathCondition):
269     def __init__(self):
270         self.first_node_init = []
271         self.init = []
272         self.last_node_init = []
273
274     def regulation(self, nb_elements):
275         if len(self.init) == 0:
276             self.init = [0] * nb_elements
277         self.regulation_private(self.init, nb_elements, "Init")
278
279         if len(self.first_node_init) == 0:
280             self.first_node_init = self.init
281         if len(self.last_node_init) == 0:
282             self.last_node_init = self.init
283
284         self.regulation_private(
285             self.first_node_init, nb_elements, "First node init"
286         )
287         self.regulation_private(
288             self.last_node_init, nb_elements, "Last node init"
289         )
```

Figure A.5: Fichier *biorbd_optim/__init__.py*

BiorbdOptim

07/04/20

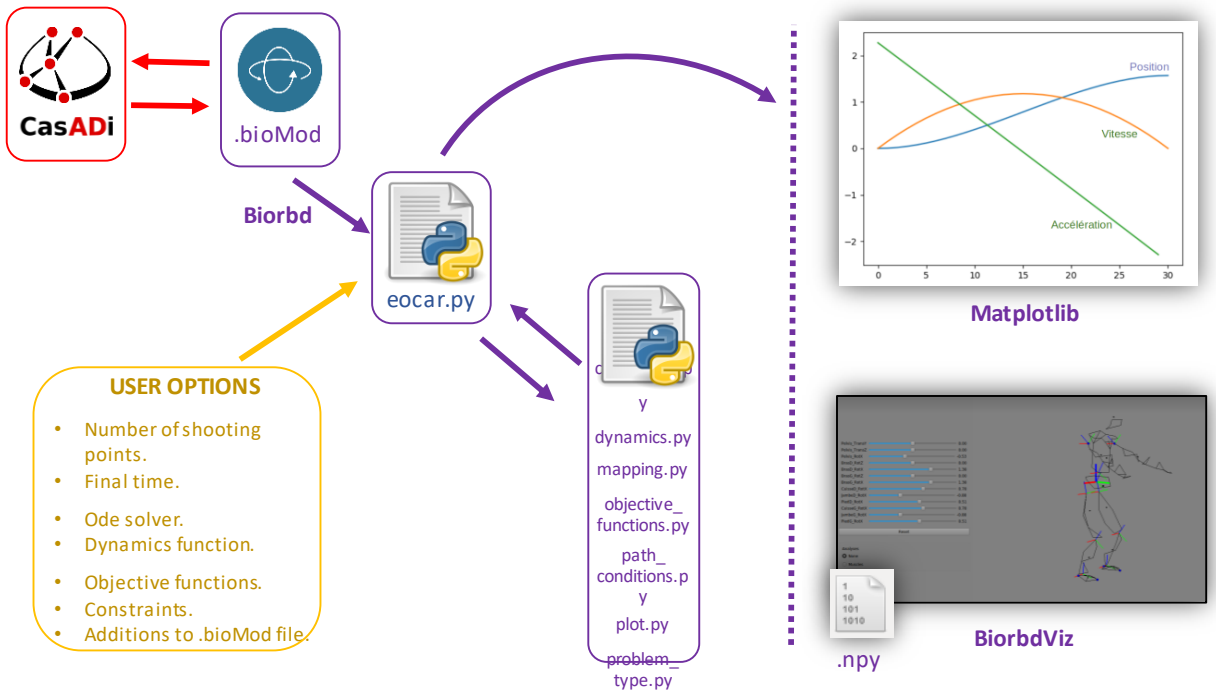
pyomeca /BiorbdOptim

https://github.com/pyomeca/BiorbdOptim

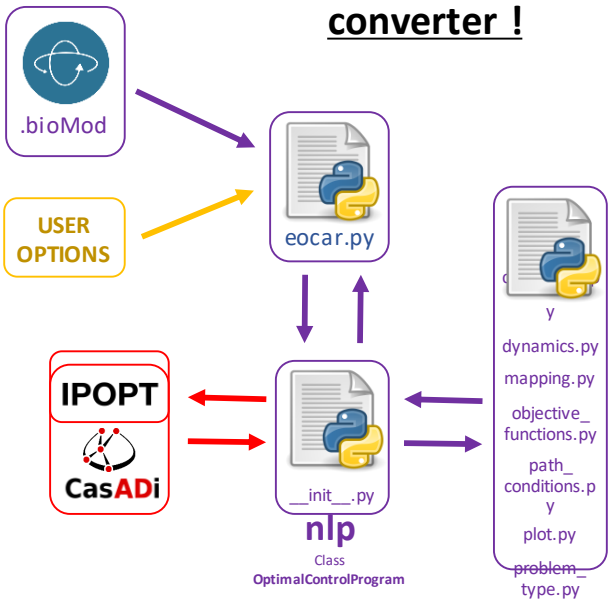
Developed by :

Benjamin MICHAUD
Paul WEGIEL
Théophile GOUSSELOT

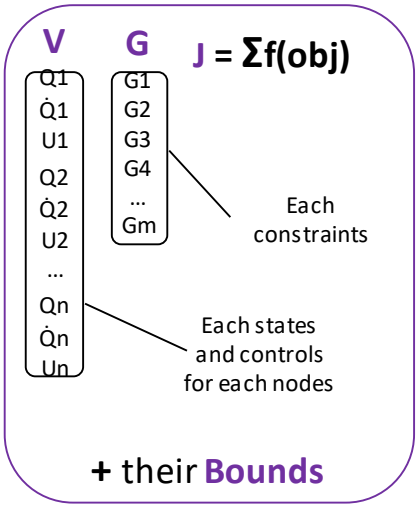
Paul WEGIEL &
Théophile GOUSSELOT



Nothing more than a converter !



What does Ipopt wants ?



Return **V** completed

NEW PROJECT

1 Copy/Paste eocar.py in new Folder and create new branch in Git.

2 Adds .bioMod file `.bioMod`

3 Adapts eocar.py to your problem

Have to change :

- Path to bioMod file.
- Number of shooting points.
- Final time.
- Ode solver.
- Dynamics function.
- Objective functions.
- Constraints.
- Bounds

⚠ **DO NOT TOUCH THE "BIORBD_OPTIM" FILES.**

ADDS FUNCTIONALITY

? New objective function ?

- Adds a static method in class ObjectiveFunction

`objective_functions.py`

? New constraint ?

- Adds a static method in class Constraint
- Adds a call to this function at the end of "add_constraints" method.

`constraints.py`

... Same method for all files.

⚠ Ask a Pull request.

⚠ DO NOT FORGET TO USE YOUR new functionnality in "eocar.py".

Figure A.6: Première présentation à l'attention des membres de l'équipe de commande optimale.

BiorbdOptim

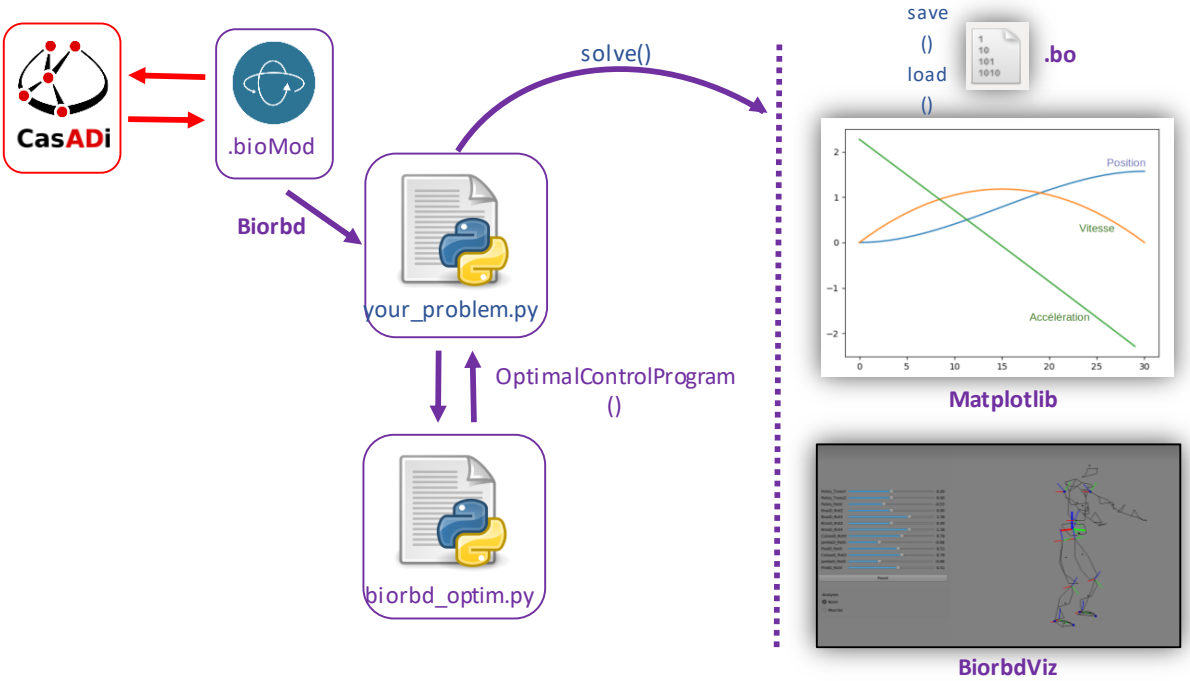
22/05/20

pyomeca /BiorbdOptim

https://github.com/pyomeca/BiorbdOptim

Developed by :

Paul WEGIEL &
Théophile GOUSSELOT



https://github.com/pyomeca/BiorbdOptim

NEW PROJECT

1 **Watch examples folder** Getting_started, ...

2 **Let's try !** examples/sandbox

ADDS FUNCTIONALITY

? **Constraint ? Objective ? Type of problem ? Etc.**

- Do your commits.
- Start a pull request to pyomeca

! **DON'T FORGET TO TEST YOUR COMMITS WITH PYTEST**
AND TO BLACK YOUR CODE (-l 120**)**

3 **Nothing to do ?**
https://github.com/pyomeca/BiorbdOptim/issues

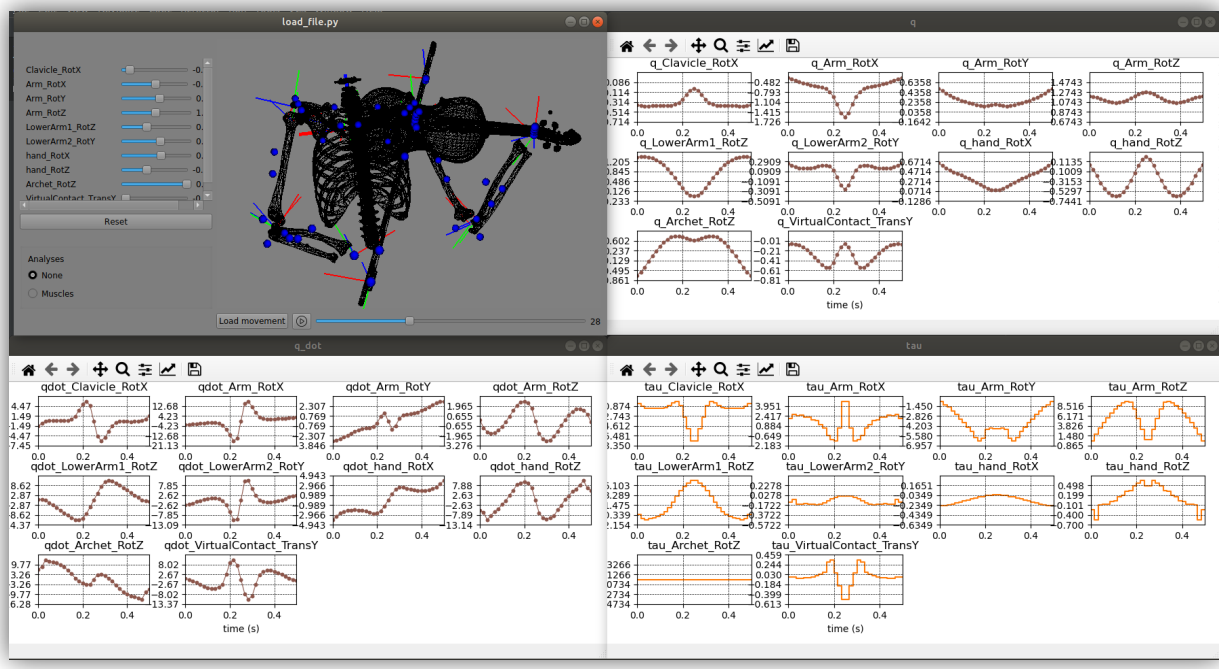


Figure A.7: Deuxième présentation à l'attention des membres du laboratoire S2M.

A. Bibliography

[1] Zhengru Ren, Roger Skjetne, and Zhen Gao. A crane overload protection controller for blade lifting operation based on model predictive control. *Energies*, 12:50, 12 2018.

[2] E. Marshal B. Ackermann, R. Adams. The effect of scapula taping on electromyographic activity and musical performance in professional violinists. *Australian Journal of Physiotherapy, vol.48, no. 3, pp. 197–203*, 2002.

Epilogue

A. List of Figures

A. List of Tables