red[1] red[2]red[3]

Bioptim, a Python interface for Musculoskeletal Optimal Control in Biomechanics

***Abstract*— The abstract**

Keywords – TODO

I. Introduction

Biomechanics researchers rely on numerical simulations of motion to gain understanding on a variety of scientific topics such as the physiological causes of movement disorders and their consequences on health [REF], the estimation of non-measurable physiological quantities (e.g., muscle forces)[REF] and the optimality of human movement [REF]. The musculoskeletal models used in these simulations generally have a large number of degrees of freedom and they are governed by several ordinary differential equations (ODEs) which mainly describe multibody and muscle activation dynamics. The complexity of these systems has led scientists to formulate their simulations as optimal control problems (OCP), relying on efficient non-linear optimization software to find trajectories that fulfill a desired task while enforcing the system dynamics and minimizing a cost (e.g. motion duration, energy expenditure, matching experimental data, etc.). Up to very recently, there was no off-the-shelf software available to the community to quickly formulate and solve such musculoskeletal OCPs. Consequently, researchers had to develop their own solutions, with little or no dissemination to the community, limiting synergies between researchers.

As a result, many approaches coexist to formulate and solve OCPs in the biomechanical literature. The formulation, also called discretization, consists in turning a continuous trajectory optimization problem into a generic discrete non-linear program (NLP) that is solved using a dedicated algorithm. The main family of so-called *direct* transcription methods comes from numerical optimal control. They consist in straightforwardly choosing the state and/or the control as optimization variables at a given number of points along the trajectory and they rely on the integration of the system dynamics between these points.

For instance, the *direct collocation* method has shown its efficiency in some studies investigating human motion [REF, à prendre dans papier MOCO]. It consists in approximating the integration of the system dynamics using polynomials that describe the state and control trajectories. Its main advantages are that it leads to very sparse NLPs, that knowledge about the state trajectory can be used in the initialization, and that it handles unstable systems well [**?**]. Its major disadvantage is that adaptive integration error control implies regridding the whole problem and thus changes the NLP dimensions, discarding its use for such application [REF]. *Direct multiple shooting* is another direct method that was also applied with success in a lot of biomechanics [REF] and robotics [REF] studies. Its advantages are mostly the same as for direct collocation in addition to combine integration error control with fixed NLP dimensions, as it relies on possibly adaptive ODE solvers to integrate the system dynamics. Besides direct methods, other choices can be made, as in [**?**] [+ REF Begon], where the optimization variables are instants at which a switch in the motor strategy occurs, using polynomials function (4th, 5th order) in-between, or in [**?**] [+ REF Huchez, Mombaur, McPhee, Opensim]], where the optimization variables are the coefficients of fourth order polynomial approximations of the states, with linking conditions to enforce the continuity of the controls. These last approaches are less generic than the direct methods as they either require a prior knowledge about the state and control trajectories. Most of the time, when investigating complex biomechanics issues, we do not have this information.

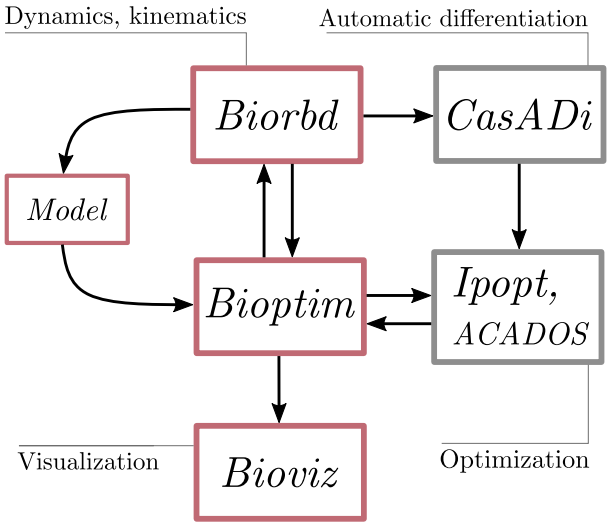

Fig. 1: *Bioptim* dependencies flowchart. The red-boxed software are developed by the S2M team. The *Bioptim* part is further detailed in Fig. 1.

Concerning the non-linear solver, a variety of software exist and have been used to solve transcribed musculoskeletal NLPs. They can use different heuristics: interior point methods (*Ipopt*, [REF]) or sequential quadratic programming (*snopt* [REF], *ACADOS* [REF]), but they are all gradient based.

† These authors have contributed equally to this work and share first authorship.
[a] Laboratoire de Simulation et Modélisation du Mouvement, Faculté de Médecine, Université de Montréal, Laval, QC, Canada
* BENJAM@umontreal.ca

Therefore, derivatives of the NLP cost function and constraints are required to perform optimization. These derivatives can be obtained by finite differences (often implemented but inaccurate thus comprising convergence) or computed exactly using automatic differentiation (requiring to write all dependencies of the software in symbolic variables) [CasADi].

In order to promote the use of musculoskeletal optimal control among biomechanics researcher, we identified a strong need for a dedicated tool, as shown by the recently launched *OpenSim Moco* [REF, Opensim]. The biomechanics community being mainly composed of software users [REF], such a tool should request a flexible user interface written in a widely used high-level and if possible open-source language (e.g. Python) with a low-level core (e.g. C++) for efficiency.

To develop such a software, four interrelated components are essential to us: *i)* a musculoskeletal modeling software, with a visualization module (multibody kinematics and dynamics, muscle dynamics, etc.), *ii)* a method for automatic differentiation, *iii)* a discretization approach, and *iv)* one or several nonlinear programming (NLP) solvers. General-purpose optimal control software (e.g. *Gpops-II* [REF], *Muscod-II* [REF], *Acado* [REF]) address *ii)* to *iv)* but they need to be interfaced with a musculoskeletal modeling module and they do not provide any built-in biomechanics features (physiological cost functions, kinematic constraints, etc.). In that sense, the aforementioned *OpenSim Moco*, is a welcome initiative that draws its strength from its integration with the widely used *OpenSim*. However, it faces the following limitations: it uses finite differences to avoid the complexity of adapting the *OpenSim* codebase to support automatic differentiation, it uses direct collocation as transcription method, preventing the use of adaptive ODE solvers and it is not as flexible as required by the community, since it requires the user to develop new features, such as new objective functions, in C++.

The objective of the present paper is to introduce *Bioptim*, an open-source optimal control software dedicated to musculoskeletal biomechanics. *Bioptim* is based on C++ code for computational efficiency but the user interface is written in Python for flexibility and ease-of-use. The OCP transcription uses direct multiple shooting to preserve the possibility of using arbitrarily accurate ODE solvers for the integration, which is fully parallelized for more efficiency. *Bioptim*'s core is fully written in *CasADi* symbolics to benefit from algorithmic differentiation and to exploit *CasADi*'s interface with several non-linear solvers (*Ipopt*, *Snopt*). Moreover, *Bioptim* is interfaced with the cutting-edge solver *acados*, a recent NLP solver dedicated to direct multiple shooting, intended for real-time applications. The purpose of *Bioptim* is to allow fast and flexible musculoskeletal OCP formulation and solving by providing a framework with a lot of typical biomechanics problem already implemented and customizable.

The paper is organized as follows: first, the design and implementation of *Bioptim* are described. Next, the versatility and performances of *Bioptim* are shown through a variety of examples available online.

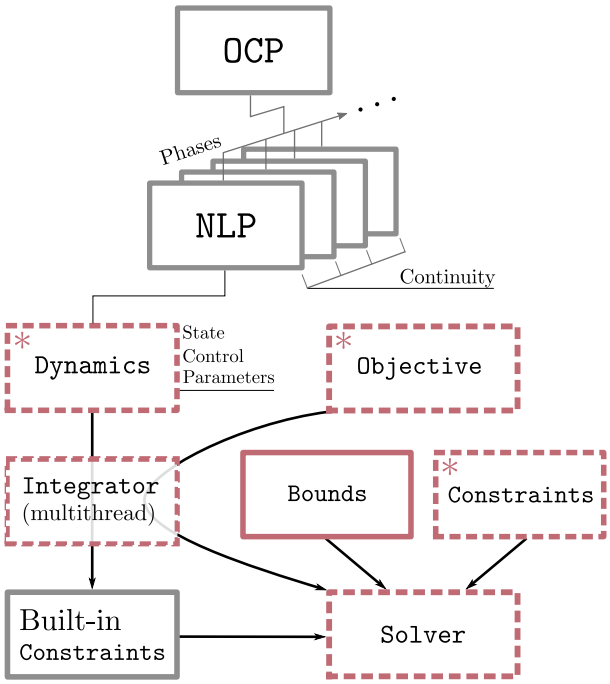

Fig. 2: *Bioptim* design flowchart. The red box correspond to objects that must be filled in by the user. The red-dashed boxes correspond to pre-implemented objects already available to the users. The * stands for easily customizable objects.

II. Implementation and Design

A. Implementation and dependencies

Bioptim is the top layer of a series of libraries (*Biorbd*: dynamics and MSK modeling; *CasADi*: automatic differentiation; *Ipopt*/*ACADOS*: optimization; *Bioviz*: visualization). Within this software suite, *Bioptim*'s main role is to shape the problem to allow its dependencies to communicate efficiently, while providing an intuitive and flexible interface to the user (Fig. 1). Therefore, it was written in Python for its flexibility and its widespread use among researchers. However, all intensive calculations behind the interface are performed in C/C++, keeping *Bioptim* both fast and easy to customize.

B. Design

Bioptim shapes and solves optimal control problems whose two required entries are a model (*.bioMod* file) and an OCP. The model file contains the geometrical characteristics and the segment inertial parameters as well as optional elements, namely, the markers, the actuators of the model (muscles and joint torques possibly with torque/angle/velocity relationships) as well as bounds on joint kinematics and torques. It also allows the user to design or import meshes for visualization purposes. The `OCP` consists in a combination of nonlinear problems (NLPs) that allows for the formulation of multi-staged OCPs. Each `NLP` has the following attributes: *1)* a dynamics type, *2)* an objective function set, *3)* a constraint set, *4)* variables bounds, *5)* a number of shooting points and the duration of the problem and *6)* initial guesses. Based on these inputs, *Bioptim* properly sets up the multiple shooting transcription of the OCP, with appropriate continuity

constraints (between the shooting nodes and the phases) and shapes it up to feed the chosen nonlinear solver (*Ipopt* or *ACADOS*). Next, we develop the different attributes of each `NLP`:

1) Dynamics: The dynamics defines which variables are states ($\mathbf{x}$), controls ($\mathbf{u}$) and parameters ($\mathbf{p}$), the latter being time-independent. Then, it implements the ordinary differential equation governing the state dynamics:

$$\dot{\mathbf{x}} = f(\mathbf{x}, \mathbf{u}, \mathbf{p}). \tag{1}$$

More than 10 dynamics are already implemented in *Bioptim*[1], among which the controls can be muscle excitations, muscle activations and/or joint torques, the states can be muscle activations and/or joint kinematics. They can include contact points, external forces, etc. Even if these dynamics types exhaustively span the current usages in biomechanics, a custom dynamics type is also pre-implemented to easily customize problems.

2) Objective function set: In line with the optimal control formalism, there are two main types of objective functions, namely Lagrange and Mayer. Lagrange types are running objectives, integrated over the NLP duration. Mayer types are time-specific objectives. Classically, they correspond to a terminal objective, but to be more versatile, they can be defined at any instant in *Bioptim*.

Objective functions can depend on any of the optimization variable, *i.e.* the controls, the states, the parameters and the duration of the problem. A lot of objective function types are already implemented in *Bioptim* ($>$ 20), among which tracking / minimizing, on states / controls / markers / contact forces / problem duration, etc. Should one go missing, a custom objective type is also possible to define.

When declaring the desired list of objective functions for a given NLP, each objective function type is associated with a weight, and the user can choose on which components of the vector variables the objective must apply. If applicable (for tracking objective functions mainly), the user must also specify the numerical target of the objective.

3) Constraint set: Classically, constraints are hard penalties of the optimization problem, i.e., a solution will not be considered optimal, unless all constraints (equality or inequality) are met. The `Constraint` class contains a variety of already implemented constraints. Some of them are specific functions, commonly useful in biomechanical problems (e.g. non-slipping contact point, non-linear bounds on torque depending on the state, etc.), the others have their equivalent in the `ObjectiveFunction` class. Should one go missing, a custom constraint type is also possible to define.

4) Bounds: Essentially, the `Bounds` are constraints directly related to the states, the controls and the parameters. They are useful for defining model-related constraints such as kinematic, torque or muscle excitation / activation limits.

[1] github link

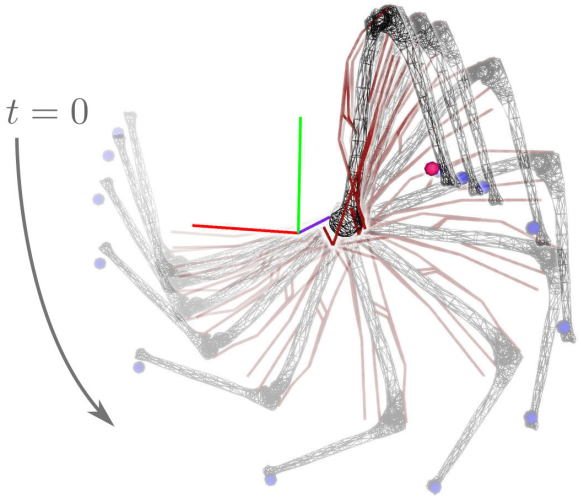

Fig. 3: Snapshots of an optimized activation-driven pointing task with *ACADOS*. The arm starts facing upwards in left hand part of the picture and ends facing downwards in the right hand part. The marker fized on the Ulna is depicted in blue and the scene-fixed target marker is depicted in red. Red lines show the lines of actions of the muscles.

5) Shooting points and problem duration: In a direct multiple shooting formulation, the total duration of the problem is divided into smaller intervals whose initial values are called shooting points. In *Bioptim*, the user is asked to define a number of shooting points and a problem duration, per phase. Possibly, the problem duration can be part of the optimization variables, allowing for, e.g., minimal time formulations.

6) Initial guesses: Once the problem is set up, the user can provide an `InitialGuess` for all the optimization variables, at each shooting point. This feature aims at providing prior information to the solver. Several `InterpolationTypes` are implemented in *Bioptim* (constant, linear, spline, each shooting point, etc.), to quickly let the user define the initial guesses. A custom `InterpolationType` is also possible to implement.

III. Examples

In this section, six applications are presented to illustrate the versatility of *Bioptim* and give a practical overview on how to use its main features. The settings and performances (convergence time, single shooting integration error, optimized objective) of each OCP are summarized in Tab. **??**. When possible, problems were solved with both *Ipopt* and *ACADOS*. In the following, bold symbols denote vectors and starred ones denote reference or tracked quantities.

A. Muscle activation driven pointing task

In this first example, the goal was to achieve a muscle activation driven pointing task using a 2-DoF arm model with 6 muscle elements. In addition to muscle-induced torques, pure joint torques were added to compensate for the model weaknesses. The main term (highest weight) of the objective

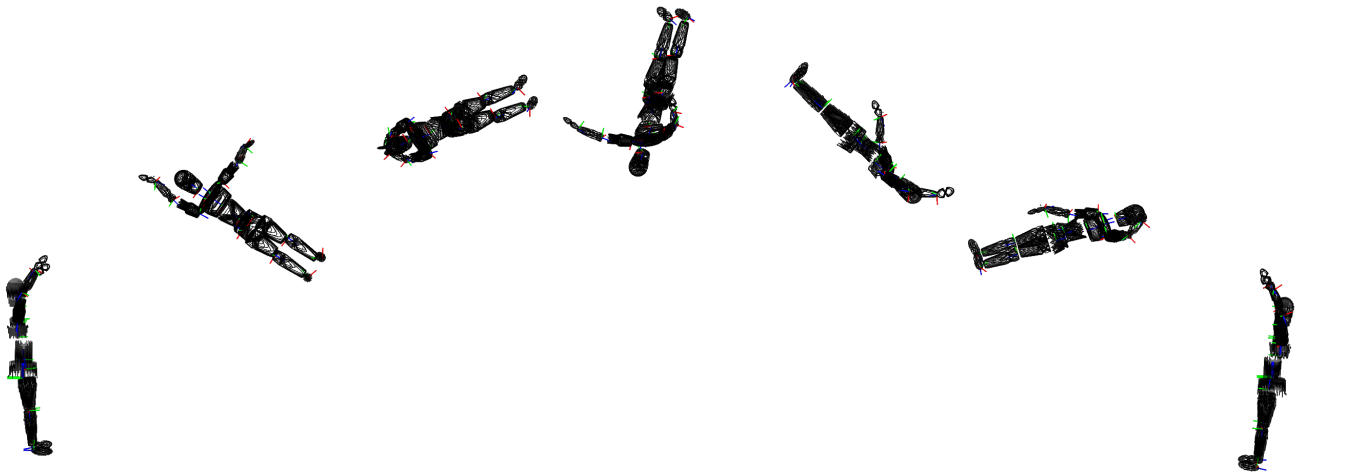

Fig. 4: Snapshots of maximally twisting somersaults driven by shoulder torque actuators and a free base whose rotation is either expressed by Euler angles (top) or by quaternions (bottom).

function (Eq. 2) is a Mayer objective, corresponding to the pointing tasks at the final node, to superimpose two markers, the first one, $\mathbf{m_u}$, fixed in the Ulna system of coordinates and the second one, $\mathbf{m_s^*}$, fixed in the scene. The three Lagrange terms were added for control regularization (muscle activation $\mathbf{a}$ and joint torques $\boldsymbol{\tau}$) and for state ($\mathbf{x}$) regularization:

$$\mathcal{C} = \omega_1 \underbrace{\|\mathbf{m_u}(T) - \mathbf{m_s^*}\|^2}_{\texttt{TRACK\_MARKERS}} \int_{t=0}^{T} \underbrace{\|\mathbf{a}\|^2}_{\texttt{MIN\_ACTIVATION}} + \underbrace{\|\boldsymbol{\tau}\|^2}_{\texttt{MIN\_TORQUE}} + \underbrace{\|\mathbf{x}\|^2}_{\texttt{MIN\_STATE}} \; dt, \tag{2}$$

where T is the duration of the motion, and $\omega_1 = 1e5$. The movement lasted for 2 seconds and was discretized using 50 shooting nodes with a 5-steps RK4 integration in-between. The problem was solved using *Ipopt* (with exact Hessian computations) and *ACADOS* (with a Gauss-Newton approximation of the Hessian) resulting in two very close solutions. *ACADOS* was about 50 times faster than *Ipopt* and was better at enforcing the continuity constraints (as shown by the single shooting error in Tab. **??**). *Ipopt* however ended up with a smaller optimized objective (20.8 *vs* 23.2), leading to a more optimal solution than *ACADOS*. Superimposed snapshots of the optimal motion found with *ACADOS* are displayed in Fig. 3. It is worth mentioning that for the purpose of this illustration, no constraint was given on the shoulder range of motion to ensure physiological muscle trajectories.

B. Quaternion base twisting somersault

[10pt]article [usenames]color amssymb amsmath [utf8]inputenc

$In this example of acrobatic sports biomechanics, the goal was to maximize the twist rotation ($

ϕ) of an 8-DoF model in a backward somersault and to illustrate *Bioptim*'s ability to handle quaternionic representations of rotations. The model was composed of a 6-DoF root segment and two 1-DoF torque actuated shoulder joints. Two different numerical descriptions of the root segment rotations were used: Euler angles and quaternions. The objective function was as follows:

$$\mathcal{J} = \int_0^T \underbrace{\omega_1 \dot{\phi}}_{\texttt{MIN\_TWIST}} + \underbrace{\omega_2 \|\boldsymbol{\tau}\|^2}_{\texttt{MIN\_TORQUE}} \; dt, \tag{3}$$

with $\omega_1 = -1$ (resulting in the maximization of the first term) and $\omega_2 = 10^{-6}$, T is the duration of the movement and $\boldsymbol{\tau}$ is the torque control vector. The first term of the objective function (Eq. 3) corresponds to maximizing the change in twist rotation and the second term is for control regularization.

The movement lasted for approximately 1 second and was discretized with 80 shooting nodes. The optimal kinematics were different for the two types of models (Fig. 4) because of the presence of local minima. However, both models take profits of a common biomechanical strategy (i.e. tilting the body to bring closer together the twist axis and the angular momentum vector) highlighting the equivalence of the two rotation representations. Euler angles have the advantage to be easily interpretable, but they suffer from the loss of a DoF at the Gimbal lock (leading to numerical instabilities). The use of a quaternion-based representation tackles this numerical stability issue when a joint is free to rotate on a three-dimensional range of motion. Quaternion's integration must be handled with care [**?**], indeed, when representing orientations, quaternions must be unitary and thus belong to a constrained manifold (namely, the unit 3-sphere S^3). However, classical numerical integration schemes such as Runge–Kutta methods treat unit quaternions as if they were arbitrarily defined in $\mathbb{R}^4$. To overcome this challenge, *Bioptim* performs a normalization after each Runge–Kutta iteration to project non-unitary quaternions onto S^3

(3)

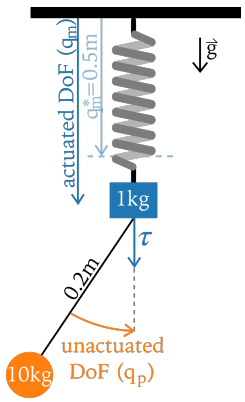

Fig. 5: Spring-mass-pendulum model of Ex. **??**.

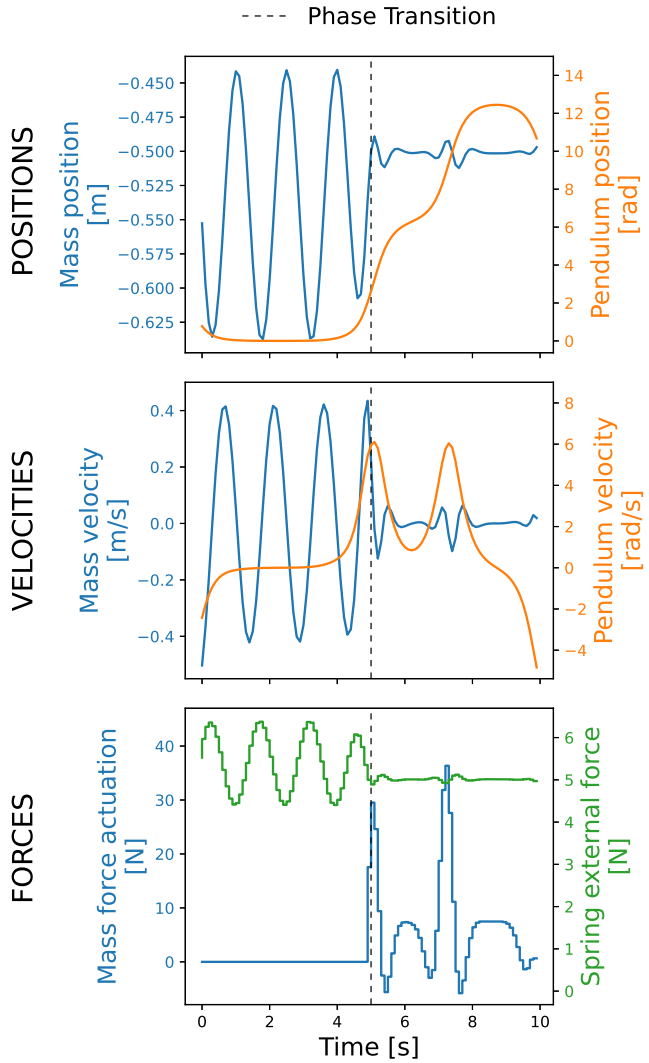

Fig. 6: Two-phases kinematics of the mass-pendulum-spring system. Gray dashed lines show the phase transition, blue lines are related to the mass (position velocity and external force acting on it), red lines are related to the pendulum (position and velocity) and the green line depicts the spring force.