Comparison of Optimal Control Software Frameworks in the Context of Bipedal Walking

Julian Eßer

February 13, 2020

Contents

Chapter 1

Introduction

Chapter 2

Crocoddyl - LAAS-CNRS

2.1 Introduction

Crocoddyl is an **optimal control library for robot control under contact sequence**. Its solver is based on an efficient Differential Dynamic Programming (DDP) algorithm. Crocoddyl computes optimal trajectories along with optimal feedback gains. It uses Pinocchio for fast computation of robot dynamics and its analytical derivatives.
Crocoddyl is focused on multi-contact optimal control problem (MCOP) which as the form:

$$\mathbf{X}^*, \mathbf{U}^* = \begin{Bmatrix} \mathbf{x}_0^*, \cdots, \mathbf{x}_N^* \\ \mathbf{u}_0^*, \cdots, \mathbf{u}_N^* \end{Bmatrix} = \arg\min_{\mathbf{X},\mathbf{U}} \sum_{k=1}^{N} \int_{t_k}^{t_k+\Delta t} l(\mathbf{x}, \mathbf{u}) dt$$

subject to

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, \mathbf{u}),$$
$$\mathbf{x} \in \mathcal{X}, \mathbf{u} \in \mathcal{U}, \boldsymbol{\lambda} \in \mathcal{K}.$$

where

- the state $\mathbf{x} = (\mathbf{q}, \mathbf{v})$ lies in a manifold, e.g. Lie manifold $\mathbf{q} \in SE(3) \times \mathbb{R}^{n_j}$, n_j being the number of degrees of freedom of the robot.
- the system has underactuacted dynamics, i.e. $\mathbf{u} = (\mathbf{0}, \boldsymbol{\tau})$,
- $\mathcal{X}, \mathcal{U}$ are the state and control admissible sets, and
- $\mathcal{K}$ represents the contact constraints.

Note that $\boldsymbol{\lambda} = \mathbf{g}(\mathbf{x}, \mathbf{u})$ denotes the contact force, and is dependent on the state and control.

2.2 Features

According to the Github repository [1], it comprises the following features:
Crocoddyl is **versatible**:

- various optimal control solvers (DDP, FDDP, BoxDDP, etc) - single and multi-shooting methods

- analytical and sparse derivatives via Pinocchio
- Euclidian and non-Euclidian geometry friendly via Pinocchio
- handle autonomous and nonautomous dynamical systems
- numerical differentiation support

Crocoddyl is **efficient** and **flexible**:

- cache friendly,
- multi-thread friendly
- Python bindings (including models and solvers abstractions)
- C++ 98/11/14/17/20 compliant
- extensively tested

2.3 Workflow

2.3.1 Define an Action Model (Dynamics+Costs)

In crocoddyl, an action model combines dynamics and cost models. Each node, in our optimal control problem, is described through an action model. In order to describe a problem, we need to provide ways of computing the dynamics, the cost functions and their derivatives. All these are described inside the action model.

To understand the mathematical aspects behind an action model, let's first get a locally linearize version of our optimal control problem as:

$$\mathbf{X}^*(\mathbf{x}_0), \mathbf{U}^*(\mathbf{x}_0) = \arg\max_{\mathbf{X},\mathbf{U}} = cost_T(\delta\mathbf{x}_N) + \sum_{k=1}^{N} cost_t(\delta\mathbf{x}_k, \delta\mathbf{u}_k)$$

subject to

$$dynamics(\delta\mathbf{x}_{k+1}, \delta\mathbf{x}_k, \delta\mathbf{u}_k) = \mathbf{0},$$

where

$$cost_T(\delta\mathbf{x}) = \frac{1}{2}\begin{bmatrix}1\\ \delta\mathbf{x}\end{bmatrix}^\top \begin{bmatrix}0 & \mathbf{l_x}^\top\\ \mathbf{l_x} & \mathbf{l_{xx}}\end{bmatrix}\begin{bmatrix}1\\ \delta\mathbf{x}\end{bmatrix}$$

$$cost_t(\delta\mathbf{x}, \delta\mathbf{u}) = \frac{1}{2}\begin{bmatrix}1\\ \delta\mathbf{x}\\ \delta\mathbf{u}\end{bmatrix}^\top \begin{bmatrix}0 & \mathbf{l_x}^\top & \mathbf{l_u}^\top\\ \mathbf{l_x} & \mathbf{l_{xx}} & \mathbf{l_{ux}}^\top\\ \mathbf{l_u} & \mathbf{l_{ux}} & \mathbf{l_{uu}}\end{bmatrix}\begin{bmatrix}1\\ \delta\mathbf{x}\\ \delta\mathbf{u}\end{bmatrix}$$

$$dynamics(\delta\mathbf{x}_{k+1}, \delta\mathbf{x}_k, \delta\mathbf{u}_k) = \delta\mathbf{x}_{k+1} - (\mathbf{f_x}\delta\mathbf{x}_k + \mathbf{f_u}\delta\mathbf{u}_k)$$

where an action model defines a **time interval** of this problem:

- $actions = dynamics + cost$

Important notes:

- An action model describes the dynamics and cost functions for a node in our optimal control problem.
- Action models lie in the discrete time space.
- For debugging and prototyping, we have also implemented numerical differentiation (NumDiff) abstractions.

These computations depend only on the definition of the dynamics equation and cost functions. However to asses efficiency, crocoddyl uses **analytical derivatives** computed from Pinocchio.

2.3.2 Differential Action Model

Optimal control solvers require the time-discrete model of the cost and the dynamics. However, it's often convenient to implement them in continuous time (e.g. to combine with abstract integration rules). In crocoddyl, this continuous-time action models are called "Differential Action Model (DAM)". And together with predefined "Integrated Action Models (IAM)", it possible to retrieve the time-discrete action model.
At the moment, we have:

- a simpletic Euler and
- a Runge-Kutte 4 integration rules.

An optimal control problem can be written from a set of DAMs as:

$$\mathbf{X}^*(\mathbf{x}_0), \mathbf{U}^*(\mathbf{x}_0) = \arg\max_{\mathbf{X},\mathbf{U}} = cost_T(\delta\mathbf{x}_N) + \sum_{k=1}^{N} \int_{t_k}^{t_k+\Delta t} cost_t(\delta\mathbf{x}_k, \delta\mathbf{u}_k) dt$$

subject to

$$dynamics(\delta\mathbf{x}_{k+1}, \delta\mathbf{x}_k, \delta\mathbf{u}_k) = \mathbf{0},$$

where

$$cost_T(\delta\mathbf{x}) = \frac{1}{2} \begin{bmatrix} 1 \\ \delta\mathbf{x} \end{bmatrix}^\top \begin{bmatrix} 0 & \mathbf{l_x}^\top \\ \mathbf{l_x} & \mathbf{l_{xx}} \end{bmatrix} \begin{bmatrix} 1 \\ \delta\mathbf{x} \end{bmatrix}$$

$$cost_t(\delta\mathbf{x}, \delta\mathbf{u}) = \frac{1}{2} \begin{bmatrix} 1 \\ \delta\mathbf{x} \\ \delta\mathbf{u} \end{bmatrix}^\top \begin{bmatrix} 0 & \mathbf{l_x}^\top & \mathbf{l_u}^\top \\ \mathbf{l_x} & \mathbf{l_{xx}} & \mathbf{l_{ux}}^\top \\ \mathbf{l_u} & \mathbf{l_{ux}} & \mathbf{l_{uu}} \end{bmatrix} \begin{bmatrix} 1 \\ \delta\mathbf{x} \\ \delta\mathbf{u} \end{bmatrix}$$

$$dynamics(\delta\dot{\mathbf{x}}, \delta\mathbf{x}, \delta\mathbf{u}) = \delta\dot{\mathbf{x}} - (\mathbf{f_x}\delta\mathbf{x} + \mathbf{f_u}\delta\mathbf{u})$$

Optimal control solvers often need to compute a quadratic approximation of the action model (as previously described); this provides a search direction (computeDirection). Then it's needed to try the step along this direction (tryStep).
Typically calc and calcDiff do the precomputations that are required before computeDirection and tryStep respectively (inside the solver). These functions update the information of:

- **calc**: update the next state and its cost value

$$\delta\dot{\mathbf{x}}_{k+1} = \mathbf{f}(\delta\mathbf{x}_k, \mathbf{u}_k)$$

- **calcDiff**: update the derivatives of the dynamics and cost (quadratic approximation)

$$\mathbf{f_x}, \mathbf{f_u} \quad (dynamics)$$

$$\mathbf{l_x}, \mathbf{l_u}, \mathbf{l_{xx}}, \mathbf{l_{ux}}, \mathbf{l_{uu}} \quad (cost)$$

2.3.3 Integrated Action Model

General speaking, the system's state can lie in a manifold M where the state rate of change lies in its tangent space $T_{\mathbf{x}}M$. There are few **operators that needs to be defined** for different rutines inside our solvers:

- $\mathbf{x}_{k+1} = integrate(\mathbf{x}_k, \delta\mathbf{x}_k) = \mathbf{x}_k \oplus \delta\mathbf{x}_k$
- $\delta\mathbf{x}_k = difference(\mathbf{x}_{k+1}, \mathbf{x}_k) = \mathbf{x}_{k+1} \ominus \mathbf{x}_k$

where $\mathbf{x} \in M$ and $\delta\mathbf{x} \in T_{\mathbf{x}}M$.

And we also need to defined the **Jacobians** of these operators with respect to the first and second arguments:

- $\frac{\partial \mathbf{x} \oplus \delta\mathbf{x}}{\partial \mathbf{x}}, \frac{\partial \mathbf{x} \oplus \delta\mathbf{x}}{\partial \delta\mathbf{x}} = Jintegrante(\mathbf{x}, \delta\mathbf{x})$
- $\frac{\partial \mathbf{x}_2 \ominus \mathbf{x}_2}{\partial \mathbf{x}_1}, \frac{\partial \mathbf{x}_2 \ominus \mathbf{x}_1}{\partial \mathbf{x}_1} = Jdifference(\mathbf{x}_2, \mathbf{x}_1)$

For instance, a state that lies in the Euclidean space will have the typical operators:

- $integrate(\mathbf{x}, \delta\mathbf{x}) = \mathbf{x} + \delta\mathbf{x}$
- $difference(\mathbf{x}_2, \mathbf{x}_1) = \mathbf{x}_2 - \mathbf{x}_1$
- $Jintegrate(\cdot, \cdot) = Jdifference(\cdot, \cdot) = \mathbf{I}$

These defines are encapsulated inside the State class. **For Pinocchio models, we have implemented the StatePinocchio class which can be used for any robot model.**

2.3.4 Solving the Optimal Control Problem

Our optimal control solver interacts with a defined ShootingProblem. A **shooting problem** represents a **stack of action models** in which an action model defines a specific node along the OC problem.

First we need to create an action model from DifferentialFwdDynamics. We use it for building terminal and running action models. In this example, we employ an simpletic Euler integration rule.

Next we define the set of cost functions for this problem. One could formulate

- Running costs (related to individual states)
- Terminal costs (related to the final state)

in order to penalize, for example, the state error, control error, or end-effector pose error.

Onces we have defined our shooting problem, we create a DDP solver object and pass some callback functions for analysing its performance.

2.4 Application to Bipedal Walking

In crocoddyl, we can describe the multi-contact dynamics through holonomic constraints for the support legs. From the Gauss principle, we have derived the model as:

$$\begin{bmatrix} \mathbf{M} & \mathbf{J}_c^{\top} \\ \mathbf{J}_c & \mathbf{0} \end{bmatrix} \begin{bmatrix} \dot{\mathbf{v}} \\ -\boldsymbol{\lambda} \end{bmatrix} = \begin{bmatrix} \boldsymbol{\tau} - \mathbf{h} \\ -\dot{\mathbf{J}}_c \mathbf{v} \end{bmatrix}$$

.

This DAM is defined in "DifferentialActionModelFloatingInContact" class.

Given a predefined contact sequence and timings, we build per each phase a specific multi-contact dynamics. Indeed we need to describe **multi-phase optimal control problem**. One can formulate the multi-contact optimal control problem (MCOP) as follows:

$$\mathbf{X}^*, \mathbf{U}^* = \begin{Bmatrix} \mathbf{x}_0^*, \cdots, \mathbf{x}_N^* \\ \mathbf{u}_0^*, \cdots, \mathbf{u}_N^* \end{Bmatrix} = \arg\min_{\mathbf{X},\mathbf{U}} \sum_{p=0}^{P} \sum_{k=1}^{N(p)} \int_{t_k}^{t_k+\Delta t} l_p(\mathbf{x}, \mathbf{u}) dt$$

subject to

$$\dot{\mathbf{x}} = \mathbf{f}_p(\mathbf{x}, \mathbf{u}), \text{for } t \in [\tau_p, \tau_{p+1}]$$

$$\mathbf{g}(\mathbf{v}^{p+1}, \mathbf{v}^p) = \mathbf{0}$$

$$\mathbf{x} \in \mathcal{X}_p, \mathbf{u} \in \mathcal{U}_p, \boldsymbol{\lambda} \in \mathcal{K}_p.$$

where $\mathbf{g}(\cdot, \cdot, \cdot)$ describes the contact dynamics, and they represents terminal constraints in each walking phase. In this example we use the following **impact model**:

$$\mathbf{M}(\mathbf{v}_{next} - \mathbf{v}) = \mathbf{J}_{impulse}^T$$

$$\mathbf{J}_{impulse} \mathbf{v}_{next} = \mathbf{0}$$

$$\mathbf{J}_c \mathbf{v}_{next} = \mathbf{J}_c \mathbf{v}$$

Chapter 3

Drake - MIT CSAIL

Chapter 4

Control Toolbox - ETH Zurich

Chapter 5

Comparison

Chapter 6

Conclusion and Outlook

Bibliography

[1] Rohan Budhiraja Carlos Mastalli, Nicolas Mansard, et al. Crocoddyl: a fast and flexible optimal control library for robot control under contact sequence. https://gepgitlab.laas.fr/loco-3d/crocoddyl/wikis/home, 2019.