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

Julian Eßer

February 26, 2020

Contents

Chapter 1

INTRODUCTION

Chapter 2

CROCODDYL - LAAS-CNRS

2.1 Introduction

2.1.1 Motivation

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 [1].
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.

2.1.2 Features

According to the description in 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

2.2 How-To

2.2.1 Install

2.2.1.1 Two ways to go

Basically there are existing two ways of installing Crocoddyl:

- Option 1: Installation via the *robotpkg* package manager
- Option 2: Installation from source

I personally would recommend the installation through *robotpkg*, since it preserves you from dealing with the multiple dependencies of Crocoddyl and therefore seems to be the faster approach. Generally you should decide beforehand which python version you want to use. This effects the robotpkg version as well as the export of the PYTHONPATH variable.

2.2.1.2 Installation via robotpkg (preferred)

Steps for installing via robotpkg according to the installation section of [1]

1. Add robotpkg as source repository to apt:

   ```
   sudo tee /etc/apt/sources.list.d/robotpkg.list <<EOF
   deb [arch=amd64] http://robotpkg.openrobots.org/wip/packages/debian/pub $(lsb_release -sc) robotpkg
   deb [arch=amd64] http://robotpkg.openrobots.org/packages/debian/pub $(lsb_release -sc) robotpkg
   EOF
   ```

2. Register the authentication certificate of robotpkg:

   ```
   curl http://robotpkg.openrobots.org/packages/debian/robotpkg.key | sudo apt-key add -
   ```

3. You need to run at least once apt update to fetch the package descriptions:

   ```
   sudo apt-get update
   ```

4. The installation of Crocoddyl:

   ```
   sudo apt install robotpkg-py27-crocoddyl # for Python 2
   sudo apt install robotpkg-py36-crocoddyl # for Python 3
   ```

5. Finally you will need to configure your environment variables (watch out for the python version!), e.g.:

   ```
   export PATH=/opt/openrobots/bin:$PATH
   export PKG_CONFIG_PATH=/opt/openrobots/lib/pkgconfig:$PKG_CONFIG_PATH
   export LD_LIBRARY_PATH=/opt/openrobots/lib:$LD_LIBRARY_PATH
   export PYTHONPATH=/opt/openrobots/lib/python3.6/site-packages:$PYTHONPATH
   ```

2.2.1.3 (Installation from source)

If you prefer installing Crocoddyl from source, the following steps should do the work:

```
git clone https://github.com/loco-3d/crocoddyl.git
git submodule update --init
mkdir build && cd build
export PKG_CONFIG_PATH=/opt/openrobots/lib/pkgconfig
cmake -DCMAKE_INSTALL_PREFIX=/opt/openrobots  ..
make
sudo make install
```

Additionally you will have to install the dependent libraries (i.e. pinocchio, example-robot-data (optional for examples, install Python loaders), gepetto-viewer-corba (optional for display), jupyter (optional for notebooks) and matplotlib (optional for examples) and fix the incude paths.

2.2.2 Running the Examples

Since the installation through robotpkg did not provide you with the examples from the git repository, you should clone the repo [1] for getting the data. You do not have to build the library, since it already is installed. In the cloned repository go to */examples*. For running e.g. the bipedal walking example, just type

```
python3 bipedal_walk.py
```

and you will see the calculations for optimal gait trajectories running in the console. You propably want to view your results now. For displaying the results, we need to install the gepetto-viewer:

```
 sudo apt install robotpkg-py36-qt4-gepetto-viewer-corba
```

The examples provide a *plot* and *display* argument. In order to display the 3D results and also plot some data, just do

```
gepetto-gui
```

for starting the 3D environment and then, in another terminal

```
python3 bipedal_walk.py display plot
```

2.3 Abstract Workflow

For each node (i.e. each timestep) of the optimal control problem,

1. Load robot data (URDF, SRDF, Meshes)
2. Define Action Models (Dynamics+Costs) for running and terminal states
   - Setup a cost model
   - Add the desired cost functions (state, control, frame-placement etc.)
   - Calculate Integrated & Differential Action Model (IAM/DAM) based on the model
3. Define the optimal control problem (knots+IAMs, initPose)
4. Solve the Shooting Problem

2.4 Issues and Insights

2.4.1 Issues encountered

Since Crocoddyl currently is under active development, there frequently will occure smaller incompatibilities because of versioning issues.This is a brief overview of emerged difficulties:

- Python versioning errors in the examples.

  The examples most often are written for python2, which means that if you are under python3, you will have to adapt some commands (e.g. lists handling, matplotlib, print).

- Crocoddyl versioning errors in the examples.

  Since Crocoddyl depends on other libraries (i.e. Pinnochio, example_robot_data), there sometimes occured errors with the class because they were not updated.

- Confusions displaying the results via the Gepetto-Viewer

  The Gepetto-Viewer is used for displaying the robots and resulting trajectories from optimization. The examples only contain out of the box solutions. If one wants to simply display a robot in some specified pose (e.g. the initial pose) the following, quite unintuitive, commands have to be applied:

  ```
  display = crocoddyl.GepettoDisplay(rh5_legs, 4, 4, frameNames=[rightFoot, leftFoot])
  display.display(xs=[x0])
  ```

2.4.2 Cost Functions

Notes:

- The cost function can contain multiple *cost items* (i.e state/control error, frame displacements or center of mass tracking).
- Weights are considered in the costs via scalar multiplication with the identity matrices (Ix, Ixx etc.) of the according cost item.
- These weighted matrices of cost items are simply summed up within a *costModelContainer*.

2.4.3 Joint Limits

- Input Data: Within the URDF file, for each joint there are specified the
  - torque limit (effort),
  - position limits (lower, upper),
  - velocity limit.
- These limits are not automatically taken into account in Crocoddyl when solving a shooting problem, but explicitly have to be adressed.
- **Torque Limits**: Require the use of specific solvers, standard ddp is not sufficient. Implemented solvers that can handle torque limits explicitly are:
  - BoxDDP (Compare Tassa method [2])
  - BoxFDDP (Novel solver that is under development at LAAS)

  State Limits (Pos/Vel): Position and Velocity limits can be handled via penalization, i.e. added as cost terms to the optimization problem.

2.5 Discussion of Examples and RH5 Integration

This section contains a brief overview of some of the examples that have been modified. Please note that **core explanations have been added to the examples where seemed appropriate**. All figures and additionaly videos are contained in this repository within the /media directory.

As stated in the ReadMe of the Repo, Crocoddyl comes with some introductory examples that are written as Jupyter notebooks (.ipynb). While

```
examples/notebooks/introduction_to_crocoddyl.ipynb
```

offers a more conceptual overview about crocoddyl, other ones represent basic underactuated systems (e.g. Cartpole swing-up, Bipdeal Walking).

2.5.1 Manipulator: Multi-Point Trajectory

The task in the tutorial

```
examples/notebooks/arm_manipulation.ipynb
```

was to find an optimal trajectory for a manipulator from an initial configuration to a target point (red ball).

2.5.1.1 Extending the Example

I extended this example to a multi-point optimal control problem by considering four targets to reach in a row. This extended example can be found in

```
/examples/arm_manipulation_trajectory.py
```

Differences to the existing one-target example include:

- Defining an array of the four targets in space
- Setup individual cost functions for each of the sequences
- Setup running and terminal models for the sequences and finally
- Define the shooting problem as row of these sequences
- Optimize the weights of the cost functions successively for the four sequences.

2.5.1.2 Results: Multiple Targets

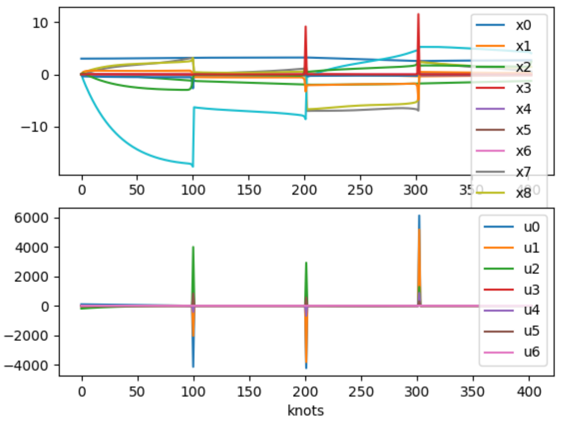


Figure 2.1: Multi-Point Optimal Control of the Manipulator for reaching four targets. The high-control peaks could be handled by a more advanced solver (e.g. box-ddp), but were not performed here.

2.5.2 Introduction: Bipedal Walking in Crocoddyl

- A long walk consists of multiple gaitphases, each phase is a single shooting problem.
- These problems are generated with *createWalkingProblem()* involving one left and one right foot step.
- Each shooting problem contains various locomotion phases
  - Double support at beginning (both legs on ground) via *createSwingFootModel()*
  - Right step (Swing-up and swing-down phase equally distributed) init via *createFootstepModels()*
  - Double support again
  - Left step
- In the end, all knots of all phases are basically one *swingFootModel*. They only vary in the adressed foot, and if there is a CoM task or a swingFootTask activated.
- The *swingFootModel* is an IAM containing a
  - 6D multi-contact model,
  - Cost model (CoM position tracking, contact friction cone, foot placement) and
  - Differentiation (DAM) and Integration (IAM) routines.

2.5.3 Bipdeal Walking: RH5 Legs

2.5.3.1 Integration Work

The main issue that had to be solved was related to the underlying URDF file of the robot. In particular:

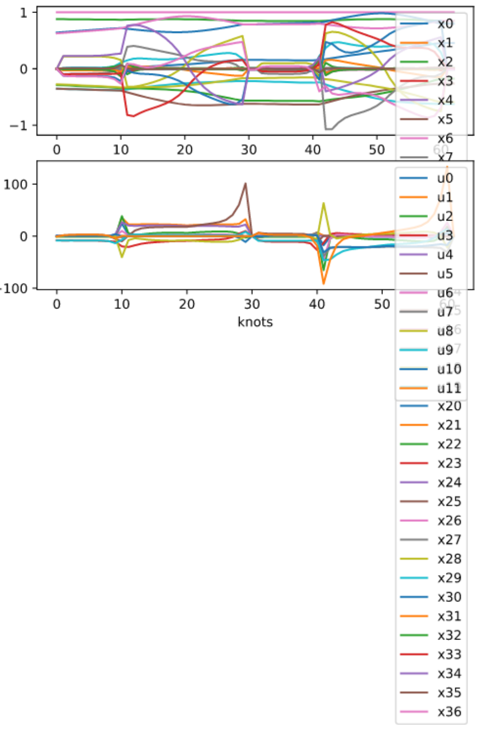
- Choosing one of our several files (abstract-smurf)
- Cutting out everything apart from the legs and the root joint
- Fixing the mesh file paths:
  We usually define the path relative to the URDF file location, e.g.

  `"../meshes/stl/RH5_Root_Link.001.stl".`

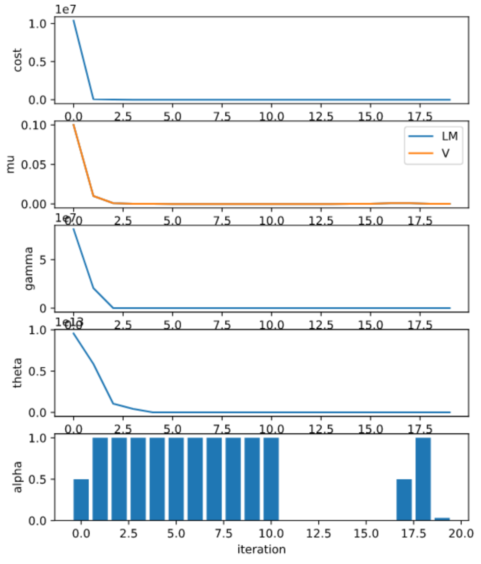
  The integrated URDF parser in Crocoddyl instead, expects the path specified via the package URI, e.g.

  `"package://abstract-smurf/meshes/stl/RH5_Root_Link.001.stl".`
- Adjust the contact frames for the Walking Problem.

2.5.3.2 Results: Performing two Steps



(a) Optimal Trajectory and Conrol Inputs



(b) Convergence of Solution

Chapter 3

DRAKE - MIT CSAIL

Chapter 4

CONTROL TOOLBOX - ETH Zurich

Chapter 5

COMPARISON

Chapter 6

CONCLUSION AND OUTLOOK

Appendix A

Background: Crocoddyl Workflow

This chapter contains details on the workflow in Crocoddyl and presents some of the underlying math. This information can be found in [1] within examples/notebooks/introduction_to_crocoddyl.ipynb.

A.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.

A.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)

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

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

A.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.**

A.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.

A.5 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**:

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

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

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

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.

[2] Yuval Tassa, Nicolas Mansard, and Emo Todorov. Control-limited differential dynamic programming. In *2014 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1168–1175. IEEE, 2014.