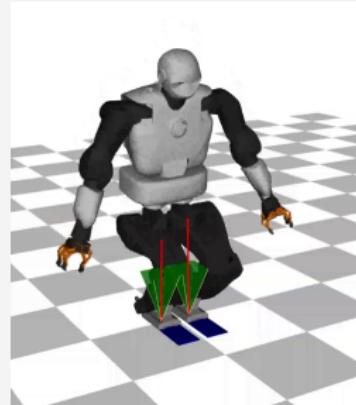
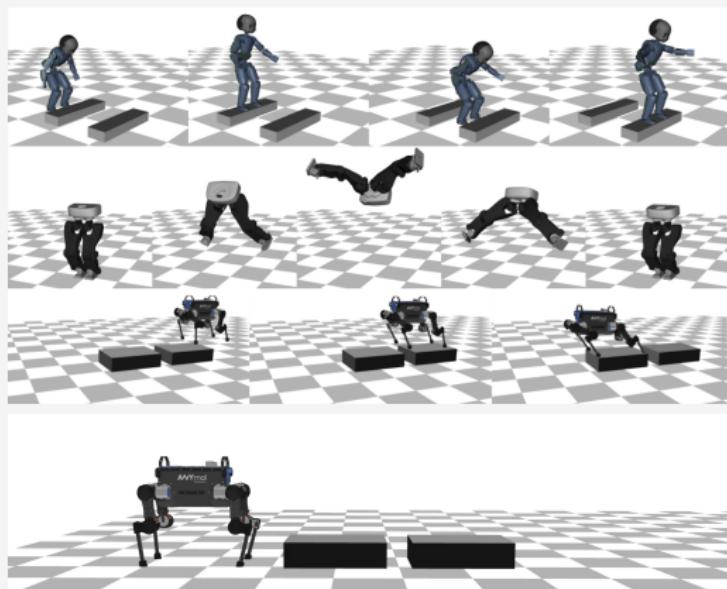


Crocoddyl: An Efficient Multi-Contact Optimal Control Framework

Implementation and tutorial



Carlos Mastalli
University of Edinburgh

The
Alan Turing
Institute



Overview

1. Introduction

2. Core API 1.0

Exercise: unicycle towards the origin

3. Core API 2.0

Exercise: cartpole swing up

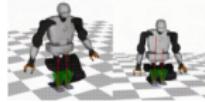
4. Contact dynamics API

Exercise: whole-body manipulation

Exercise: bipedal walking (optional)

Introduction

Contact RObot COnrol by Differential DYnamic programming Library (crocoddyl)



► Introduction

Crocoddyl is an optimal control library for robot control under contact sequence. Its solvers are based on novel and efficient Differential Dynamic Programming (DDP) algorithms. **Crocoddyl** computes optimal trajectories along with optimal feedback gains. It uses **Pinocchio** for fast computation of robots dynamics and their analytical derivatives.

The source code is released under the [BSD 3-Clause license](#).

Authors: [Carlos Mastalli](#) and [Rohan Budhiraja](#)

Instructors: Nicolas Mansard

With additional support from the Gepetto team at LAAS-CNRS and MEMMO project. For more details see [Section Credits](#)

¹<https://github.com/loco-3d/crocoddyl>

Contact RObot COntrol by Differential DYnamic programming Library (crocoddyl)



Crocoddyl is an optimal control library for robot control under contact sequence. Its solvers are based on novel and efficient Differential Dynamic Programming (DDP) algorithms. **Crocoddyl** computes optimal trajectories along with optimal feedback gains. It uses **Pinocchio** for fast computation of robots dynamics and their analytical derivatives.

The source code is released under the [BSD 3-Clause license](#).

Authors: [Carlos Mastalli](#) and [Rohan Budhiraja](#)

Instructors: Nicolas Mansard

With additional support from the Gepetto team at LAAS-CNRS and MEMMO project. For more details see
[Section Credits](#)

¹<https://github.com/loco-3d/crocoddyl>

Contact RObot COntrol by Differential DYnamic programming Library (crocoddyl)



Crocoddyl is an optimal control library for robot control under contact sequence. Its solvers are based on novel and efficient Differential Dynamic Programming (DDP) algorithms. **Crocoddyl** computes optimal trajectories along with optimal feedback gains. It uses **Pinocchio** for fast computation of robots dynamics and their analytical derivatives.

The source code is released under the [BSD 3-Clause license](#).

Authors: [Carlos Mastalli](#) and [Rohan Budhiraja](#)

Instructors: Nicolas Mansard

With additional support from the Gepetto team at LAAS-CNRS and MEMMO project. For more details see
[Section Credits](#)

¹<https://github.com/loco-3d/crocoddyl>

Contact RObot COnrol by Differential DYnamic programming Library (crocoddyl)



Crocoddyl is an optimal control library for robot control under contact sequence. Its solvers are based on novel and efficient Differential Dynamic Programming (DDP) algorithms. **Crocoddyl** computes optimal trajectories along with optimal feedback gains. It uses **Pinocchio** for fast computation of robots dynamics and their analytical derivatives.

The source code is released under the [BSD 3-Clause license](#).

Authors: [Carlos Mastalli](#) and [Rohan Budhiraja](#)

Instructors: Nicolas Mansard

With additional support from the Gepetto team at LAAS-CNRS and MEMMO project. For more details see
[Section Credits](#)

¹<https://github.com/loco-3d/crocoddyl>

Contact RObot COnrol by Differential DYnamic programming Library (crocoddyl)



Crocoddyl is an optimal control library for robot control under contact sequence. Its solvers are based on novel and efficient Differential Dynamic Programming (DDP) algorithms. **Crocoddyl** computes optimal trajectories along with optimal feedback gains. It uses **Pinocchio** for fast computation of robots dynamics and their analytical derivatives.

The source code is released under the [BSD 3-Clause license](#).

Authors: [Carlos Mastalli](#) and [Rohan Budhiraja](#)

Instructors: Nicolas Mansard

With additional support from the Geppetto team at LAAS-CNRS and MEMMO project. For more details see
[Section Credits](#)

¹<https://github.com/loco-3d/crocoddyl>

Contact RObot COnrol by Differential DYnamic programming Library (crocoddyl)



Crocoddyl is an optimal control library for robot control under contact sequence. Its solvers are based on novel and efficient Differential Dynamic Programming (DDP) algorithms. **Crocoddyl** computes optimal trajectories along with optimal feedback gains. It uses **Pinocchio** for fast computation of robots dynamics and their analytical derivatives.

The source code is released under the [BSD 3-Clause license](#).

Authors: [Carlos Mastalli](#) and [Rohan Budhiraja](#)

Instructors: Nicolas Mansard

With additional support from the Gepetto team at LAAS-CNRS and MEMMO project. For more details see
[Section Credits](#)

¹<https://github.com/loco-3d/crocoddyl>

Main features

Crocoddyl is versatile:

- ▶ various optimal control solvers

¹<https://github.com/loco-3d/crocoddyl>

Main features

Crocoddyl is versatile:

- ▶ various optimal control solvers
- ▶ single and multi-shooting methods

¹<https://github.com/loco-3d/crocoddyl>

Main features

Crocoddyl is versatile:

- ▶ various optimal control solvers
- ▶ single and multi-shooting methods
- ▶ analytical and sparse derivatives

¹<https://github.com/loco-3d/crocoddyl>

Main features

Crocoddyl is versatile:

- ▶ various optimal control solvers
- ▶ single and multi-shooting methods
- ▶ analytical and sparse derivatives
- ▶ Euclidean and non-Euclidean geometry friendly

¹<https://github.com/loco-3d/crocoddyl>

Crocoddyl is versatile:

- ▶ various optimal control solvers
- ▶ single and multi-shooting methods
- ▶ analytical and sparse derivatives
- ▶ Euclidean and non-Euclidean geometry friendly
- ▶ autonomous and non-autonomous systems

¹<https://github.com/loco-3d/crocoddyl>

Crocoddyl is versatile:

- ▶ various optimal control solvers
- ▶ single and multi-shooting methods
- ▶ analytical and sparse derivatives
- ▶ Euclidean and non-Euclidean geometry friendly
- ▶ autonomous and non-autonomous systems
- ▶ numerical and automatic differentiation support

¹<https://github.com/loco-3d/crocoddyl>

Main features

Crocoddyl is efficient and flexible:

- ▶ cache friendly

¹<https://github.com/loco-3d/crocoddyl>

Main features

Crocoddyl is efficient and flexible:

- ▶ cache friendly
- ▶ multi-thread friendly

¹<https://github.com/loco-3d/crocoddyl>

Main features

Crocoddyl is efficient and flexible:

- ▶ cache friendly
- ▶ multi-thread friendly
- ▶ Python bindings (including models and solvers abstractions)

¹<https://github.com/loco-3d/crocoddyl>

Main features

Crocoddyl is efficient and flexible:

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

¹<https://github.com/loco-3d/crocoddyl>

Main features

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

¹<https://github.com/loco-3d/crocoddyl>

Main features

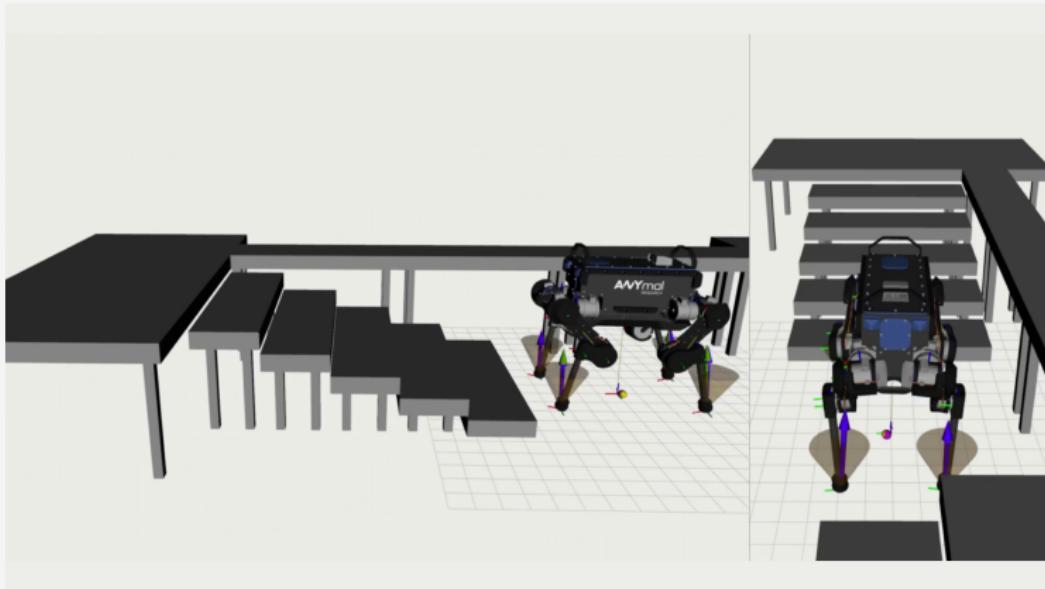
Crocoddyl is efficient and flexible:

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

¹<https://github.com/loco-3d/crocoddyl>

Scope and Motivation

*Fast whole-body model predictive control for legged robots
... to generate motion within the actuation limits*

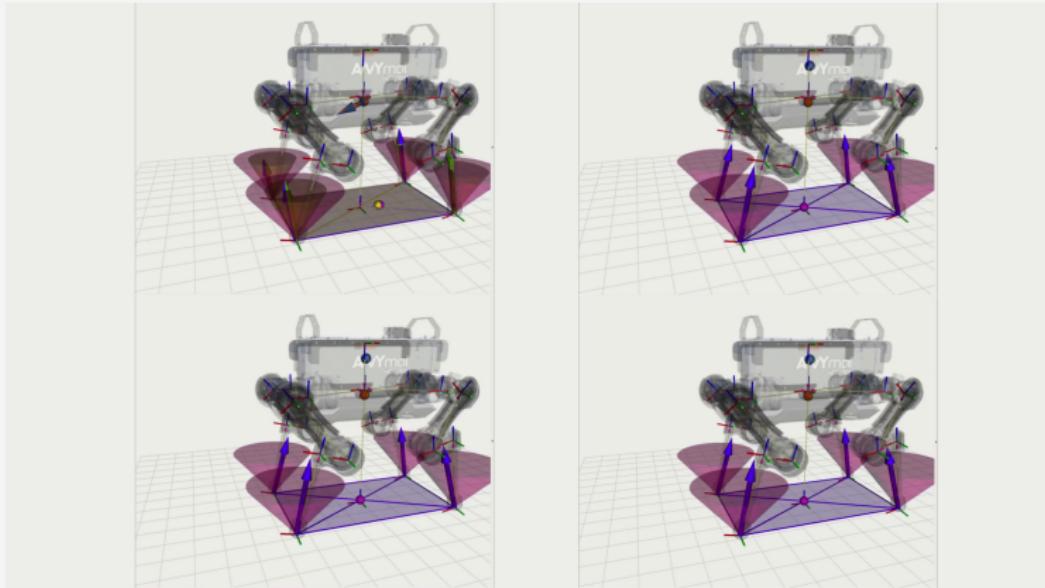


The
Alan Turing
Institute



Scope and Motivation

*Fast whole-body model predictive control for legged robots
... to regulate attitude in highly-dynamic maneuvers*



The
Alan Turing
Institute



Optimal control problem

$$\begin{aligned} \min_{\mathbf{x}, \mathbf{u}} \quad & l_N(\mathbf{x}_N) + \sum_{k=0}^{N-1} l_k(\mathbf{x}_k, \mathbf{u}_k) \\ \text{s.t.} \quad & \mathbf{x}_{k+1} = \mathbf{f}_k(\mathbf{x}_k, \mathbf{u}_k) \\ & \mathbf{g}_k(\mathbf{x}_k, \mathbf{u}_k) \leq \mathbf{0} \\ & \mathbf{x}_k \in \mathcal{X}, \mathbf{u}_k \in \mathcal{U} \end{aligned}$$

- ▶ terminal and running costs

Optimal control problem

$$\begin{aligned} \min_{\mathbf{x}, \mathbf{u}} \quad & l_N(\mathbf{x}_N) + \sum_{k=0}^{N-1} l_k(\mathbf{x}_k, \mathbf{u}_k) \\ \text{s.t.} \quad & \mathbf{x}_{k+1} = \mathbf{f}_k(\mathbf{x}_k, \mathbf{u}_k) \\ & \mathbf{g}_k(\mathbf{x}_k, \mathbf{u}_k) \leq \mathbf{0} \\ & \mathbf{x}_k \in \mathcal{X}, \mathbf{u}_k \in \mathcal{U} \end{aligned}$$

- ▶ terminal and running costs
- ▶ state lies in a differentiable manifold $\mathbf{x}_i \in \mathcal{Q}$

Optimal control problem

$$\begin{aligned} \min_{\mathbf{x}, \mathbf{u}} \quad & l_N(\mathbf{x}_N) + \sum_{k=0}^{N-1} l_k(\mathbf{x}_k, \mathbf{u}_k) \\ \text{s.t.} \quad & \mathbf{x}_{k+1} = \mathbf{f}_k(\mathbf{x}_k, \mathbf{u}_k) \\ & \mathbf{g}_k(\mathbf{x}_k, \mathbf{u}_k) \leq \mathbf{0} \\ & \mathbf{x}_k \in \mathcal{X}, \mathbf{u}_k \in \mathcal{U} \end{aligned}$$

- ▶ terminal and running costs
- ▶ state lies in a differentiable manifold $\mathbf{x}_i \in \mathcal{Q}$
- ▶ system dynamics

Optimal control problem

$$\begin{aligned} \min_{\mathbf{x}, \mathbf{u}} \quad & l_N(\mathbf{x}_N) + \sum_{k=0}^{N-1} l_k(\mathbf{x}_k, \mathbf{u}_k) \\ \text{s.t.} \quad & \mathbf{x}_{k+1} = \mathbf{f}_k(\mathbf{x}_k, \mathbf{u}_k) \\ & \mathbf{g}_k(\mathbf{x}_k, \mathbf{u}_k) \leq \mathbf{0} \\ & \mathbf{x}_k \in \mathcal{X}, \mathbf{u}_k \in \mathcal{U} \end{aligned}$$

- ▶ terminal and running costs
- ▶ state lies in a differentiable manifold $\mathbf{x}_i \in \mathcal{Q}$
- ▶ system dynamics
- ▶ path constraints

Optimal control problem

$$\begin{aligned} \min_{\mathbf{x}, \mathbf{u}} \quad & l_N(\mathbf{x}_N) + \sum_{k=0}^{N-1} l_k(\mathbf{x}_k, \mathbf{u}_k) \\ \text{s.t.} \quad & \mathbf{x}_{k+1} = \mathbf{f}_k(\mathbf{x}_k, \mathbf{u}_k) \\ & \mathbf{g}_k(\mathbf{x}_k, \mathbf{u}_k) \leq \mathbf{0} \\ & \mathbf{x}_k \in \mathcal{X}, \mathbf{u}_k \in \mathcal{U} \end{aligned}$$

- ▶ terminal and running costs
- ▶ state lies in a differentiable manifold $\mathbf{x}_i \in \mathcal{Q}$
- ▶ system dynamics
- ▶ path constraints
- ▶ state and control admissible sets

Core API 1.0

A Key Concept

To increase efficiency, we assume a Markovian problem

$$\begin{aligned} \min_{\mathbf{x}, \mathbf{u}} \quad & l_N(\mathbf{x}_N) + \sum_{k=0}^{N-1} l_k(\mathbf{x}_k, \mathbf{u}_k) \\ \text{s.t.} \quad & \mathbf{x}_{k+1} = \mathbf{f}_k(\mathbf{x}_k, \mathbf{u}_k) \\ & \mathbf{g}_k(\mathbf{x}_k, \mathbf{u}_k) \leq \mathbf{0} \\ & \mathbf{x}_k \in \mathcal{X}, \mathbf{u}_k \in \mathcal{U} \end{aligned}$$

A Key Concept

To increase efficiency, we assume a Markovian problem

$$\begin{aligned} \min_{\mathbf{x}, \mathbf{u}} \quad & l_N(\mathbf{x}_N) + \sum_{k=0}^{N-1} l_k(\mathbf{x}_k, \mathbf{u}_k) \\ \text{s.t.} \quad & \mathbf{x}_{k+1} = \mathbf{f}_k(\mathbf{x}_k, \mathbf{u}_k) \\ & \mathbf{g}_k(\mathbf{x}_k, \mathbf{u}_k) \leq \mathbf{0} \\ & \mathbf{x}_k \in \mathcal{X}, \mathbf{u}_k \in \mathcal{U} \end{aligned}$$

A Key Concept

To increase efficiency, we assume a Markovian problem

$$\begin{aligned} \min_{\mathbf{x}, \mathbf{u}} \quad & l_N(\mathbf{x}_N) + \sum_{k=0}^{N-1} l_k(\mathbf{x}_k, \mathbf{u}_k) && \text{(cost)} \\ \text{s.t.} \quad & \mathbf{x}_{k+1} = \mathbf{f}_k(\mathbf{x}_k, \mathbf{u}_k) && \text{(dynamics)} \\ & \mathbf{g}_k(\mathbf{x}_k, \mathbf{u}_k) \leq \mathbf{0} && \text{(constraints)} \\ & \mathbf{x}_k \in \mathcal{X}, \mathbf{u}_k \in \mathcal{U} && \text{(bounds)} \end{aligned}$$

A Key Concept

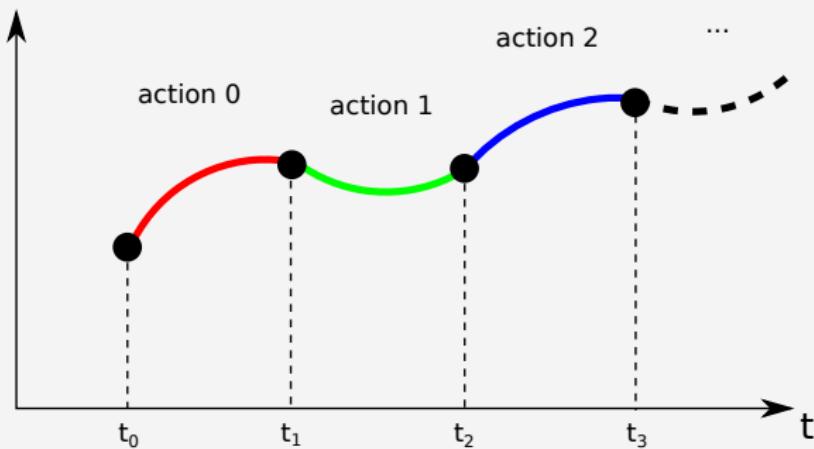
To increase efficiency, we assume a Markovian problem

$$\begin{aligned} \min_{\mathbf{x}, \mathbf{u}} \quad & l_N(\mathbf{x}_N) + \sum_{k=0}^{N-1} l_k(\mathbf{x}_k, \mathbf{u}_k) && \text{(cost)} \\ \text{s.t.} \quad & \mathbf{x}_{k+1} = \mathbf{f}_k(\mathbf{x}_k, \mathbf{u}_k) && \text{(dynamics)} \\ & \mathbf{g}_k(\mathbf{x}_k, \mathbf{u}_k) \leq \mathbf{0} && \text{(constraints)} \\ & \mathbf{x}_k \in \mathcal{X}, \mathbf{u}_k \in \mathcal{U} && \text{(bounds)} \end{aligned}$$

They are defined within the so-called action model.

A Key Concept

To increase efficiency, we assume a Markovian problem



Action model

Main functions to implement for an action model

- ▶ calc: forward simulation

```
import crocoddyl
import numpy as np

model = crocoddyl.ActionModelUnicycle()
data = model.createData()

x = model.state.rand()
u = np.random.rand(model.nu)
model.calc(data, x, u)
print data.xnext # next state
print data.cost # cost value
```

Action model

Main functions to implement for an action model

- ▶ calc: forward simulation
- ▶ calcDiff: backward propagation

```
import crocoddyl
import numpy as np

model = crocoddyl.ActionModelUnicycle()
data = model.createData()

x = model.state.rand()
u = np.random.rand(model.nu)
model.calc(data, x, u)
model.calcDiff(data, x, u)
print data.Fx, data.Fu # dynamics derivatives
print data.Lx, data.Lu, data.Lxx, data.Lxu, data.Luu # cost derivatives
```

Deriving an unicycle action model

```
import crocoddyl as croco
import numpy as np

class Unicycle(croco.ActionModelAbstract):
    def __init__(self):
        croco.ActionModelAbstract.__init__(self, croco.StateVector(3), 2, 5)
        self.dt, self.w_x, self.w_u = .1, 10., 1.

    def calc(self, data, x, u):
        px, py, theta, v, w = x, u
        c, s, dt = np.cos(theta), np.sin(theta), self.dt
        data.xnext[:] = np.array([[px + c * v * dt],
                                  [py + s * v * dt],
                                  [theta + w * dt]])
        data.r[:3], data.r[3:] = self.w_x * x, self.w_u * u
        data.cost = .5 * sum(data.r**2)

    def calcDiff(self, data, x, u):
        px, py, theta, v, w = x, u
        c, s, dt = np.cos(theta), np.sin(theta), self.dt
        nx, nu = self.state.nx, self.nu
        data.Fx[:, :] = np.array([[1, 0, -s * v * dt],
                                  [0, 1, c * v * dt],
                                  [0, 0, 1]])
        data.Fu[:, :] = np.array([[c * dt, 0], [s * dt, 0], [0, dt]])
        data.Lx[:] = x * ([self.w_x**2] * nx)
        data.Lu[:] = u * ([self.w_u**2] * nu)
        data.Lxx[range(nx), range(nx)] = [self.w_x**2] * nx
        data.Luu[range(nu), range(nu)] = [self.w_u**2] * nu
```

State

It defines the differential state manifold:

- ▶ diff: $x_1 \ominus x_2$

State

It defines the differential state manifold:

- ▶ diff: $\mathbf{x}_1 \ominus \mathbf{x}_2$
- ▶ integrate: $\mathbf{x}_0 \oplus \delta \mathbf{x}$

```
import crocoddyl

nx = 3 # state dimension
state = crocoddyl.StateVector(nx)

x0 = state.rand() # state.zero()
x1 = state.rand()

dx = state.diff(x0, x1)
x2 = state.integrate(x0, dx)
print dx
print x2 # Equals to x1
```

State

It defines the differential state manifold:

- ▶ diff: $\mathbf{x}_1 \ominus \mathbf{x}_2$
- ▶ integrate: $\mathbf{x}_0 \oplus \delta \mathbf{x}$
- ▶ Jacobians of the operators

```
import crocoddyl

nx = 3 # state dimension
state = crocoddyl.StateVector(nx)

x0 = state.rand() # state.zero()
x1 = state.rand()

dx = state.diff(x0, x1)
x2 = state.integrate(x0, dx)
print dx
print x2 # Equals to x1

ddiff_x0, ddiff_x1 = state.Jdiff(x0, x1)
dint_x0, dint_dx = state.Jintegrate(x0, dx)
print ddiff_x0, ddiff_x1
print dint_x0, dint_dx
```

Solving an optimal control problem

The problem formulation and its resolution are decoupled

```
import crocoddyl

N = 10 # horizon
model = crocoddyl.ActionModelUnicycle()

x0 = model.state.rand()
problem = crocoddyl.ShootingProblem(x0, [model] * N, model)

fddp = crocoddyl.SolverFDDP(problem) # feasibility-driven DDP (more information
                                      in https://cmastalli.github.io/
                                      publications/crocoddyl20icra.pdf)
fddp.setCallbacks([crocoddyl.CallbackVerbose()])

fddp.solve() # to warm-start the solver use fddp.solve(xs, us)
```

Developing a new solver

There are a few dedicated functions needed to implement a new solver

```
import crocoddyl

class MyNewSolver(crocoddyl.SolverAbstract):
    def __init__(self, problem):
        crocoddyl.SolverAbstract.__init__(self, problem)
        # initialize my stuffs

    def solve(self, init_xs=[], init_us=[], maxiter=100, isFeasible=False,
              regInit=None):
        self.setCandidate(init_xs, init_us, isFeasible)
        # run self.computeDirection and self.tryStep

    def computeDirection(self, recalc=True):
        # compute the search direction, recalc=True updates derivatives

    def tryStep(self, stepLength=1):
        # try the search direction computed by self.computeDirection

    def expectedImprovement(self):
        # compute the expected improvement of the iteration
```

Core API 1.0:

Unicycle towards the origin

Unicycle towards the origin

The objective are:

- ▶ Get more familiar with Crocoddyl API
- ▶ Understand how the cost weights affect the problem resolution

More instructions in the following Jupyter notebook:

https://github.com/loco-3d/crocoddyl/blob/master/examples/notebooks/unicycle_towards_origin.ipynb

Core API 2.0

Differential action model

It describes a time-continuous action model

```
import crocoddyl

nq, nu = 3, 2
model = crocoddyl.DifferentialActionModelLQR(nq, nu)
data = model.createData()

x = model.state.rand()
u = np.random.rand(model.nu)
model.calc(data, x, u)
print data.xout # next state
print data.cost # cost value

model.calcDiff(data, x, u)
print data.Fx, data.Fu # dynamics derivatives
print data.Lx, data.Lu, data.Lxx, data.Lxu, data.Luu # cost derivatives
```

Integrated action model

And we can combine it with any integration scheme (integral cost and dynamics)

```
import crocoddyl

nq, nu = 3, 2
dt = 1e-3
diffModel = crocoddyl.DifferentialActionModelLQR(nq, nu)
model = crocoddyl.IntegratedActionModelEuler(model, dt)
```

Integrated action model

And we can combine it with any integration scheme (integral cost and dynamics)

```
import crocoddyl

nq, nu = 3, 2
dt = 1e-3
diffModel = crocoddyl.DifferentialActionModelLQR(nq, nu)
model = crocoddyl.IntegratedActionModelEuler(model, dt)
```

It is possible to derive new differential and integrated action models as for action models

Core API 2.0:

Cartpole swing up

Cartpole swing up

The objective are:

- ▶ Get more familiar with Crocoddyl API
- ▶ Learn how to implement a differential action model

More instructions in the following Jupyter notebook:

https://github.com/loco-3d/crocoddyl/blob/master/examples/notebooks/cartpole.swing_up.ipynb

Contact dynamics API

Multi-contact optimal control

$$\min_{\mathbf{x}_s, \mathbf{u}_s} I_N(\mathbf{x}_N) + \sum_{k=0}^{N-1} \int_{t_k}^{t_k + \Delta t_k} I_k(\mathbf{x}_k, \mathbf{u}_k, \boldsymbol{\lambda}_k) dt$$

$$\text{s.t. } \mathbf{q}_{k+1} = \mathbf{q}_k \oplus \int_{t_k}^{t_k + \Delta t_k} \mathbf{v}_{k+1} dt, \quad (\text{integrator})$$

$$\mathbf{v}_{k+1} = \mathbf{v}_k + \int_{t_k}^{t_k + \Delta t_k} \dot{\mathbf{v}}_k dt,$$

$$\begin{bmatrix} \dot{\mathbf{v}}_k \\ -\boldsymbol{\lambda}_k \end{bmatrix} = \begin{bmatrix} \mathbf{M} & \mathbf{J}_c^\top \\ \mathbf{J}_c & \mathbf{0} \end{bmatrix}^{-1} \begin{bmatrix} \boldsymbol{\tau}_b \\ -\mathbf{a}_0 \end{bmatrix}, \quad (\text{contact dynamics})$$

$$\mathbf{R}\boldsymbol{\lambda}_{\mathcal{C}(k)} \leq \mathbf{r}, \quad (\text{friction-cone})$$

$$\log(\mathbf{p}_{\mathcal{G}(k)}(\mathbf{q}_k)^{-1} \mathbf{M}_{\mathbf{f}_{\mathcal{G}(k)}}) = \mathbf{0}, \quad (\text{contact placement})$$

$$\bar{\mathbf{x}} \leq \mathbf{x}_k \leq \underline{\mathbf{x}}, \quad (\text{state bounds})$$

$$\bar{\mathbf{u}} \leq \mathbf{u}_k \leq \underline{\mathbf{u}}, \quad (\text{control bounds})$$

Multi-contact optimal control

$$\min_{\mathbf{x}_s, \mathbf{u}_s} I_N(\mathbf{x}_N) + \sum_{k=0}^{N-1} \int_{t_k}^{t_k + \Delta t_k} I_k(\mathbf{x}_k, \mathbf{u}_k, \boldsymbol{\lambda}_k) dt$$

$$\text{s.t. } \mathbf{q}_{k+1} = \mathbf{q}_k \oplus \int_{t_k}^{t_k + \Delta t_k} \mathbf{v}_{k+1} dt, \quad (\text{integrator})$$

$$\mathbf{v}_{k+1} = \mathbf{v}_k + \int_{t_k}^{t_k + \Delta t_k} \dot{\mathbf{v}}_k dt,$$

$$\begin{bmatrix} \ddot{\mathbf{v}}_k \\ -\boldsymbol{\lambda}_k \end{bmatrix} = \begin{bmatrix} \mathbf{M} & \mathbf{J}_c^\top \\ \mathbf{J}_c & \mathbf{0} \end{bmatrix}^{-1} \begin{bmatrix} \boldsymbol{\tau}_b \\ -\mathbf{a}_0 \end{bmatrix}, \quad (\text{contact dynamics})$$

$$\mathbf{R}\boldsymbol{\lambda}_{\mathcal{C}(k)} \leq \mathbf{r}, \quad (\text{friction-cone})$$

$$\log(\mathbf{p}_{\mathcal{G}(k)}(\mathbf{q}_k)^{-1} \mathbf{M}_{\mathbf{f}_{\mathcal{G}(k)}}) = \mathbf{0}, \quad (\text{contact placement})$$

$$\bar{\mathbf{x}} \leq \mathbf{x}_k \leq \underline{\mathbf{x}}, \quad (\text{state bounds})$$

$$\bar{\mathbf{u}} \leq \mathbf{u}_k \leq \underline{\mathbf{u}}, \quad (\text{control bounds})$$

Multi-contact optimal control

$$\min_{\mathbf{x}_s, \mathbf{u}_s} I_N(\mathbf{x}_N) + \sum_{k=0}^{N-1} \int_{t_k}^{t_k + \Delta t_k} I_k(\mathbf{x}_k, \mathbf{u}_k, \boldsymbol{\lambda}_k) dt$$

$$\text{s.t. } \mathbf{q}_{k+1} = \mathbf{q}_k \oplus \int_{t_k}^{t_k + \Delta t_k} \mathbf{v}_{k+1} dt, \quad (\text{integrator})$$

$$\mathbf{v}_{k+1} = \mathbf{v}_k + \int_{t_k}^{t_k + \Delta t_k} \dot{\mathbf{v}}_k dt,$$

$$\begin{bmatrix} \dot{\mathbf{v}}_k \\ -\boldsymbol{\lambda}_k \end{bmatrix} = \begin{bmatrix} \mathbf{M} & \mathbf{J}_c^\top \\ \mathbf{J}_c & \mathbf{0} \end{bmatrix}^{-1} \begin{bmatrix} \boldsymbol{\tau}_b \\ -\mathbf{a}_0 \end{bmatrix}, \quad (\text{contact dynamics})$$

$$\mathbf{R}\boldsymbol{\lambda}_{\mathcal{C}(k)} \leq \mathbf{r}, \quad (\text{friction-cone})$$

$$\log(\mathbf{p}_{\mathcal{G}(k)}(\mathbf{q}_k)^{-1} \mathbf{o} \mathbf{M}_{\mathbf{f}_{\mathcal{G}(k)}}) = \mathbf{0}, \quad (\text{contact placement})$$

$$\bar{\mathbf{x}} \leq \mathbf{x}_k \leq \underline{\mathbf{x}}, \quad (\text{state bounds})$$

$$\bar{\mathbf{u}} \leq \mathbf{u}_k \leq \underline{\mathbf{u}}, \quad (\text{control bounds})$$

Multi-contact optimal control

$$\min_{\mathbf{x}_s, \mathbf{u}_s} I_N(\mathbf{x}_N) + \sum_{k=0}^{N-1} \int_{t_k}^{t_k + \Delta t_k} I_k(\mathbf{x}_k, \mathbf{u}_k, \boldsymbol{\lambda}_k) dt$$

$$\text{s.t. } \mathbf{q}_{k+1} = \mathbf{q}_k \oplus \int_{t_k}^{t_k + \Delta t_k} \mathbf{v}_{k+1} dt, \quad (\text{integrator})$$

$$\mathbf{v}_{k+1} = \mathbf{v}_k + \int_{t_k}^{t_k + \Delta t_k} \dot{\mathbf{v}}_k dt,$$

$$\begin{bmatrix} \dot{\mathbf{v}}_k \\ -\boldsymbol{\lambda}_k \end{bmatrix} = \begin{bmatrix} \mathbf{M} & \mathbf{J}_c^\top \\ \mathbf{J}_c & \mathbf{0} \end{bmatrix}^{-1} \begin{bmatrix} \boldsymbol{\tau}_b \\ -\mathbf{a}_0 \end{bmatrix}, \quad (\text{contact dynamics})$$

$$\mathbf{R}\boldsymbol{\lambda}_{\mathcal{C}(k)} \leq \mathbf{r}, \quad (\text{friction-cone})$$

$$\log(\mathbf{p}_{\mathcal{G}(k)}(\mathbf{q}_k)^{-1} \mathbf{o} \mathbf{M}_{\mathbf{f}_{\mathcal{G}(k)}}) = \mathbf{0}, \quad (\text{contact placement})$$

$$\bar{\mathbf{x}} \leq \mathbf{x}_k \leq \underline{\mathbf{x}}, \quad (\text{state bounds})$$

$$\bar{\mathbf{u}} \leq \mathbf{u}_k \leq \underline{\mathbf{u}}, \quad (\text{control bounds})$$

Contact dynamics

$$\begin{bmatrix} \dot{\mathbf{v}}_k \\ -\boldsymbol{\lambda}_k \end{bmatrix} = \begin{bmatrix} \mathbf{M} & \mathbf{J}_c^\top \\ \mathbf{J}_c & \mathbf{0} \end{bmatrix}^{-1} \begin{bmatrix} \boldsymbol{\tau}_b \\ -\mathbf{a}_0 \end{bmatrix}$$

```
import crocoddyl as croco
import pinocchio as pin
import example_robot_data as robots

rmodel = robots.loadICub().model

state = croco.StateMultibody(rmodel)
actuation = croco.ActuationModelFloatingBase(state)
contacts = croco.ContactModelMultiple(state, actuation.nu)
costs = croco.CostModelSum(state, actuation.nu)

# ... define contacts and costs

model = croco.DifferentialActionModelContactFwdDynamics(state, actuation,
                                                          contacts, costs, 0., True)
```

Contact dynamics

$$\begin{bmatrix} \dot{\mathbf{v}}_k \\ -\lambda_k \end{bmatrix} = \begin{bmatrix} \mathbf{M} & \mathbf{J}_c^\top \\ \mathbf{J}_c & \mathbf{0} \end{bmatrix}^{-1} \begin{bmatrix} \boldsymbol{\tau}_b \\ -\mathbf{a}_0 \end{bmatrix}$$

```
# Defining the contact frames
Mref = croco.FramePlacement(rmodel.getFrameId("r_sole"), pin.SE3.Random())
xref = croco.FrameTranslation(rmodel.getFrameId("l_sole"), pin.SE3.Random().
                               translation)
contact_6d = croco.ContactModel6D(state, Mref, actuation.nu)
contact_3d = croco.ContactModel3D(state, xref, actuation.nu)
```

Cost functions

A cost is described by a residual vector $\mathbf{r}(\cdot)$ and an activation function $\mathbf{a}(\cdot)$:

$$\mathbf{l}(\mathbf{x}, \mathbf{u}) = \mathbf{a}(\mathbf{r}(\mathbf{x}, \mathbf{u}))$$

Cost functions

A cost is described by a residual vector $\mathbf{r}(\cdot)$ and an activation function $\mathbf{a}(\cdot)$:

$$\mathbf{l}(\mathbf{x}, \mathbf{u}) = \mathbf{a}(\mathbf{r}(\mathbf{x}, \mathbf{u}))$$

There are a few activation functions available:

- ▶ (Weighted) Quadratic
- ▶ (Weighted) Quadratic barriers
- ▶ Smooth abs

Cost functions

A cost is described by a residual vector $\mathbf{r}(\cdot)$ and an activation function $\mathbf{a}(\cdot)$:

$$\mathbf{l}(\mathbf{x}, \mathbf{u}) = \mathbf{a}(\mathbf{r}(\mathbf{x}, \mathbf{u}))$$

There are a few activation functions available:

- ▶ (Weighted) Quadratic
- ▶ (Weighted) Quadratic barriers
- ▶ Smooth abs

There is a range of different cost functions:

- ▶ State and control
- ▶ Frame placement, translation, rotation, velocity
- ▶ CoM
- ▶ Centroidal momentum and forces

Cost functions

A cost is described by a residual vector $\mathbf{r}(\cdot)$ and an activation function $\mathbf{a}(\cdot)$:

$$\mathbf{l}(\mathbf{x}, \mathbf{u}) = \mathbf{a}(\mathbf{r}(\mathbf{x}, \mathbf{u}))$$

```
# Define CoM cost function
cref = np.array([0., 0., 1.])
comTrack = croco.CostModelCoMPosition(state, cref, actuation.nu)
costModel.addCost("comTrack", comTrack, 1e3)
```

Friction cone and contact placement penalization

We could define soft-constraints using, for instances, quadratic barriers:

Friction cone

$$R\lambda_{C(k)} \leq r$$

```
# Defining friction cone soft-constraint
nsurf, mu = np.array([0., 0., 1.]), 0.7
frictionCone = croco.FrictionCone(nsurf, mu, 4, False)
bounds = croco.ActivationBounds(frictionCone.lb, frictionCone.ub) # magic here
activation = croco.ActivationModelQuadraticBarrier(bounds)
frFriction = croco.FrameFrictionCone(rmodel.getFrameId("r_sole"), frictionCone)
frictionCost = croco.CostModelContactFrictionCone(state, activation, frFriction,
                                                    actuation.nu)
costs.addCost("r_sole_frictionCone", frictionCost, 1e3)
```

Friction cone and contact placement penalization

We could define soft-constraints using, for instances, quadratic barriers:

Contact placement

$$\log(\mathbf{p}_{\mathcal{G}(k)}(\mathbf{q}_k)^{-1} \mathbf{o} \mathbf{M}_{\mathbf{f}_{\mathcal{G}(k)}}) = \mathbf{0}$$

```
# Defining a contact placement soft-constraint
xref = croco.FrameTranslation(rmodel.getFrameId("l_sole"), Mref.translation)
placementCost = croco.CostModelFrameTranslation(state, xref, actuation.nu)
costModel.addCost("l_sole_footPlacement", placementCost, 1e6)
```

Contact dynamics API:
Whole-body manipulation

Whole-body manipulation

The objective are:

- ▶ Get more familiar with Contact dynamics API
- ▶ Understand how to build a whole-body manipulation problem

More instructions in the following Jupyter notebook:

https://github.com/loco-3d/crocoddyl/blob/master/examples/notebooks/whole_body_manipulation.ipynb

Contact dynamics API:

Bipedal walking

Bipedal walking

The objective is:

- ▶ Understand how the multi-contact locomotion is affected by changes in the step timings

More instructions in the following Jupyter notebook:

https://github.com/loco-3d/crocoddyl/blob/master/examples/notebooks/bipedal_walking.ipynb