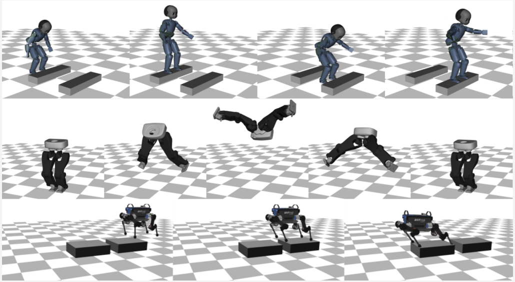
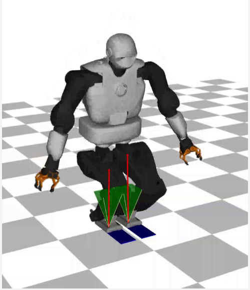
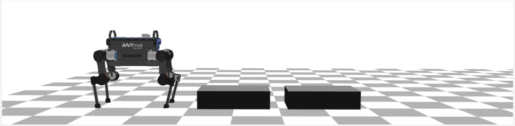
# Crocoddyl: An Efficient Multi-Contact Optimal Control Framework

Implementation and tutorial







**Carlos Mastalli**
University of Edinburgh

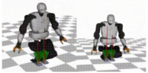
## Overview

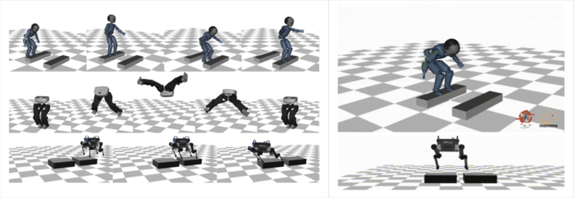
1. Introduction
2. Core API 1.0
   Exercise: unicycle towards the origin
3. Core API 2.0
   Exercise: cartpole swing up
4. Contact dynamics API
   Exercise: whole-body manipulation
5. More insight of optimal control
   Exercise: bipedal walking (optional)

# Introduction

# Introduction

## Contact RObot COntrol 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.

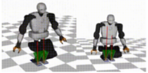
The source code is released under the BSD 3-Clause license.

**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

# Introduction

## 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.

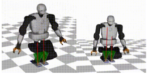
**Authors:** Carlos Mastalli and Rohan Budhiraja
**Instructors:** Nicolas Mansard
**With additional support from the Gepetto team at LAAS-CNRS and MEMMO project. For more details see Section Credits**

[1] `https://github.com/loco-3d/crocoddyl`

The Alan Turing Institute Gepetto memmo

# Introduction

## 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.

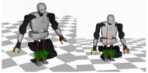
The source code is released under the BSD 3-Clause license.

**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

# Introduction

## 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.

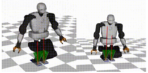
The source code is released under the BSD 3-Clause license.

**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

# Introduction

## 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.

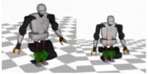
**Authors:** Carlos Mastalli and Rohan Budhiraja
**Instructors:** Nicolas Mansard
**With additional support from the Gepetto team at LAAS-CNRS and MEMMO project. For more details see Section Credits**

[1] https://github.com/loco-3d/crocoddyl

The Alan Turing Institute Gepetto memmo

# Introduction

## 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

# 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

# Main features

**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

# Main features

**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:

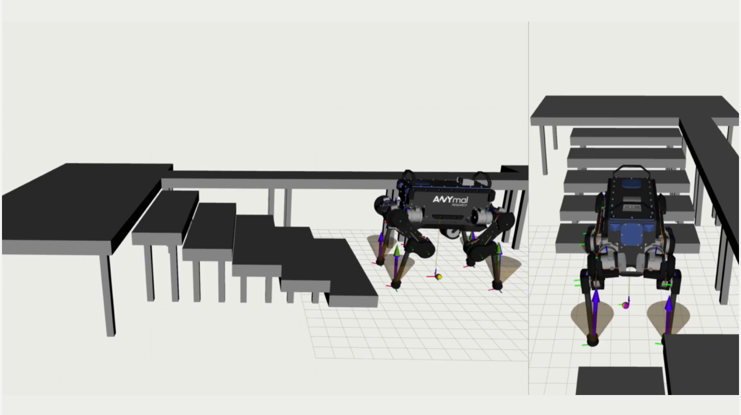
- cache friendly
- multi-thread friendly
- Python bindings (including models and solvers abstractions)
- C++ 98/11/14/17/20 compliant
- extensively tested
- automatic code generation support

[1] `https://github.com/loco-3d/crocoddyl`

# Scope and Motivation

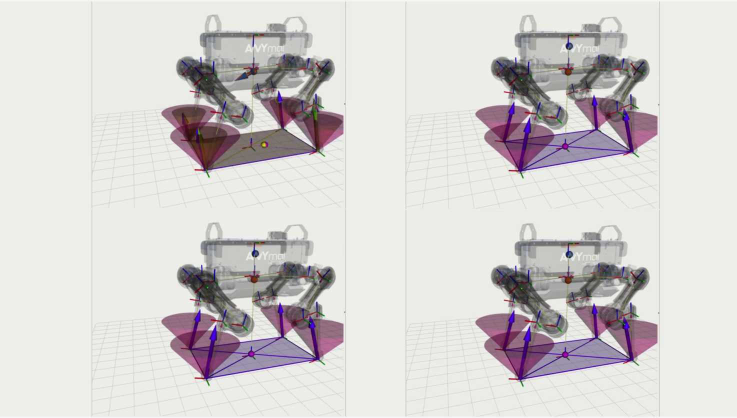
*Fast whole-body model predictive control for legged robots*

... *to generate motion within the actuation limits*



## Scope and Motivation

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



The Alan Turing Institute   Gepetto   The University of Edinburgh   memmo

## 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

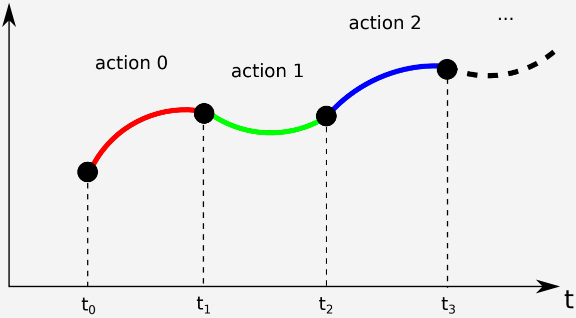
*To increase efficiency, we assume a Markovian problem*

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

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

```python
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: $\mathbf{x}_1 \ominus \mathbf{x}_2$

# State

It defines the differential state manifold:

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

```python
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

```python
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)
```

# Core API 1.0:

# Unicycle towards the origin

## Unicycle towards the origin

The objective are:

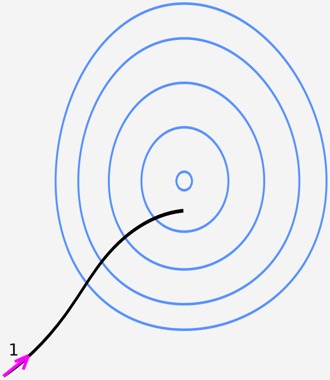
- Get more familiar with Crocoddyl API
- Understand how the cost weights affect the problem resolution

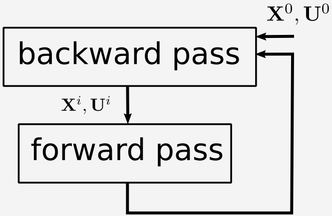
More instructions in the following Jupyter notebook:

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

# Core API 2.0

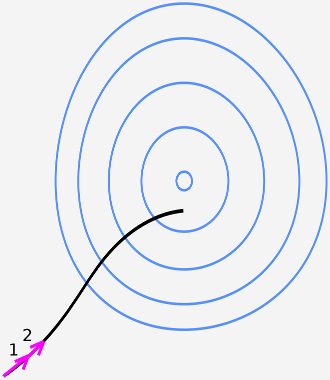
## Developing a new solver

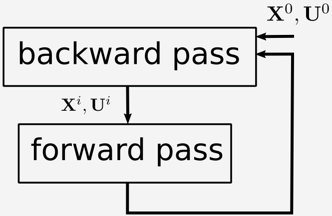



**Blue curves represents the level-set of the cost function**

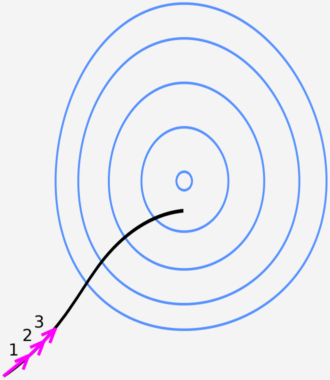
# Developing a new solver

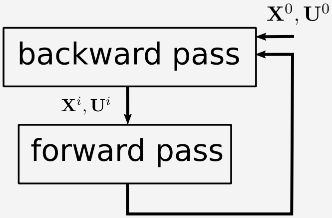



**Black curve represents the system dynamics (equality constraint)**

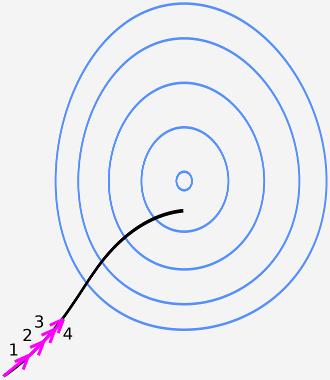
# Developing a new solver

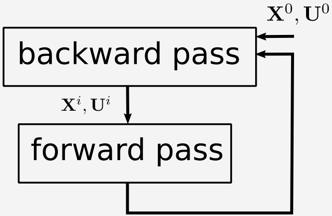



**Search direction is computed from the problem derivatives (arrow)**

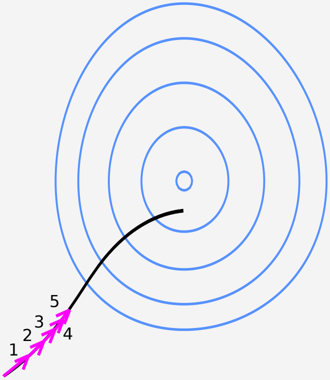
## Developing a new solver

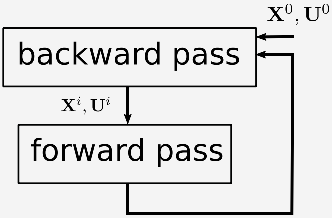



**An expected-improvement procedure evaluates the direction given a defined step length**

# Developing a new solver



# Developing a new solver

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

```python
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
```

# 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

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

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

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

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

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

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

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

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

## Multi-contact optimal control

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

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

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

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

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

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

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

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

# Multi-contact optimal control

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

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

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

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

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

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

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

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

# Multi-contact optimal control

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

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

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

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

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

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

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

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

# Contact dynamics

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

```python
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

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

```python
# 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 $\mathbf{a}(\cdot)$:

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

```python
# 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

$$\mathbf{R}\boldsymbol{\lambda}_{\mathcal{C}(k)} \leq \mathbf{r}$$

```python
# 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

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

```python
# 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`

## Optimal Control Families

- Indirect Methods (Pontryagin's Minimum Principle (PMP))
  - Hamiltonian: $H(\mathbf{x}, \boldsymbol{\lambda}, \mathbf{u}) = l(\mathbf{x}, \mathbf{u}) + \boldsymbol{\lambda}^\top \mathbf{f}(\mathbf{x}, \mathbf{u})$

## Optimal Control Families

- Indirect Methods (Pontryagin's Minimum Principle (PMP))
  - Hamiltonian: $H(\mathbf{x}, \boldsymbol{\lambda}, \mathbf{u}) = l(\mathbf{x}, \mathbf{u}) + \boldsymbol{\lambda}^\top \mathbf{f}(\mathbf{x}, \mathbf{u})$
  - Get optimal control input:

    $\mathbf{u}(\mathbf{x}, \boldsymbol{\lambda}) = \arg\min_{\mathbf{u}} H(\mathbf{x}, \boldsymbol{\lambda}, \mathbf{u}) \quad \text{s.t.} \quad \mathbf{g}(\mathbf{x}, \mathbf{u}) \leq \mathbf{0}$

## Optimal Control Families

- Indirect Methods (Pontryagin's Minimum Principle (PMP))
  - Hamiltonian: $H(\mathbf{x}, \boldsymbol{\lambda}, \mathbf{u}) = l(\mathbf{x}, \mathbf{u}) + \boldsymbol{\lambda}^\top \mathbf{f}(\mathbf{x}, \mathbf{u})$
  - Get optimal control input:

    $$\mathbf{u}(\mathbf{x}, \boldsymbol{\lambda}) = \arg\min_{\mathbf{u}} H(\mathbf{x}, \boldsymbol{\lambda}, \mathbf{u}) \quad \text{s.t.} \quad \mathbf{g}(\mathbf{x}, \mathbf{u}) \leq \mathbf{0}$$
  - State-costate integration:

    $$\begin{array}{lll} \text{State:} & \dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, \mathbf{u}), & \mathbf{x}(t_0) = \mathbf{x}_0 \\ \text{Costate:} & \dot{\boldsymbol{\lambda}} = -\nabla_{\mathbf{x}} H(\mathbf{x}, \boldsymbol{\lambda}, \mathbf{u}), & \boldsymbol{\lambda}(t_N) = l_N(\mathbf{x}_N) \end{array}$$

## Optimal Control Families

- Indirect Methods (Pontryagin's Minimum Principle (PMP))
  - Hamiltonian: $H(\mathbf{x}, \boldsymbol{\lambda}, \mathbf{u}) = l(\mathbf{x}, \mathbf{u}) + \boldsymbol{\lambda}^\top \mathbf{f}(\mathbf{x}, \mathbf{u})$
  - Get optimal control input:

    $$\mathbf{u}(\mathbf{x}, \boldsymbol{\lambda}) = \arg\min_{\mathbf{u}} H(\mathbf{x}, \boldsymbol{\lambda}, \mathbf{u}) \quad \text{s.t.} \quad \mathbf{g}(\mathbf{x}, \mathbf{u}) \leq \mathbf{0}$$
  - State-costate integration:

    State: $\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, \mathbf{u}), \qquad \mathbf{x}(t_0) = \mathbf{x}_0$

    Costate: $\dot{\boldsymbol{\lambda}} = -\nabla_{\mathbf{x}} H(\mathbf{x}, \boldsymbol{\lambda}, \mathbf{u}), \qquad \boldsymbol{\lambda}(t_N) = l_N(\mathbf{x}_N)$

## Optimal Control Families

- Indirect Methods (Pontryagin's Minimum Principle (PMP))
  - Hamiltonian: $H(\mathbf{x}, \boldsymbol{\lambda}, \mathbf{u}) = l(\mathbf{x}, \mathbf{u}) + \boldsymbol{\lambda}^\top \mathbf{f}(\mathbf{x}, \mathbf{u})$
  - Get optimal control input:

    $$\mathbf{u}(\mathbf{x}, \boldsymbol{\lambda}) = \arg\min_{\mathbf{u}} H(\mathbf{x}, \boldsymbol{\lambda}, \mathbf{u}) \quad \text{s.t.} \quad \mathbf{g}(\mathbf{x}, \mathbf{u}) \leq \mathbf{0}$$

  - State-costate integration:

    $$\begin{aligned} &\text{State:} && \dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, \mathbf{u}), && \mathbf{x}(t_0) = \mathbf{x}_0 \\ &\text{Costate:} && \dot{\boldsymbol{\lambda}} = -\nabla_{\mathbf{x}} H(\mathbf{x}, \boldsymbol{\lambda}, \mathbf{u}), && \boldsymbol{\lambda}(t_N) = l_N(\mathbf{x}_N) \end{aligned}$$

- Direct Methods (Transcription to NLP)

  Solve the resulting NL program

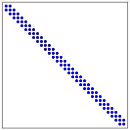
  $$\begin{aligned} \min_{\mathbf{x}_s, \mathbf{u}_s} \ & \phi(\mathbf{x}_s, \mathbf{u}_s) \\ \text{s.t.} \ & \mathbf{g}(\mathbf{x}_s, \mathbf{u}_s) = \mathbf{0}, \\ & \mathbf{h}(\mathbf{x}_s, \mathbf{u}_s) \leq \mathbf{0}, \end{aligned}$$

# Classical DDP vs Direct Method (SQP)

## Faster iteration, feedback policy

NLP

$$\begin{bmatrix} \delta\mathbf{x}_0 \\ \delta\mathbf{u}_0 \\ \boldsymbol{\lambda}_0^+ \\ \vdots \\ \delta\mathbf{x}_N \\ \delta\mathbf{u}_N \\ \boldsymbol{\lambda}_N^+ \\ \delta\mathbf{x}_{N+1} \\ \boldsymbol{\lambda}_{N+1}^+ \end{bmatrix} =$$



KKT matrix

$$^{-1} \begin{bmatrix} \boldsymbol{\phi}_0 \\ \mathbf{g}_0 \\ \vdots \\ \boldsymbol{\phi}_N \\ \mathbf{g}_N \\ \boldsymbol{\phi}_{N+1} \end{bmatrix}$$

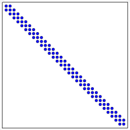
$$\begin{bmatrix} \mathbf{X}_{i+1} \\ \mathbf{U}_{i+1} \end{bmatrix} = \begin{bmatrix} \mathbf{X}_i \\ \mathbf{U}_i \end{bmatrix} + \alpha \begin{bmatrix} \delta\mathbf{X}_i \\ \delta\mathbf{U}_i \end{bmatrix}$$

# Classical DDP vs Direct Method (SQP)

## Faster iteration, feedback policy

NLP

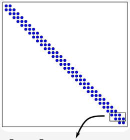
$$\begin{bmatrix} \delta\mathbf{x}_0 \\ \delta\mathbf{u}_0 \\ \boldsymbol{\lambda}_0^+ \\ \vdots \\ \delta\mathbf{x}_N \\ \delta\mathbf{u}_N \\ \boldsymbol{\lambda}_N^+ \\ \delta\mathbf{x}_{N+1} \\ \boldsymbol{\lambda}_{N+1}^+ \end{bmatrix} = [\,\cdot\,]^{-1} \begin{bmatrix} \boldsymbol{\phi}_0 \\ \mathbf{g}_0 \\ \vdots \\ \boldsymbol{\phi}_N \\ \mathbf{g}_N \\ \boldsymbol{\phi}_{N+1} \end{bmatrix}$$



KKT matrix

$$\begin{bmatrix} \mathbf{X}_{i+1} \\ \mathbf{U}_{i+1} \end{bmatrix} = \begin{bmatrix} \mathbf{X}_{i} \\ \mathbf{U}_{i} \end{bmatrix} + \alpha \begin{bmatrix} \delta\mathbf{X}_{i} \\ \delta\mathbf{U}_{i} \end{bmatrix}$$

DDP



$$\begin{bmatrix} \delta\mathbf{u}_N \\ \boldsymbol{\lambda}_N^+ \end{bmatrix} = [\,\cdot\,]^{-1} \begin{bmatrix} \boldsymbol{\phi}_N \\ \mathbf{g}_N \end{bmatrix}$$

**Matrix factorization is $O(n^3)$**

# Classical DDP vs Direct Method (SQP)

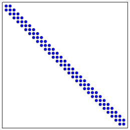
## Faster iteration, feedback policy

$$\begin{bmatrix} \mathbf{X}_{i+1} \\ \mathbf{U}_{i+1} \end{bmatrix} = \begin{bmatrix} \mathbf{X}_{i} \\ \mathbf{U}_{i} \end{bmatrix} + \alpha \begin{bmatrix} \delta\mathbf{X}_{i} \\ \delta\mathbf{U}_{i} \end{bmatrix}$$

**Matrix factorization is $O(n^3)$**

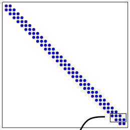
NLP

$$\begin{bmatrix} \delta\mathbf{x}_0 \\ \delta\mathbf{u}_0 \\ \boldsymbol{\lambda}_0^+ \\ \vdots \\ \delta\mathbf{x}_N \\ \delta\mathbf{u}_N \\ \boldsymbol{\lambda}_N^+ \\ \delta\mathbf{x}_{N+1} \\ \boldsymbol{\lambda}_{N+1}^+ \end{bmatrix} = [\text{KKT matrix}]^{-1} \begin{bmatrix} \phi_0 \\ \mathbf{g}_0 \\ \vdots \\ \phi_N \\ \mathbf{g}_N \\ \phi_{N+1} \end{bmatrix}$$



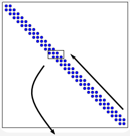
KKT matrix

DDP



$$\begin{bmatrix} \delta\mathbf{u}_N \\ \boldsymbol{\lambda}_N^+ \end{bmatrix} = \boxed{\cdots}^{-1} \begin{bmatrix} \phi_N \\ \mathbf{g}_N \end{bmatrix}$$

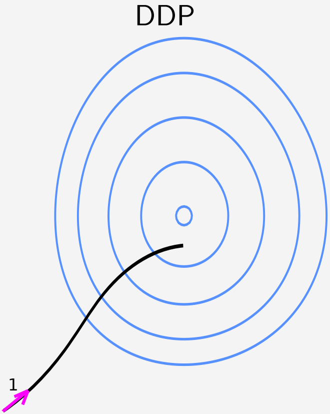
...

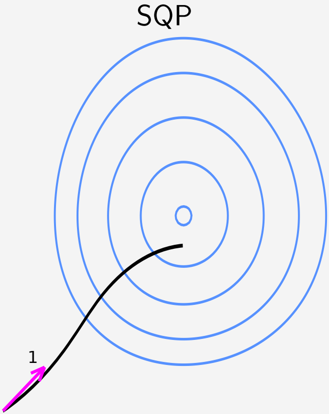


$$\begin{bmatrix} \delta\mathbf{u}_i \\ \boldsymbol{\lambda}_i^+ \end{bmatrix} = \boxed{\cdots}^{-1} \begin{bmatrix} \phi_i \\ \mathbf{g}_i \end{bmatrix}$$

# Classical DDP vs Direct Method (SQP)

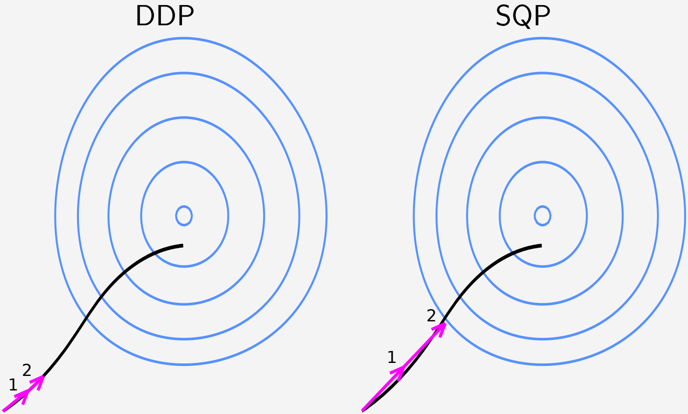
Slower convergence, poor globalization



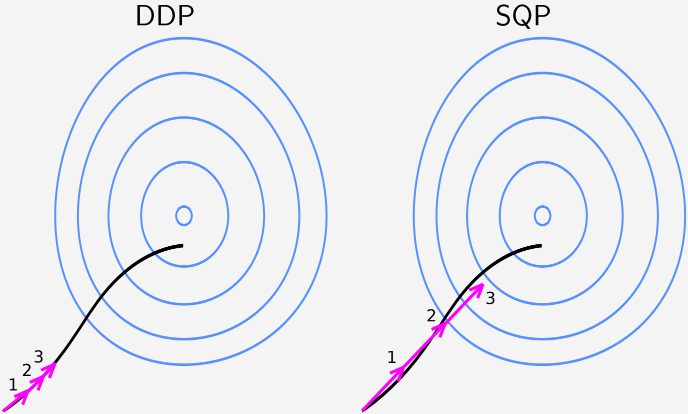
# Classical DDP vs Direct Method (SQP)

## Slower convergence, poor globalization



# Classical DDP vs Direct Method (SQP)

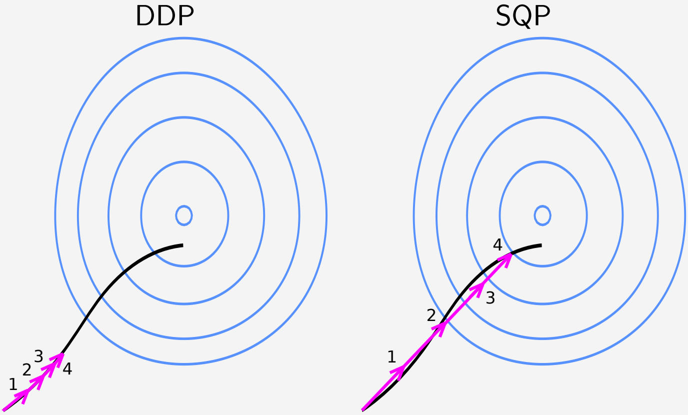
## Slower convergence, poor globalization




**A merit function (SQP) accepts some constraint violations**

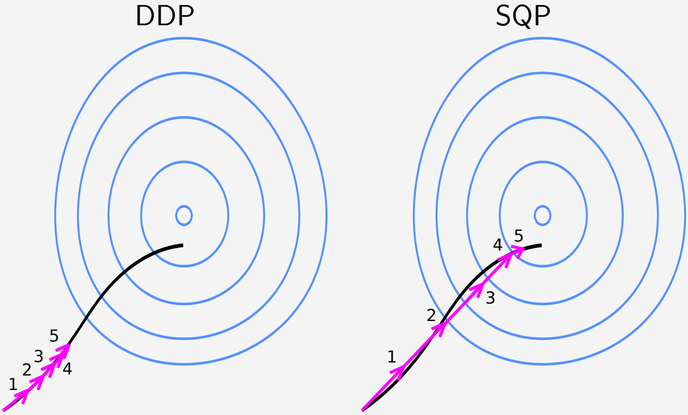
# Classical DDP vs Direct Method (SQP)

Slower convergence, poor globalization



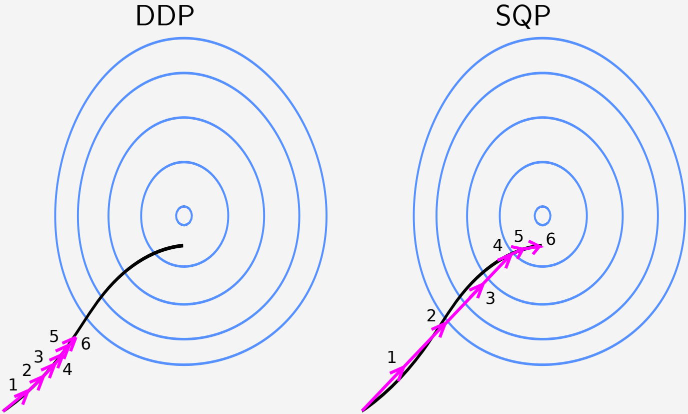
# Classical DDP vs Direct Method (SQP)

Slower convergence, poor globalization



# Classical DDP vs Direct Method (SQP)

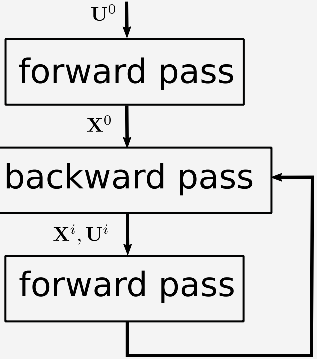
## Slower convergence, poor globalization




**Nonlinear rollout (DDP) does small steps due to constraint satisfaction**

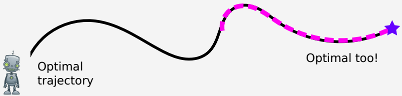
# Classical DDP vs Direct Method (SQP)

## Single shooting, control warm-start



## Understanding the classical DDP

- Sequence of simpler Hamiltonian (Bellman)



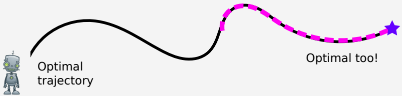

$$\mathbf{u}_N(\mathbf{x}_N, \boldsymbol{\lambda}_N) = \arg\min_{\mathbf{u}_N} H_N(\mathbf{x}_N, \boldsymbol{\lambda}_N, \mathbf{u}_N) \quad \text{s.t.} \quad \mathbf{g}_N(\mathbf{x}_N, \mathbf{u}_N) \leq \mathbf{0}$$

$$\vdots$$

$$\mathbf{u}_0(\mathbf{x}_0, \boldsymbol{\lambda}_0) = \arg\min_{\mathbf{u}_0} H_0(\mathbf{x}_0, \boldsymbol{\lambda}_0, \mathbf{u}_0) \quad \text{s.t.} \quad \mathbf{g}_0(\mathbf{x}_0, \mathbf{u}_0) \leq \mathbf{0}$$

## Understanding the classical DDP

- Sequence of simpler Hamiltonian (Bellman)




$$\begin{aligned}\mathbf{u}_N(\mathbf{x}_N, \boldsymbol{\lambda}_N) &= \arg\min_{\mathbf{u}_N} H_N(\mathbf{x}_N, \boldsymbol{\lambda}_N, \mathbf{u}_N) \quad \text{s.t.} \quad \mathbf{g}_N(\mathbf{x}_N, \mathbf{u}_N) \leq \mathbf{0} \\ &\vdots \\ \mathbf{u}_0(\mathbf{x}_0, \boldsymbol{\lambda}_0) &= \arg\min_{\mathbf{u}_0} H_0(\mathbf{x}_0, \boldsymbol{\lambda}_0, \mathbf{u}_0) \quad \text{s.t.} \quad \mathbf{g}_0(\mathbf{x}_0, \mathbf{u}_0) \leq \mathbf{0}\end{aligned}$$

## Understanding the classical DDP

- Sequence of simpler Hamiltonian (Bellman)
- LQ apprx. of the Hamiltonian

$$H_k(\cdot) = \frac{1}{2}\begin{bmatrix} 1 \\ \delta\mathbf{x}_{k+1} \end{bmatrix}^\top \begin{bmatrix} 0 & V_{\mathbf{x}_{k+1}}^\top \\ V_{\mathbf{x}_{k+1}} & V_{\mathbf{xx}_{k+1}} \end{bmatrix} \begin{bmatrix} 1 \\ \delta\mathbf{x}_{k+1} \end{bmatrix} + \frac{1}{2}\begin{bmatrix} 1 \\ \delta\mathbf{x}_k \\ \delta\mathbf{u}_k \end{bmatrix}^\top \begin{bmatrix} 0 & \mathbf{l}_{\mathbf{x}_k}^\top & \mathbf{l}_{\mathbf{u}_k}^\top \\ \mathbf{l}_{\mathbf{x}_k} & \mathbf{l}_{\mathbf{xx}_k} & \mathbf{l}_{\mathbf{xu}_k} \\ \mathbf{l}_{\mathbf{u}_k} & \mathbf{l}_{\mathbf{xu}_k}^\top & \mathbf{l}_{\mathbf{uu}_k} \end{bmatrix} \begin{bmatrix} 1 \\ \delta\mathbf{x}_k \\ \delta\mathbf{u}_k \end{bmatrix},$$

where $V_{\mathbf{x}_k}, V_{\mathbf{xx}_k}$ describe the costate $\boldsymbol{\lambda}_k$.

The Alan Turing Institute

## Understanding the classical DDP

- Sequence of simpler Hamiltonian (Bellman)
- LQ apprx. of the Hamiltonian

$$H_k(\cdot) = \frac{1}{2}\begin{bmatrix} 1 \\ \delta\mathbf{x}_{k+1} \end{bmatrix}^\top \begin{bmatrix} 0 & V_{\mathbf{x}_{k+1}}^\top \\ V_{\mathbf{x}_{k+1}} & V_{\mathbf{xx}_{k+1}} \end{bmatrix} \begin{bmatrix} 1 \\ \delta\mathbf{x}_{k+1} \end{bmatrix} + \frac{1}{2}\begin{bmatrix} 1 \\ \delta\mathbf{x}_k \\ \delta\mathbf{u}_k \end{bmatrix}^\top \begin{bmatrix} 0 & \mathbf{l}_{\mathbf{x}_k}^\top & \mathbf{l}_{\mathbf{u}_k}^\top \\ \mathbf{l}_{\mathbf{x}_k} & \mathbf{l}_{\mathbf{xx}_k} & \mathbf{l}_{\mathbf{xu}_k} \\ \mathbf{l}_{\mathbf{u}_k} & \mathbf{l}_{\mathbf{xu}_k}^\top & \mathbf{l}_{\mathbf{uu}_k} \end{bmatrix} \begin{bmatrix} 1 \\ \delta\mathbf{x}_k \\ \delta\mathbf{u}_k \end{bmatrix},$$

where $V_{\mathbf{x}_k}, V_{\mathbf{xx}_k}$ describe the costate $\boldsymbol{\lambda}_k$.

## Understanding the classical DDP

- Sequence of simpler Hamiltonian (Bellman)
- LQ apprx. of the Hamiltonian
- PMP to each simpler problem

$$\delta\mathbf{u}_k^*(\delta\mathbf{x}_k) = \overbrace{\arg\min_{\delta\mathbf{u}_k} \frac{1}{2}\begin{bmatrix} 1 \\ \delta\mathbf{x}_k \\ \delta\mathbf{u}_k \end{bmatrix}^\top \begin{bmatrix} 0 & \mathbf{Q}_{\mathbf{x}_k}^\top & \mathbf{Q}_{\mathbf{u}_k}^\top \\ \mathbf{Q}_{\mathbf{x}_k} & \mathbf{Q}_{\mathbf{xx}_k} & \mathbf{Q}_{\mathbf{xu}_k} \\ \mathbf{Q}_{\mathbf{u}_k} & \mathbf{Q}_{\mathbf{xu}_k}^\top & \mathbf{Q}_{\mathbf{uu}_k} \end{bmatrix} \begin{bmatrix} 1 \\ \delta\mathbf{x}_k \\ \delta\mathbf{u}_k \end{bmatrix}}^{Hamiltonian = H(\delta\mathbf{x}_k, V_{\mathbf{x}_k}, V_{\mathbf{xx}_k}, \delta\mathbf{u}_k, k)},$$

$$\text{s.t.} \qquad \mathbf{g}(\mathbf{x}_k \oplus \delta\mathbf{x}_k, \mathbf{u}_k + \delta\mathbf{u}_k) \leq \mathbf{0}, \qquad \text{(path constraints)}$$

## Understanding the classical DDP

- Sequence of simpler Hamiltonian (Bellman)
- LQ apprx. of the Hamiltonian
- PMP to each simpler problem
- State-costate integration

State integration (forward pass):

$$\begin{aligned}
\mathbf{x}_0^{i+1} &= \bar{\mathbf{x}}_0 && \text{(initial condition)} \\
\mathbf{u}_k^{i+1} &= \mathbf{u}_k^i + \delta\mathbf{u}_k^i && \text{(PMP solution)} \\
\mathbf{x}_{k+1}^{i+1} &= \mathbf{f}(\mathbf{x}_k^{i+1}, \mathbf{u}_k^{i+1}) && \text{(rollout)}
\end{aligned}$$

## Understanding the classical DDP

- Sequence of simpler Hamiltonian (Bellman)
- LQ apprx. of the Hamiltonian
- PMP to each simpler problem
- State-costate integration

State integration (forward pass):

$$
\begin{aligned}
\mathbf{x}_0^{i+1} &= \bar{\mathbf{x}}_0 && \text{(initial condition)} \\
\mathbf{u}_k^{i+1} &= \mathbf{u}_k^i + \delta\mathbf{u}_k^i && \text{(PMP solution)} \\
\mathbf{x}_{k+1}^{i+1} &= \mathbf{f}(\mathbf{x}_k^{i+1}, \mathbf{u}_k^{i+1}) && \text{(rollout)}
\end{aligned}
$$

## Understanding the classical DDP

- Sequence of simpler Hamiltonian (Bellman)
- LQ apprx. of the Hamiltonian
- PMP to each simpler problem
- State-costate integration

State integration (forward pass):

$$
\begin{aligned}
\mathbf{x}_0^{i+1} &= \bar{\mathbf{x}}_0 && \text{(initial condition)} \\
\mathbf{u}_k^{i+1} &= \mathbf{u}_k^{i} + \delta\mathbf{u}_k^{i} && \text{(PMP solution)} \\
\mathbf{x}_{k+1}^{i+1} &= \mathbf{f}(\mathbf{x}_k^{i+1}, \mathbf{u}_k^{i+1}) && \text{(rollout)}
\end{aligned}
$$

## Understanding the classical DDP

- Sequence of simpler Hamiltonian (Bellman)
- LQ apprx. of the Hamiltonian
- PMP to each simpler problem
- State-costate integration

  Costate integration (backward pass):

$$
\begin{aligned}
V_{\mathbf{x}_N},\ V_{\mathbf{xx}_N} &= l_{\mathbf{x}_N},\ l_{\mathbf{xx}_N} && \text{(terminal condition)}\\
V_{\mathbf{x}_i} &= \mathbf{Q}_{\mathbf{x}_i} - \mathbf{Q}_{\mathbf{xu}_i}\mathbf{Q}_{\mathbf{uu}_i}^{-1}\mathbf{Q}_{\mathbf{u}_i} && \text{(costate Jacobian)}\\
V_{\mathbf{xx}_i} &= \mathbf{Q}_{\mathbf{xx}_i} - \mathbf{Q}_{\mathbf{xu}_i}\mathbf{Q}_{\mathbf{uu}_i}^{-1}\mathbf{Q}_{\mathbf{ux}_i} && \text{(costate Hessian)}\\
dV &= -\frac{1}{2}\mathbf{Q}_{\mathbf{u}_i}^{\top}\mathbf{Q}_{\mathbf{uu}_i}^{-1}\mathbf{Q}_{\mathbf{u}_i} && \text{(costate rate)}
\end{aligned}
$$

# Understanding the classical DDP

- Sequence of simpler Hamiltonian (Bellman)
- LQ apprx. of the Hamiltonian
- PMP to each simpler problem
- State-costate integration

  Costate integration (backward pass):

$$
\begin{aligned}
V_{\mathbf{x}_N},\ V_{\mathbf{xx}_N} &= l_{\mathbf{x}_N},\ l_{\mathbf{xx}_N} && \text{(terminal condition)}\\
V_{\mathbf{x}_i} &= \mathbf{Q}_{\mathbf{x}_i} - \mathbf{Q}_{\mathbf{xu}_i}\mathbf{Q}^{-1}_{\mathbf{uu}_i}\mathbf{Q}_{\mathbf{u}_i} && \text{(costate Jacobian)}\\
V_{\mathbf{xx}_i} &= \mathbf{Q}_{\mathbf{xx}_i} - \mathbf{Q}_{\mathbf{xu}_i}\mathbf{Q}^{-1}_{\mathbf{uu}_i}\mathbf{Q}_{\mathbf{ux}_i} && \text{(costate Hessian)}\\
dV &= -\frac{1}{2}\mathbf{Q}^{\top}_{\mathbf{u}_i}\mathbf{Q}^{-1}_{\mathbf{uu}_i}\mathbf{Q}_{\mathbf{u}_i} && \text{(costate rate)}
\end{aligned}
$$

## Understanding the classical DDP

- Sequence of simpler Hamiltonian (Bellman)
- LQ apprx. of the Hamiltonian
- PMP to each simpler problem
- State-costate integration

Costate integration (backward pass):

$$
\begin{aligned}
V_{\mathbf{x}_N},\ V_{\mathbf{xx}_N} &= l_{\mathbf{x}_N},\ l_{\mathbf{xx}_N} && \text{(terminal condition)} \\
V_{\mathbf{x}_i} &= \mathbf{Q}_{\mathbf{x}_i} - \mathbf{Q}_{\mathbf{xu}_i}\mathbf{Q}_{\mathbf{uu}_i}^{-1}\mathbf{Q}_{\mathbf{u}_i} && \text{(costate Jacobian)} \\
V_{\mathbf{xx}_i} &= \mathbf{Q}_{\mathbf{xx}_i} - \mathbf{Q}_{\mathbf{xu}_i}\mathbf{Q}_{\mathbf{uu}_i}^{-1}\mathbf{Q}_{\mathbf{ux}_i} && \text{(costate Hessian)} \\
dV &= -\frac{1}{2}\mathbf{Q}_{\mathbf{u}_i}^{\top}\mathbf{Q}_{\mathbf{uu}_i}^{-1}\mathbf{Q}_{\mathbf{u}_i} && \text{(costate rate)}
\end{aligned}
$$

# Contact dynamics API:

# Bipedal walking

# Bipedal walking

The objective is:

- Understand how to 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`