



# Simulation #1

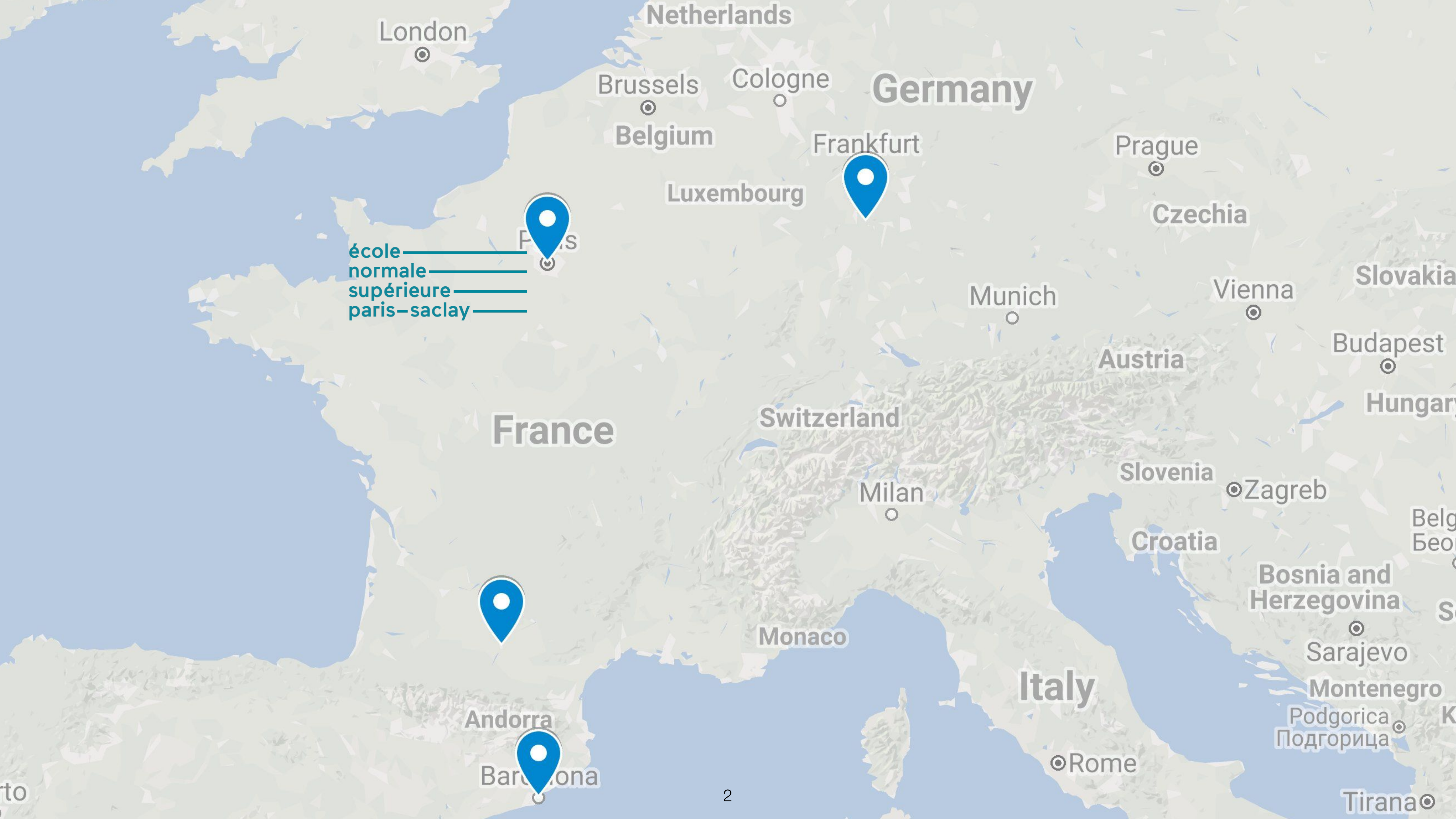
Rigid body dynamic models and algorithms

**Justin Carpentier**

*Researcher, INRIA and ENS, Paris*  
[justin.carpentier@inria.fr](mailto:justin.carpentier@inria.fr)

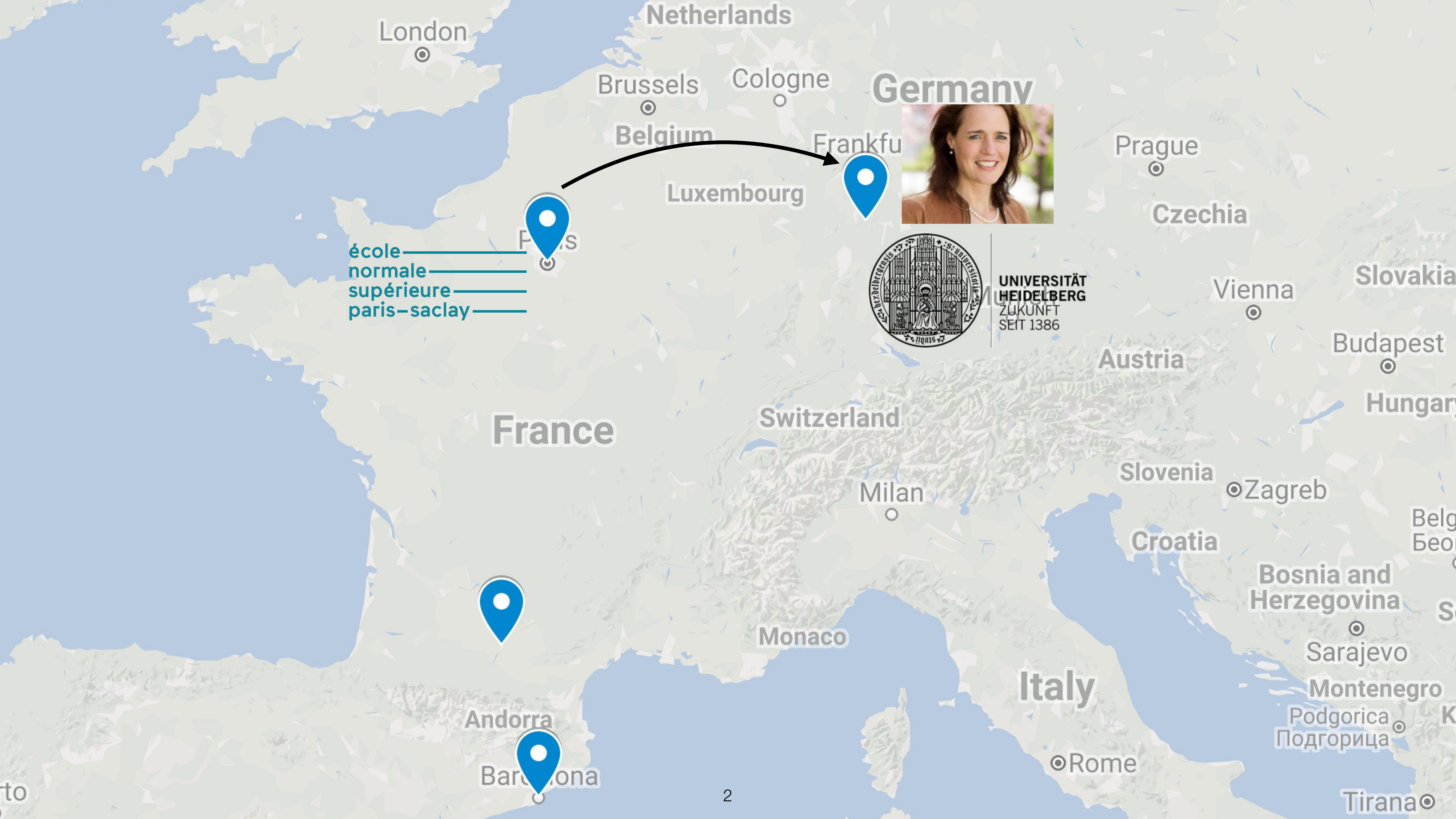






école normale supérieure paris-saclay





école  
normale  
supérieure  
paris-saclay



UNIVERSITÄT  
HEIDELBERG  
ZUKUNFT  
SEIT 1386





École  
normale  
supérieure  
paris-saclay



UNIVERSITÄT  
HEIDELBERG  
ZUKUNFT  
SEIT 1386







London

Netherlands

Germany

Prague

Czechia

Slovakia

Vienna

Budapest

Hungary

Austria

Switzerland

Slovenia

Zagreb

Belgium

Croatia

Bosnia and Herzegovina

Sarajevo

Montenegro

Podgorica  
Подгорица

Tirana

Italy

Rome

Monaco

Milan

France

Luxembourg

Frankfurt

Paris

École normale supérieure  
paris-saclay



UNIVERSITÄT  
HEIDELBERG  
ZUKUNFT  
SEIT 1386







# Robotics @ WILLOW

Perception, Learning, and Control







## Team leader:

- \* Justin Carpentier (Inria, 100%)

## Faculties:

- \* Jean Ponce (ENS, 50%)
- \* Cordelia Schmid (Inria, 50%)
- \* Stéphane Caron (Inria, 100%)
- \* Shizhe Chen (Inria, 100%)

## External collaborators:

- \* Josef Sivic (CTU)

## Post-docs:

- \* Etienne Moullier
- \* Ajay Sathya
- \* Ewen Dantec
- \* Etienne Ménager

## Engineers:

- \* Etienne Arlaud (IR Inria)
- \* Pierre-Guillaume Raverdy (IR Inria)
- \* Joris Vaillant
- \* Mégane Millan
- \* Olivier Roussel (RAIMBOW-DEFROST)

## Visitors:

- \* Kateryna Zorina (CTU Prague)
- \* Bruce Wingo (Georgia Tech)

## Assistant:

- \* ♥ Julien Guieu ♥



## PhD students:

- \* Antoine Bambade
- \* Yann De Mont Marin
- \* Oumayma Bounou
- \* Quentin Le Lidec
- \* Guillaume Le Moing
- \* Lucas Ventura (ENPC)
- \* Thomas Chabal
- \* Adrien Bardes (Meta)
- \* Wilson Jallet (LAAS-CNRS)
- \* Nicolas Chahine (DXO)
- \* Elliot Vincent (ENPC)
- \* Théo Bodrito
- \* Matthieu Futeral-Peter
- \* Zerui Chen
- \* Gabriel Fiastre
- \* Zeeshan Khan
- \* François Garderes (LV)
- \* Umit Bora Gokbakan (LAAS-CNRS)
- \* Ludovic de Maestris (LAAS-CNRS)
- \* Fabian Schramm (ISIR)
- \* Ricardo Garcia Pinel
- \* Louis Montaut (CTU)
- \* Roland Andrews
- \* Armand Jordana (NYU)



**Axe 1**  
Visual recognition  
for images and videos

**Axe 2**  
Learning embedded  
representations



**Willow<sup>2</sup>**

**Axe 4**  
Modeling, analysis and extraction  
of 3D objects and scenes

**Axe 3**  
Image restoration and  
enhancement



**Axe 5**  
Robotics as  
a Data Science

**Axe 1**  
Visual recognition  
for images and videos

**Axe 2**  
Learning embedded  
representations



**Willow<sup>2</sup>**

**Axe 4**  
Modeling, analysis and extraction  
of 3D objects and scenes

**Axe 3**  
Image restoration and  
enhancement



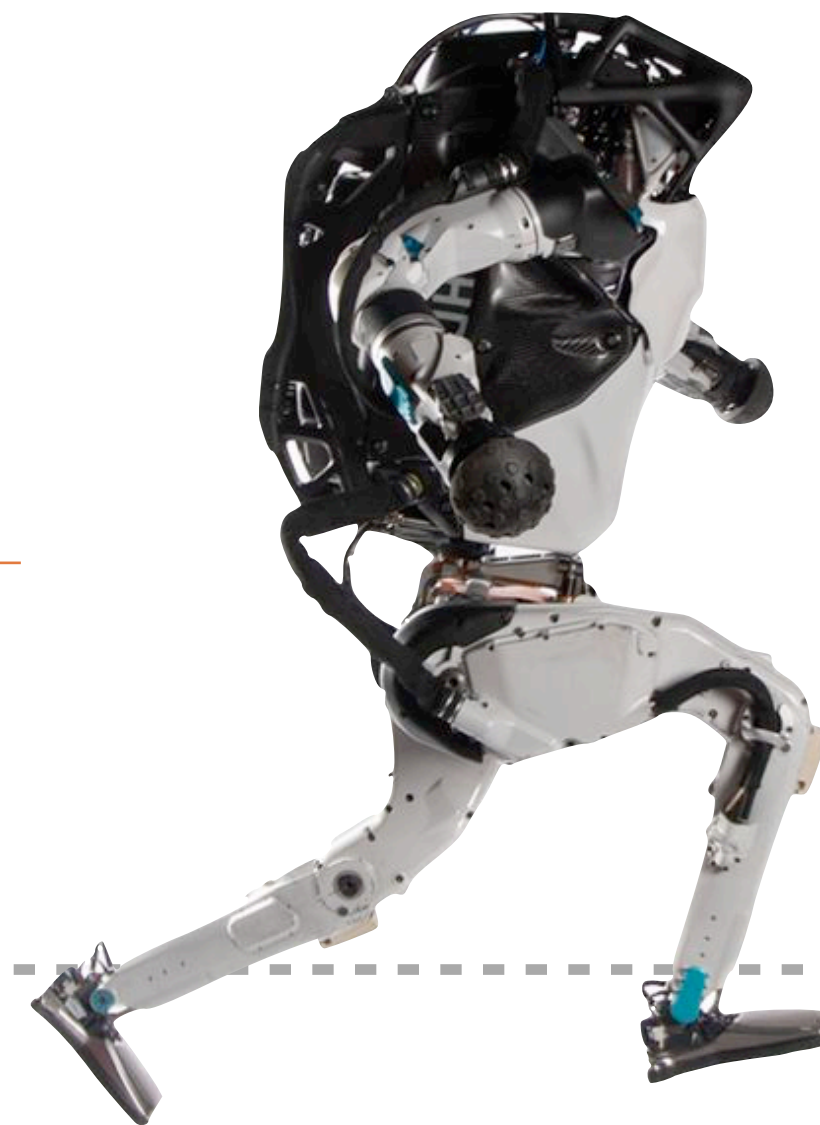
# Robotics: a *multidisciplinary* perspective

Optimization

Machine Learning

Perception

Control



SOFTWARE

HARDWARE

Electrical  
Engineering

Mechanical  
Engineering



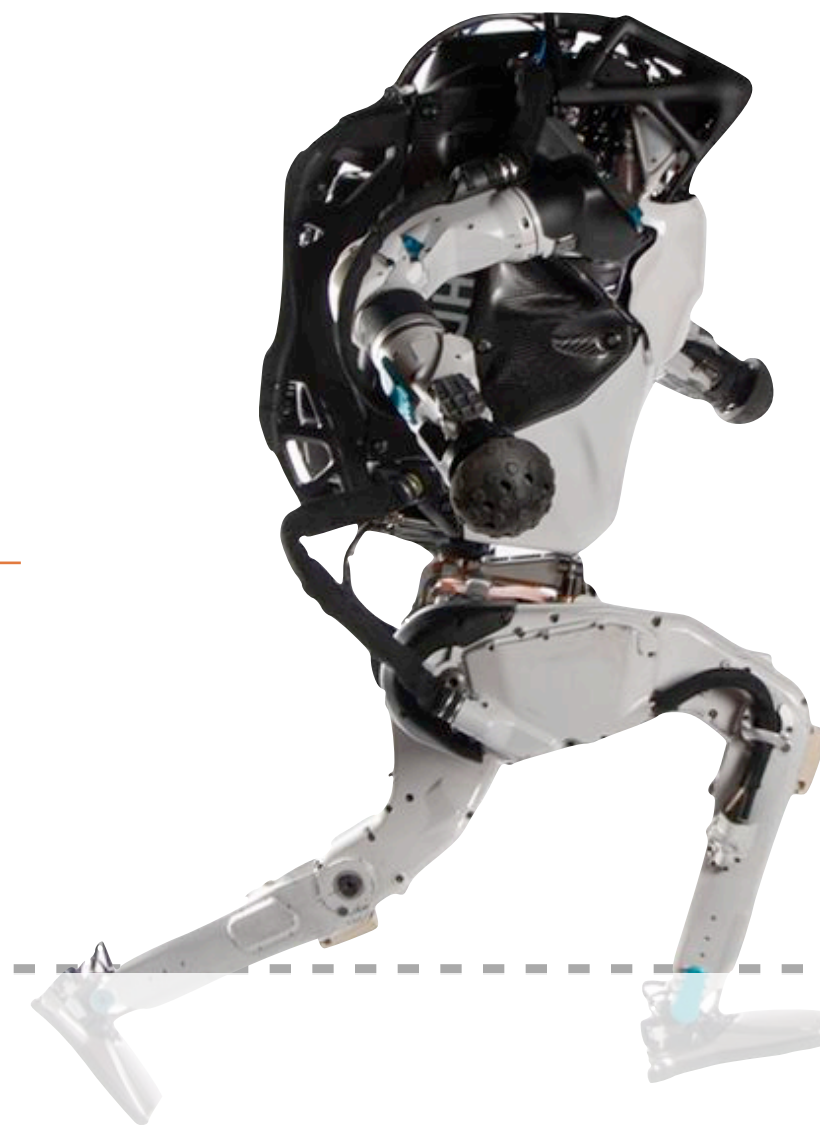
# Robotics: a *multidisciplinary* perspective

Optimization

Machine Learning

Perception

Control



SOFTWARE

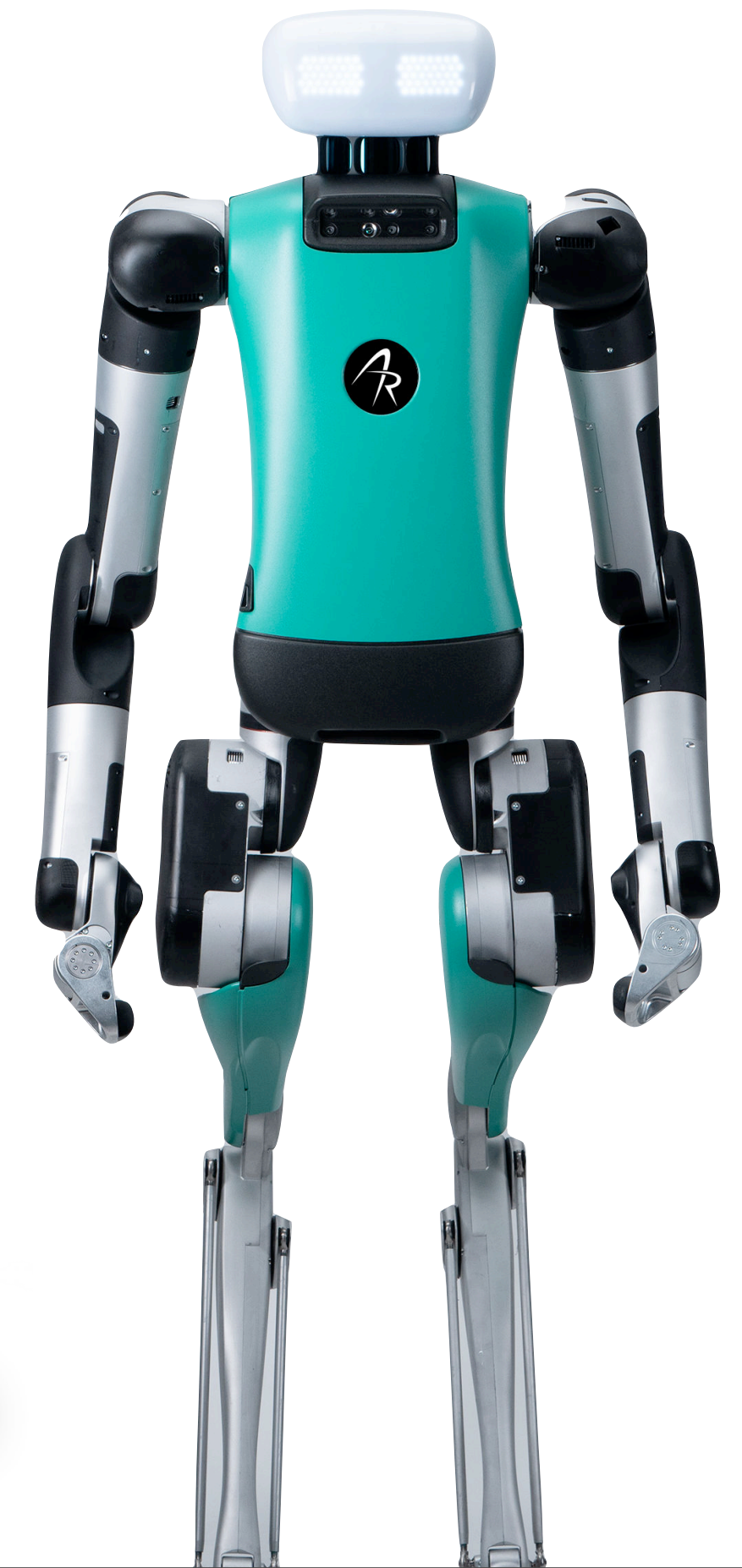
HARDWARE

Electrical  
Engineering

Mechanical  
Engineering



# Pushing the limits of AGILITY and DEXTERITY



manipulation

loco-manipulation

locomotion





# Open-source and efficient Robotics software

The Swiss army knife for robotics



# Open-source and generic-purposed robotics software



PROXIMAL ALGORITHMS FOR ROBOTICS AND BEYOND









**WE'RE  
HIRING!**



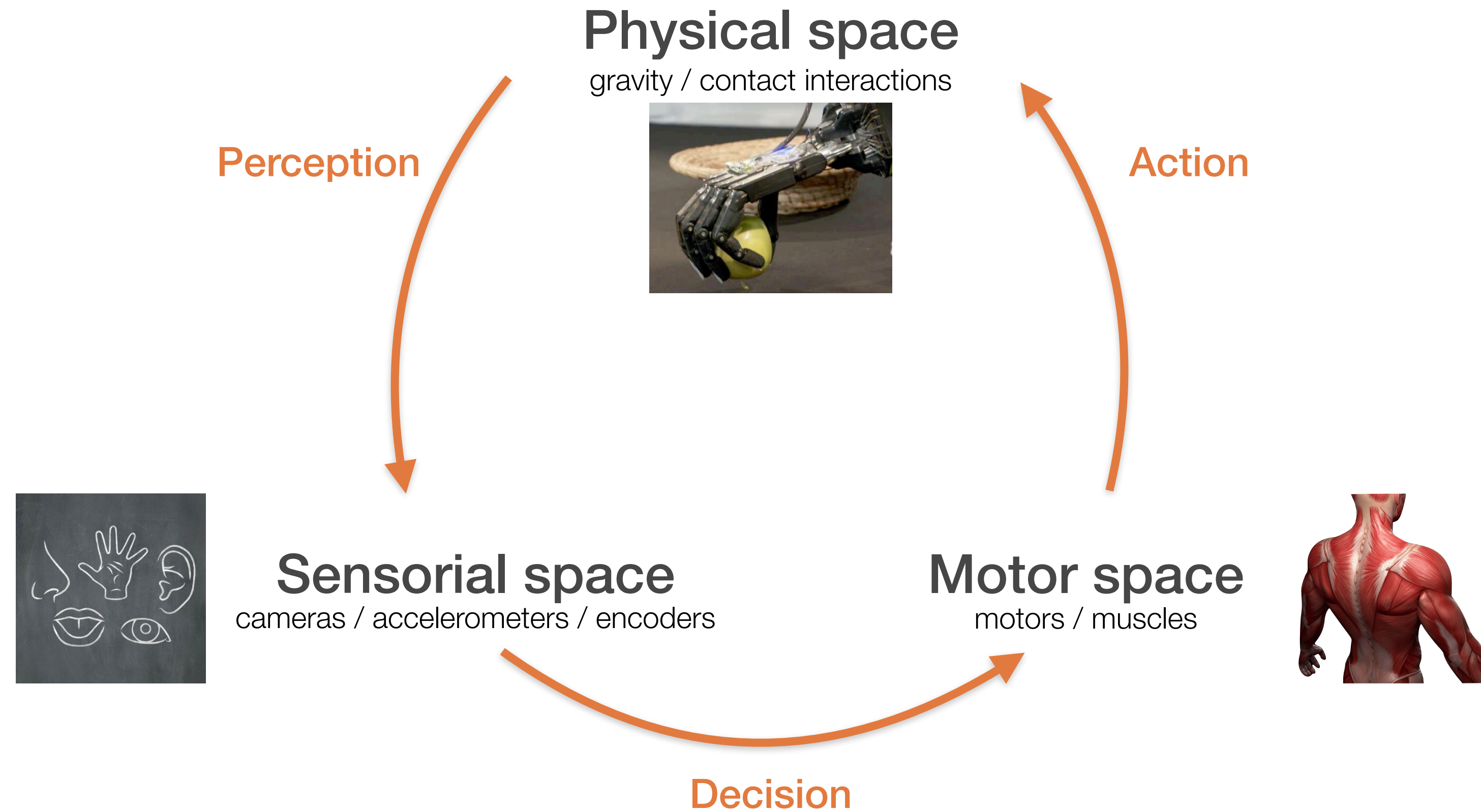


# Physical simulation in robotics

The whys and wherefores

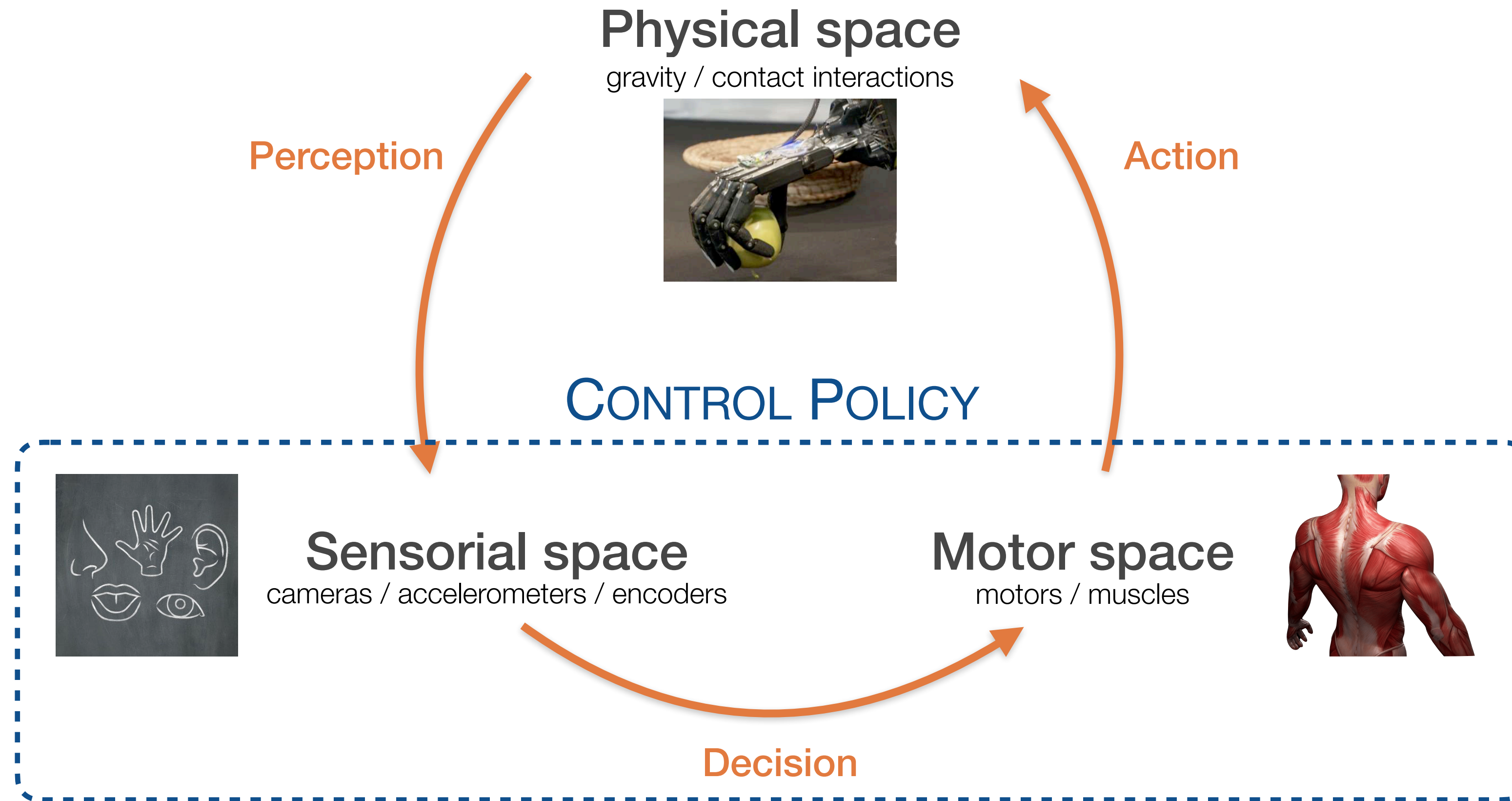


# Robotics: the modern way of controlling a robot



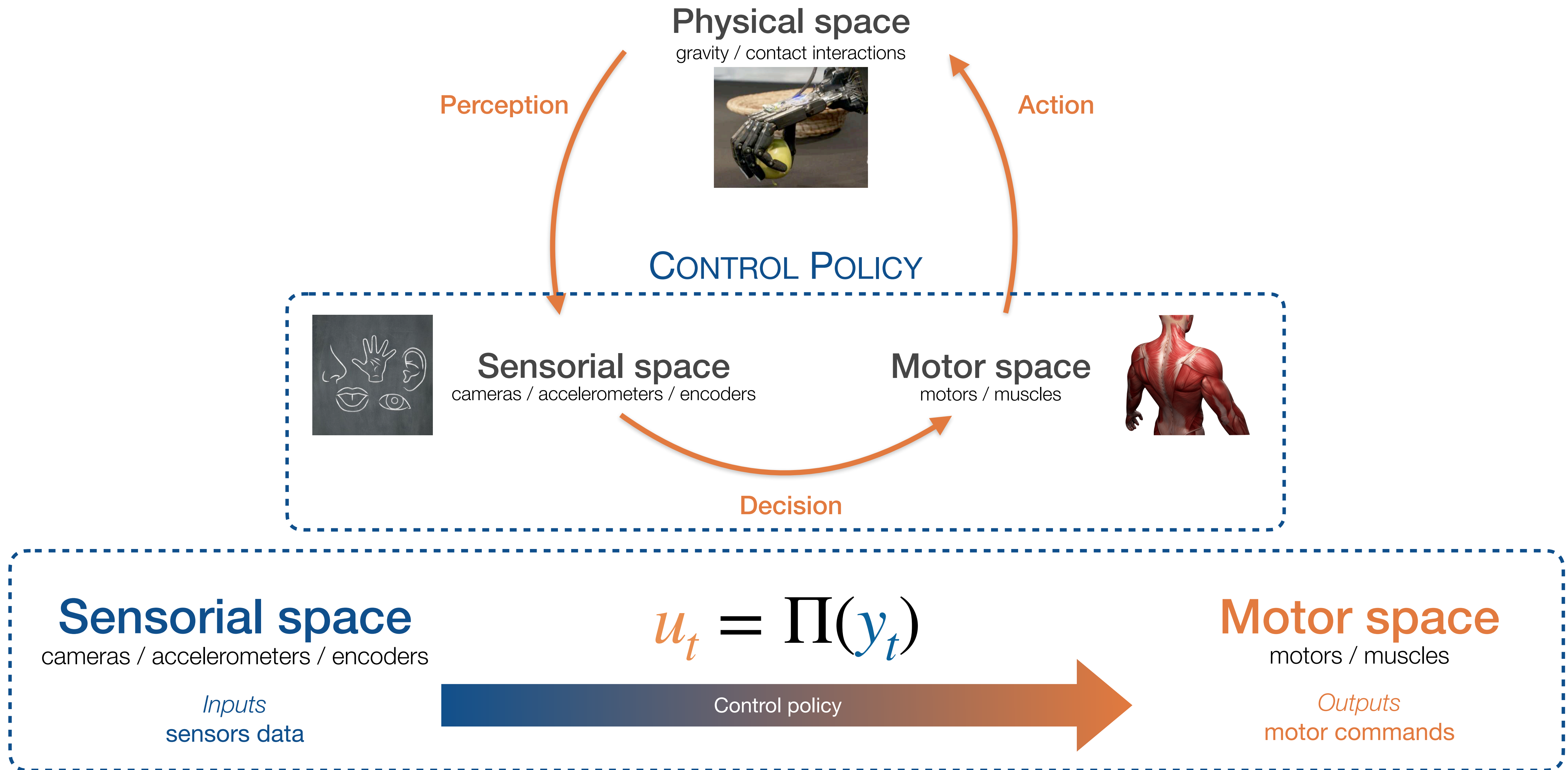


# Robotics: the modern way of controlling a robot





# Robotics: the modern way of controlling a robot





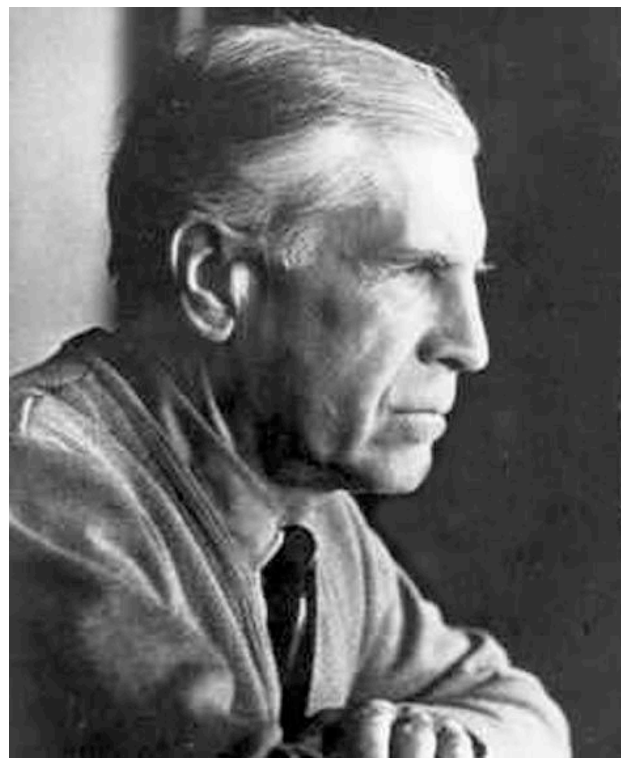
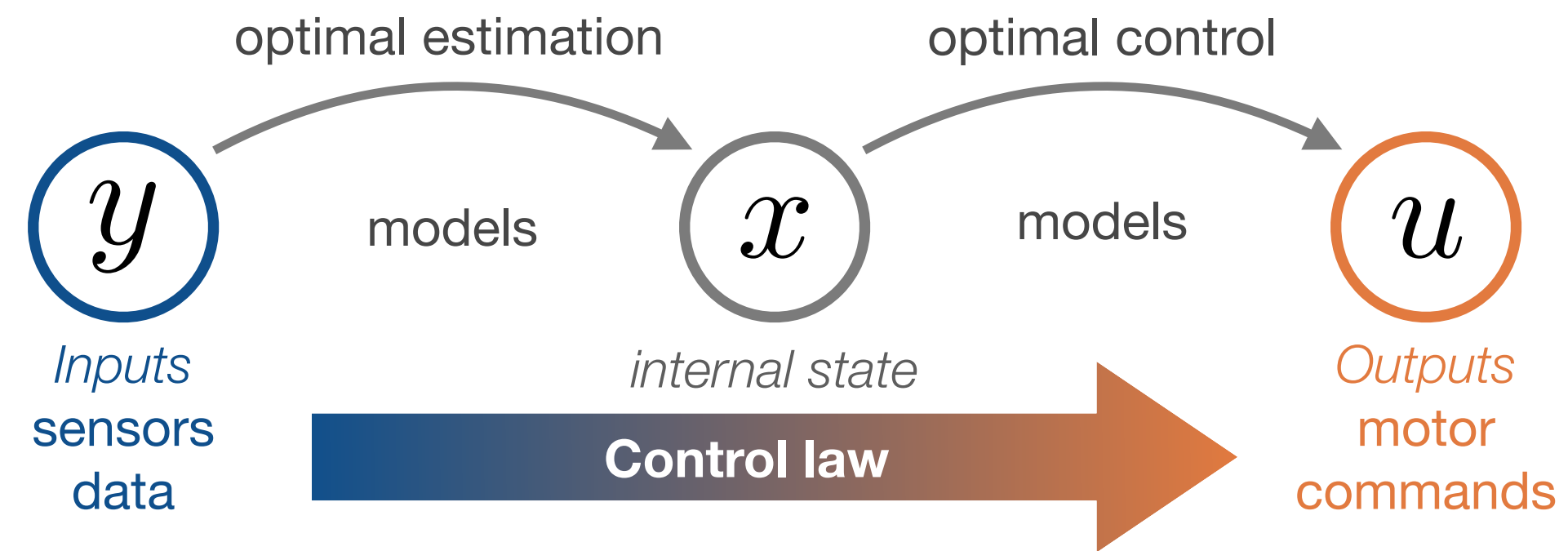
# Robotics: the modern way of controlling a robot



# Robotics: the modern way of controlling a robot

## OPTIMAL CONTROL

$$\min_{x(\cdot), u(\cdot)} \int_0^T l_t(x(t), u(t)) dt$$
$$\dot{x}(t) = f_t(x(t), u(t))$$
$$x(0) = x_0$$





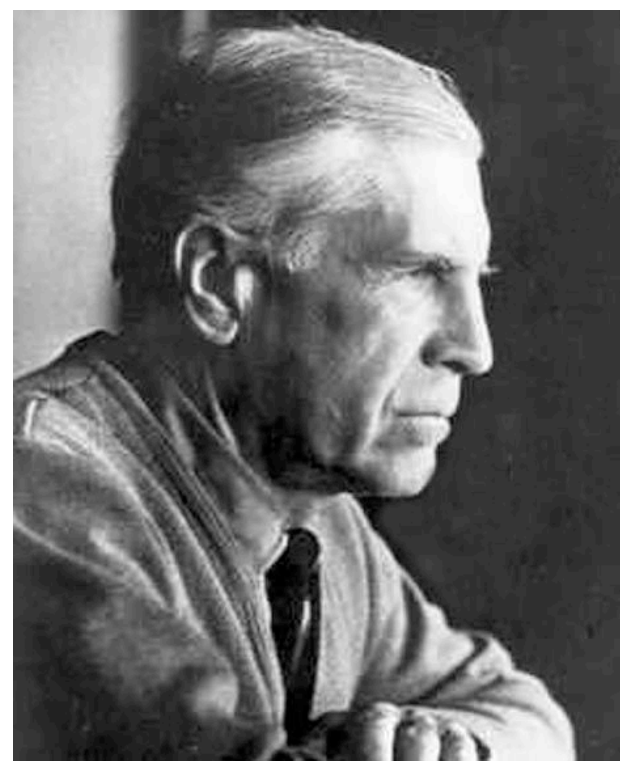
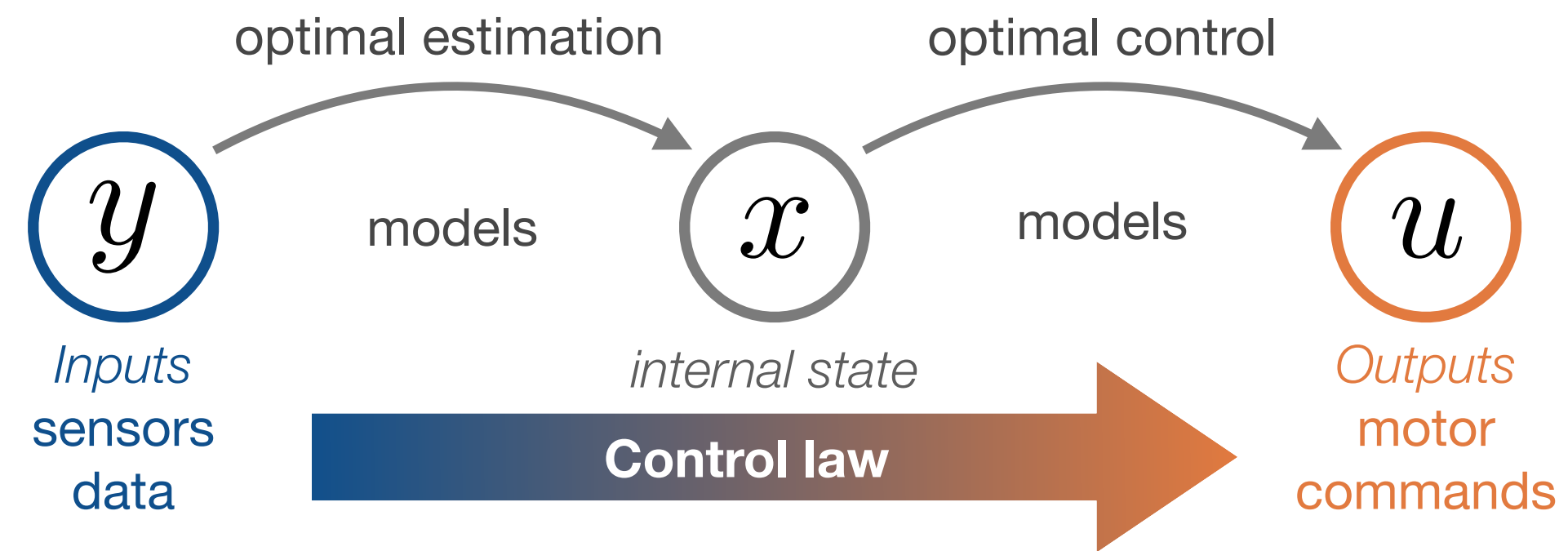
# Robotics: the modern way of controlling a robot

## OPTIMAL CONTROL

$$\min_{x(\cdot), u(\cdot)} \int_0^T l_t(x(t), u(t)) dt$$

$$\dot{x}(t) = f_t(x(t), u(t))$$

$$x(0) = x_0$$

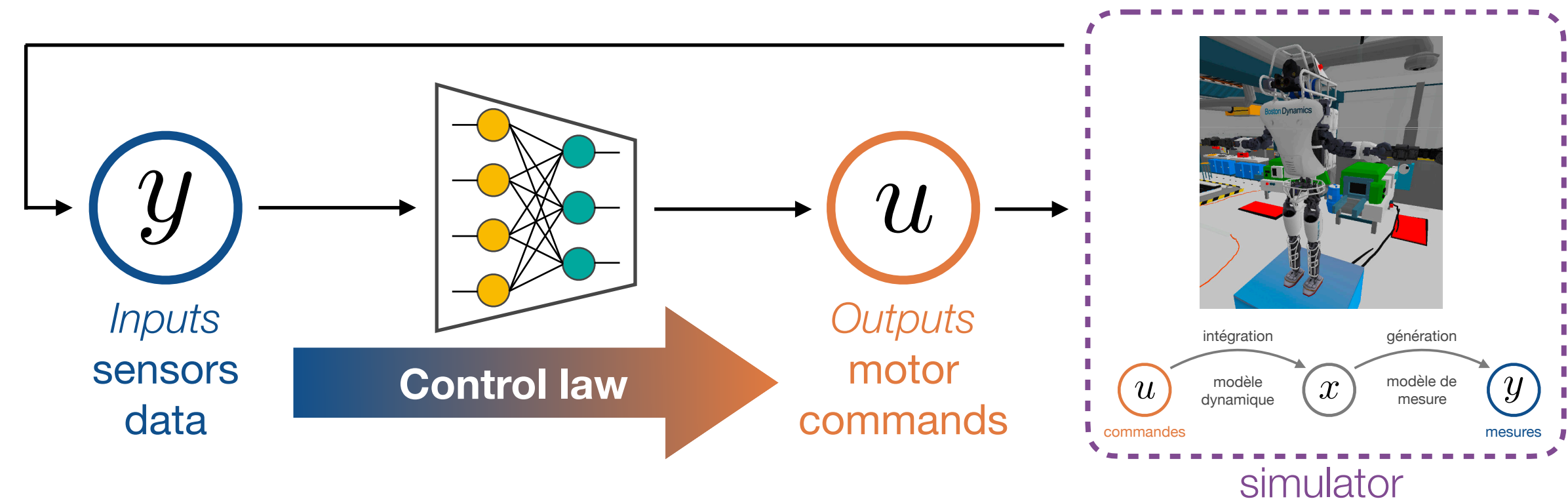


## POLICY LEARNING

$$\min_{\theta} \mathbb{E}_{\xi} \left[ \int_0^T l_t(x(t), \pi_{\theta}(x(t), \xi(t))) dt \right]$$

$$\dot{x}(t) = f_t(x(t), \pi_{\theta}(x_t, \xi_t))$$

$$x(0) \sim \xi_0$$





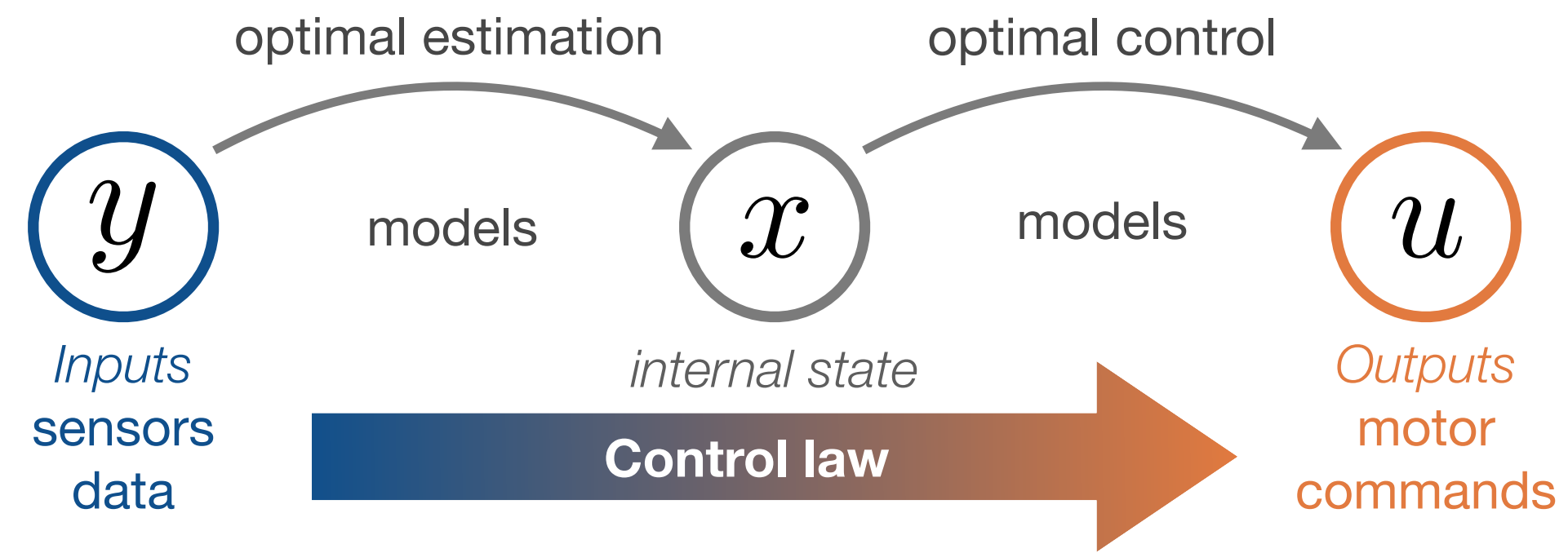
# Robotics: the modern way of controlling a robot

## OPTIMAL CONTROL

$$\min_{x(\cdot), u(\cdot)} \int_0^T l_t(x(t), u(t)) dt$$

$$\dot{x}(t) = f_t(x(t), u(t))$$

$$x(0) = x_0$$

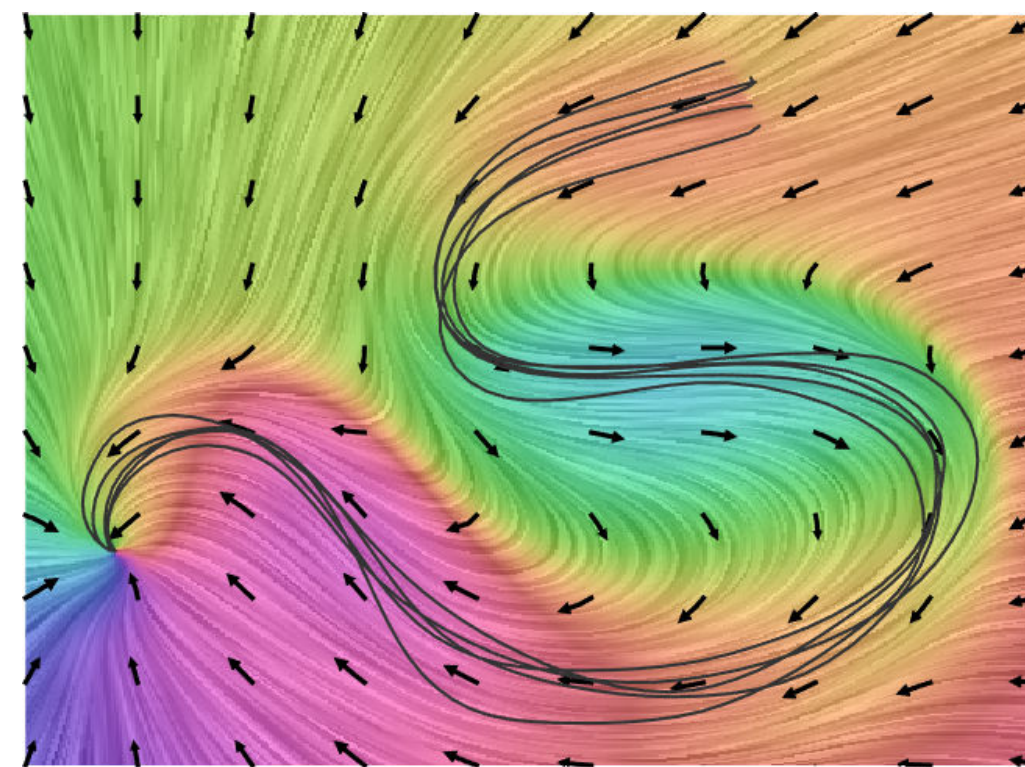
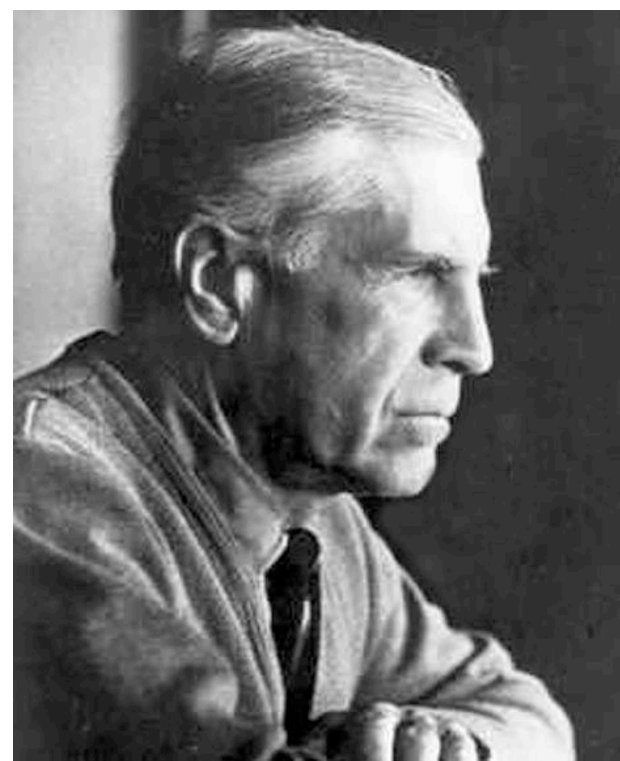
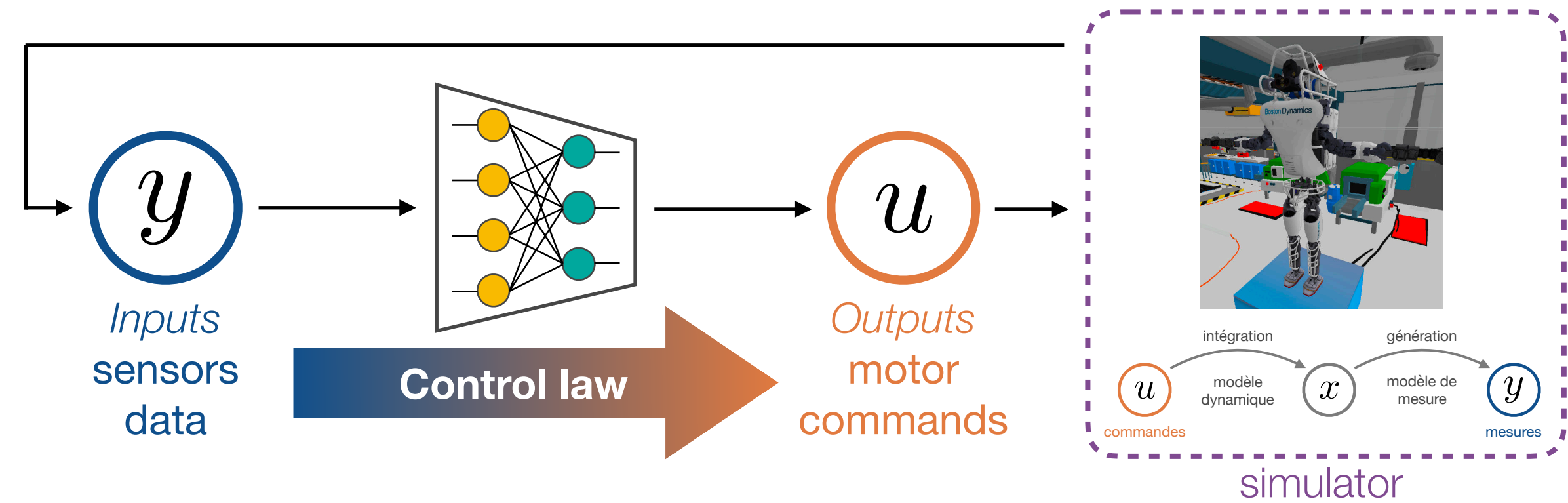


## POLICY LEARNING

$$\min_{\theta} \mathbb{E}_{\xi} \left[ \int_0^T l_t(x(t), \pi_{\theta}(x(t), \xi(t))) dt \right]$$

$$\dot{x}(t) = f_t(x(t), \pi_{\theta}(x_t, \xi_t))$$

$$x(0) \sim \xi_0$$



Source: Callinon et al.





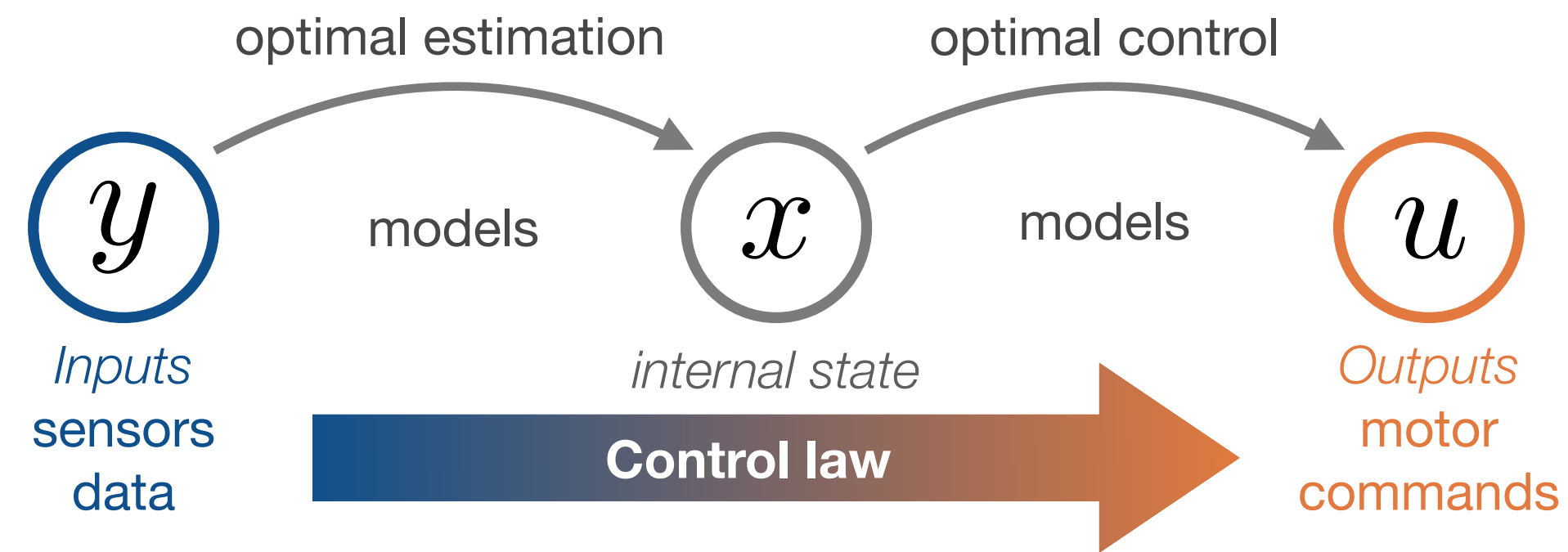
# Robotics: the modern way of controlling a robot

## OPTIMAL CONTROL

$$\min_{x(\cdot), u(\cdot)} \int_0^T l_t(x(t), u(t)) dt$$

$$\dot{x}(t) = f_t(x(t), u(t))$$

$$x(0) = x_0$$

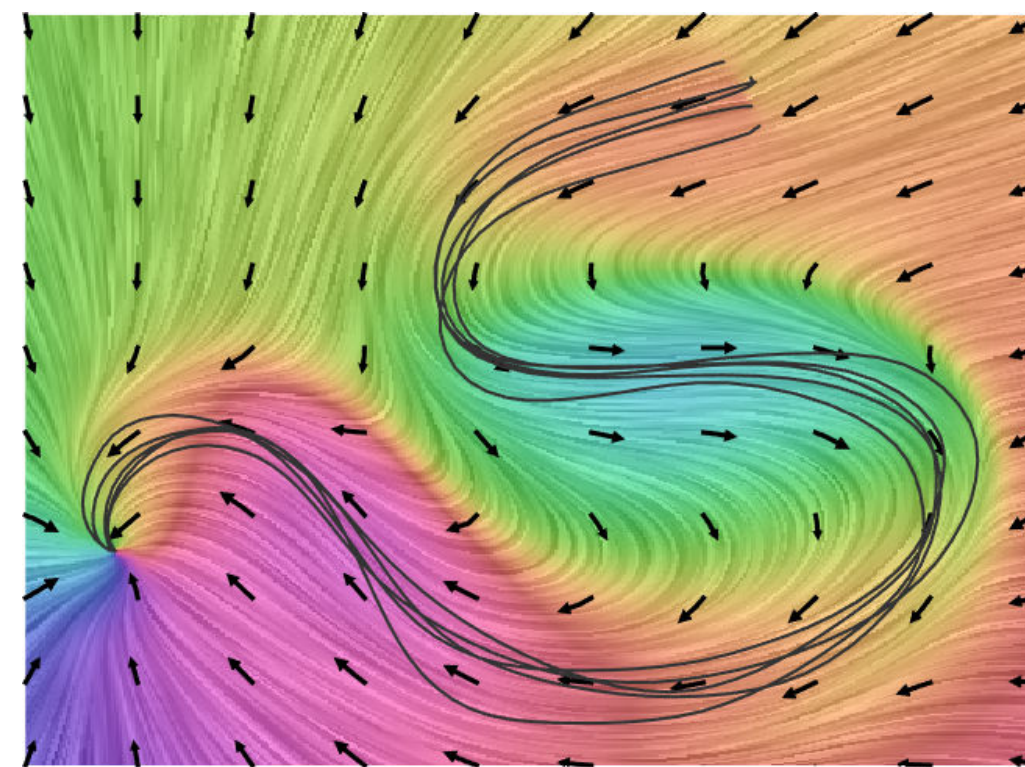
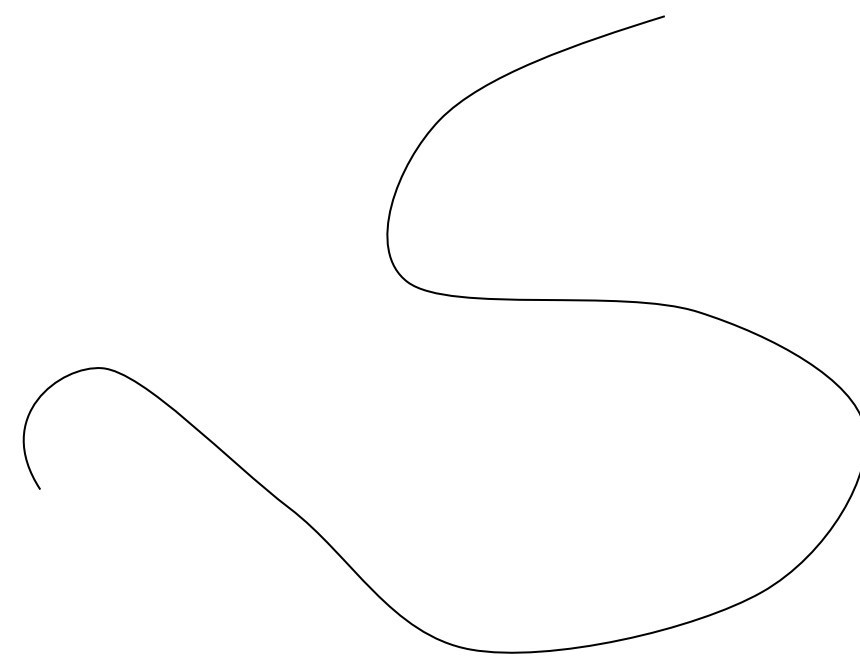
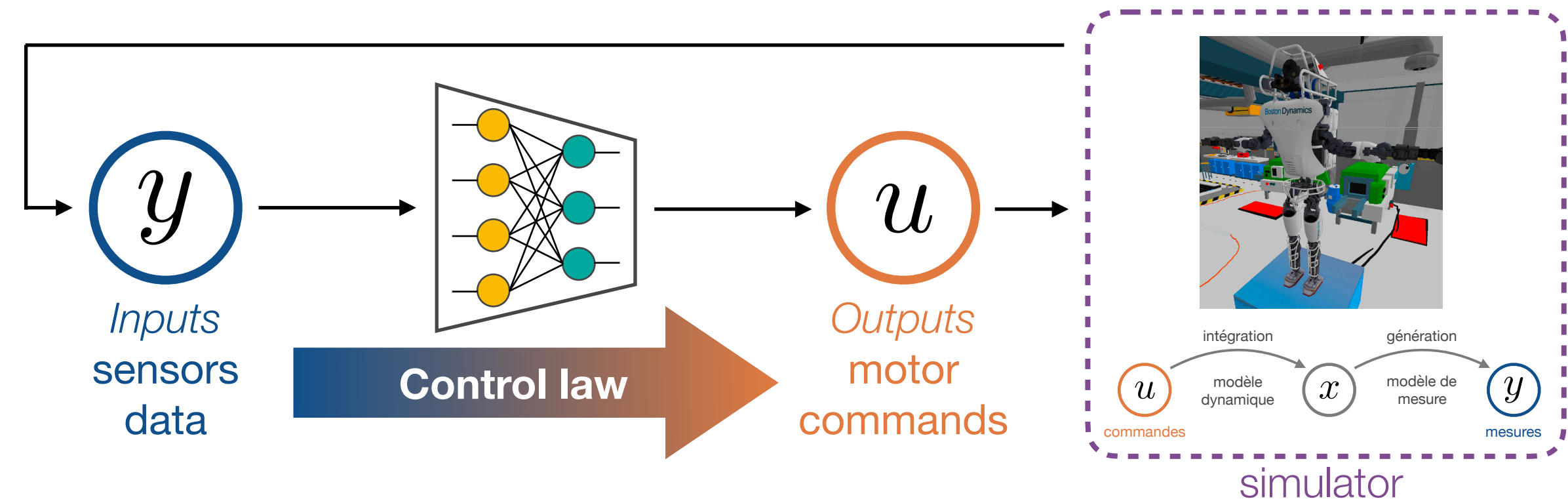


## POLICY LEARNING

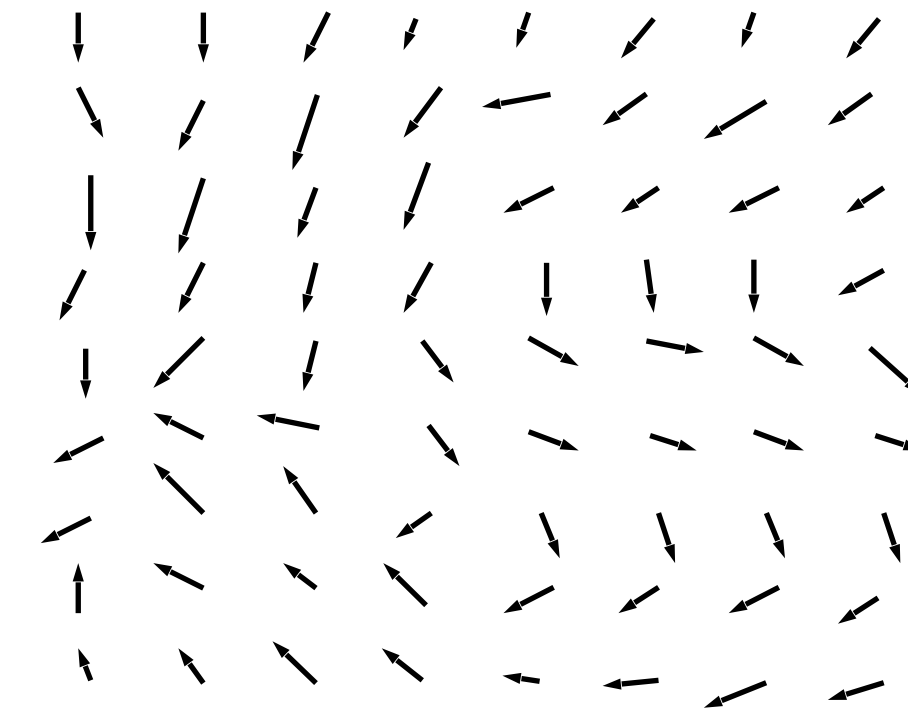
$$\min_{\theta} \mathbb{E}_{\xi} \left[ \int_0^T l_t(x(t), \pi_{\theta}(x(t), \xi(t))) dt \right]$$

$$\dot{x}(t) = f_t(x(t), \pi_{\theta}(x_t, \xi_t))$$

$$x(0) \sim \xi_0$$

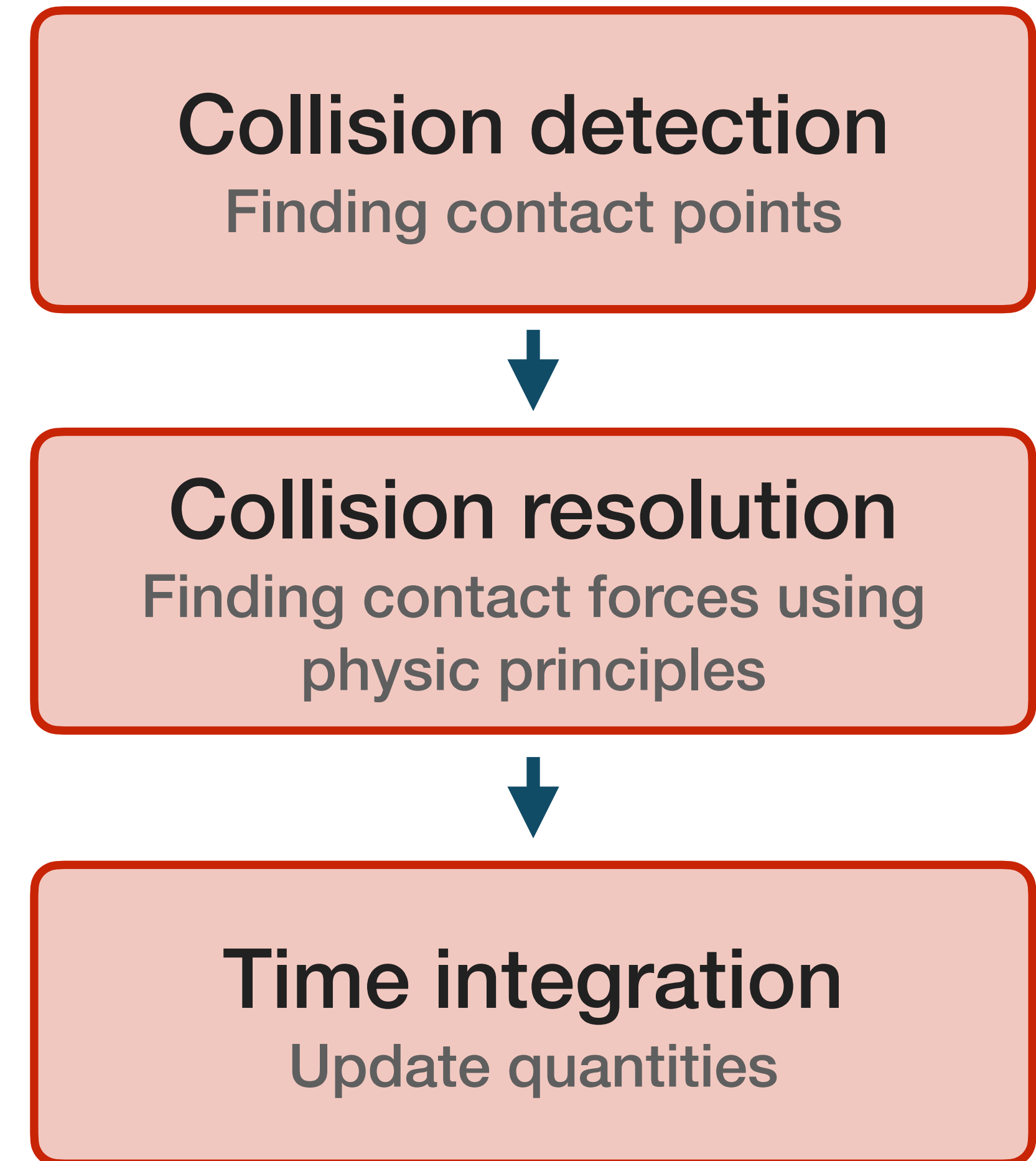
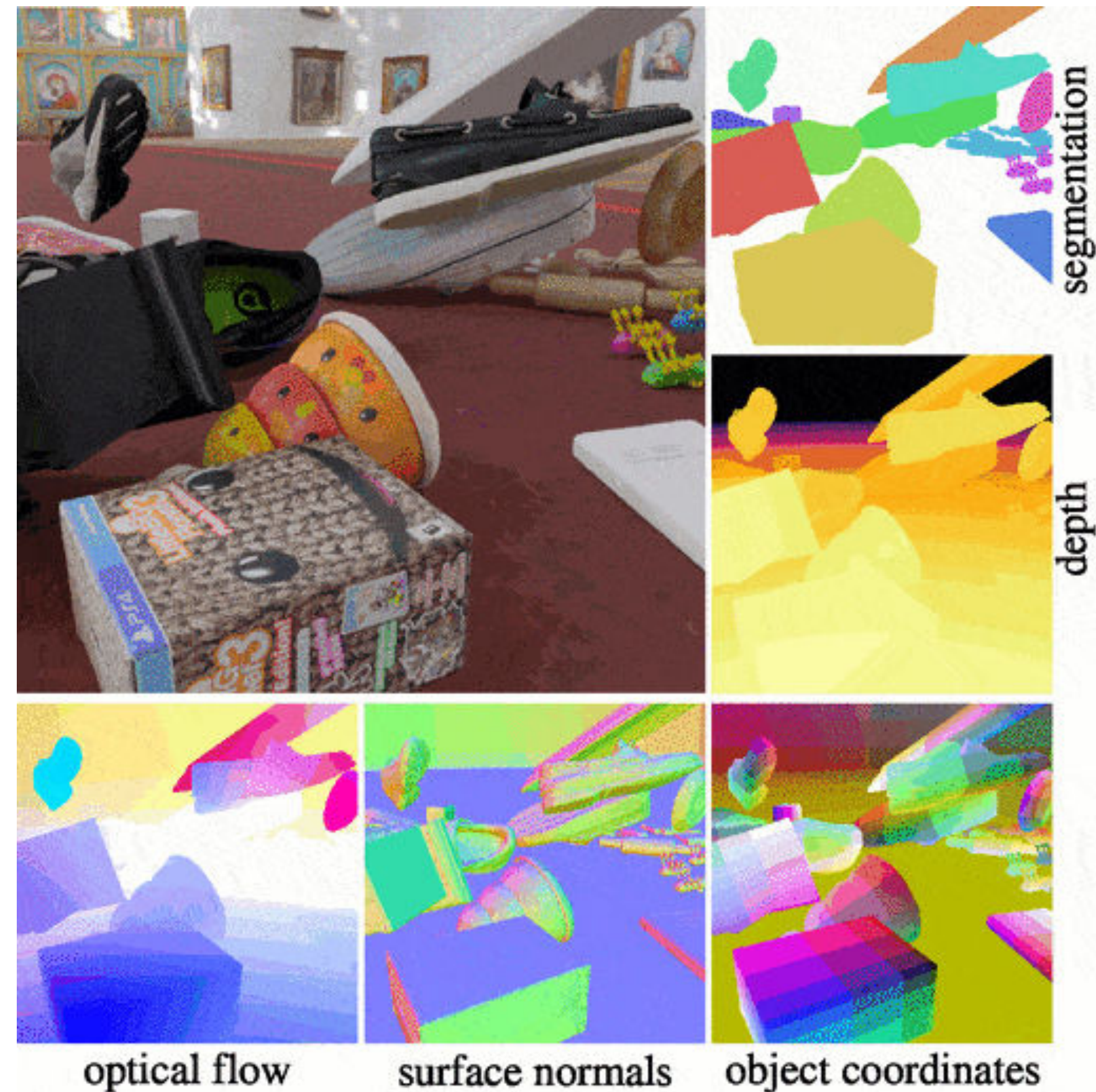


Source: Callinon et al.



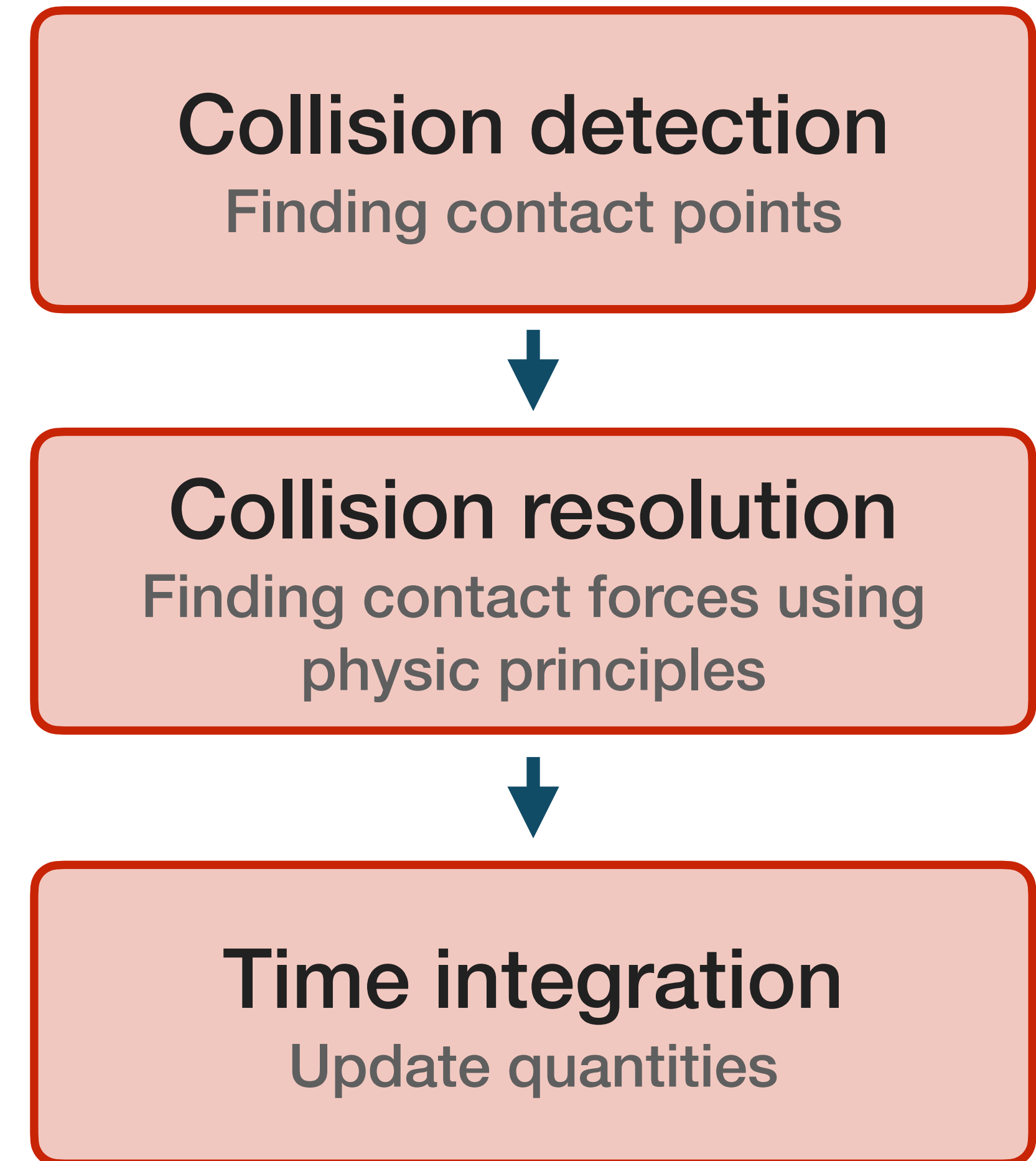
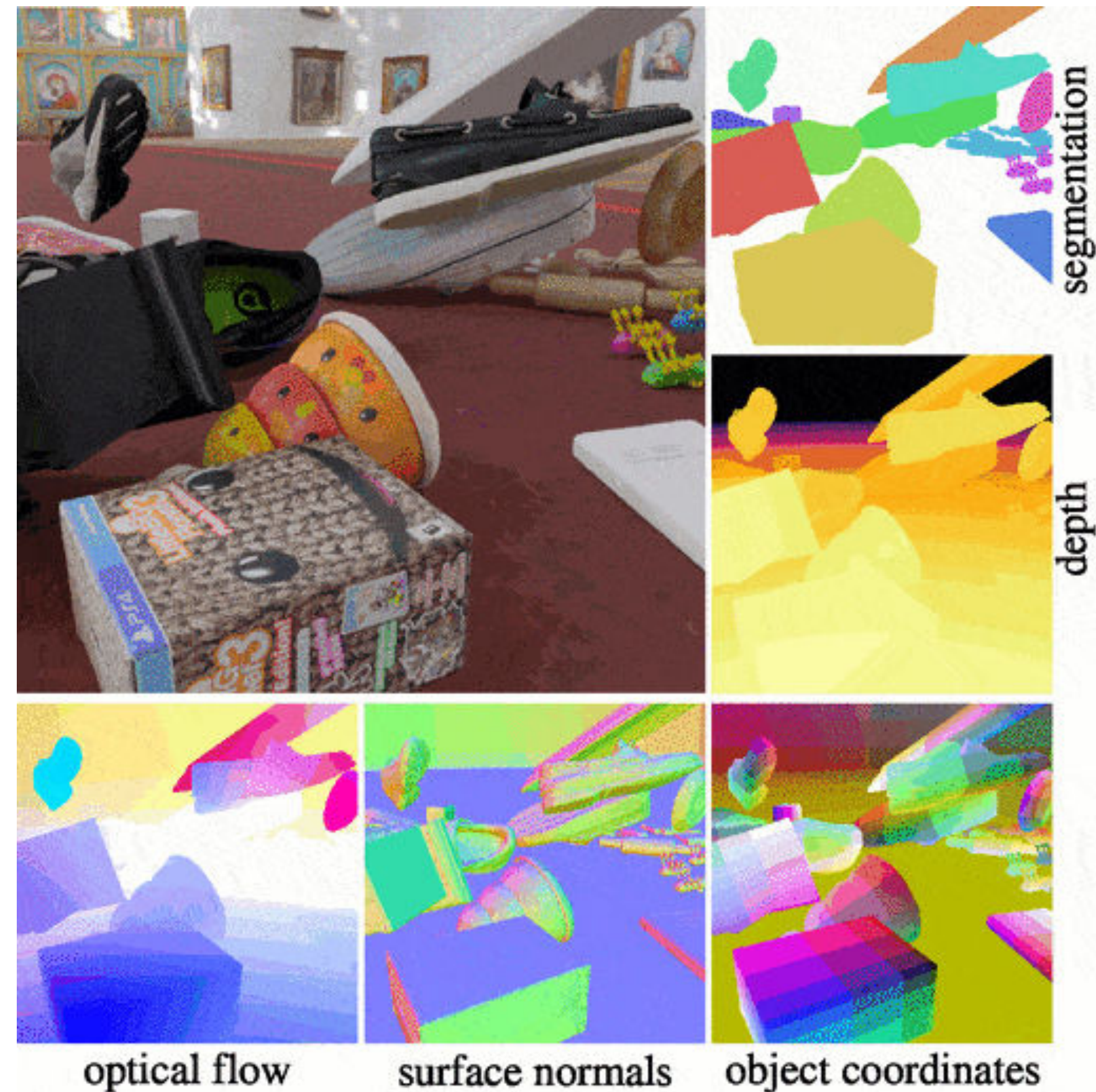


# Physical simulation





# Physical simulation





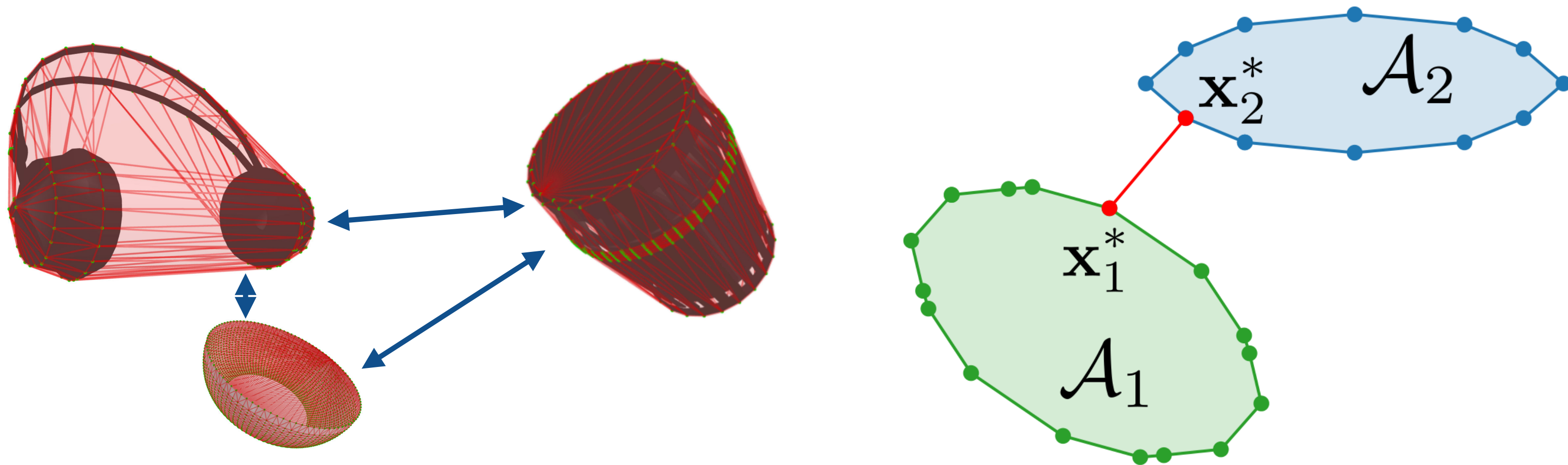


Louis Montaut

# Collision detection

Collision detection  
Finding contact points

Find the **closest points** between the two collision geometries:



IEEE TRANSACTIONS ON ROBOTICS, VOL. [?], NO. [?], MONTH 2023

## GJK++: Leveraging Acceleration Methods for Faster Collision Detection

Louis Montaut, Quentin Le Lidec, Vladimir Petrik, Josef Sivic and Justin Carpentier

**Abstract**—Collision detection is a fundamental computational problem in various domains, such as robotics, computational physics, and computer graphics. In general, collision detection is tackled as a computational geometry problem, with the so-called Gilbert, Johnson, and Keerthi (GJK) algorithm being the most adopted solution nowadays. While introduced in 1988, GJK remains the most effective solution to compute the distance or the collision between two 3D convex geometries. Over the years, it was shown to be efficient, scalable, and generic, operating on a broad class of convex shapes, ranging from simple primitives (sphere, ellipsoid, box, cone, capsule, etc.) to complex meshes involving thousands of vertices. In this article, we introduce several contributions to accelerate collision detection and distance computation between convex geometries by leveraging the fact that these two problems are fundamentally optimization problems. Notably, we establish that the GJK algorithm is a specific sub-case of the well-established Frank-Wolfe (FW) algorithm in convex optimization. By adapting recent works linking Polyak and Nesterov accelerations to Frank-Wolfe methods, we also propose two accelerated extensions of the classic GJK algorithm. Through an extensive benchmark over millions of collision pairs involving objects of daily life, we show that these two accelerated GJK extensions significantly reduce the overall computational burden of collision detection, leading to up to two times faster computation timings. Finally, we hope this work will significantly reduce the computational cost of modern robotic simulators, allowing the speed-up of modern robotic applications that heavily rely on simulation, such as reinforcement learning or trajectory optimization.

**Index Terms**—Convex Optimization, Collision Detection, Computational Geometry, Computer Graphics, Simulation, Trajectory Optimization, Motion Planning

### I. INTRODUCTION

**PHYSICS** engines designed to simulate rigid bodies are an essential tool used in a wide variety of applications, notably in robotics, video games, and computer graphics [1]–[3]. Collision detection, a crucial feature of any physics engine or robot motion planner [4]–[6], consists of finding which objects are colliding or not, *i.e.* are sharing at least one common point or if there exists a separating hyper-plane between both. As simulation often needs to deal with multiple objects and run in real-time (*i.e.*, in video games) or at very high frequencies (*i.e.*,

Louis Montaut is with Inria, Département d'Informatique de l'École Normale Supérieure, PSL Research University in Paris, France and also with the Czech Institute of Informatics, Robotics and Cybernetics in Prague, Czech Republic.

Vladimir Petrik and Josef Sivic are with the Czech Institute of Informatics, Robotics and Cybernetics, Czech Technical University in Prague.

Quentin Le Lidec and Justin Carpentier are with Inria and Département d'Informatique de l'École Normale Supérieure, PSL Research University in Paris, France.

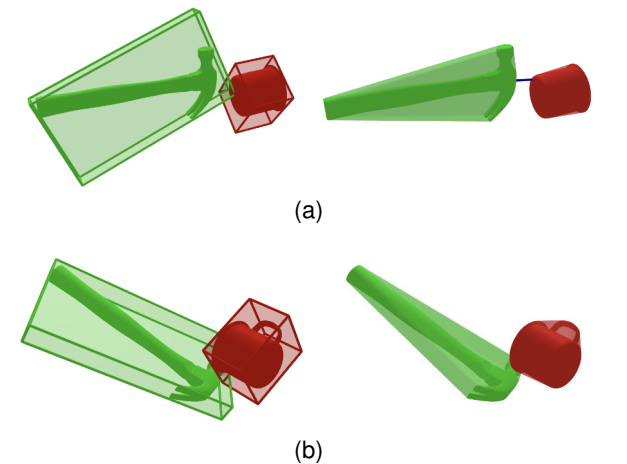


Fig. 1. Two distinct collision problems using shapes from the YCB dataset: in (a) the shapes  $\mathcal{A}_1$  and  $\mathcal{A}_2$  are not in collision ( $\text{dist}(\mathcal{A}_1, \mathcal{A}_2) > 0$ ) whereas in (b) the shapes are in collision ( $\text{dist}(\mathcal{A}_1, \mathcal{A}_2) = 0$ ). In the left column, the oriented bounding boxes (OBB) of the objects are represented in light colors. In the right column, the light colors represent the convex hull of each object. In both collision problems, (a) and (b), the broad phase finds a collision between the object's OBBs; the narrow phase must thus be called to confirm or affirm the collision. The right column corresponds to the narrow phase in which the GJK algorithm is called on the objects' convex hulls. In this paper, we propose the Polyak-accelerated GJK and Nesterov-accelerated GJK algorithms in order to accelerate collision detection.

in robotics), collision detection must be carried out as fast as possible. To reduce computational times, collision detection is usually decomposed into two phases thoroughly covered in [7]. The first phase is the so-called *broad phase* which consists in identifying which pair of simulated objects are potentially colliding. The broad phase relies on the simulated objects' bounding volumes, as shown in Fig. 1, allowing to quickly assess if the objects are *not* in collision. The second phase is the so-called *narrow phase* in which each pair identified in the broad phase is tested to check whether a collision is truly occurring. Collision detection during the narrow phase is the focus of this paper.

**Problem formulation.** We consider two convex shapes  $\mathcal{A}_1$  and  $\mathcal{A}_2$  in  $\mathbb{R}^n$  (with  $n = 2$  or  $3$  in common applications). If the shapes are not convex, we use their respective convex hulls or decompose them into a collection of convex sub-shapes [8]. The *separation distance* between  $\mathcal{A}_1$  and  $\mathcal{A}_2$ , denoted by  $\text{dist}(\mathcal{A}_1, \mathcal{A}_2) \in \mathbb{R}_+$ , can be formulated as a

0000-0000/00S00.00 © 2023 IEEE





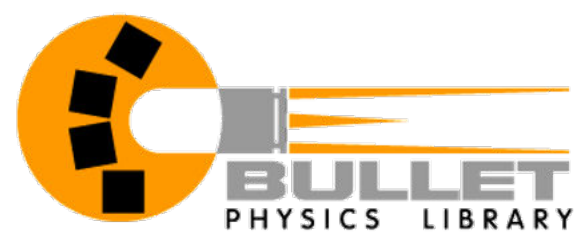
Quentin Le Lidec

# Collision resolution

**Collision resolution**  
Finding contact forces using  
physic principles

Compute the **resulting contact forces** between geometries

	Signorini	Coulomb	MDP	No shift	No internal forces	Robust	Convergence guarantees
<b>LCP</b>							
PGS [30], [29], [60], [31]	✓			✓			✓
Staggered projections [34]	✓			✓	✓	✓	
<b>CCP</b>							
PGS [61]		✓	✓	✓			✓
MuJoCo [32]		✓	✓		✓	✓	✓
ADMM (Alg. 3)		✓	✓	✓	✓	✓	✓
<b>RaiSim</b> [33]	✓	✓		✓			
<b>NCP</b>							
PGS	✓	✓	✓	✓			✓
Staggered projections [6]	✓	✓	✓	✓	✓	✓	



## Contact Models in Robotics: a Comparative Analysis

Quentin Le Lidec<sup>1,†</sup>, Wilson Jallet<sup>1,2</sup>, Louis Montaut<sup>1,3</sup>, Ivan Laptev<sup>1</sup>, Cordelia Schmid<sup>1</sup>, and Justin Carpentier<sup>1</sup>

**Abstract**—Physics simulation is ubiquitous in robotics. Whether in model-based approaches (e.g., trajectory optimization), or model-free algorithms (e.g., reinforcement learning), physics simulators are a central component of modern control pipelines in robotics. Over the past decades, several robotic simulators have been developed, each with dedicated contact modeling assumptions and algorithmic solutions. In this article, we survey the main contact models and the associated numerical methods commonly used in robotics for simulating advanced robot motions involving contact interactions. In particular, we recall the physical laws underlying contacts and friction (i.e., Signorini condition, Coulomb’s law, and the maximum dissipation principle), and how they are transcribed in current simulators. For each physics engine, we expose their inherent physical relaxations along with their limitations due to the numerical techniques employed. Based on our study, we propose theoretically grounded quantitative criteria on which we build benchmarks assessing both the physical and computational aspects of simulation. We support our work with an open-source and efficient C++ implementation of the existing algorithmic variations. Our results demonstrate that some approximations or algorithms commonly used in robotics can severely widen the reality gap and impact target applications. We hope this work will help motivate the development of new contact models, contact solvers, and robotic simulators in general, at the root of recent progress in motion generation in robotics.

**Index Terms**—Physical simulation, Numerical optimization.

### I. INTRODUCTION

**S**IMULATION is a fundamental tool in robotics. Control algorithms, like trajectory optimization (TO) or model predictive control (MPC) algorithms, rely on physics simulators to evaluate the dynamics of the controlled system. Reinforcement Learning (RL) algorithms operate by trial and error and require a simulator to avoid time-consuming and costly failures on real hardware. Robot co-design aims at finding optimal hardware design and morphology and thus extensively rely on simulation to prevent tedious physical validation. In practice, roboticists also usually perform simulated safety checks before running a new controller on their robots. These applications are evidence for a wide range of research areas in robotics where simulation is critical.

To be effective and valuable in practice, robot simulators must meet some fidelity or efficiency levels, depending on the use case. For instance, trajectory optimization algorithms, e.g. iLQR[1] or DDP [2], [3], use physics simulation to evaluate the

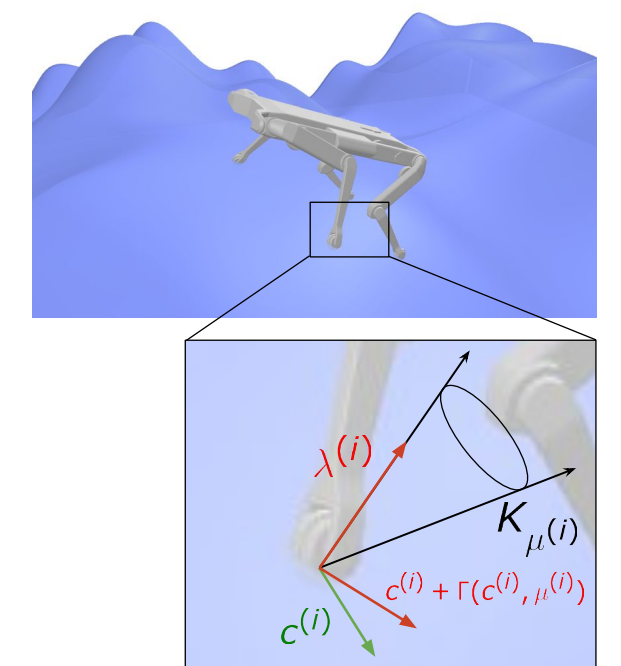


Fig. 1. Illustration of the dynamics of frictional contacts between rigid bodies which are governed by the Signorini condition, the Coulomb’s law, and the maximum dissipation principle. The combination of these three principles leads to the Non-linear Complementarity Problem (14).

system dynamics and leverage finite differences or the recent advent of differentiable simulators [4], [5], [6], [7], [8] to compute derivatives. If the solution lacks precision, the real and planned trajectories may quickly diverge, impacting *de facto* the capacity of such control solutions to be deployed on real hardware. To absorb such errors, the Model Predictive Control (MPC) [9], [10] control paradigm exploits state feedback by repeatedly running Optimal Control (OC) algorithms at high-frequency rates (e.g., 1kHz) [11], [12]. The frequency rate is one factor determining the robustness of this closed-loop algorithm to modeling errors and perturbations; thus, the efficiency of the simulation becomes critical. Although RL [13] is considered as a model-free approach, physical models are still at work to generate the samples that are indispensable for learning control policies. In fact, the vast number of required samples is the main bottleneck during training, as days or years of simulation, which corresponds to billions of calls to a simulator, are necessary [14], [15], [16]. Therefore, the efficiency of the simulator directly determines

<sup>1</sup>Inria - Département d’Informatique de l’École normale supérieure, PSL Research University. Email: [Firstname.Lastname@inria.fr](mailto:Firstname.Lastname@inria.fr)  
<sup>2</sup>LAAS-CNRS, 7 av. du Colonel Roche, 31400 Toulouse  
<sup>3</sup>Czech Institute of Informatics, Robotics and Cybernetics, Czech Technical University, Prague, Czech Republic  
<sup>†</sup>Corresponding author

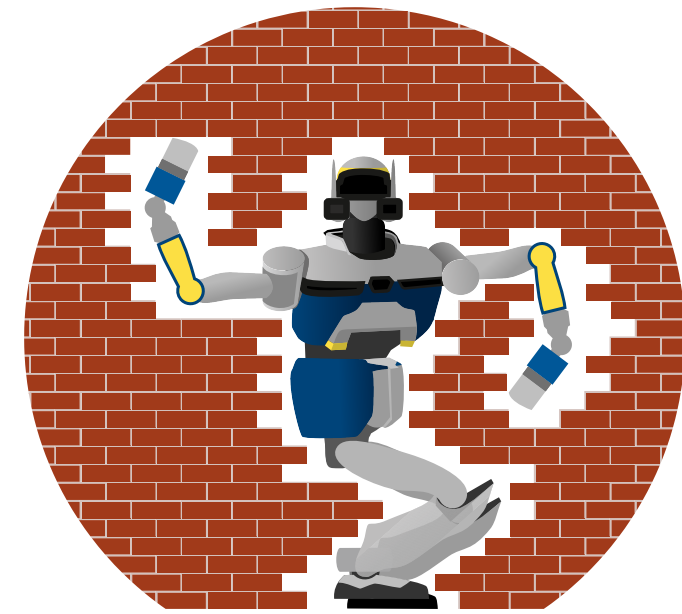


# Rigid body dynamics

The basic principles



# What is Pinocchio?



# Pinocchio

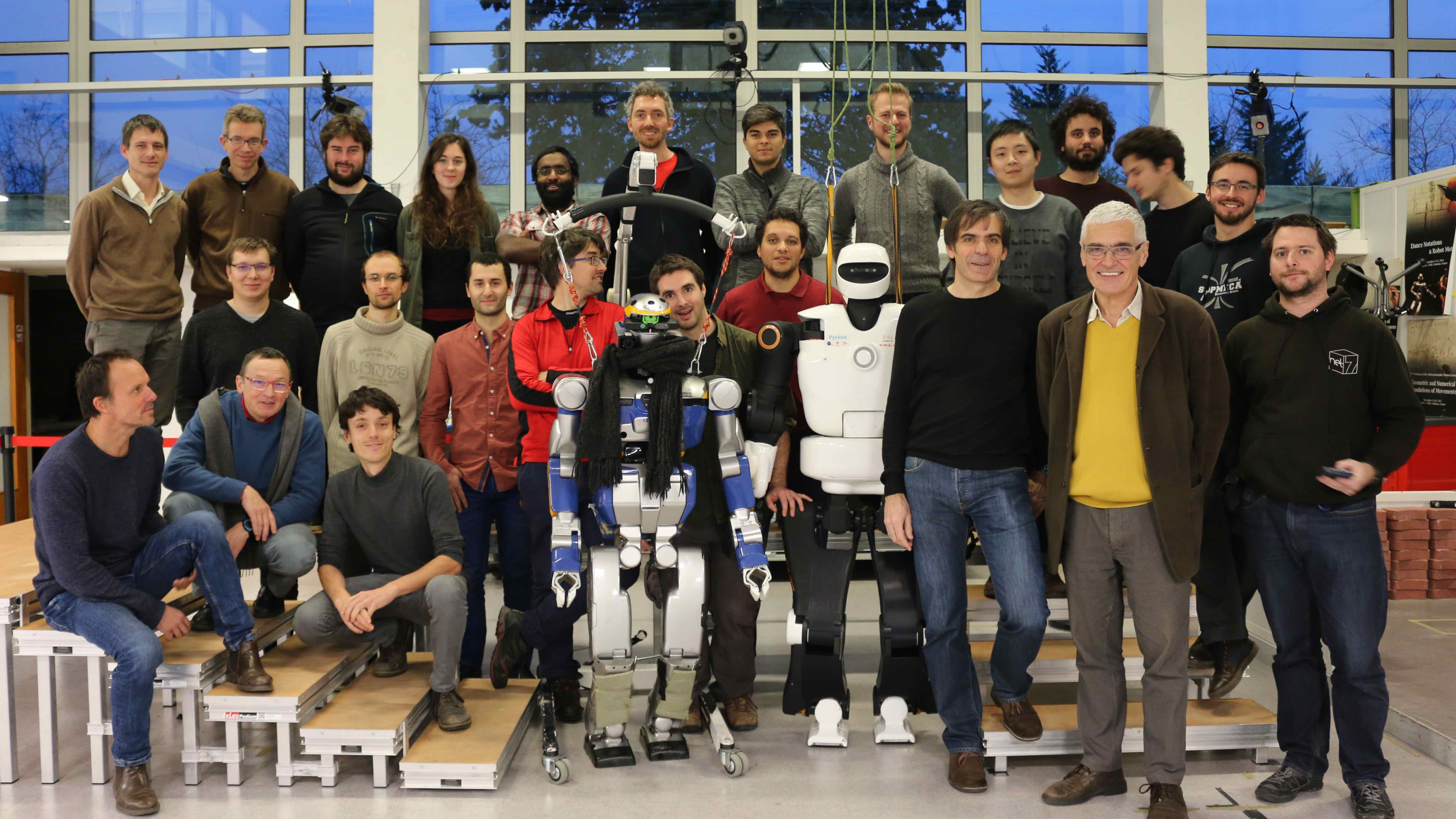
Efficient and versatile rigid body dynamics algorithms

Pinocchio is an **open-source** and **efficient** framework implementing most common **rigid body dynamics algorithms** written in **C++** and coming with **Python bindings**



[github.com/stack-of-tasks/pinocchio](https://github.com/stack-of-tasks/pinocchio)









# Pinocchio

Efficient and versatile rigid body dynamics algorithms

Pinocchio is an **open-source** and **highly efficient** framework for **simulation, planning and control** used in robotics, biomechanics, civil engineering, etc.

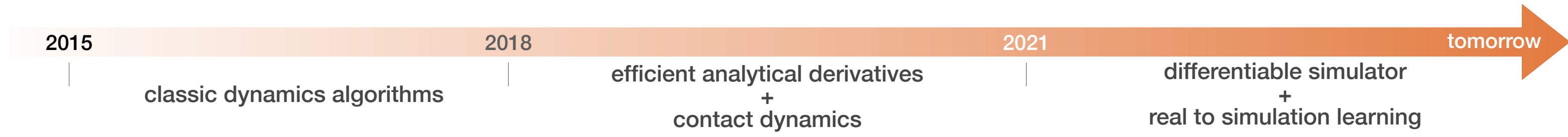
Resulting from a **joint** and **fruitful** collaboration between Willow and Gepetto (LAAS-CNRS), with an active roadmap:

**In brief:**

- ▶ C++ / Python
- ▶ BSD-2 license
- ▶ 5k+ commits
- ▶ 100k+ lines of code
- ▶ 4k downloads per day
- ▶ online documentation
- ▶ code generation CPU/GPU
- ▶ automatic differentiation
- ▶ deployed on major OS
- ▶ examples + tutorials

**Worldwide community:**

- ▶ 100+ academic labs
- ▶ 20+ universities for teaching robotics
- ▶ many robotic companies, among them:





# *A true influencer*





# The Rigid Body Dynamics Algorithms

**Goal:** exploit at best the **sparsity** induced by the kinematic tree

The Articulated Body Algorithm

$$\ddot{q} = \text{ForwardDynamics} (q, \dot{q}, \tau, \lambda_c)$$

Simulation

Control

$$\tau = \text{InverseDynamics} (q, \dot{q}, \ddot{q}, \lambda_c)$$

The Recursive Newton-Euler Algorithm

$$M(q)\ddot{q} + C(q, \dot{q}) + G(q) = \tau + J_c^T(q)\lambda_c$$

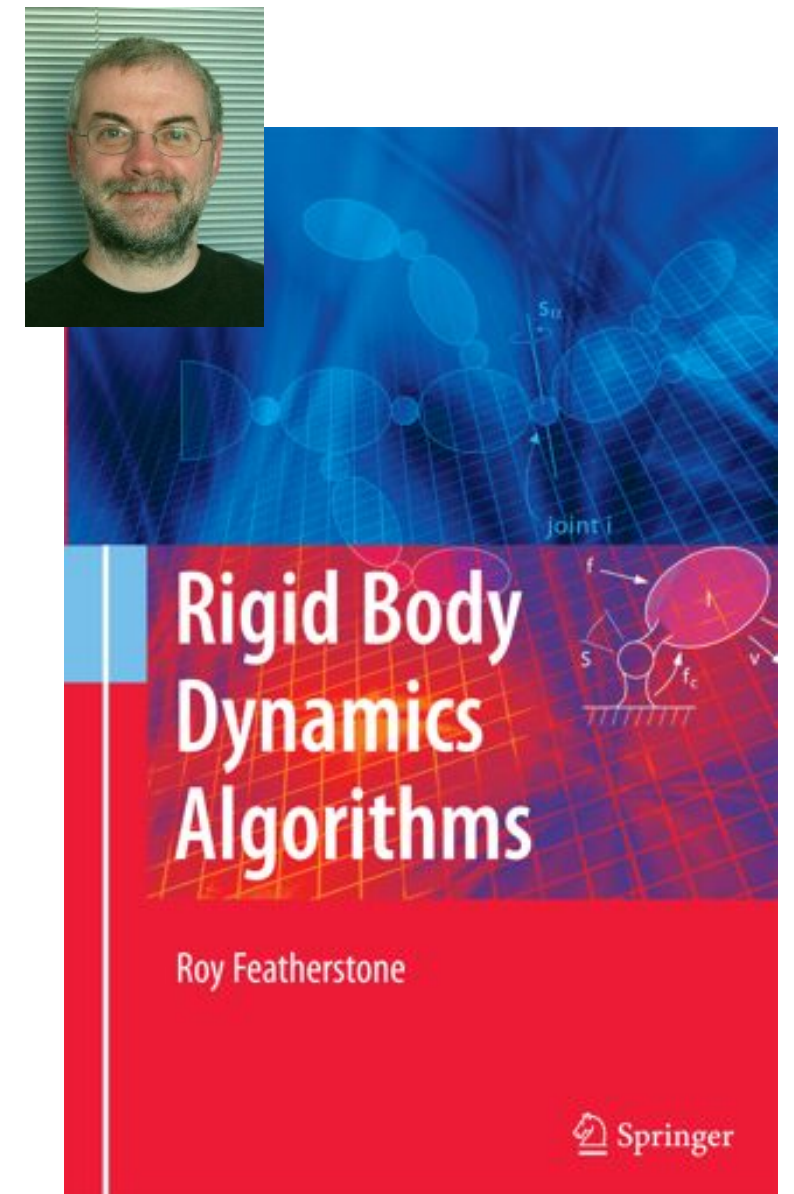
Mass  
Matrix

Coriolis  
centrifugal

Gravity

Motor  
torque

External  
forces



Roy Featherstone



# The Rigid Body Dynamics Algorithms

**Goal:** exploit at best the **sparsity** induced by the kinematic tree

## The Articulated Body Algorithm

$$\ddot{q} = \text{ForwardDynamics} (q, \dot{q}, \tau, \lambda_c)$$

Simulation

Control

$$\tau = \text{InverseDynamics} (q, \dot{q}, \ddot{q}, \lambda_c)$$

## The Recursive Newton-Euler Algorithm

$$M(q)\ddot{q} + C(q, \dot{q}) + G(q) = \tau + J_c^T(q)\lambda_c$$

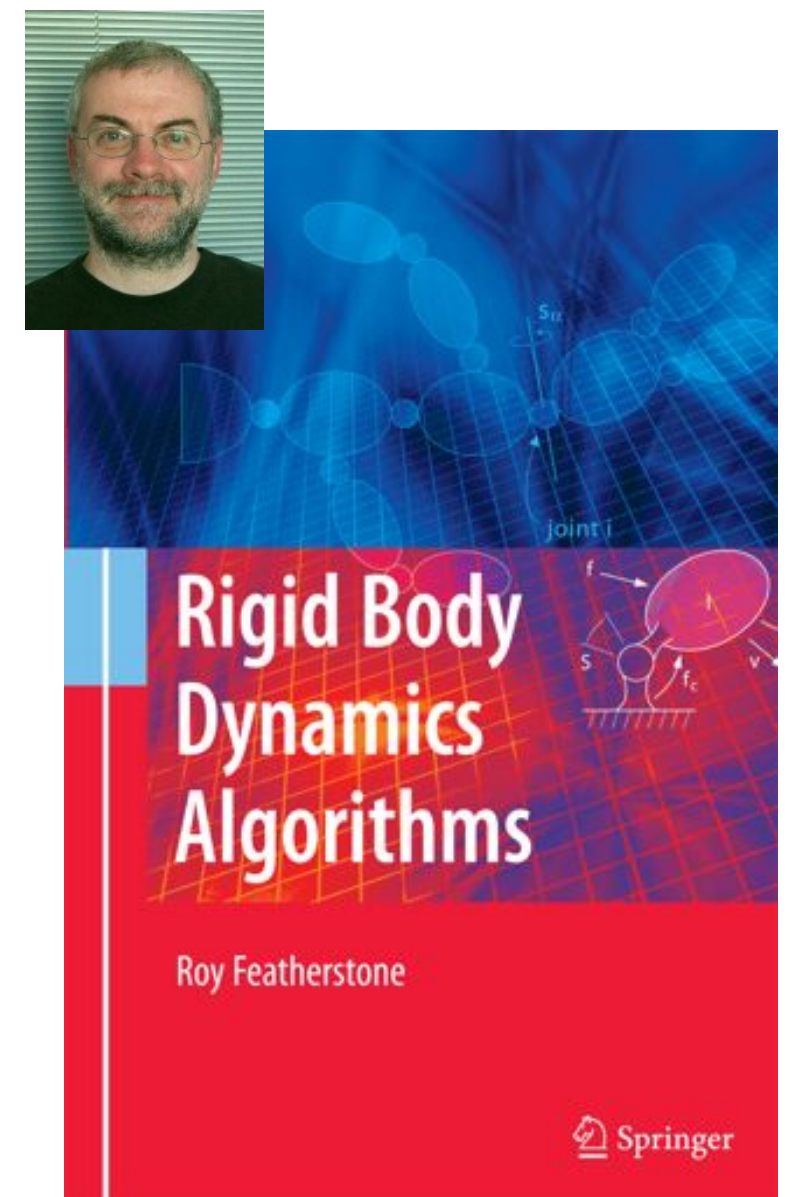
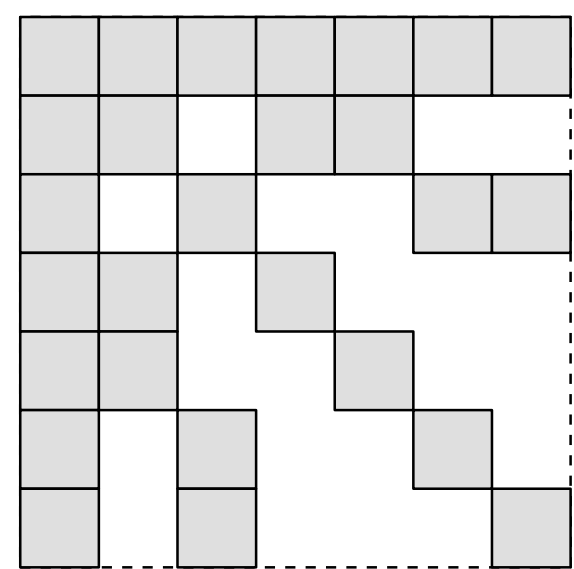
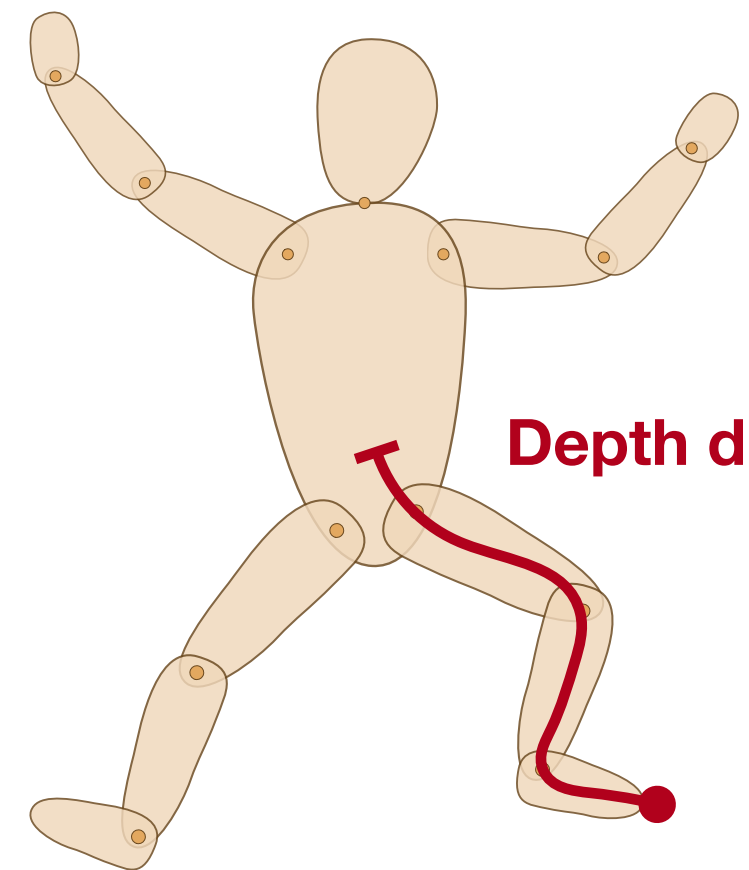
Mass Matrix

Coriolis centrifugal

Gravity

Motor torque

External forces



Roy Featherstone





# The main features of Pinocchio



# The main features of Pinocchio

- ▶ supports a large number of joints (revolute, prismatic, free-flyer, etc.) **[flexible]**



# The main features of Pinocchio

- ▶ supports a large number of joints (revolute, prismatic, free-flyer, etc.) **[flexible]**
- ▶ handles the complete sparsity via the Featherstone algorithms **[fast]**



# The main features of Pinocchio

- ▶ supports a large number of joints (revolute, prismatic, free-flyer, etc.) [flexible]
- ▶ handles the complete sparsity via the Featherstone algorithms [fast]
- ▶ implements classic + advanced rigid body dynamics algorithms [versatile]



# The main features of Pinocchio

- ▶ supports a large number of joints (revolute, prismatic, free-flyer, etc.) **[flexible]**
- ▶ handles the complete sparsity via the Featherstone algorithms **[fast]**
- ▶ implements classic + advanced rigid body dynamics algorithms **[versatile]**
- ▶ deals with Lie group geometry **[accurate]**



# The main features of Pinocchio

- ▶ supports a large number of joints (revolute, prismatic, free-flyer, etc.) **[flexible]**
- ▶ handles the complete sparsity via the Featherstone algorithms **[fast]**
- ▶ implements classic + advanced rigid body dynamics algorithms **[versatile]**
- ▶ deals with Lie group geometry **[accurate]**
- ▶ analytical derivatives **[online predictive control, reinforcement learning]**



# The main features of Pinocchio

- ▶ supports a large number of joints (revolute, prismatic, free-flyer, etc.) **[flexible]**
- ▶ handles the complete sparsity via the Featherstone algorithms **[fast]**
- ▶ implements classic + advanced rigid body dynamics algorithms **[versatile]**
- ▶ deals with Lie group geometry **[accurate]**
- ▶ analytical derivatives **[online predictive control, reinforcement learning]**
- ▶ automatic differentiation **[flexible]**



# The main features of Pinocchio

- ▶ supports a large number of joints (revolute, prismatic, free-flyer, etc.) **[flexible]**
- ▶ handles the complete sparsity via the Featherstone algorithms **[fast]**
- ▶ implements classic + advanced rigid body dynamics algorithms **[versatile]**
- ▶ deals with Lie group geometry **[accurate]**
- ▶ analytical derivatives **[online predictive control, reinforcement learning]**
- ▶ automatic differentiation **[flexible]**
- ▶ source code generation **[dedicated to each architecture]**



# The main features of Pinocchio

- ▶ supports a large number of joints (revolute, prismatic, free-flyer, etc.) **[flexible]**
- ▶ handles the complete sparsity via the Featherstone algorithms **[fast]**
- ▶ implements classic + advanced rigid body dynamics algorithms **[versatile]**
- ▶ deals with Lie group geometry **[accurate]**
- ▶ analytical derivatives **[online predictive control, reinforcement learning]**
- ▶ automatic differentiation **[flexible]**
- ▶ source code generation **[dedicated to each architecture]**
- ▶ Python bindings **[fast prototyping]**



# The main features of Pinocchio

- ▶ supports a large number of joints (revolute, prismatic, free-flyer, etc.) **[flexible]**
- ▶ handles the complete sparsity via the Featherstone algorithms **[fast]**
- ▶ implements classic + advanced rigid body dynamics algorithms **[versatile]**
- ▶ deals with Lie group geometry **[accurate]**
- ▶ analytical derivatives **[online predictive control, reinforcement learning]**
- ▶ automatic differentiation **[flexible]**
- ▶ source code generation **[dedicated to each architecture]**
- ▶ Python bindings **[fast prototyping]**
- ▶ multi-thread friendly **[fast]**



# The main features of Pinocchio

- ▶ supports a large number of joints (revolute, prismatic, free-flyer, etc.) [flexible]
- ▶ handles the complete sparsity via the Featherstone algorithms [fast]
- ▶ implements classic + advanced rigid body dynamics algorithms [versatile]
- ▶ deals with Lie group geometry [accurate]
- ▶ analytical derivatives [online predictive control, reinforcement learning]
- ▶ automatic differentiation [flexible]
- ▶ source code generation [dedicated to each architecture]
- ▶ Python bindings [fast prototyping]
- ▶ multi-thread friendly [fast]
- ▶ collision detection with HPP-FCL [simulation]



# The main features of Pinocchio

- ▶ supports a large number of joints (revolute, prismatic, free-flyer, etc.) [flexible]
- ▶ handles the complete sparsity via the Featherstone algorithms [fast]
- ▶ implements classic + advanced rigid body dynamics algorithms [versatile]
- ▶ deals with Lie group geometry [accurate]
- ▶ analytical derivatives [online predictive control, reinforcement learning]
- ▶ automatic differentiation [flexible]
- ▶ source code generation [dedicated to each architecture]
- ▶ Python bindings [fast prototyping]
- ▶ multi-thread friendly [fast]
- ▶ collision detection with HPP-FCL [simulation]
- ▶ reads robot model from URDF, SDF, etc. [compatibility]



# Closed-loop kinematics and bilateral constraints

## Proximal and Sparse Resolution of Constrained Dynamic Equations

Justin Carpentier  
Inria, École normale supérieure  
CNRS, PSL Research University  
75005 Paris, France  
Email: justin.carpentier@inria.fr

Rohan Budhiraja  
Inria Paris  
75012 Paris, France  
Email: rohan.budhiraja@inria.fr

Nicolas Mansard  
LAAS-CNRS, ANITI  
University of Toulouse  
31400 Toulouse, France  
Email: nicolas.mansard@laas.fr

**Abstract**—Control of robots with kinematic constraints like loop-closure constraints or interactions with the environment requires solving the underlying constrained dynamics equations of motion. Several approaches have been proposed so far in the literature to solve these constrained optimization problems, for instance by either taking advantage in part of the sparsity of the kinematic tree or by considering an explicit formulation of the constraints in the problem resolution. Yet, not all the constraints allow an explicit formulation and in general, approaches of the state of the art suffer from singularity issues, especially in the context of redundant or singular constraints. In this paper, we propose a unified approach to solve forward dynamics equations involving constraints in an efficient, generic and robust manner. To this aim, we first (i) propose a proximal formulation of the constrained dynamics which converges to an optimal solution in the least-square sense even in the presence of singularities. Based on this proximal formulation, we introduce (ii) a sparse Cholesky factorization of the underlying Karush–Kuhn–Tucker matrix related to the constrained dynamics, which exploits at best the sparsity of the kinematic structure of the robot. We also show (iii) that it is possible to extract from this factorization the Cholesky decomposition associated to the so-called **Operational Space Inertia Matrix**, inherent to task-based control frameworks or physic simulations. These new formulation and factorization, implemented within the Pinocchio library, are benchmark on various robotic platforms, ranging from classic robotic arms or quadrupeds to humanoid robots with closed kinematic chains, and show how they significantly outperform alternative solutions of the state of the art by a factor 2 or more.

### I. INTRODUCTION

As soon as a robot makes contacts with the world or is endowed with loop closures in its design, its dynamics is governed by the constrained equations of motion. From a phenomenological point of view, these equations of motion follow the so-called least-action principle, also known under the name of the Maupertuis principle which dates back to the 17<sup>th</sup> century. This principle states that the motion of the system follows the closest possible acceleration to the free-falling acceleration (in the sense of the kinetic metric) which fulfils the constraints. In other words, solving the constrained equations of motion boils down to solving a constrained optimization problem where forces acts as the Lagrange multipliers of the motion constraints.

This principle has been exploited by our community since the seminal work of Baraf [1], which is here our main source of inspiration. He initially proposed to formulate the

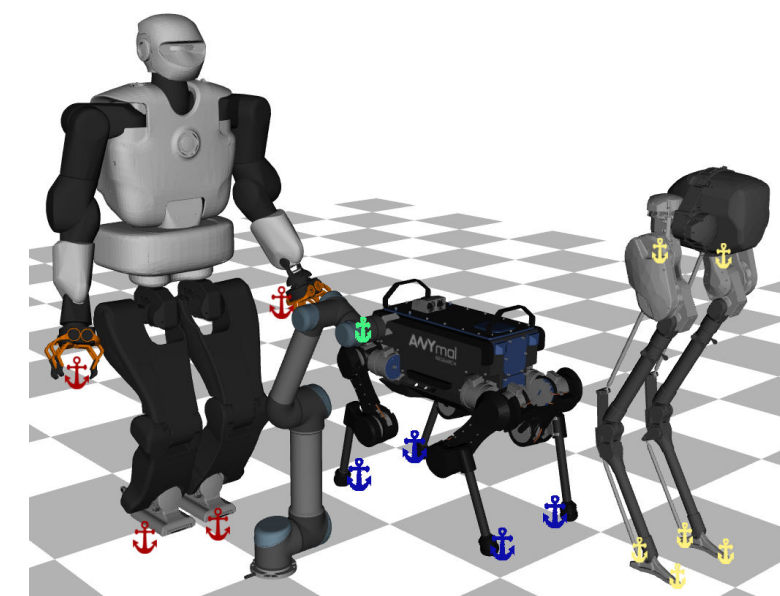
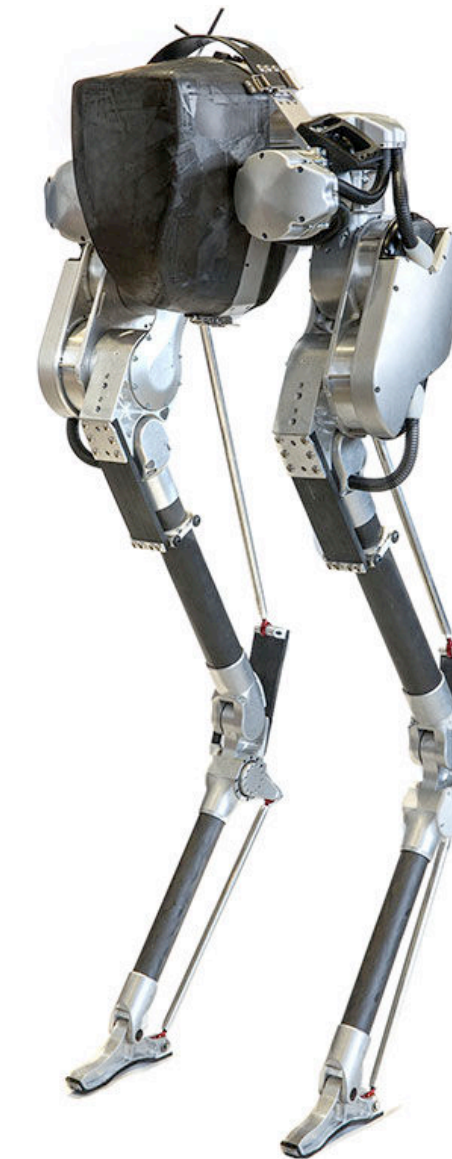


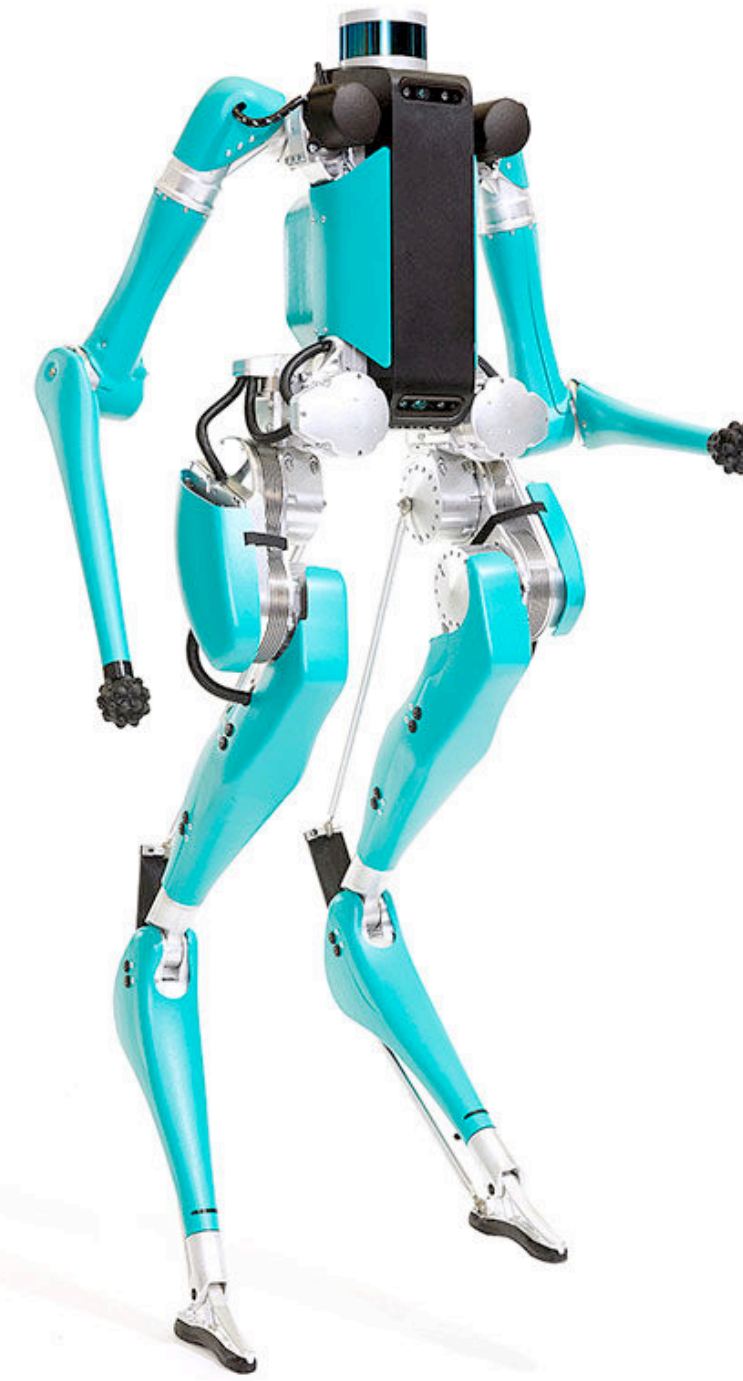
Fig. 1. Robotic systems may be subject to different types of constraints: point contact constraints (quadrupeds), flat foot constraints (humanoids), closed kinematic chains (parallel robots, here the 4-bar linkages of Cassie) or even contact with the end effectors (any robot). Each colored "anchor" here shows a possible kinematic constraint applied on the dynamics of the robot. In this paper, we introduce a generic approach to handle all these types of constraints, contacts and kinematic closures, in a unified and efficient manner, even in the context of ill-posed or singular cases.

dynamics with maximal coordinates (i.e. each rigid body is represented by its 6 coordinates of motion) as a sparse constrained optimization problem, and proposed an algorithm to solve it in linear time. While maximal coordinates are interesting for their versatility and largely used in simulation [2], working directly in the configuration space with generalized coordinates presents several advantages [16] that we propose to exploit in this paper.

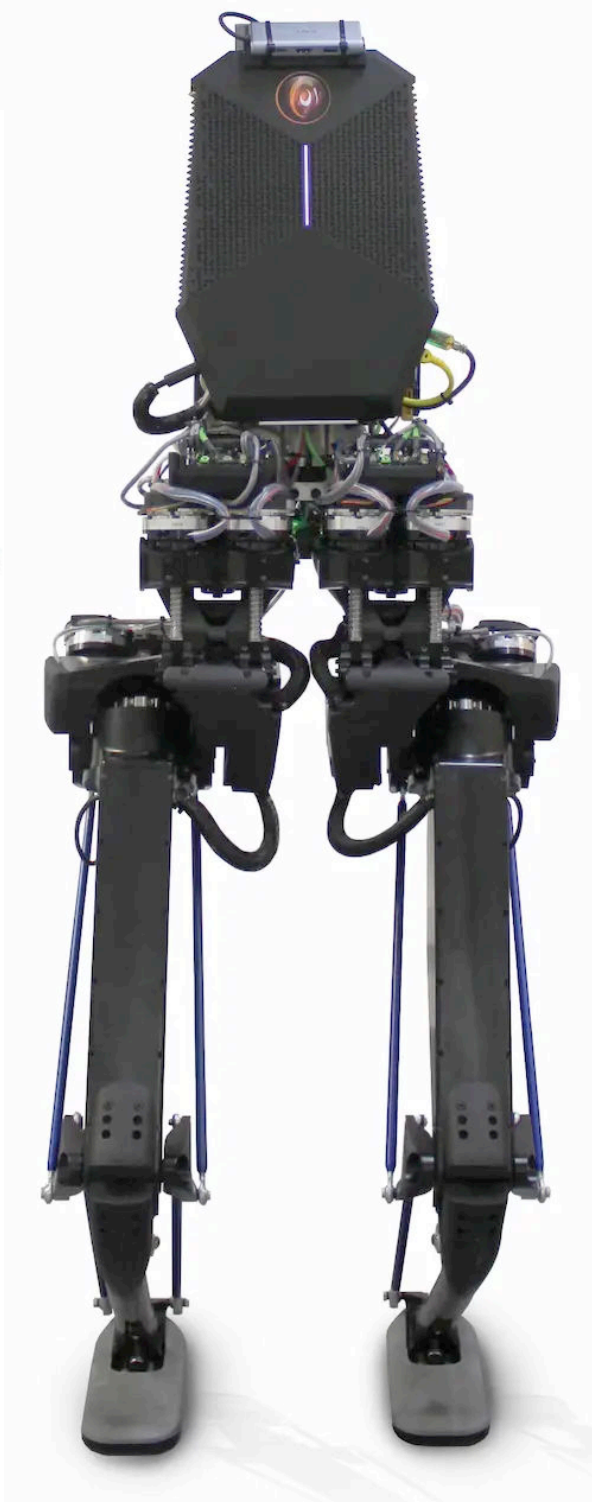
Some constraints can be put under an explicit form, i.e. there exists a reduced parametrization of the configuration that is free of constraints. This is often the case for classical kinematic closures [37, 16]. Yet explicit formulation is not always possible, and in particular is not possible for the common case of contact constraints [42]. We address here the more generic case where the constraints are written under an implicit form i.e. the configuration should nullify a set of equations, which makes it possible to handle any kind of



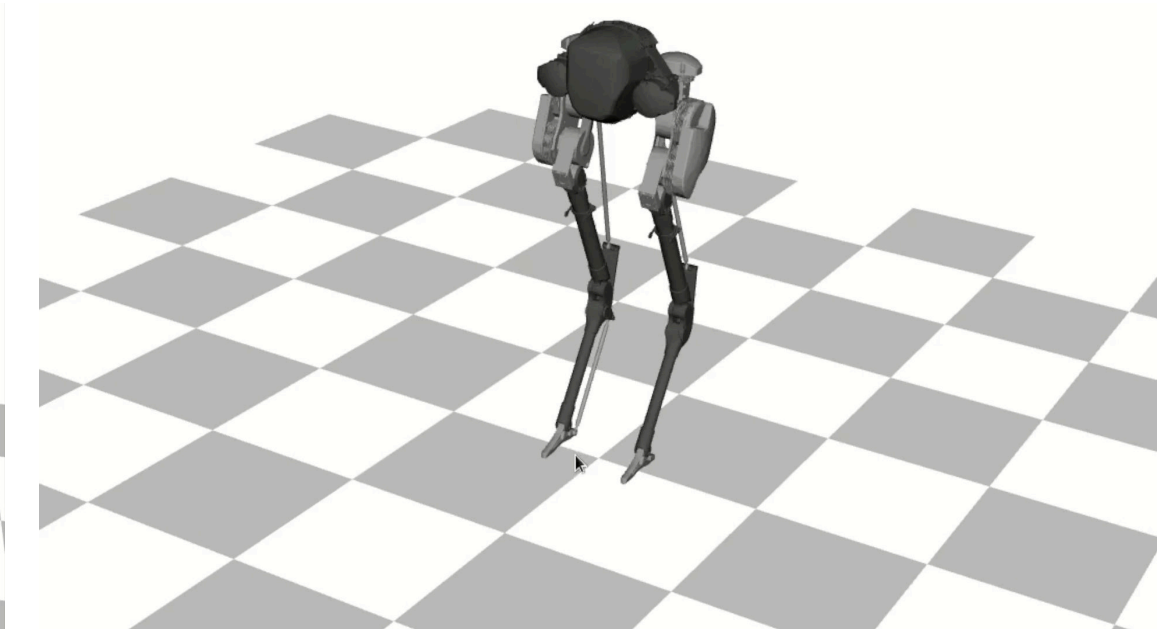
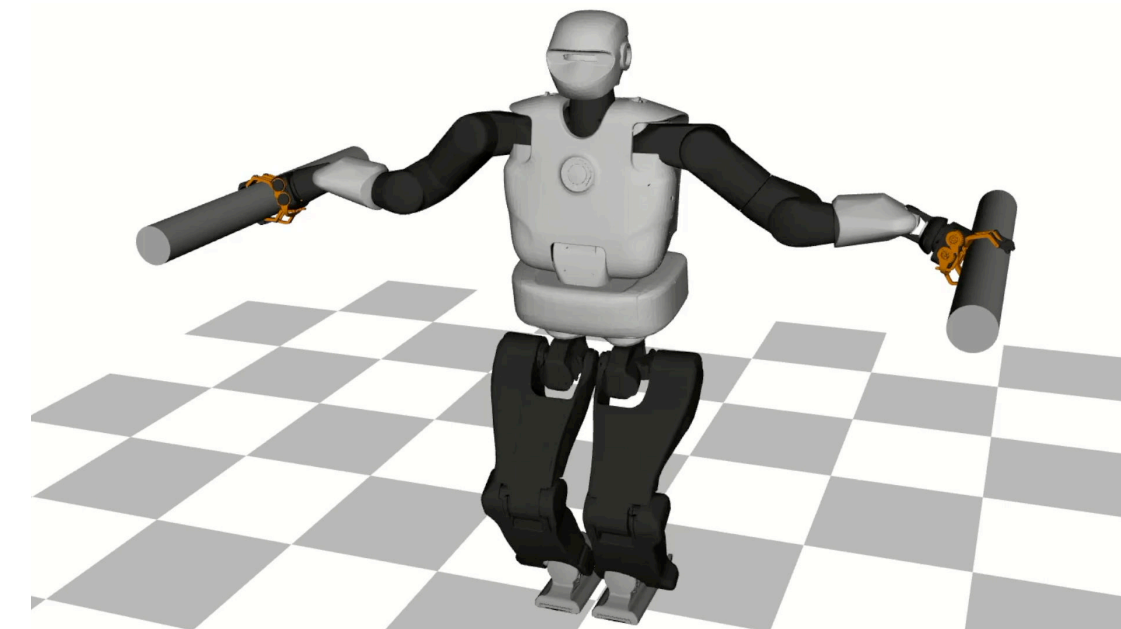
CASSIE



DIGIT



KANGAROO





# Closed-loop kinematics and bilateral constraints

## Proximal and Sparse Resolution of Constrained Dynamic Equations

Justin Carpentier  
Inria, École normale supérieure  
CNRS, PSL Research University  
75005 Paris, France  
Email: justin.carpentier@inria.fr

Rohan Budhiraja  
Inria Paris  
75012 Paris, France  
Email: rohan.budhiraja@inria.fr

Nicolas Mansard  
LAAS-CNRS, ANITI  
University of Toulouse  
31400 Toulouse, France  
Email: nicolas.mansard@laas.fr

**Abstract**—Control of robots with kinematic constraints like loop-closure constraints or interactions with the environment requires solving the underlying constrained dynamics equations of motion. Several approaches have been proposed so far in the literature to solve these constrained optimization problems, for instance by either taking advantage in part of the sparsity of the kinematic tree or by considering an explicit formulation of the constraints in the problem resolution. Yet, not all the constraints allow an explicit formulation and in general, approaches of the state of the art suffer from singularity issues, especially in the context of redundant or singular constraints. In this paper, we propose a unified approach to solve forward dynamics equations involving constraints in an efficient, generic and robust manner. To this aim, we first (i) propose a proximal formulation of the constrained dynamics which converges to an optimal solution in the least-square sense even in the presence of singularities. Based on this proximal formulation, we introduce (ii) a sparse Cholesky factorization of the underlying Karush–Kuhn–Tucker matrix related to the constrained dynamics, which exploits at best the sparsity of the kinematic structure of the robot. We also show (iii) that it is possible to extract from this factorization the Cholesky decomposition associated to the so-called **Operational Space Inertia Matrix**, inherent to task-based control frameworks or physic simulations. These new formulation and factorization, implemented within the Pinocchio library, are benchmark on various robotic platforms, ranging from classic robotic arms or quadrupeds to humanoid robots with closed kinematic chains, and show how they significantly outperform alternative solutions of the state of the art by a factor 2 or more.

### I. INTRODUCTION

As soon as a robot makes contacts with the world or is endowed with loop closures in its design, its dynamics is governed by the constrained equations of motion. From a phenomenological point of view, these equations of motion follow the so-called least-action principle, also known under the name of the Maupertuis principle which dates back to the 17<sup>th</sup> century. This principle states that the motion of the system follows the closest possible acceleration to the free-falling acceleration (in the sense of the kinetic metric) which fulfils the constraints. In other words, solving the constrained equations of motion boils down to solving a constrained optimization problem where forces acts as the Lagrange multipliers of the motion constraints.

This principle has been exploited by our community since the seminal work of Baraf [1], which is here our main source of inspiration. He initially proposed to formulate the

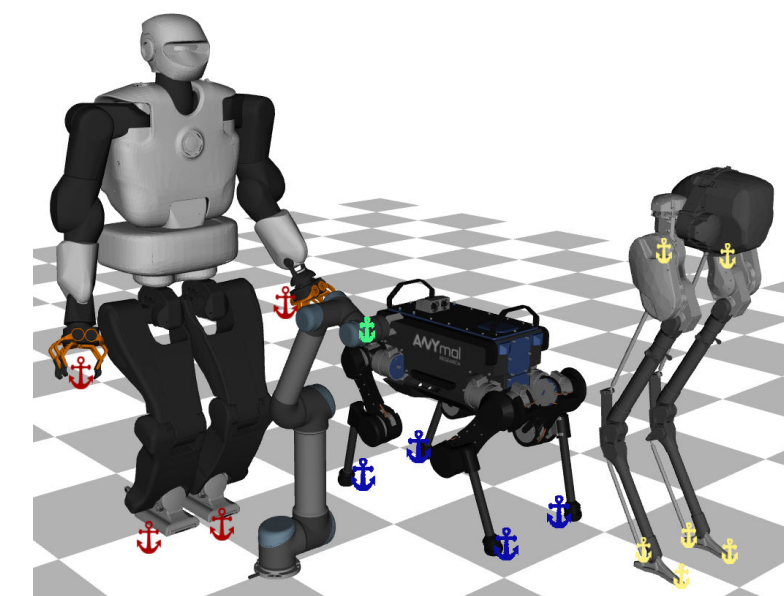
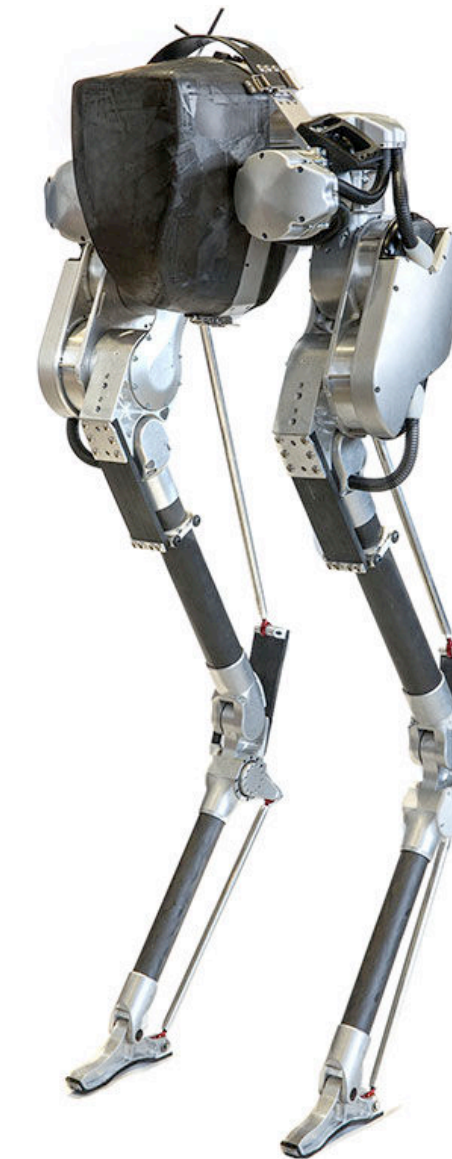


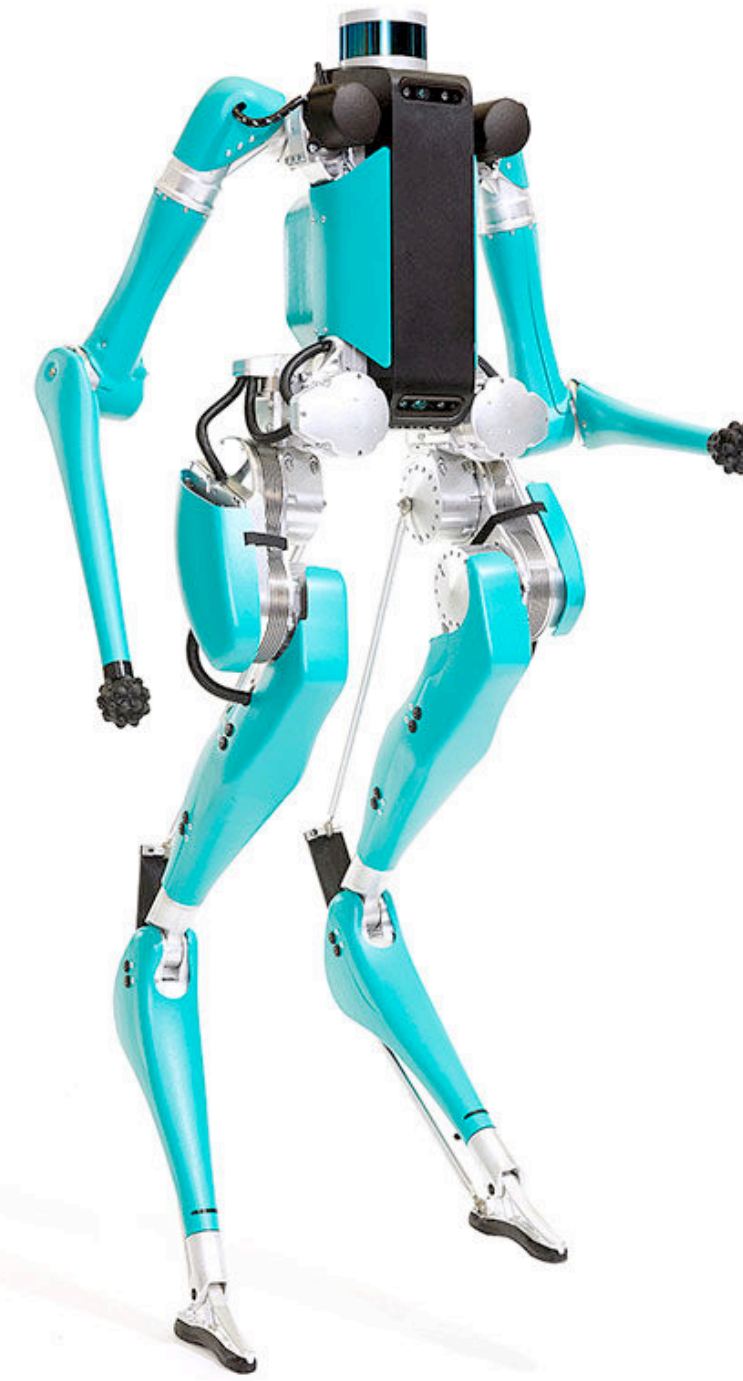
Fig. 1. Robotic systems may be subject to different types of constraints: point contact constraints (quadrupeds), flat foot constraints (humanoids), closed kinematic chains (parallel robots, here the 4-bar linkages of Cassie) or even contact with the end effectors (any robot). Each colored "anchor" here shows a possible kinematic constraint applied on the dynamics of the robot. In this paper, we introduce a generic approach to handle all these types of constraints, contacts and kinematic closures, in a unified and efficient manner, even in the context of ill-posed or singular cases.

dynamics with maximal coordinates (i.e. each rigid body is represented by its 6 coordinates of motion) as a sparse constrained optimization problem, and proposed an algorithm to solve it in linear time. While maximal coordinates are interesting for their versatility and largely used in simulation [2], working directly in the configuration space with generalized coordinates presents several advantages [16] that we propose to exploit in this paper.

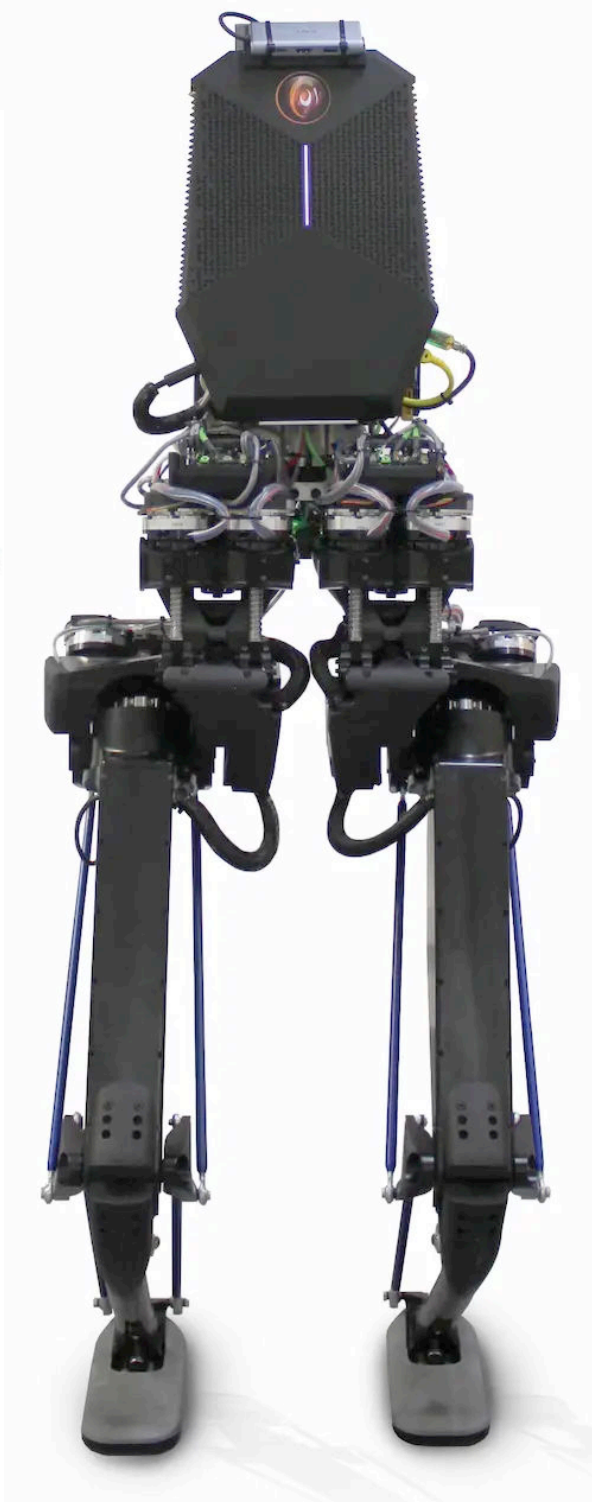
Some constraints can be put under an explicit form, i.e. there exists a reduced parametrization of the configuration that is free of constraints. This is often the case for classical kinematic closures [37, 16]. Yet explicit formulation is not always possible, and in particular is not possible for the common case of contact constraints [42]. We address here the more generic case where the constraints are written under an implicit form i.e. the configuration should nullify a set of equations, which makes it possible to handle any kind of



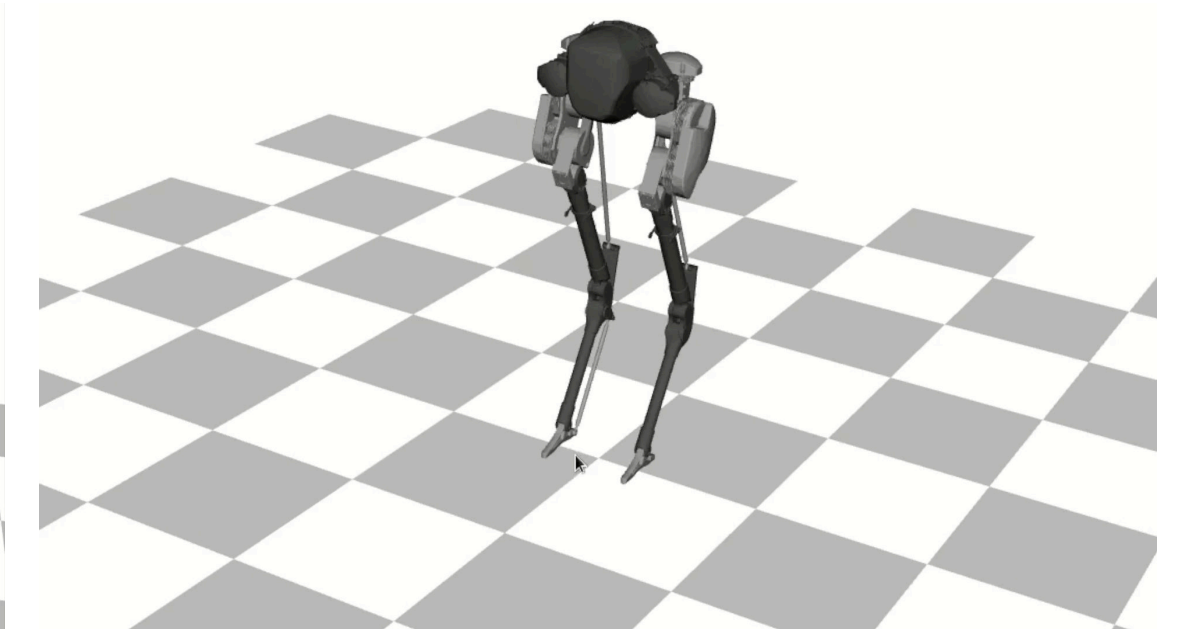
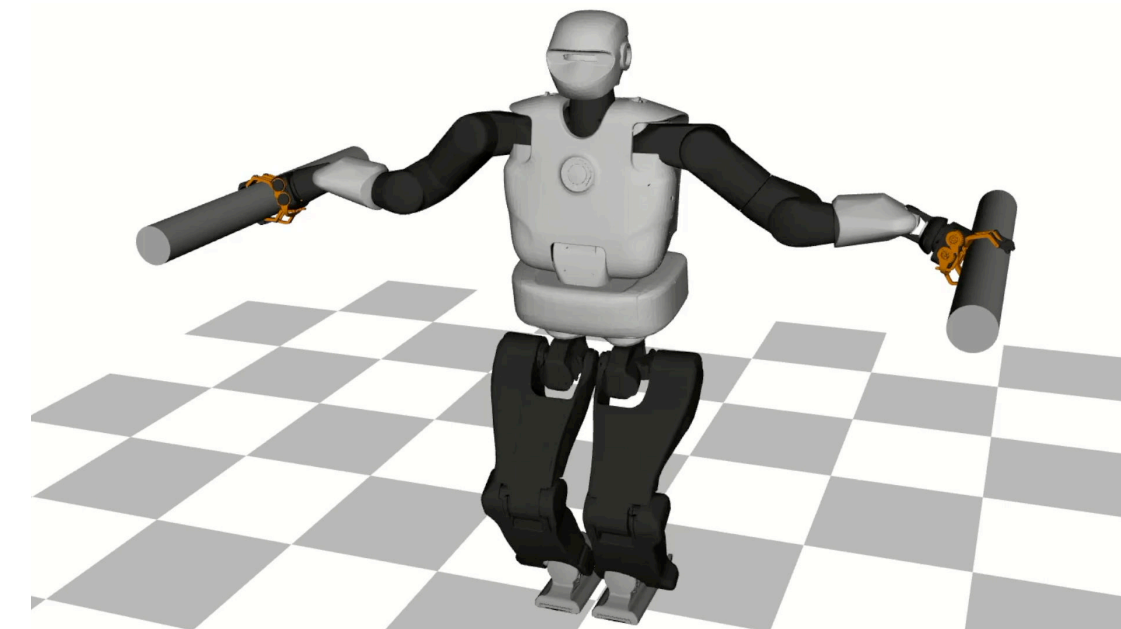
CASSIE



DIGIT



KANGAROO





# The central paradigm

The key aspect is the explicit splitting between **model** and **data**:

```
algorithm<Scalar> (model, data, arg1, arg2, ...)
```

**full**                  **constant**                  **cached**  
                  **templization**                  **quantity**                  **variables**



# The central paradigm

The key aspect is the explicit splitting between **model** and **data**:

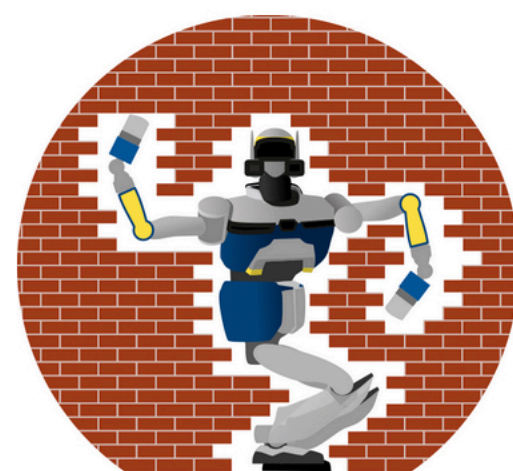
```
algorithm<Scalar> (model, data, arg1, arg2, ...)
```

**full**                  **constant**                  **cached**  
                  **templization**                  **quantity**                  **variables**

## Main advantages

- ▶ the compiler guesses what is constant, what varies
- ▶ no online memory allocation
- ▶ good prediction/anticipation of the CPU
- ▶ algorithms are easier to write
- ▶ and much more ...





# Pinocchio

Efficient and versatile rigid body dynamics algorithms

License **BSD 2-Clause** docs **online** coverage **93.00%** downloads **568k** conda-forge **v2.9.1** pypi package **2.6.19**

**Pinocchio** instantiates the state-of-the-art Rigid Body Algorithms for poly-articulated systems based on revisited Roy Featherstone's algorithms. Besides, **Pinocchio** provides the analytical derivatives of the main Rigid-Body Algorithms like the Recursive Newton-Euler Algorithm or the Articulated-Body Algorithm.

**Pinocchio** is first tailored for robotics applications, but it can be used in extra contexts (biomechanics, computer graphics, vision, etc.). It is built upon Eigen for linear algebra and FCL for collision detection. **Pinocchio** comes with a Python interface for fast code prototyping, **directly accessible** through **Conda**.

**Pinocchio** is now at the heart of various robotics software as **Crocodyl**, an open-source and efficient Differential Dynamic Programming solver for robotics, the **Stack-of-Tasks**, an open-source and versatile hierarchical controller framework or the **Humanoid Path Planner**, an open-source software for Motion and Manipulation Planning.

If you want to learn more on **Pinocchio** internal benefits and main features, we invite you to read the related [paper](#) and the online [documentation](#).

If you want to directly dive into **Pinocchio**, only one single line is sufficient (assuming you have Conda):

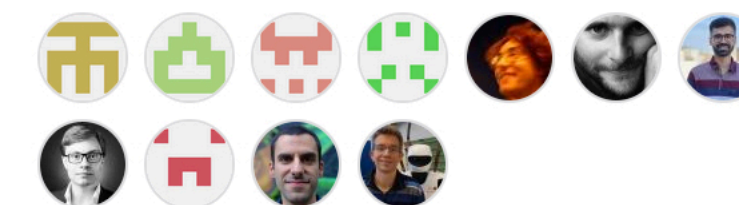
```
conda install pinocchio -c conda-forge
```

or via pip (currently only available on Linux):

```
pip install pin
```

## Table of contents

- [Pinocchio main features](#)
- [Documentation](#)
- [Examples](#)
- [Tutorials](#)
- [Performances](#)



+ 39 contributors

### Environments 1

**github-pages** Active

### Languages



<https://github.com/stack-of-tasks/pinocchio/>



# Installing Pinocchio



[github.com/stack-of-tasks/pinocchio](https://github.com/stack-of-tasks/pinocchio)



# Installing Pinocchio



[github.com/stack-of-tasks/pinocchio](https://github.com/stack-of-tasks/pinocchio)



```
conda install pinocchio -c conda-forge
```



# Installing Pinocchio



[github.com/stack-of-tasks/pinocchio](https://github.com/stack-of-tasks/pinocchio)



```
conda install pinocchio -c agm-ws-2023
```



# Pinocchio Cheat Sheet

## Get started

easy install	conda install -c conda-forge pinocchio
import	import pinocchio as pin
	from pinocchio.utils import *
<b>documentation</b>	pin.Model?

## Spatial quantities

Transforms	
SE3	aMb = pin.SE3(aRb, apb)
unit transformation	M = pin.SE3(1) or pin.SE3.Identity()
random transformation	pin.SE3.Random()
rotation matrix	M.rotation
translation vector	M.translation
SE3 inverse	bMa = aMb.inverse()
SE3 action	aMc = aMb * bMc
action matrix	aXb = aMb.action
homogeneous matrix	aHb = aMb.homogeneous
log operation SE3 → 6D	pin.log(M)
exp operation	pin.exp(M)
Spatial Velocity	
Motion	m = pin.Motion(v, w)
linear acceleration	m.linear
angular acceleration	m.angular
SE3 action	v.a = aMb * v.b
Spatial Acceleration	
used in algorithms	a = ( $\dot{\omega}$ , $\dot{v}_O$ )
get classical acceleration	a' = a + ( $0, \omega \times v_O$ )
	pin.classicAcceleration(v, a, [aMb])
Spatial Force	
Force	f = pin.Force(l, n)
linear force	f.linear
torque	f.angular
SE3 action	f.a = aMb * f.b
Spatial Inertia	
Inertia	Y = pin.Inertia(mass, com, I)
mass	Y.mass
center of mass pos.	Y.lever
rotational inertia	Y.inertia
Geometry	
Quaternion	quat = pin.Quaternion(R)
Angle Axis	aa = pin.AngleAxis(angle, axis)
Useful converters	
SE3 → (x,y,z,quat)	pin.se3ToXYZQUAT(M)
(x,y,z,quat) → SE3	pin.XYZQUATToSE3(vec)

## Data

Data related to the model	data = pin.Data(model)
	data = model.createData()
joint data	data.joints
joint/[frame] placements	data.oMi / [data.oMf]
joint velocities	data.v
joint accelerations	data.a
joint forces	data.f
mass matrix	data.M
non linear effects	data.nle
centroidal momentum	data.hg
centroidal matrix	data.Ag
centroidal inertia	data.Ig

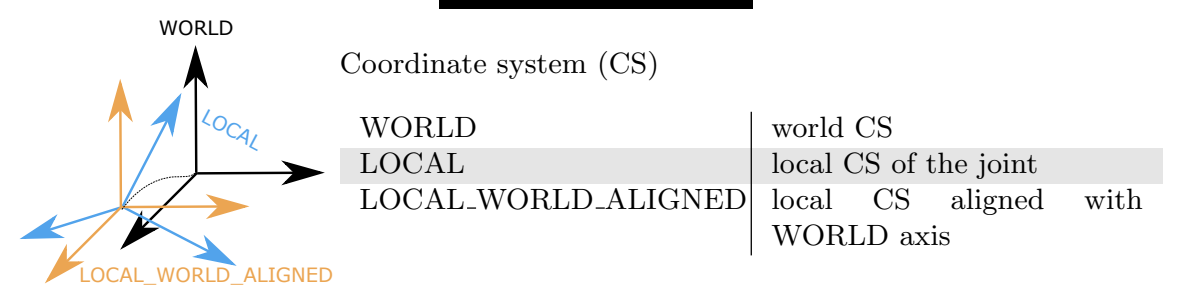
## Model

Model of the kinematic tree	model = pin.Model()
model name	model.name
joint names	model.names
joint models	model.joints
joint placements	model.placements
link inertias	model.inertias
frames	model.frames
# position variables	model.nq
# velocity variables	model.nv
Methods	use ? to get doc and input arguments
add joint	model.addJoint
append body	model.appendBodyToJoint
add frame	model.addFrame
append child into parent model	model.appendModel
build reduced body	model.buildReducedModel

## Parsers

load an URDF file	pin.buildModelFromUrdf(filename, [root_joint])
load a SDF file	pin.buildModelFromSdf(filename, [root_joint], root_link_name, parent_guidance)

## Reference Frames



## Frames

placement of all operational frames	pin.updateFramePlacements(model, data)
current frame placements wrt origin	data.oMf
frame velocity	pin.getFrameVelocity(model, data, frame_id, ref_frame)
frame acceleration	pin.getFrameAcceleration(model, data, frame_id, ref_frame)
frame acceleration	pin.getFrameClassicalAcceleration(model, data, frame_id, ref_frame)
frames placement	pin.framesForwardKinematics(model, data, q)
frame jacobian	pin.computeFrameJacobian(model, data, q, frame_id, ref_frame)
frame jacobian time variation	pin.frameJacobianTimeVariation(model, data, q, v, frame_id, ref_frame)
partial derivatives of the spatial velocity	pin.getFrameVelocityDerivatives(model, data, frame_id, ref_frame)
partial derivatives of the spatial velocity	pin.getFrameVelocityDerivatives(model, data, joint_id, placement ref_frame)
partial derivatives of the spatial acceleration	pin.getFrameVelocityDerivatives(model, data, frame_id, ref_frame)
partial derivatives of the spatial acceleration	pin.getFrameAccelerationDerivatives(model, data, joint_id, placement ref_frame)

## Configuration

random configuration	pin.randomConfiguration(model, [lower_bound, upper_bound])
neutral configuration	pin.neutral(model)
normalized configuration	pin.normalize(model, q)
difference configurations	pin.difference(model, q1, q2)
distance configurations	pin.distance(model, q1, q2)
squared distance configurations	pin.squareDistance(model, q1, q2)
interpolate configuration	pin.interpolate(model, q1, q2, alpha)
integrate configuration	pin.integrate(model, q, v)
partial derivatives of difference	pin.dDifference(model, q1, q2, [arg_pos])
partial derivatives of integration	pin.dIntegrate(model, q, v, [arg_pos])

## Collision

placement collision obj	pin.updateGeometryPlacements(model, data, geometry_model, geometry_data, [q])
collisions detection for all pairs	pin.computeCollisions(model, data, geometry_model, geometry_data, q)
collisions detection for a pair	pin.computeCollisions(geometry_model, geometry_data, pair_index)
distance from collision	pin.computeDistance(geometry_model, geometry_data, [pair_index])
distance from collision each pair	pin.computeDistances([model, data], geometry_model, geometry_data, [q])
geometry volume radius	pin.computeBodyRadius(model, geometry_model, geometry_data)
BroadPhase	pin.computeCollisions(broadphase_manager, callback)
	pin.computeCollisions(broadphase_manager, stop_at_first_collision)
+ forward kinematics to update geometry placements	pin.computeCollisions(model, data, broadphase_manager, q, stop_at_first_collision)

## Center of Mass

total mass of model	pin.computeTotalMass(model, [data])
mass of each subtree	pin.computeSubtreeMasses(model, data)
center of mass (COM)	pin.centerOfMass(model, data, q, [v, a], [compute_subtree_com])
Jacobian COM	pin.jacobianCenterOfMass(model, data, [q], [compute_subtree_com])

## Energy

FK and kinetic Energy	pin.computeKineticEnergy(model, data, [q, v])
FK and potential Energy	pin.computePotentialEnergy(model, data, [q, v])
FK and mechanical Energy	pin.computeMechanicalEnergy(model, data, [q, v])



**Kinematics**

forward kinematics (FK)	<code>pin.forwardKinematics(model, data, q, [v,[a]])</code>
FK derivatives	<code>pin.computeForwardKinematicsDerivatives(model, data, q, v, a)</code>
$\begin{bmatrix} \frac{\partial v}{\partial q} & \frac{\partial v}{\partial \dot{q}} \end{bmatrix}^{WORLD}$	<code>pin.getJointVelocityDerivatives(model, data, joint_id, pin.ReferenceFrame.WORLD)</code>
$\begin{bmatrix} \frac{\partial v}{\partial q} & \frac{\partial a}{\partial \dot{q}} & \frac{\partial a}{\partial \ddot{q}} \end{bmatrix}^{LOCAL}$	<code>pin.getJointAccelerationDerivatives(model, data, joint_id, pin.ReferenceFrame.LOCAL)</code>

**Jacobian**

full model Jacobian → data.J	<code>pin.computeJointJacobians(model, data, [q])</code>
joint Jacobian	<code>pin.getJointJacobian(model, data, joint_id, ref_frame)</code>
full model dJ/dt	<code>pin.computeJointJacobiansTimeVariation(model, data, q, v)</code>
joint dJ/dt	<code>pin.getJointJacobianTimeVariation(model, data, joint_id, ref_frame)</code>

**Forward Dynamics**

Articulated-Body Algorithm $\dot{q}$	<code>pin.aba(model, data, q, v, tau, [f_ext])</code>
Joint Space Inertia Matrix Inv	<code>pin.computeMinverse(model, data, [q])</code>
Composite Rigid-Body Algorithm	<code>pin.crba(model, data, q)</code>

**Inverse Dynamics**

Recursive Newton-Euler Algorithm	<code>pin.rnea(model, data, q, v, a, [f_ext])</code>
generalized gravity	<code>pin.computeGeneralizedGravity(model, data, q)</code>
$\dot{\tau}$ , $\dot{d}\tau$ , $\dot{d}a$	<code>pin.computeRNEADerivatives(model, data, q, v, a, [f_ext])</code>

**Centroidal**

Centroidal momentum	<code>pin.computeCentroidalMomentum(model, data, [q, v])</code>
Centroidal momentum + time derivatives	<code>pin.computeCentroidalMomentumTimeVariation(model, data, [q, v, a])</code>

**General**

all terms (check doc)	<code>pin.computeAllTerms(model, data, q, v)</code>
-----------------------	---

**Kinematic Regressor**

kinematic regressor	<code>pin.computeJointKinematicRegressor(model, data, joint_id, ref_frame, [placement])</code>
kinematic regressor	<code>pin.computeFrameKinematicRegressor(model, data, frame_id, ref_frame)</code>

**Regressor**

static regressor	<code>pin.computeStaticRegressor(model, data, q)</code>
body regressor	<code>pin.bodyRegressor(velocity, acceleration)</code>
body attached to joint regressor	<code>pin.jointBodyRegressor(model, data, joint_id)</code>
body attached to frame regressor	<code>pin.frameBodyRegressor(model, data, frame_id)</code>
joint torque regressor	<code>pin.computeJointTorqueRegressor(model, data, q, v, a)</code>

**Contact Jacobian**

kinematic Jacobian of constraint model	<code>pin.getConstraintJacobian(model, data, contact_model, contact_data)</code>
kinematic Jacobian of set of constraint models	<code>pin.getConstraintJacobian(model, data, contact_models, contact_datas)</code>

**Contact Dynamics**

constrained dynamics with contacts	<code>pin.forwardDynamics(model, data, [q, v,] tau, constraint_jacobian, constraint_drift, damping)</code>
impact dynamics with contacts	<code>pin.impulseDynamics(model, data, [q,] v.before, constraint_jacobian, restitution_coefficient, damping)</code>
inverse of the constraint matrix	<code>pin.computeKKTContactDynamicMatrixInverse(model, data, q, constraint_jac, damping)</code>

**Constraint Dynamics**

allocate memory	<code>pin.initConstraintDynamics(model, data, contact_models)</code>
forward dynamics with contact constraints	<code>pin.constraintDynamics(model, data, q, v, tau, contact_models, contact_datas, [prox_settings])</code>
derivatives of the forward dynamics with kinematic constraints	<code>pin.computeConstraintDynamicsDerivatives(model, data, contact_models, contact_datas, prox_settings)</code>

**Impulse Dynamics**

impulse dynamics with contact constraints	<code>pin.impulseDynamics(model, data, q, v, contact_models, contact_datas, r_coeff, mu)</code>
impulse dynamics derivatives	<code>pin.computeImpulseDynamicsDerivatives(model, data, contact_models, contact_datas, r_coeff, prox_settings)</code>

**Cholesky**

Cholesky decomposition of the joint space inertia matrix	<code>pin.cholesky.decompose(model, data)</code>
$x$ of $Mx = y$	<code>pin.cholesky.solve(model, data, v)</code>
inverse of the joint space inertia matrix	<code>pin.cholesky.computeMinv(model, data)</code>

**Viewer**

<b>Get started</b>	
create viewer	<code>mv = pin.visualize.MeshcatVisualizer</code>
load model	<code>viz = mv(model, collision_model, visual_model)</code>
initialize	<code>viz.initViewer(loadModel=True)</code>
display	<code>viz.display(q)</code>
<b>Add basic shapes</b>	
sphere	<code>viz.viewer[name].set_object(meshcat.geometry.Sphere(size), material)</code>
box	<code>viz.viewer[name].set_object(meshcat.geometry.Box([sizex, sizey, sizez]), material)</code>
<b>Display</b>	
change placement of geometry [name]	<code>viz.viewer[name].set_transform(meshcat_transform(xyzquat_placement))</code>



# Citing Pinocchio

```
@inproceedings{carpentier2019pinocchio,  
  title={The Pinocchio C++ library -- A fast and flexible implementation of rigid body dynamics algorithms},  
  author={Carpentier, Justin and Saurel, Guilhem and Buondonno, Gabriele and Mirabel, Joseph and Lemerle, Vincent},  
  booktitle={IEEE International Symposium on System Integrations (SII)},  
  year={2019}  
}
```

and the following one for the link to the GitHub codebase:

```
@misc{pinocchioweb,  
  author = {Justin Carpentier and Florian Valenza and Nicolas Mansard and others},  
  title = {Pinocchio: fast forward and inverse dynamics for poly-articulated systems},  
  howpublished = {https://stack-of-tasks.github.io/pinocchio},  
  year = {2015--2021}  
}
```

The algorithms for the analytical derivatives of rigid-body dynamics algorithms are detailed here:

```
@inproceedings{carpentier2018analytical,  
  title = {Analytical Derivatives of Rigid Body Dynamics Algorithms},  
  author = {Carpentier, Justin and Mansard, Nicolas},  
  booktitle = {Robotics: Science and Systems},  
  year = {2018}  
}
```

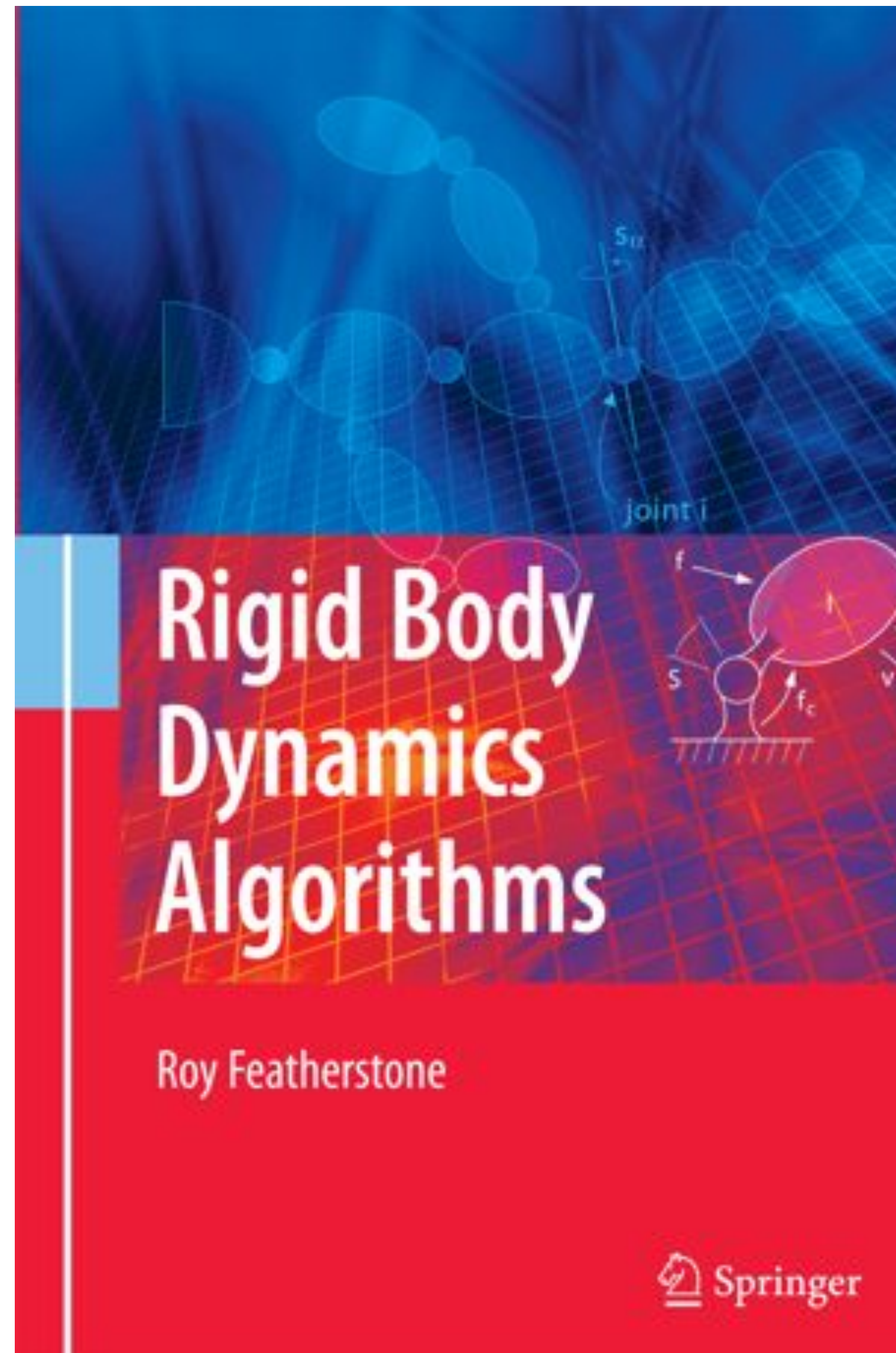


# Spatial quantities

The basic principles

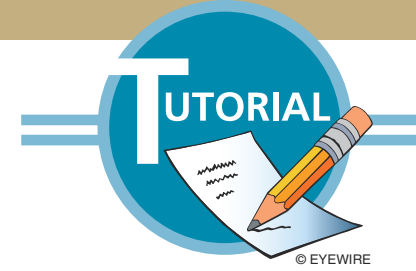


# The “Bible”





# A short intro to the “Bible”



## A Beginner's Guide to 6-D Vectors (Part 1)

What They Are, How They Work, and How to Use Them

BY ROY FEATHERSTONE

A rigid body has six degrees of motion freedom, so why not use six-dimensional (6-D) vectors to describe its motions and the forces acting upon it? In fact, some roboticists already do this, and the practice is becoming more common. The purpose of this tutorial is to present a beginner's guide to 6-D vectors in sufficient detail that a reader can begin to use them as a practical problem-solving tool right away. This tutorial covers the basics, and Part 2 will cover the application of 6-D vectors to a variety of robot kinematics and dynamics calculations.

6-D vectors come in various forms. The particular kind presented here is called *spatial vectors*. They are the tool that the author has been using for nearly 30 years to invent dynamics algorithms and write dynamics calculation software. Other kinds of 6-D vector include screws, motors, and Lie algebras. More will be said about them at the end of this tutorial. The differences between the various kinds of 6-D vector are relatively small. The more you understand any one of them, the easier it gets to understand the others.

The obvious advantage of 6-D vectors is that they cut the volume of algebra. Instead of having to define two three-dimensional (3-D) vectors to describe a force, another two to describe an acceleration, and writing two equations of motion for each body, a 6-D vector notation lets you pair up corresponding 3-D vectors and equations. The immediate result is a tidier, more compact notation involving fewer quantities and fewer equations. However, anyone who thinks that 6-D vectors

are only a convenient notation for organizing 3-D vectors is missing half the point. 6-D vectors are tools for thought. They have their own physical meanings and mathematical properties. They let you solve a problem more directly, and at a higher level of abstraction, by letting you think in 6-D, which is easier than it sounds.

Using spatial vectors (and other kinds of 6-D vector) lets you formulate a problem more succinctly, solve it more quickly and in fewer steps, present the solution more clearly to others, implement it in fewer lines of code, and debug the software more easily. Furthermore, there is no loss of efficiency: spatial-vector software can be just as efficient as 3-D-vector software, despite the higher level of abstraction.

The rest of this tutorial is chiefly concerned with explaining what spatial vectors are and how to use them. It highlights the differences between solving a rigid-body problem using 3-D vectors and solving the same problem using spatial vectors, so that the reader can get an idea of what it means to think in 6-D.

### A Note on Notation

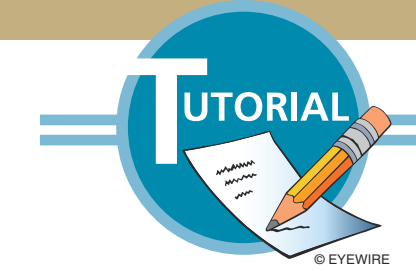
When using spatial vectors, it is convenient to employ symbols like  $f$ ,  $v$ , and  $a$  (or  $\dot{v}$ ) to denote quantities like force, velocity, and acceleration. However, these same symbols are equally useful for 3-D vectors. Thus, whenever spatial and 3-D vectors are discussed together, there is a possibility of name clashes. To resolve these clashes, we shall use the following rule: in any context where a spatial symbol needs to be distinguished from a 3-D symbol, the spatial symbol is given a hat (e.g.,  $\hat{f}$  and  $\hat{v}$ ). These hats are dropped when they are no longer needed. An

Digital Object Identifier 10.1109/MRA.2010.937853

SEPTEMBER 2010

1070-9932/10/\$26.00©2010 IEEE

IEEE Robotics & Automation Magazine 83



## A Beginner's Guide to 6-D Vectors (Part 2)

From Equations to Software

BY ROY FEATHERSTONE

Spatial vectors are six-dimensional (6-D) vectors that describe the motions of rigid bodies and the forces acting upon them. In Part 1, we saw how spatial vectors can simplify the process of expressing and analyzing the dynamics of a simple rigid-body system. In this tutorial, we shall examine the application of spatial vectors to various problems in robot kinematics and dynamics. To demonstrate that spatial vectors are both a tool for analysis and a tool for computation, we shall consider both the mathematical solution of a problem and the computer code to calculate the answer.

To illustrate the power of spatial vectors, we shall consider the class of robots having branched connectivity. This class includes legged robots, humanoids and multifingered grippers, as well as traditional serial robot arms; however, it does not include robots with kinematic loops, such as parallel robots. To cope with this degree of generality, we shall take a model-based approach: the robot mechanism is described by means of a standard set of quantities stored in a model data structure, and the equations, algorithms, and computer code are designed to use those quantities in their calculations.

Following the same pattern as Part 1, this tutorial starts with a specific example and proceeds to analyze it in detail; the example in this instance being the computer code to implement a model-based inverse dynamics calculation using the recursive Newton-Euler algorithm. Subsequent sections

then examine a variety of topics in kinematics and present the two main recursive algorithms for forward dynamics: the composite-rigid-body algorithm and the articulated-body algorithm.

It is assumed that the readers have already read Part 1 [6], or equivalent material, and therefore, they are familiar with the notation and basic concepts of spatial vector algebra.

### A Computational Example

Inverse dynamics is the problem of calculating the forces required to produce a given acceleration. It is a relatively easy problem, and therefore, a good place to start. A model-based inverse dynamics calculation can be expressed mathematically as

$$\tau = \text{ID}(\text{model}, q, \dot{q}, \ddot{q}), \quad (1)$$

where  $q$ ,  $\dot{q}$ ,  $\ddot{q}$ , and  $\tau$  denote vectors of joint position, velocity, acceleration, and force variables, respectively, and *model* denotes a data structure containing a description of the robot. The objective is to calculate the numeric value of ID given the numeric values of its arguments.

Figure 1 shows the MATLAB source code for an implementation of (1) using the recursive Newton-Euler algorithm. This is a complete implementation: you could type it in right now (minus the line numbers) and get it to work, provided you also typed in the (very short) definitions of the functions `jointCalc`, `crf`, and `crf`, which are discussed later in this tutorial. The code in Figure 1 can calculate the inverse dynamics of

Digital Object Identifier 10.1109/MRA.2010.939560

IEEE Xplore. Downloaded on December 10, 2025 at 15:59:29 UTC from IEEE Xplore. Restrictions apply.



# What is a **spatial vector**?

## **Euclidian** vector (3-D)

**Euclidean vector** provides a complete description of the state of motion of a particle

Motion equation of a particle

$$\begin{array}{ccc} & \mathbf{f} = \frac{\mathbf{d}}{\mathbf{dt}} \mathbf{h} & \\ \nearrow & & \nwarrow \\ \text{force} & & \text{linear momentum} \end{array}$$

## **Spatial** vector (6-D)

**Spatial vector** provides a complete description of the state of motion of a rigid body

Motion equation of a rigid body

$$\begin{array}{ccc} & \hat{\mathbf{f}} = \frac{\mathbf{d}}{\mathbf{dt}} \hat{\mathbf{h}} & \\ \nearrow & & \nwarrow \\ \text{spatial force} & & \text{spatial momentum} \end{array}$$

Spatial vectors combine **the linear and angular aspects** of rigid-body motion or force into a **single vectorial quantity**



# *Why are* **spatial vectors** so useful?

Spatial vectors are **concise notations** for describing, analyzing, and evaluating the **kinematics and dynamics quantities** of rigid bodies and poly-articulated systems:

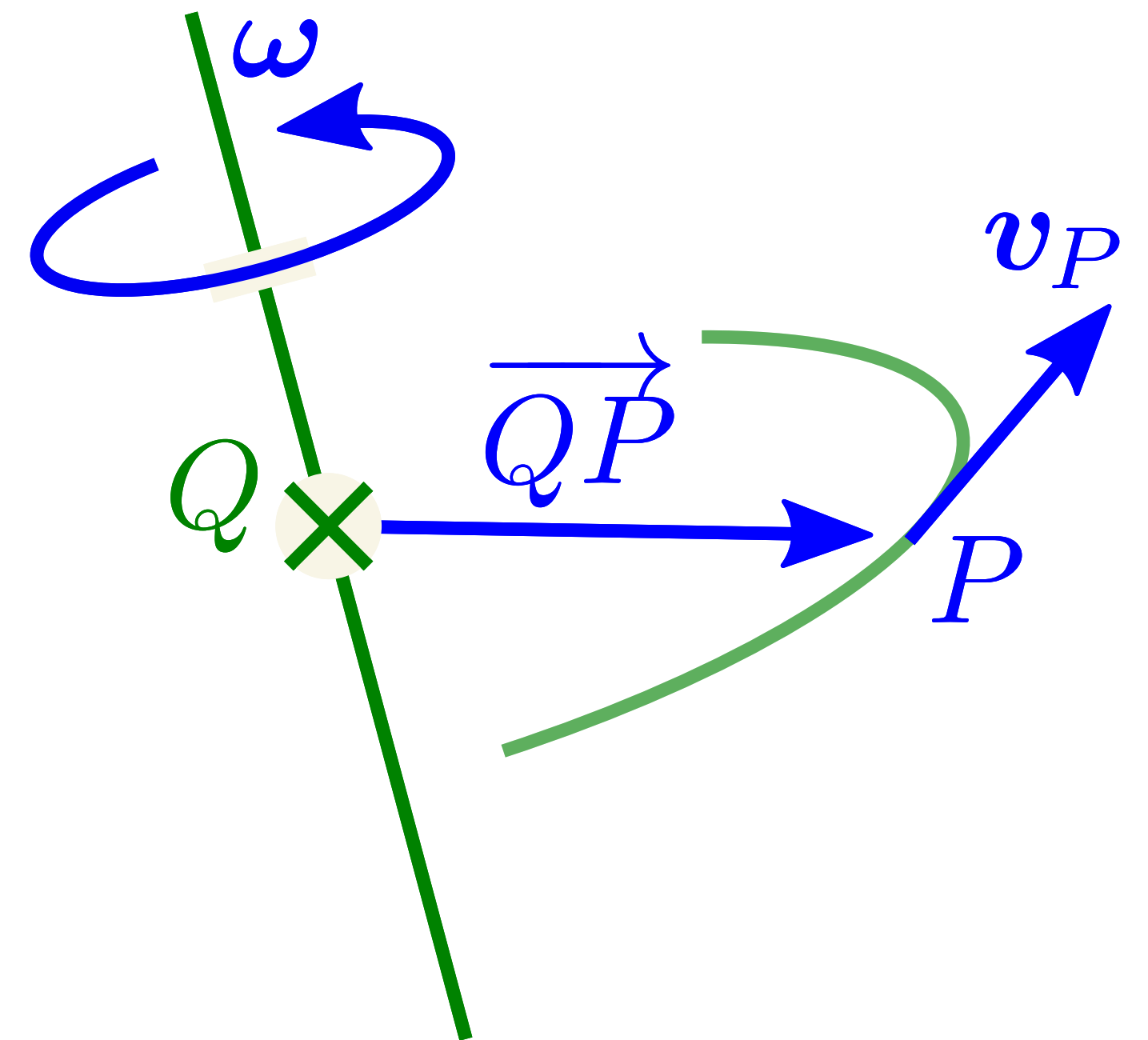
- ▶ compact representation, easy to learn
- ▶ fewer equations, thus fewer (programming) mistakes
- ▶ less (programming) effort
- ▶ really suited for modern CPU architectures (e.g., SIMD instructions)



# Spatial velocity: the angular part

The velocity **vector field**  $V_Q$  associated with the **angular velocity**  $\omega$  and the **point**  $Q$  is given by:

$$V_Q(P) = \omega \times \overrightarrow{QP}$$



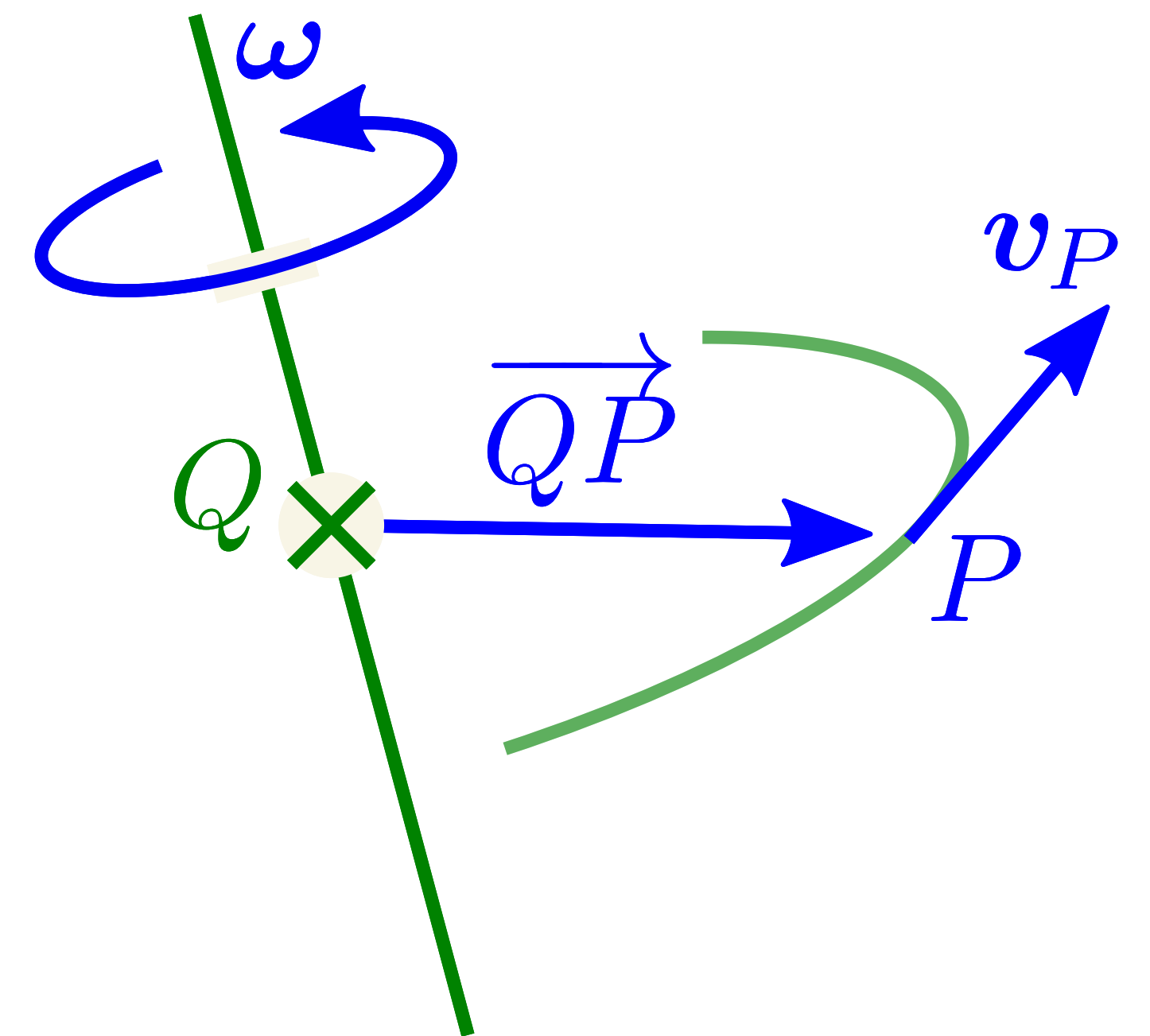
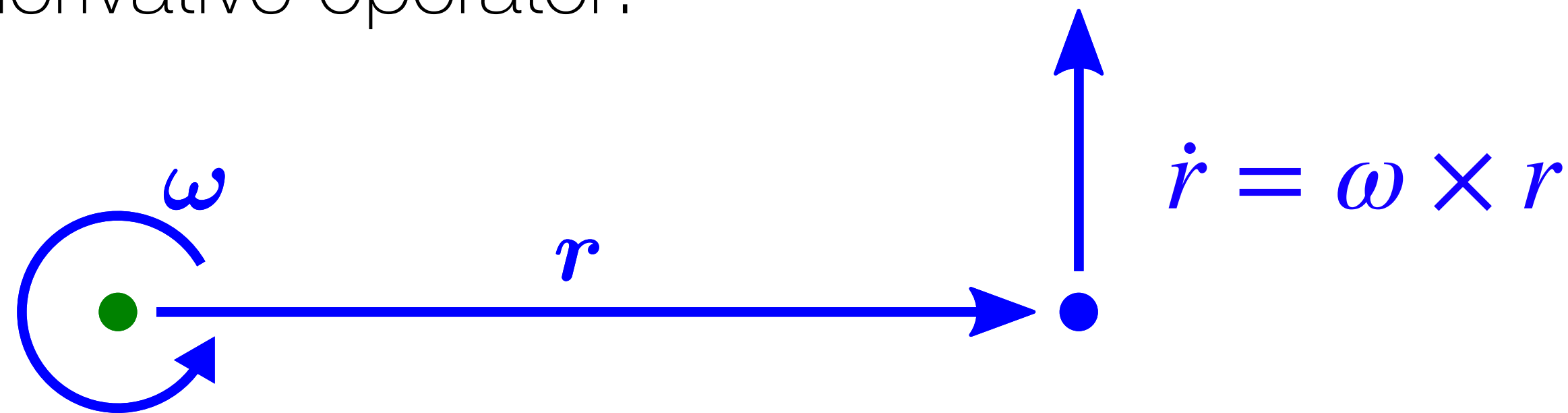


# Spatial velocity: the angular part

The velocity **vector field**  $V_Q$  associated with the **angular velocity**  $\omega$  and **the point**  $Q$  is given by:

$$V_Q(P) = \omega \times \overrightarrow{QP}$$

Note that the cross product plays the role of time derivative operator:





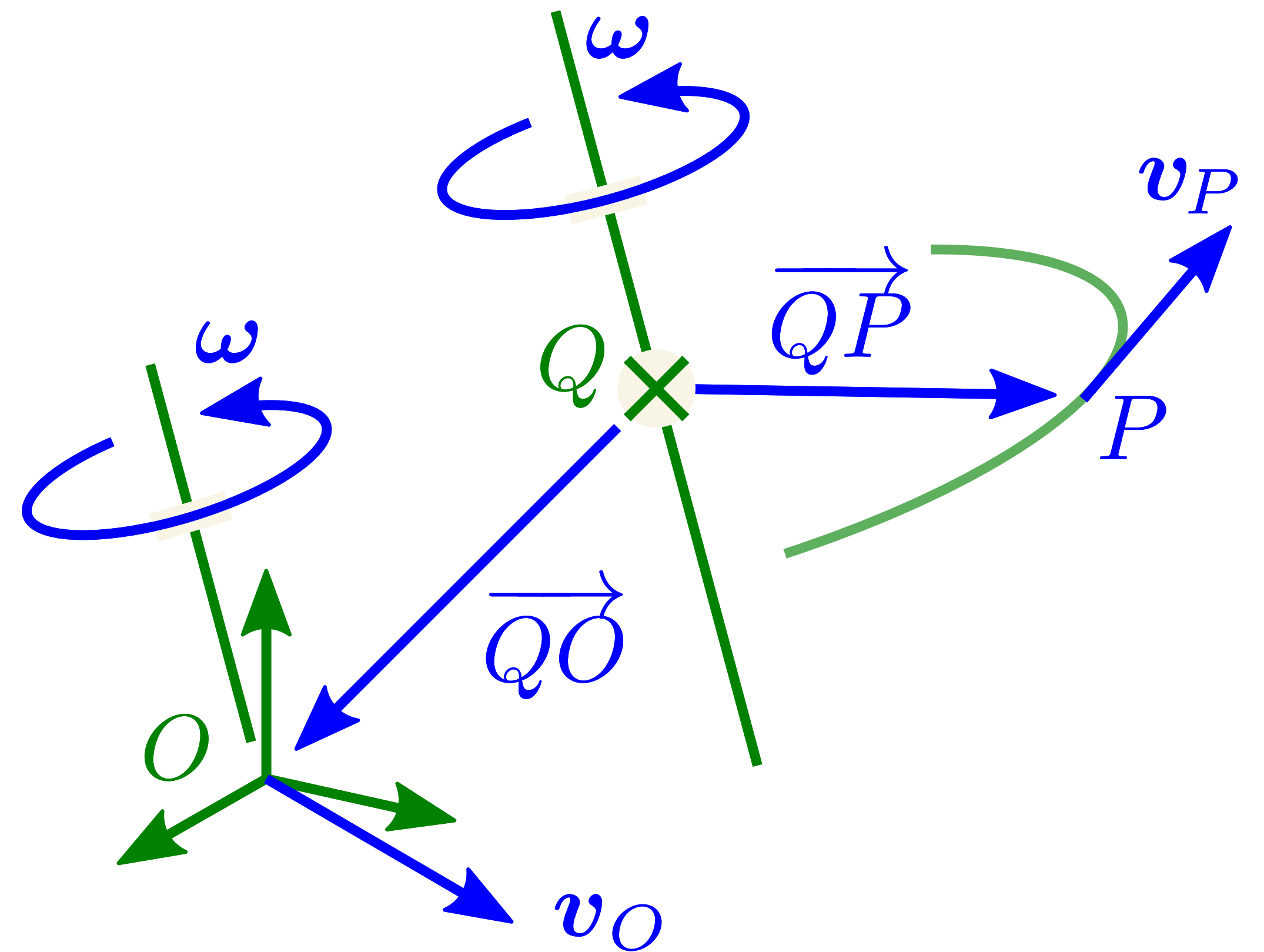
# Spatial velocity

We can also get the **linear velocity** associated with a frame centered in  $O$

$$V_Q(P) = V_O(P) + \omega \times \overrightarrow{QO}$$

leading to the spatial vector:

$$v_O = \begin{bmatrix} v_O \\ \omega \end{bmatrix} = \begin{bmatrix} v_{Ox} \\ v_{Oy} \\ v_{Oz} \\ \omega_x \\ \omega_y \\ \omega_z \end{bmatrix}$$





# Main algorithms

From kinematics to dynamics algorithms



# The Rigid Body Dynamics Algorithms

**Goal:** exploit at best the **sparsity** induced by the kinematic tree

The Articulated Body Algorithm

$$\ddot{q} = \text{ForwardDynamics} (q, \dot{q}, \tau, \lambda_c)$$

Simulation

Control

$$\tau = \text{InverseDynamics} (q, \dot{q}, \ddot{q}, \lambda_c)$$

The Recursive Newton-Euler Algorithm

$$M(q)\ddot{q} + C(q, \dot{q}) + G(q) = \tau + J_c^T(q)\lambda_c$$

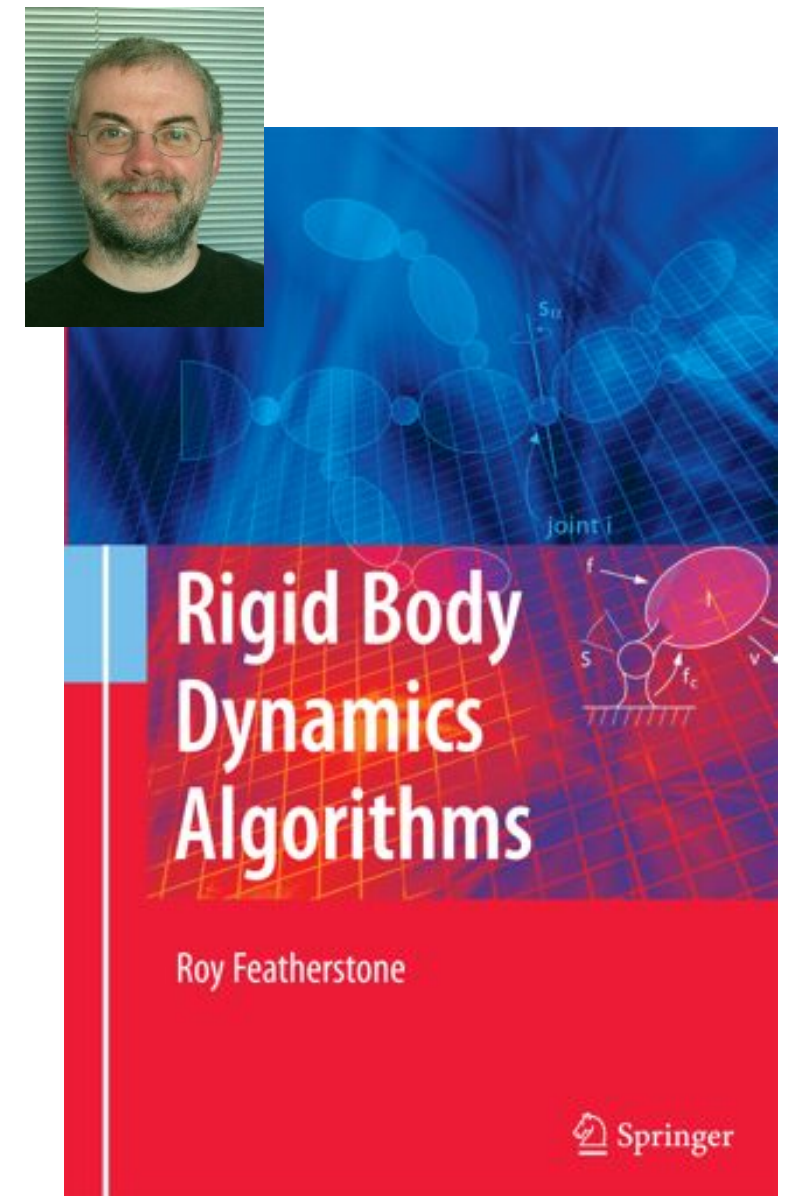
Mass  
Matrix

Coriolis  
centrifugal

Gravity

Motor  
torque

External  
forces

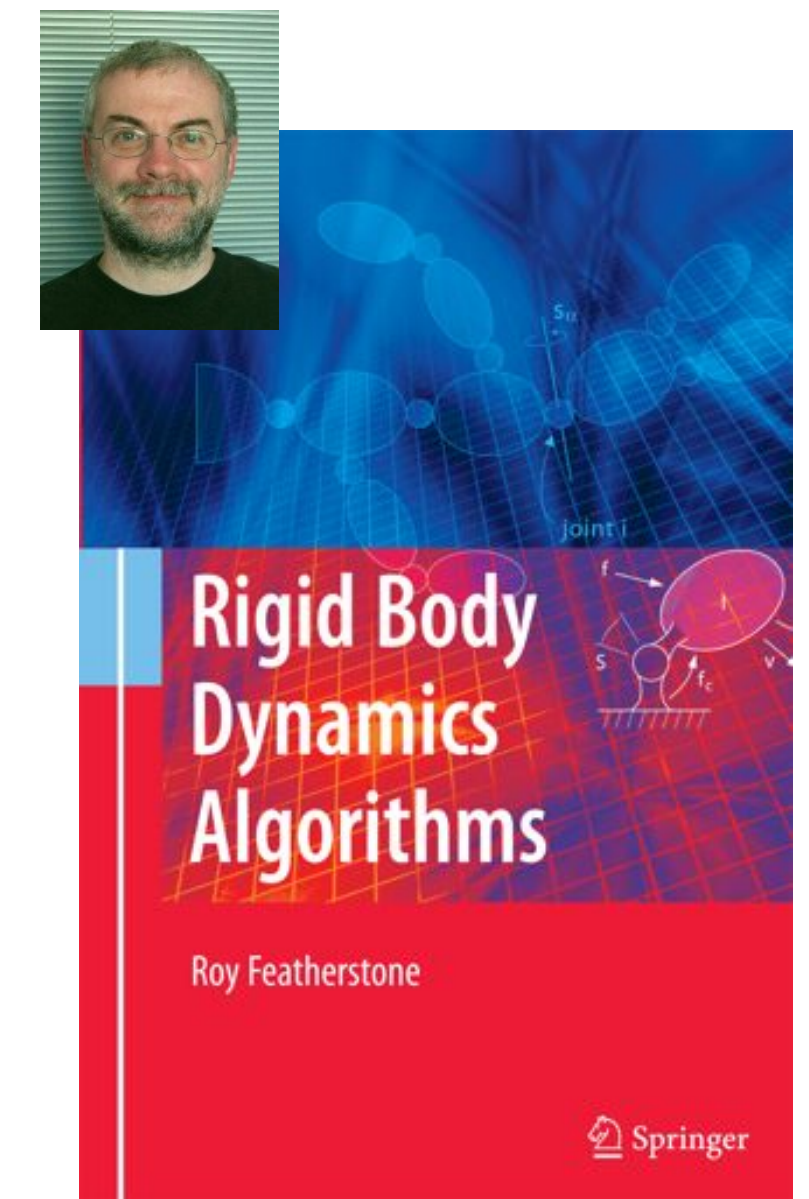


Roy Featherstone



# The Rigid Body Dynamics Algorithms

**Goal:** exploit at best the **sparsity** induced by the kinematic tree



Roy Featherstone

## The Articulated Body Algorithm

$$\ddot{q} = \text{ForwardDynamics} (q, \dot{q}, \tau, \lambda_c)$$

Simulation

Control

$$\tau = \text{InverseDynamics} (q, \dot{q}, \ddot{q}, \lambda_c)$$

## The Recursive Newton-Euler Algorithm

$$M(q)\ddot{q} + C(q, \dot{q}) + G(q) = \tau + J_c^T(q)\lambda_c$$

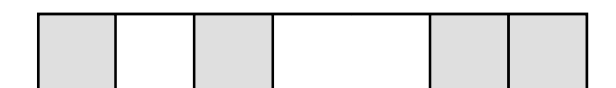
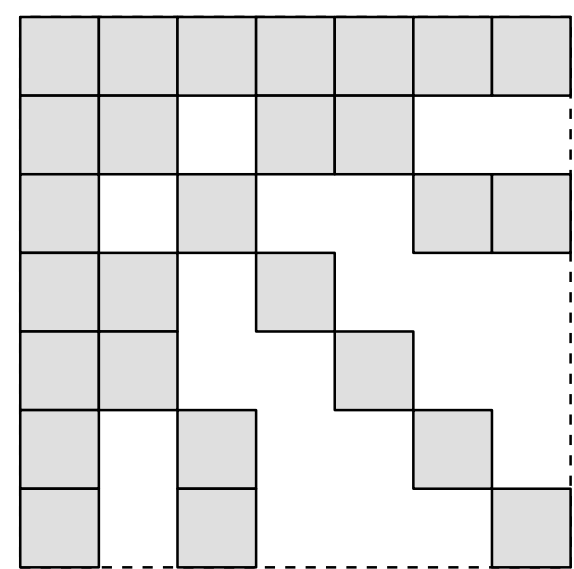
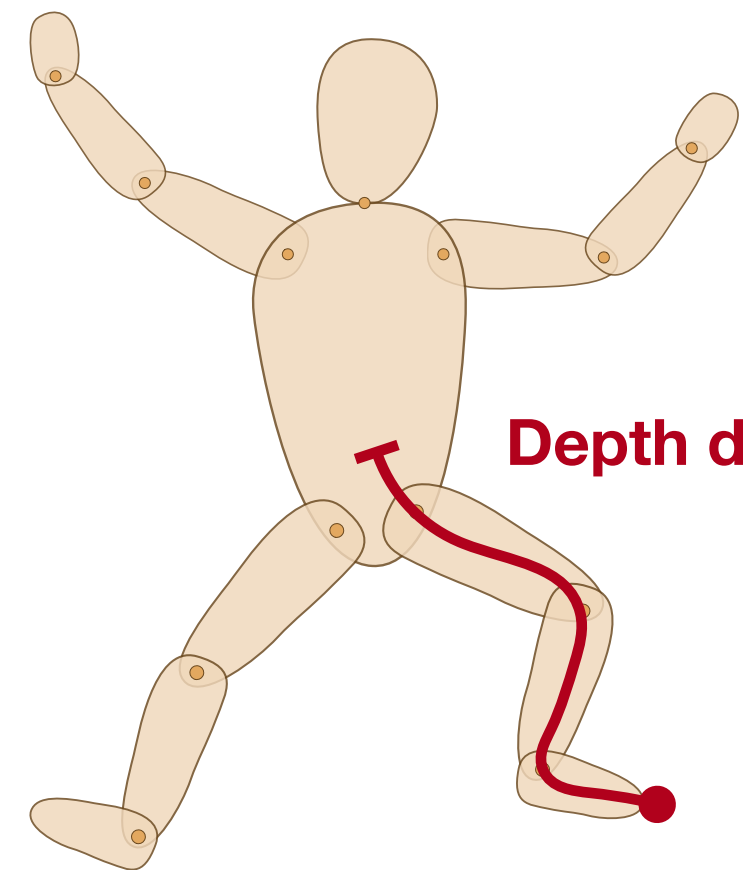
Mass Matrix

Coriolis centrifugal

Gravity

Motor torque

External forces





# The Recursive Newton-Euler Algorithm (1980)

Control

$$\tau = \text{InverseDynamics} (q, \dot{q}, \ddot{q}, \lambda_c)$$

The Recursive Newton-Euler Algorithm

$$\tau = M(q)\ddot{q} + C(q, \dot{q}) + G(q) - J_c^T(q)\lambda_c$$

Motor  
torque

Mass  
Matrix

Coriolis  
centrifugal

Gravity

External  
forces

Basic Equations:

$$v_0 = 0$$

$$a_0 = -a_g$$

$$v_i = v_{\lambda(i)} + S_i \dot{q}_i$$

$$a_i = a_{\lambda(i)} + S_i \ddot{q}_i + \dot{S}_i \dot{q}_i$$

$$f_i^B = I_i a_i + v_i \times^* I_i v_i$$

$$f_i = f_i^B - f_i^x + \sum_{j \in \mu(i)} f_j$$

$$\tau_i = S_i^T f_i$$

Equations in Body Coordinates:

$$v_0 = 0$$

$$a_0 = -a_g$$

$$v_{Ji} = S_i \dot{q}_i$$

$$c_{Ji} = \dot{S}_i \dot{q}_i$$

$$v_i = {}^i X_{\lambda(i)} v_{\lambda(i)} + v_{Ji}$$

$$a_i = {}^i X_{\lambda(i)} a_{\lambda(i)} + S_i \ddot{q}_i + c_{Ji} + v_i \times v_{Ji}$$

$$f_i^B = I_i a_i + v_i \times^* I_i v_i$$

$$f_i = f_i^B - {}^i X_0^* f_i^x + \sum_{j \in \mu(i)} {}^i X_j^* f_j$$

$$\tau_i = S_i^T f_i$$

Algorithm:

$$v_0 = 0$$

$$a_0 = -a_g$$

for  $i = 1$  to  $N_B$  do

$$[X_J, S_i, v_J, c_J] = \text{jcalc}(\text{jtype}(i), q_i, \dot{q}_i)$$

$${}^i X_{\lambda(i)} = X_J X_T(i)$$

if  $\lambda(i) \neq 0$  then

$${}^i X_0 = {}^i X_{\lambda(i)} {}^{\lambda(i)} X_0$$

end

$$v_i = {}^i X_{\lambda(i)} v_{\lambda(i)} + v_J$$

$$a_i = {}^i X_{\lambda(i)} a_{\lambda(i)} + S_i \ddot{q}_i + c_J + v_i \times v_J$$

$$f_i = I_i a_i + v_i \times^* I_i v_i - {}^i X_0^* f_i^x$$

end

for  $i = N_B$  to 1 do

$$\tau_i = S_i^T f_i$$

if  $\lambda(i) \neq 0$  then

$$f_{\lambda(i)} = f_{\lambda(i)} + {}^{\lambda(i)} X_i^* f_i$$

end

end



# The Articulated-Body Algorithm (1983)

## The Articulated Body Algorithm

$$\ddot{q} = \text{ForwardDynamics}(q, \dot{q}, \tau, \lambda_c)$$

Simulation

$$\ddot{q} = M^{-1}(q) \left( \tau - C(q, \dot{q}) - G(q) + J_c^T(q) \lambda_c \right)$$

Mass  
Matrix Inverse

Motor  
torque

Coriolis  
centrifugal

Gravity

External  
forces

Equations (in body coordinates):

Pass 1

$$v_0 = 0$$

$$v_{Ji} = S_i \dot{q}_i$$

$$c_{Ji} = \dot{S}_i \dot{q}_i$$

$$v_i = {}^i X_{\lambda(i)} v_{\lambda(i)} + v_{Ji}$$

$$c_i = c_{Ji} + v_i \times v_{Ji}$$

$$p_i = v_i \times^* I_i v_i - {}^i X_0^* f_i^x$$

Pass 2

$$I_i^A = I_i + \sum_{j \in \mu(i)} {}^i X_j^* I_j^a j X_i$$

$$p_i^A = p_i + \sum_{j \in \mu(i)} {}^i X_j^* p_j^a$$

$$U_i = I_i^A S_i$$

$$D_i = S_i^T U_i$$

$$u_i = \tau_i - S_i^T p_i^A$$

$$I_i^a = I_i^A - U_i D_i^{-1} U_i^T$$

$$p_i^a = p_i^A + I_i^a c_i + U_i D_i^{-1} u_i$$

Pass 3

$$a_0 = -a_g$$

$$a'_i = {}^i X_{\lambda(i)} a_{\lambda(i)} + c_i$$

$$\ddot{q}_i = D_i^{-1} (u_i - U_i^T a'_i)$$

$$a_i = a'_i + S_i \ddot{q}_i$$

Algorithm:

$$v_0 = 0$$

for  $i = 1$  to  $N_B$  do

$$[X_J, S_i, v_J, c_J] = \text{jcalc}(\text{jtype}(i), q_i, \dot{q}_i)$$

$${}^i X_{\lambda(i)} = X_J X_T(i)$$

if  $\lambda(i) \neq 0$  then

$${}^i X_0 = {}^i X_{\lambda(i)} {}^{\lambda(i)} X_0$$

end

$$v_i = {}^i X_{\lambda(i)} v_{\lambda(i)} + v_J$$

$$c_i = c_J + v_i \times v_J$$

$$I_i^A = I_i$$

$$p_i^A = v_i \times^* I_i v_i - {}^i X_0^* f_i^x$$

end

for  $i = N_B$  to 1 do

$$U_i = I_i^A S_i$$

$$D_i = S_i^T U_i$$

$$u_i = \tau_i - S_i^T p_i^A$$

if  $\lambda(i) \neq 0$  then

$$I_i^a = I_i^A - U_i D_i^{-1} U_i^T$$

$$p_i^a = p_i^A + I_i^a c_i + U_i D_i^{-1} u_i$$

$$I_{\lambda(i)}^A = I_{\lambda(i)}^A + {}^{\lambda(i)} X_i^* I_i^a {}^i X_{\lambda(i)}$$

$$p_{\lambda(i)}^A = p_{\lambda(i)}^A + {}^{\lambda(i)} X_i^* p_i^a$$

end

end

$$a_0 = -a_g$$

for  $i = 1$  to  $N_B$  do

$$a'_i = {}^i X_{\lambda(i)} a_{\lambda(i)} + c_i$$

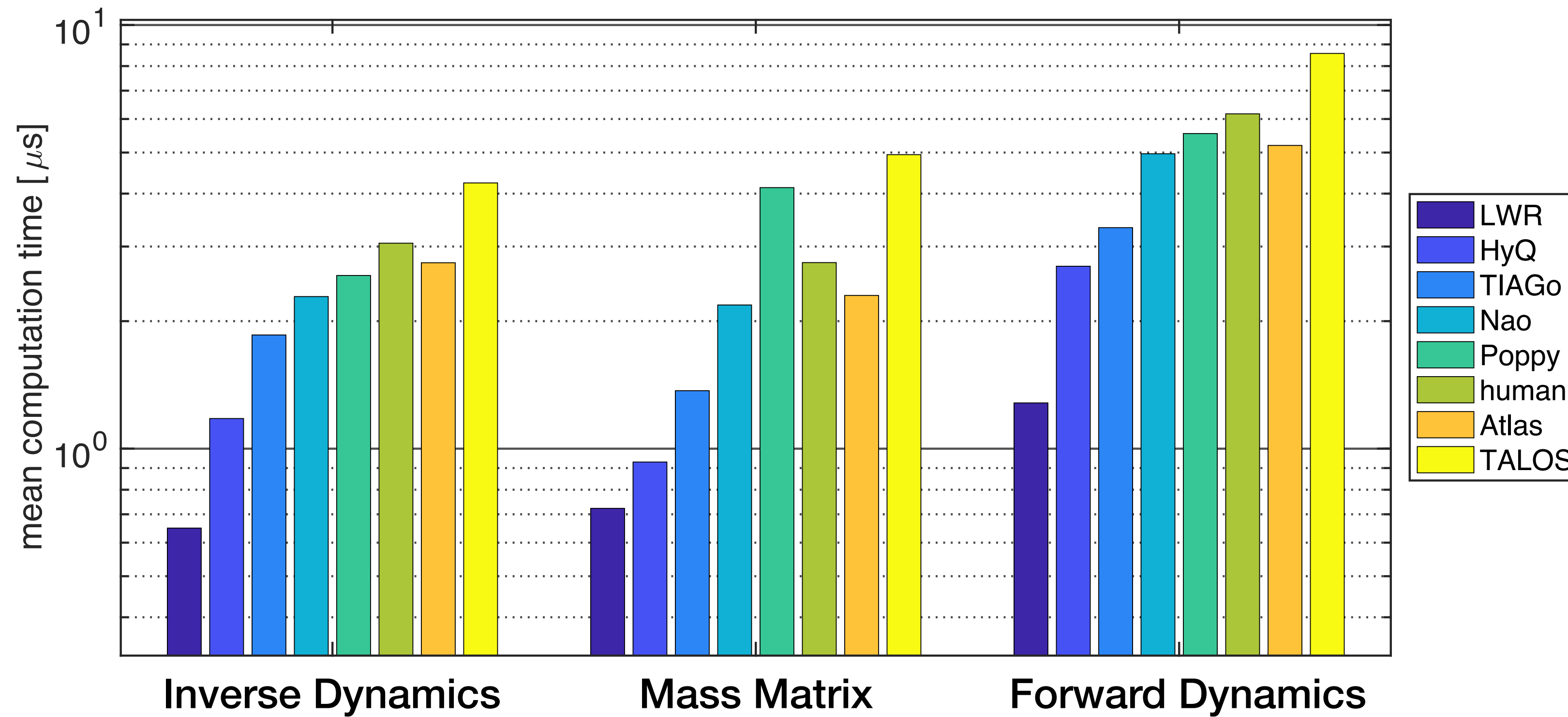
$$\ddot{q}_i = D_i^{-1} (u_i - U_i^T a'_i)$$

$$a_i = a'_i + S_i \ddot{q}_i$$

end



# Benchmarks of basic algorithms



Kuka LWR



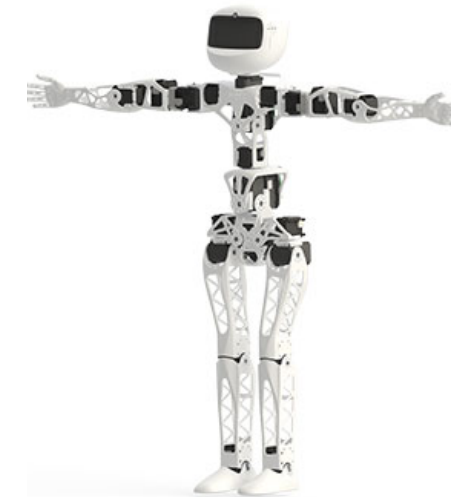
HyQ



TIAGo



NAO



Poppy



Human



ATLAS

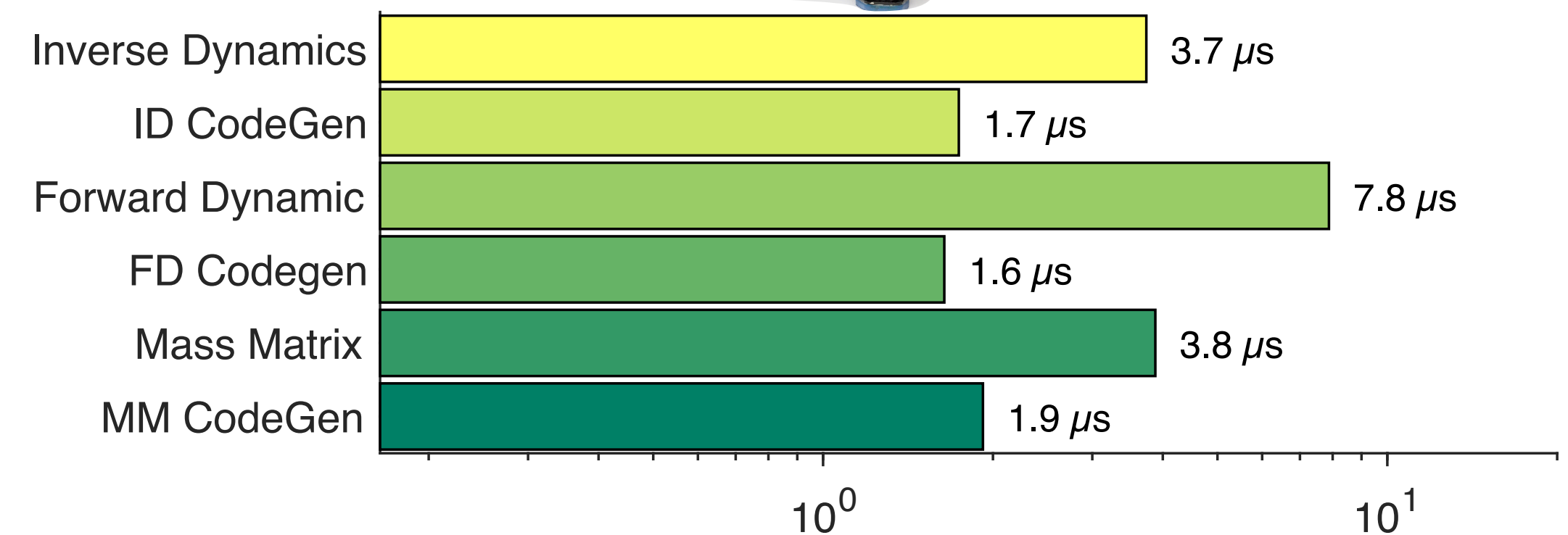
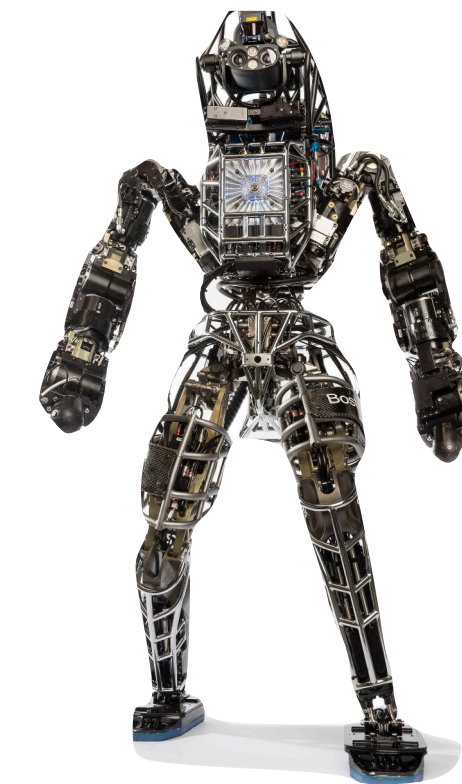
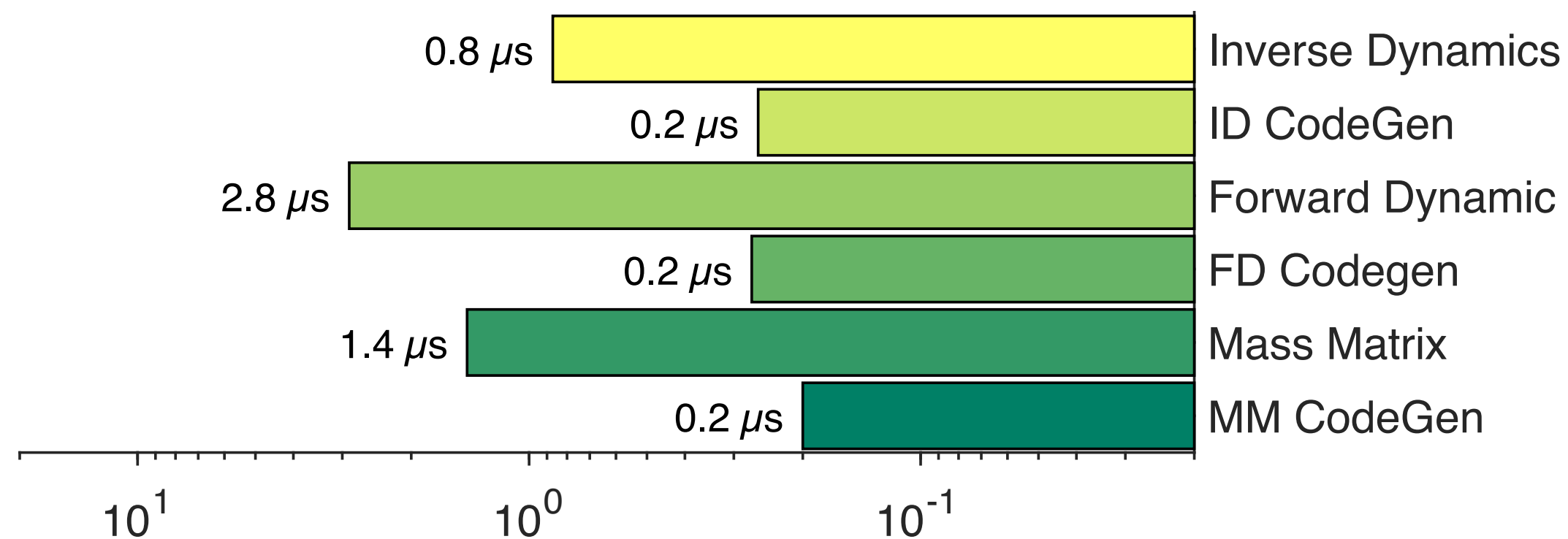


TALOS



# Code generation

Pinocchio also supports **source code generation**:  
you can **compile on the fly (JIT paradigm)** your code  
for the best performances on your hardware





# Analytical derivatives

Beyond automatic differentiation for fast and reliable computations




# Analytical Derivatives of Robot Dynamics

Numerical Optimal Control or Reinforcement Learning approaches require access to **Forward or Inverse Dynamics** functions and their **partial derivatives**

## Inverse Dynamics

$$\tau = \mathbf{ID}(q, \dot{q}, \ddot{q}, \lambda_c)$$


$$\frac{\partial \mathbf{ID}}{\partial q}, \frac{\partial \mathbf{ID}}{\partial \dot{q}}, \frac{\partial \mathbf{ID}}{\partial \ddot{q}}, \frac{\partial \mathbf{ID}}{\partial \lambda_c}$$

## Forward Dynamics

$$\ddot{q} = \mathbf{FD}(q, \dot{q}, \tau, \lambda_c)$$


$$\frac{\partial \mathbf{FD}}{\partial q}, \frac{\partial \mathbf{FD}}{\partial \dot{q}}, \frac{\partial \mathbf{FD}}{\partial \tau}, \frac{\partial \mathbf{FD}}{\partial \lambda_c}$$



# Classic ways to evaluate Numerical Derivatives

## Finite Differences

- > Consider the input function as a **black-box**

$$y = f(x)$$

- > Add a **small increment** on the input variable

$$\frac{dy}{dx} \approx \frac{f(x + dx) - f(x)}{dx}$$

### Pros

- > Works for any input function
- > Easy implementation

### Cons

- > Not efficient
- > Sensitive to numerical rounding errors



# Classic ways to evaluate Numerical Derivatives

## Finite Differences

- > Consider the input function as a **black-box**

$$y = f(x)$$

- > Add a **small increment** on the input variable

$$\frac{dy}{dx} \approx \frac{f(x + dx) - f(x)}{dx}$$

### Pros

- > Works for any input function
- > Easy implementation

### Cons

- > Not efficient
- > Sensitive to numerical rounding errors

## Automatic Differentiation

- > This time, we know the **elementary operations** in f

$$y = f(x) = a \cdot \cos(x)$$

- > Apply the **chain rule formula** and use derivatives of basic functions

$$\frac{dy}{dx} = \underbrace{\frac{da}{dx}}_{=0} \cdot \cos(x) + a \cdot \frac{d \cos(x)}{dx} = -a \cdot \sin(x)$$

### Pros

- > Efficient frameworks
- > Very accurate

### Cons

- > Requires specific implementation
- > Not able to exploit spatial algebra derivatives



# Analytical Derivatives of Dynamics Algorithms

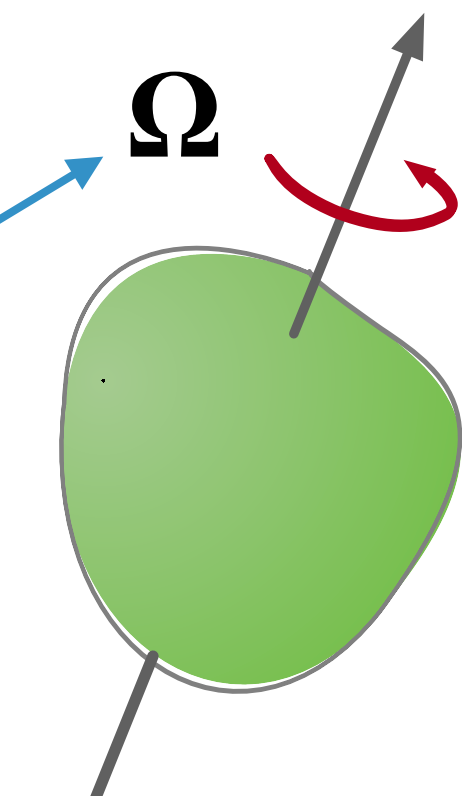
## Why analytical derivatives?

We must exploit the **intrinsic geometry** of the **differential operators** involved in rigid motions

$$\frac{dR}{dt} = R [\Omega]_{\times}$$

orientation matrix

velocity vector



The diagram shows a green, irregularly shaped rigid body. A grey arrow passes through its center, representing the rotation axis. A red curved arrow around this axis indicates the direction of rotation. A blue arrow labeled  $\Omega$  points from the text 'velocity vector' to the rotation axis, representing the angular velocity vector.



# Analytical Derivatives of Dynamics Algorithms

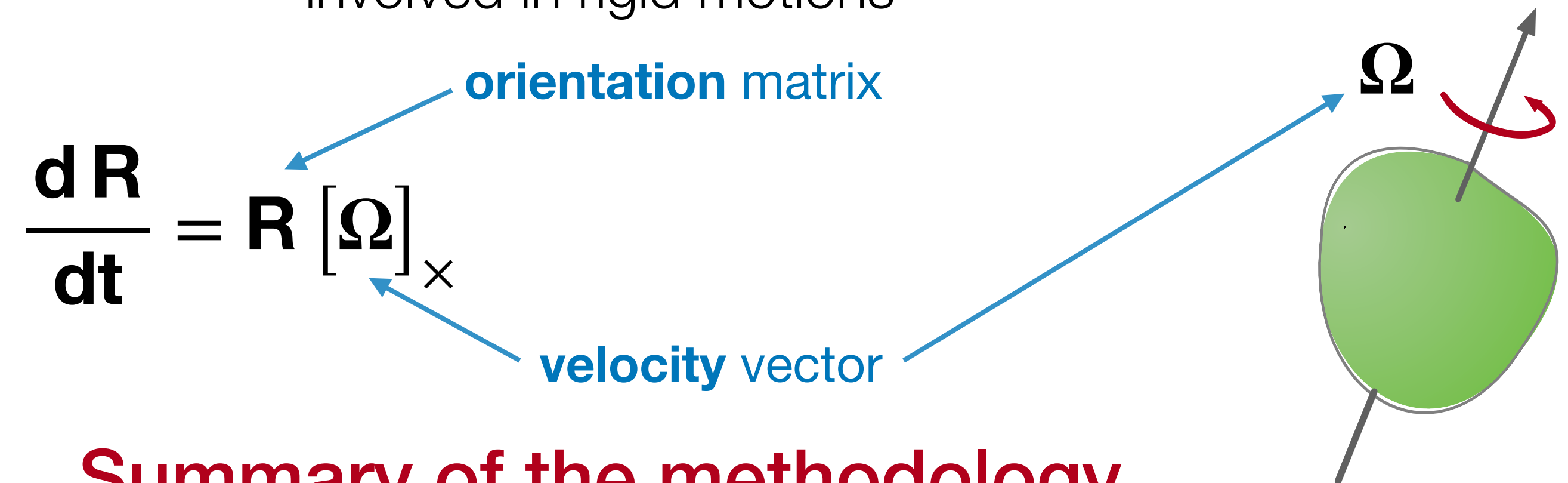
The Recursive Newton-Euler algorithm  
to compute  $\tau = \mathbf{ID}(\mathbf{q}, \dot{\mathbf{q}}, \ddot{\mathbf{q}})$

```

Algorithm:
v0 = 0
a0 = -ag
for i = 1 to NB do
  [XJ, Si, vJ, cJ] =
    jcalc(jtype(i), qi, q̇i)
  iXλ(i) = XJ XT(i)
  if λ(i) ≠ 0 then
    iX0 = iXλ(i) λ(i)X0
  end
  vi = iXλ(i) vλ(i) + vJ
  ai = iXλ(i) aλ(i) + Si q̈i
    + cJ + vi × vJ
  fi = Ii ai + vi ×* Ii vi - iX0* fix
end
for i = NB to 1 do
  τi = SiT fi
  if λ(i) ≠ 0 then
    fλ(i) = fλ(i) + λ(i)Xi* fi
  end
end
end
    
```

## Why analytical derivatives?

We must exploit the **intrinsic geometry** of the **differential operators**  
involved in rigid motions



## Summary of the methodology

Applying the **chain rule formula** on each line of the Recursive Newton-Euler algorithm  
**AND** exploiting the **sparsity** of spatial operations



# Analytical Derivatives of Dynamics Algorithms

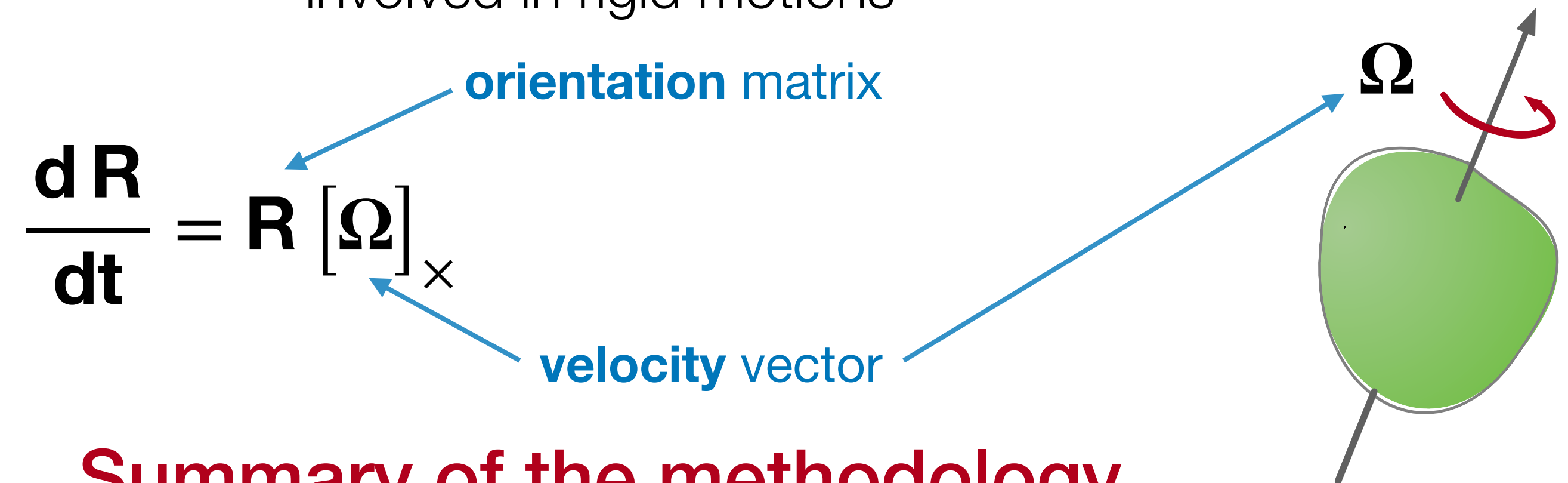
The Recursive Newton-Euler algorithm to compute  $\tau = \mathbf{ID}(\mathbf{q}, \dot{\mathbf{q}}, \ddot{\mathbf{q}})$

```

Algorithm:
v0 = 0
a0 = -ag
for i = 1 to NB do
  [XJ, Si, vJ, cJ] =
    jcalc(jtype(i), qi, q̇i)
  iXλ(i) = XJ XT(i)
  if λ(i) ≠ 0 then
    iX0 = iXλ(i) λ(i)X0
  end
  vi = iXλ(i) vλ(i) + vJ
  ai = iXλ(i) aλ(i) + Si q̈i
    + cJ + vi × vJ
  fi = Ii ai + vi ×* Ii vi - iX0* fix
end
for i = NB to 1 do
  τi = SiT fi
  if λ(i) ≠ 0 then
    fλ(i) = fλ(i) + λ(i)Xi* fi
  end
end
end
    
```

## Why analytical derivatives?

We must exploit the **intrinsic geometry** of the **differential operators** involved in rigid motions



## Summary of the methodology

Applying the **chain rule formula** on each line of the Recursive Newton-Euler algorithm **AND** exploiting the **sparsity** of spatial operations

## Outcome

A **simple** but **efficient** algorithm, relying on spatial algebra **AND** keeping a minimal complexity of  $O(Nd)$  **WHILE** the state of the art is  $O(N^2)$



# Analytical Derivatives of Robot Dynamics

Forward Dynamics and Inverse Dynamics are reciprocal functions:

$$\mathbf{FD} \circ \mathbf{ID} = \mathbf{ID} \circ \mathbf{FD} = \mathit{Id}$$

which leads to the following relation:

$$\frac{\partial \mathbf{FD}}{\partial X} \mathbf{ID} + \mathbf{FD} \frac{\partial \mathbf{ID}}{\partial X} = \mathbf{0}$$

thus:

$$\frac{\partial \mathbf{FD}}{\partial X} = - \mathbf{FD} \frac{\partial \mathbf{ID}}{\partial X} \mathbf{FD}$$



# Analytical Derivatives of Robot Dynamics

## Analytical Derivatives of Rigid Body Dynamics Algorithms

Justin Carpentier and Nicolas Mansard  
Laboratoire d'Analyse et d'Architecture des Systèmes and Université de Toulouse  
Email: justin.carpentier@laas.fr

**Abstract**—Rigid body dynamics is a well-established framework in robotics. It can be used to expose the analytic form of kinematic and dynamic functions of the robot model. So far, two major algorithms, namely the recursive Newton-Euler algorithm (RNEA) and the articulated body algorithm (ABA), have been proposed to compute the inverse dynamics and the forward dynamics in a few microseconds. Evaluating their derivatives is an important challenge for various robotic applications (optimal control, estimation, co-design or reinforcement learning). However it remains time consuming, whether using finite differences or automatic differentiation. In this paper, we propose new algorithms to efficiently compute them thanks to closed-form formulations. Using the chain rule and adequate algebraic differentiation of spatial algebra, we firstly differentiate explicitly RNEA. Then, using properties about the derivative of function composition, we show that the same algorithm can also be used to compute the derivatives of ABA with a marginal additional cost. For this purpose, we introduce a new algorithm to compute the inverse of the joint-space inertia matrix, without explicitly computing the matrix itself. All the algorithms are implemented in our open-source C++ framework called Pinocchio. Benchmarks show computational costs varying between 3 microseconds (for a 7-dof arm) up to 17 microseconds (for a 36-dof humanoid), outperforming the alternative approaches of the state of the art.

### I. INTRODUCTION

Rigid-body-dynamics algorithms [7] are a well-established framework at the heart of many recent robotic applications, and have become even popular in related domains such as biomechanics and computer animation. This is mostly due to their ability to compute in a generic and efficient way the kinematic and dynamic quantities that describe the motion of poly-articulated systems. Rigid body dynamics algorithms are for example crucial for the control and the stabilization of quadruped and humanoid robots [10, 15, 17]. Additionally, optimal control and trajectory optimization are becoming standard approaches to control complex robotic systems [26, 16], generate human-like or avatar motions [28, 22], or for instance in the context of simultaneous design and control of robots [27, 8, 14]. They mostly rely on an accurate integration of the forward dynamics together with the differentiation of the resulting quantities with respect to the state, model parameters and control variables of the system. A large part of the total computational cost of such optimization algorithms (up to 90%) is spent in computing these derivatives.

This work is supported by the RoboCom++ FLAG-ERA JTC 2016 proposal and the European project MEMMO (GA-780684).

Evaluating the partial derivatives of the dynamics can be performed in several manners. The simplest way is to approximate them by finite differences, i.e. evaluating several times the input dynamics while adding a small increment on the input variables. The main advantage is to systematize the derivation process by considering the function to differentiate as a black box. It comes at the price of calling  $n + 1$  times the input function (with  $n$  the number of input variables). It is also sensitive to numerical rounding errors. Yet, if this approach has shown to be fast enough to be applied on real systems [29, 16], it requires fine parallelization. Another methodology is to analytically derive the Lagrangian equation of motion [11]. Lagrangian derivation gives a better insight into the structure of the derivatives but leads to dense computations. It fails to exploit efficiently the sparsity induced by the kinematic model, in a similar way than rigid body dynamics algorithms do. A last method is to rely on automatic differentiation of rigid body dynamics algorithms as implemented in the control toolbox Drake [30] and more recently exploited by Gifftthaler et al. [12]. The idea is to overload the scalar type of the input variables, by applying the chain rule formula in an automatic way knowing the derivatives of basic functions (cos, sin or exp), to obtain the partial derivatives. Automatic differentiation typically requires intermediate computations which are hard to avoid or to simplify. Using code generation [12] can mitigate this issue but is a costly technological process to set up.

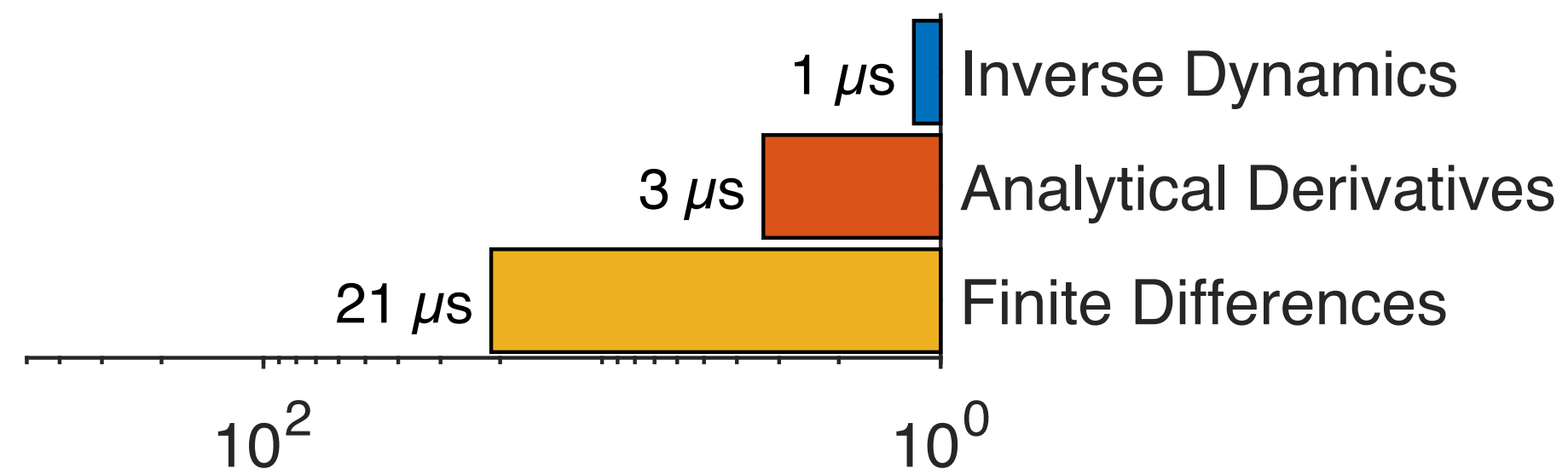
In this paper, we rather propose to analytically derive the rigid-body-dynamics algorithms in order to speed up the computation of the derivatives. Our formulation provides a better insight into the mathematical structure of the derivatives. We are then able to exploit the inherent structure of spatial algebra (e.g. the cross product operator) at the root of rigid-body-dynamics algorithms, while the aforementioned approaches are in fact not able to do so. Our method extends previous works on serial chains with loop closures Lee et al. [18] to any kinematic tree while exploiting the more expressive spatial algebra. We also provide a complete, efficient and open source implementation on which our benchmarks are based.

This paper is made of two concomitant contributions. In a first contribution we establish in a concise way the analytical derivatives of the inverse dynamics through the differentiation of the so-called recursive Newton-Euler algorithm (RNEA) [19, 7]. The second contribution concerns the analytical derivatives of the forward dynamics. Rather than

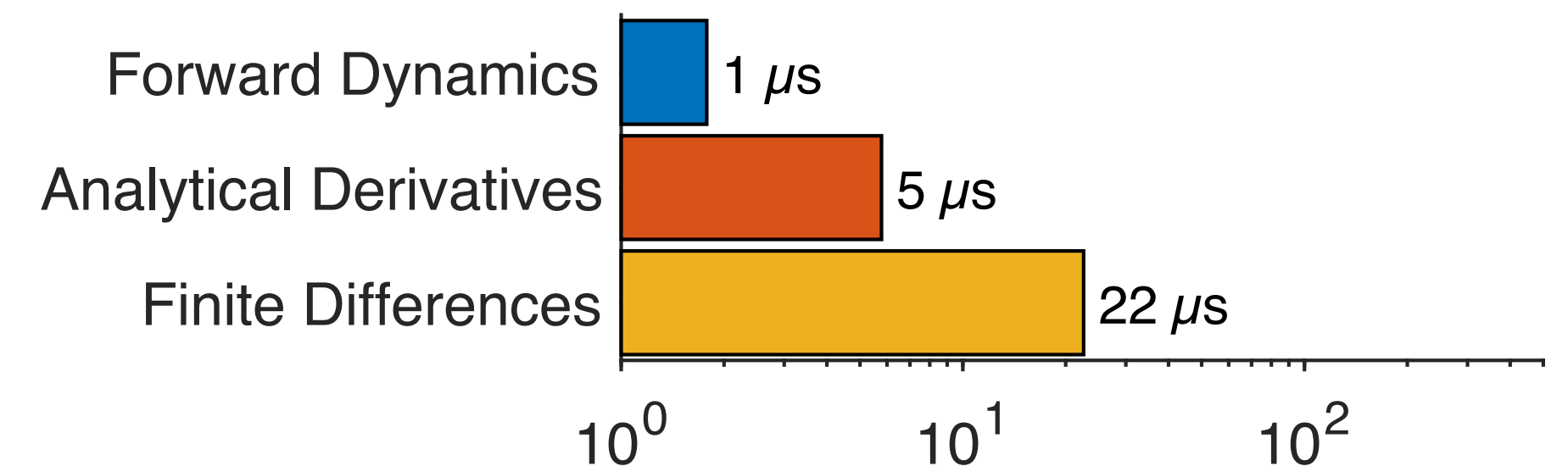


# Benchmarks of analytical derivatives

## Inverse Dynamics



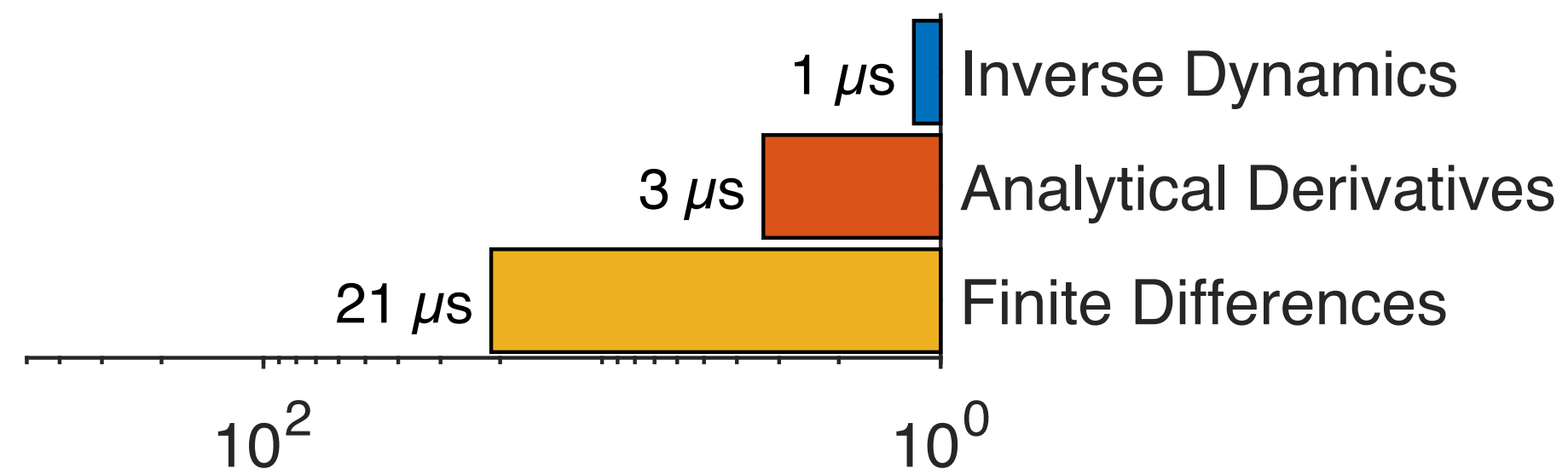
## Forward Dynamics



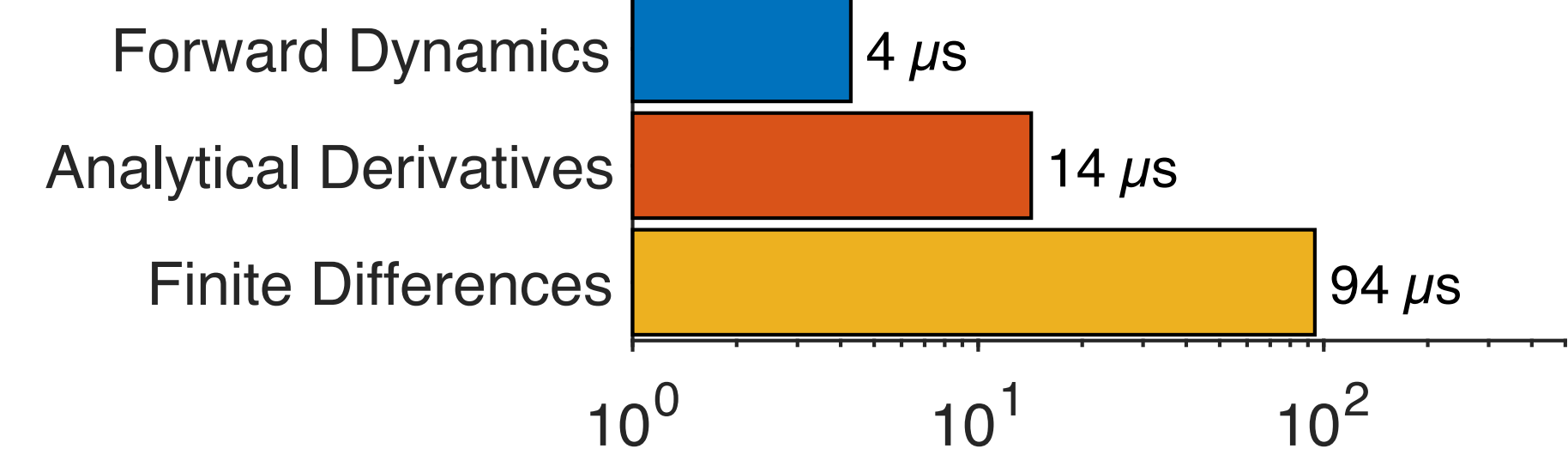
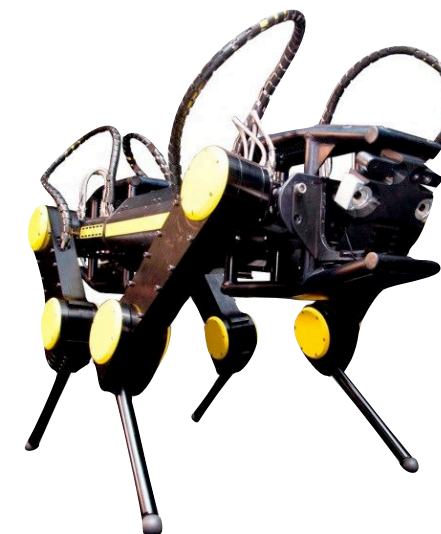
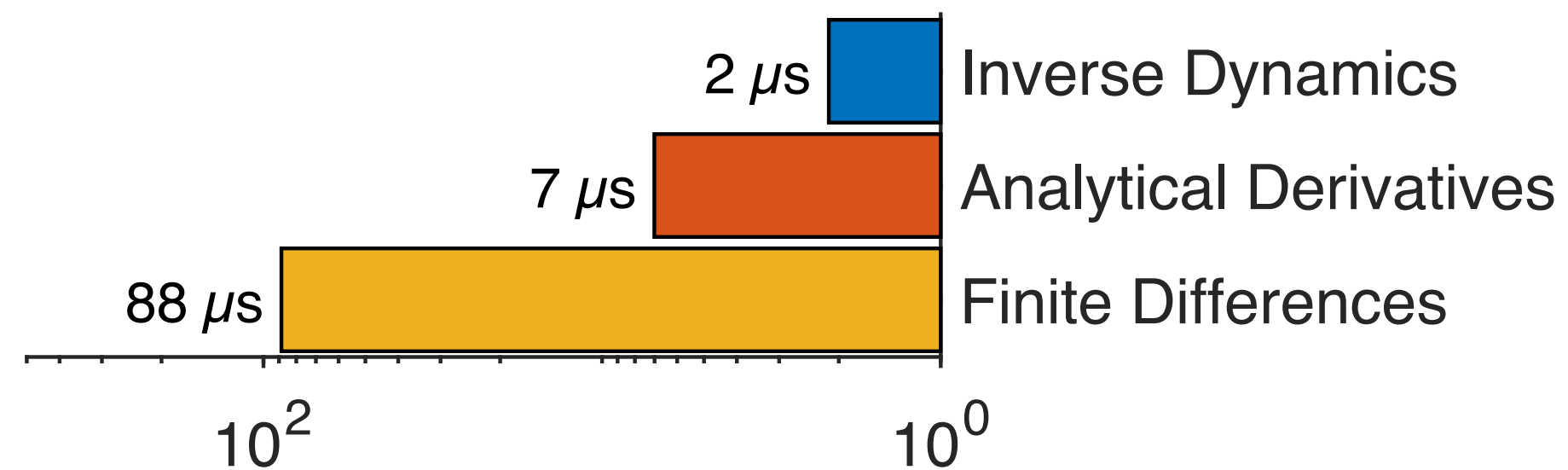
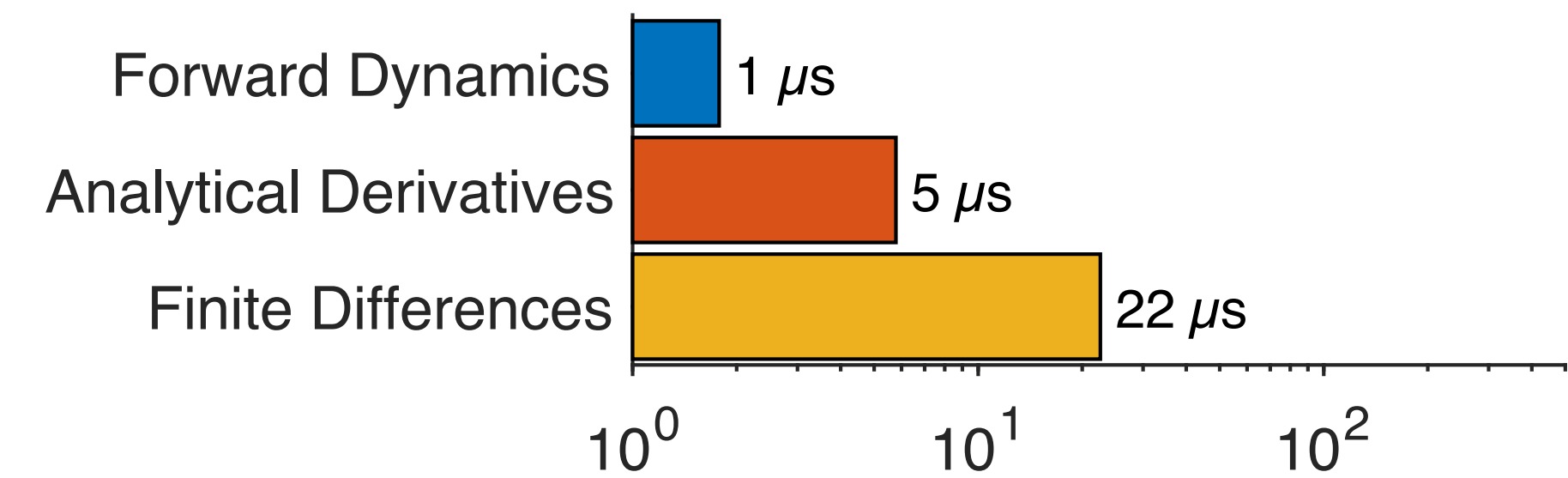


# Benchmarks of analytical derivatives

## Inverse Dynamics



## Forward Dynamics

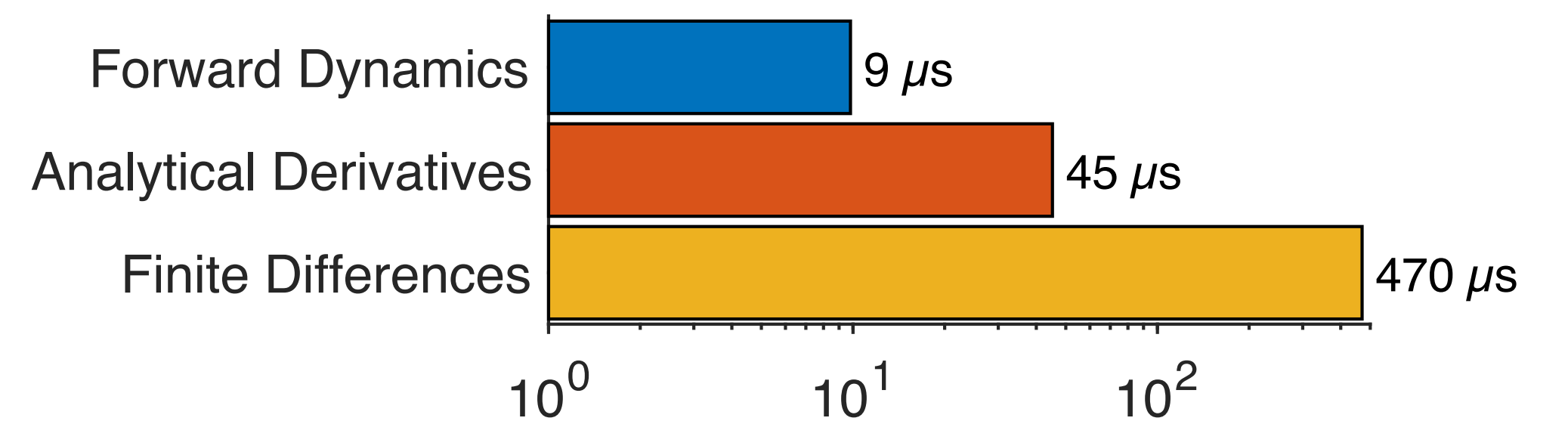
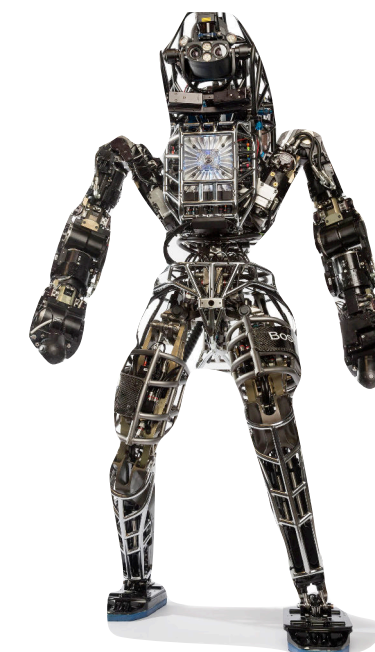
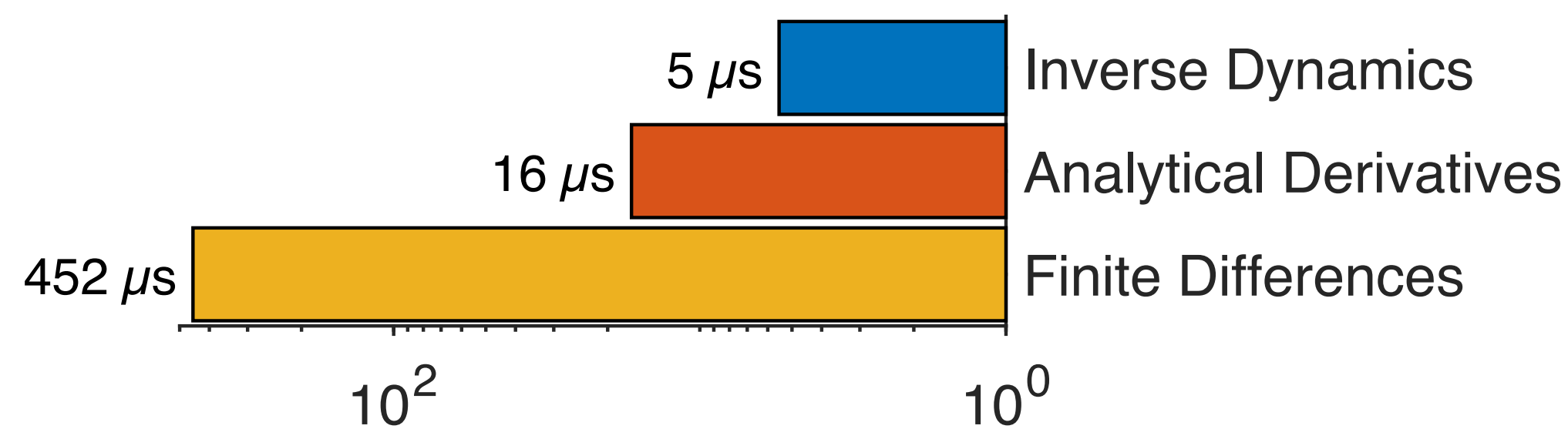
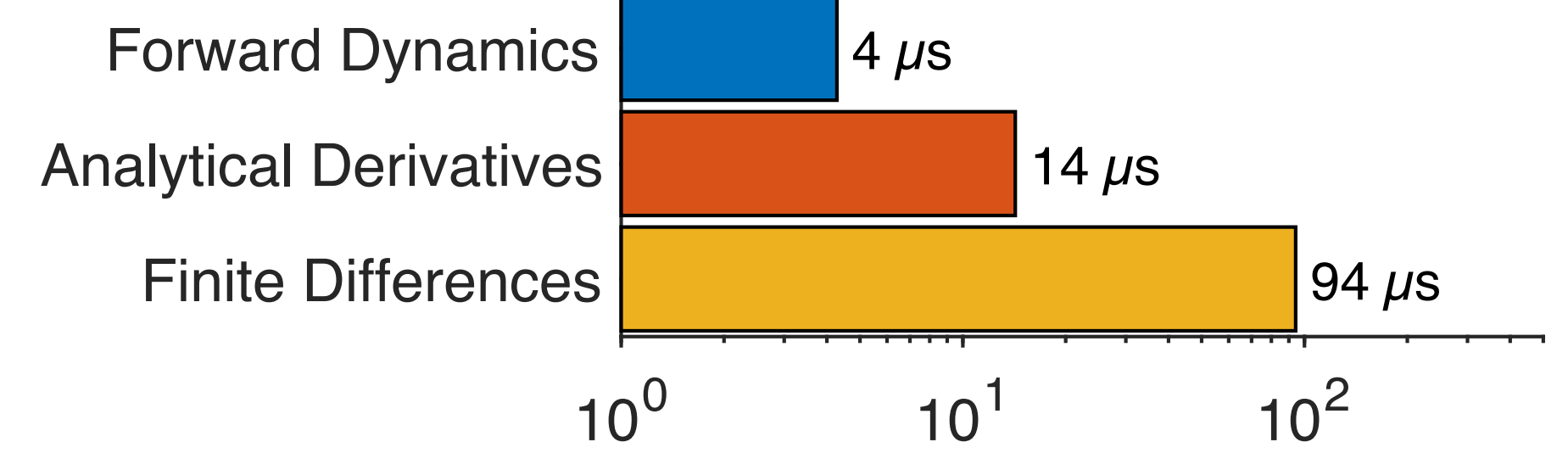
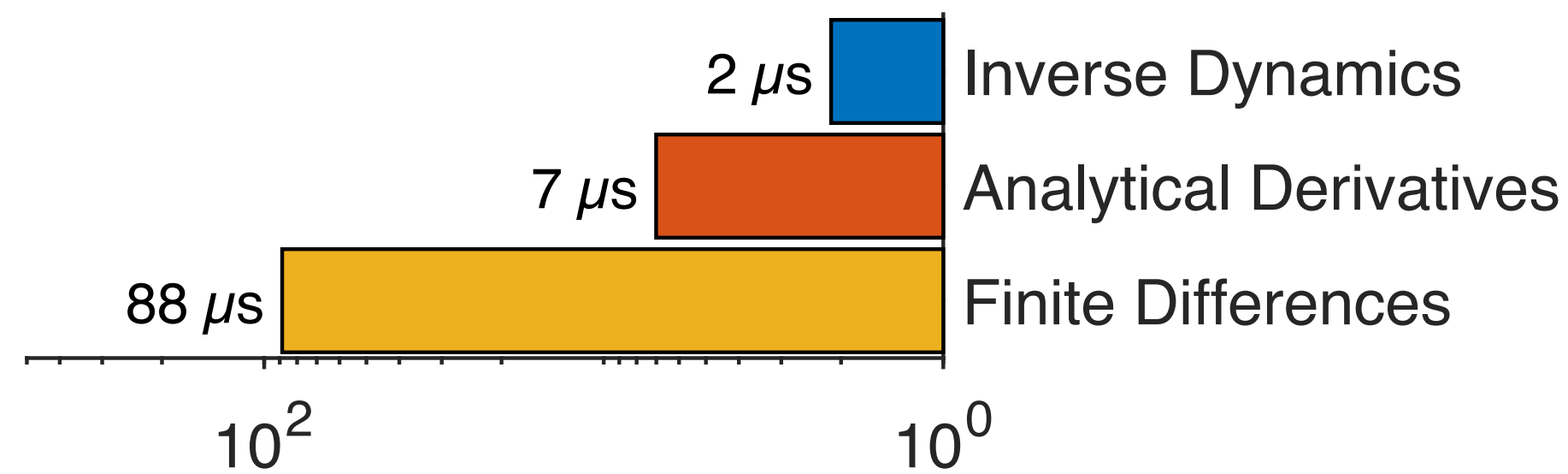
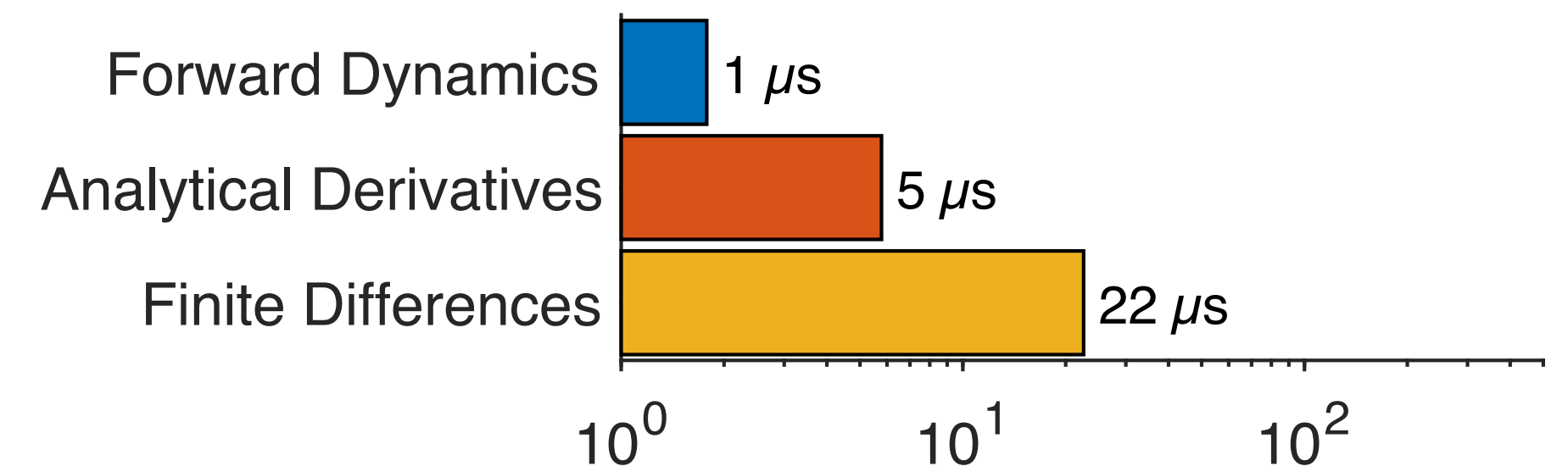
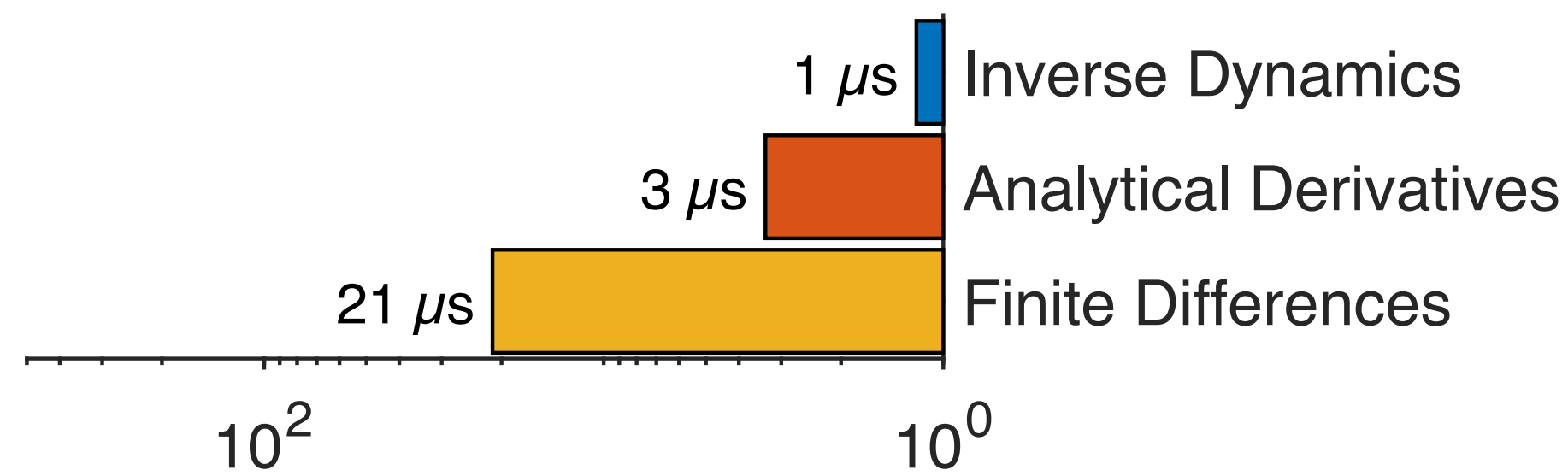




# Benchmarks of analytical derivatives

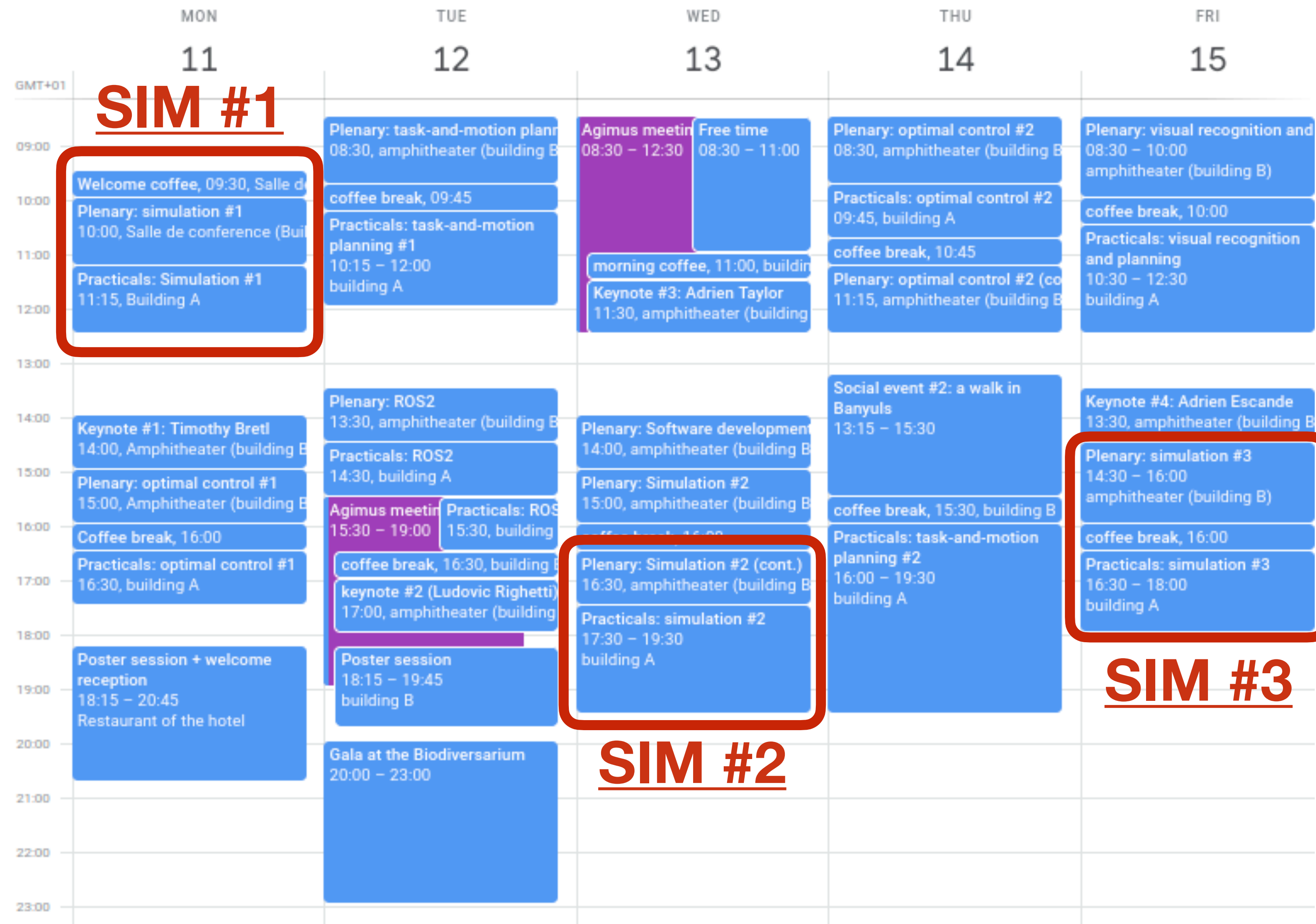
## Inverse Dynamics

## Forward Dynamics





# Simulation classes of the week





Time to play  
Ready?!





# AGIMUS 2023 Winter School

---

[Main website](#)

[Chat room](#)

## Tutorials

---

In this winter school, we will cover three main different topics:

- [Simulation](#)
  - [Simulation #1: Rigid body dynamics](#)
- [Optimal control](#)
- [Motion planning]

## Installing dependencies

---

Dependencies for the coursework include specific versions of Pinocchio, hpp-fcl, crocodyl, alligator, and other software that have yet to be fully released. We provide two ways to install the required packages for Mac OS and Linux: conda/mamba or docker.

### Conda installation [Mac OS Intel, Mac OS ARM, Linux x64]

All the required packages are available on the following [channel](#). Conda can be easily installed on your machine by following these [instructions](#).

You can install a package by typing in your terminal:

```
conda create -n aws python=3.10
conda activate aws
conda install -c agm-ws-2023 my_package_name
```



The two first lines create a new environment named `aws` and then activate it. The third line installs `my_package_name` using the [channel](#) of the AGIMUS winter school where the packages have been compiled.

You can also consider installing additional tools via pip, like:
















```
pip install tqdm meshcat ipython
```



We also invite you to leverage [visual studio code](#) to play with the Jupyter notebooks. Don't forget to install the [Jupyter module](#) for [visual studio code](#).



 **jcarpent** sim: move all the materials to a dedicated dir b6f6660 · 6 minutes ago  History

Name	Last commit message	Last commit date
 ..		
 1_geometry_and_dynamics.ipynb	sim: move all the materials to a dedicated dir	6 minutes ago
 2_derivatives.ipynb	sim: move all the materials to a dedicated dir	6 minutes ago
 README.md	sim: move all the materials to a dedicated dir	6 minutes ago
 confused.png	sim: move all the materials to a dedicated dir	6 minutes ago
 contact_planner.py	sim: move all the materials to a dedicated dir	6 minutes ago
 costs.py	sim: move all the materials to a dedicated dir	6 minutes ago
 dexp.py	sim: move all the materials to a dedicated dir	6 minutes ago
 logo-pinocchio.png	sim: move all the materials to a dedicated dir	6 minutes ago
 magic_donotload.py	sim: move all the materials to a dedicated dir	6 minutes ago
 proto.py	sim: move all the materials to a dedicated dir	6 minutes ago
 question.png	sim: move all the materials to a dedicated dir	6 minutes ago
 recap.png	sim: move all the materials to a dedicated dir	6 minutes ago
 solutions.py	sim: move all the materials to a dedicated dir	6 minutes ago
 vizutils.py	sim: move all the materials to a dedicated dir	6 minutes ago



