Differentiable Simulation

STELIAN COROS, ETH Zurich
MILES MACKLIN, Nvidia
BERNHARD THOMASZEWSKI, Université de Montréal & ETH Zurich
NILS THÜREY, Technical University of Munich

Differentiable simulation is emerging as a fundamental building block for many cutting-edge applications in computer graphics, vision and robotics, among others. This course provides an introduction to this topic and an overview of state-of-the-art methods in this context. Starting with the basics of dynamic mechanical systems, we will present a general theoretical framework for differentiable simulation, which we will specialize to rigid bodies, deformable solids, and fluids. A particular focus will be on the different alternatives for computing simulation derivatives, ranging from analytical expressions via sensitivity analysis to reverse-mode automatic differentiation. As an important step towards real-world applications, we also present extensions to non-smooth phenomena such as frictional contact. Finally, we will discuss different ways of integrating differentiable simulation into machine learning frameworks.

The material covered in this course is based on the author's own works and experience, complemented by a state-of-the-art review of this young but rapidly evolving field. It will be richly illustrated, annotated, and supported by examples ranging from robotic manipulation of deformable materials to simulation-based capture of dynamic fluids. The theoretical parts will be accompanied by source code examples that will be made available to participants prior to this course.

ACM Reference Format:
Stelian Coros, Miles Macklin, Bernhard Thomaszewski, and Nils Thürey. 2021. Differentiable Simulation. 1, 1 (June 2021), 1 page. https://doi.org/10.1145/nnnnnnn.nnnnnnn

SINGLE SENTENCE SUMMARY

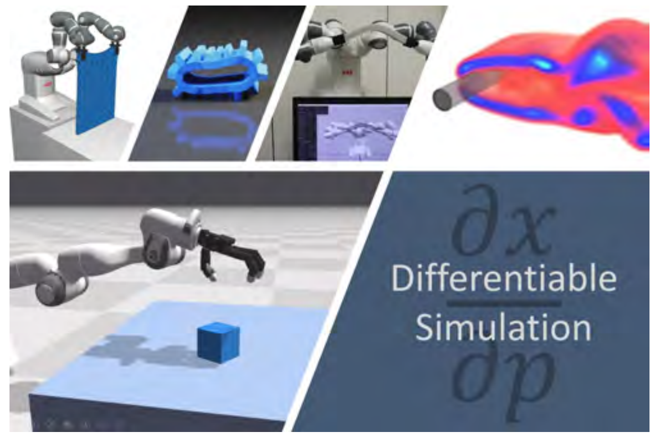
This course provides an introduction to the theory and practice of differentiable simulation of dynamic mechanical systems (rigid bodies, deformable solids, and fluids) with applications to robotic manipulation, parameter estimation, and fluid capture.

INTENDED AUDIENCE

This course is aimed at students and practitioners from the field of physics-based animation who are interested to learn about the theory and practice of differentiable simulation, and to researchers with computer vision and machine learning backgrounds who would like to leverage differentiable simulation for their applications.

PREREQUISITES

Besides a solid background knowledge in linear algebra, basic calculus, and ordinary differential equations, we also expect a working knowledge in physics-based simulation of rigid bodies, deformable solids, and fluids.



COURSE RATIONAL

Participants will gain an understanding of the theory and practice of differentiable simulation. This knowledge will be valuable both for practitioners seeking to use differentiable simulation for solving specific problems, as well as researchers interested in exploring this emerging field.

PEDAGOGICAL INTENTIONS AND METHODS

Our approach is to provide a self-contained account of the basic theory, accompanied by source code examples, while providing pointers to the literature for advanced topics. This, we believe, will provide participants with basic theoretical and practical understanding of differentiable simulation, and provide directions for deepening their knowledge as desired. If the format allows for it, we will actively solicit questions from the audience at regular intervals.

Authors' addresses: Stelian Coros, ETH Zurich; Miles Macklin, Nvidia; Bernhard Thomaszewski, Université de Montréal & ETH Zurich; Nils Thürey, Technical University of Munich.

2021. XXXX-XXXX/2021/6-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Course Content and Syllabus

Introduction . . . . (10 min.)
Stelian Coros, Miles Macklin, Bernhard Thomaszewski, Nils Thürey

Differentiable Simulation Basics . . . . (30 min.)
Stelian Coros, Bernhard Thomaszewski

Computing Simulation Derivatives . . . . (25 min.)
Bernhard Thomaszewski, Miles Macklin

Differentiable Simulation with Frictional Contact . . . . (20 min.)
Stelian Coros, Miles Macklin

Integration with Machine Learning Frameworks . . . . (20 min.)
Nils Thürey

Break (15 min.)

Robotic Manipulation of Deformable Materials . . . . (30 min.)
Stelian Coros, Bernhard Thomaszewski

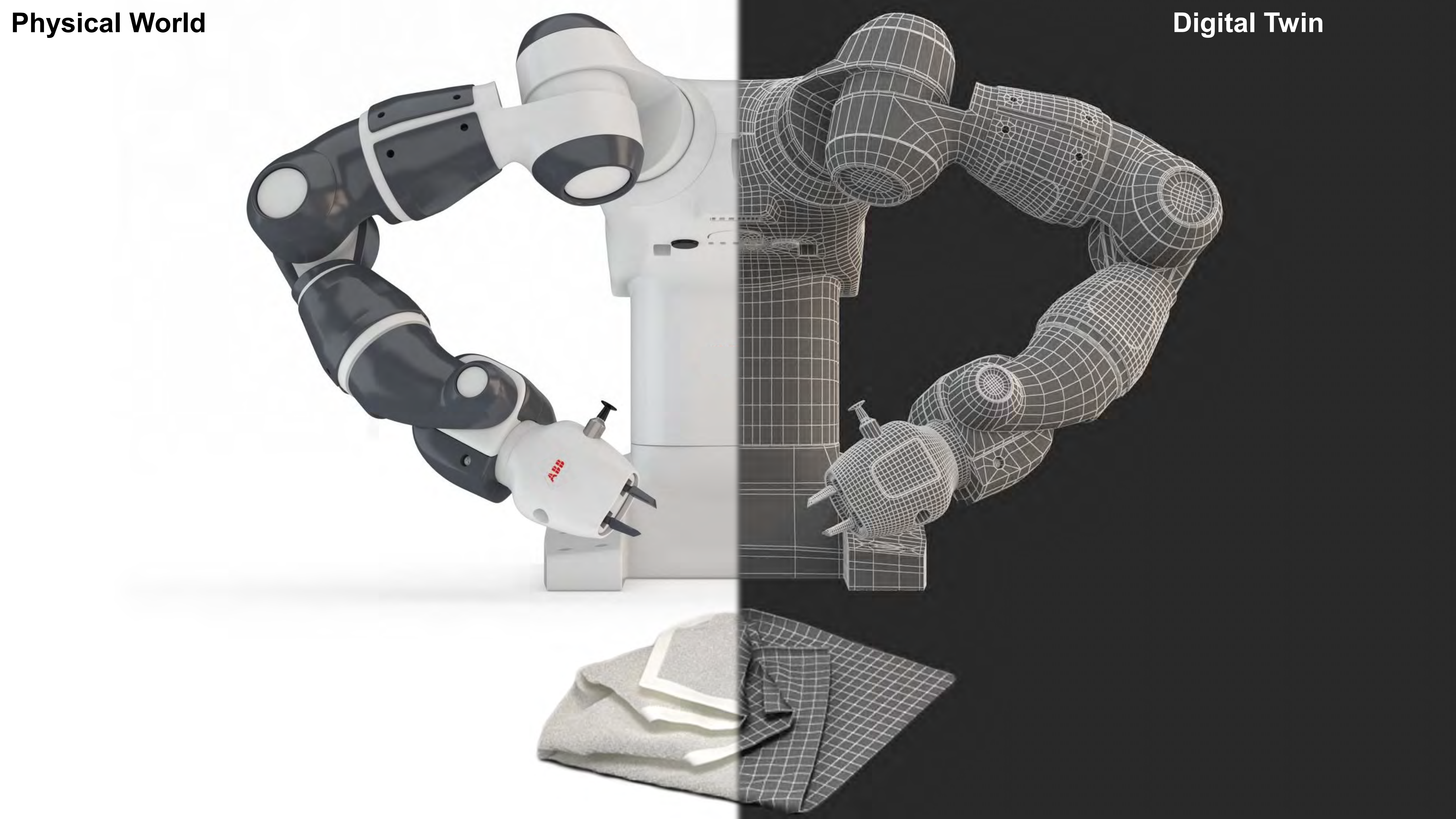
Industry-Scale Differentiable Simulation . . . . (30 min.)
Miles Macklin

Differentiable Simulation and Deep-Learning for Fluids . . . . . (30 min.)
Nils Thürey

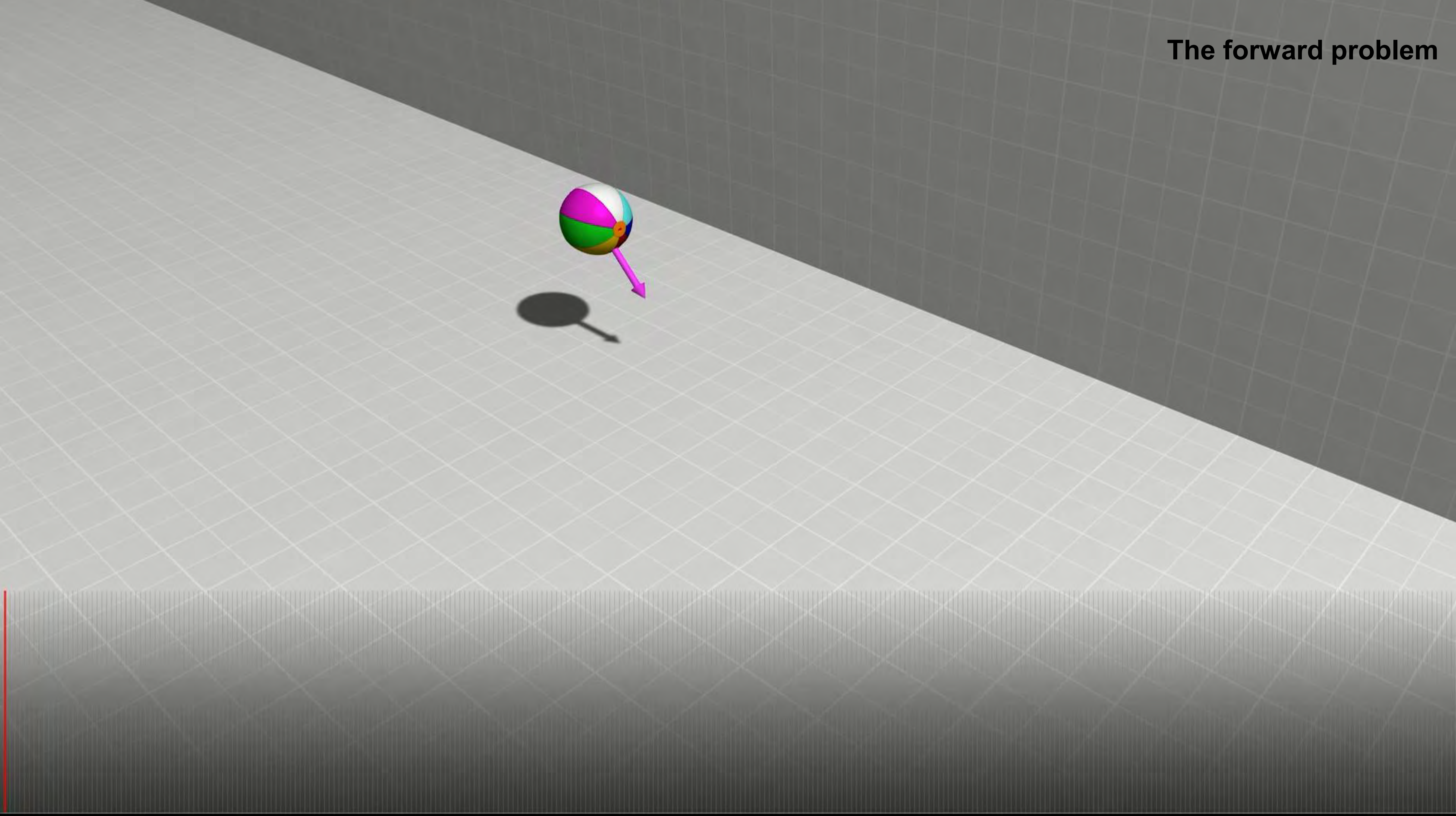
Conclusions and Questions . . . . (15 min.)
Stelian Coros, Miles Macklin, Bernhard Thomaszewski, Nils Thürey

DIFFERENTIABLE SIMULATION BASICS

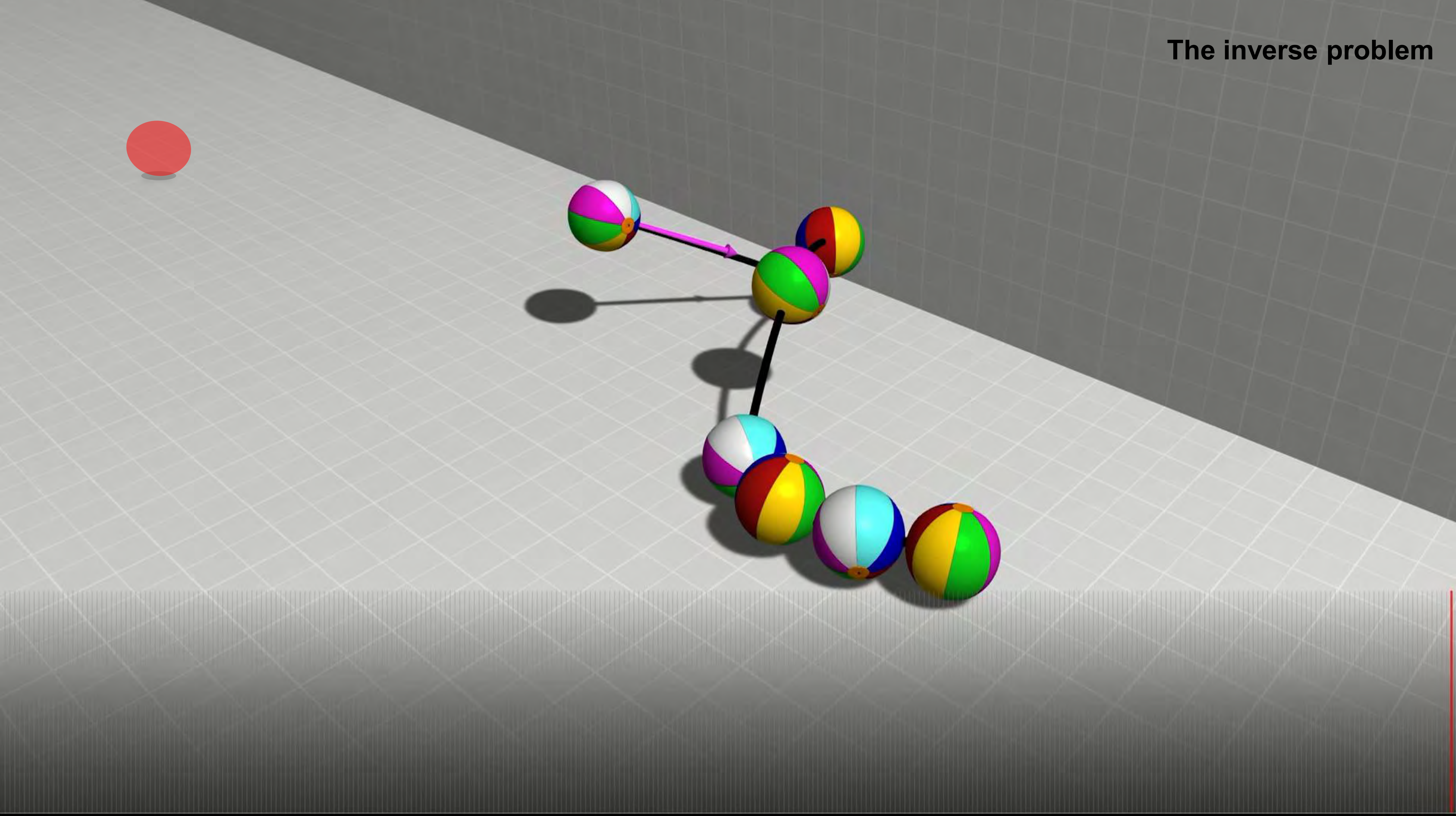
Stelian Coros, Bernhard Thomaszewski


Physical World
Digital Twin

The forward problem



The inverse problem



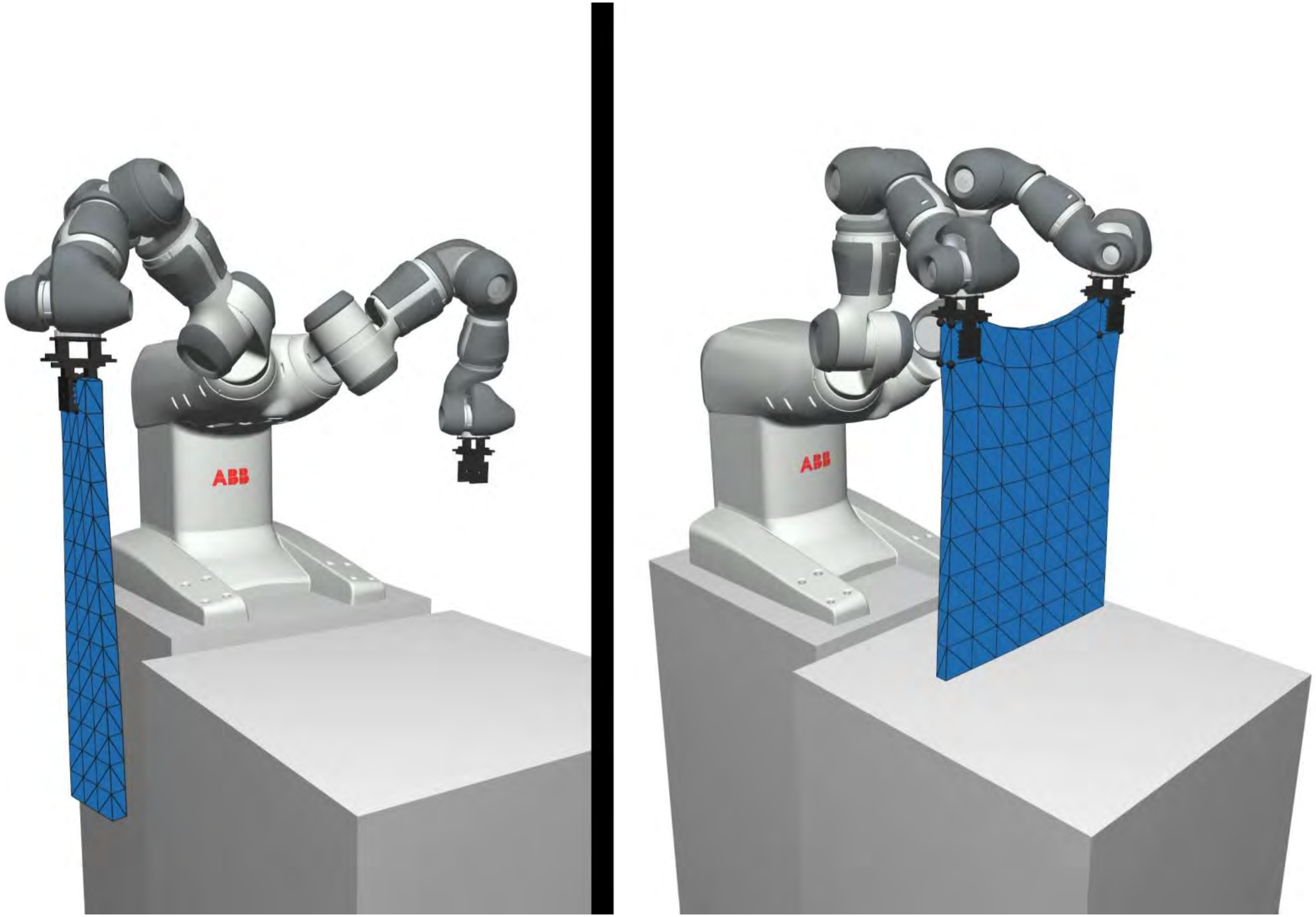
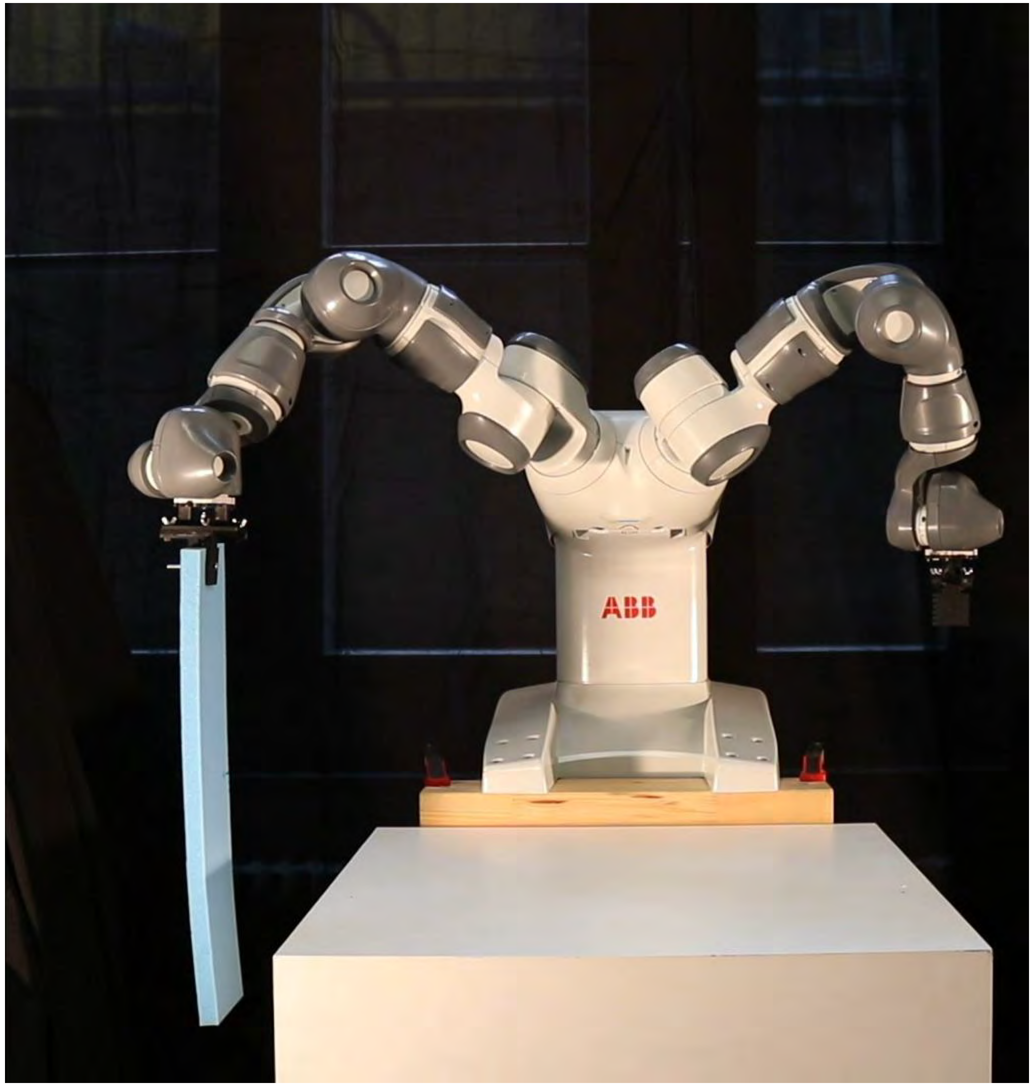
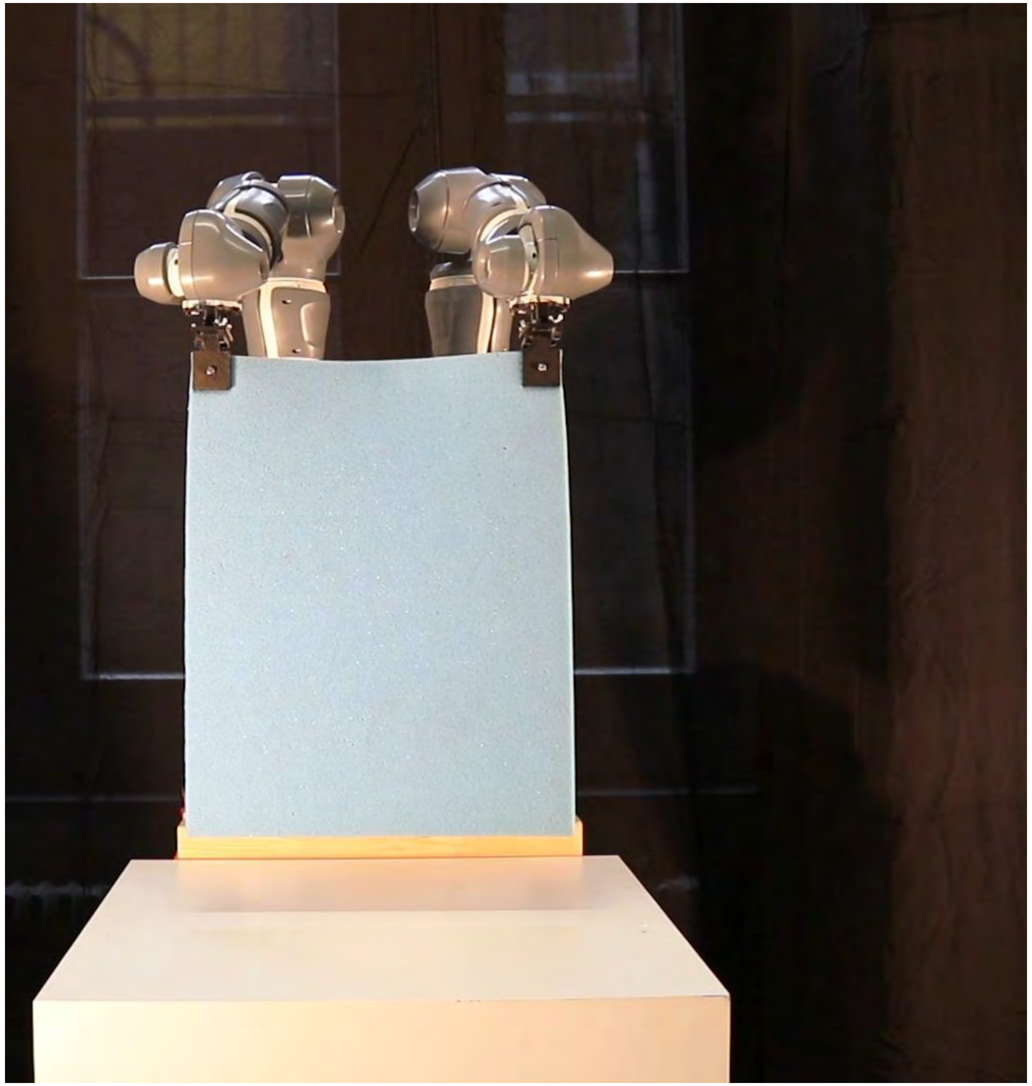


ABB
ABB

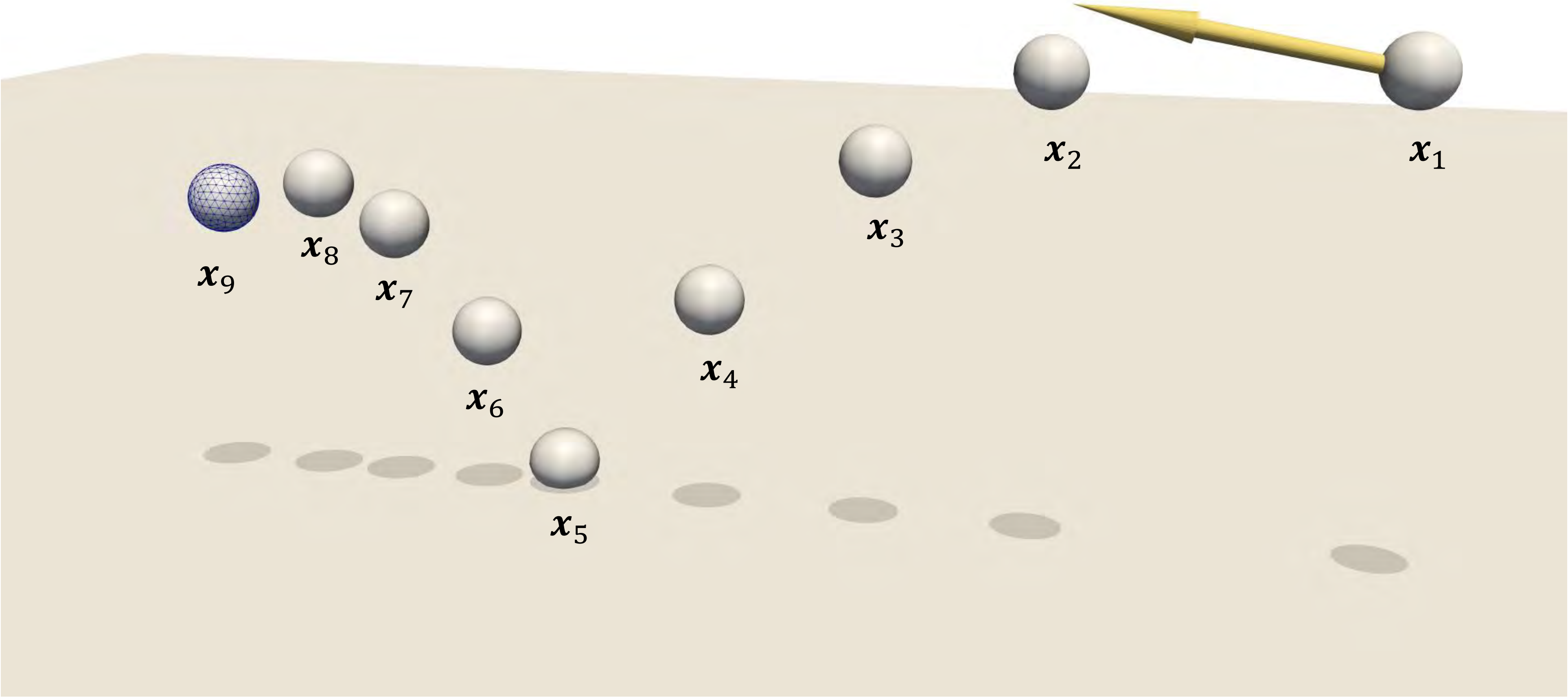


ABB

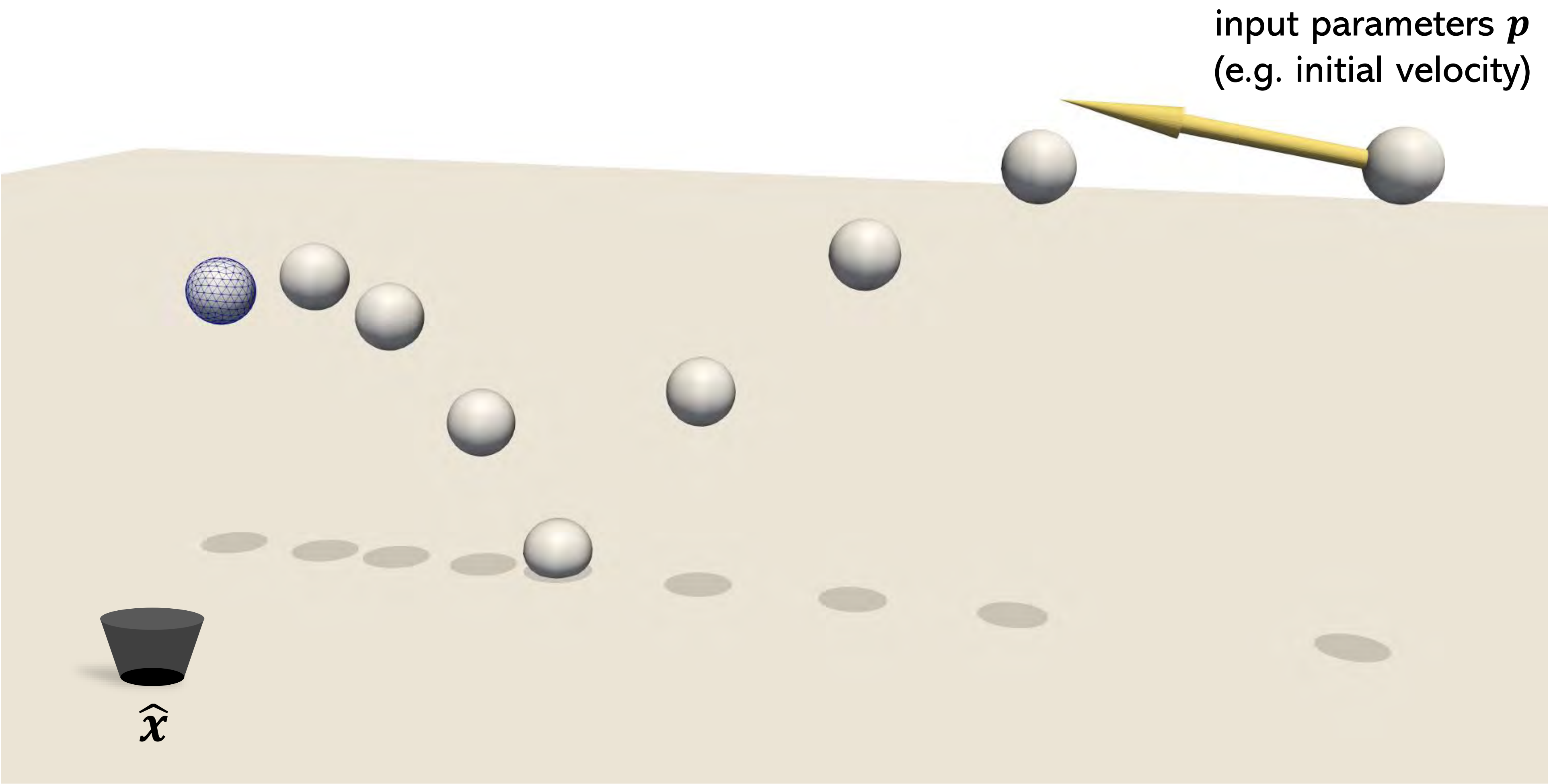


Traditional Simulation

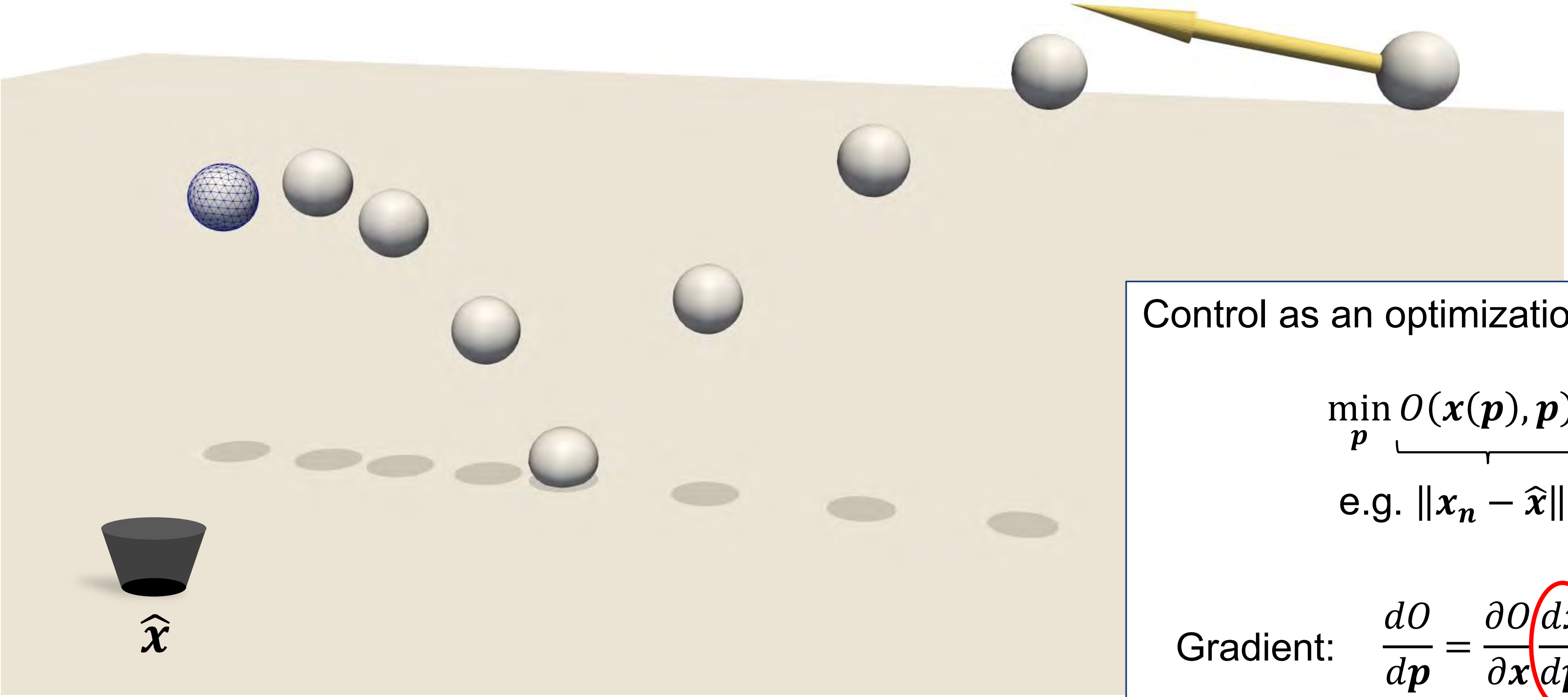
Traditional Simulation



A control problem



A control problem



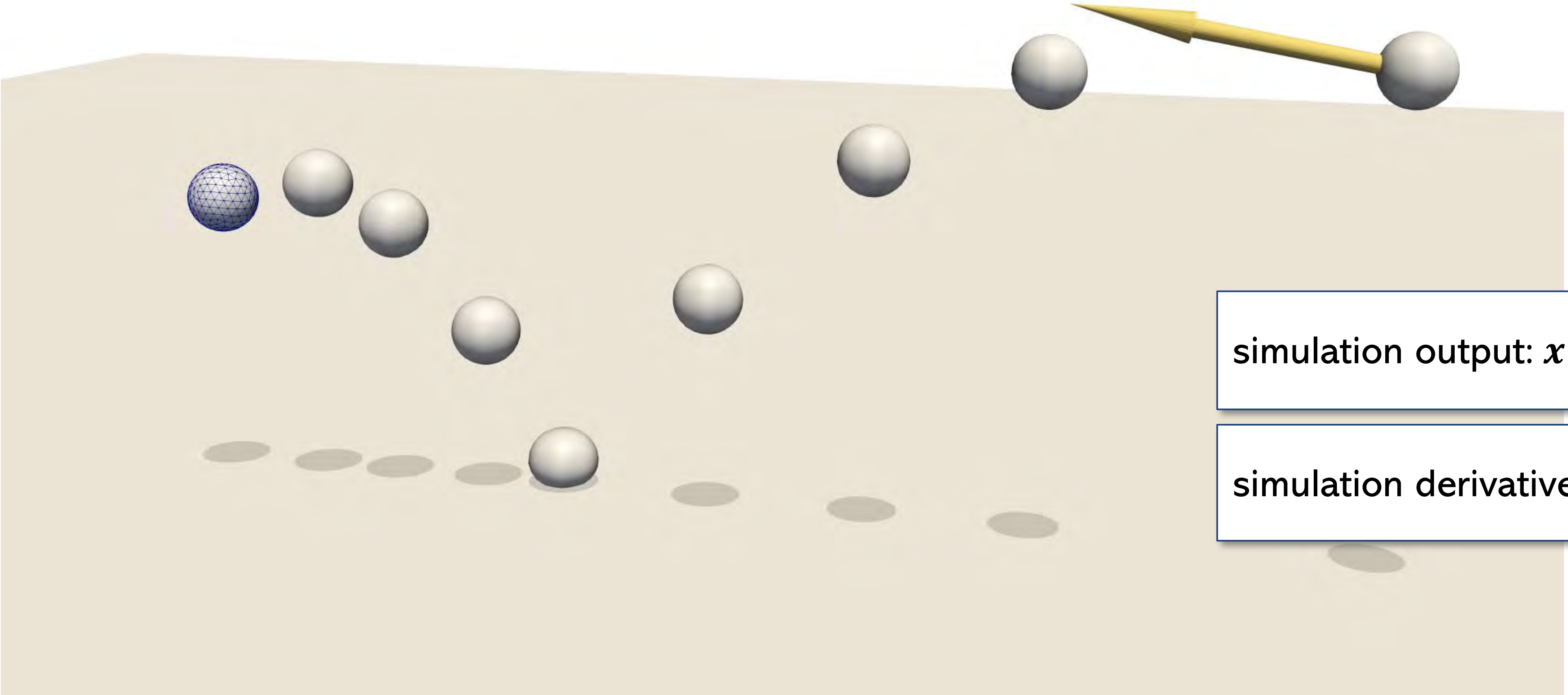
Control as an optimization problem:

$$\min_{\boldsymbol{p}} \underbrace{O(\boldsymbol{x}(\boldsymbol{p}), \boldsymbol{p})}_{}$$

e.g. $\|\boldsymbol{x_n} - \hat{\boldsymbol{x}}\|_2^2$

Gradient: $\frac{dO}{d\boldsymbol{p}} = \frac{\partial O}{\partial \boldsymbol{x}} \frac{d\boldsymbol{x}}{d\boldsymbol{p}} + \frac{\partial O}{\partial \boldsymbol{p}}$

Differentiable Simulation



Briefly, how it works – forward simulation

Start from Newton's 2nd law of motion → $\mathbf{M}\ddot{\boldsymbol{x}} = \boldsymbol{F}(\boldsymbol{x}, \boldsymbol{p})$

Explicit time stepping:
$\mathbf{M}\ddot{\boldsymbol{x}}_k = F(\boldsymbol{x}_{k-1}, \boldsymbol{p})$

Implicit time stepping:
$\mathbf{M}\ddot{\boldsymbol{x}}_k = F(\boldsymbol{x}_k, \boldsymbol{p})$

Use Newton's Method to find $\boldsymbol{x}_k$ such that $\mathbf{M}\ddot{\boldsymbol{x}}_k = F(\boldsymbol{x}_k, \boldsymbol{p})$

where, for example, $\ddot{\boldsymbol{x}}_k \approx \frac{\boldsymbol{x}_k - 2\boldsymbol{x}_{k-1} + \boldsymbol{x}_{k-2}}{h^2}$, $\dot{\boldsymbol{x}}_k \approx \frac{\boldsymbol{x}_k - \boldsymbol{x}_{k-1}}{h}$

Briefly, how it works – forward simulation

Simulation output:

$$\boldsymbol{x} = \begin{bmatrix} \quad \\ \quad \\ \quad \\ \quad \end{bmatrix} \begin{matrix} \rightarrow \\ \rightarrow \\ \\ \rightarrow \end{matrix} \underbrace{\begin{bmatrix} \mathbf{M}\ddot{\boldsymbol{x}}_1 - F(\boldsymbol{x}_1, \boldsymbol{p}) \\ \mathbf{M}\ddot{\boldsymbol{x}}_2 - F(\boldsymbol{x}_2, \boldsymbol{p}) \\ \vdots \\ \mathbf{M}\ddot{\boldsymbol{x}}_n - F(\boldsymbol{x}_n, \boldsymbol{p}) \end{bmatrix}}_{\boldsymbol{G}(\boldsymbol{x}(\boldsymbol{p}), \boldsymbol{p})}$$

Where:

- $\boldsymbol{p}$ is the input driving the simulation
- what we want is $\frac{d\boldsymbol{x}}{d\boldsymbol{p}}$
- $\boldsymbol{x}(\boldsymbol{p})$ does not have an analytic form

But:

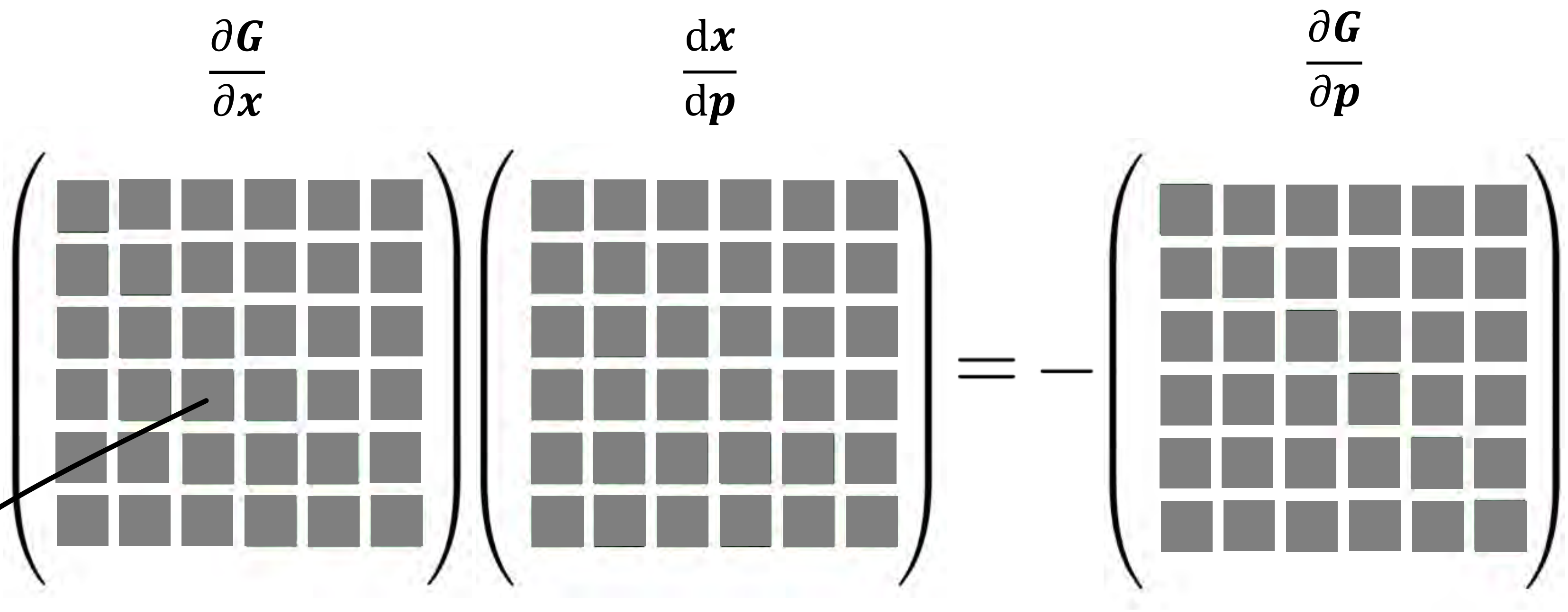
- for *any* $\boldsymbol{p}$, we compute $\boldsymbol{x}(\boldsymbol{p})$ such that $\boldsymbol{G}(\boldsymbol{x}(\boldsymbol{p}), \boldsymbol{p}) = 0$

Briefly, how it works – simulation derivatives

$$\boldsymbol{G}(\boldsymbol{x}(\boldsymbol{p}), \boldsymbol{p}) = \boldsymbol{0}, \forall \boldsymbol{p}$$

$$\frac{d\boldsymbol{G}}{d\boldsymbol{p}} = \boldsymbol{0} = \frac{\partial \boldsymbol{G}}{\partial \boldsymbol{x}}\frac{d\boldsymbol{x}}{d\boldsymbol{p}} + \frac{\partial \boldsymbol{G}}{\partial \boldsymbol{p}}$$

$$\frac{d\boldsymbol{x}}{d\boldsymbol{p}} = -\left(\frac{\partial \boldsymbol{G}}{\partial \boldsymbol{x}}\right)^{-1}\frac{\partial \boldsymbol{G}}{\partial \boldsymbol{p}}$$



$\frac{\partial \boldsymbol{G}_i}{\partial \boldsymbol{x}_j}$ (how does the “F-Ma” residual at time step $\boldsymbol{i}$ change wrt system configuration at time step $\boldsymbol{j}$)

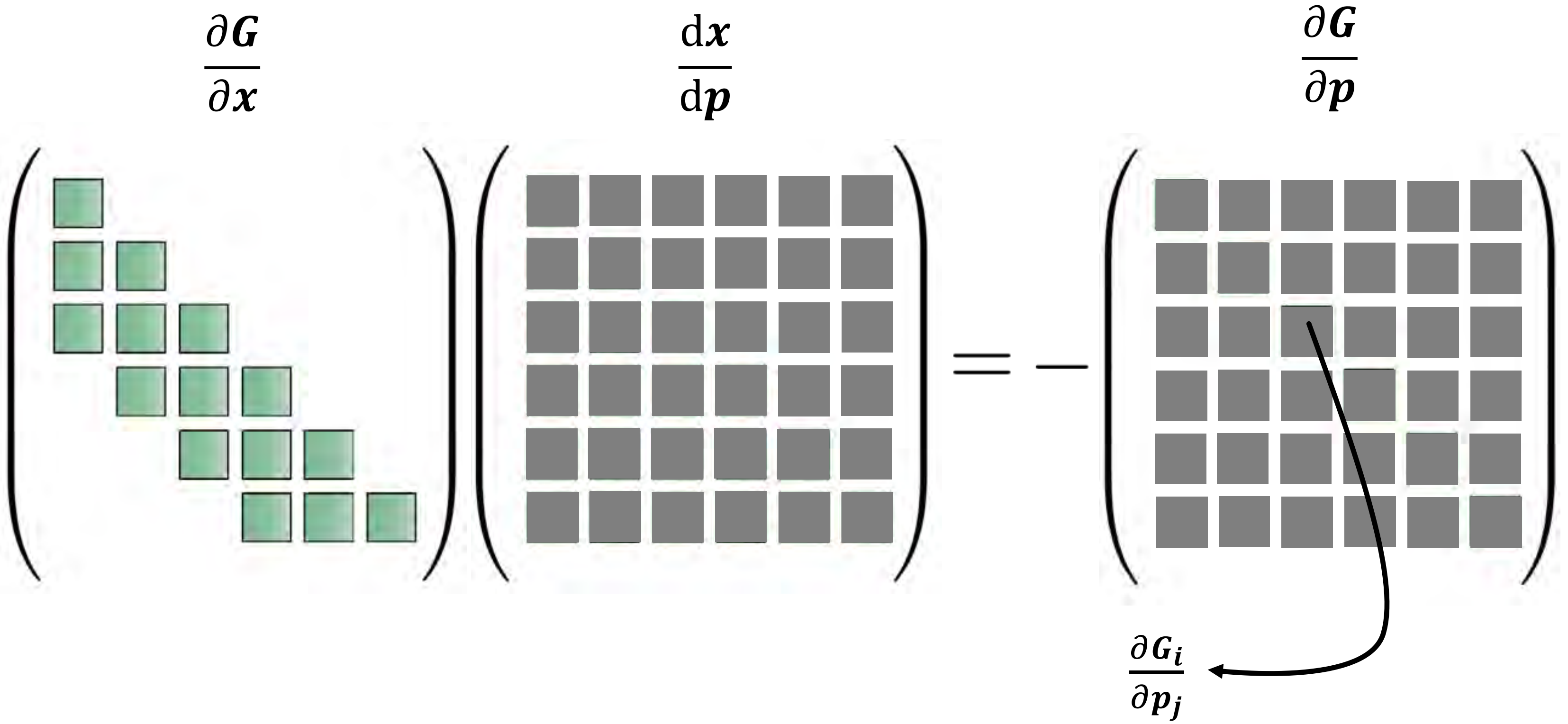
$$\boldsymbol{G}_{\mathrm{k}} = M\frac{\boldsymbol{x}_k - 2\boldsymbol{x}_{k-1} + \boldsymbol{x}_{k-2}}{h^2} - F(\boldsymbol{x}_k, \boldsymbol{p})$$

Briefly, how it works – simulation derivatives

$$\boldsymbol{G}(\boldsymbol{x}(\boldsymbol{p}),\boldsymbol{p}) = \boldsymbol{0}, \forall \boldsymbol{p}$$

$$\frac{d\boldsymbol{G}}{d\boldsymbol{p}} = \boldsymbol{0} = \frac{\partial \boldsymbol{G}}{\partial \boldsymbol{x}}\frac{d\boldsymbol{x}}{d\boldsymbol{p}} + \frac{\partial \boldsymbol{G}}{\partial \boldsymbol{p}}$$

$$\frac{d\boldsymbol{x}}{d\boldsymbol{p}} = -\left(\frac{\partial \boldsymbol{G}}{\partial \boldsymbol{x}}\right)^{-1}\frac{\partial \boldsymbol{G}}{\partial \boldsymbol{p}}$$



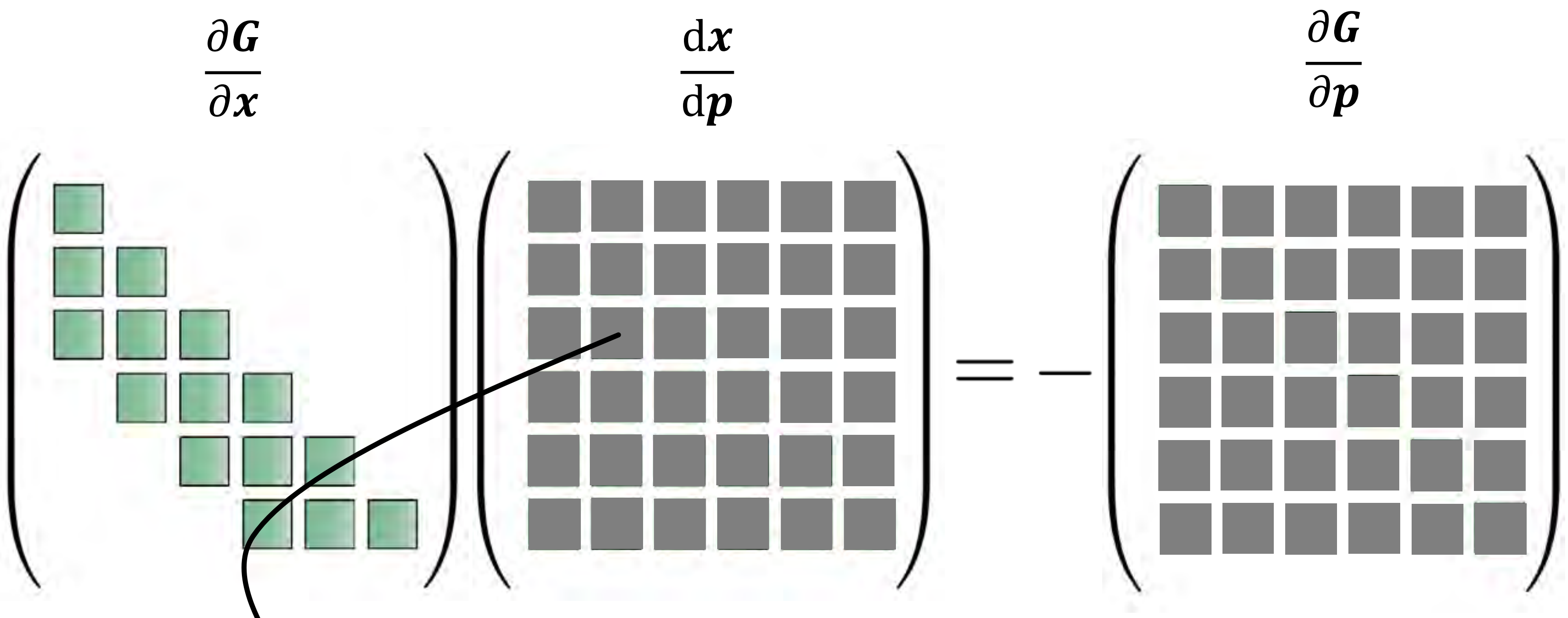
(how does the “F-Ma” residual at time step $\boldsymbol{i}$ change wrt the j[th] input parameter $\boldsymbol{p_j}$)

Briefly, how it works – simulation derivatives

$$\boldsymbol{G}(\boldsymbol{x}(\boldsymbol{p}),\boldsymbol{p}) = \boldsymbol{0}, \forall \boldsymbol{p}$$

$$\frac{d\boldsymbol{G}}{d\boldsymbol{p}} = \boldsymbol{0} = \frac{\partial \boldsymbol{G}}{\partial \boldsymbol{x}}\frac{d\boldsymbol{x}}{d\boldsymbol{p}} + \frac{\partial \boldsymbol{G}}{\partial \boldsymbol{p}}$$

$$\frac{d\boldsymbol{x}}{d\boldsymbol{p}} = -\left(\frac{\partial \boldsymbol{G}}{\partial \boldsymbol{x}}\right)^{-1}\frac{\partial \boldsymbol{G}}{\partial \boldsymbol{p}}$$



$\frac{\mathrm{d}\boldsymbol{x}_i}{\mathrm{d}\boldsymbol{p}_j}$ (how does the system's configuration at time step $\boldsymbol{i}$ change with respect to the $\boldsymbol{j}^{\text{th}}$ input parameter)

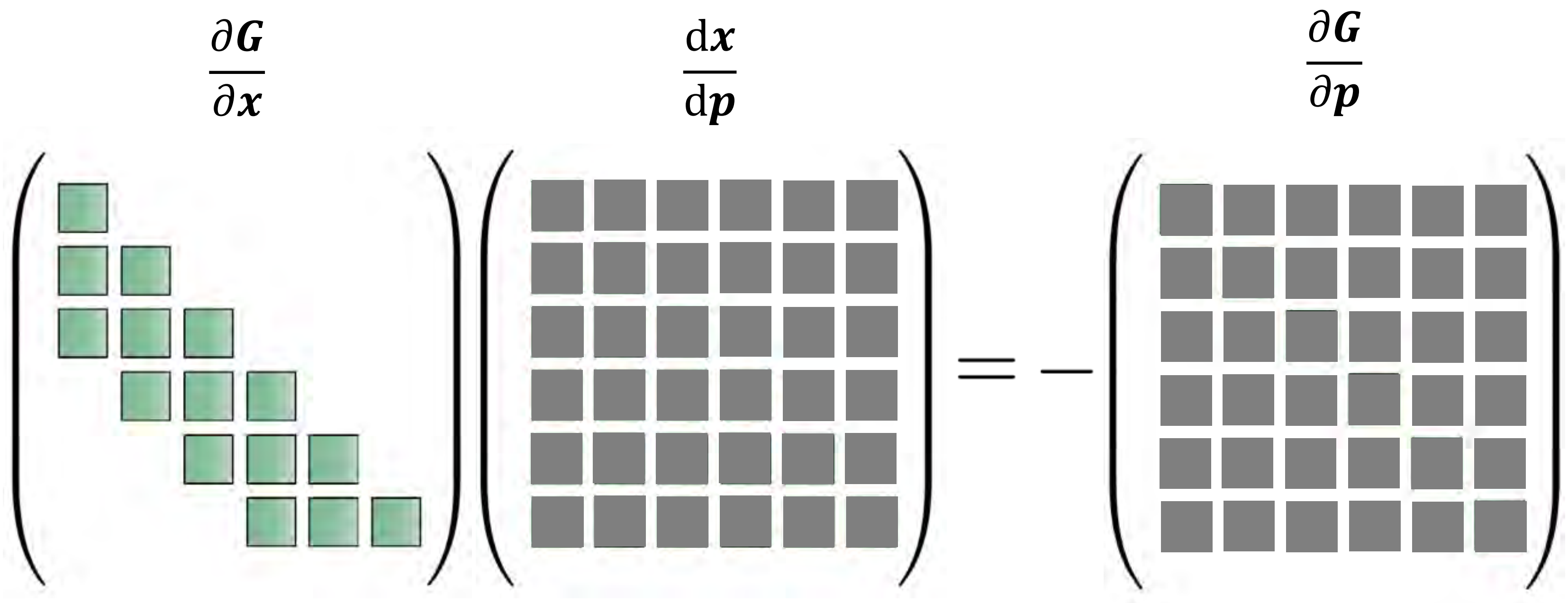
CRL

Briefly, how it works – simulation derivatives

$$\boldsymbol{G}(\boldsymbol{x}(\boldsymbol{p}),\boldsymbol{p}) = \boldsymbol{0}, \forall \boldsymbol{p}$$

$$\frac{d\boldsymbol{G}}{d\boldsymbol{p}} = \boldsymbol{0} = \frac{\partial \boldsymbol{G}}{\partial \boldsymbol{x}}\frac{d\boldsymbol{x}}{d\boldsymbol{p}} + \frac{\partial \boldsymbol{G}}{\partial \boldsymbol{p}}$$

$$\frac{d\boldsymbol{x}}{d\boldsymbol{p}} = -\left(\frac{\partial \boldsymbol{G}}{\partial \boldsymbol{x}}\right)^{-1}\frac{\partial \boldsymbol{G}}{\partial \boldsymbol{p}}$$



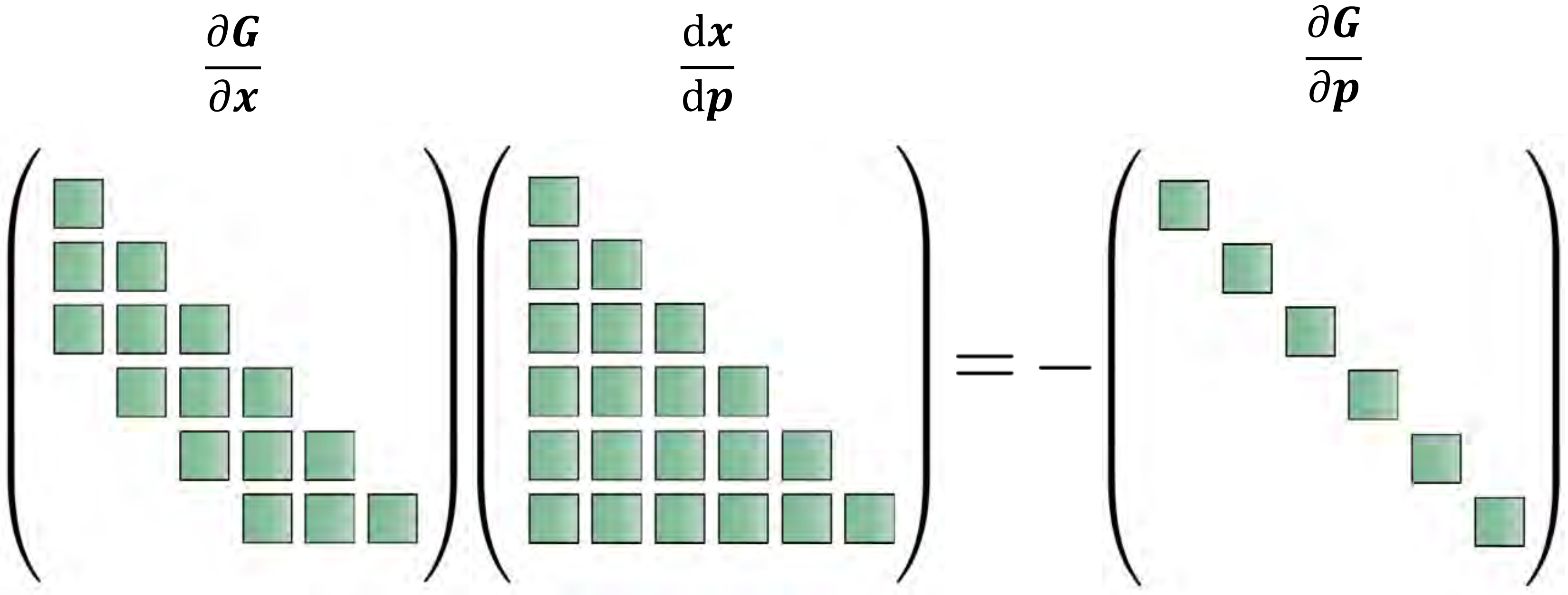
Note: the sparsity structure of $\frac{\partial \boldsymbol{G}}{\partial \boldsymbol{p}}$ (and $\frac{\mathrm{d}\boldsymbol{x}}{\mathrm{d}\boldsymbol{p}}$) depends on the type of problem we are solving. Specialized solvers that exploit this structure can easily be developed.

Briefly, how it works – simulation derivatives

$$\boldsymbol{G}(\boldsymbol{x}(\boldsymbol{p}),\boldsymbol{p}) = \boldsymbol{0}, \forall \boldsymbol{p}$$

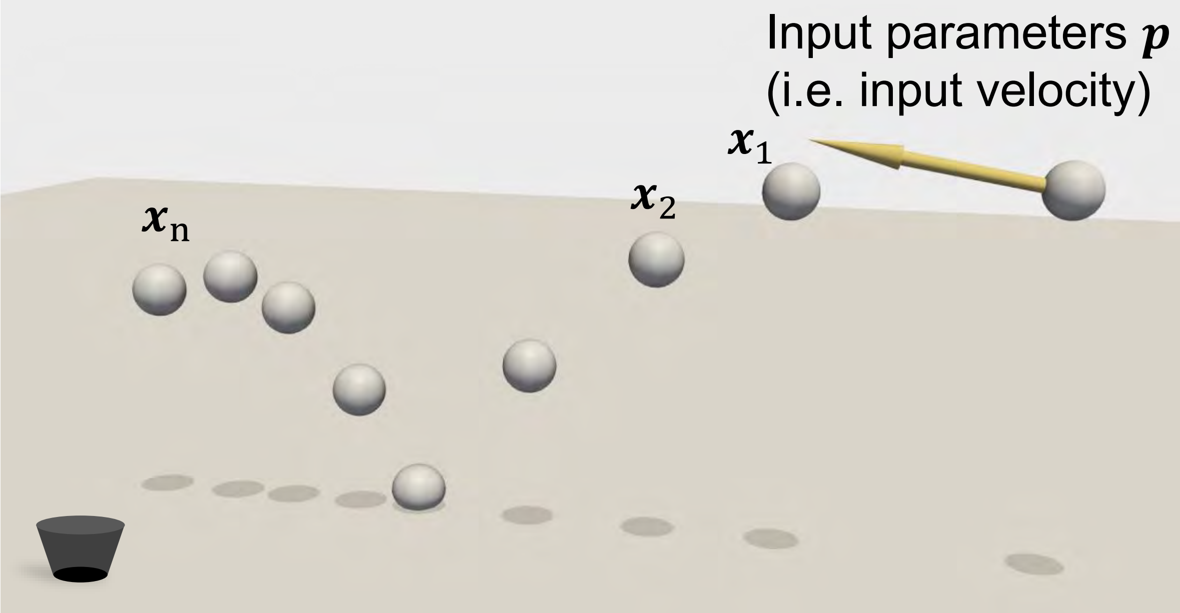
$$\frac{d\boldsymbol{G}}{d\boldsymbol{p}} = \boldsymbol{0} = \frac{\partial \boldsymbol{G}}{\partial \boldsymbol{x}}\frac{d\boldsymbol{x}}{d\boldsymbol{p}} + \frac{\partial \boldsymbol{G}}{\partial \boldsymbol{p}}$$

$$\frac{d\boldsymbol{x}}{d\boldsymbol{p}} = -\left(\frac{\partial \boldsymbol{G}}{\partial \boldsymbol{x}}\right)^{-1}\frac{\partial \boldsymbol{G}}{\partial \boldsymbol{p}}$$



Note: the sparsity structure of $\frac{\partial \boldsymbol{G}}{\partial \boldsymbol{p}}$ (and $\frac{\mathrm{d}\boldsymbol{x}}{\mathrm{d}\boldsymbol{p}}$) depends on the type of problem we are solving. Specialized solvers that exploit this structure can easily be developed.

Briefly, how it works – the inverse simulation loop



Until convergence

$$\Delta\mathbf{p} = -\left(\frac{\partial O}{\partial x}\underbrace{\frac{dx}{dp}}_{\left(\frac{\partial \boldsymbol{G}}{\partial \boldsymbol{x}}\right)^{-1}\frac{\partial \boldsymbol{G}}{\partial \boldsymbol{p}}} + \frac{\partial O}{\partial p}\right)$$

$$\alpha = \text{line\_search}(\Delta\mathbf{p})$$

$$\mathbf{p} = \mathbf{p} + \alpha\Delta\mathbf{p};$$

$$\mathbf{x} = \text{simulate}(\mathbf{x}, \mathbf{p})$$

end

Computing search directions

- Gradient descent: $\Delta \boldsymbol{p} = -\frac{dO}{d\boldsymbol{p}} = -\left(\frac{\partial O}{\partial \boldsymbol{x}}\boldsymbol{S} + \frac{\partial O}{\partial \boldsymbol{p}}\right)$, where $\boldsymbol{S} := \frac{d\boldsymbol{x}}{d\boldsymbol{p}}$

- Newton's method: $\Delta \boldsymbol{p} = -H^{-1}\frac{dO}{d\boldsymbol{p}}$, where

$$H = \frac{d^2O}{d\boldsymbol{p}^2} = \cancel{\frac{\partial O}{\partial \boldsymbol{x}}\left(\boldsymbol{S}^T\frac{\partial \boldsymbol{S}}{\partial \boldsymbol{x}} + \frac{\partial \boldsymbol{S}}{\partial \boldsymbol{p}}\right)} + \boldsymbol{S}^T\frac{\partial^2 O}{\partial \boldsymbol{x}^2}\boldsymbol{S} + \cancel{\boldsymbol{S}^T\frac{\partial^2 O}{\partial \boldsymbol{x}\partial \boldsymbol{p}}} + \cancel{\frac{\partial^2 O}{\partial \boldsymbol{x}\partial \boldsymbol{p}}^T\boldsymbol{S}} + \frac{\partial^2 O}{\partial \boldsymbol{p}^2}$$

- Generalized Gauss-Newton: $\Delta \boldsymbol{p} = -\widetilde{H}^{-1}\frac{dO}{d\boldsymbol{p}}$, where

$$\widetilde{H} = \boldsymbol{S}^T\frac{\partial^2 O}{\partial \boldsymbol{x}^2}\boldsymbol{S} + \frac{\partial^2 O}{\partial \boldsymbol{p}^2}$$

Gauss-Newton: dense linear system solve

$$\widetilde{H} = \boldsymbol{S}^T \frac{\partial^2 O}{\partial \boldsymbol{x}^2} \boldsymbol{S} + \frac{\partial^2 O}{\partial \boldsymbol{p}^2}$$

$$\widetilde{H} \Delta \boldsymbol{p} = -\frac{dO}{d\boldsymbol{p}}$$

Gauss-Newton – sparse reformulation

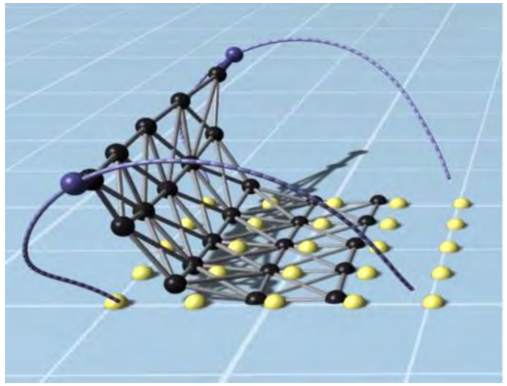
$$\begin{bmatrix} \frac{\partial^2 O}{\partial \boldsymbol{x}^2} & 0 & \frac{\partial \boldsymbol{G}}{\partial \boldsymbol{x}}^T \\ 0 & \frac{\partial^2 O}{\partial \boldsymbol{p}^2} & \frac{\partial \boldsymbol{G}}{\partial \boldsymbol{p}}^T \\ \frac{\partial \boldsymbol{G}}{\partial \boldsymbol{x}} & \frac{\partial \boldsymbol{G}}{\partial \boldsymbol{p}} & \boldsymbol{0} \end{bmatrix} \begin{bmatrix} \Delta \boldsymbol{x} \\ \Delta \boldsymbol{p} \\ \Delta \boldsymbol{\lambda} \end{bmatrix} = \begin{bmatrix} \boldsymbol{0} \\ -\frac{dO}{d\boldsymbol{p}} \\ \boldsymbol{0} \end{bmatrix}$$

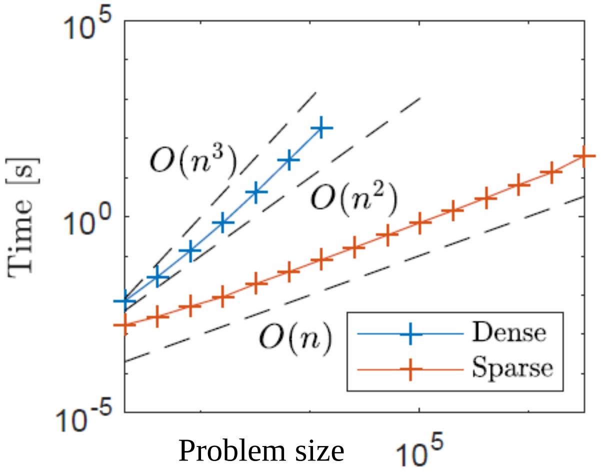
Sparse Gauss Newton for accelerated sensitivity analysis

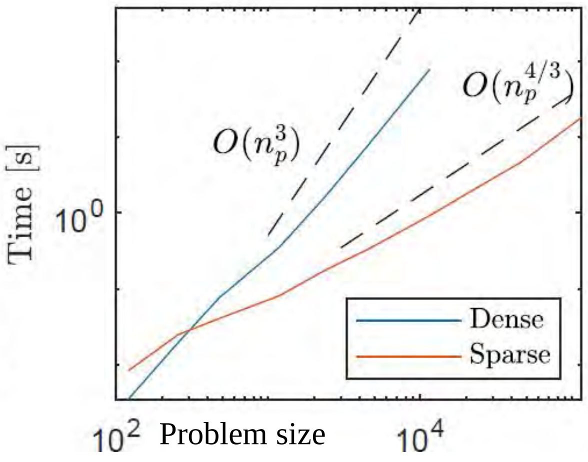
[Zehnder et al., Trans. on Graphics, 2021]

- *Dense and sparse linear systems give same search direction* $\Delta\boldsymbol{p}$
- *Sparse system leads to asymptotically better performance*
- *Formal connection between Sensitivity Analysis and the method of Lagrange Multipliers*

Sparse Gauss Newton for accelerated sensitivity analysis







[Zehnder et al., TOG 2021]

DIFFERENTIABLE SIMULATION: COMPUTING DERIVATIVES

Miles Macklin

GOAL

- Minimize a scalar loss function s() w.r.t system parameters $\mathbf{x}(t_0)$

$$s\left(\mathbf{x}(t_1)\right) = s\left(\mathbf{x}(t_0) + \int_{t_0}^{t_1} f(\mathbf{x}(t))dt\right)$$

System State:

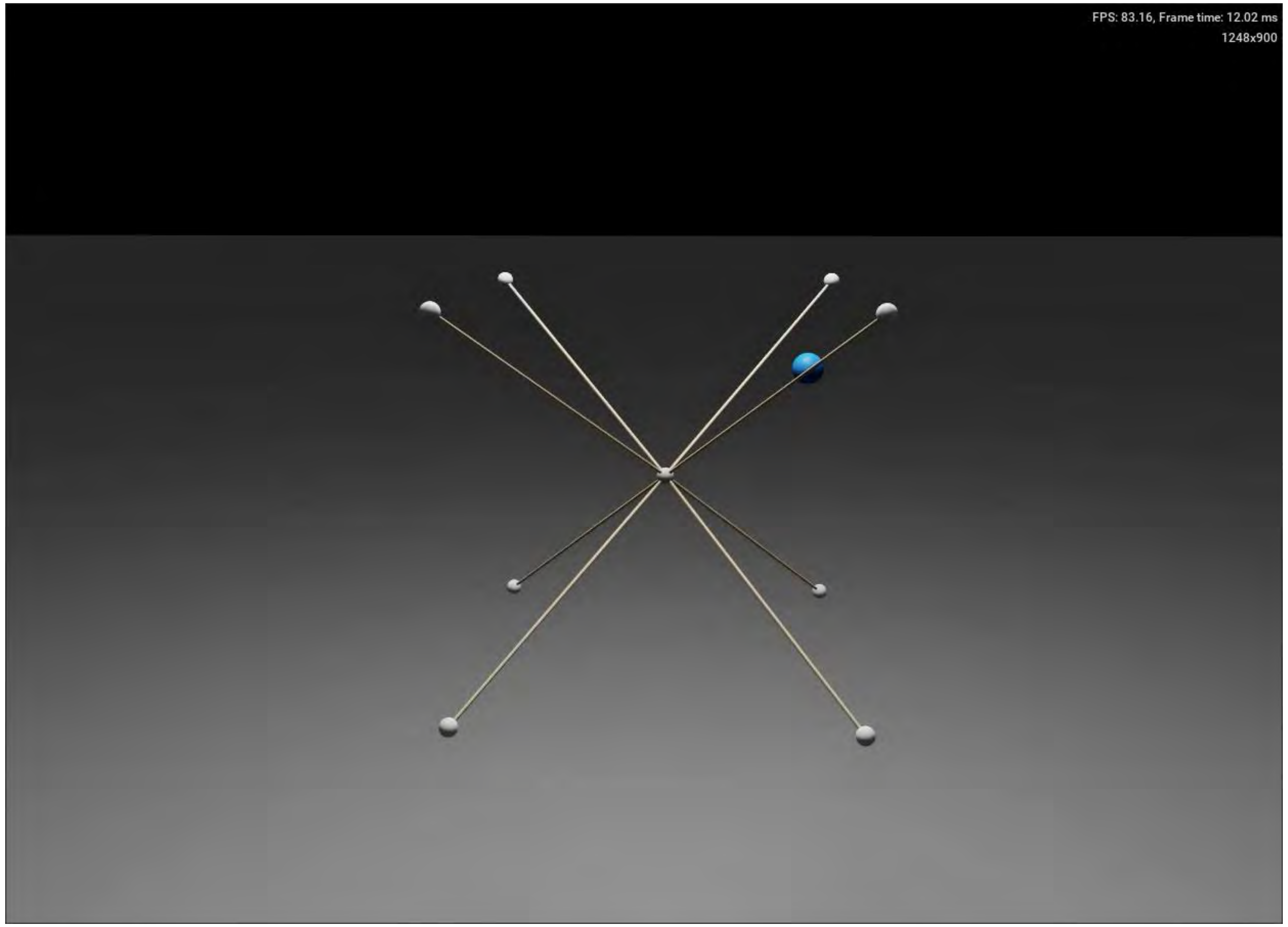
$$\mathbf{x}(t) = \begin{bmatrix} \mathbf{q} \\ \dot{\mathbf{q}} \\ \theta \end{bmatrix}$$

Forward ODE:

$$\dot{\mathbf{x}}(t) = f\left(\mathbf{x}(t)\right)$$

EXAMPLE - MASS SPRING CAGE

- Minimize distance of center particle to target after 2 sec
- Optimize over spring rest lengths
- 3-4 LBFGS iterations



AUTO DIFFERENTIATION

COMPUTING GRADIENTS

- For optimization we want the gradient of scalar loss $s(\mathbf{x})$ at $t = t_0$
- Define the adjoint of a variable as $\mathbf{x}^*$
- Goal: given $\mathbf{x}^*(t_1)$ compute $\mathbf{x}^*(t_0)$

Adjoint Variable

$$\mathbf{x}^*(t) = \frac{\partial s}{\partial \mathbf{x}}^T = \begin{bmatrix} \frac{\partial s}{\partial x_1} \\ \vdots \\ \frac{\partial s}{\partial x_n} \end{bmatrix}$$

$$\mathbf{x}, \mathbf{x}^* \in \mathbb{R}^n$$

CONTINUOUS ADJOINT METHOD

- Computes gradient of scalar loss function via. reverse ODE

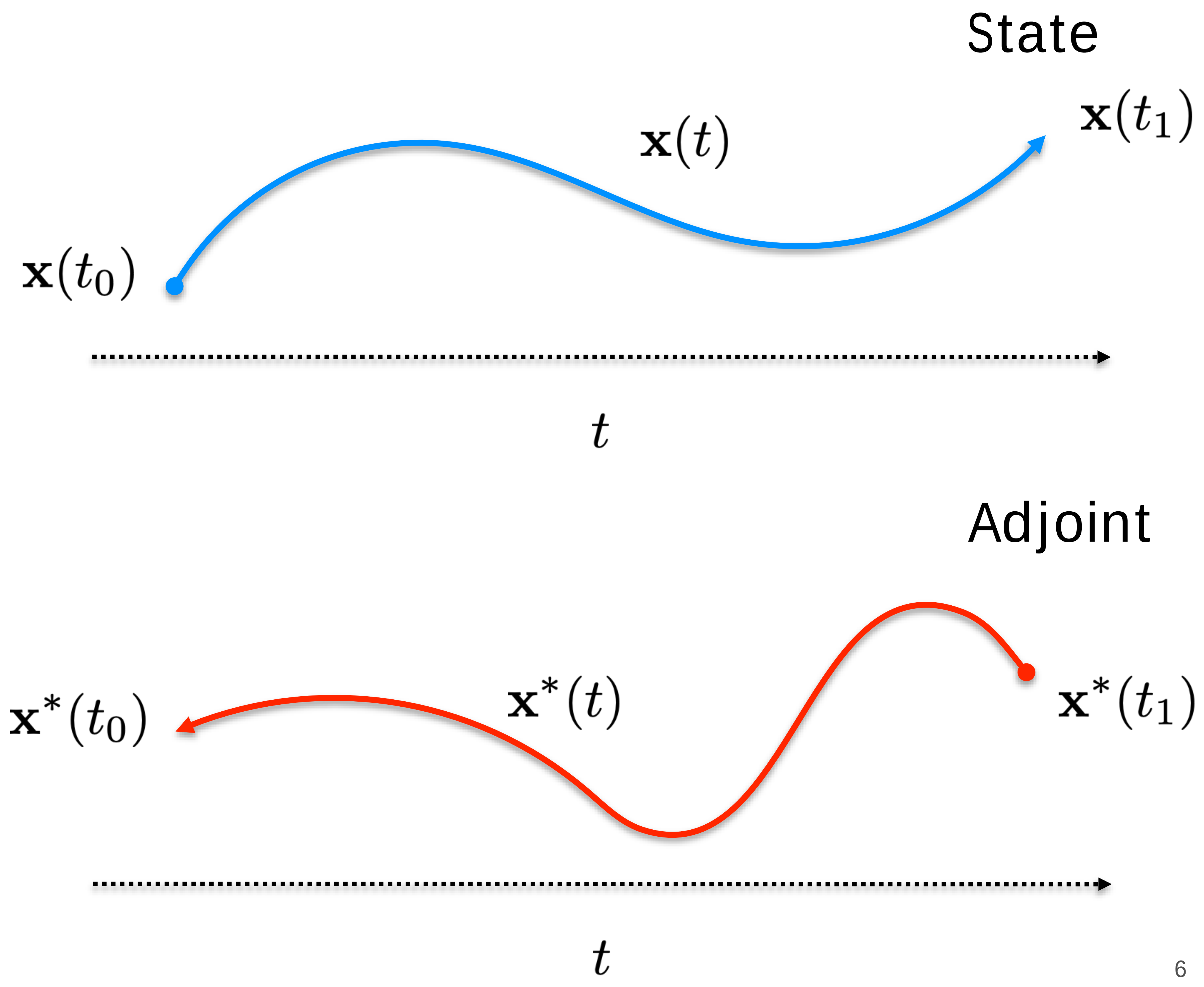
Forward ODE:

$$\dot{\mathbf{x}}(t) = f(\mathbf{x}(t))$$

Calculus of Variations

Reverse ODE:

$$\dot{\mathbf{x}}^*(t) = -\frac{\partial f}{\partial \mathbf{x}}^T \mathbf{x}^*(t)$$



DISCRETE ADJOINT METHOD

- Replace ODE with time-stepping equations:

$$\mathbf{x}^{t+1} = f(\mathbf{x}^t)$$

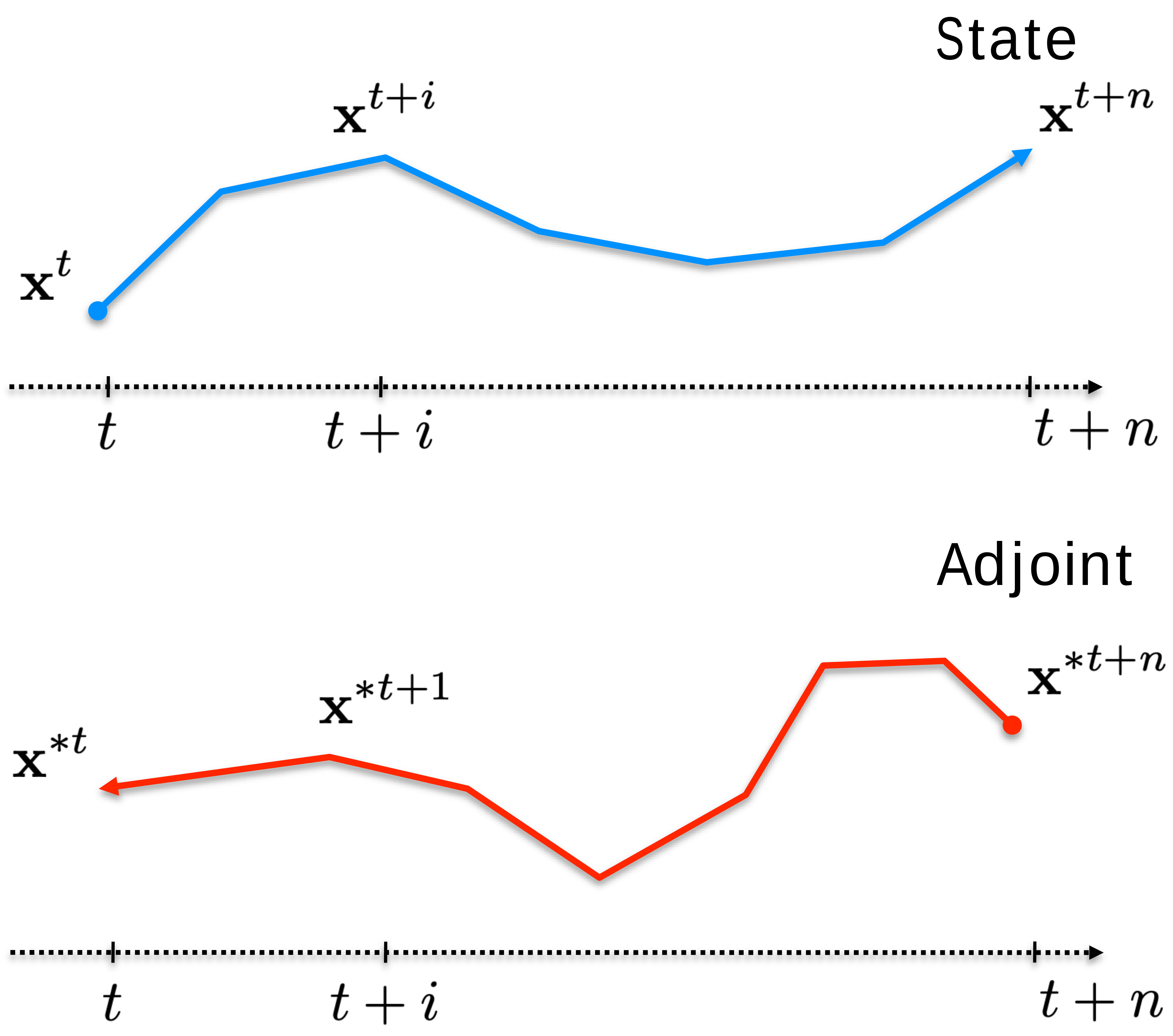
- Discrete trajectory + loss:

$$s(\mathbf{x}^{t+n}) = s(f(f(f(\mathbf{x}^t))$$

- Apply chain rule:

$$\mathbf{x}^{*t} = \frac{\partial s}{\partial \mathbf{x}}\bigg|_{t+0}^{T} = \frac{\partial f}{\partial \mathbf{x}}\bigg|_{t+0}^{T} \cdot \frac{\partial f}{\partial \mathbf{x}}\bigg|_{t+1}^{T} \cdot \frac{\partial f}{\partial \mathbf{x}}\bigg|_{t+2}^{T} \cdot \frac{\partial s}{\partial \mathbf{x}}\bigg|_{t+3}^{T}$$

- Two ways to evaluate the chain rule..



FORWARD ACCUMULATION (TANGENT MODE)

- Forward:

$$\frac{\partial s(f(g(x)))}{\partial x} = \frac{\partial s}{\partial f}\left(\frac{\partial f}{\partial g}\frac{\partial g}{\partial x}\right)$$

$$s : \mathbb{R}^n \to \mathbb{R}$$
$$f : \mathbb{R}^m \to \mathbb{R}^n$$
$$g : \mathbb{R}^p \to \mathbb{R}^m$$

$$\boxed{\mathbb{R}^{1\times p}} = \boxed{\mathbb{R}^{1\times n}} \cdot \left(\boxed{\mathbb{R}^{n\times m}} \cdot \boxed{\mathbb{R}^{m\times p}}\right)$$

- Evaluate inside->out
- Simple, but large matrix multiplies are expensive
- Use forward mode when outputs >> params (e.g.: vector valued loss)

REVERSE ACCUMULATION (ADJOINT MODE)

- Reverse:

$$\frac{\partial s(f(g(x)))}{\partial x} = \left(\frac{\partial s}{\partial f}\frac{\partial f}{\partial g}\right)\frac{\partial g}{\partial x}$$

$$s : \mathbb{R}^n \to \mathbb{R}$$
$$f : \mathbb{R}^m \to \mathbb{R}^n$$
$$g : \mathbb{R}^p \to \mathbb{R}^m$$

$$\boxed{\mathbb{R}^{1\times p}} = \left(\boxed{\mathbb{R}^{1\times n}} \cdot \boxed{\mathbb{R}^{n\times m}}\right) \cdot \boxed{\mathbb{R}^{m\times p}}$$

- Evaluate outside->in
- Use reverse mode when outputs << params (e.g.: scalar valued loss)

CONTINUOUS/DISCRETE SIDE-BY-SIDE

Continuous Loss:

$$s\left(\mathbf{x}(t_0) + \int_{t_0}^{t_1} f(\mathbf{x}(t))dt\right)$$

Forward ODE:

$$\dot{\mathbf{x}}(t) = f(\mathbf{x}(t))$$

Reverse ODE:

$$\dot{\mathbf{x}}^*(t) = -\frac{\partial f}{\partial \mathbf{x}}^T \mathbf{x}^*(t)$$

Discrete Loss:

$$s(\mathbf{x}^{t+n}) = s(f(f(f(\mathbf{x}^t)))$$

Forward Time-stepping:

$$\mathbf{x}^{t+1} = f(\mathbf{x}^t)$$

Reverse Time-stepping:

$$\mathbf{x^*}^{t-1} = \frac{\partial f}{\partial \mathbf{x}}^T \mathbf{x^*}^t$$

ADJOINT OF A FUNCTION

- Given a function:

$$f(\mathbf{x}, \mathbf{y}) \to \mathbf{z}$$

- Define adjoint (*) as follows:

Adjoint Variable: $\mathbf{z}^* = \frac{\partial s}{\partial \mathbf{z}}^T$

$$f^*(\mathbf{x}, \mathbf{y}, \mathbf{z}^*) \to (\mathbf{x}^*, \mathbf{y}^*)$$

$$f^*(\mathbf{x}, \mathbf{y}, \mathbf{z}^*) \equiv \left(\frac{\partial f}{\partial \mathbf{x}}^T \mathbf{z}^*, \frac{\partial f}{\partial \mathbf{y}}^T \mathbf{z}^* \right)$$

- Adjoint returns derivative of scalar loss with respect to function inputs

ADJOINT EXAMPLES

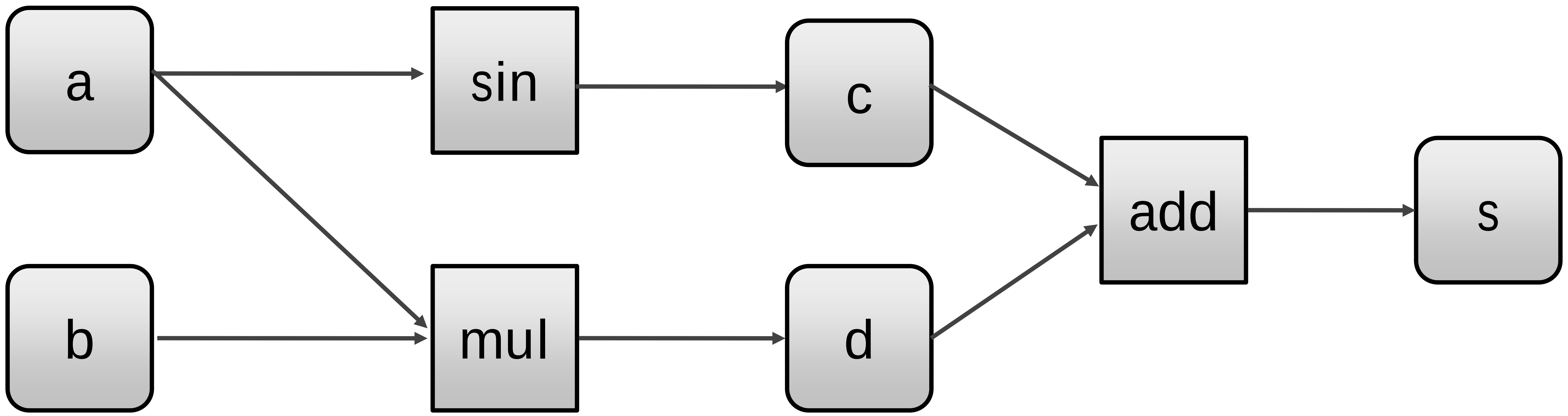
$$z = f(x, y) = x + y \qquad f^*(x, y, z^*) = [z^*, z^*]$$

$$z = f(x, y) = xy \qquad f^*(x, y, z^*) = [yz^*, xz^*]$$

$$z = f(x) = sin(x) \qquad f^*(x, z^*) = [cos(x)z^*]$$

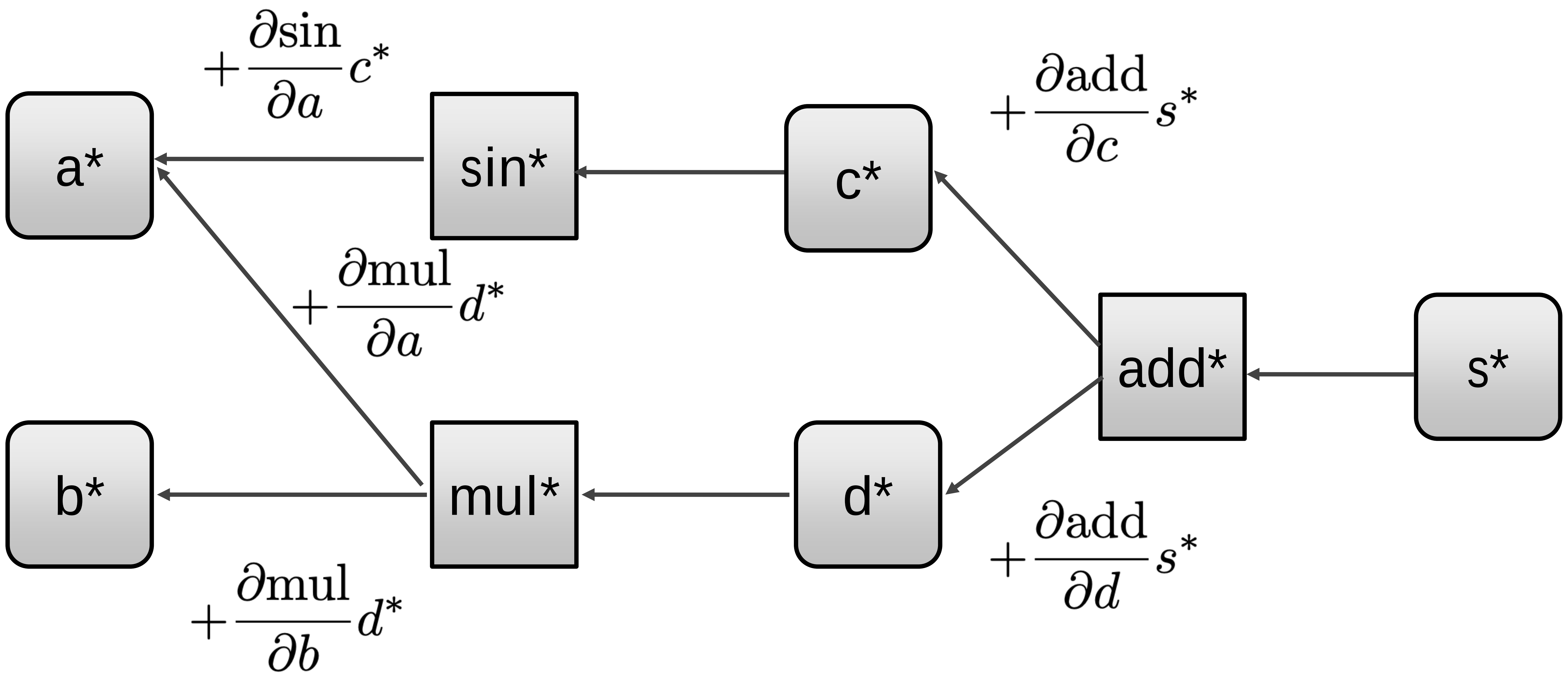
REVERSE MODE AUTO. DIFF

- Example: $s(a, b) = \sin(a) + ab$
- Forward evaluation graph:



REVERSE MODE AUTO. DIFF

- Example: $s(a,b) = \sin(a) + ab$



Solution:

$$\frac{\partial s}{\partial a} = \cos(a) + b$$
$$\frac{\partial s}{\partial b} = a$$

Adjoint Variables:

$$a^* = \frac{\partial s}{\partial a}^T$$

Seed Variable:

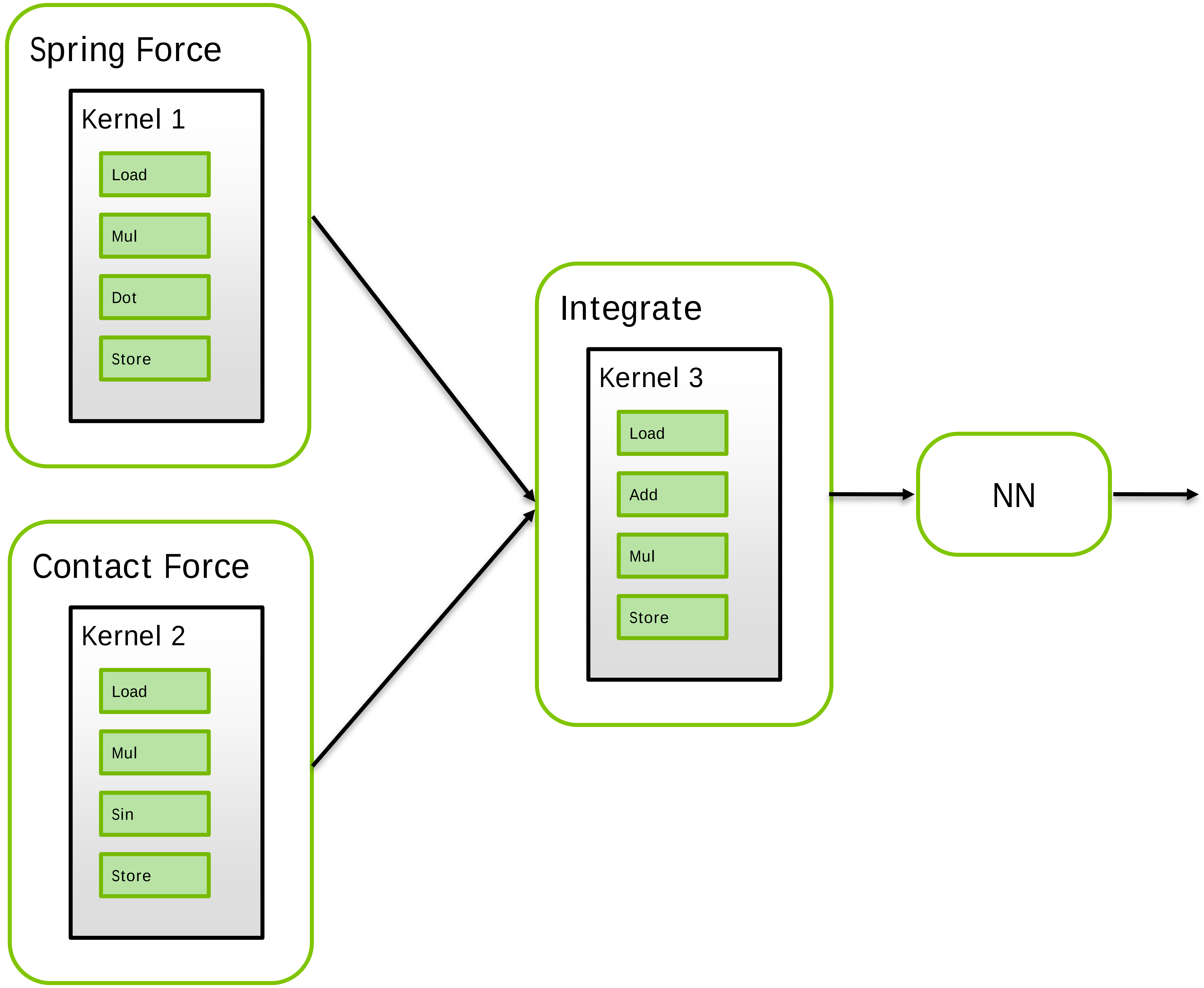
$$s^* = \frac{\partial s}{\partial s}^T = 1$$

AUTODIFF FRAMEWORKS

- Graph Evaluation
  - Runtime
  - Functional, tensor centric
  - PyTorch, TensorFlow
- Program Transformation
  - Compile time
  - Imperative, thread centric
  - DiffTaichi, Google Tangent, Tapenade, dFlex
- Symbolic
  - Expression rewriting
  - Matlab, Mathematica, Maple

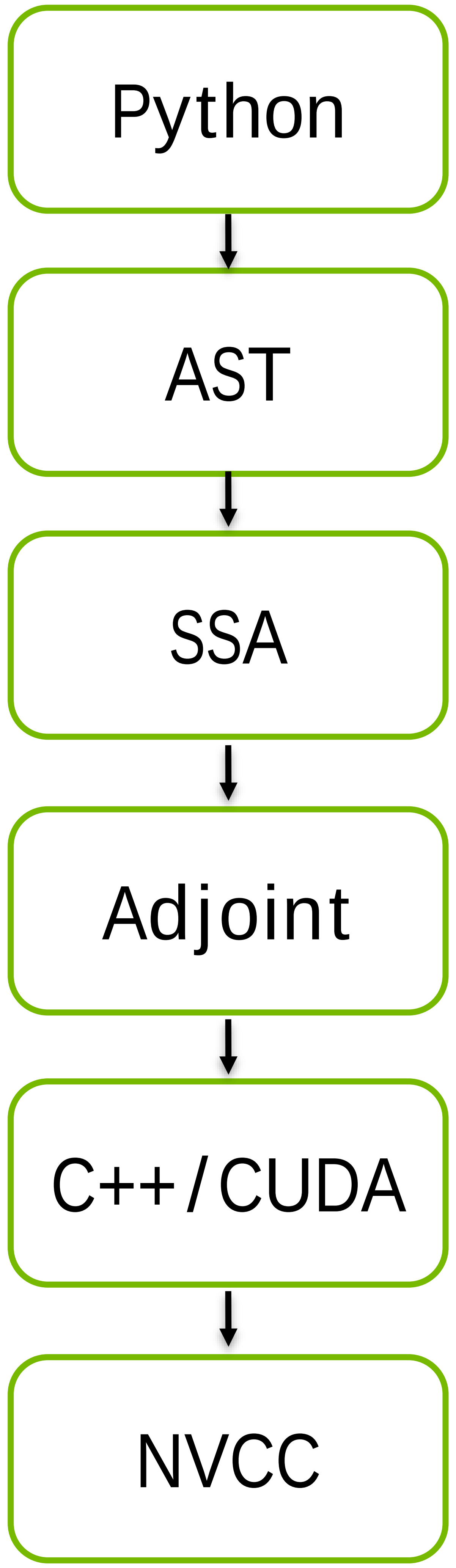
3-LEVEL AUTO DIFF

- Top level
  - computation graph + tape
  - e.g.: loss functions, NN model
- Middle level
  - forward/backward kernels
  - e.g.: force evaluation
- Bottom level
  - mathematical primitives
  - e.g.: sin, cos, dot, cross, etc



PROGRAM TRANSFORMATION

- Middle level auto-diff
- Given abstract syntax tree generate a function's adjoint:
  - Traverse tree (`import ast`)
  - Convert to static single assignment (SSA) form
  - Run function forward (recording state)
  - Run function backward (accumulate gradients)



SIMPLE EXAMPLE

- Python->C++ SSA
- State is local in registers
- Kernel fusion is implicit
- Flexible indexing
- Gather/Scatter Ops
- Runtime JIT compilation

```
@df.func
def simple(a : float, b: float):
   return df.sin(a) + a*b
```

```
void simple_reverse(
   float var_a,
   float var_b,
   float& adj_a,
   float& adj_b,
   float adj_s)
{
   //---------
   // dual vars
   float adj_0 = 0.0f;
   float adj_1 = 0.0f;
   float adj_2 = 0.0f;
   //---------
   // forward
   float var_0 = df::sin(var_a);
   float var_1 = df::mul(var_a, var_b);
   float var_2 = df::add(var_0, var_1);
   //---------
   // reverse
   df::adj_add(var_0, var_1, adj_0, adj_1, adj_2);
   df::adj_mul(var_a, var_b, adj_a, adj_b, adj_1);
   df::adj_sin(var_a, adj_a, adj_s);
}
```

COMPLEX KERNEL

- Triangle bending force kernel
- Many nested expressions
- Random access loads/stores
- Atomic-add accumulation

```
@df.kernel
def eval_bending(x : df.tensor(df.float3),
            v : df.tensor(df.float3),
            indices : df.tensor(int),
            rest : df.tensor(float),
            ke : float,
            kd : float,
            f : df.tensor(df.float3)):

    tid = df.tid()

    # triangle indices
    i = df.load(indices, tid*4+0)
    j = df.load(indices, tid*4+1)
    k = df.load(indices, tid*4+2)
    l = df.load(indices, tid*4+3)

    rest_angle = df.load(rest, tid)

    # load positions
    x1 = df.load(x, i)
    x2 = df.load(x, j)
    x3 = df.load(x, k)
    x4 = df.load(x, l)

    # load velocities
    v1 = df.load(v, i)
    v2 = df.load(v, j)
    v3 = df.load(v, k)
    v4 = df.load(v, l)

    n1 = df.cross(x3-x1, x4-x1)
    n2 = df.cross(x4-x2, x3-x2)

    n1_length = df.length(n1)
    n2_length = df.length(n2)

    rcp_n1 = 1.0/n1_length
    rcp_n2 = 1.0/n2_length
    cos_theta = df.dot(n1, n2)*rcp_n1*rcp_n2

    n1 = n1*rcp_n1*rcp_n1
    n2 = n2*rcp_n2*rcp_n2

    e = x4-x3
    e_hat = df.normalize(e)
    e_length = df.length(e)

    s = df.sign(df.dot(df.cross(n2, n1), e_hat))
    angle = df.acos(cos_theta)*s

    d1 = n1*e_length
    d2 = n2*e_length
    d3 = n1*df.dot(x1-x4, e_hat) + n2*df.dot(x2-x4, e_hat)
    d4 = n1*df.dot(x3-x1, e_hat) + n2*df.dot(x3-x2, e_hat)

    # elastic forces
    f_elastic = ke*(angle - rest_angle)

    # damping forces
    f_damp = kd*(df.dot(d1, v1) + df.dot(d2, v2) + df.dot(d3, v3) + df.dot(d4, v4))

    # total force, proportional to edge length
    f_total = 0.0 - e_length*(f_elastic + f_damp)
    df.atomic_add(f, i, d1*f_total)
    df.atomic_add(f, j, d2*f_total)
    df.atomic_add(f, k, d3*f_total)
    df.atomic_add(f, l, d4*f_total)
```

PROGRAM TRANSFORMATION - ALGORITHM

- Given an AST
- Recursive program to generate the adjoint of a function
- Assume that each node knows how to compute f, f*

```
forward_ops = []
reverse_ops = []

eval(AST.Node):

  # evaluate inputs
  inputs = []
  for each input i in node.f:
    inputs.push_back(eval(i))

  # output
  forward_ops.push_back{var_n = node.f(inputs)}
  reverse_ops.push_front{[adj_j, adj_k, …] += node.fadj(inputs)}

  return var_n
```

VERIFICATION

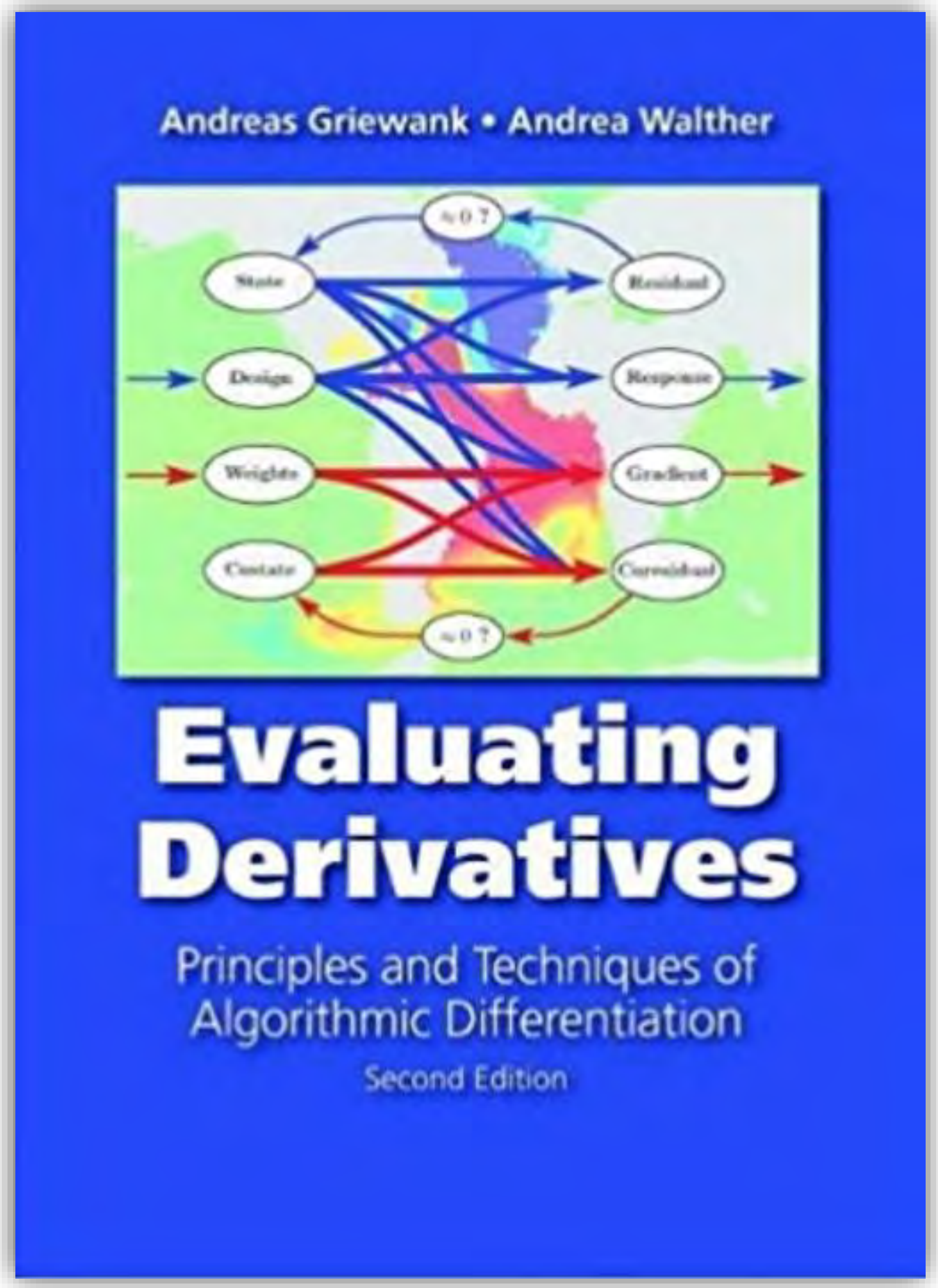
- Check gradients via. finite differencing
- torch.autograd.gradcheck
- Call function adjoint with each basis vector to evaluate full Jacobian

$$\mathbf{J}_i^T = f^*(\delta_i)$$

- n calls, one for each output

FURTHER READING

- [Griewank & Walther 2008]
- Covers program transformation approach in detail
- Many more optimizations possible
- I rely on NVCC to do the heavy lifting



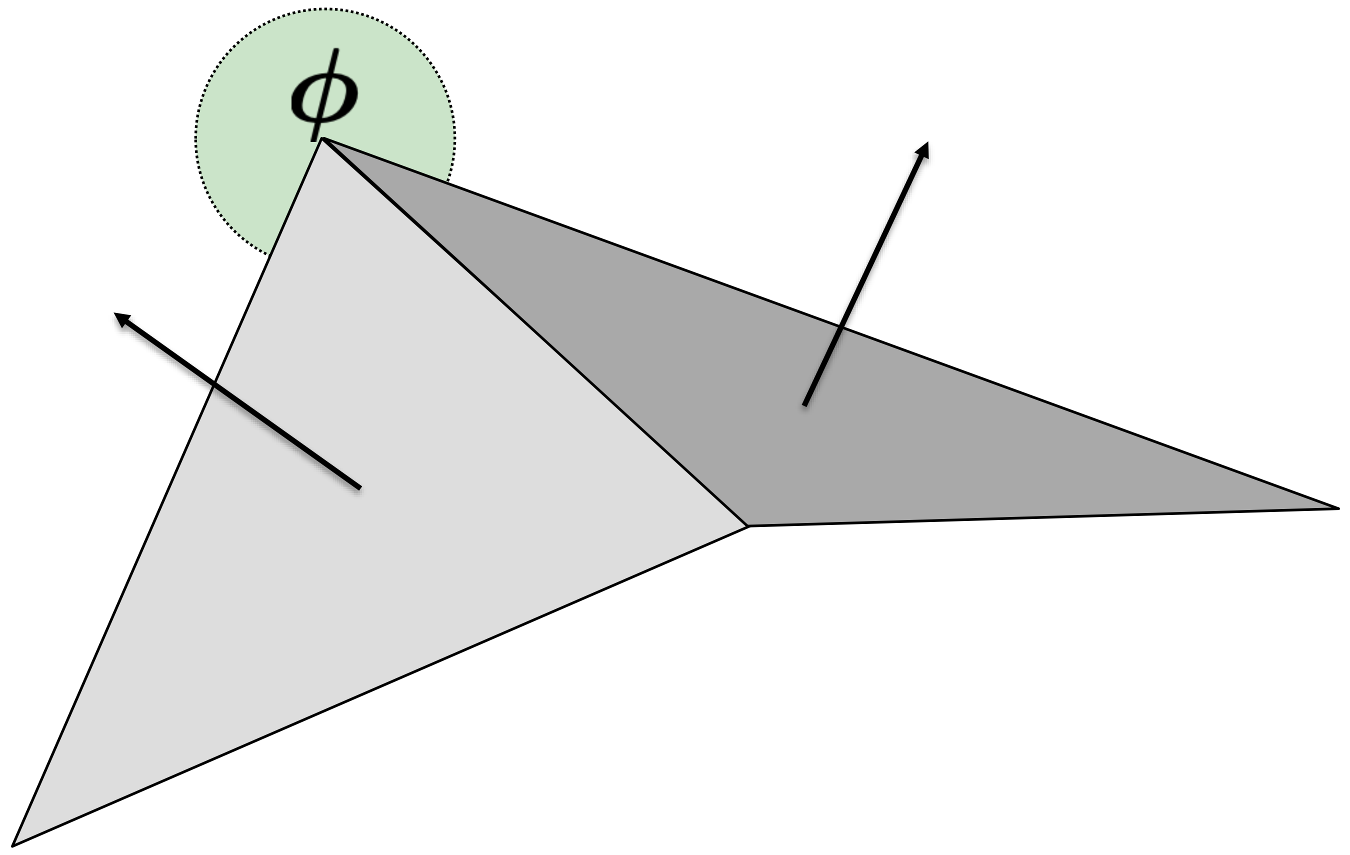
SIMULATION

OPTIMIZATION

- Once we have gradients we can optimize using our favourite numerical method, e.g.:
  - Gradient descent (stochastic, accelerated, etc)
  - Quasi-Newton (LBFGS, Hessian approximating, etc)
  - Conjugate Gradient (Krylov methods)
- PPO up to 10x less efficient than vanilla gradient-based descent [Hu et al. 2018]

EXAMPLE - THIN SHELLS

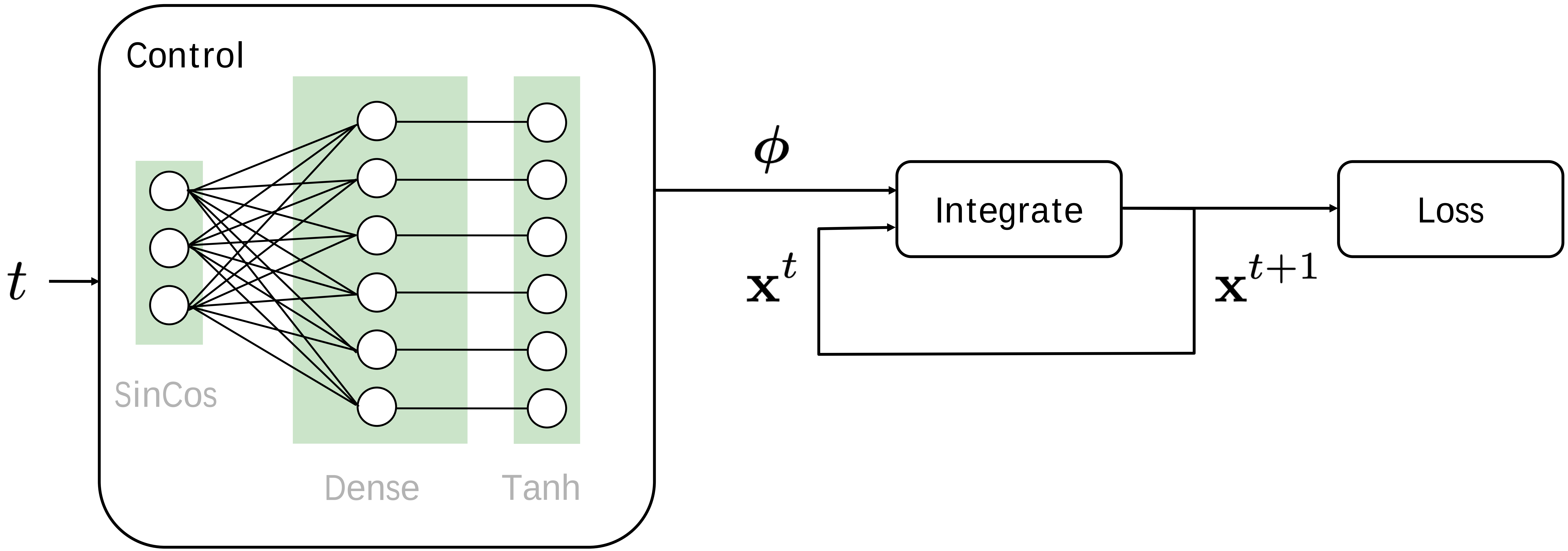
- Thin-shell FEM
- 2D NeoHookean + bending energy
- Activations into bending energy
- Lift + Drag model



[Smith et al. 2018]

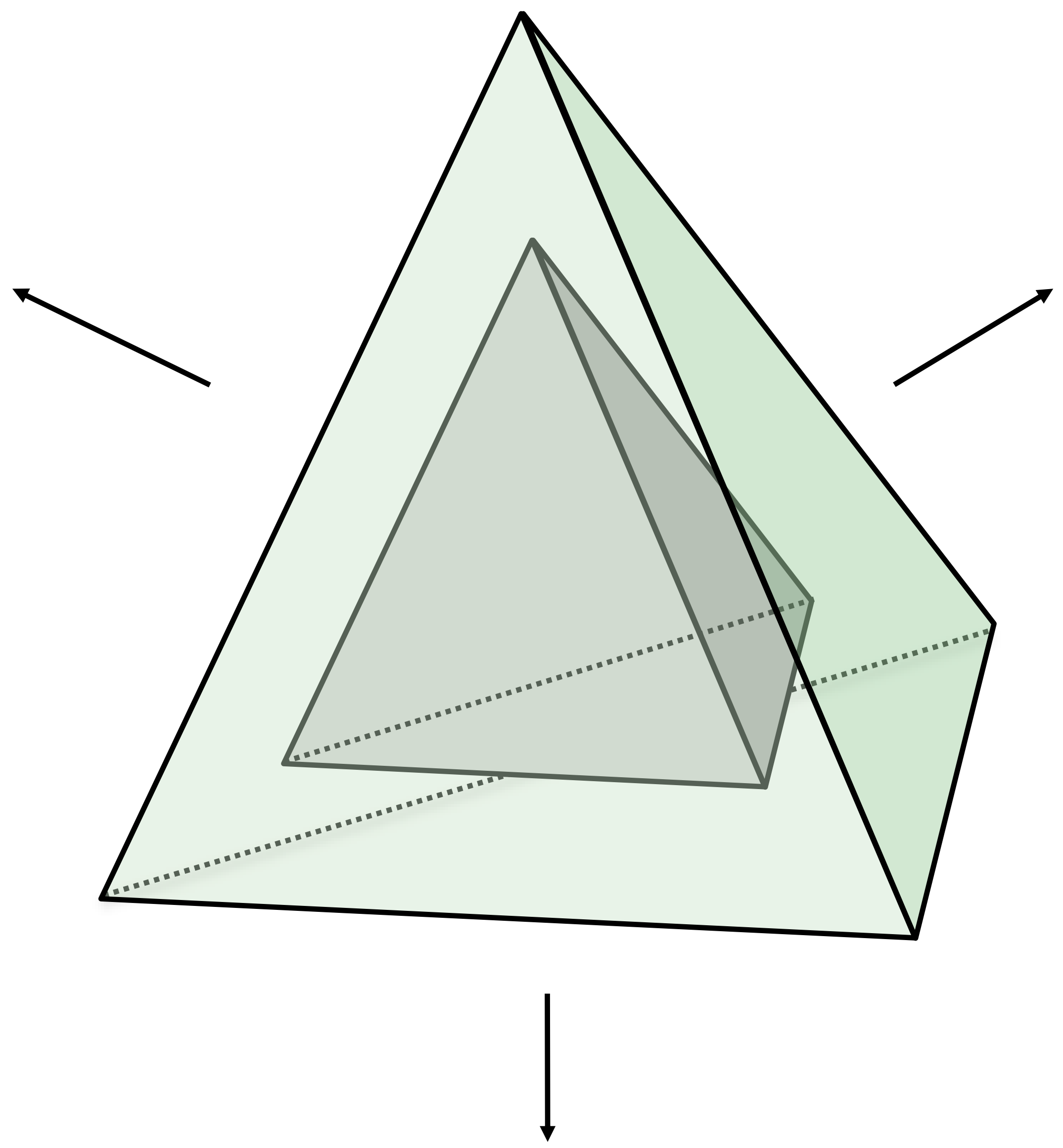
[Bridson et al. 2002]

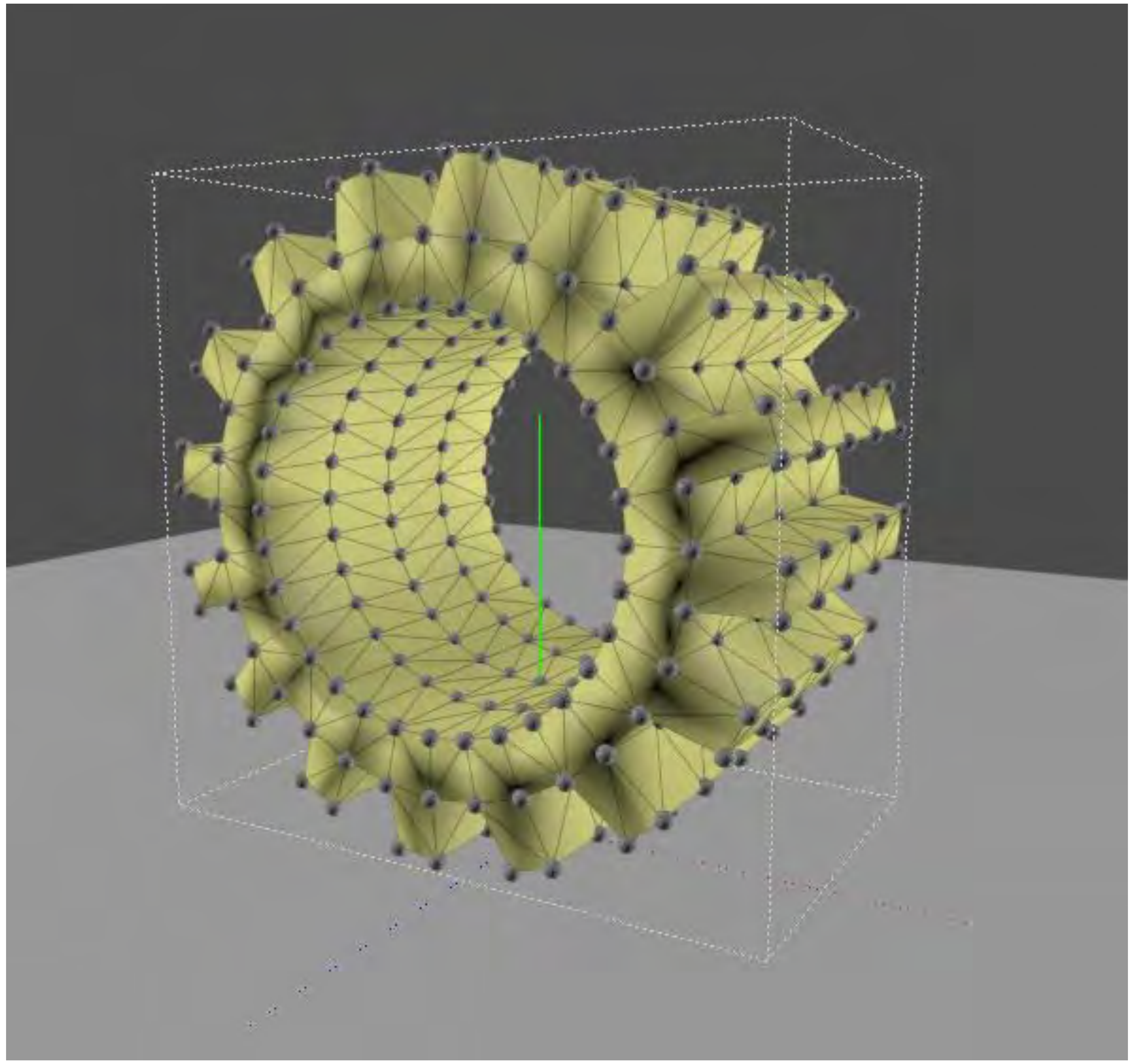
EXAMPLE - OPEN LOOP CONTROL



EXAMPLE - SOLID FEM

- Pixar’s stable NeoHookean model
- Tetrahedral elements
- Volumetric activations

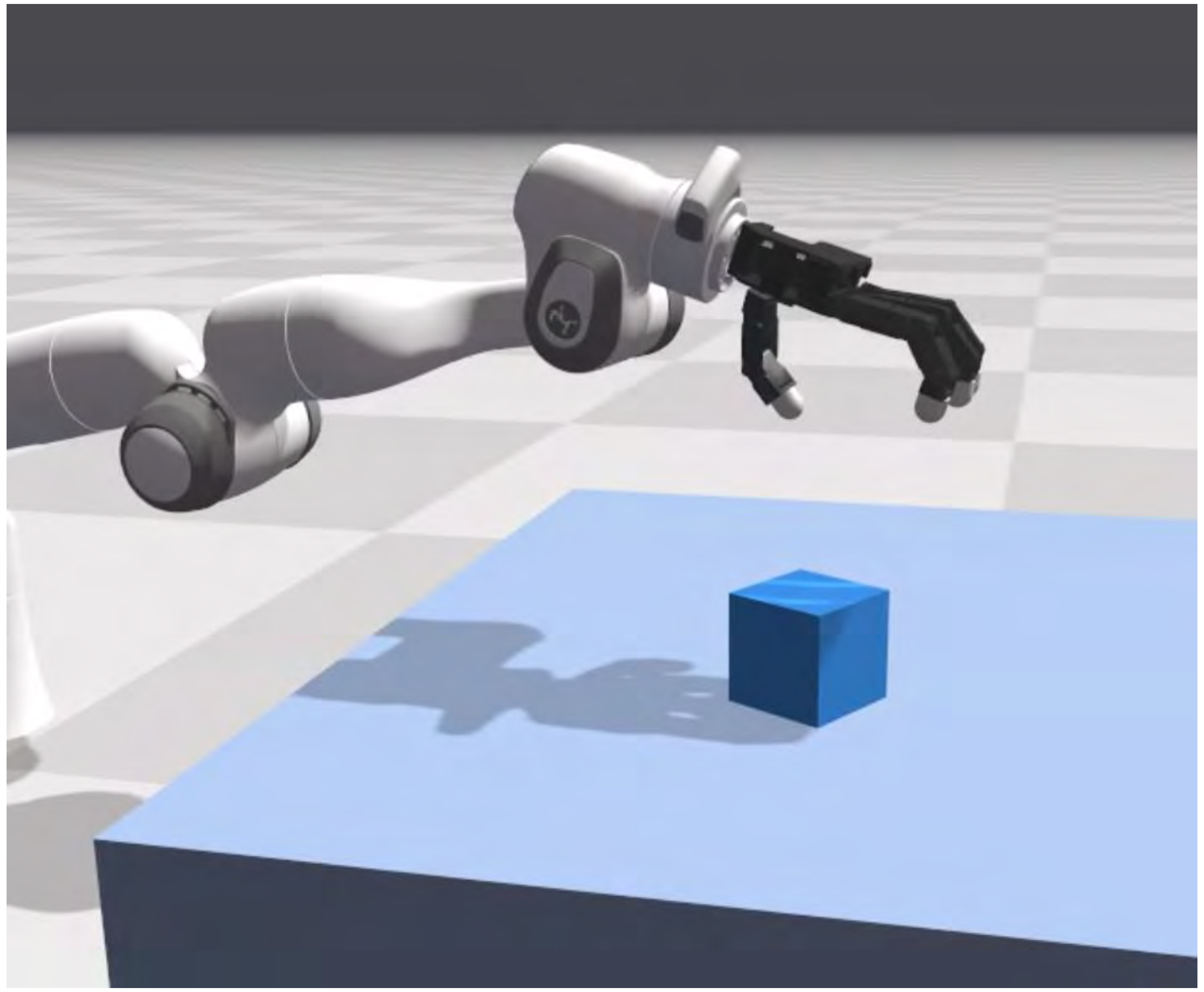




File Edit Create Window Testing Utilities Help
Viewport
Camera
RTX Real-time
Root Layer
FPS: 66.12, Frame time: 15.13 ms
926x545
Common... Post Pro... Real-Tim... Path Tra... Layers Stage
Search
Name
Type
mesh_3 Mesh
Xform Xform
scene Xform
particle_instancer PointInstancer
cloth Mesh
plane_0 Mesh
Looks Scope
OmniGlass Material
OmniPBR Material
OmniPBR_Opacity Material
OmniPBR_01 Material
RectLight RectLight
RectLight_01 RectLight
Camera Camera
Details
Root Layer (selected)
Properties
f:/gitlab/dflex/tests/outputs/top_track.usda
Content Console Test Runner Movie Capture USD Paths
Search
This PC +

RIGID ARTICULATIONS

- Reduced coordinate Featherstone (CRBA)
- Prismatic, Revolute, Spherical, Fixed, Free
- Joint / MTU-based actuation
- URDF, MJCF import



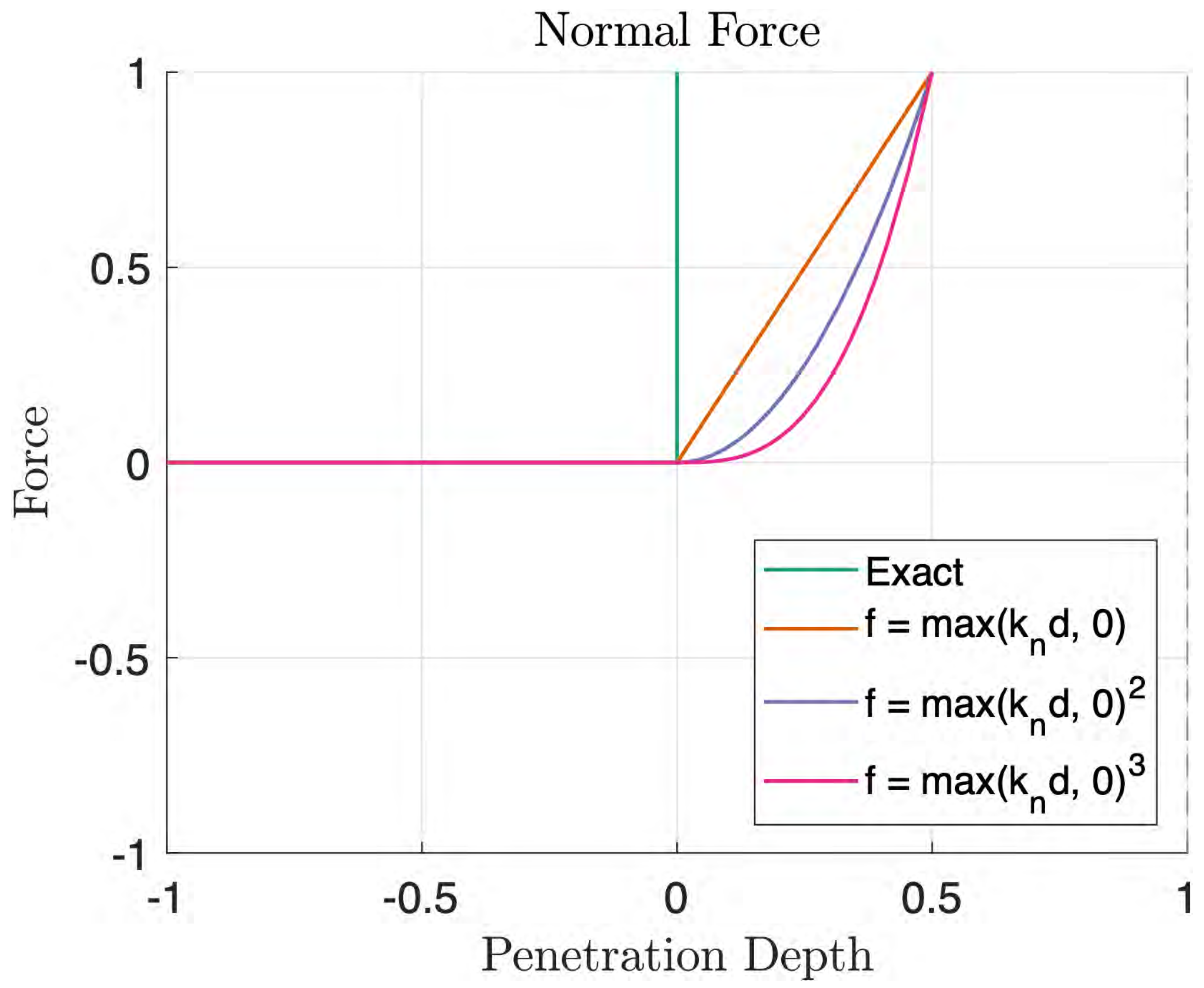

File Edit Create Window Physics Rendering OmniGraph Testing IK Utilities Help
CACHE: OFF
LIVE SYNC: OFF
Viewport
Camera
RTX Real-time
Root Layer
FPS: 78.62, Frame time: 12.72 ms
1280x720
Layer
Common Rendering...
Post Processing Se...
Path-Traced Mode...
Real-Time Mode Se...
Search
Root Layer (Authoring Layer)
root
Details
PhysX Properties
Audio Settings
Content
Console
USD Paths
Test Runner
Movie Capture
Search
Omniverse
localhost (signed out)
Add New Connection
This PC
[Desktop]
[Documents]
[Downloads]
[Pictures]
C:
E:
F:
H:

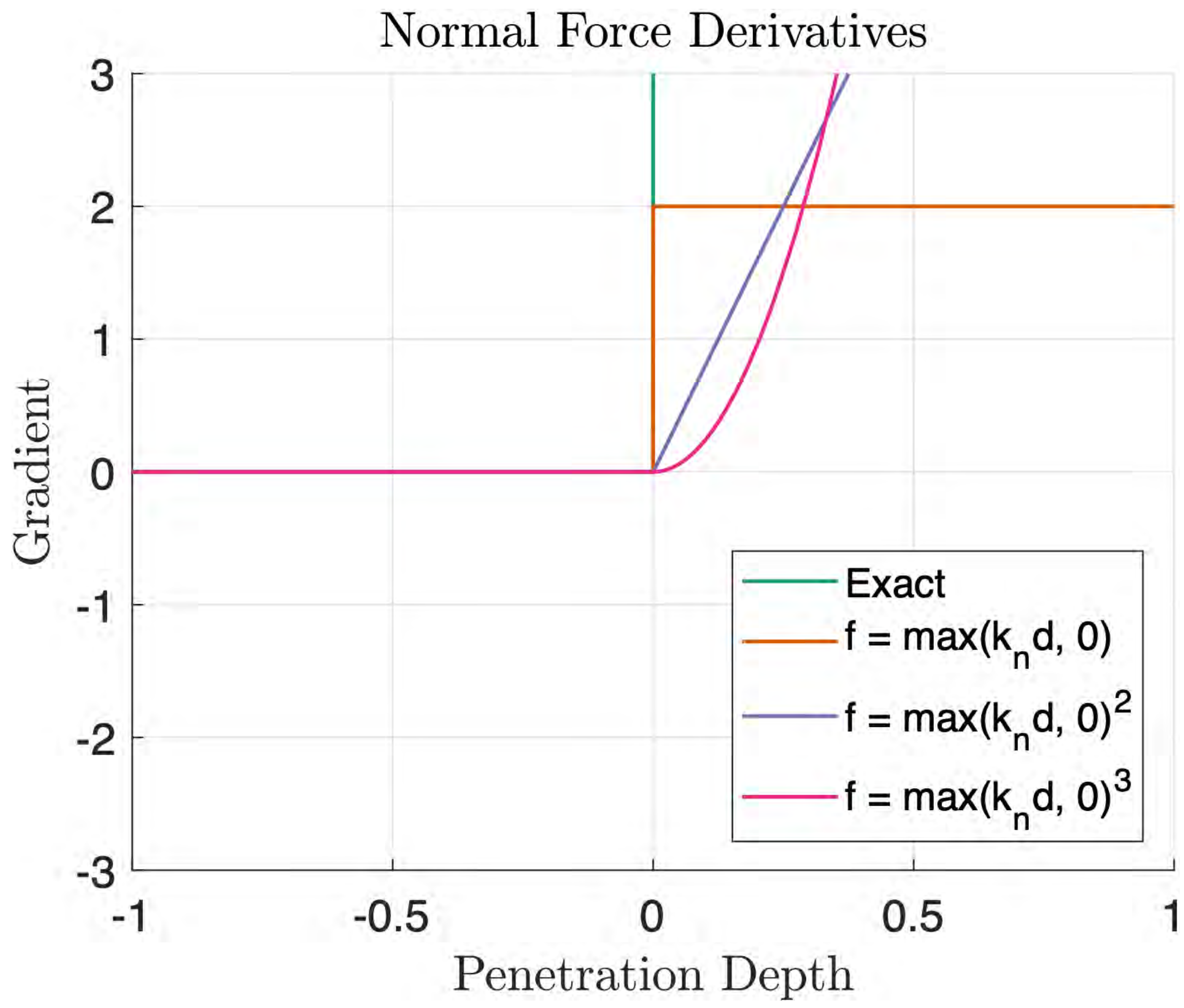

File Edit Create Window Physics Rendering OmniGraph Testing Animation IK Utilities Help
CACHE: OFF
LIVE SYNC: OFF
Viewport
Perspective
RTX Real-time
Root Layer
FPS: 66.18, Frame time: 15.11 ms
1280x720
Layer
Stage
Search
Name
Type
root
Xform
World (defaultPrim)
Xform
defaultLight
DistantLight
Looks
Scope
PreviewSurface
Material
PreviewSurface_01
Material
SphereLight
SphereLight
Details
PhysX Properties
Audio Settings
(nothing selected)
Select USD Primitive or Layer
Content
Console
USD Paths
Test Runner
Movie Capture
Search
Omniverse
localhost (signed out)
Add New Connection
This PC
[Desktop]
[Documents]
[Downloads]
[Pictures]
C:
E:
F:
H:

File Edit Create Window Physics Rendering OmniGraph Testing IK Animation Utilities Help
CACHE: OFF
LIVE SYNC: OFF
Viewport
Camera
RTX Real-time
Root Layer
FPS: 49.73, Frame time: 20.11 ms
1280x720
Layer
Stage
Real-Time Mode Settin...
Path-Traced Mode Sett...
Common Rendering Set...
Indirect Diffuse Light
Ambient Light Color R:0.100 G:0.100 B:0.100 Reset
Ambient Light Intensity 0.500 Reset
Enable Ambient Occlusion (AO) Reset
AO - Square root of temporal window length 3 Reset
AO - Minimum Samples per Pixel 1 Reset
AO - Maximum Samples per Pixel 9 Reset
AO - Ray Length 250.000 Reset
Enable Indirect Diffuse GI Reset
Enable Cached GI Reset
Cached GI - Energy Preserving Reset
Cached GI - Samples Per Pixel 1 Reset
Indirect Diffuse GI - Intensity 1.000 Reset
Indirect Diffuse GI - Max Bounces 2 Reset
Indirect Diffuse GI - PSTF Cache Update Tile 10 Reset
Indirect Diffuse GI - Lower Resolution Reset
Multi-GPU
Enable Multi-GPU Reset
Tile Overlap 32 Reset
Details
PhysX Properties
Audio Settings
(nothing selected)
Select USD Primitive or Layer
Content
Console
USD Paths
Test Runner
Movie Capture
Search
Omniverse
localhost (signed out)
Add New Connection
This PC
[Desktop]
[Documents]
[Downloads]
[Pictures]
C:
E:
F:
H:

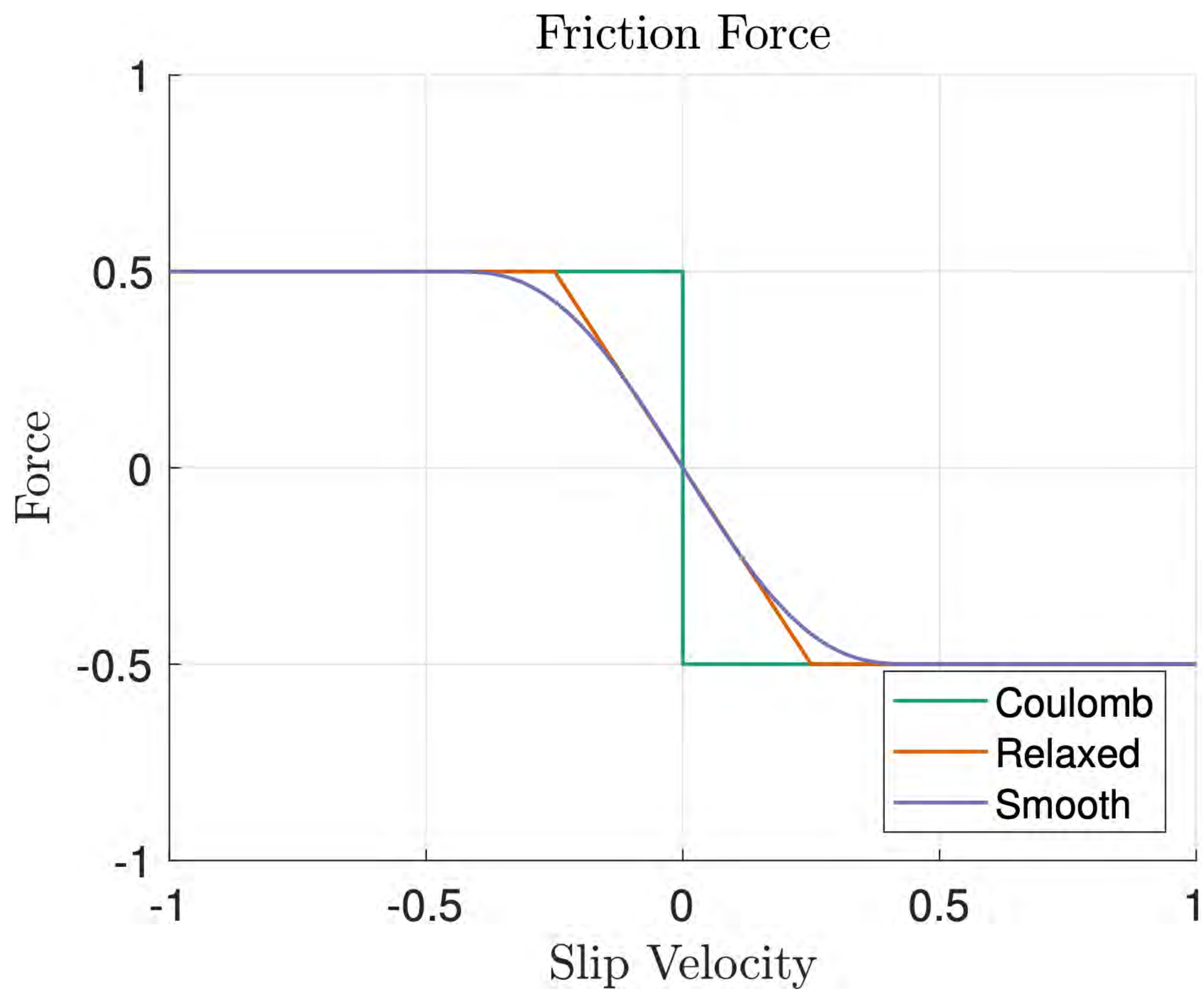
CONTACT

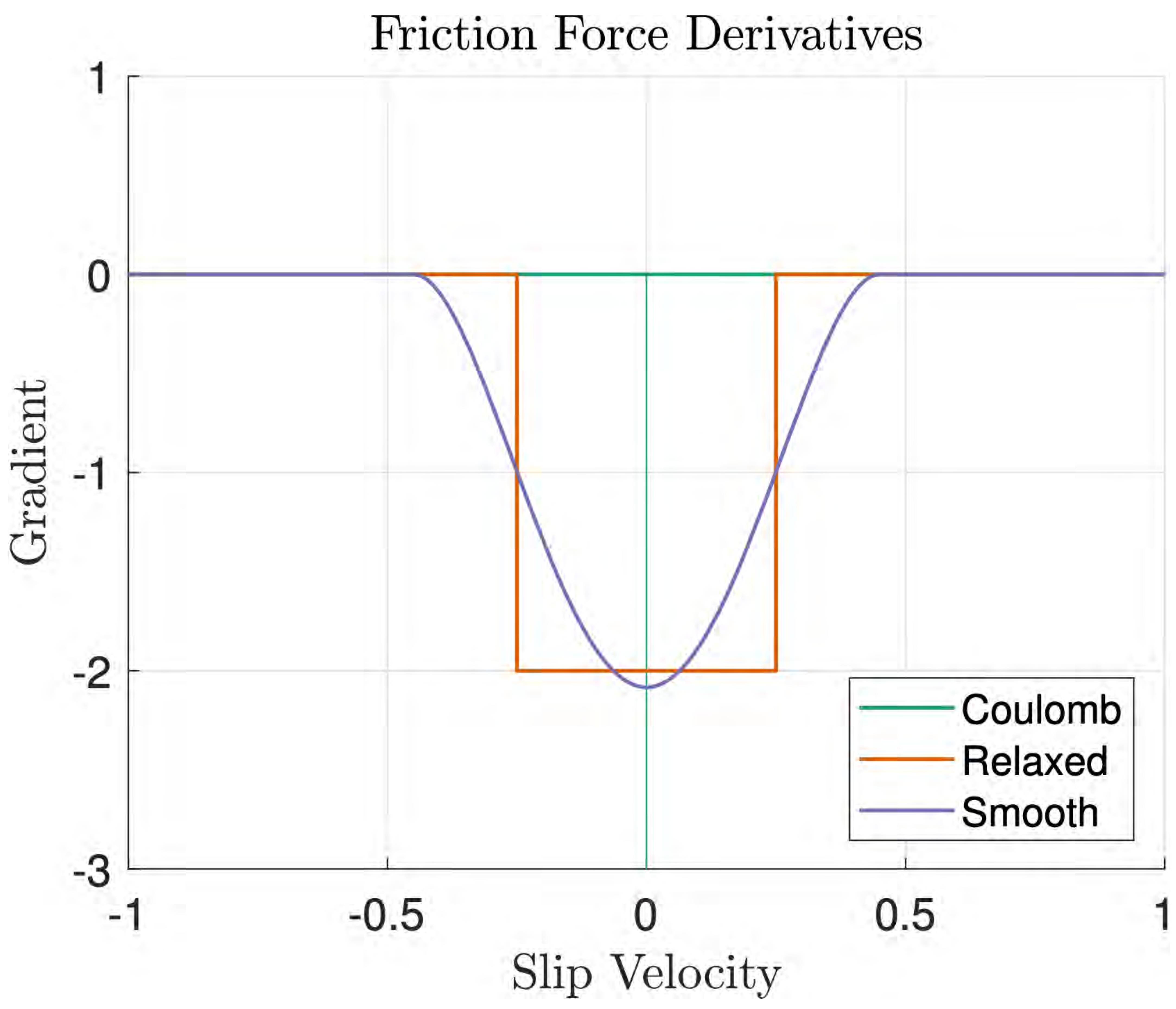
CONTACT SMOOTHNESS





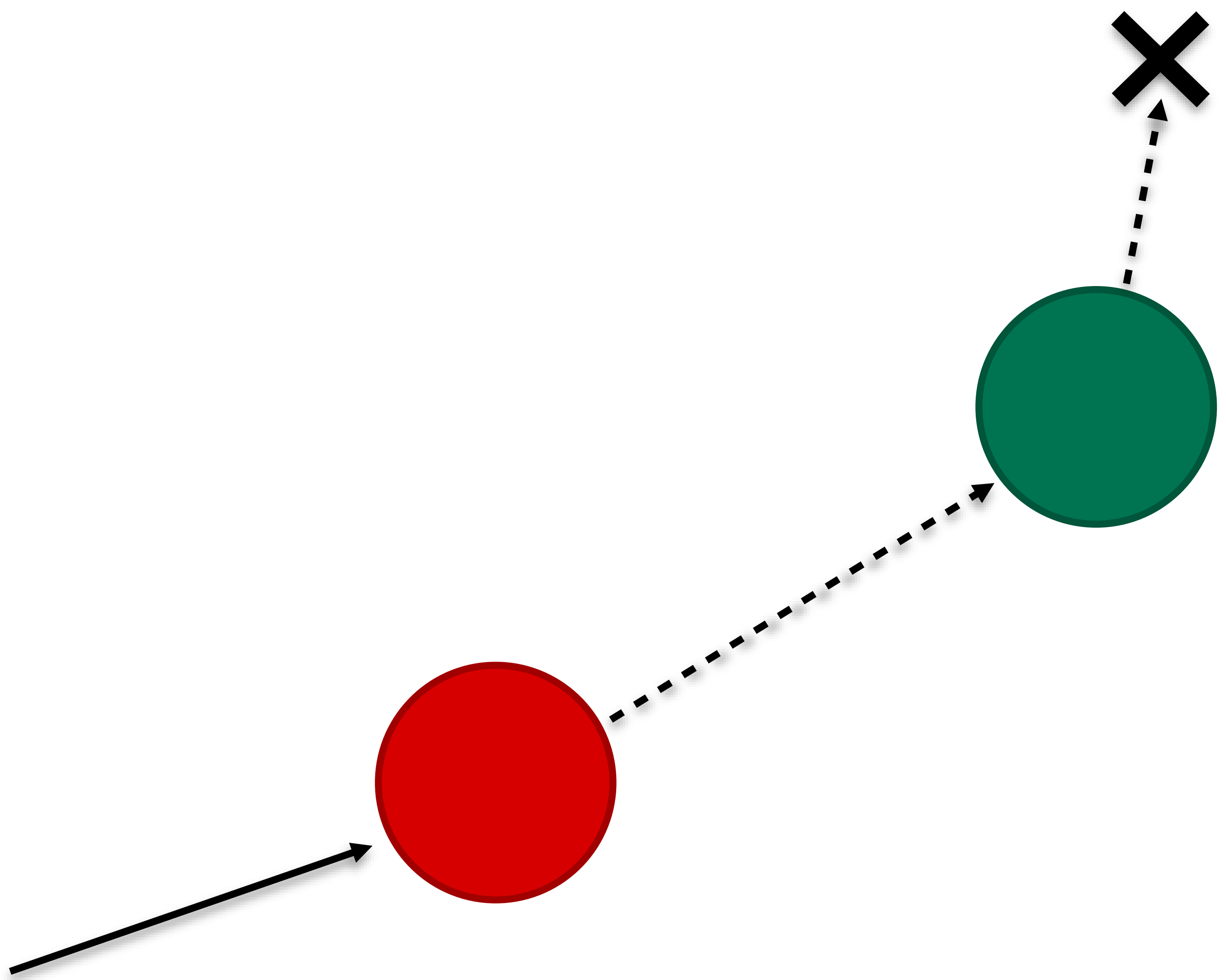
FRICTION SMOOTHNESS

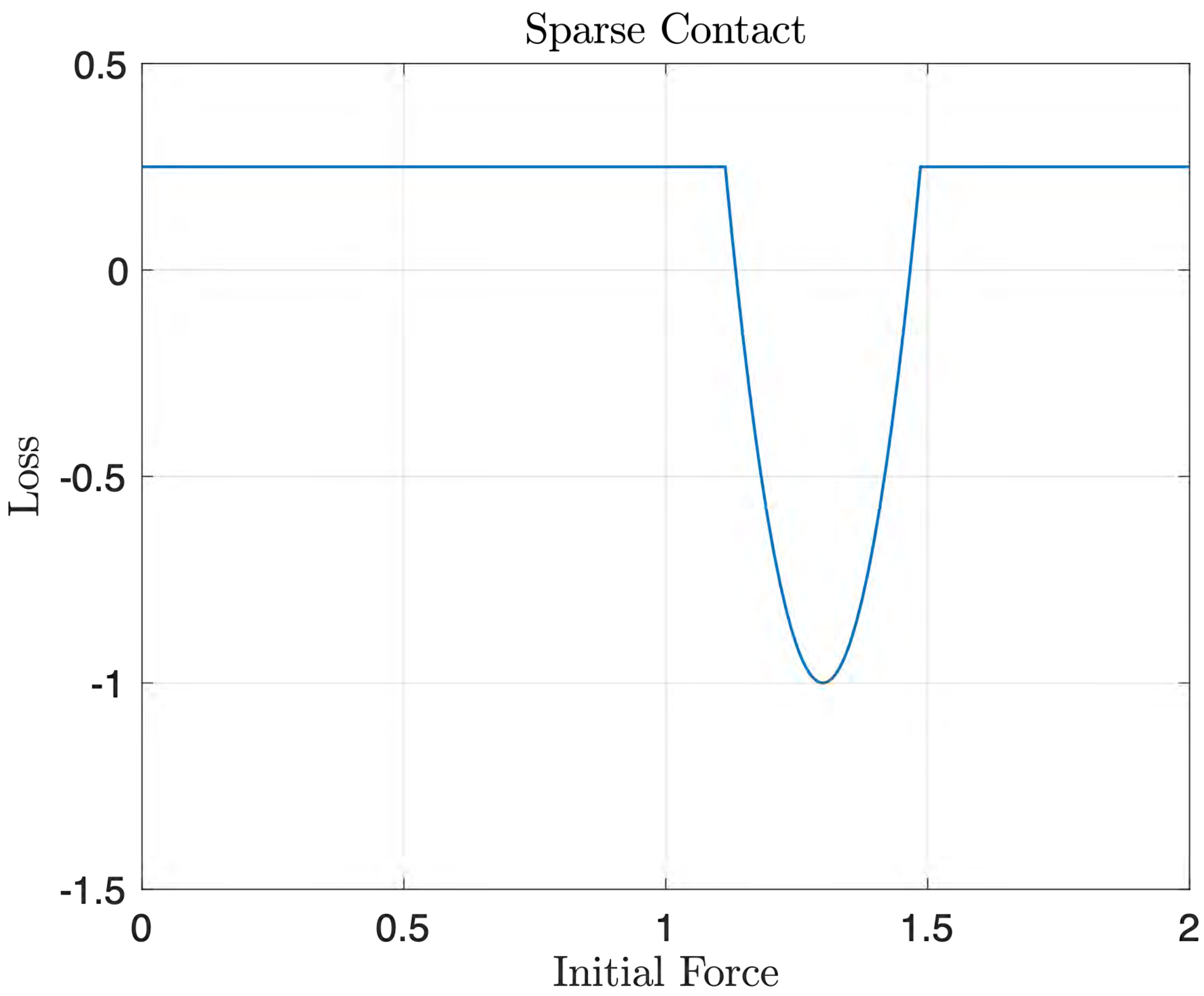




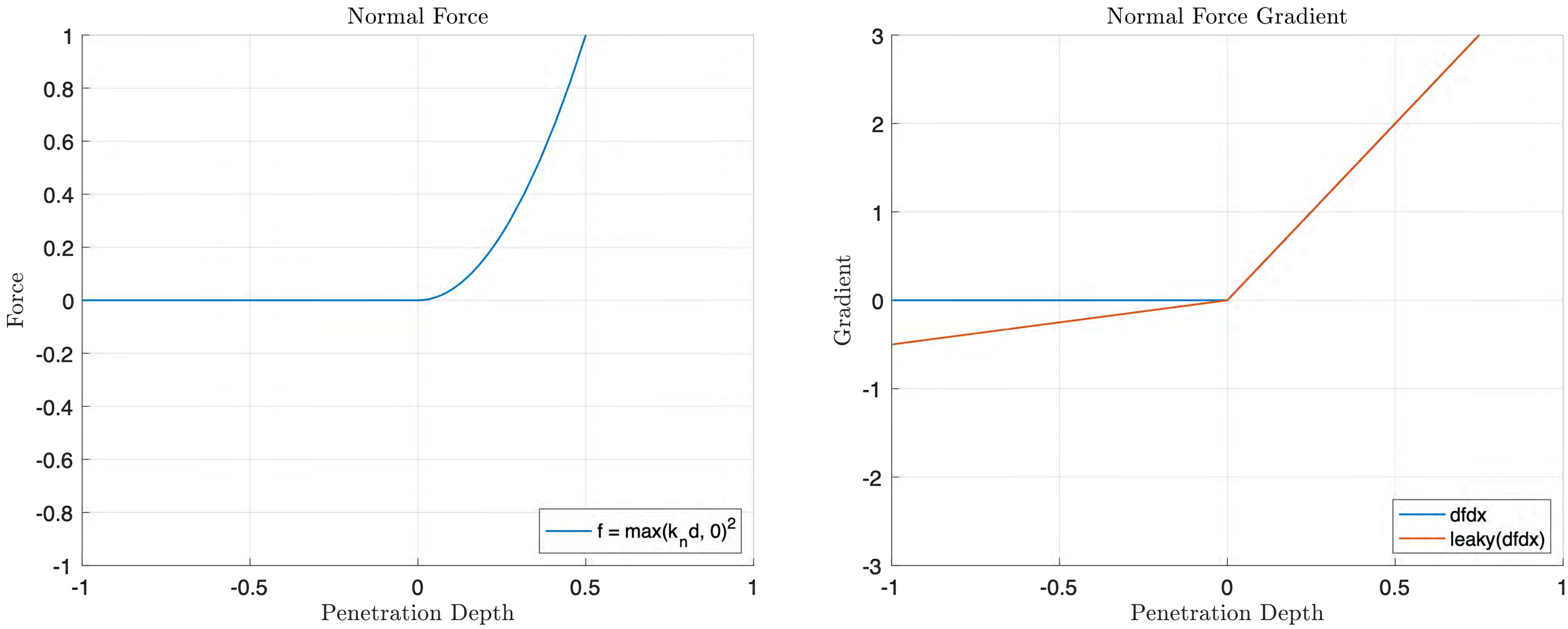
CONTACT SPARSENESS

- No gradient information until contact
- Optimization stuck at local minima





CONTACT + LEAKY GRADIENTS



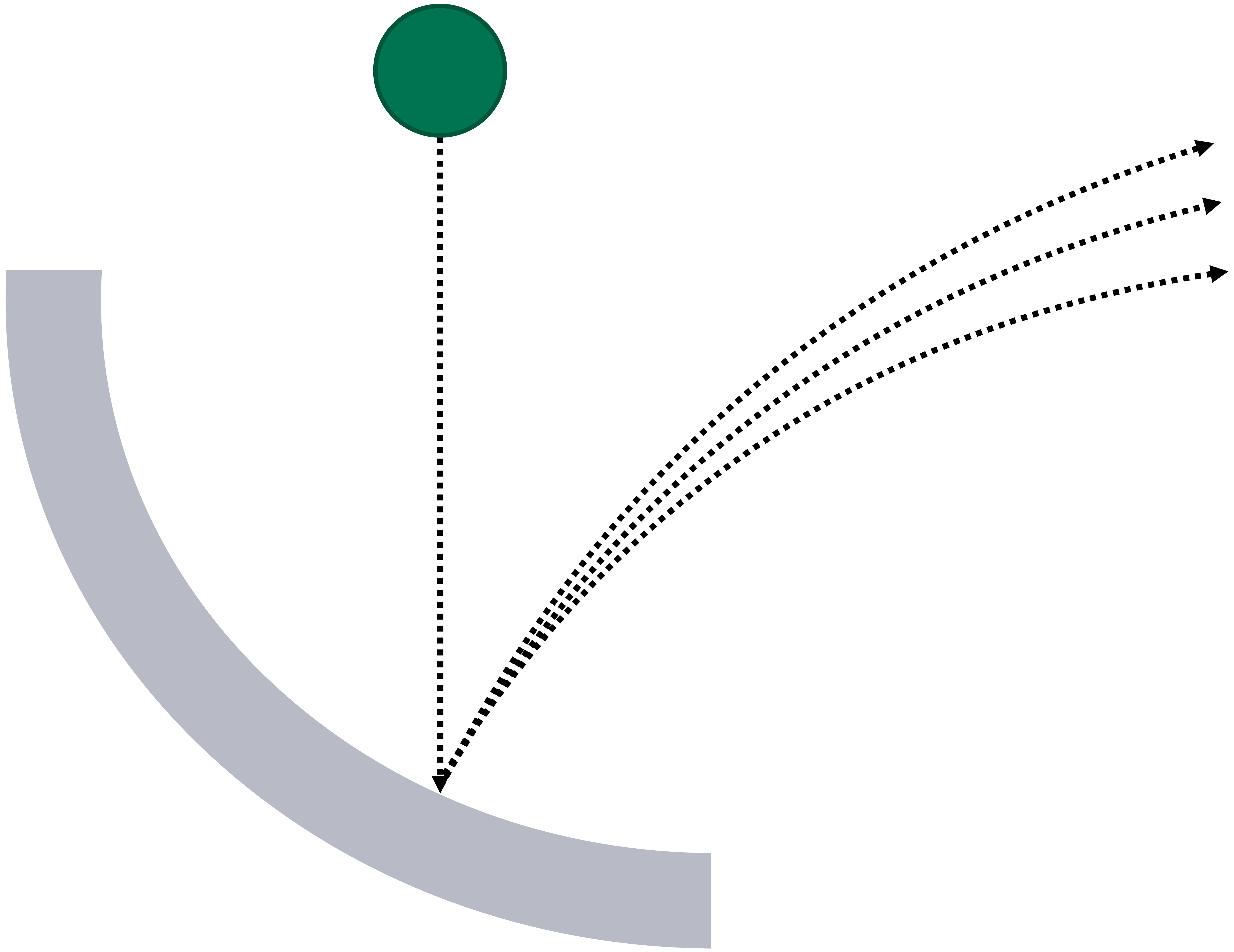
CONTACT GENERATION

- Polygons -> contact points + normals
- Computational geometry problem
- Not autodiff friendly
- SDFs are promising
- e.g.: adjoint of SDF normal:

$$n(\mathbf{x}) = \nabla\phi(\mathbf{x})$$

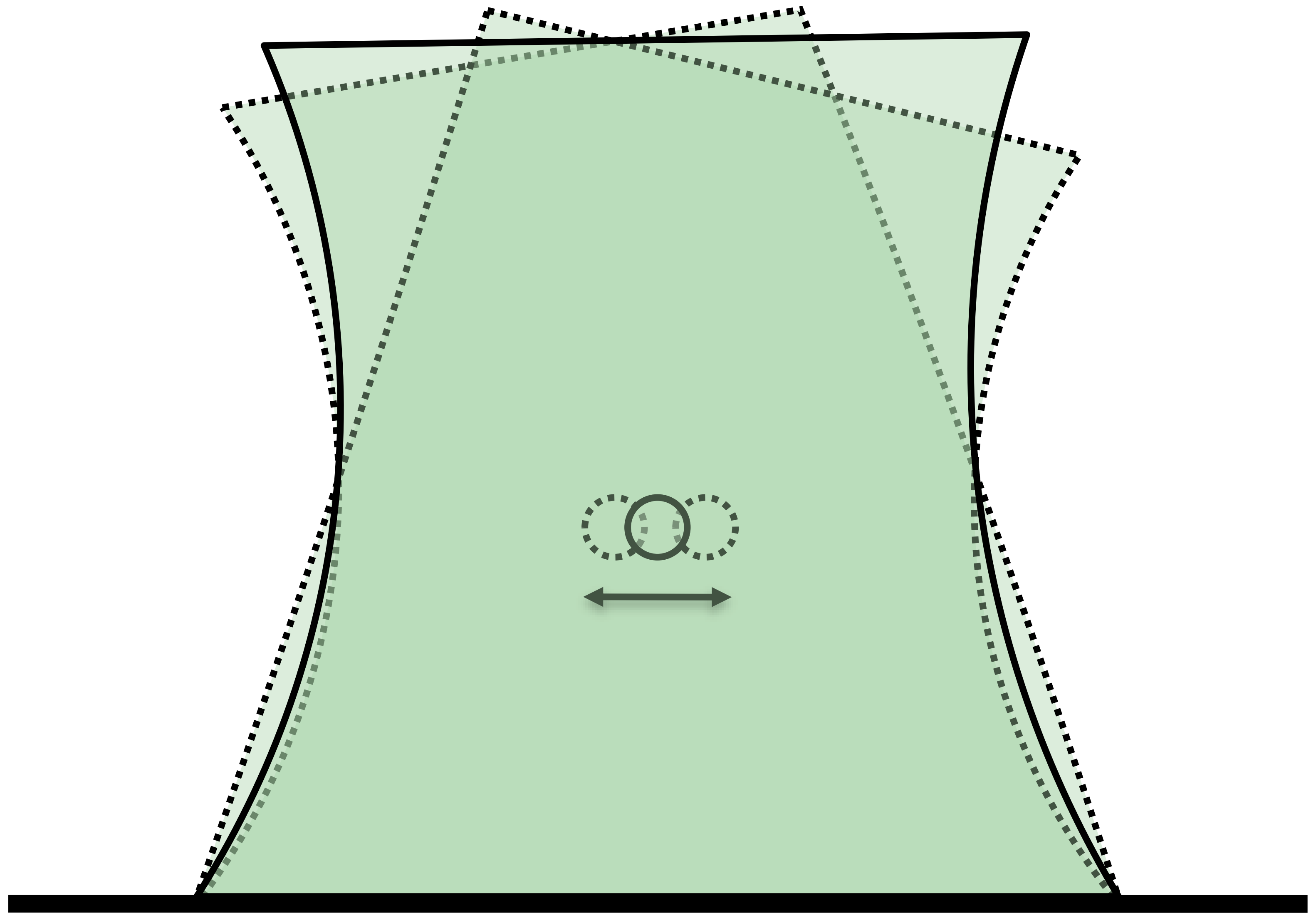
$$n^*(\mathbf{x}) = \nabla^2\phi(\mathbf{x})s^*$$

- I3D paper on SDFs accepted



LOCAL MINIMA

- Gradient methods are sensitive to local minima
- e.g.: oscillations of elastic bodies
- Momentum methods like Nesterov can ‘jump’ over these to some degree
- Combine stochastic exploration with gradient-based optimization



IMPLICIT METHODS

- What if we cannot write the solution to our time-stepping equations in a closed form? e.g.:

- Explicit Euler:

$$\begin{bmatrix}\dot{\mathbf{q}}^{t+1}\\ \mathbf{q}^{t+1}\end{bmatrix} = \begin{bmatrix}\dot{\mathbf{q}}^{t}\\ \mathbf{q}^{t}\end{bmatrix} + \Delta t\begin{bmatrix}\mathbf{M}^{-1}\nabla\Psi(\mathbf{q}^{t})\\ \dot{\mathbf{q}}^{t}\end{bmatrix} \longrightarrow \mathbf{f}(\mathbf{x}^{t}) = \mathbf{x}^{t+1}$$

- Implicit Euler:

$$\begin{bmatrix}\dot{\mathbf{q}}^{t+1}\\ \mathbf{q}^{t+1}\end{bmatrix} - \begin{bmatrix}\dot{\mathbf{q}}^{t}\\ \mathbf{q}^{t}\end{bmatrix} + \Delta t\begin{bmatrix}\mathbf{M}^{-1}\nabla\Psi(\mathbf{q}^{t+1})\\ \dot{\mathbf{q}}^{t+1}\end{bmatrix} = \mathbf{0} \longrightarrow \mathbf{f}(\mathbf{x}^{t}, \mathbf{x}^{t+1}) = \mathbf{0}$$

IMPLICIT FUNCTION THEOREM

- Implicit function theorem to the rescue! We can still compute adjoint w.r.t input:

$$\mathbf{f}(\mathbf{x}, \mathbf{y}(\mathbf{x})) = \mathbf{0}$$

- Gradient of 'output' y w.r.t. 'input' x:

$$\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \left(\frac{\partial \mathbf{f}}{\partial \mathbf{y}}\right)^{-1} \frac{\partial \mathbf{f}}{\partial \mathbf{x}}$$

- Corollary: even if f() is nonlinear f*() only requires solving a linear system!

FUTURE WORK

- Differentiable CUDA / HLSL / Slang compiler

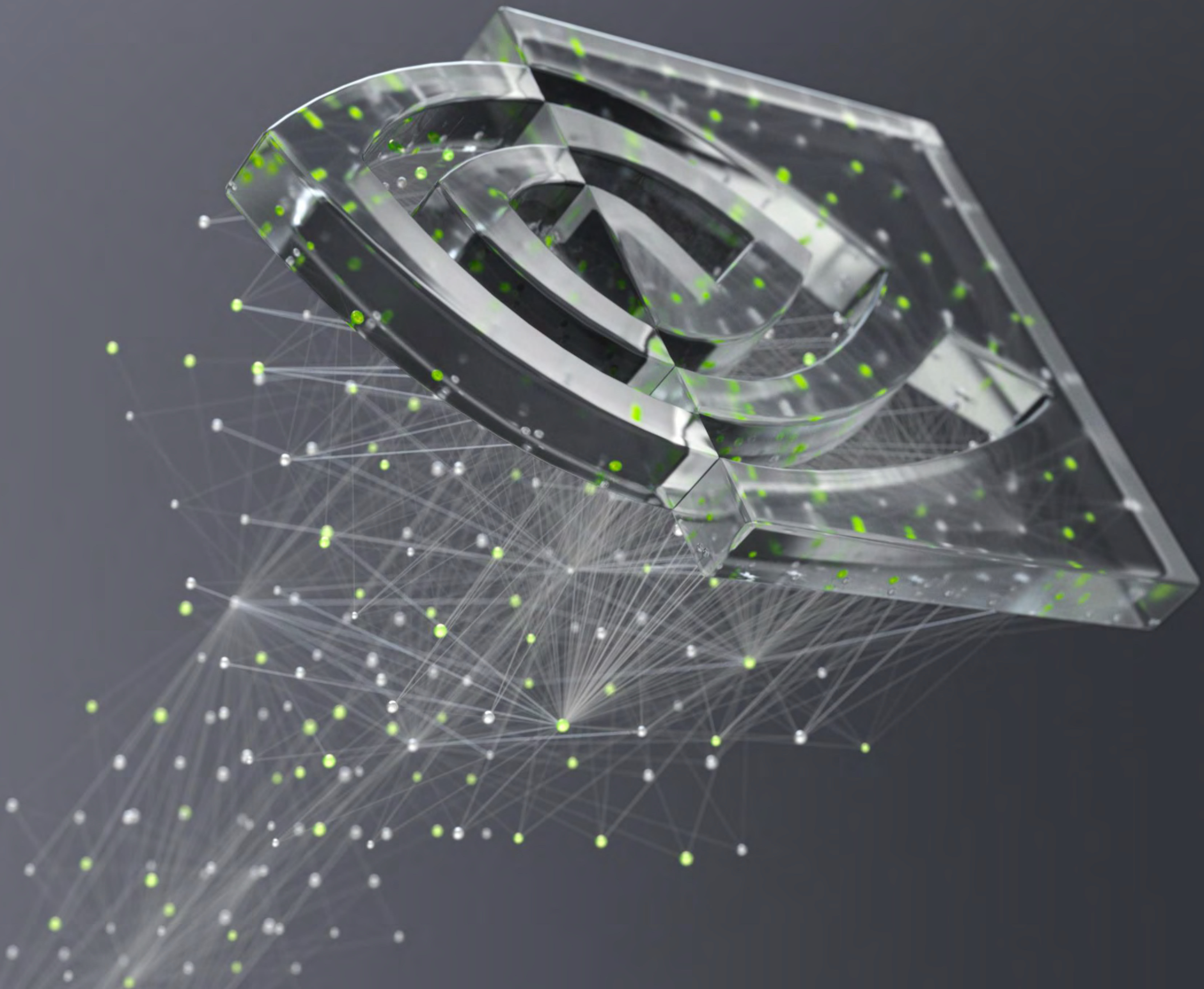
- Optimization through contact
- Application to reinforcement learning
- Application to system identification, e.g.: estimate friction/mass/restitution

- Extension to more physical models

REFERENCES

- dFlex:
  - Repo: https://gitlab-master.nvidia.com/mmacklin/dflex
  - Paper: https://drive.google.com/open?id=1JJaEXPXmv8TMrPIzx66b5JpHKAwHrha4
  - Video: https://drive.google.com/open?id=1iwp7wKunc1E0rUFmwf46RMZvNgzFSG80
- Jos's SIGGRAPH talk:
  - Video: https://developer.nvidia.com/siggraph/2019/video/sig903-vid
- Related work:

  - *Griewank, A., & Walther, A. (2008). Evaluating derivatives: Principles and techniques of algorithmic differentiation*
  - *Margossian (2019). A Review of Automatic Differentiation and its Efficient Implementation*
  - *McNamara et al. (2004). Fluid Control Using the Adjoint Method*
  - *Wojtan et al. (2006). Keyframe Control of Complex Particle Systems Using the Adjoint Method*
  - *Hu et al. (2020). DiffTaichi: Differentiable Programming for Physical Simulation*



NVIDIA

SIGGRAPH ASIA 2021 TOKYO

DIFFERENTIABLE SIMULATION FOR FRICTIONAL CONTACT

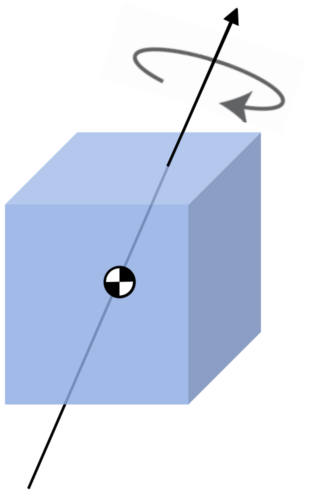
Stelian Coros, Bernhard Thomaszewski

Differentiable Simulation

Mechanical Systems
Rigid & Soft bodies

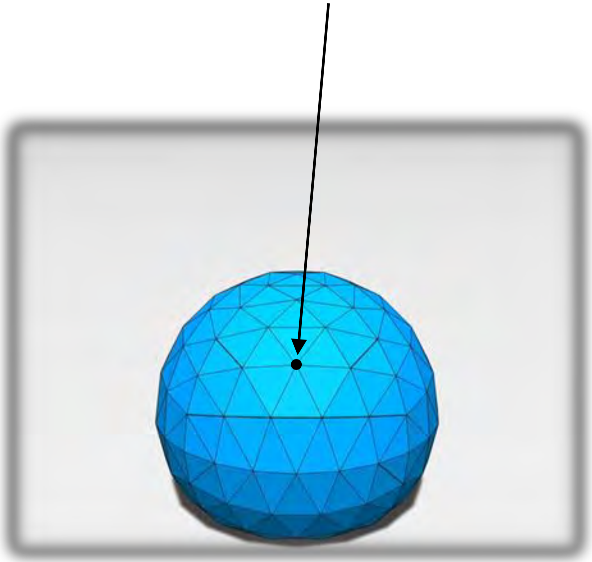
Rigid/soft multi-body systems

$\boldsymbol{x}^i := (\boldsymbol{c}_x, \boldsymbol{c}_y, \boldsymbol{c}_z, \boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\gamma})$

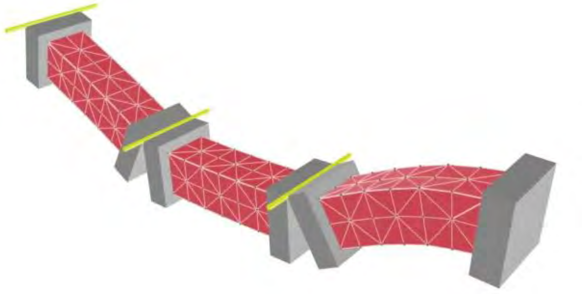


rigid bodies

$\boldsymbol{x}^i$



soft objects



coupling springs

Differentiable Simulation

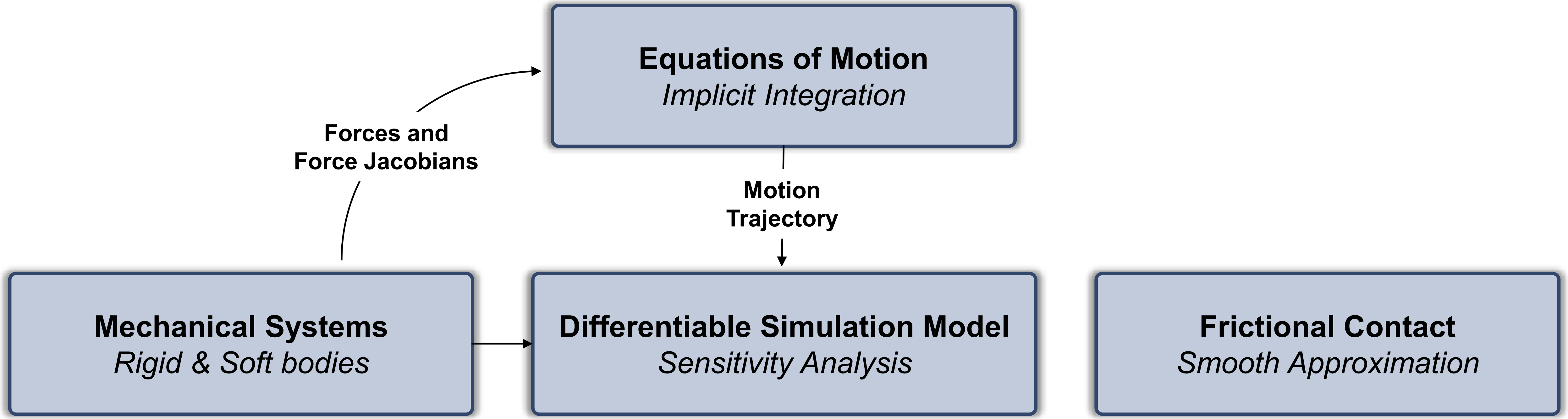
Equations of Motion
Implicit Integration

Forces and
Force Jacobians

Mechanical Systems
Rigid & Soft bodies

$$\boldsymbol{F}_{int} = -\boldsymbol{\nabla}_{\boldsymbol{x}} W(\boldsymbol{x}, C)$$

Differentiable Simulation



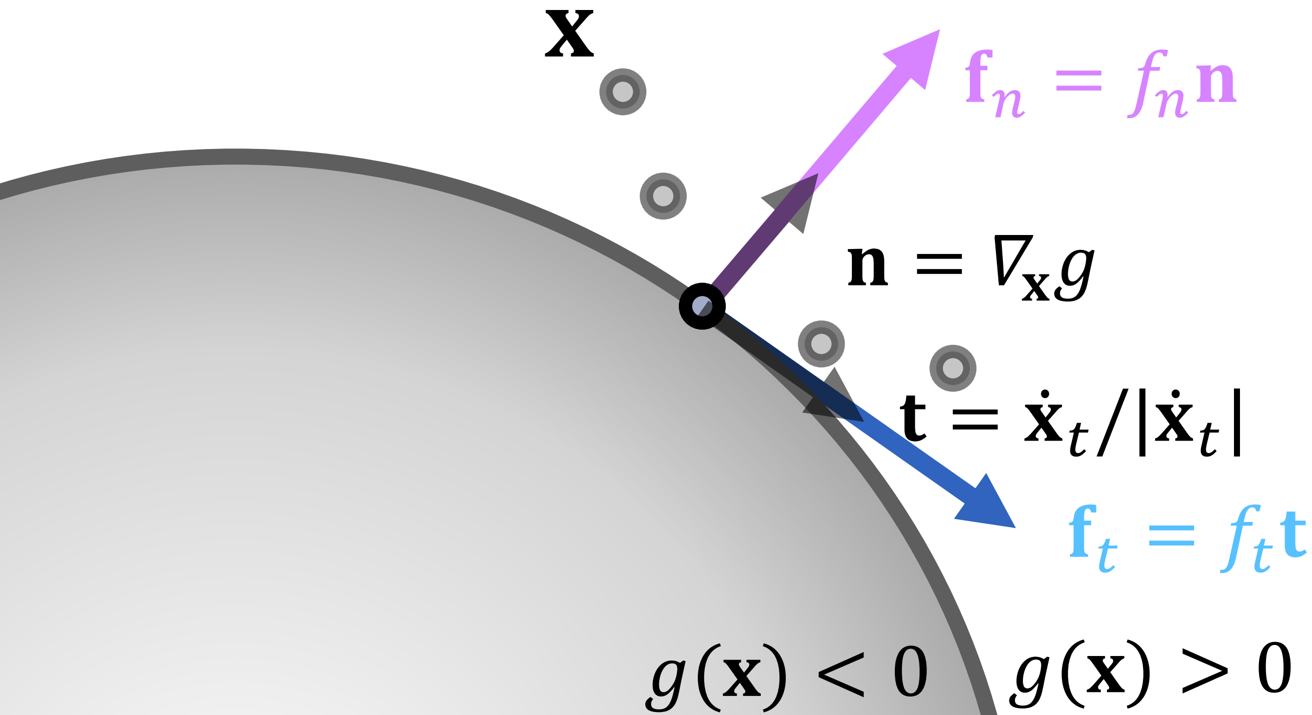
Coulomb’s law of friction: no slip occurs between two solids if $f_t \leq \mu f_n$

Differentiable approximation of frictional contact

non-penetration

constraints on normal force:

$$f_n \geq 0,\ f_n g = 0$$

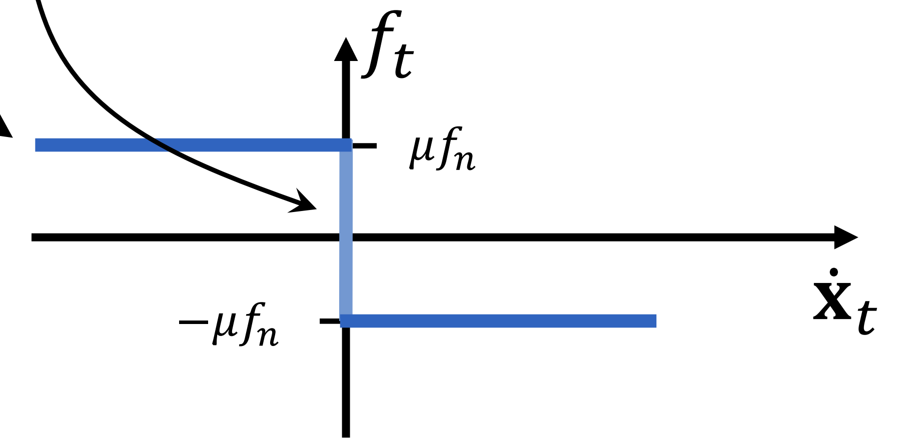


friction

constraints on tangent force:

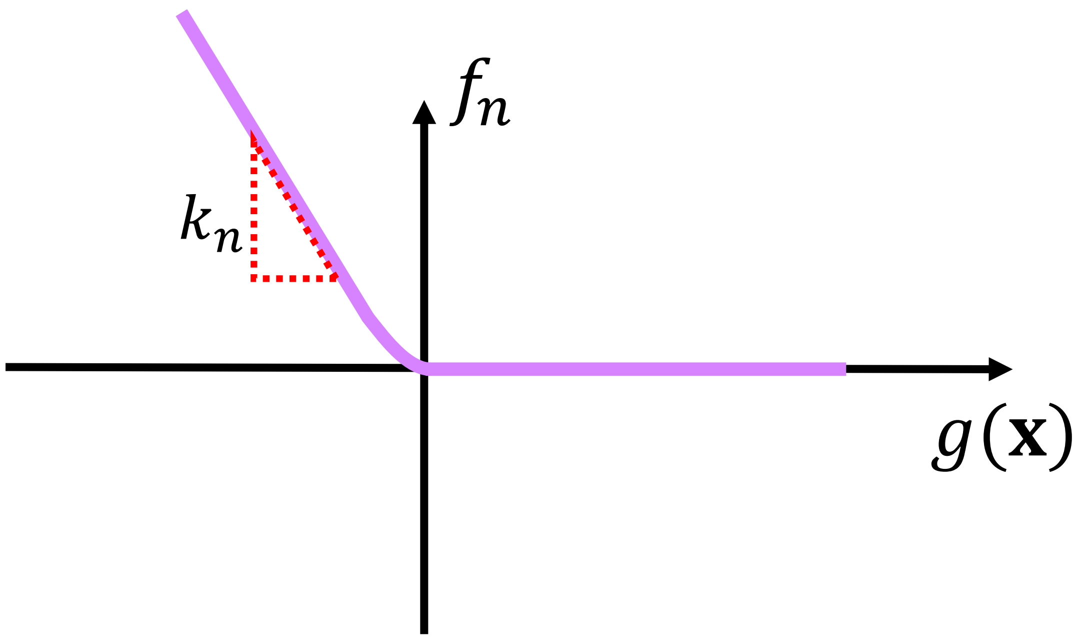
stick $\dot{\mathbf{x}}_t = 0$ iff $f_t \leq \mu f_n$

slip $\mathbf{f}_t = -\mu f_n \mathbf{t}$

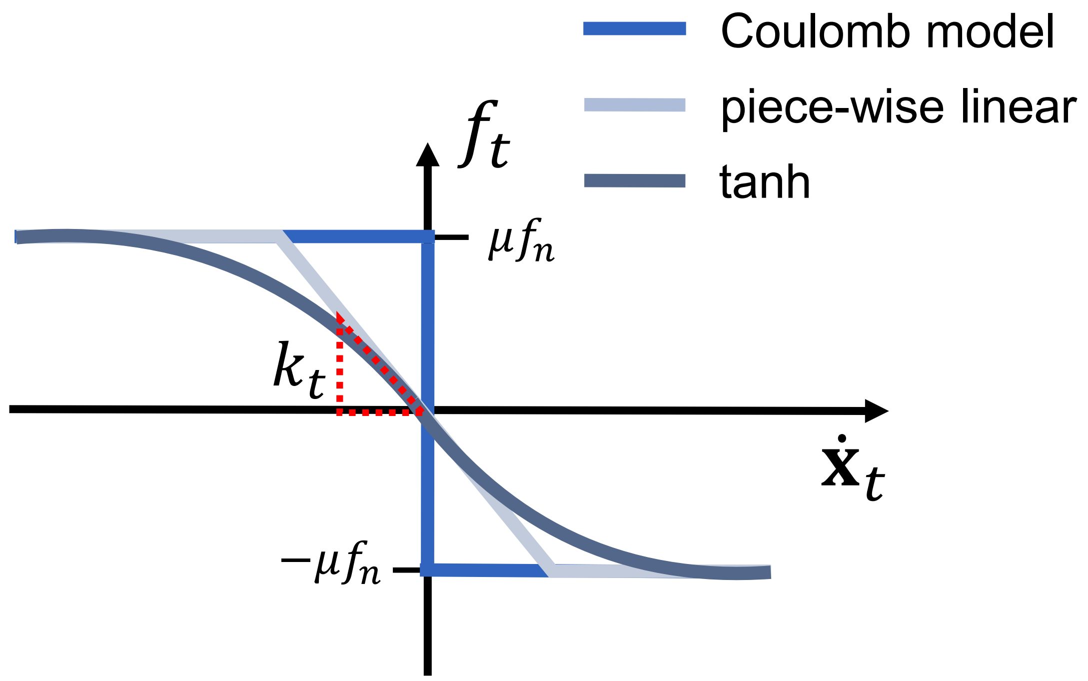


Frictional contact: smooth approximations

e.g. $\mathbf{f}_n = k_n softmax(-g, 0)\mathbf{n}$

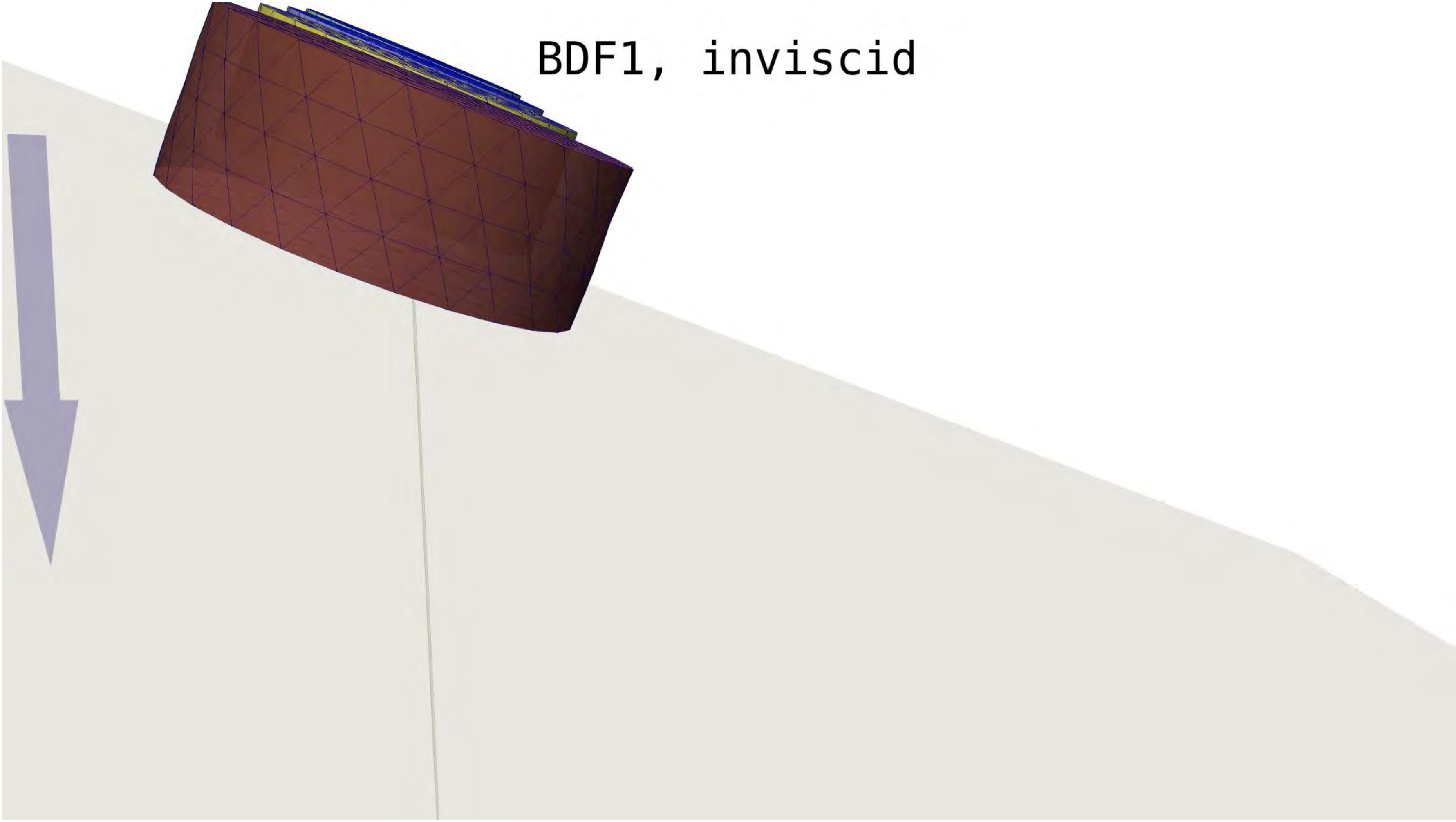


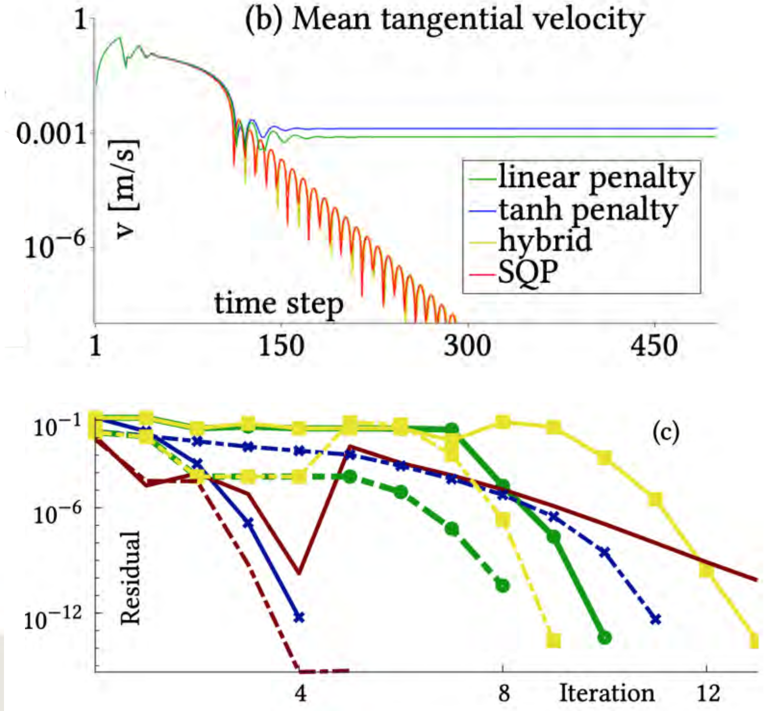
normal force



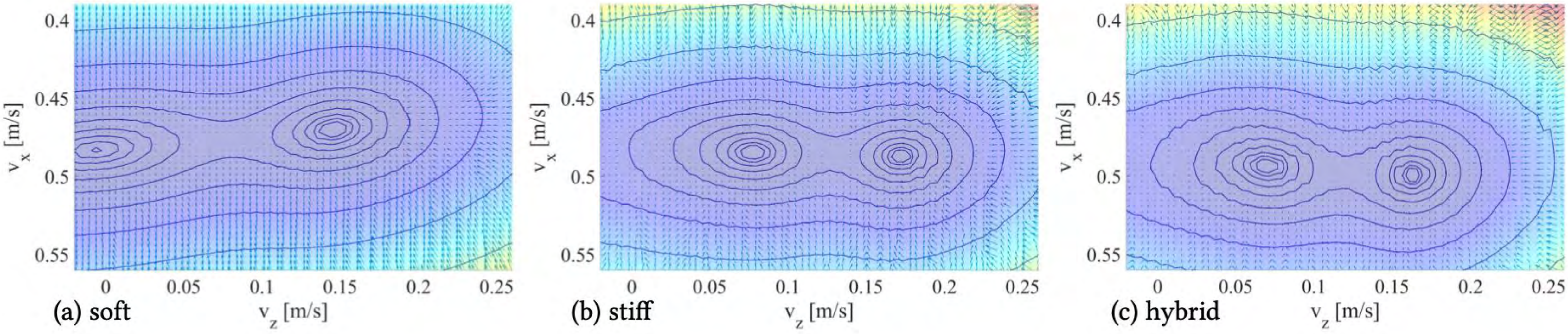
tangent force

Differentiable friction models: accuracy vs convergence speed



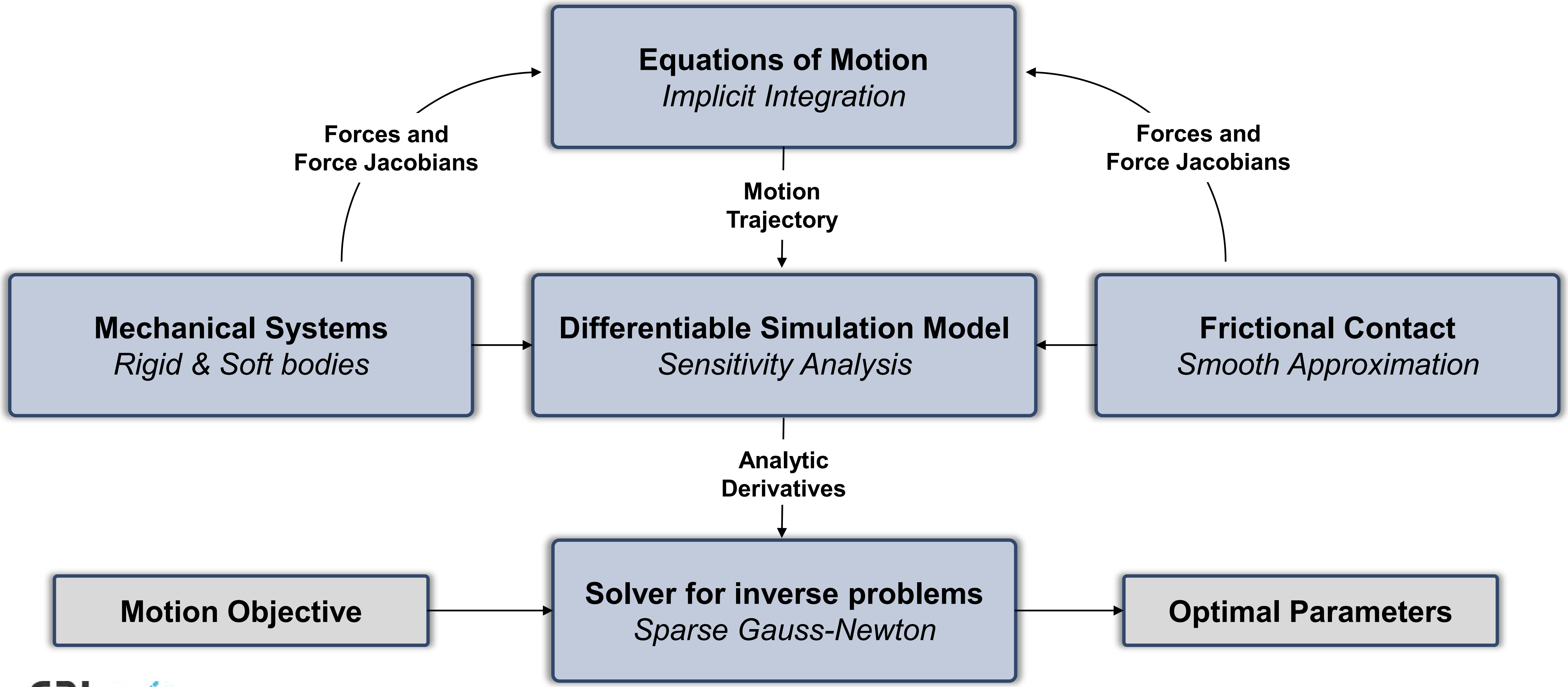


Differentiable friction models: impact on optimization landscapes



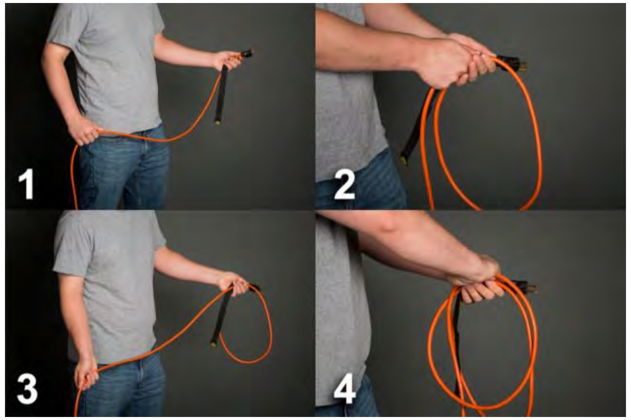
objective function (shading, isolines) and gradients (arrows)

Differentiable Simulation



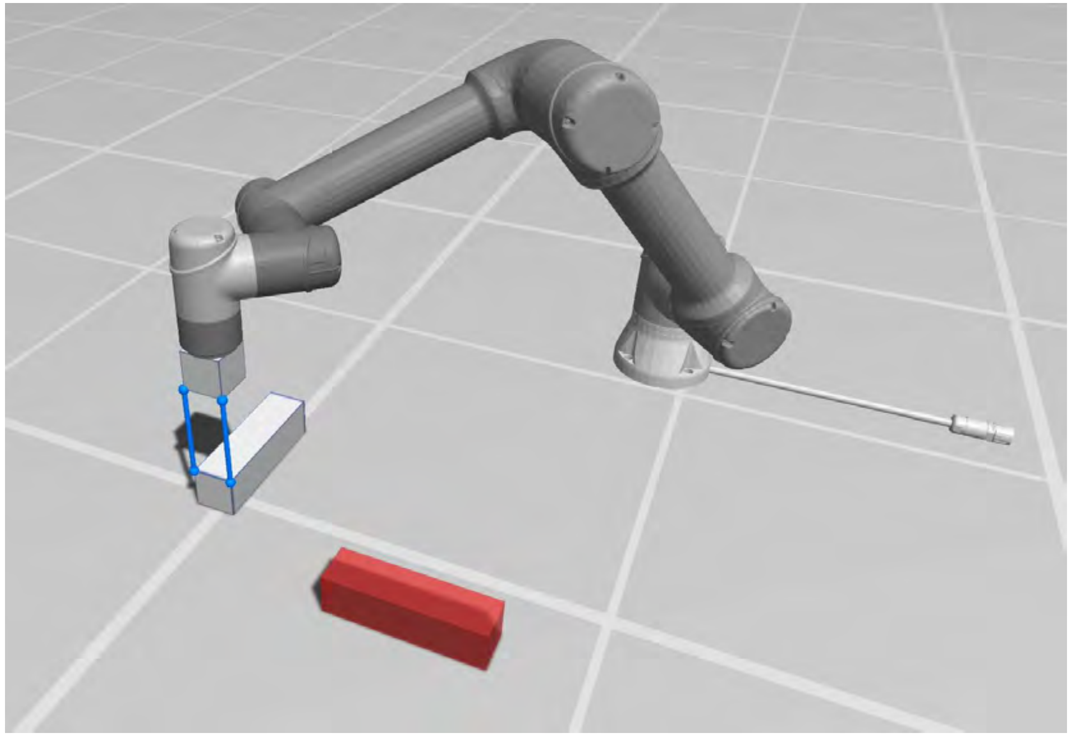
ROBOTIC MANIPULATION OF SOFT MATERIALS

Stelian Coros, Bernhard Thomaszewski

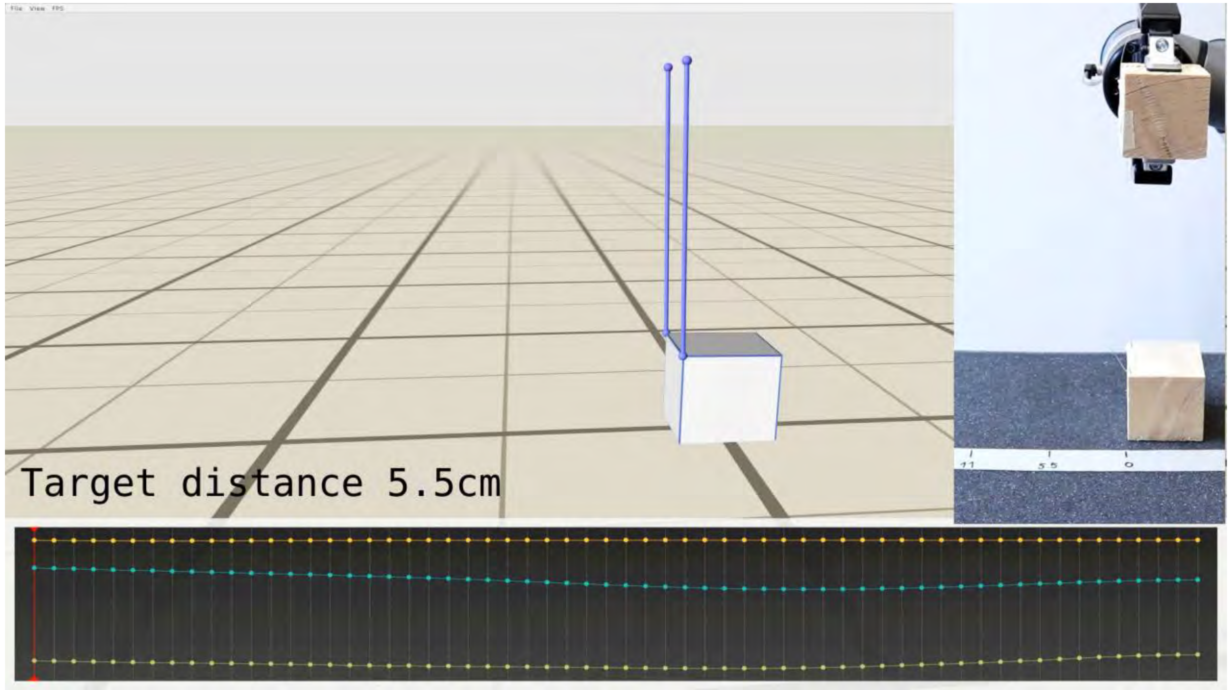

1
2
3
4

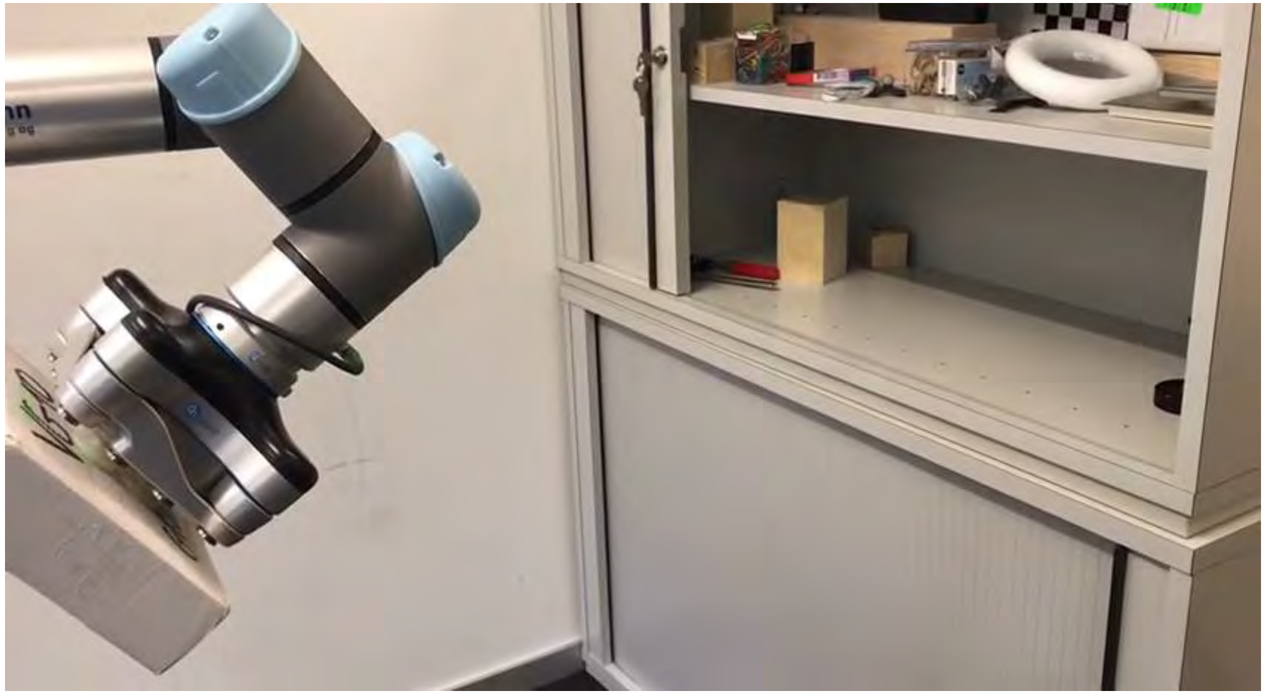
Our goal: human-level dexterity

Physics-in-the-loop motion planning

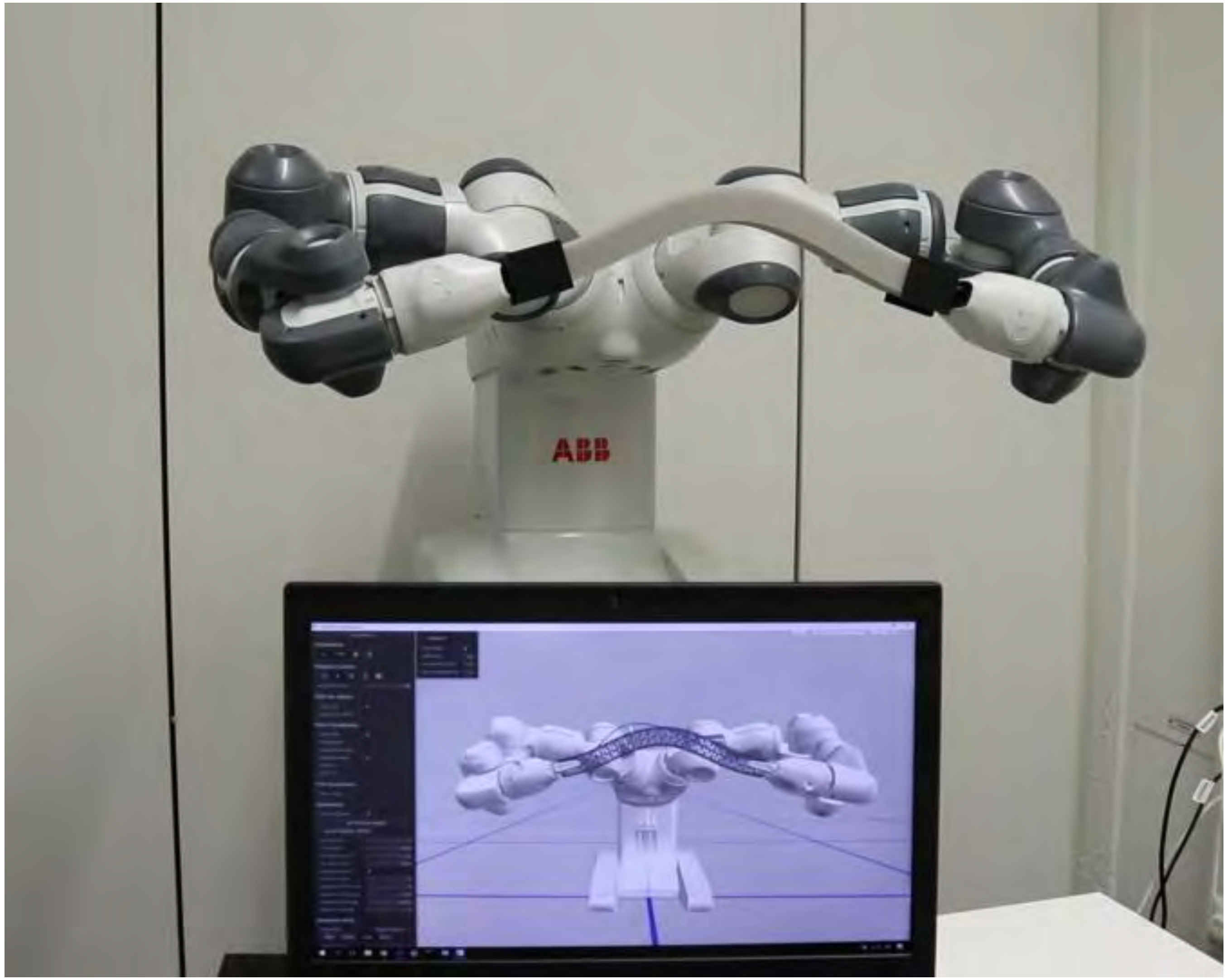


Physics-in-the-loop motion planning





Robotic Manipulation of Soft Materials

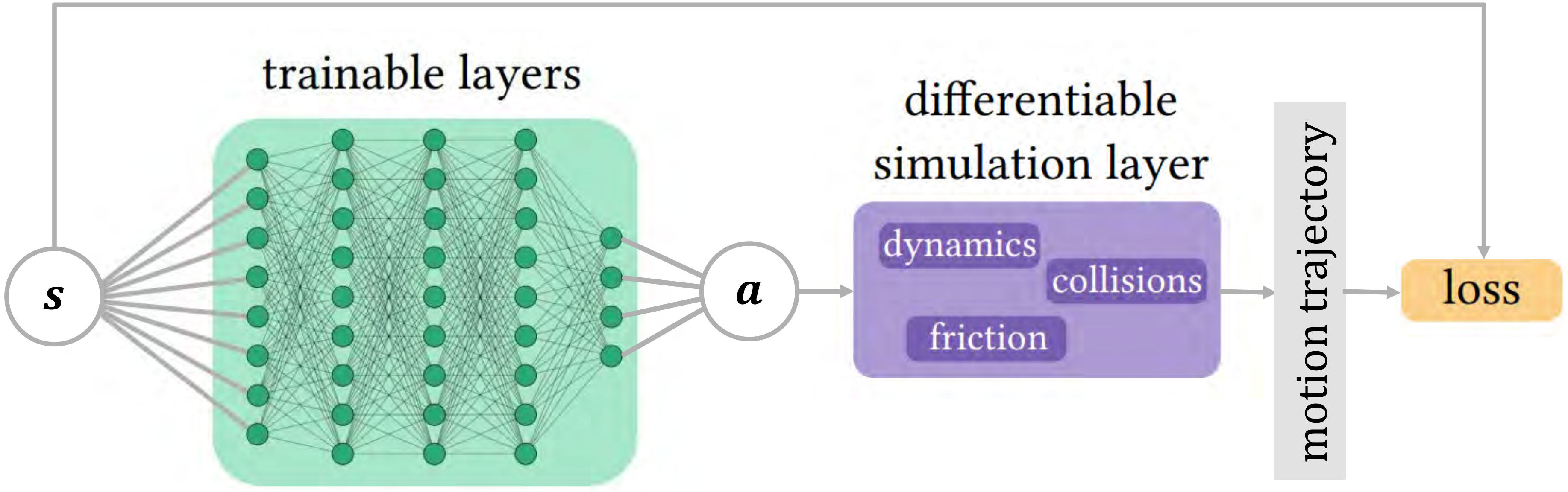


[CRL, IROS ‘18]

Differentiable Simulation: use cases

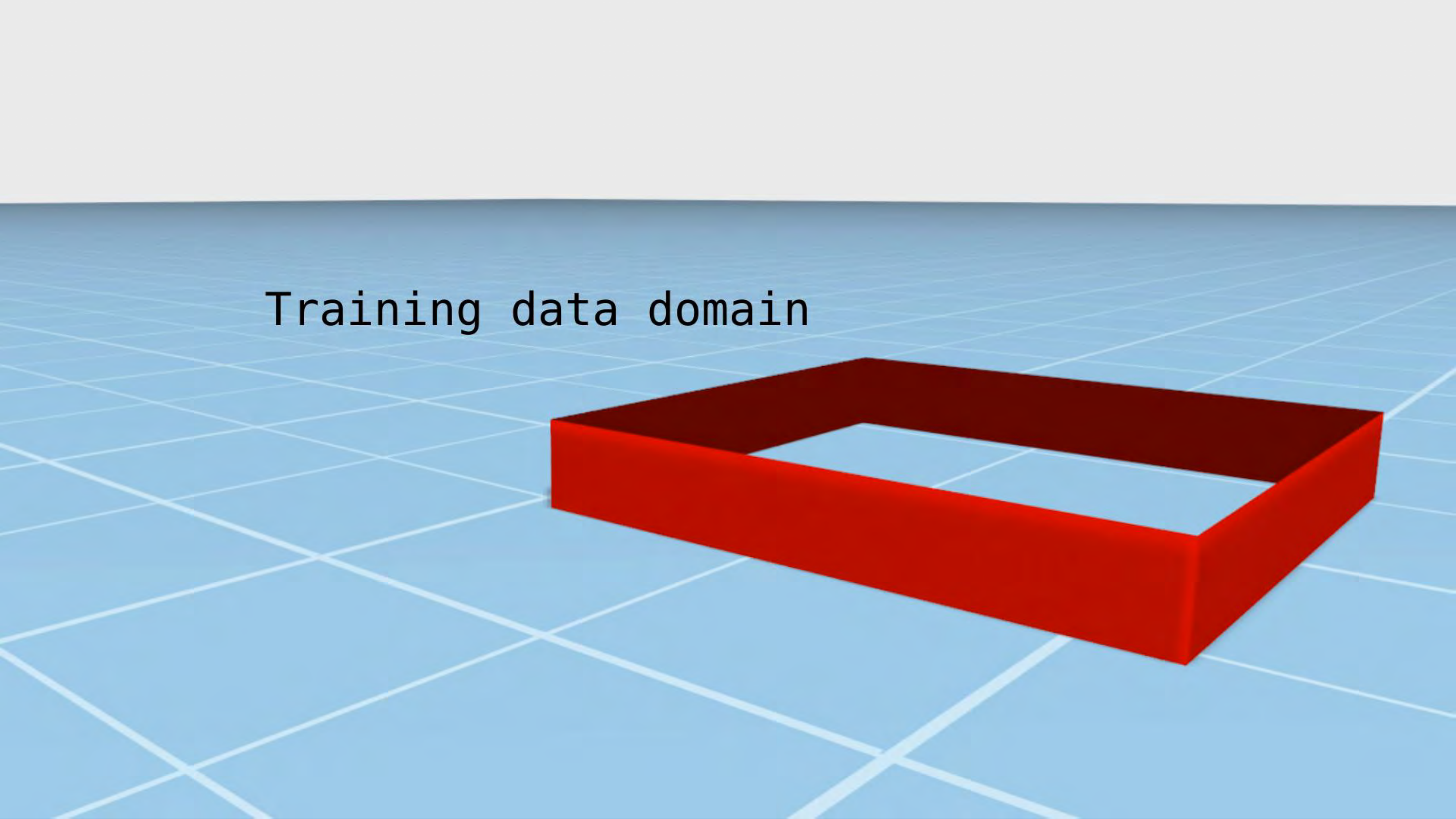
Optimization criterion	Decision Variables		
	control parameters		
Motion goals	whole body control; model predictive control; trajectory optimization		

Self-supervised learning



$$\pi_\theta : \mathbb{S} \rightarrow \mathbb{A}$$

[Geilinger et al., SIGGRAPH Asia 2020]


Training data domain

Trajectory vs policy optimization

Trajectory Optimization

$$\min_{\boldsymbol{a}} O(\boldsymbol{x}(s_0, \boldsymbol{a}), \boldsymbol{a})$$

Policy Optimization

$$\min_{\boldsymbol{\theta}} \int_s O\left(\boldsymbol{x}\big(s, \boldsymbol{\pi}_{\boldsymbol{\theta}}(s)\big), \boldsymbol{\pi}_{\boldsymbol{\theta}}(s)\right) ds$$

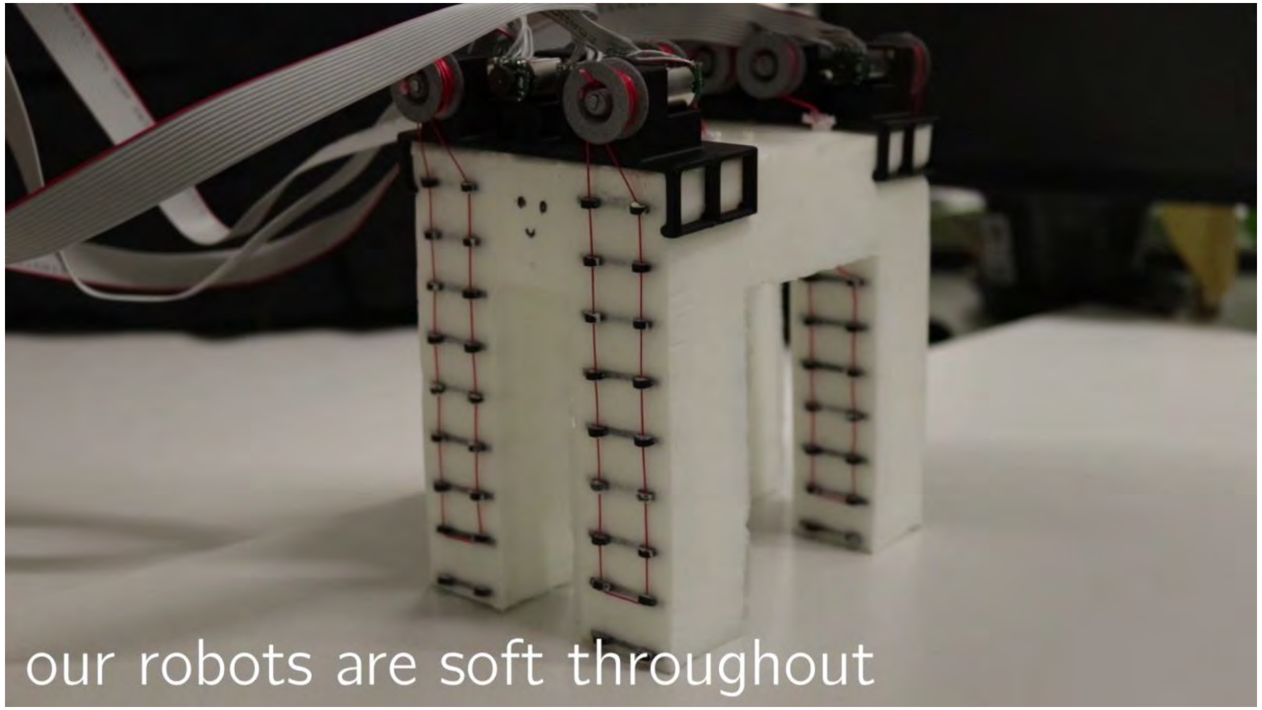
[Mora et al., ICML 2021]

Differentiable Simulation: use cases

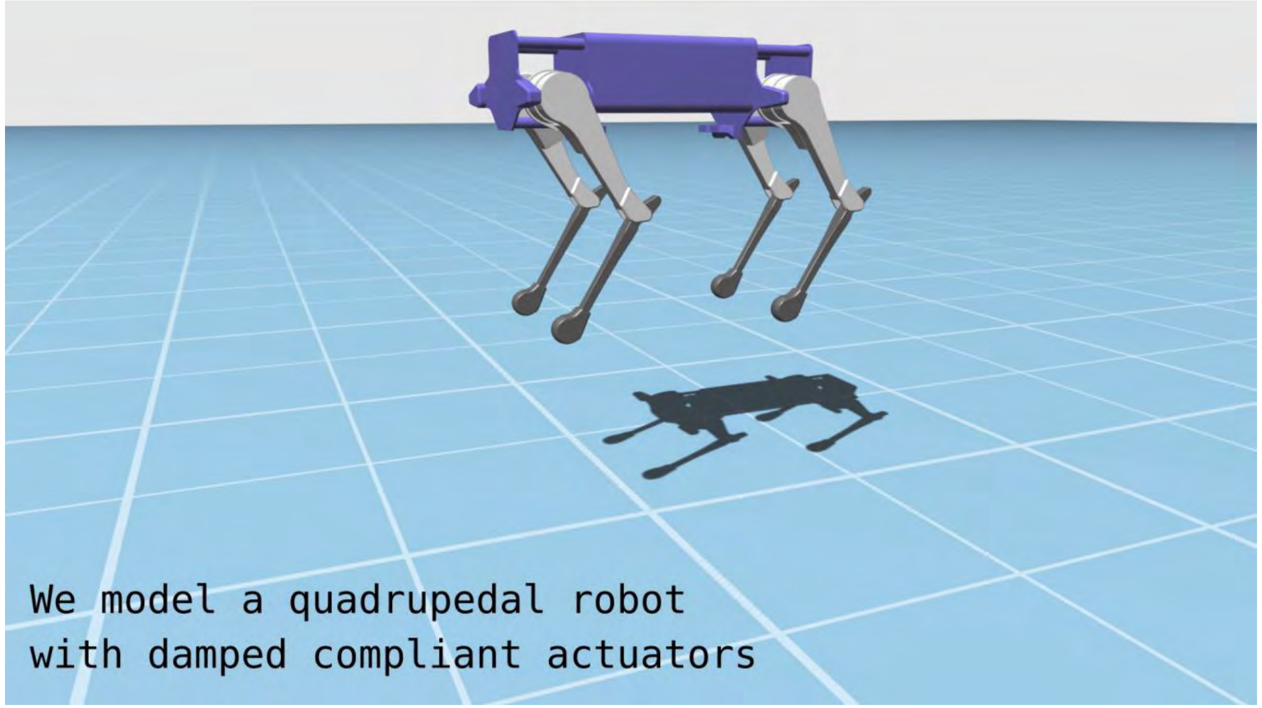
Optimization criterion	Decision Variables		
	control parameters	**policy parameters**	
Motion goals	whole body control; model predictive control; trajectory optimization	self-supervised learning; policy optimization	

[Tang et al., SIGGRAPH Asia 2020; Bern et al., SIGGRAPH 2017; Megaro et al., SIGGRAPH 2017]

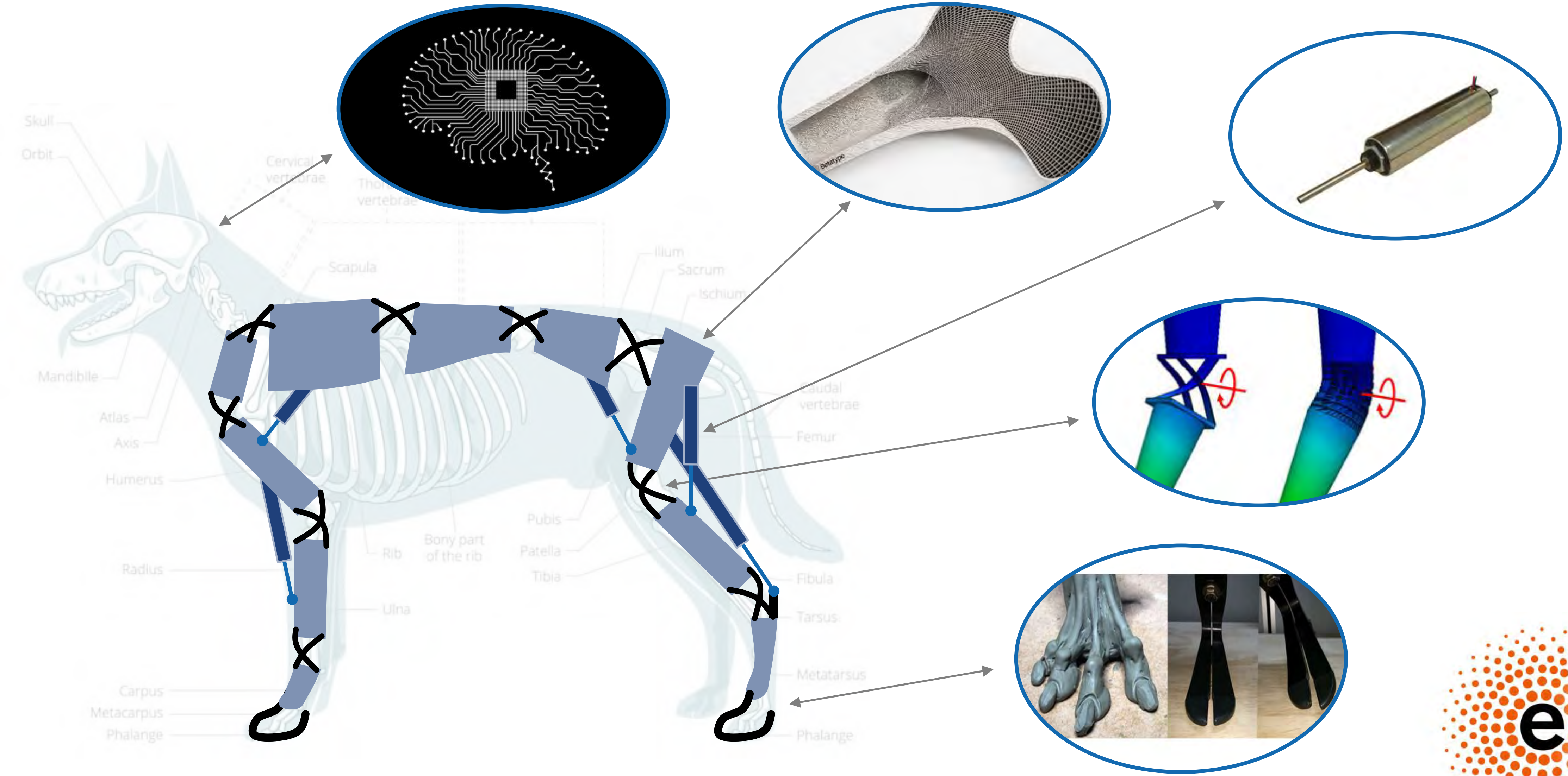
Locomotion for soft and hybrid rigid/compliant robots



[Bern et al., RSS 2019]



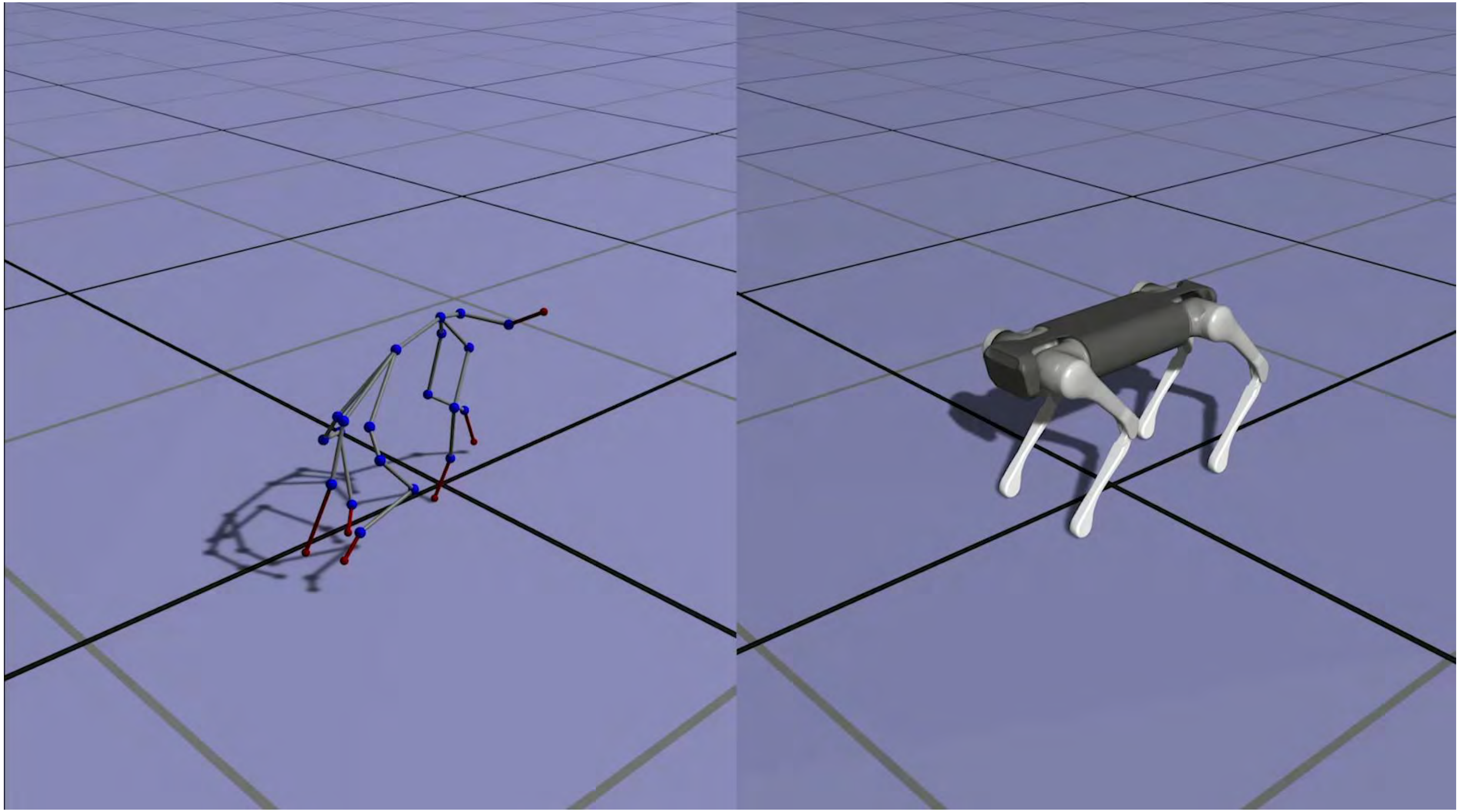
[Geilinger et al., SIGGRAPH Asia 2020]


Skull
Orbit
Cervical vertebrae
Thoracic vertebrae
Scapula
Ilium
Sacrum
Ischium
Mandibile
Atlas
Axis
Humerus
Caudal vertebrae
Femur
Pubis
Bony part of the rib
Rib
Patella
Tibia
Fibula
Radius
Ulna
Tarsus
Metatarsus
Carpus
Metacarpus
Phalange
Phalange

Differentiable Simulation: use cases

Optimization criterion	Decision Variables		
	control parameters	**policy parameters**	**model parameters**
Motion goals	whole body control; model predictive control; trajectory optimization	self-supervised learning; policy optimization	computational design

Motion tracking for lifelike quadrupedal robots

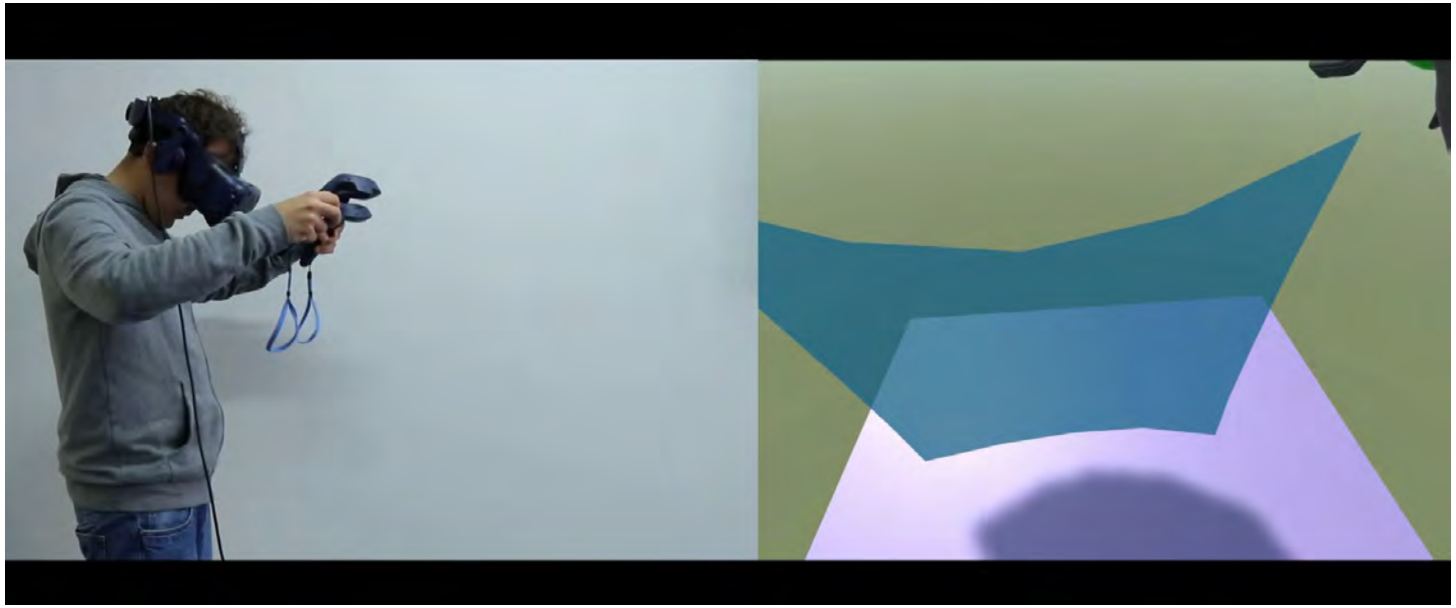


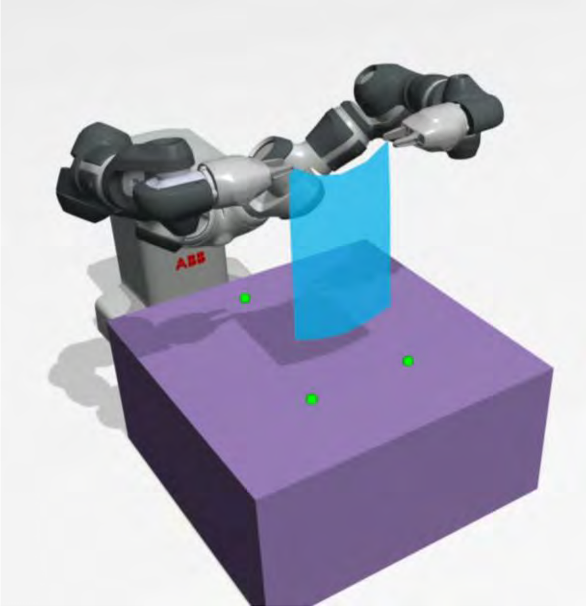
[Kang et al., IROS 2021]

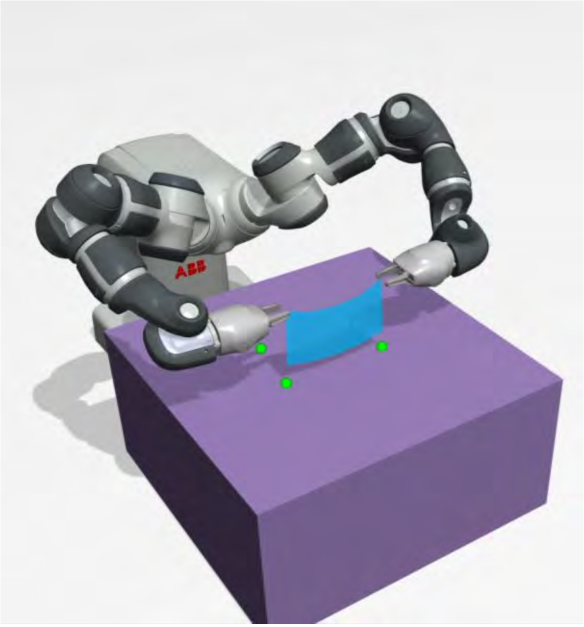
Differentiable Simulation: use cases

Optimization criterion	Decision Variables		
	control parameters	**policy parameters**	**model parameters**
Motion goals	whole body control; model predictive control; trajectory optimization	self-supervised learning; policy optimization	computational design
Model-data mismatch	motion tracking controllers; motion/pose estimation		

Demonstration learning and generalization







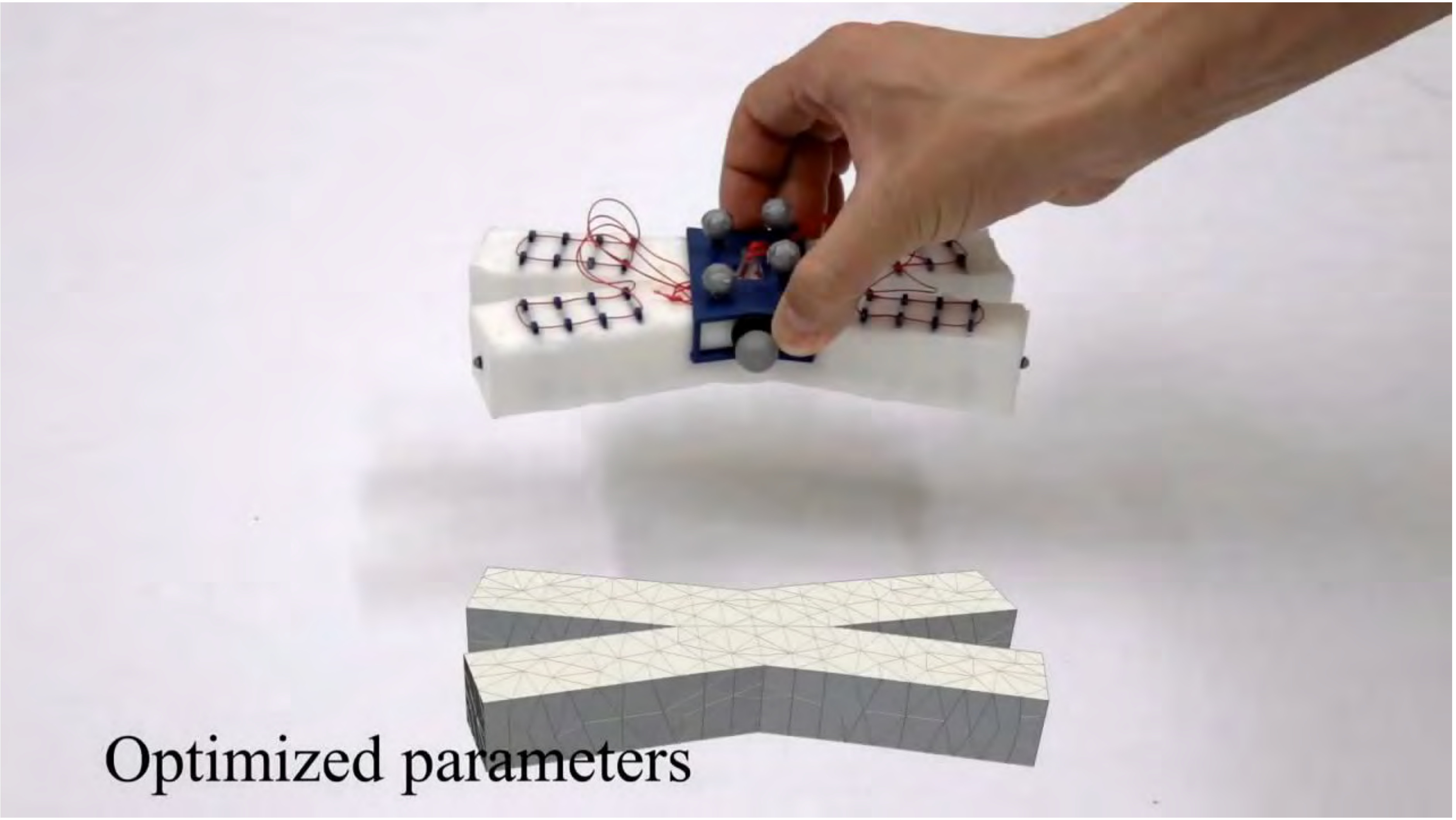
Differentiable Simulation: use cases

Optimization criterion	Decision Variables		
	control parameters	**policy parameters**	**model parameters**
Motion goals	whole body control; model predictive control; trajectory optimization	self-supervised learning; policy optimization	computational design
Model-data mismatch	motion tracking controllers; motion/pose estimation	learning by demonstration	

Real to sim

[Hahn et al., Siggraph Asia 2019]

Real to sim to real



[Hahn et al., Siggraph Asia 2019]

Differentiable Simulation: use cases

Optimization criterion	Decision Variables		
	control parameters	**policy parameters**	**model parameters**
Motion goals	whole body control; model predictive control; trajectory optimization	self-supervised learning; policy optimization	computational design
Model-data mismatch	motion tracking controllers; motion/pose estimation	learning by demonstration	real to sim

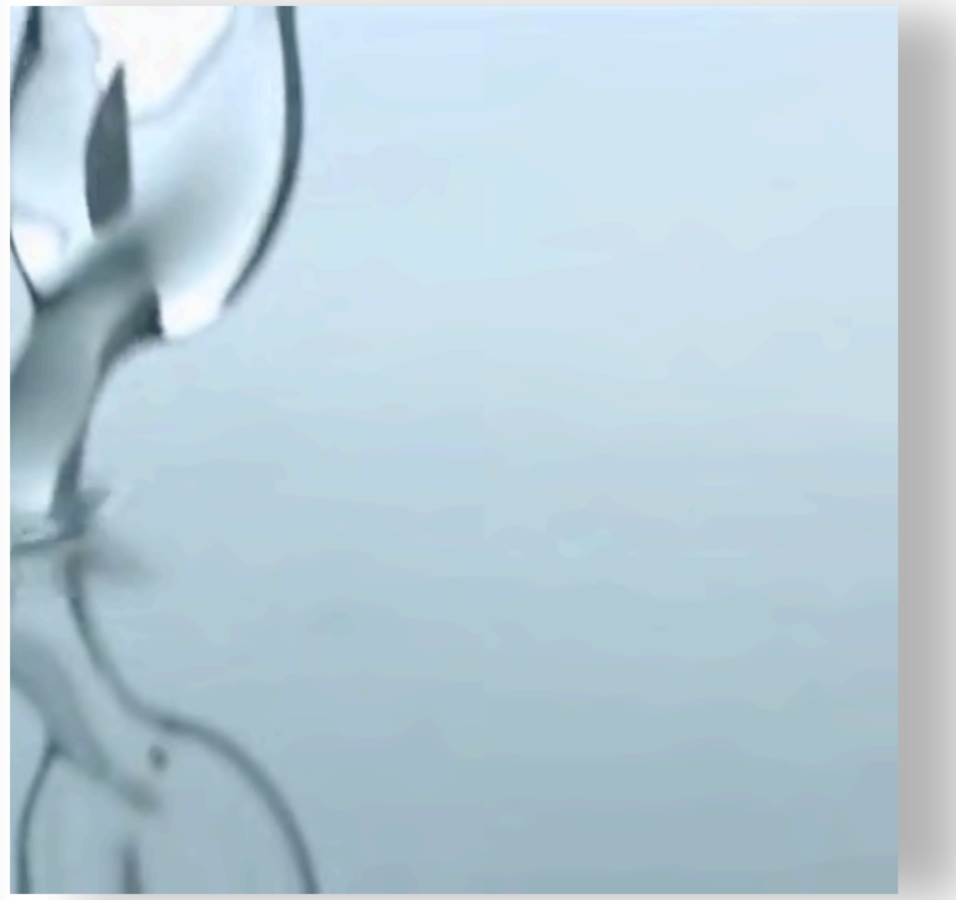
DIFFERENTIABLE SIMULATION AND DEEP LEARNING FOR FLUIDS

Nils Thürey

Physical Phenomena

Everywhere around us…

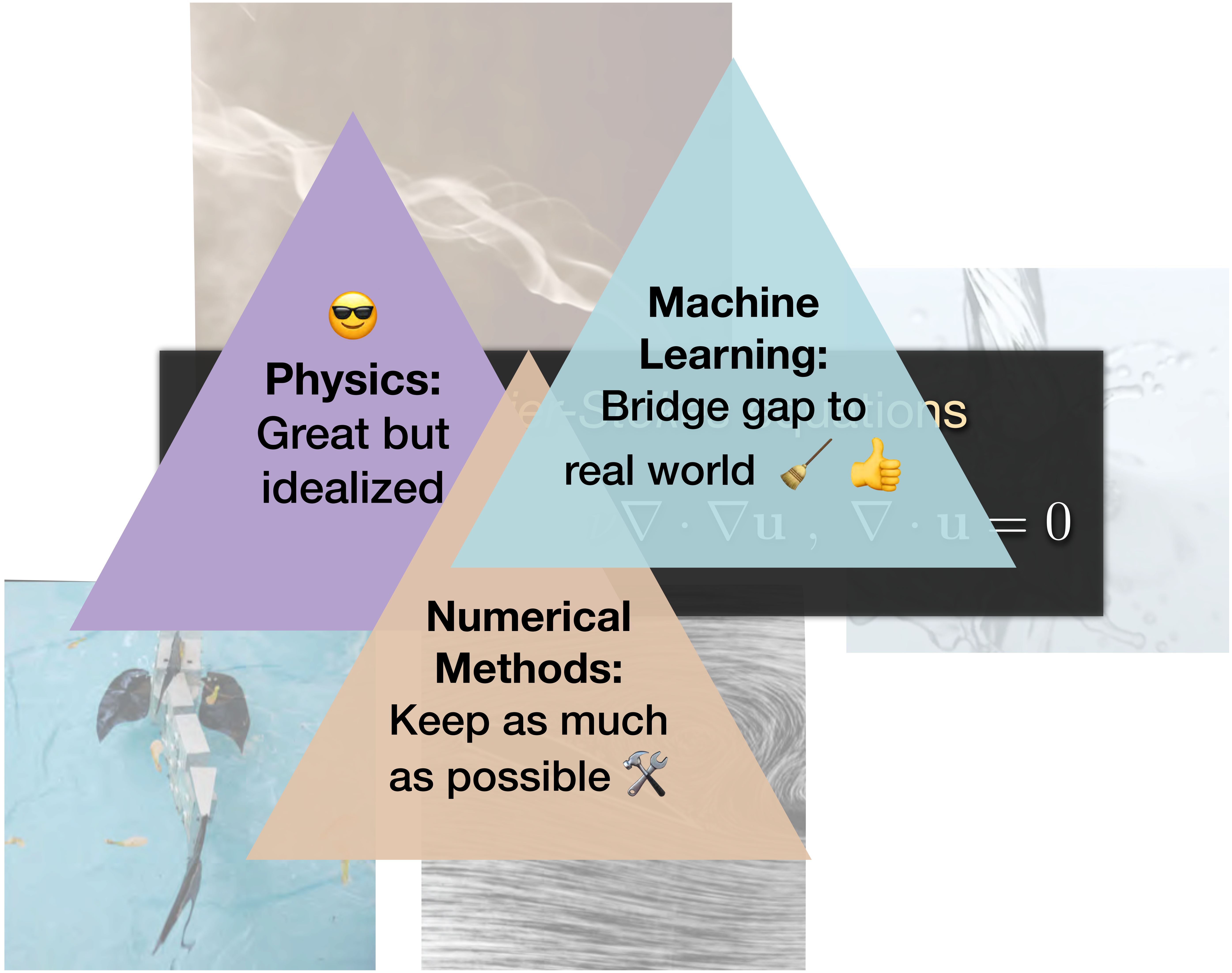
- Fluid Mechanics
- Robotic Control

Physical Phenomena

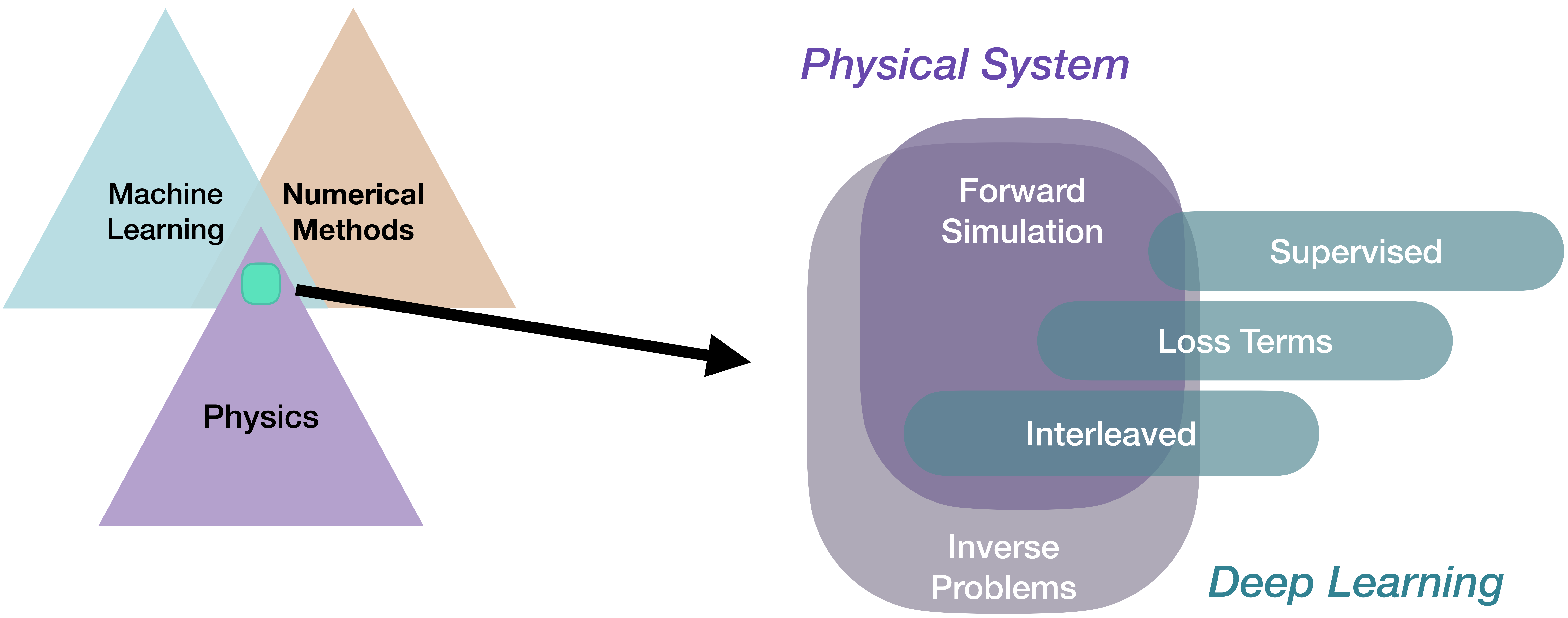
Everywhere around us…

- Fluid Mechanics
- Robotic Control
- Thermodynamics
- Plasma Physics
- Medical Simulations
- Many more…



Physics-Based Learning

How to combine?



Related & Own Work

- *Holl et. al*: Learning to Control PDEs with Differentiable Physics
- *Um et. al*: Solver-in-the-Loop: Learning from Differentiable Physics to Interact with PDE-Solvers
- *Bar-Sinai et. al*: Learning data-driven discretizations for partial differential equations
- *Raissi et. al:* Hidden physics models: Machine learning of nonlinear partial differential equations
- *Chen et. al.*: Neural ordinary differential equations
- *Morton et. al:* Deep dynamical modeling and control of unsteady fluid flows

“Differentiable Physics”

Overview

Discretized PDE $\mathcal{P}$ with phase space states $\mathbf{s}$

Learn via gradient $\partial\mathcal{P}/\partial\mathbf{s}$

Requires differentiable physics simulator for $\mathcal{P}$

Note: not a soft-constrained residual loss of $\mathcal{P}$

→ Tight integration of numerical methods and learning process

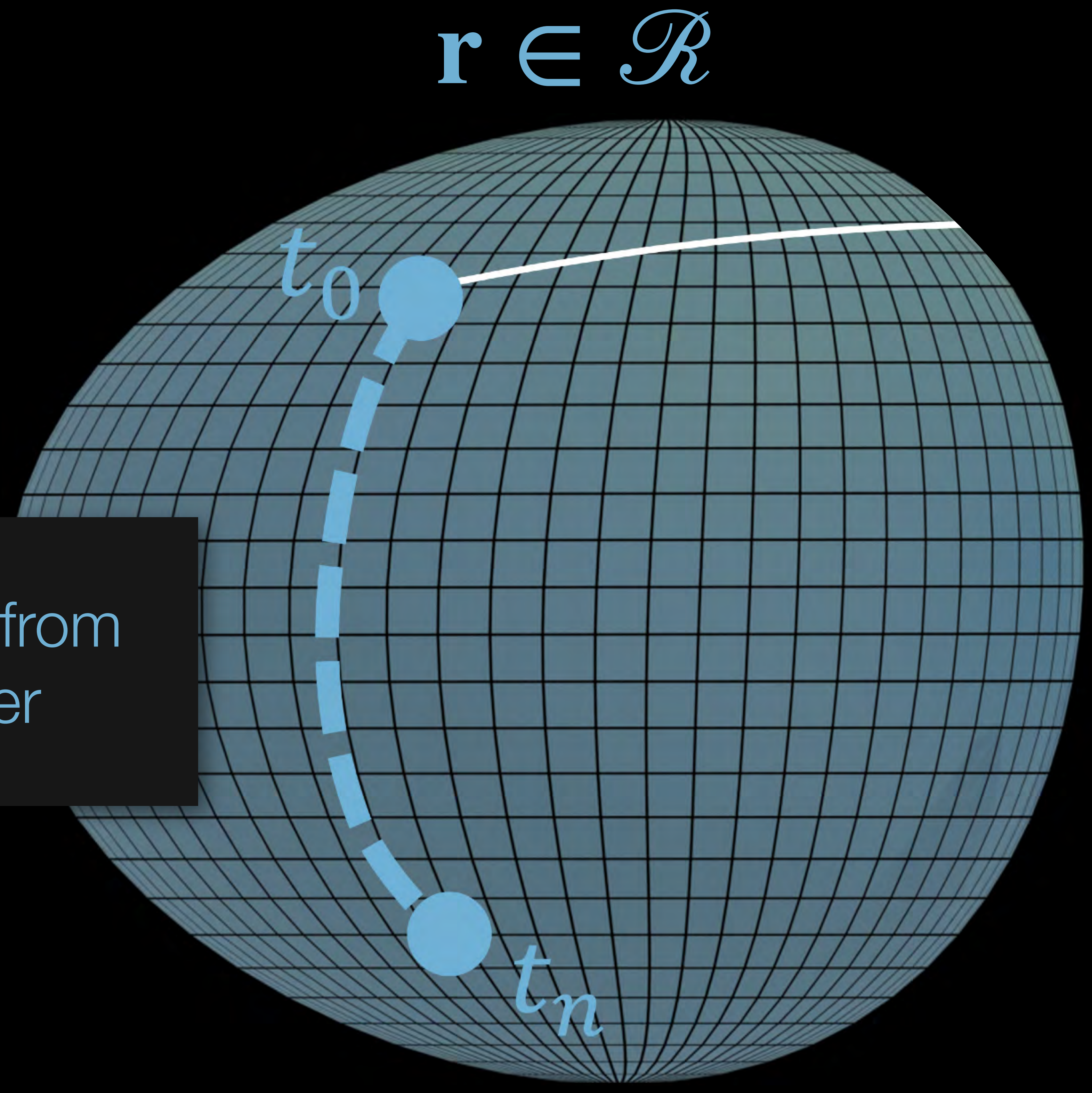
Differentiable Physics Example 1

Um et. al: Solver-in-the-Loop: Learning from Differentiable Physics to Interact with PDE-Solvers

Reducing Numerical Errors

"Solver-in-the-Loop"

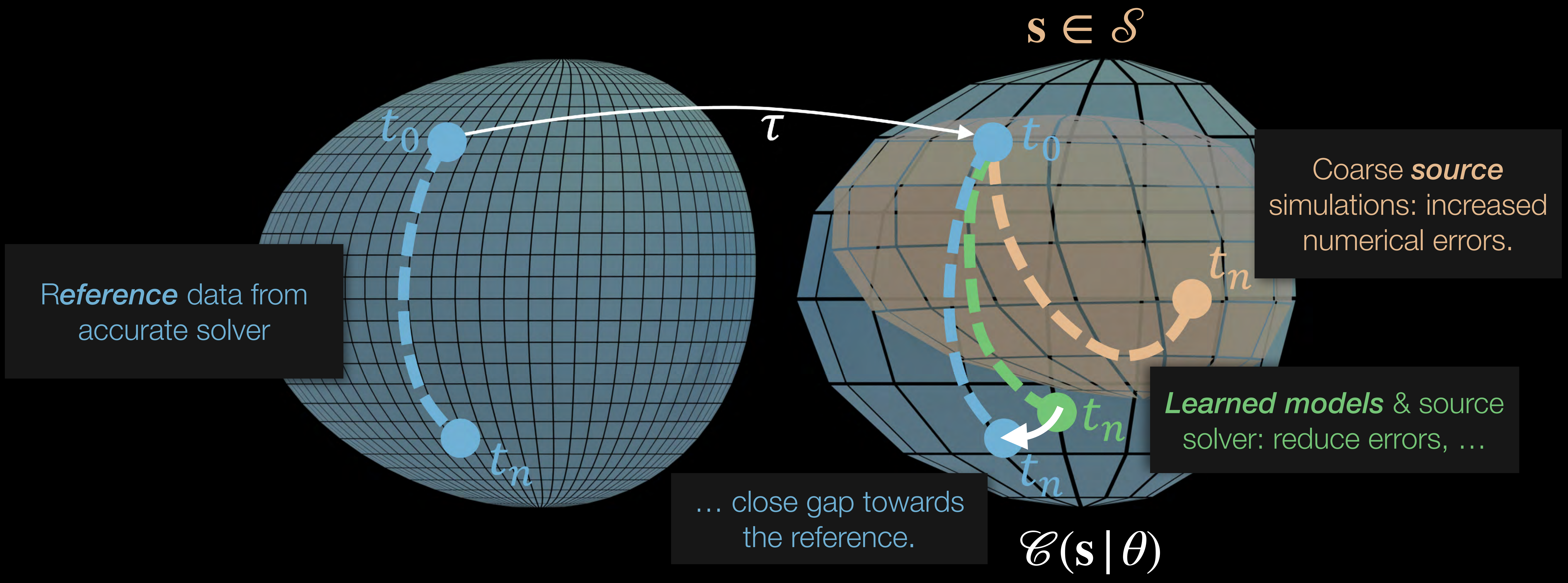
PDE: $\mathscr{P}$



Reference data from accurate solver

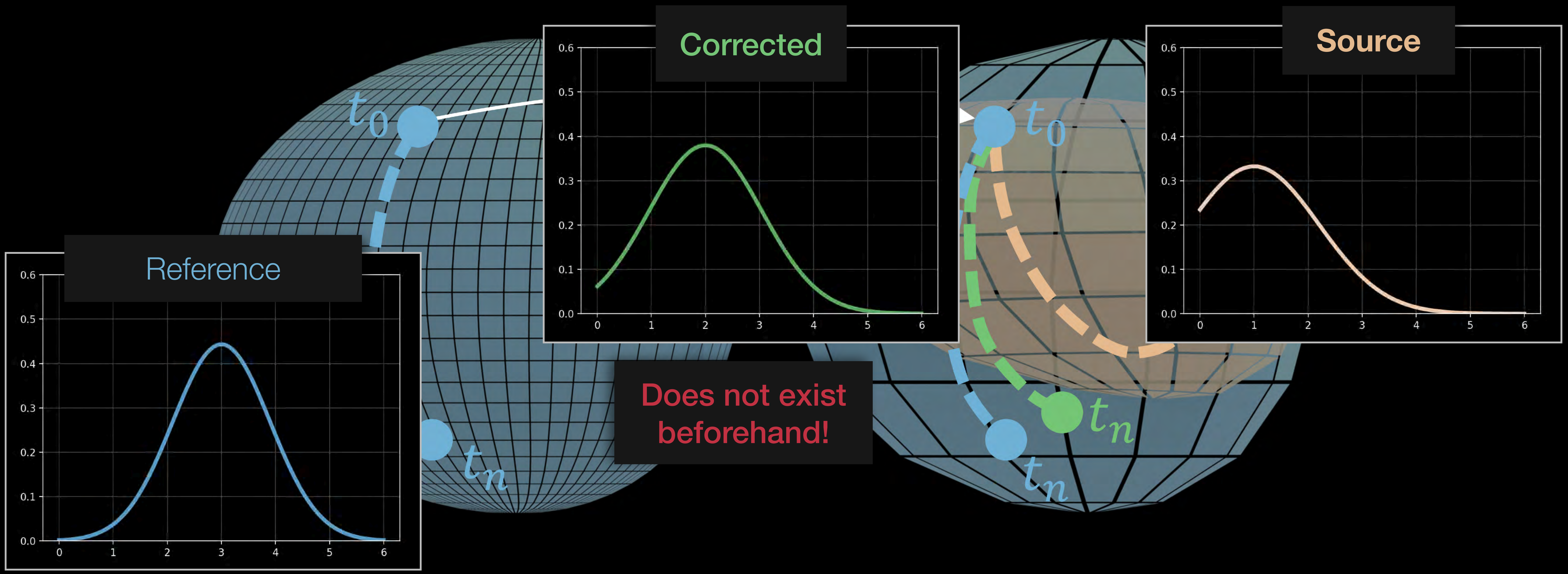
Reducing Numerical Errors

"Solver-in-the-Loop"



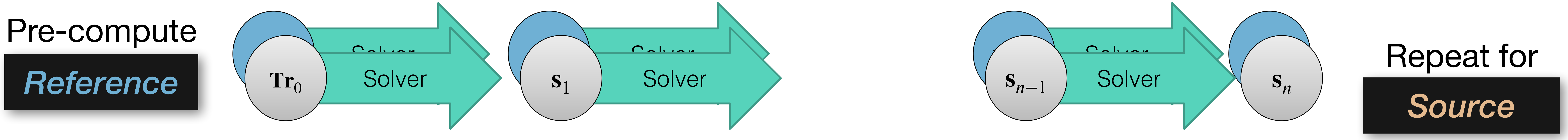
Reducing Numerical Errors

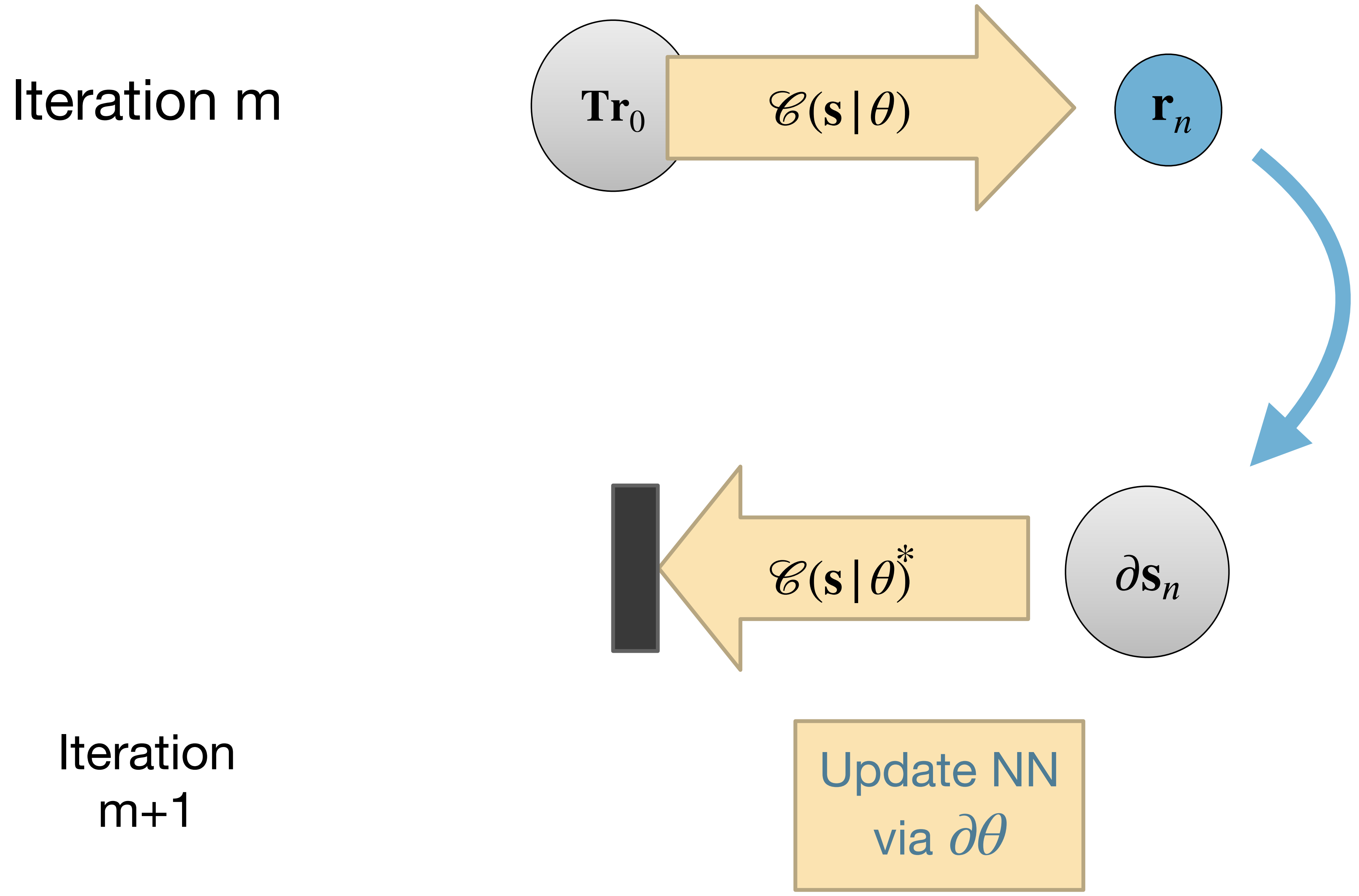
Shift of Input Feature Distributions



Reducing Numerical Errors

Supervised Learning from Physical PDEs





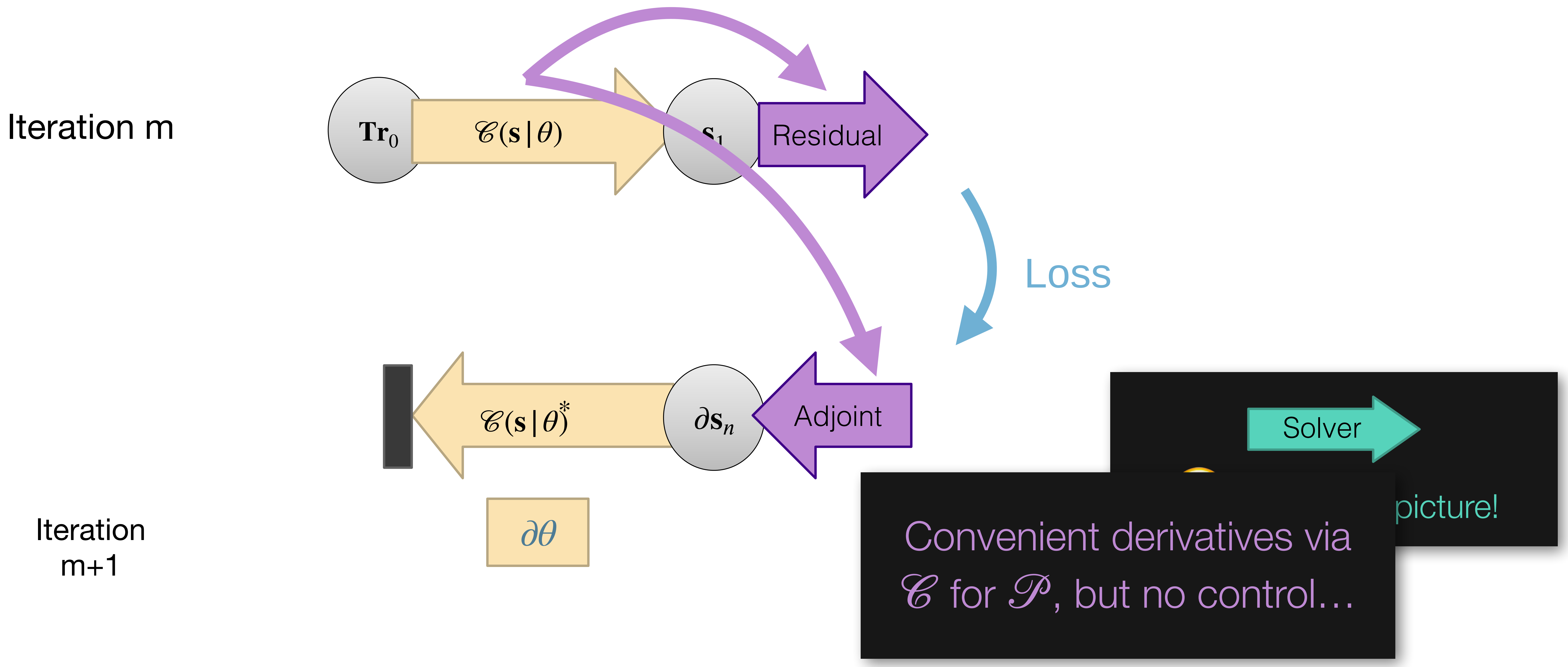
Initial state from reference, i.e. $\mathbf{s}_0 = \mathbf{Tr}_0$

Fully supervised loss: $\|\mathbf{s}_i + \mathscr{C}(\mathbf{s}_i\,|\,\theta) - \mathbf{Tr}_i\|^2$

Reducing Numerical Errors

Partial Interaction via Physical Residuals

Reduced variant: Include residual of $\mathscr{P}$ in loss formulation
(*physics-constrained* / *physics-informed*)



Reducing Numerical Errors

Differentiable Physics Solvers

Provide $\partial \mathscr{P} / \partial \mathbf{s}$ via physics solver

Leverage existing NM for accurate & reliable discretization

Right "*granularity*" can require custom operations

E.g.: Poisson solve with $A = \nabla \cdot \nabla$, derivative for $\partial A^{-1}b/\partial b = A^{-1}$

Chain together via AutoDiff

Highly efficient solvers available!

Reducing Numerical Errors

Learning Objective with Differentiable Physics

Modified state after n steps $\tilde{\mathbf{s}}_{t+n} = (\mathcal{P}_s\mathcal{C})^n\ (\mathbf{Tr}_t)$

Correction function $\mathcal{C}(\tilde{\mathbf{s}}\,|\,\theta)$ depends on states modified by $\mathcal{P}_s\mathcal{C}$

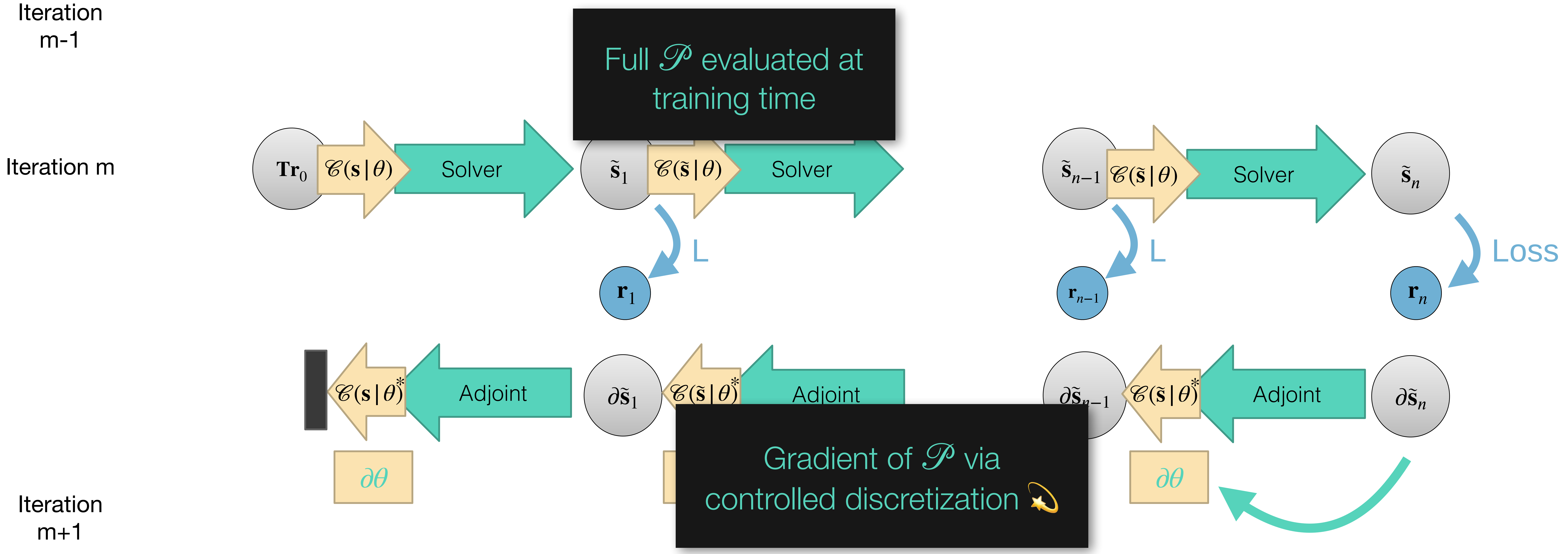
Objective for differentiable physics training:

$$\text{argmin}_\theta \sum_t \sum_{i=0}^{n-1} \|\mathcal{P}_s(\tilde{\mathbf{s}}_{t+i}) + \mathcal{C}(\mathcal{P}_s(\tilde{\mathbf{s}}_{t+i})\,|\,\theta) - \mathbf{Tr}_{t+i+1}\|^2$$

Reducing Numerical Errors

Learning via Differentiable Physics

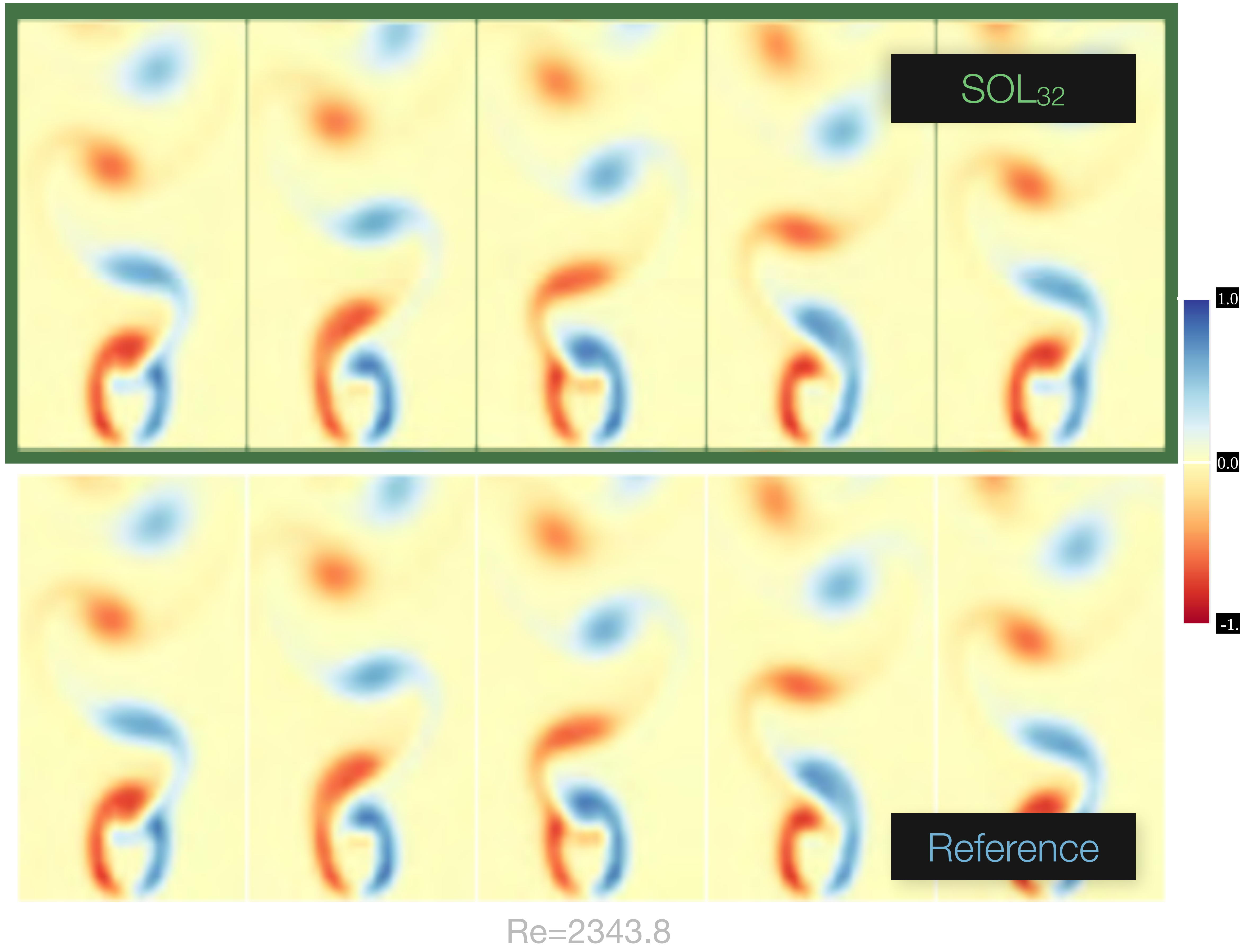
Correction via network for each unrolled simulation step $\mathscr{C}(\tilde{\mathbf{s}}\,|\,\theta)$



A few more Details…

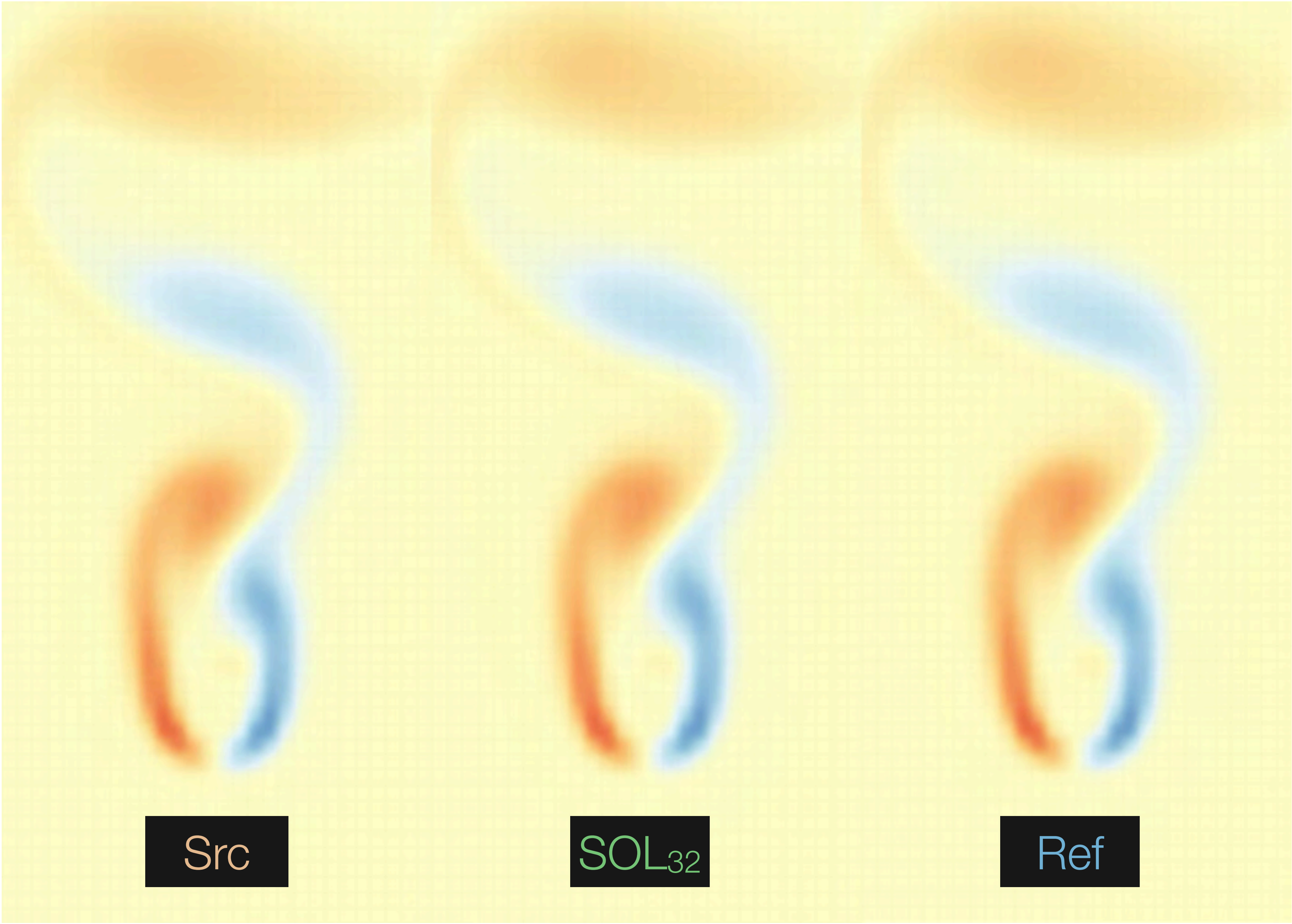
Unsteady Wake Flow in 2D

- Setup: Reference is 4x
- 3000 frames training data, Re $\in \{98 \;..\; 3125\}$
- Test data: new Re Nr.s
- Source MAE: 0.146
- SOL_{32} MAE: 0.013
- More than 10x reduction

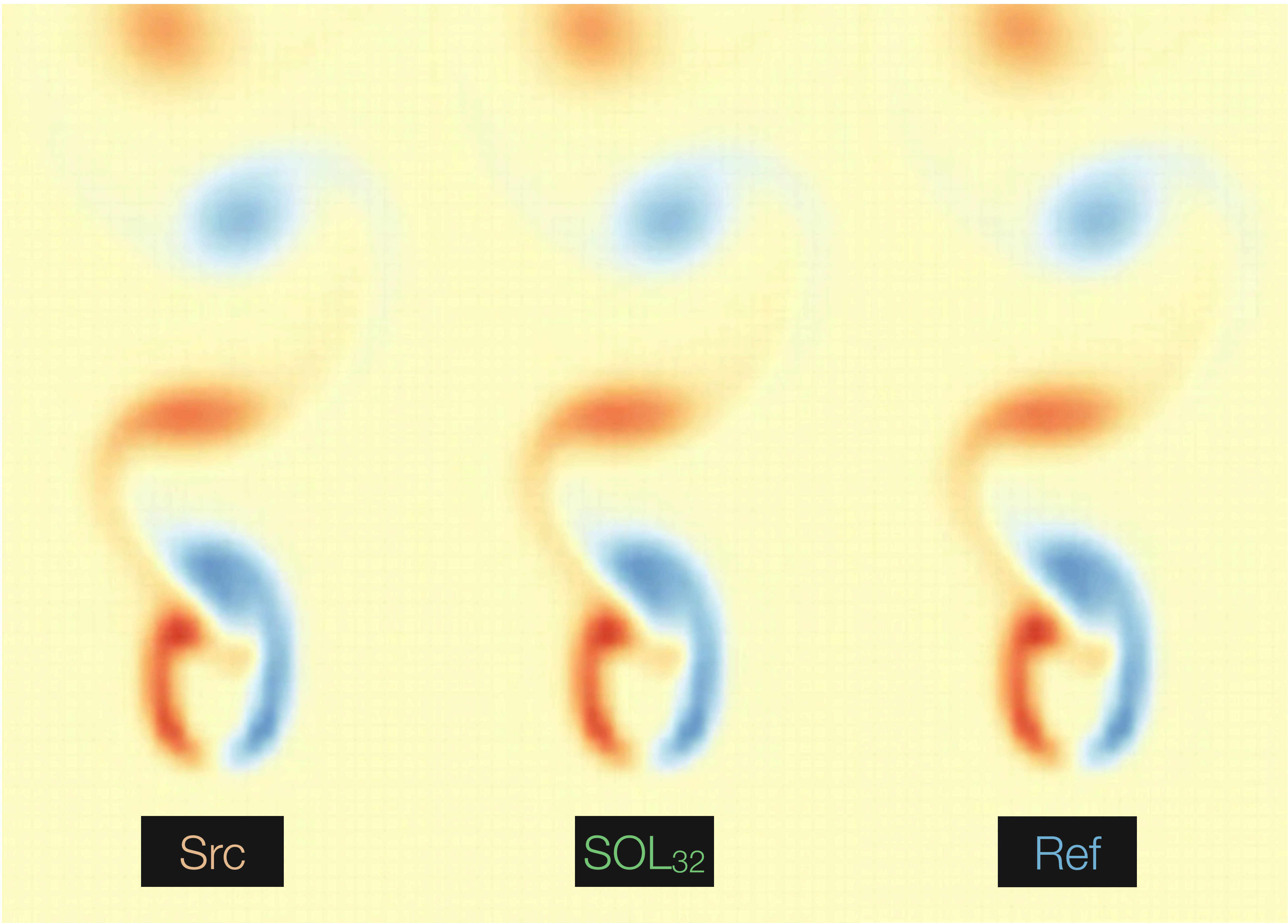


Unsteady Wake Flow 2D

Evaluation Details in the Paper - Two Test Samples in Motion:



Re=146.5



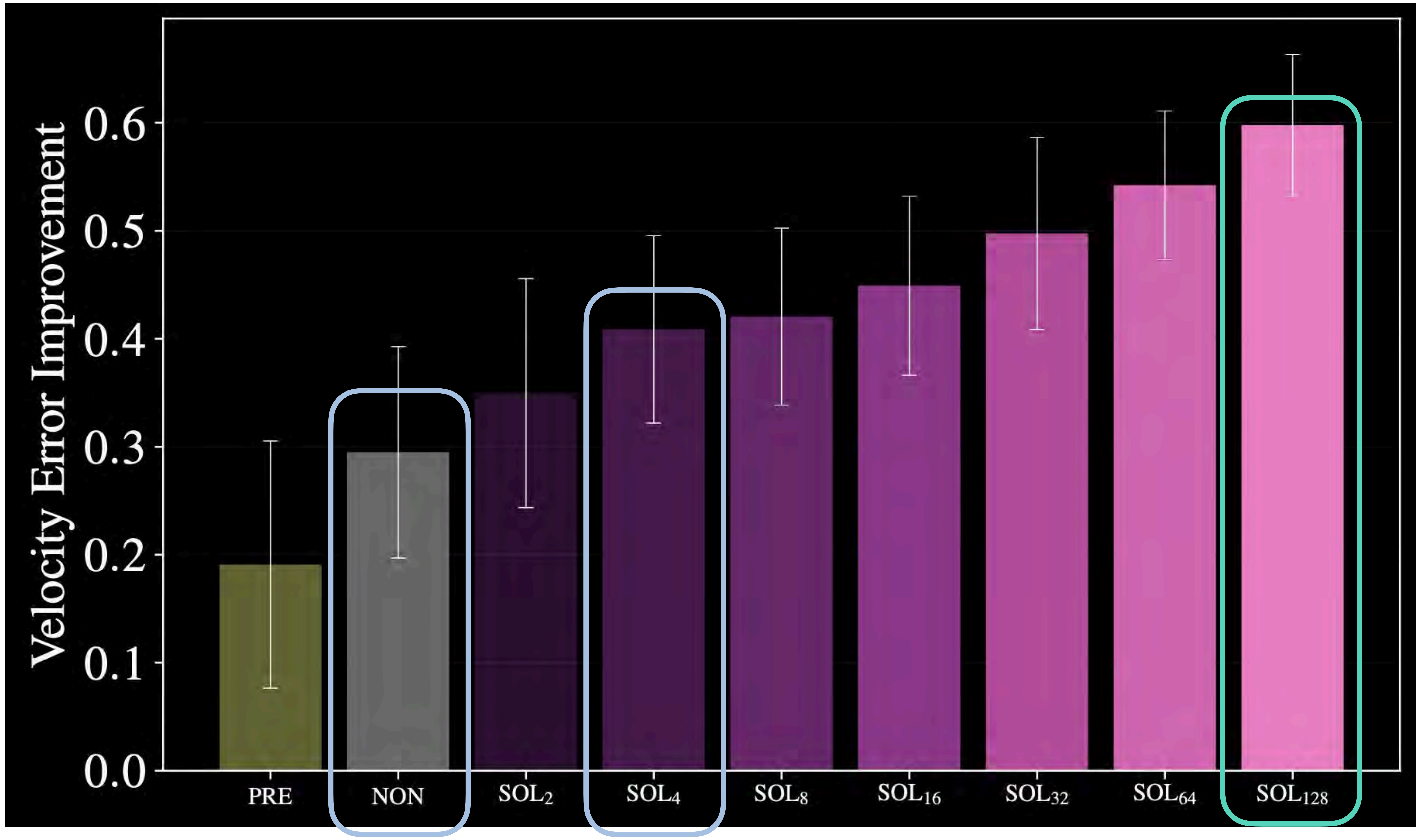
Re=2343.75

Looking into the Future

Learning via a Large Number of Simulation Steps

Evaluation:

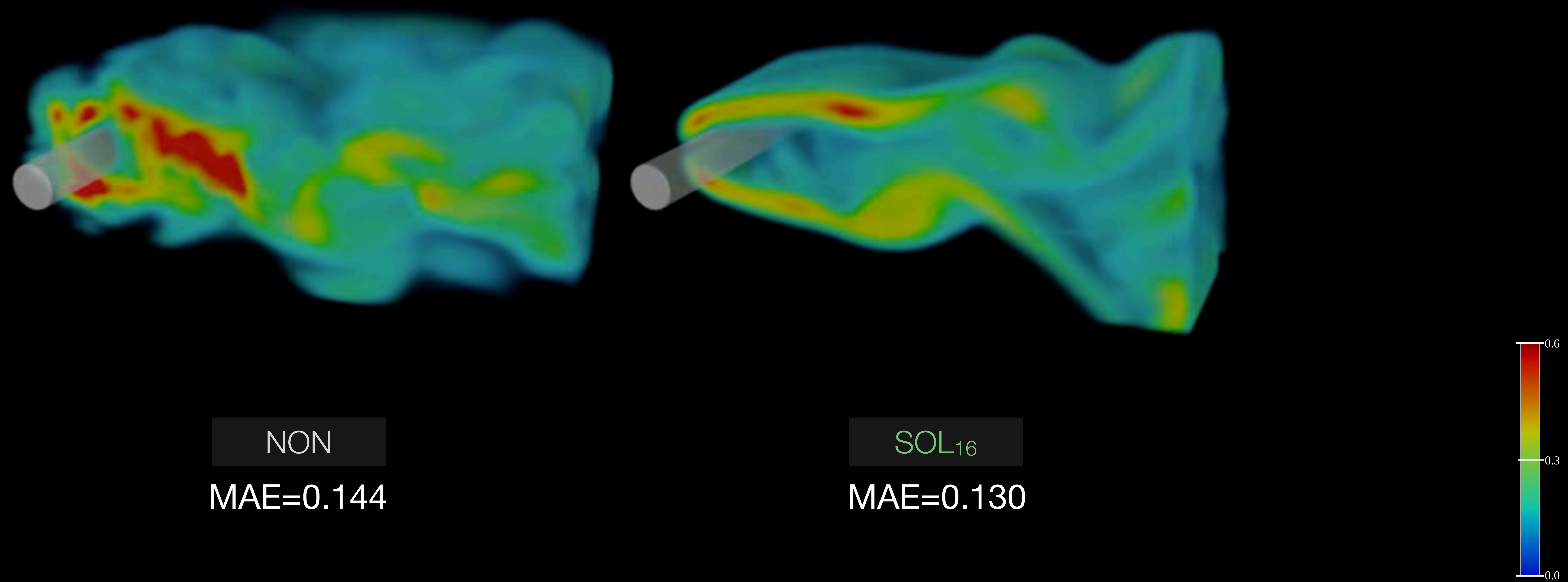
- MAE Improvement over Src
- Supervised training: 29%
- D.P. with 4 steps: 41%
- D.P. with 128 steps: 60%



Long-term Stability

Unsteady Wake Flow (250 time steps)

3D Test Case, Re=468.8



3D Results

Unsteady Wake Flow

Second 3D Test Case, Re=546.9

MAE=0.167

MAE=0.130

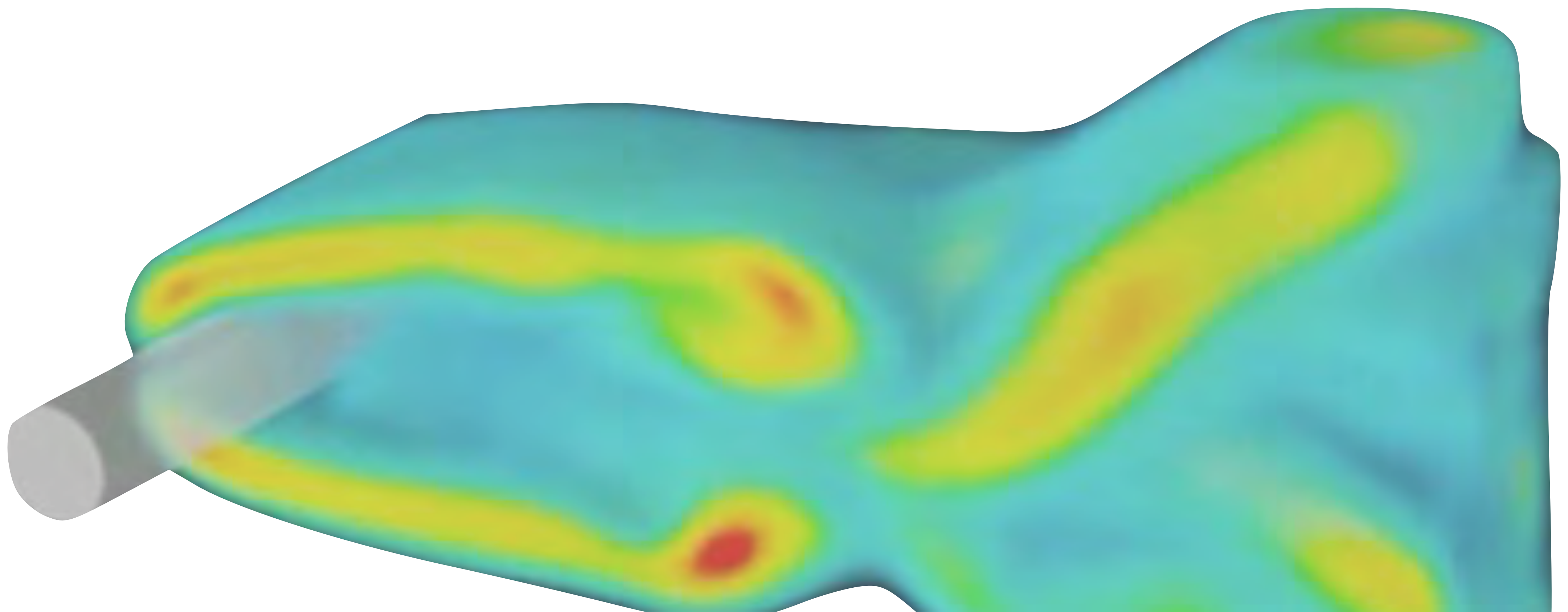
Performance

3D Wake Flow - Avg. Runtimes for 100 time steps

- CPU-based reference simulation: 913.2 seconds
- Source solver with CNN: 13.3 seconds

  ➡ *Speed-up of 68x*

- Future hardware support (e.g., A14/M1)



Differentiable Physics Example 2

Holl et. al: Learning to Control PDEs with Differentiable Physics

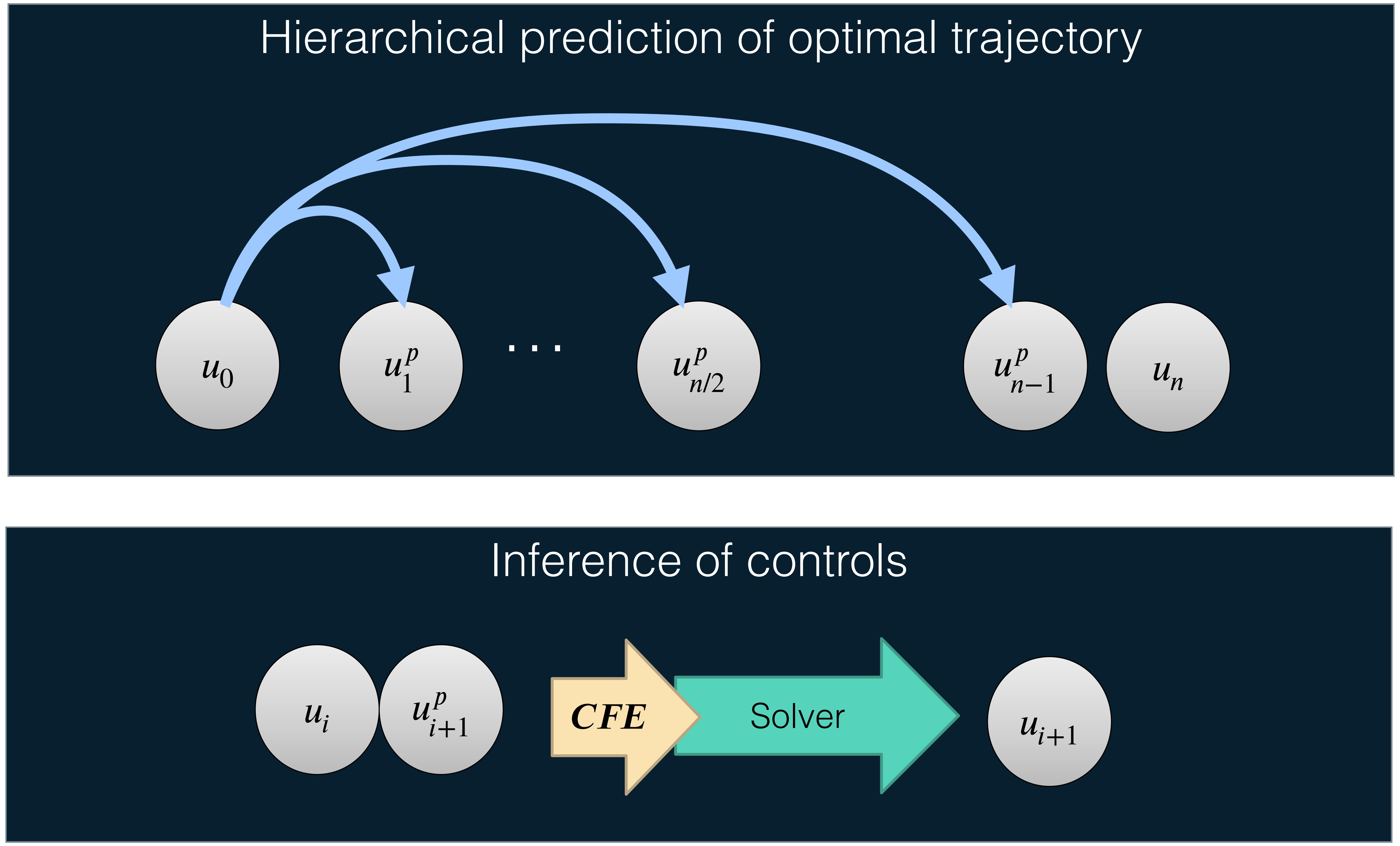
Solving Control Problems

Solving Control Problems

Long-term Planning

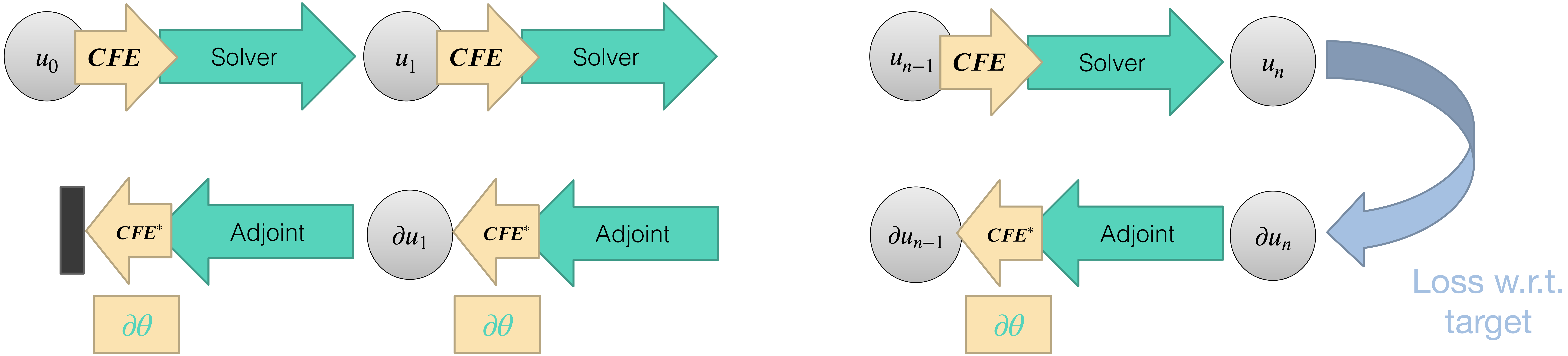
Task 1: Predict

Task 2: Correct



Solving Control Problems

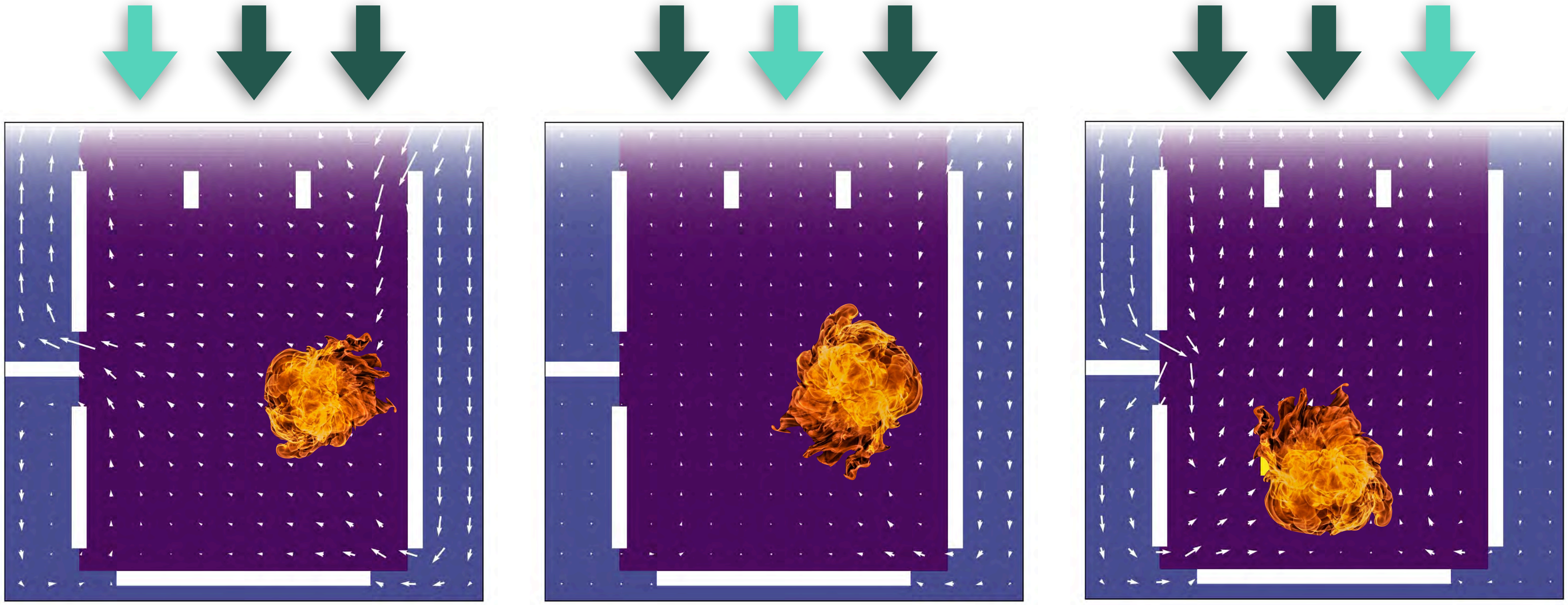
Learning Control Forces (*CFE* Network)



2D Navier-Stokes

Move Marker (Yellow) to Target Region

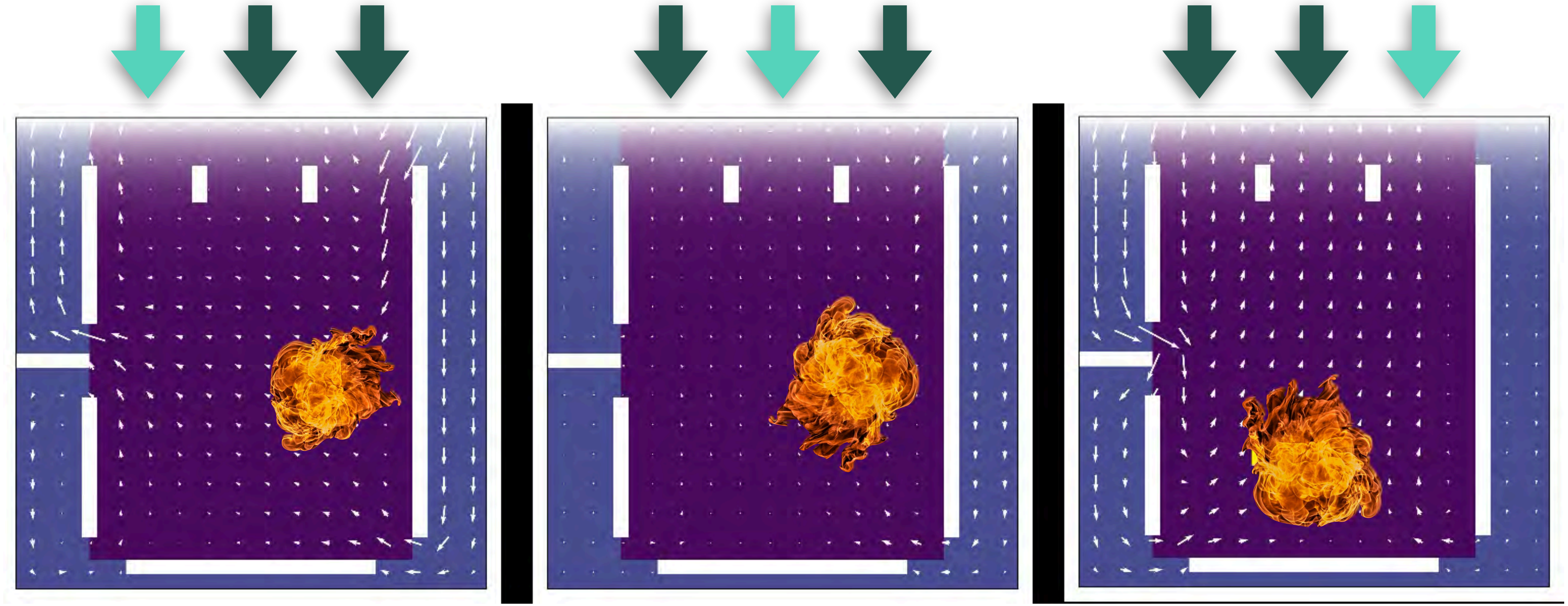
Indirect Control in Blue Region ("*Ventilation*")



2D Navier-Stokes

Move Marker (Yellow) to Target Region

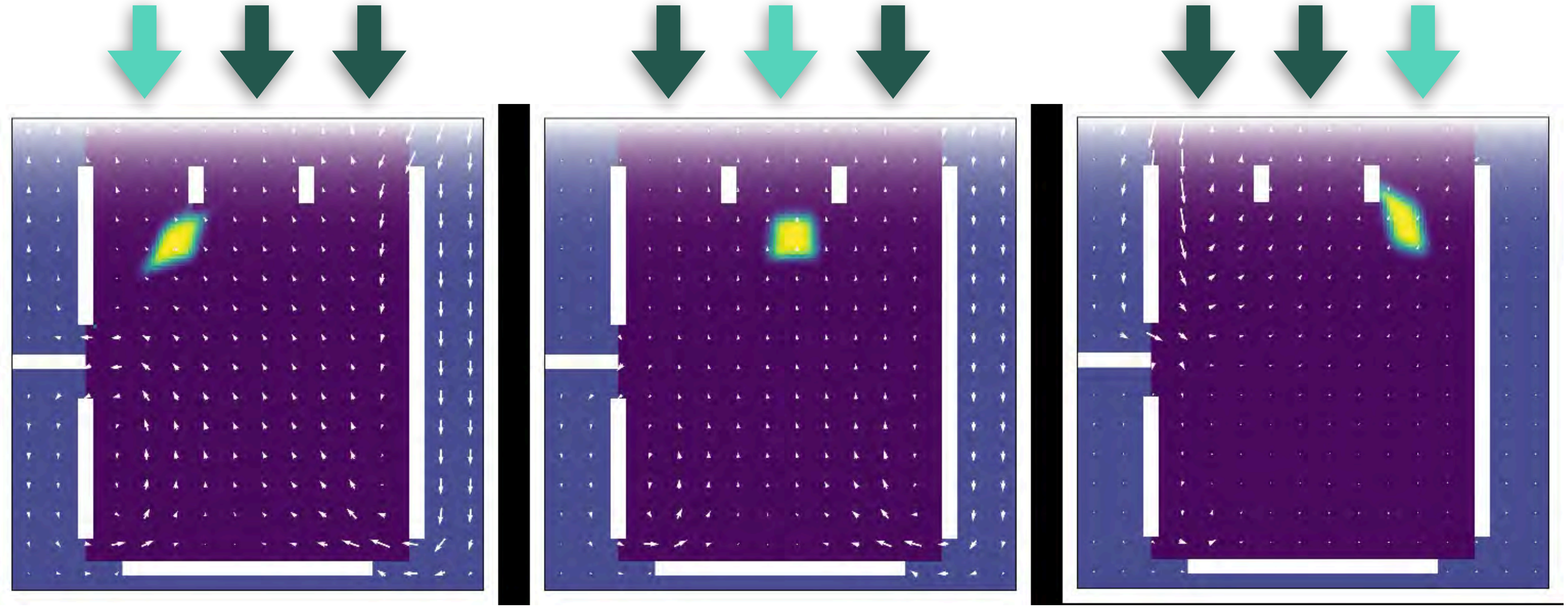
Indirect Control in Blue Region (“*Ventilation*”)



2D Navier-Stokes

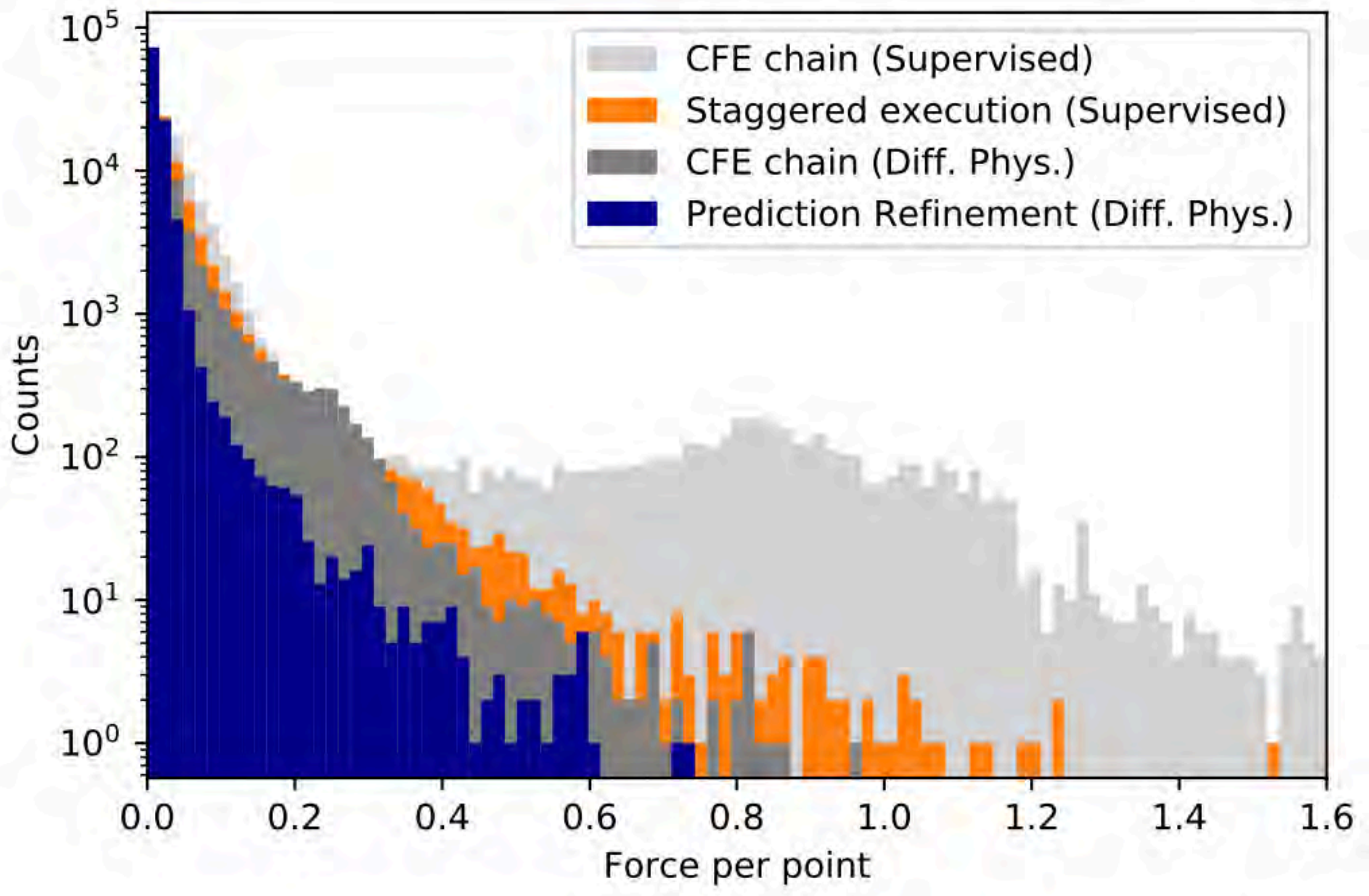
Move Marker (Yellow) to Target Region

Indirect Control in Blue Region (“*Ventilation*”)



Quantitative Evaluation

(Additional Details in the Paper)



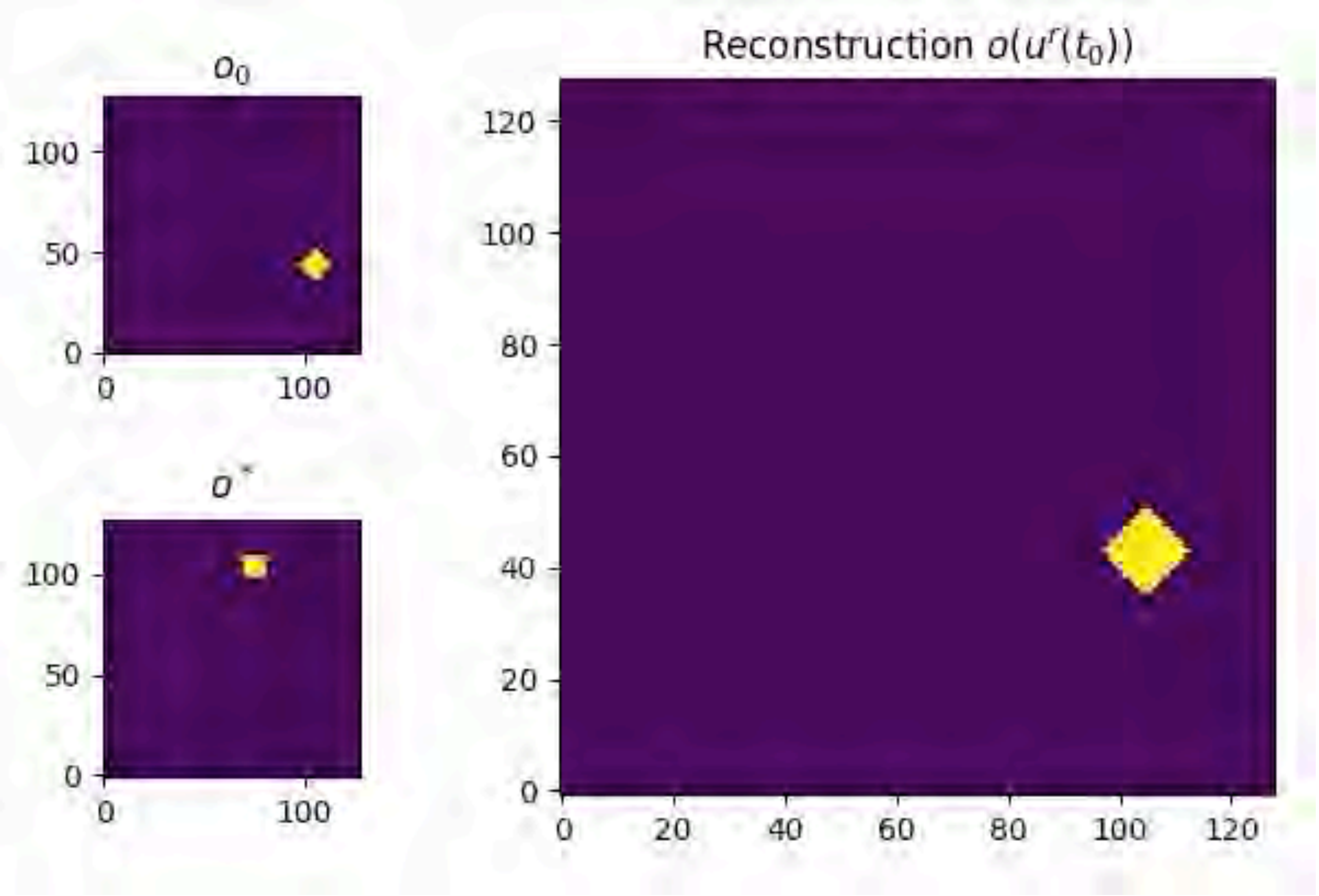
- CFE chain (Sup): 83.4 ± 2.0

- CFE chain (D.P.): 28.8 ± 0.8

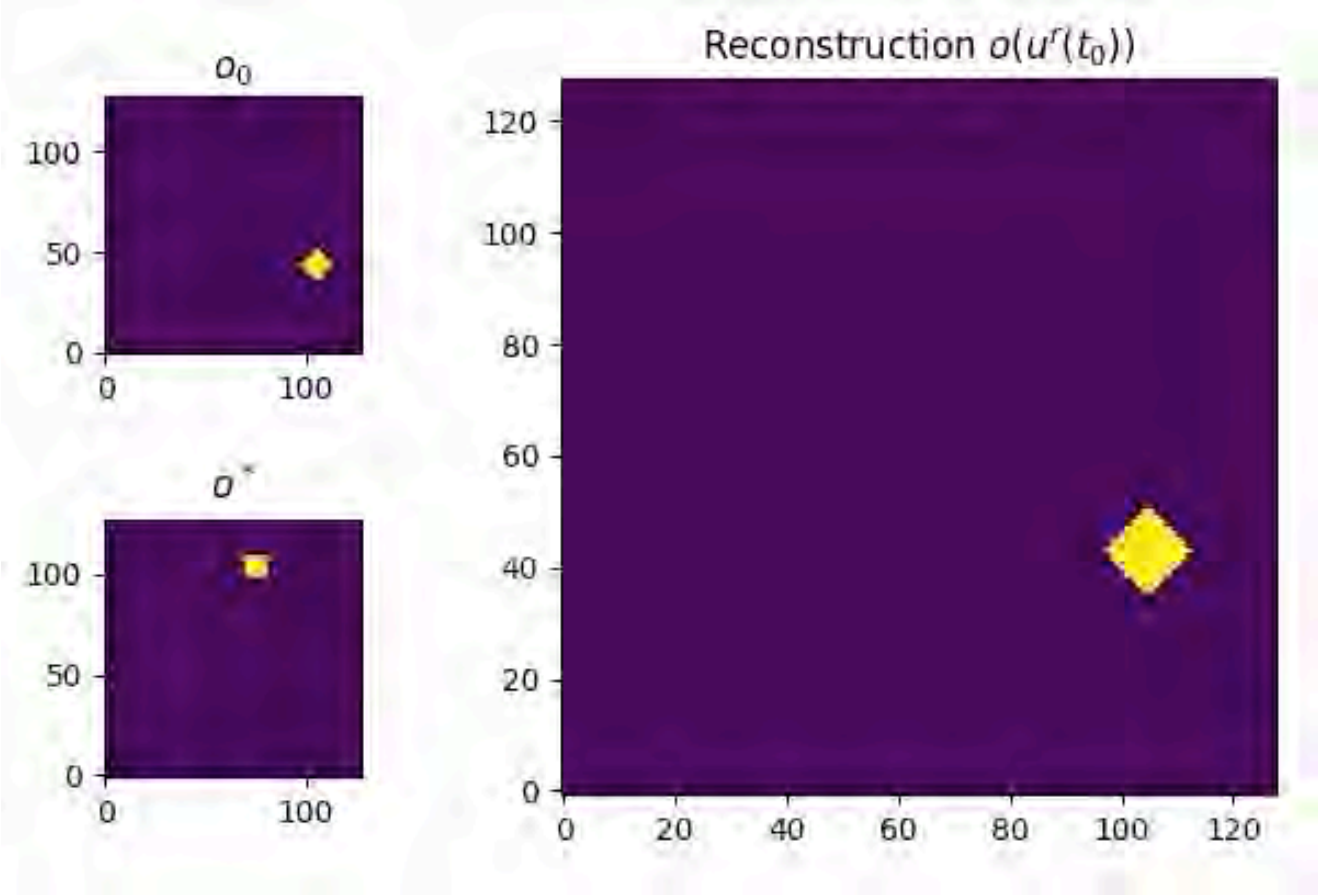
- Prediction Ref. (D.P.): 14.2 ± 0.7

Versus Classical Optimization

Why not just use Adjoint Method & Co? 🤨



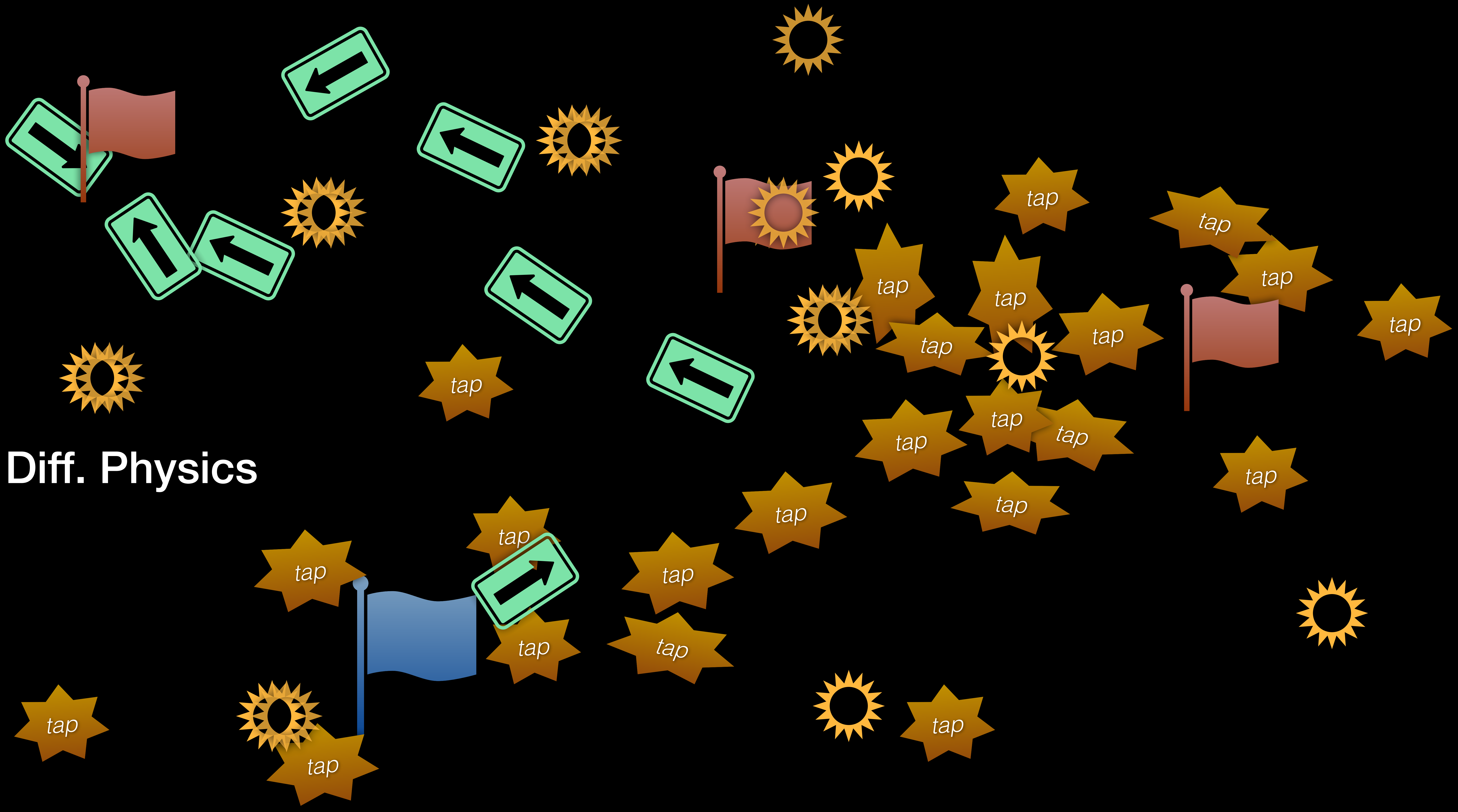
→ **Expensive, gets stuck in local minima**
(ca. 131s & 1500 steps)



→ **Learns "global" view via solution manifold**
(ca. 0.5s)

Discussion - Non-linear Optimization

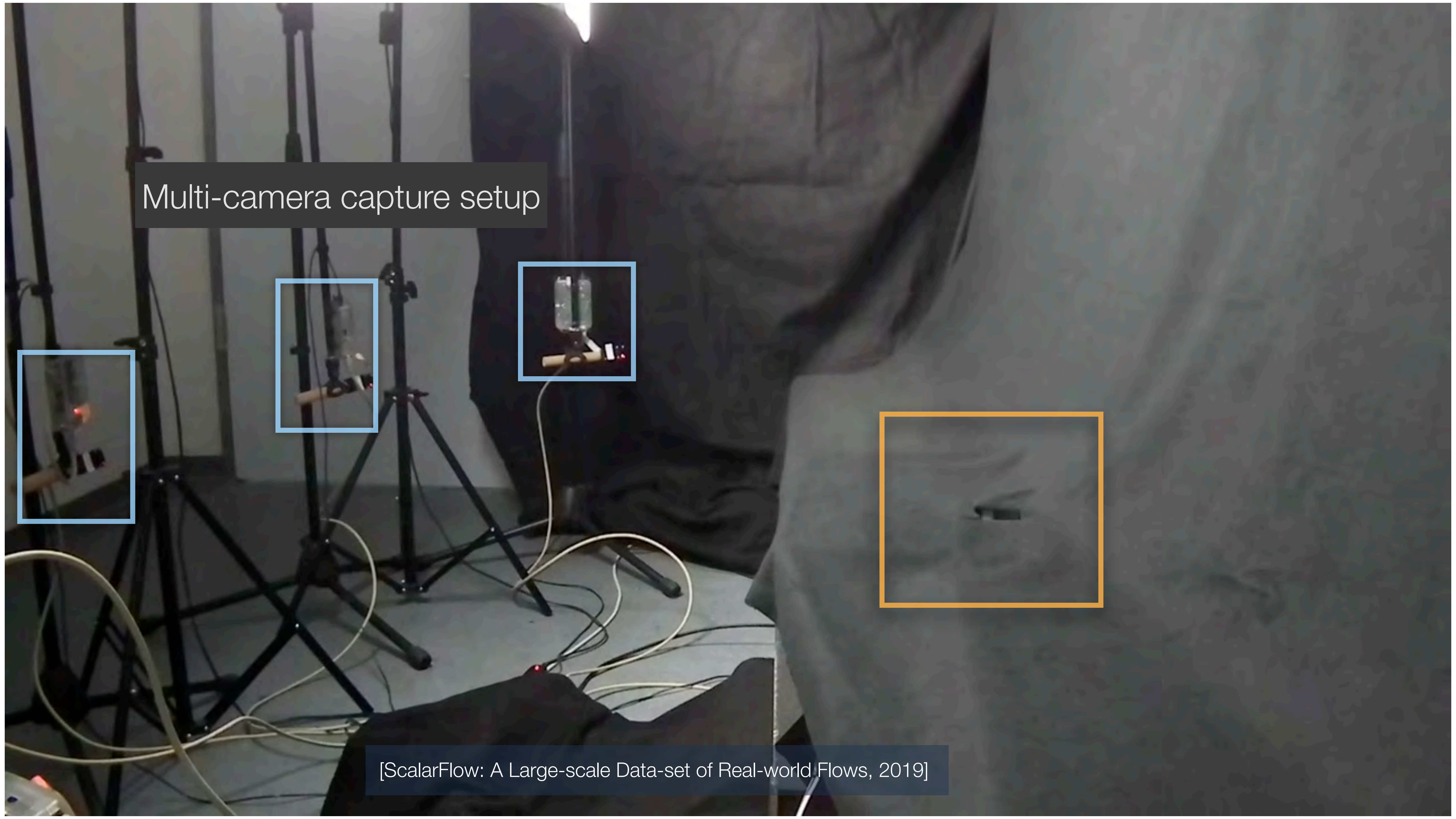
Supervised Learning
Differentiable Physics

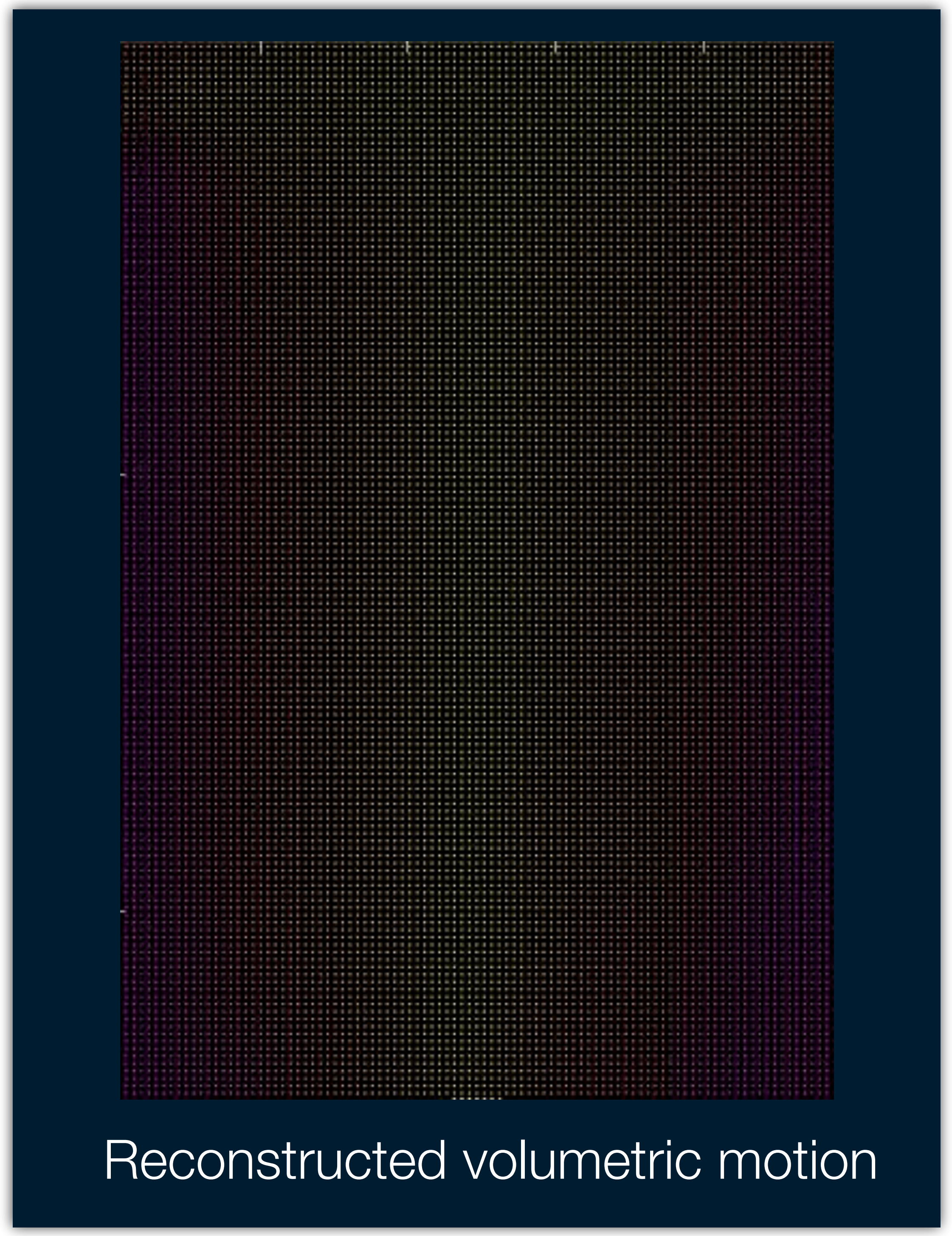


Supervised + Diff. Physics

Outlook

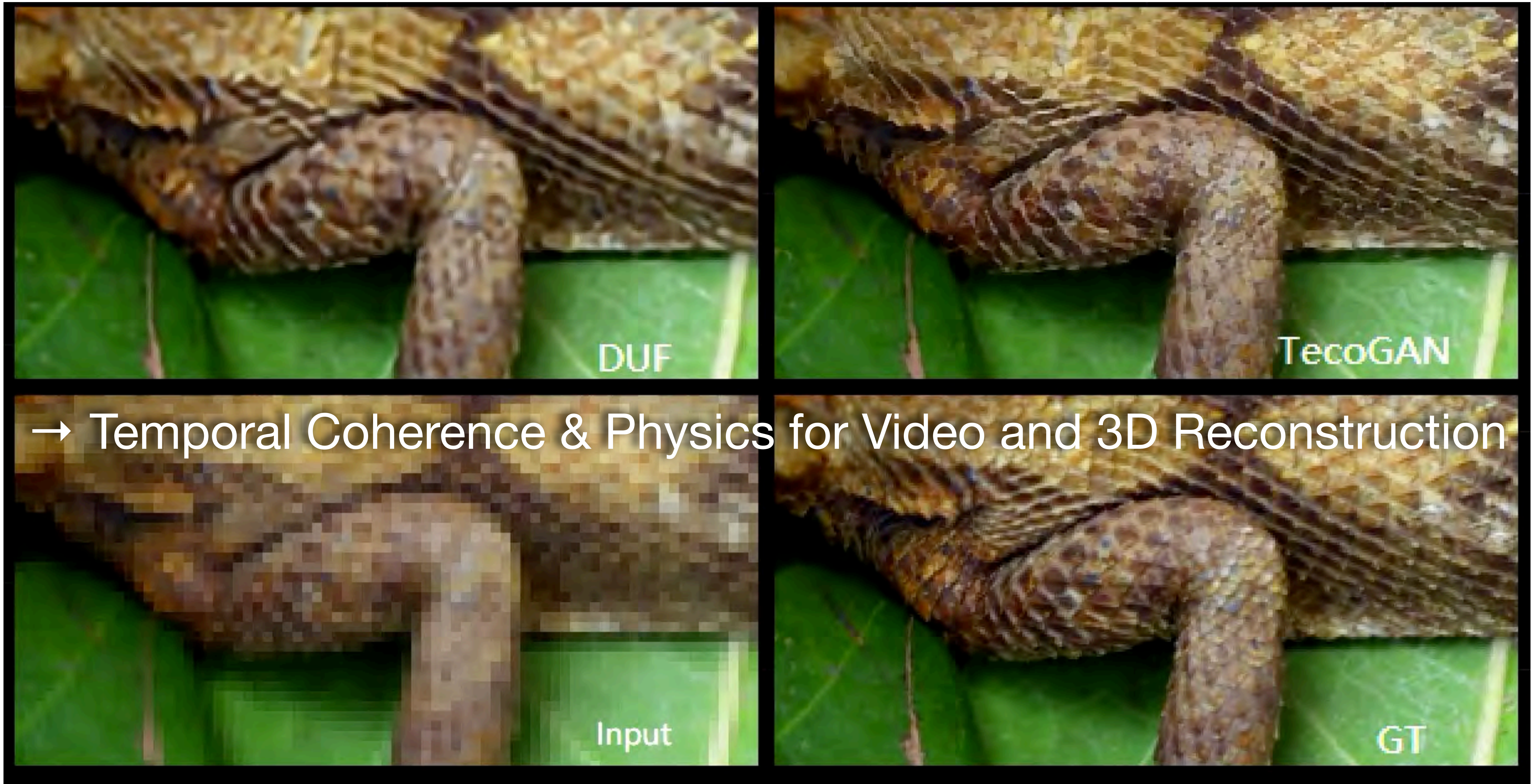
Learn from Real-world Observations





Outlook

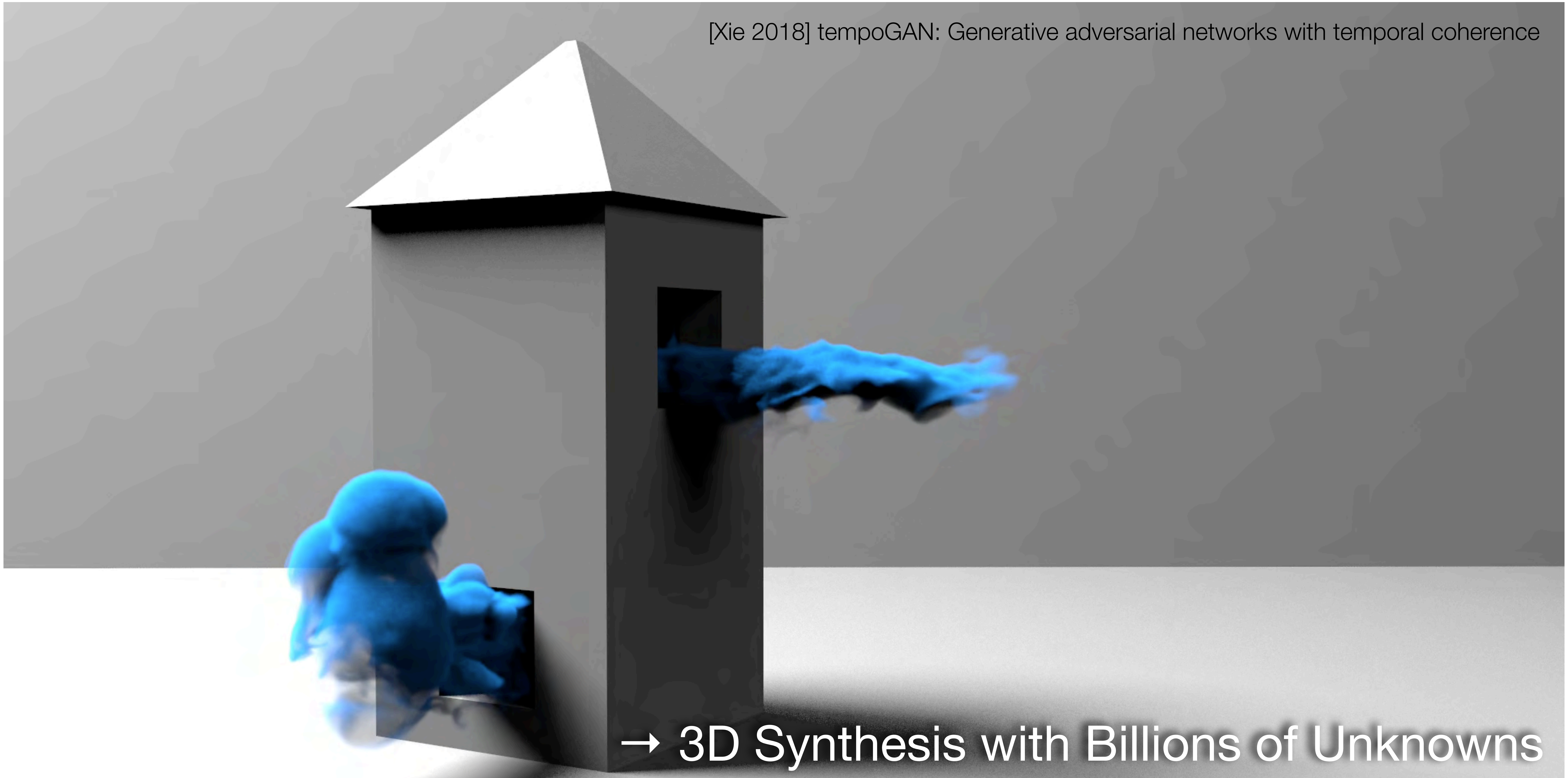
Synergies between Vision & Graphics



→ Temporal Coherence & Physics for Video and 3D Reconstruction

Outlook

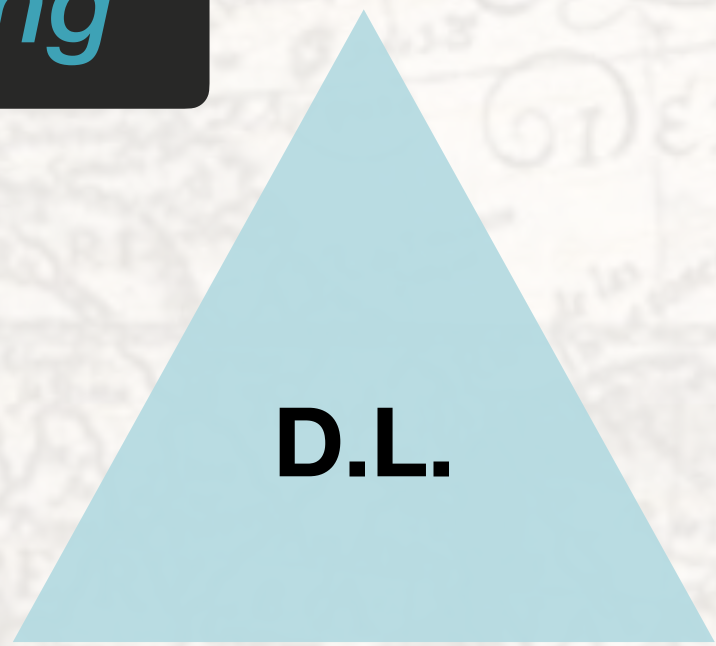
Large-scale Effects



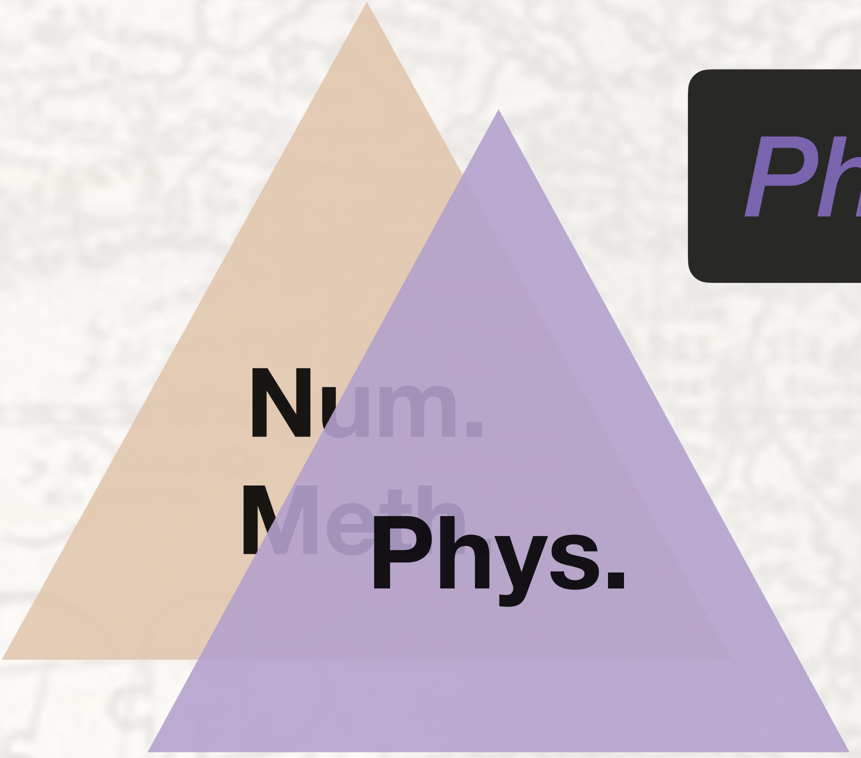
Summary

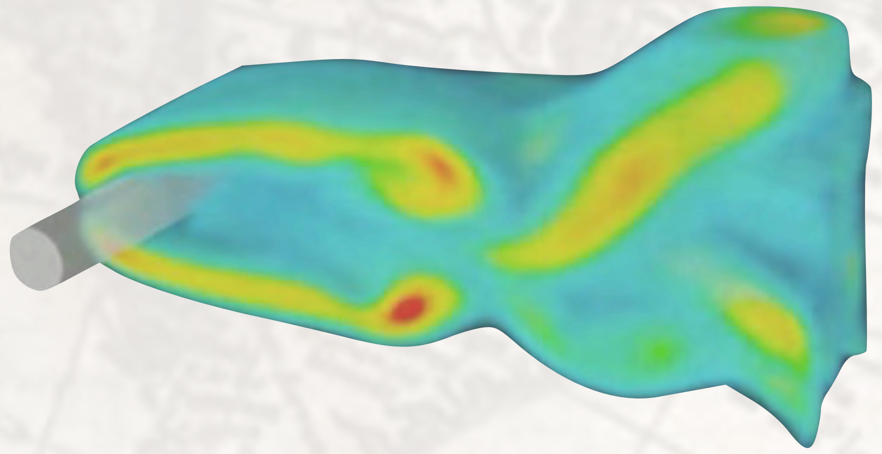
Differentiable Simulations as Tool to bridge Physics & Learning 🤗

Deep Learning

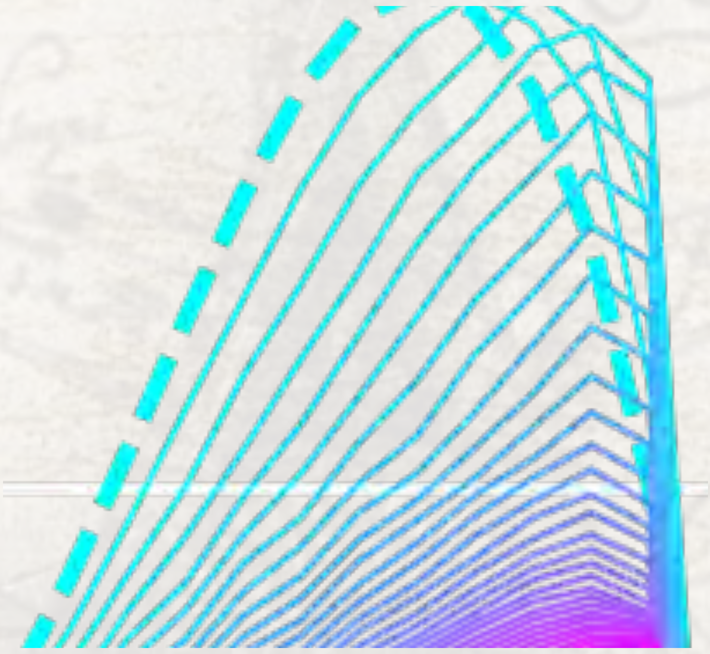


Physical Systems

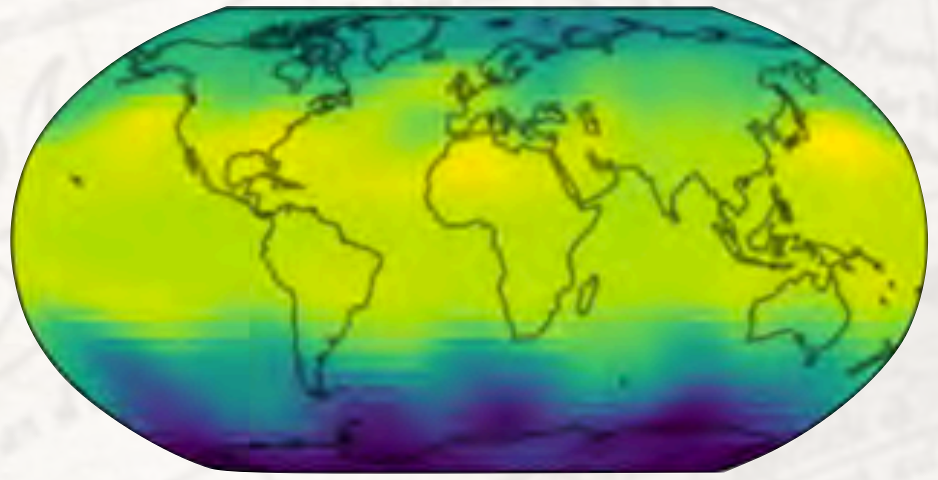




Correction



Control



Next: NWP?