ChainQueen: A Real-Time Differentiable Physical Simulator for Soft Robotics

Yuanming Hu, Jiancheng Liu*, Andrew Spielberg*, Joshua B. Tenenbaum, William T. Freeman, Jiajun Wu, Daniela Rus, Wojciech Matusik

Abstract—Physical simulators have been widely used in robot planning and control. Among them, differentiable simulators are particularly favored, as they can be incorporated into gradient-based optimization algorithms that are efficient in solving inverse problems such as optimal control and motion planning. Simulating deformable objects is, however, more challenging compared to rigid body dynamics. The underlying physical laws of deformable objects are more complex, and the resulting systems have orders of magnitude more degrees of freedom and therefore they are significantly more computationally expensive to simulate. Computing gradients with respect to physical design or controller parameters is typically even more computationally challenging. In this paper, we propose a real-time, differentiable hybrid Lagrangian-Eulerian physical simulator for deformable objects, ChainQueen, based on the Moving Least Squares Material Point Method (MLS-MPM). MLS-MPM can simulate deformable objects including contact and can be seamlessly incorporated into inference, control and co-design systems. We demonstrate that our simulator achieves high precision in both forward simulation and backward gradient computation. We have successfully employed it in a diverse set of control tasks for soft robots, including problems with nearly 3,000 decision variables.

I. INTRODUCTION

Robot planning and control algorithms often rely on physical simulators for prediction and optimization [1], [2]. In particular, differentiable physical simulators enable the use of gradient-based optimizers, significantly improving control efficiency and precision. Motivated by this, there has been extensive research on differentiable rigid body simulators, using approximate [3], [4] and exact [5], [6], [7] methods.

Significant challenges remain for deformable objects. First, simulating the motion of deformable objects is slow, because they have much higher degrees of freedom (DoFs). Second, contact detection and resolution is challenging for deformable objects, due to their changing geometries and potential self-collisions. Third, closed-form and efficient computation of gradients is challenging in the presence of contact. As a consequence, current simulation methods for soft objects cannot be effectively used for solving inverse problems such as optimal control and motion planning.

In this paper, we introduce a real-time, differentiable physical simulator for deformable objects, building upon the Moving Least Squares Material Point Method (MLS-MPM) [8]. We name our simulator ChainQueen. The Material Point Method (MPM) is a hybrid Lagrangian-Eulerian method that uses both particles and grid nodes for simulation [9]. MLS-MPM accelerates and simplifies traditional MPM using a moving least squares force discretization. In ChainQueen, we introduce the first fully differentiable MLS-MPM simulator with respect to both state and model parameters, with both forward simulation and back-propagation running efficiently on GPUs. We demonstrate the ability to efficiently calculate gradients with respect to the entire simulation. This enables many novel applications for soft robotics including optimization-based closed-loop controller design, trajectory optimization, and co-design of robot geometry, materials, and control.

As a particle-grid-based hybrid simulator, MPM simulates objects of various states, such as liquid (e.g., water), granular materials (e.g., sand), and elastoplastic materials (e.g., snow and human tissue). ChainQueen focuses on elastic materials for soft robotics. It is fully differentiable and 4–9× faster than the current state-of-the-art. Numerical and experimental validation suggest that ChainQueen achieves high precision in both forward simulation and backward gradient computation.

ChainQueen’s differentiability allows it to support gradient-based optimization for control and system identification. By performing gradient descent on controller parameters, our simulator is capable of solving these inverse problems on a diverse set of complex tasks, such as optimizing a 3D soft walker controller given an objective. Similarly, gradient descent on physical design parameters, enables inference of physical properties (e.g. mass, density and Young’s modulus) of objects and optimizing design for a desired task.

In addition to benchmarking ChainQueen’s performance and demonstrating its capabilities on a diverse set of inverse problems, we have interfaced our simulator with high-level python scripts to make ChainQueen user-friendly. Users at all levels will be able to develop their own soft robotics systems using our simulator, without the need to understand its low-level details. We will open-source our code and data and we hope they can benefit the robotics community.

II. RELATED WORK

A. Material Point Method

The material point method has been extensively developed from both a solid mechanics [9] and computer graphics [10] perspective. As a hybrid Eulerian-Lagrangian method, MPM

* Or 乾坤, literally "everything between the sky and the earth."
has demonstrated its versatility in simulating snow [11], sand [13], [14], non-Newtonian fluids [15], cloth [16], [17], solid-fluid coupling [18], [19], rigid body coupling, and cutting [8], [20] also proposed an adaptive MPM scheme to discretize continuum mechanics, which can be derived through the weak form of conservation laws. Such a physically-based approach makes it easier to match simulation with real-world experiments. Second, MPM is friendly to parallelization on modern hardware architectures. Closely related to our work is a high-performance GPU implementation [21] by Gao et al., which has demonstrated its versatility in simulating snow [11], solid-fluid coupling [18], [19], rigid body coupling, and cutting [8], [20].

There are many benefits of using MPM for soft robotics. First, MPM is a well-founded and physically-accurate discretization method and can be derived through the weak form of conservation laws. Such a physically-based approach makes it easier to match simulation with real-world experiments. Second, MPM is friendly to parallelization on modern hardware architectures. Closely related to our work is a high-performance GPU implementation [21] by Gao et al., which has demonstrated its versatility in simulating snow [11], solid-fluid coupling [18], [19], rigid body coupling, and cutting [8], [20].

We briefly cover the basics of MLS-MPM and readers are referred to [10] and [8] for a comprehensive introduction of MPM and MLS-MPM, respectively. The material point method is a hybrid Eulerian-Lagrangian method, where both particles and grid nodes are used. Simulation state information is transferred back-and-forth between these two representations. We summarize the notations we use in this paper in Table I. Subscripts are used to denote particle (p) and grid nodes (i), while superscripts (n, n + 1) are used to distinguish quantities in different time steps. The MLS-MPM simulation cycle has three steps:

1. **Particle-to-grid transfer (P2G)**. Particles transfer mass \(m_p\), momentum \([m\mathbf{v}_p]^T\), and stress-contributed impulse to their neighbouring grid nodes, using the Affine Particle-in-Cell method (APIC) [28] and moving least squares force discretization [8], weighted by a compact B-spline kernel \(N\):

   \[
   m_i^n = \sum_p N(x_i - x_p^n) m_p, \tag{3}
   \]

   \[
   \mathbf{G}_p^n = -\frac{4}{\Delta x^2} \mathbf{D} v_p^n \mathbf{P}_p^n \mathbf{F}_p^{nT} + m_p \mathbf{G}_p^n, \tag{4}
   \]

   \[
   \mathbf{p}_i^n = \sum_p N(x_i - x_p^n) \left[ m_p \mathbf{v}_p^n + \mathbf{G}_p^n (x_i - x_p^n) \right]. \tag{5}
   \]

2. **Grid operations.** Grid momentum is normalized into grid velocity after division by grid mass:

   \[
   \mathbf{v}_i^n = \frac{1}{m_i} \mathbf{p}_i^n. \tag{6}
   \]

   Note that neighbouring particles interact with each other through their shared grid nodes, and collisions are handled automatically. Here we omit boundary conditions and gravity for simplicity.

3. **Grid-to-particle transfer (G2P).** Particles gather updated velocity \(\mathbf{v}_p^{n+1}\), local velocity field gradients \(\mathbf{C}_p^{n+1}\) and position \(\mathbf{x}_p^{n+1}\). The constitutive model properties (e.g., deformation gradients \(\mathbf{F}_p^{n+1}\)) are updated:

   \[
   \mathbf{v}_p^{n+1} = \sum_i N(x_i - x_p^n) \mathbf{v}_i^n, \tag{7}
   \]

   \[
   \mathbf{C}_p^{n+1} = \frac{4}{\Delta x^2} \sum_i N(x_i - x_p^n) \mathbf{v}_i^n (x_i - x_p^n)^T, \tag{8}
   \]

   \[
   \mathbf{F}_p^{n+1} = (I + \Delta t \mathbf{C}_p^{n+1}) \mathbf{F}_p^n, \tag{9}
   \]

   \[
   \mathbf{x}_p^{n+1} = \mathbf{x}_p^n + \Delta t \mathbf{v}_p^{n+1}. \tag{10}
   \]

- **TABLE I: List of notations for MLS-MPM.**

| Symbol | Type | Affiliation | Meaning |
|--------|------|-------------|---------|
| \(\Delta t\) | scalar | time step size |
| \(\Delta x\) | scalar | grid cell size |
| \(x_p\) | vector | particle | position |
| \(v_p\) | scalar | particle | initial volume |
| \(v_p\) | vector | particle | velocity |
| \(\mathbf{C}_p\) | matrix | particle | affine velocity field |
| \(\mathbf{F}_p\) | matrix | particle | PK1 stress \((\partial \mathbf{\psi}_p / \partial \mathbf{F}_p)\) |
| \(\mathbf{\sigma}_{pa}\) | matrix | particle | actuation Cauchy stress |
| \(\mathbf{A}_p\) | matrix | particle | actuation stress (material space) |
| \(\mathbf{x}_i\) | vector | node | position |
| \(m_i\) | scalar | node | mass |
| \(v_i\) | vector | node | velocity |
| \(p_i\) | vector | node | momentum, i.e., \(m_i \mathbf{v}_i\) |
| \(N\) | scalar | | quadratic B-spline function |

We use the moving least squares material point method (MLS-MPM) [8] to discretize continuum mechanics, which is governed by the following two equations:

\[
\rho \frac{D\mathbf{v}}{Dt} = \nabla \cdot \mathbf{\sigma} + \rho \mathbf{g} \quad \text{(momentum conservation)}, \tag{1}
\]

\[
\frac{D\rho}{Dt} + \rho \nabla \cdot \mathbf{v} = 0 \quad \text{(mass conservation)}. \tag{2}
\]

We briefly cover the basics of MLS-MPM and readers are referred to [10] and [8] for a comprehensive introduction of MLS-MPM. The material point method is a hybrid Eulerian-Lagrangian method, where both particles and grid nodes are used. Simulation state information is transferred back-and-forth between these two representations. We summarize the notations we use in this paper in Table I. Subscripts are used to denote particle (p) and grid nodes (i), while superscripts (n, n + 1) are used to distinguish quantities in different time steps. The MLS-MPM simulation cycle has three steps:

1. **Particle-to-grid transfer (P2G)**. Particles transfer mass \(m_p\), momentum \([m\mathbf{v}_p]^T\), and stress-contributed impulse to their neighbouring grid nodes, using the Affine Particle-in-Cell method (APIC) [28] and moving least squares force discretization [8], weighted by a compact B-spline kernel \(N\):

   \[
   m_i^n = \sum_p N(x_i - x_p^n) m_p, \tag{3}
   \]

   \[
   \mathbf{G}_p^n = -\frac{4}{\Delta x^2} \mathbf{D} v_p^n \mathbf{P}_p^n \mathbf{F}_p^{nT} + m_p \mathbf{G}_p^n, \tag{4}
   \]

   \[
   \mathbf{p}_i^n = \sum_p N(x_i - x_p^n) \left[ m_p \mathbf{v}_p^n + \mathbf{G}_p^n (x_i - x_p^n) \right]. \tag{5}
   \]

2. **Grid operations.** Grid momentum is normalized into grid velocity after division by grid mass:

   \[
   \mathbf{v}_i^n = \frac{1}{m_i} \mathbf{p}_i^n. \tag{6}
   \]

   Note that neighbouring particles interact with each other through their shared grid nodes, and collisions are handled automatically. Here we omit boundary conditions and gravity for simplicity.

3. **Grid-to-particle transfer (G2P)**. Particles gather updated velocity \(\mathbf{v}_p^{n+1}\), local velocity field gradients \(\mathbf{C}_p^{n+1}\) and position \(\mathbf{x}_p^{n+1}\). The constitutive model properties (e.g., deformation gradients \(\mathbf{F}_p^{n+1}\)) are updated:

   \[
   \mathbf{v}_p^{n+1} = \sum_i N(x_i - x_p^n) \mathbf{v}_i^n, \tag{7}
   \]

   \[
   \mathbf{C}_p^{n+1} = \frac{4}{\Delta x^2} \sum_i N(x_i - x_p^n) \mathbf{v}_i^n (x_i - x_p^n)^T, \tag{8}
   \]

   \[
   \mathbf{F}_p^{n+1} = (I + \Delta t \mathbf{C}_p^{n+1}) \mathbf{F}_p^n, \tag{9}
   \]

   \[
   \mathbf{x}_p^{n+1} = \mathbf{x}_p^n + \Delta t \mathbf{v}_p^{n+1}. \tag{10}
   \]
For soft robotics, we additionally introduce an actuation model. Inspired by actuators such as [29], we designed
an actuation model that expands or stretches particle $p$ via an additional Cauchy stress $A_p = F_p \sigma_{po} F_p^T$, with
$\sigma_{po} = \text{Diag}(\alpha_x, \alpha_y, \alpha_z)$ – the stress in the material space. This framework supports the use of other differentiable
actuation models including pneumatic, hydraulic, and cable-driven actuators. Fig. 1 illustrates forward simulation and
back propagation.

ML-S-MPM is naturally differentiable. Though the forward direction has been extensively used in computer graphics, the backward direction (differentiation or back-propagation) is largely unexplored.

Based on the gradients we have derived analytically, we have designed a high-performance implementation that
resembles the traditional forward MPM cycle: backward P2G (scatter particle gradients to grid), grid operations, and
backward G2P (gather grid gradients to particles).\textsuperscript{‡} Gradients of state at the end of a time step with respect to states at the starting of the time step can be computed using the chain rule. With the single-step gradients computed, applying the chain rule at a higher level from the final state all-the-way to the initial state yields gradients of the final state with respect to the initial state, as well as to the controller parameters
that are used in each state. We cache all the simulation states in memory, using a “memo” object. Though the underlying
differentiation is complicated, we have designed a simple high-level TensorFlow interface on which end-users can build
their applications (Fig. 2).

Our high-performance implementation\textsuperscript{‡} takes advantage of the computational power of modern GPUs through CUDA. We also implemented a reference implementation in TensorFlow. Note that programming physical simulation as a “computation graph” using high-level frameworks such as TensorFlow is less inefficient. In fact, when all the overheads are gone, our optimized CUDA solver is $132 \times$ faster than the TensorFlow reference version. This is because TensorFlow is optimized towards deep learning applications where data granularity
is much larger and memory access pattern is much more regular than physical simulation, and limited CPU-GPU bandwidth.
In contrast, our CUDA implementation is tailored for MLS

\textsuperscript{†}

\textsuperscript{‡}

Please see the supplemental document for the gradient derivations.

Based the \textit{Taichi} [30] open source computer graphics library.

---

Fig. 1: One time step of MLS-MPM. Top arrows are for forward simulation and bottom ones are for back propagation. A controller is embedded in the P2G process to generate actuation given particle configurations.

TABLE II: Performance comparisons on a NVIDIA GTX 1080 Ti GPU. \textbf{F} stands for forward simulation and \textbf{B} stands for backward differentiation. \textbf{TF} indicates the TensorFlow implementation. When benchmarking our simulator with CUDA we use the C++ instead of python interface to avoid the extra overhead due to the TensorFlow runtime library.

| Approach   | Impl. | \# Particles | Time per Frame |
|------------|-------|--------------|----------------|
| Flex (3D)  | CUDA  | 8,024        | 3.5 ms (286 FPS) |
| Ours (3D, F) | CUDA  | 8,000        | 0.392 ms (2,551 FPS) |
| Ours (3D, B) | CUDA  | 8,000        | 0.406 ms (2,463 FPS) |
| Flex (3D)  | CUDA  | 61,238       | 6 ms (167 FPS)    |
| Ours (3D, F) | CUDA  | 64,000       | 1.594 ms (628 FPS) |
| Ours (3D, B) | CUDA  | 64,000       | 1.774 ms (563 FPS) |
| Ours (3D, F) | CUDA  | 512,000      | 10.501 ms (92 FPS)  |
| Ours (3D, B) | CUDA  | 512,000      | 11.594 ms (86 FPS)  |
| Ours (2D, F) | TF    | 6,400        | 13.2 ms (76 FPS)   |
| Ours (2D, B) | TF    | 6,400        | 35.7 ms (28 FPS)   |
| Ours (2D, F) | CUDA  | 6,400        | 0.10 ms (10,000 FPS) |
| Ours (2D, B) | CUDA  | 6,400        | 0.14 ms (7,162 FPS) |

MPM and explicitly optimized for parallelism and locality, thus delivering high-performance.

IV. EVALUATION

In this section, we conduct a comprehensive study of the efficiency and accuracy of our system, in both 2D and 3D.

A. Efficiency

Instead of using complex geometries, a simple falling cube is used for performance benchmarking, to ensure easy analysis and reproducibility. We benchmark the performance of our CUDA simulator against NVIDIA Flex [31], a popular PBD physical simulator capable of simulating deformable objects. Note that both PBD and MLS-MPM needs substepping iterations to ensure high stiffness. To ensure fair comparison, we set a Young’s modulus, Poisson’s ration and density so that visually ChainQueen gives similar results to Flex. We used two steps per frame and four iterations per step in Flex.

Note that setting exactly the same parameters is not possible since in PBD there is no explicitly defined physical quantity such as Young’s modulus.

We summarize the quantitative performance in Table II. Our CUDA simulator provides higher speed than Flex, when the number of particles are the same. It is also worth noting
that the TensorFlow implementation is much slower, due to excessive runtime overheads.

B. Accuracy

We design five test cases to evaluate the accuracy of both forward simulation and backward gradient evaluation:

1) A1 (analytic, 3D, float32 precision): final position w.r.t. initial velocity (with collision). This case tests conservation of momentum, gradient accuracy and stability of back-propagation.

2) A2 (analytic, 3D, float32 precision): same as A1 but with one collision to a friction-less wall.

3) B (numeric, 2D, float64 precision): colliding billiards. This case tests gradient accuracy and stability in more complex cases where analytic solutions do not exist. We used float64 precision for accurate finite difference results.

4) C (numeric, 2D, float64 precision): finger controller. This case tests gradient accuracy of controller parameters, which are used repeatedly in the whole simulation process.

5) D1 (experimental, pneumatic actuator, actuation) In order to evaluate our simulator’s real-world accuracy, we compared the deformation of a physical actuator to a virtual one. The physical actuator has four pneumatic chambers which can be inflated with an external pump, arranged in a cross-shape. Inflating the individual chambers bends the actuator away from that chamber. The actuator was cast using Smooth-On Dragon Skin 30.

6) D2 (experimental, pneumatic actuator, bouncing) In a second test, we dropped the same actuator from a 15 cm height, and compared its dynamic motion to a simulation.

In 3D analytic test cases, where gradients w.r.t. initial parameters can be directly evaluated as in Table III. For the experimental comparisons, the results are shown in Fig. 3. In addition to our simulator’s high performance and accuracy, it is worth noting that the gradients remain stable in the long term, within up to 1000 time steps.

V. INFERENCE, CONTROL AND CO-DESIGN

The most attractive feature of our simulator is the existence of quickly computable gradients, which allows the use of much more efficient gradient-based optimization algorithms. In this section, we show the effectiveness of our differentiable simulator on gradient-based optimization tasks, including physical inference, control for soft robotics, and co-design of robotic arms.
Fig. 3: Experiments on the pneumatic leg. **Row (A, B)** Footage and simulator results of a bouncing experiment with the leg dropping at 15 cm. **Row (C, D)** Actuation test.

### A. Physical Parameter Inference

ChainQueen can be used to infer physical system properties given its observed motion, e.g. perform gradient descent to infer the relative densities of two colliding elastic balls (see figure above, ball A moving to the right hitting ball B, and ball B arrives the destination C). Gradient-based optimization infers that relative density of ball A is $2.26$, which constitutes to the correct momentum to push B to C. Such capability makes it useful for real-world robotic tasks such as system identification.

### B. Control

We can optimize regression-based controllers for soft robots and efficiently discover stable gaits. The controller takes as input the state vector $z$, which includes target position, the center of mass position, and velocity of each composed soft component. In our examples, the actuation vector $a$ for up to 16 actuators is generated by the controller in each time step. During optimization, we perform gradient descent on variables $W$ and $b$, where $a = \tanh(Wz + b)$ is the actuation-generating controller.

We have designed a series of experiments including the 2D biped runner (Fig. 4) and robotic finger, and 3D quadrupedal runner (Fig. 6), crawler and robotic arm. Gradient-based optimizers successfully compute desired controllers within only tens or hundreds of iterations. Visual results are included in the supplemental video.

To emphasize the merits of gradient-based approaches, we compare our control method with proximal policy optimization (PPO) [32], a state-of-the-art reinforcement learning algorithm. PPO is an actor-critic method which relies on sampled gradients of a reward function in order to optimize a policy. This sampling-based approach is model-free; it relies on gradients of the rewards with respect to controller parameters, but not with respect to the physical model for updates. For our comparison, we use velocity projected onto the direction toward the goal as the reward. We use a simplified single link version (with only two adjacent actuators) of Fig. 5 and the 2D runner Fig. 4 as a benchmark. Quantitative results for the finger are shown in Fig. 7. We performed a similar comparison on the 2D walker, the controller optimized by ChainQueen for the 2D walker starts functioning well within 20 minutes; by comparison the policy recovered by PPO still chose nearly-random actions after over 4 hours of training; demonstrating that for certain soft locomotion tasks our gradient-based method can be more efficient than model-free approaches.

### C. Co-design

Our simulator is capable of not only providing gradients with respect to dynamics and controller parameters, but also with respect to structural design parameters, enabling co-design of soft robots. To demonstrate this, we designed a multi-link robot arm (two links, two joints each with two side-by-side actuators; all parts deformable). Similar to shooting method trajectory optimization, actuation for each time step is solved for, along with the time-invariant Young's modulus of the system for each particle. In our task, we optimized the end-effector of the arm to reach a goal ball with final 0 arm velocity, and minimized for actuation cost $\sum_{i=0}^{N} u_i^T u_idt$, where $u_i$ is the actuation vector at timestep $i$, and $N$ is the total number of timesteps. This is a dynamic task and

\[ \text{§Note that this is functionally extremely similar to a distance loss; the cumulative reward } \int_{0}^{T} v_{\text{goal}} dt = D - \| x_T - x_{\text{goal}} \|, \text{ where } D \text{ is the initial distance and } x_T \text{ and } x_{\text{goal}} \text{ represent world coordinates of the robot at time } T \text{ and of the goal, respectively. As velocity toward the goal increases, final distance to the goal decreases.} \]
Fig. 5: Final poses of the arm swing task. Lighter colors refer to stiffer regions. (c) Final pose of the fixed-stiffness 300% initial Young’s modulus arm. (d) Final pose of the fixed-stiffness 300% initial Young’s modulus arm. (e) Final pose of the co-optimized arm. Actuation cost is 95.5% that of the fixed 100% initial Young’s modulus arm and converges. Only the co-optimized arm is able to fully reach its target. The final optimized spatially varying stiffness of the arm has lower stiffness on the outside of the bend, and higher stiffness inside, promoting more bend to the left. Qualitatively, this is similar in effect to the pleating on soft robot fingers.

Fig. 6: A 3D quadrupedal runner. Please see the supplemental video for more results.

Fig. 7: Gradient-free optimization using PPO and gradient-descent based on ChainQueen, on the 2D finger task. Thanks to the accurate gradient information, even the most vanilla optimizer can beat state-of-the-art reinforcement learning algorithms by one order of magnitude regarding optimization speed. (Left) single, fixed target. (Middle) random targets. (Right) random targets, larger range. Curves are smoothed over 10, 100 and 100 iterations respectively. The x-axis is simulation iterations and y-axis the loss.

VI. DISCUSSION

We have presented ChainQueen, a differentiable simulator for soft robotics, and demonstrated how it can be deployed for inference, control, and co-design. ChainQueen has the potential to accelerate the development of soft robots. We have also developed a high-performance GPU implementation for ChainQueen, which we plan to open source.

ACKNOWLEDGMENTS

We would like to thank Chenfanfu Jiang, Ming Gao and Kui Wu for the insightful discussions.
Supplemental Document

In this document, we discuss the detailed steps for backward gradient computation in ChainQueen, i.e. the differentiable Moving Least Squares Material Point Method (MLS-MPM) [8]. Again, we summarize the notations in Table IV We assume fixed particle mass \( m_p \), volume \( V_p^0 \), hyperelastic constitutive model (with potential energy \( \psi_p \) or Young’s modulus \( E_p \) and Poisson’s ratio \( \nu_p \)) for simplicity.

| Symbol | Type       | Affiliation | Meaning                        |
|--------|------------|-------------|--------------------------------|
| \( \Delta t \) | scalar     | time step size |                                |
| \( \Delta x \) | scalar     | grid cell size |                                |
| \( x_p \) | vector     | particle     | position                        |
| \( V_p^0 \) | scalar     | particle     | initial volume                  |
| \( v_p \) | vector     | particle     | velocity                        |
| \( C_p \) | matrix     | particle     | affine velocity field [28]      |
| \( P_p \) | matrix     | particle     | PK1 stress \( (\partial \psi_p / \partial F_p) \) |
| \( \sigma_{pa} \) | matrix     | particle     | actuation Cauchy stress         |
| \( A_p \) | matrix     | particle     | actuation stress (material space) |
| \( F_p \) | matrix     | particle     | deformation gradient            |
| \( x_i \) | vector     | node         | position                        |
| \( m_i \) | scalar     | node         | mass                            |
| \( v_i \) | vector     | node         | velocity                        |
| \( p_i \) | vector     | node         | momentum, i.e. \( m_i v_i \)   |
| \( N \) | scalar     | node         | quadratic B-spline function     |

VII. VARIABLE DEPENDENCIES

The MLS-MPM time stepping is defined as follows:

\[
P^n_p = P^n_p (F^n_p) + F_p \sigma^{n}_{pa} \tag{11}
\]

\[
m^n_i = \sum_p N(x_i - x^n_p)m_p \tag{12}
\]

\[
p^n_i = \sum_p N(x_i - x^n_p) \left[ m_p v^n_p + \left( -\frac{4}{\Delta x^2} \Delta t V_p^n P^n_p F^n_p + m_p C^n_p \right) (x_i - x^n_p) \right] \tag{13}
\]

\[
v^n_i = \frac{1}{m^n_i} p^n_i \tag{14}
\]

\[
v^{n+1}_i = \sum_i N(x_i - x^n_p) v^n_i \tag{15}
\]

\[
C^{n+1}_p = \frac{4}{\Delta x^2} \sum_i N(x_i - x^n_p) v^n_i (x_i - x^n_p)^T \tag{16}
\]

\[
F^{n+1}_p = (I + \Delta t C^{n+1}_p) F^n_p, \tag{17}
\]

\[
x^{n+1}_p = x^n_p + \Delta t v^{n+1}_p \tag{18}
\]
The forward variable dependency is as follows:

\[ x_{p}^{n+1} \leftarrow x_{p}^{n}, v_{p}^{n+1} \quad (20) \]
\[ v_{p}^{n+1} \leftarrow x_{p}^{n}, v_{i}^{n} \quad (21) \]
\[ C_{p}^{n+1} \leftarrow x_{p}^{n}, v_{i}^{n} \quad (22) \]
\[ F_{p}^{n+1} \leftarrow F_{p}^{n}, C_{p}^{n+1} \quad (23) \]
\[ p_{i}^{n} \leftarrow x_{p}^{n}, C_{p}^{n}, v_{p}^{n}, P_{p}^{n}, F_{p}^{n} \quad (24) \]
\[ v_{i}^{n} \leftarrow p_{i}^{n}, m_{i}^{n} \quad (25) \]
\[ P_{p}^{n} \leftarrow F_{p}^{n}, \sigma_{pa}^{n} \quad (26) \]
\[ m_{i}^{n} \leftarrow x_{p}^{n} \quad (27) \]
\[ (28) \]

During back-propagation, we have the following reversed variable dependency:

\[ x_{p}^{n+1}, v_{p}^{n+1}, C_{p}^{n+1}, p_{i}^{n+1}, m_{i} \leftarrow x_{p}^{n} \quad (29) \]
\[ p_{i}^{n} \leftarrow v_{p}^{n} \quad (30) \]
\[ x_{p}^{n+1} \leftarrow v_{p}^{n+1} \quad (31) \]
\[ v_{p}^{n+1}, C_{p}^{n+1} \leftarrow v_{i}^{n} \quad (32) \]
\[ F_{p}^{n+1}, P_{p}^{n}, P_{i}^{n} \leftarrow F_{p}^{n} \quad (33) \]
\[ F_{p}^{n+1} \leftarrow C_{p}^{n+1} \quad (34) \]
\[ p_{i}^{n} \leftarrow C_{p}^{n} \quad (35) \]
\[ v_{i}^{n} \leftarrow p_{i}^{n} \quad (36) \]
\[ v_{i}^{n} \leftarrow m_{i}^{n} \quad (37) \]
\[ P_{i}^{n} \leftarrow P_{p}^{n} \quad (38) \]
\[ P_{p}^{n} \leftarrow \sigma_{pa}^{n} \quad (39) \]
\[ (40) \]

We reverse swap two sides of the equations for easier differentiation derivation:

\[ x_{p}^{n} \rightarrow x_{p}^{n+1}, v_{p}^{n+1}, C_{p}^{n+1}, p_{i}^{n+1}, m_{i} \quad (41) \]
\[ v_{p}^{n} \rightarrow P_{p}^{n} \quad (42) \]
\[ v_{p}^{n+1} \rightarrow x_{p}^{n+1} \quad (43) \]
\[ v_{i}^{n} \rightarrow v_{p}^{n+1}, C_{p}^{n+1} \quad (44) \]
\[ F_{p}^{n} \rightarrow F_{p}^{n+1}, P_{p}^{n}, P_{i}^{n} \quad (45) \]
\[ C_{p}^{n+1} \rightarrow F_{p}^{n+1} \quad (46) \]
\[ C_{p}^{n} \rightarrow P_{i}^{n} \quad (47) \]
\[ p_{i}^{n} \rightarrow v_{p}^{n} \quad (48) \]
\[ m_{i}^{n} \rightarrow v_{p}^{n} \quad (49) \]
\[ P_{p}^{n} \rightarrow P_{i}^{n} \quad (50) \]
\[ \sigma_{pa}^{n} \rightarrow P_{p}^{n} \quad (51) \]
\[ (52) \]

In the following sections, we derive detailed gradient relationships, in the order of actual gradient computation. The frictional boundary condition gradients are postponed to the end since it is less central, though during computation it belongs to grid operations. Back-propagation in ChainQueen is essentially a reversed process of forward simulation. The computation has three steps, backward particle to grid (P2G), backward grid operations, and backward grid to particle (G2P).
VIII. BACKWARD PARTICLE TO GRID (P2G)

(A, P2G) For \(v_{p}^{n+1}\), we have

\[
x_{p}^{n+1} = x_{p}^{n} + \Delta t v_{p}^{n+1}
\]

\[
\implies \frac{\partial L}{\partial v_{pa}^{n+1}} = \frac{\partial L}{\partial x_{p}^{n+1}} \frac{\partial x_{p}^{n+1}}{\partial v_{pa}^{n+1}}
\]

\[
= \Delta t \frac{\partial L}{\partial x_{p}^{n+1}}.
\]

(B, P2G) For \(C_{p}^{n+1}\), we have

\[
F_{p}^{n+1} = (I + \Delta t C_{p}^{n+1}) F_{p}^{n}
\]

\[
\implies \frac{\partial L}{\partial C_{pa}^{n+1}} = \frac{\partial L}{\partial F_{pa}^{n+1}} \frac{\partial F_{pa}^{n+1}}{\partial C_{pa}^{n+1}}
\]

\[
= \Delta t \sum_{\gamma} \frac{\partial L}{\partial F_{pa}^{n+1}} F_{p\gamma}^{n}.
\]

Note, the above two gradients should also include the contributions of \(\frac{\partial L}{\partial v_{p}^{n}}\) and \(\frac{\partial L}{\partial C_{p}^{n}}\) respectively, with \(n\) being the next time step.

(C, P2G) For \(v_{i}^{n}\), we have

\[
v_{i}^{n+1} = \sum_{i} N(x_{i} - x_{p}^{n}) v_{i}^{n}
\]

\[
C_{p}^{n+1} = \frac{4}{\Delta x^{2}} \sum_{i} N(x_{i} - x_{p}^{n}) v_{i}^{n} (x_{i} - x_{p}^{n})^{T}
\]

\[
\implies \frac{\partial L}{\partial v_{i\alpha}^{n}} = \left[ \sum_{p} \frac{\partial L}{\partial v_{p\alpha}^{n}} \frac{\partial v_{p\alpha}^{n}}{\partial v_{i\alpha}^{n}} + \sum_{p} \frac{\partial L}{\partial C_{p\alpha}^{n+1}} \frac{\partial C_{p\alpha}^{n+1}}{\partial v_{i\alpha}^{n}} \right]_{\alpha}
\]

\[
= \sum_{p} \frac{\partial L}{\partial v_{p\alpha}^{n+1}} N(x_{i} - x_{p}^{n}) + \frac{4}{\Delta x^{2}} N(x_{i} - x_{p}^{n}) \sum_{\beta} \frac{\partial L}{\partial C_{p\beta}^{n+1}} (x_{i\beta} - x_{p\beta})
\]

IX. BACKWARD GRID OPERATIONS

(D, grid) For \(p_{i}^{n}\), we have

\[
v_{i}^{n} = \frac{1}{m_{i}^{n}} p_{i}^{n}
\]

\[
\implies \frac{\partial L}{\partial p_{i\alpha}^{n}} = \frac{\partial L}{\partial v_{i\alpha}^{n}} \frac{\partial v_{i\alpha}^{n}}{\partial p_{i\alpha}^{n}}
\]

\[
= \frac{\partial L}{\partial v_{i\alpha}^{n}} \frac{1}{m_{i}^{n}}.
\]

(E, grid) For \(m_{i}^{n}\), we have

\[
v_{i}^{n} = \frac{1}{m_{i}^{n}} p_{i}^{n}
\]

\[
\implies \frac{\partial L}{\partial m_{i\alpha}^{n}} = \frac{\partial L}{\partial v_{i\alpha}^{n}} \frac{\partial v_{i\alpha}^{n}}{\partial m_{i\alpha}^{n}}
\]

\[
= \frac{1}{(m_{i}^{n})^{2}} \sum_{\alpha} p_{i\alpha}^{n} \frac{\partial L}{\partial v_{i\alpha}^{n}}
\]

\[
= \frac{1}{m_{i}^{n}} \sum_{\alpha} v_{i\alpha}^{n} \frac{\partial L}{\partial v_{i\alpha}^{n}}.
\]
(F, G2P) For $v^n_p$, we have
\[
p^n_i = \sum_p N(x_i - x^n_p) \left[ m_p v^n_p + \left( -\frac{4}{\Delta x^2} \Delta t V^o_p P^n_p F^n_{p} + m_p C^n_p \right) (x_i - x^n_p) \right]
\]  
(70)

\[ \implies \frac{\partial L}{\partial v^n_{p\alpha}} = \left[ \sum_i \frac{\partial L}{\partial p^n_{\alpha}} \frac{\partial p^n_{\alpha}}{\partial v^n_p} \right] \alpha \]
(71)

\[ = \sum_i N(x_i - x^n_p) m_p \frac{\partial L}{\partial p^n_{\alpha}}. \]  
(72)

(G, G2P) For $P^n_p$, we have
\[
p^n_i = \sum_p N(x_i - x^n_p) \left[ m_p v^n_p + \left( -\frac{4}{\Delta x^2} \Delta t V^o_p P^n_p F^n_{p} + m_p C^n_p \right) (x_i - x^n_p) \right]
\]  
(73)

\[ \implies \frac{\partial L}{\partial P^n_{p\alpha\beta}} = \left[ \frac{\partial L}{\partial p^n_{\alpha}} \frac{\partial p^n_{\alpha}}{\partial P^n_p} \frac{\partial P^n_p}{\partial P^n_{p\alpha\beta}} \right] \alpha \beta \]
(74)

\[ = -\sum_i N(x_i - x^n_p) \frac{4}{\Delta x^2} \Delta t V^o_p \sum_\gamma \frac{\partial L}{\partial p^n_{p\alpha}} P^n_{p\gamma\beta} (x_i - x^n_p). \]  
(75)

(H, G2P) For $F^n_p$, we have
\[
F^{n+1}_p = (I + \Delta t C^{n+1}_p) F^n_p
\]  
(76)

\[ P^n_p = P^n_p (F^n_p) + F_p \sigma^{n\alpha}_{p\alpha} \]
(77)

\[ p^n_i = \sum_p N(x_i - x^n_p) \left[ m_p v^n_p + \left( -\frac{4}{\Delta x^2} \Delta t V^o_p P^n_p F^n_{p} + m_p C^n_p \right) (x_i - x^n_p) \right]
\]  
(78)

\[ \implies \frac{\partial L}{\partial F^n_{p\alpha\beta}} = \left[ \frac{\partial L}{\partial F^{n+1}_p} \frac{\partial F^{n+1}_p}{\partial F^n_p} + \frac{\partial L}{\partial P^n_p} \frac{\partial P^n_p}{\partial F^n_p} + \frac{\partial L}{\partial P^n_{p\alpha}} \frac{\partial P^n_{p\alpha}}{\partial F^n_p} \right] \alpha \beta \]
(79)

\[ = \sum_\gamma \frac{\partial L}{\partial F^{n+1}_{p\gamma\beta}} (I_{\gamma\alpha} + \Delta t C^{n+1}_{p\gamma\alpha}) + \sum_\gamma \sum_\eta \frac{\partial L}{\partial F^n_{p\eta\gamma}} \frac{\partial^2 \Psi^n_p}{\partial F^n_{p\eta\gamma}} + \sum_\gamma \frac{\partial L}{\partial \sigma^n_{p\alpha\beta\gamma}} \]
(80)

\[ + \sum_i -N(x_i - x^n_p) \sum_\gamma \frac{\partial L}{\partial p^n_{p\gamma}} \frac{4}{\Delta x^2} \Delta t V^o_p P^n_{p\gamma\beta} (x_i - x^n_p). \]  
(81)

(I, G2P) For $C^n_p$, we have
\[
p^n_i = \sum_p N(x_i - x^n_p) \left[ m_p v^n_p + \left( -\frac{4}{\Delta x^2} \Delta t V^o_p P^n_p F^n_{p} + m_p C^n_p \right) (x_i - x^n_p) \right]
\]  
(82)

\[ \implies \frac{\partial L}{\partial C^n_{p\alpha\beta}} = \left[ \sum_i \frac{\partial L}{\partial p^n_{p\alpha}} \frac{\partial p^n_{p\alpha}}{\partial C^n_p} \right] \alpha \beta \]
(83)

\[ = \sum_i N(x_i - x^n_p) \frac{\partial L}{\partial p^n_{p\alpha}} m_p (x_i - x^n_p). \]  
(84)
(J, G2P) For \( x_p^n \), we have
\[
x_p^{n+1} = x_p^n + \Delta t x_p^n
\]
\[
v_p^{n+1} = \sum_i N(x_i - x_p^n) v_i^n
\]
\[
c_p^{n+1} = \frac{4}{\Delta x^2} \sum_i N(x_i - x_p^n) v_i^n (x_i - x_p^n)^T
\]
\[
p_i^n = \sum_p N(x_i - x_p^n) \left[ m_p v_i^n + \left( -\frac{4}{\Delta x^2} \Delta t V_f^n p_p F_p^n + m_p C_p^n \right) (x_i - x_p^n) \right]
\]
\[
m_i^n = \sum_p N(x_i - x_p^n) m_p
\]
\[
G_p := \left( -\frac{4}{\Delta x^2} V_f^n \Delta t P_p^n F_p^n + m_p C_p^n \right)
\]
\[
\Rightarrow \frac{\partial L}{\partial x_p^{n+1}} = \left[ \frac{\partial L}{\partial x_p^n} \frac{\partial x_p^{n+1}}{\partial x_p^n} + \frac{\partial L}{\partial v_p^n} \frac{\partial v_p^{n+1}}{\partial v_p^n} + \frac{\partial L}{\partial c_p^n} \frac{\partial c_p^{n+1}}{\partial c_p^n} + \frac{\partial L}{\partial p_i^n} \frac{\partial p_i^n}{\partial x_p^n} + \frac{\partial L}{\partial m_i^n} \frac{\partial m_i^n}{\partial x_p^n} \right]_\alpha
\]
\[
= \frac{\partial L}{\partial x_p^{n+1}}
\]
\[
+ \sum_i \sum_\beta \frac{\partial L}{\partial v_p^{n+1}} \frac{\partial N(x_i - x_p^n)}{\partial x_i} v_i^{n+1}_\beta
\]
\[
+ \sum_i \sum_\beta \frac{4}{\Delta x^2} \left\{ -\frac{\partial L}{\partial c_p^{n+1}} \frac{\partial N(x_i - x_p^n)}{\partial x_i} + \sum_\gamma \frac{\partial L}{\partial c_p^{n+1}} \frac{\partial N(x_i - x_p^n)}{\partial x_i} v_i^{n+1}_\beta (x_i - x_p^n) \right\}
\]
\[
+ \sum_i \sum_\beta \frac{\partial L}{\partial p_i^{n+1}} \left[ \frac{\partial N(x_i - x_p^n)}{\partial x_i} (m_p v_i^n + [G_p(x_i - x_p^n)]_\beta) - N(x_i - x_p^n) G_p^{n+1} \right]
\]
\[
+ m_p \sum_i \frac{\partial L}{\partial m_i^n} \frac{\partial N(x_i - x_p^n)}{\partial x_i}
\]

(K, G2P) For \( \sigma_{pa}^n \), we have
\[
P_p^n = P_p^n (F_p^n) + F_p \sigma_{pa}^n
\]
\[
\Rightarrow \frac{\partial L}{\partial \sigma_{pa}^n} = \left[ \frac{\partial L}{\partial P_p^n} \frac{\partial P_p^n}{\partial \sigma_{pa}^n} \right]_{\alpha \beta}
\]
\[
= \sum_\gamma \frac{\partial L}{\partial P_p^{n+1}} F_p^{n+1} \gamma \alpha.
\]
When there are boundary conditions:

(L, grid) For $v_i^n$, we have

\[ l_{in} = \sum_\alpha v_{i\alpha} n_{i\alpha} \]  \hspace{1cm} (102)

\[ v_{it} = v_i - l_{in} n_i \]  \hspace{1cm} (103)

\[ l_{it} = \sqrt{\sum_\alpha v_{i\alpha}^2} + \varepsilon \]  \hspace{1cm} (104)

\[ \mathbf{v}_{it} = \frac{1}{l_{it}} v_{it} \]  \hspace{1cm} (105)

\[ l_{it}^* = \max\{l_{it} + c_i \min\{l_{in}, 0\}, 0\} \]  \hspace{1cm} (106)

\[ v_{i*} = l_{it}^* \mathbf{v}_{it} + \max\{l_{in}, 0\} n_i \]  \hspace{1cm} (107)

\[ H(x) := [x \geq 0] \]  \hspace{1cm} (108)

\[ R := l_{it} + c_i \min\{l_{in}, 0\} \]  \hspace{1cm} (109)

\[ \frac{\partial L}{\partial l_{it}^*} = \sum_\alpha \frac{\partial L}{\partial v_{i*\alpha}} \mathbf{v}_{i\alpha} \]  \hspace{1cm} (110)

\[ \frac{\partial L}{\partial l_{it}} = \frac{\partial L}{\partial v_{i\alpha}} l_{it}^* \]  \hspace{1cm} (111)

\[ \frac{\partial L}{\partial v_{ito}} = \frac{\partial L}{\partial v_{ito}} \frac{l_{it}^*}{l_{it}} - \frac{1}{l_{it}^*} \sum_\alpha v_{ito} \frac{\partial L}{\partial v_{ito}} + \frac{\partial L}{\partial l_{it}^*} H(R) \]  \hspace{1cm} (112)

\[ \frac{\partial L}{\partial v_{i\alpha}} = v_{i\alpha} + \frac{\partial L}{\partial l_{in}} n_{i\alpha} + \frac{\partial L}{\partial v_{i\alpha}} \]  \hspace{1cm} (113)

\[ \frac{\partial L}{\partial l_{in}} = -\sum_\alpha \frac{\partial L}{\partial v_{i\alpha}} n_{i\alpha} + \frac{\partial L}{\partial l_{it}^*} H(R) c_i H(-l_{in}) + \sum_\alpha H(l_{in}) n_{i\alpha} \frac{\partial L}{\partial v_{i\alpha}} \]  \hspace{1cm} (114)

\[ \frac{\partial L}{\partial v_{i\alpha}} = \frac{\partial L}{\partial l_{in}} n_{i\alpha} + \frac{\partial L}{\partial v_{i\alpha}} \]  \hspace{1cm} (115)
References

[1] E. Todorov, T. Erez, and Y. Tassa, “Mujoce: A physics engine for model-based control,” in IROS. IEEE, 2012, pp. 5026–5033.

[2] T. Erez, Y. Tassa, and E. Todorov, “Simulation tools for model-based robotics: Comparison of bullet, havok, mujoco, ode and physx,” in ICRA. IEEE, 2015, pp. 4397–4404.

[3] M. B. Chang, T. Ullman, A. Torralba, and J. B. Tenenbaum, “A compositional object-based approach to learning physical dynamics,” ICLR 2016, 2016.

[4] D. Mrowca, C. Zhuang, E. Wang, N. Haber, L. Fei-Fei, J. B. Tenenbaum, and D. L. Yamins, “Flexible neural representation for physics prediction,” arXiv:1806.08047, 2018.

[5] J. Degraeve, M. Hermans, J. Dambre et al., “A differentiable physics engine for deep learning in robotics,” arXiv preprint arXiv:1611.01652, 2016.

[6] R. Tedrake and the Drake Development Team, “Drake: A planning, control, and analysis toolbox for nonlinear dynamical systems,” 2016. [Online]. Available: https://drake.mit.edu

[7] M. Frigerio, J. Buchli, D. G. Caldwell, and C. Semini, “RobCoGen: a code generator for efficient kinematics and dynamics of articulated robots, based on Domain Specific Languages,” vol. 7, no. 1, pp. 36–54, 2016.

[8] Y. Hu, Y. Fang, Z. Ge, Z. Qu, Y. Zhu, A. Pradhana, and C. Jiang, “A moving least squares material point method with displacement discontinuity and two-way rigid body coupling,” ACM TOG, vol. 37, no. 4, p. 150, 2018.

[9] D. Sulsky, S.-J. Zhou, and H. L. Schreyer, “Application of a particle-in-cell method to solid mechanics,” Computer physics communications, vol. 87, no. 1-2, pp. 236–252, 1995.

[10] C. Jiang, C. Schroeder, J. Teran, A. Stomakhin, and A. Selle, “The material point method for simulating continuum materials,” in ACM SIGGRAPH 2016 Courses. ACM, 2016, p. 24.

[11] A. Stomakhin, C. Schroeder, L. Chai, J. Teran, and A. Selle, “A material point method for snow simulation,” ACM Transactions on Graphics (TOG), vol. 32, no. 4, p. 102, 2013.

[12] J. Gaume, T. Gast, J. Teran, A. van Herwijnen, and C. Jiang, “Dynamic anticrack propagation in snow,” Nature communications, vol. 9, no. 1, p. 3047, 2018.

[13] G. Klár, T. Gast, A. Pradhana, C. Fu, C. Schroeder, C. Jiang, and J. Teran, “Drucker-prager elastoplasticity for sand animation,” ACM Transactions on Graphics (TOG), vol. 35, no. 4, p. 103, 2016.

[14] M. Toussaint, K. R. Allen, K. A. Smith, and J. B. Tenenbaum, “Differentiable physics and stable modes for tool-use and manipulation planning,” in ICML, 2018.

[15] A. Sanchez-Gonzalez, N. Heess, J. T. Springenberg, J. Merel, M. Riedmiller, R. Hadsell, and P. Battaglia, “Graph networks as learnable physics engines for inference and control,” in ICML, 2018.

[16] M. Gao, A. Pradhana, X. Han, Q. Guo, G. Kot, E. Sifakis, and C. Jiang, “Animating fluid sediment mixture in particle-laden flows,” ACM Trans. Graph., vol. 37, no. 4, pp. 149:1–149:11, Jul. 2018.

[17] M. Gao, A. P. Tampubolon, T. Gast, G. Klár, C. Fu, J. Teran, C. Jiang, and K. Museth, “Multi-species simulation of porous sand and water mixtures,” ACM Trans. Graph., vol. 36, no. 4, pp. 105:1–105:11, Jul. 2017.

[18] M. Gao, A. P. Tampubolon, C. Jiang, and E. Sifakis, “An adaptive generalized interpolation material point method for simulating elastoplastic materials,” ACM Trans. Graph., vol. 36, no. 6, pp. 223:1–223:12, Nov. 2017.

[19] M. Gao, Wang, K. Wu, A. Pradhana-Tampubolon, E. Sifakis, Y. Cem, and C. Jiang, “Gpu optimization of material point methods,” ACM Transactions on Graphics (TOG), vol. 32, no. 4, p. 102, 2013.

[20] P. W. Battaglia, R. Pascanu, M. Lai, D. Rezende, and K. Kavukcuoglu, “Interaction networks for learning about objects, relations and physics,” in NIPS, 2016.

[21] A. Sanchez-Gonzalez, N. Heess, J. T. Springenberg, J. Merel, M. Riedmiller, R. Hadsell, and P. Battaglia, “Graph networks as learnable physics engines for inference and control,” in ICML, 2018.

[22] F. de Avila Belbute-Peres, K. A. Smith, K. Allen, J. B. Tenenbaum, and J. Z. Kolter, “End-to-end differentiable physics for learning and control,” in Neural Information Processing Systems, 2018.

[23] M. Toussaint, K. R. Allen, K. A. Smith, and J. B. Tenenbaum, “Differentiable physics and stable modes for tool-use and manipulation planning,” in RSS, 2018.

[24] C. Schenk and D. Fox, “Spnets: Differentiable fluid dynamics for deep neural networks,” arXiv:1806.06094, 2018.

[25] M. Macklin and M. Müller, “Position based fluids,” ACM Transactions on Graphics (TOG), vol. 32, no. 4, p. 104, 2013.

[26] C. Jiang, C. Schroeder, A. Selle, J. Teran, and A. Stomakhin, “The affine particle-in-cell method,” ACM Trans. Graph., vol. 34, no. 4, pp. 51:1–51:10, 2015.

[27] M. Macklin and M. Müller, “Unified particle physics for real-time applications,” ACM Transactions on Graphics (TOG), vol. 33, no. 4, p. 153, 2014.

[28] J. Schulman, P. Wolski, P. Dhariwal, A. Radford, and O. Klimov, “Proximal policy optimization algorithms,” arXiv:1707.06347, 2017.

[29] S. G. Johnson, “The nlopt nonlinear-optimization package,” 2014.

[30] Y. Fang, Y. Hu, S.-m. Hu, and C. Jiang, “A temporally adaptive material point method with regional time stepping,” Computer graphics forum, vol. 37, 2018.