# DIFFERENTIABLE PROGRAMMING FOR PHYSICAL SIMULATION

**Anonymous authors**
Paper under double-blind review

## ABSTRACT

We study the problem of learning and optimizing through physical simulations via differentiable programming. We present DiffSim, a new differentiable programming language tailored for building high-performance differentiable physical simulations. We demonstrate the performance and productivity of our language in gradient-based learning and optimization tasks on 10 different physical simulators. For example, a differentiable elastic object simulator written in our language is $4.6\times$ shorter than the hand-engineered CUDA version yet runs as fast, and is $188\times$ faster than TensorFlow. Using our differentiable programs, neural network controllers are typically optimized within only tens of iterations. Finally, we share the lessons learned from our experience developing these simulators, that is, *differentiating physical simulators does not always yield useful gradients of the physical system being simulated*. We systematically study the underlying reasons and propose solutions to improve gradient quality.

## 1 INTRODUCTION

Differentiable physical simulators are effective components in machine learning systems. For example, de Avila Belbute-Peres et al. (2018a) and Hu et al. (2019b) have shown that controller optimization with differentiable simulators converges one to three orders of magnitude faster than model-free reinforcement learning algorithms. Unfortunately, with existing tools it is difficult to implement differentiable simulators with high performance.

We present *DiffSim*, a new differentiable programming language for high performance physical simulations on both CPU and GPU. The language uses a "megakernel" approach, allowing the programmer to naturally fuse multiple stages of computation into a single kernel. Compared to computational graphs of array operations in TensorFlow (Abadi et al., 2016) and PyTorch (Paszke et al., 2017), DiffSim kernels have higher arithmetic intensity and are therefore more efficient for physical simulation tasks. To make porting existing physical simulation algorithms easier, the language is imperative and provides support for parallel loops and control flows (such as "if" statements), which are necessary constructs in physical simulations, especially when collisions are present. The programmer directly manipulates array elements via arbitrary indexing, thus allowing partial updates of global arrays. The DiffSim compiler then automatically parallelizes the code and generates gradient kernels using reverse-mode automatic differentiation (AD).

Using our language, we are able to quickly implement and differentiate a variety of physical simulators[1]. During this process, we encountered several algorithmic and implementation issues when taking gradients of physical simulators. In particular, since physical simulators are approximations of the physical process they model (e.g. an ordinary differential equation solver would discretize the time step), the gradients of a simulator might not be a good approximation of the physical process model. We identify several issues and propose simple fixes.

The main goal of this paper is to introduce our programming language and differentiable simulators, and share our experience in engineering these differentiable programs for gradient robustness. Our contributions are:

[1]Our language, compiler, and simulator code is submitted along with the paper and will be open-sourced. All the results in this work can be reproduced by a single python script. Many visual results in this work are presented in the supplemental video.

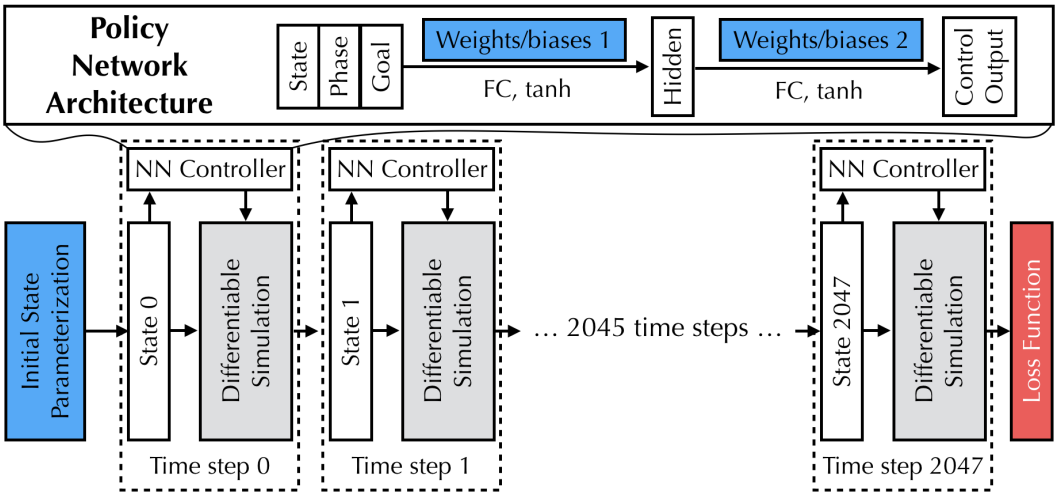



Figure 1: Our language allows us to seamlessly integrate a neural network (NN) controller and a physical simulation module, and update the weights of the controller or the initial state parametrization (blue). Our simulations typically have $512 \sim 2048$ time steps, and each time step has up to one thousand parallel operations.

- A new differentiable programming language, named "DiffSim", tailored for high performance physical simulation. It allows programmers to easily build end-to-end gradient-based deep learning systems with seamlessly integrated differentiable physical modules (Fig. 1).
- We create 10 different physical simulators in DiffSim and benchmark their performance against existing baselines. Future researchers can both reuse these differentiable simulators, and easily write new differentiable simulators using our programming language.
- We analyze the gradient behavior of these simulators and discuss why *a working physical model for forward simulation is often not good enough for differentiable simulation*. Most notably, *ignoring the precise time of impact* (a common practice in simulation scenarios when there is no high-speed object) may lead to completely misleading gradients. We demonstrate a simple yet effective solution to this problem.

## 2 The DiffSim Differentiable Programming Language

DiffSim is based on the Taichi programming language (Hu et al., 2019a). In this section, we first discuss high-level design decisions of DiffSim, and then introduce the language. For readers who are not familiar with Taichi, we provide a brief background in Appendix A. The DiffSim compiler design is discussed in Appendix B.

### 2.1 Design Decisions

Our design decisions are centered around conveniently building differentiable physical systems for deep learning.

**Imperative Programming** To make porting existing physical simulation algorithms easier, we adopt an imperative programming scheme, in contrast to the functional array programming languages that are popular in modern deep learning (Bergstra et al., 2010; Abadi et al., 2016). The user writes C-style for loops, branches, and accesses array elements directly using the Python frontend. The compiler automatically generates parallelized kernels and their gradients.

**Megakernels** Computation in DiffSim is organized into *kernels*, analogous to the layers in a deep learning framework, except the gradients of the kernels are automatically generated by automatic

differentiation. This allows us to define complex kernels without instantiating memory for every single intermediate stage. For example, in our language, the result of `a=b+c` is usually stored to a register instead of a tensor allocated in global memory. This results in much higher arithmetic intensity and memory efficiency.

**Two-stage Automatic Differentiation** To allow more control for assembling the kernels, structuring the backward pass of the gradient code, and interacting with external code, we propagate the gradients between kernels by constructing a tape, or a Wengert list (Wengert, 1964), of the kernel execution. In physical simulation, it is important for the user to perform a checkpointing optimization (Volin & Ostrovskii, 1985) and trade memory for computation, when the number of timesteps is high.

**Automatic Parallelization** We inherit the "parallel-for" construct from Taichi to map each loop iteration onto CPU/GPU threads. When necessary, the programmer can use atomic adds for thread safety. Our system can automatically differentiate these atomic operations, as long as they are order-independent.

### 2.2 The Language

Our system is built on top of a Python frontend of Taichi. Unlike Python, the DiffSim language is compiled, statically-typed, parallel, and differentiable. The goal is to allow users to write intuitive C-like control flow, while the compiler automatically parallelizes and differentiate the kernel.

Here we demonstrate the language via a differentiable mass-spring example. The goal is to optimize the rest lengths of the springs so that the triangle area formed by the three springs becomes $0.2$ at the end of the simulation. Visual illustrations and results are shown below in Fig. 2.

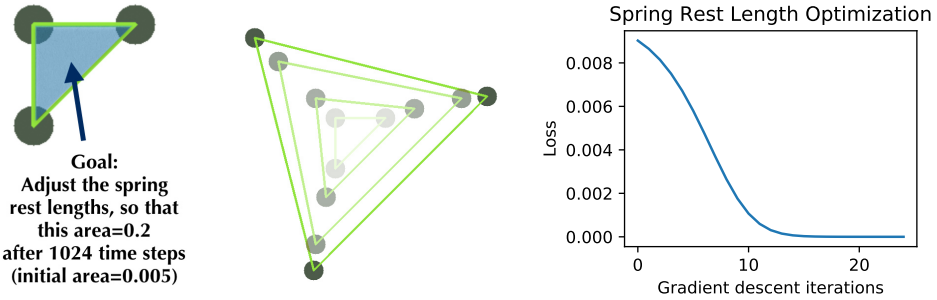



Figure 2: Optimizing the spring rest lengths so that the triangle's area reaches target of $0.2$. **Left**: Initially the springs have lengths $[0.1, 0.1, 0.14]$. **Middle:** After optimization the rest lengths are $[0.600, 0.600, 0.529]$. This means the springs will expand the triangle according to Hook's law and form a larger triangle. **Right:** The loss curve. [Reproduce: **mass_spring_simple.py**]

**Allocating Global Variables** Firstly we allocate a set of global tensors to store the simulation state. These tensors include a scalar `loss` of type `float32`, 2D tensors `x, v, force` of size `steps` $\times$`n_springs` and type `float32x2`, and 1D arrays of size `n_spring` for spring properties: `spring_anchor_a (int32), spring_anchor_b (int32), spring_length (float32).`

**Defining Kernels** A mass-spring system is modelled by Hook's law $\mathbf{F} = k(\|\mathbf{x}_a - \mathbf{x}_b\|_2 - l_0)\frac{\mathbf{x}_a - \mathbf{x}_b}{\|\mathbf{x}_a - \mathbf{x}_b\|_2}$ where $\mathbf{F}$ is spring force, $\mathbf{x}_a$ and $\mathbf{x}_b$ are the positions of two mass points, and $l_0$ is the rest length. The following kernel loops over all the springs and scatters forces to mass points:

```
@ti.kernel
def apply_spring_force(t: ti.i32):
  # Kernels can have parameters. Here t is a parameter with type int32.
  for i in range(n_springs): # A parallel for, preferably on GPU
    a, b = spring_anchor_a[i], spring_anchor_b[i]
    x_a, x_b = x[t - 1, a], x[t - 1, b]
    dist = x_a - x_b
```

```
    length = dist.norm() + 1e-4
    F = (length - spring_length[i]) * spring_stiffness * dist / length
    # Apply spring impulses to mass points. Use atomic_add for parallel safety.
    ti.atomic_add(force[t, a],  -F)
    ti.atomic_add(force[t, b],  F)
```

For each particle $i$, we use semi-implicit Euler time integration with damping:

$$\boldsymbol{v}_{t,i} = e^{-\Delta t \alpha} \boldsymbol{v}_{t-1,i} + \frac{\Delta t}{m_i} \mathbf{F}_{t,i}, \quad \mathbf{x}_{t,i} = \mathbf{x}_{t-1,i} + \Delta t \boldsymbol{v}_{t,i},$$

where $\boldsymbol{v}_{t,i}, \mathbf{x}_{t,i}, m_i$ are the velocity, position and mass of particle $i$ at time step $t$, respectively. $\alpha$ is a damping factor. The kernel is as follows:

```
@ti.kernel
def time_integrate(t: ti.i32):
  for i in range(n_objects):
    s = math.exp(-dt * damping) # Compile-time evaluation since dt and damping are constants
    v[t, i] = s * v[t - 1, i] + dt * force[t, i] / mass
    x[t, i] = x[t - 1, i] + dt * v[t, i]
```

Finally, we define the loss function:

```
@ti.kernel
def compute_loss(t: ti.i32):
  x01 = x[t, 0] - x[t, 1]
  x02 = x[t, 0] - x[t, 2]
  area = ti.abs(0.5 * (x01[0] * x02[1] - x01[1] * x02[0])) # Triangle area from cross product
  target_area = 0.1
  loss[None] = ti.sqr(area - target_area) # Everything in Taichi is a tensor.
  # "loss" is a scalar (0-D tensor), thereby indexed with [None].
```

**Assembling the Forward Program** With these components, we define the forward time integration:

```
def forward(output=None):
  for t in range(1, steps):
    apply_spring_force(t)
    time_integrate(t)
  compute_loss(steps - 1)
```

**Learning/Optimization with Gradients** The programmer uses `ti.Tape` to memorize forward kernel launches. It automatically replays the gradients of these kernels in reverse for backpropagation.

```
def main():
  # ... initialization code omitted ...
  for iter in range(200):
    clear_tensors()

    with ti.Tape(loss):
      forward()

    print('Iter=', iter, 'Loss=', loss[None])

    for i in range(n_springs): # Gradient descent
      spring_length[i] -= learning_rate * spring_length.grad[i]
```

Appendix C showcases a full DiffSim program for another minimization problem.

## 3 Evaluation and Applications

We evaluate DiffSim on 10 different physical simulators. Our examples cover large-scale continuum and small-scale rigid body simulations. All results can be reproduced with the provided script. The dynamic processes are visualized in the supplemental video.

Table 1: `diffmpm` performance comparison on an NVIDIA GTX 1080 Ti GPU. We benchmark in 2D using 6.4K particles. For the lines of code, we only include the essential implementation, excluding boilerplate code. [Reproduce: **python3 diffmpm_benchmark.py**]

| Approach | Forward Time | Backward Time | Total Time | # Lines of Code |
|---|---|---|---|---|
| TensorFlow | 13.20 ms | 35.70 ms | 48.90 ms (188.×) | 190 |
| CUDA | 0.10 ms | 0.14 ms | 0.24 ms (0.92×) | 460 |
| DiffSim | 0.11 ms | 0.15 ms | 0.26 ms (1.00×) | 110 |

**Differentiable Continuum Mechanics for Elastic Objects `[diffmpm]`** First, we build a differentiable continuum simulation for soft robotics applications as in Fig. 3. The physical system is governed by momentum and mass conservation, i.e.

$$\rho \frac{D\mathbf{v}}{Dt} = \nabla \cdot \sigma + \rho \mathbf{g}, \quad \frac{D\rho}{Dt} + \rho \nabla \cdot \mathbf{v} = 0.$$

We follow ChainQueen's implementation (Hu et al., 2019b) and use the moving least squares material point method (Hu et al., 2018) to simulate the system. DiffSim automatically transforms the program into a differentiable simulator. Compared with the original manually engineered CUDA implementation in ChainQueen, our new implementation is $4.2\times$ shorter in terms of lines of code, and runs almost as fast; compared with TensorFlow, DiffSim code is $1.7\times$ shorter and $188\times$ faster (Table 1). Using this simulator and an open-loop controller, we can easily teach a soft robot to walk.

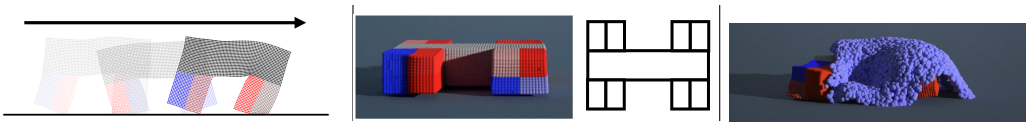


Figure 3: Controller optimization with our differentiable continuum simulators. **Left:** the 2D robot with four muscles. **Middle:** A 3D robot with 16 muscles and 30K particles crawling on the ground. [Reproduce: **python3 [diffmpm/diffmpm3d].py**] **Right:** We couple the robot (30K particles) and the liquid simulator (13K particles), and optimize its open-loop controller in this difficult situation.

**Differentiable liquid simulator `[liquid]`** We follow Tampubolon et al. (2017) and implemented a 3D differentiable liquid simulator. Our liquid simulation can be two-way coupled with elastic object simulation `[diffmpm]` (Figure 3, right).

**Differentiable mass-spring system `[mass_spring]`** We extend the mass-spring system in section 2.2 with ground collision and a NN controller. The optimization goal is to maximize the distance moved forward with 2048 time steps. We designed three mass-spring robots as shown in Fig. 4 (left).

**Differentiable billiard simulator `[billiards]`** A differentiable rigid body simulator is built for optimizing a billiards strategy (Fig. 4, middle). We used forward Euler for the billiard ball motion and conservation of momentum and kinetic energy for collision resolution.

**Differentiable rigid body simulators `[rigid_body]`** We built a more complex differentiable rigid body simulator for optimizing robot controllers (Fig. 4, right). This simulator supports rigid body collision and friction, spring forces and actuation.

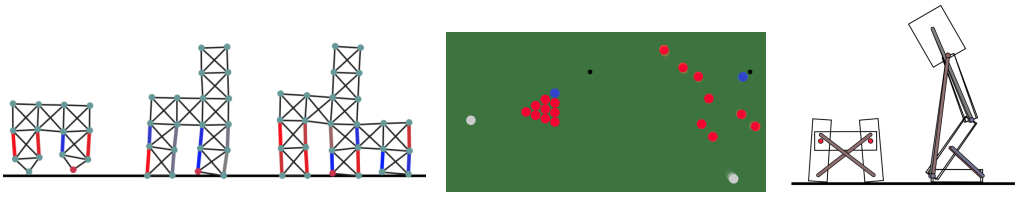
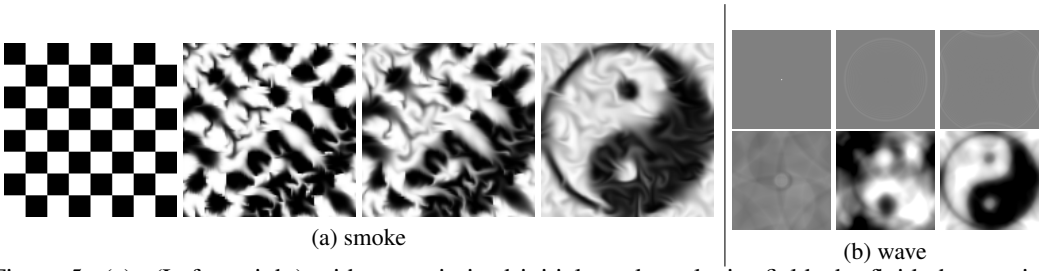


Figure 4: **Left:** Three mass-spring robots. The red and blue springs are actuated. A two layer NN is used as controller. [Reproduce: **python3 mass_spring.py [1/2/3] train**]. **Middle:** Optimizing billiards. The optimizer adjusts the initial position and velocity of the white ball, so that the blue ball will reach the target destination (black dot). [Reproduce: **python3 billiards.py**] **Right:** Optimizing a robot walking. The rigid robot is controlled with a NN controller and learned to walk in 20 gradient descent iterations. [Reproduce: **python3 rigid_body.py**]

**Differentiable incompressible fluid simulator** `[smoke]` We implemented a smoke simulator with semi-Lagrangian advection Stam (1999) and implicit pressure projection, following the example in Autograd (Maclaurin et al., 2015). Using gradient descent optimization on the initial velocity field, we are able to find a velocity field that changes the pattern of the fluid to a target image (Fig. 5a). We compare the performance of our system against Autograd and PyTorch in Table 2.

Table 2: `smoke` benchmark against Autograd and PyTorch. We used a $110 \times 110$ grid and 100 time steps, each with 10 Jacobi pressure projections. [Reproduce: **python3 [smoke/smoke_autograd/smoke_pytorch].py**]. Note that the Autograd program uses float64 precision (which approximately doubles the runtime), while PyTorch and DiffSim use float32.

| Approach | Forward Time | Backward Time | Total Time | # Essential LoC |
|---|---|---|---|---|
| PyTorch (CPU) | 451 ms | 399 ms | 850 ms (5.0×) | 74 |
| PyTorch (GPU) | 331 ms | 562 ms | 893 ms (5.2×) | 74 |
| Autograd (CPU) | 347 ms | 1445 ms | 1792 ms (10.5×) | 51 |
| DiffSim (CPU) | 83 ms | 566 ms | 649 ms (3.8×) | 75 |
| DiffSim (GPU) | 68 ms | 103 ms | 171 ms (1.0×) | 75 |



(a) smoke (b) wave

Figure 5: (a): (Left to right) with an optimized initial smoke velocity field, the fluid changes its pattern to a "Taichi" symbol. [Reproduce: **python3 smoke_taichi.py**] (b): Unoptimized (top three) and optimized (bottom three) waves at time step 3, 189, and 255. [Reproduce: **python3 wave.py**]

**Differentiable height field shallow water simulator** `[wave]` We adopt the wave equation in Wang et al. (2018) to model shallow water height field evolution:

$$\ddot{u} = c^2 \nabla^2 u + c\alpha \nabla^2 \dot{u}, \tag{1}$$

where $u$ is the height of shallow water, $c$ is the "speed of sound" and $\alpha$ is a damping coefficient. We use the $\dot{u}$ and $\ddot{u}$ notations for the first and second order partial derivatives of $u$ w.r.t time $t$ respectively. We set the "Taichi" symbol as the target pattern. Fig. 5b shows the unoptimized and optimized final wave patterns. More details on discretization is in Appendix D.

**Differentiable water renderer** `[water_renderer]` We implemented differentiable renderers to visualize the refracting water surfaces from `wave`. We use finite differences to reconstruct the water surface models based on the input height field and refract camera rays to sample the images, using bilinear interpolation for meaningful gradients. To show our system works well with other differentiable programming systems, we use an adversarial optimization goal: fool VGG-16 into thinking that the refracted squirrel image is a goldfish (Fig. 6).

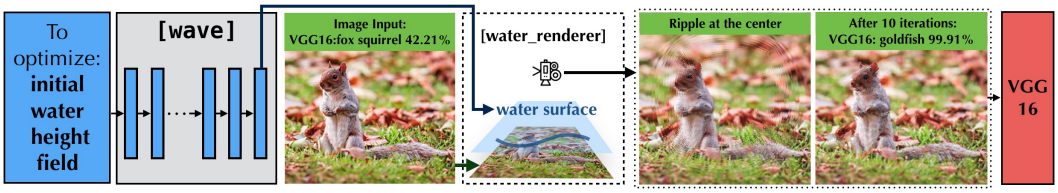
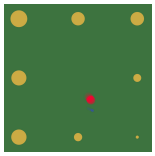



Figure 6: This three-stage program (simulation, rendering, recognition) is end-to-end differentiable. Our optimized initial water height field evolves to form a refraction pattern that perturbs the image into one that fools VGG16 (99.91% goldfish). [Reproduce: **python3 water_renderer.py**]

**Differentiable Electric Field Simulator** `[electric]` Recall Coulomb's law: $\mathbf{F} = k\frac{q_1 q_2}{r^2}\hat{\mathbf{r}}$. In the right figure, there are 8 electrodes carrying static charge. The red ball also carries static charge. The controller, which is a two-layer neural network, tries to manipulate the electrodes so that the red ball follows the path of the blue ball. The bigger the electrode, the more positive charge it carries.



**Differentiable Volume Renderer** `[volume_renderer]` We implemented a basic volume renderer that simply uses ray marching (we ignore light, scattering, etc.) to integrate a density field over each camera ray. In this task, we render a number of target images from different viewpoints, with the camera rotated around the given volume. The goal is then to optimize for the density field of the volume that would produce these target images: we render candidate images from the same viewpoints and compute an L2 loss between them and the target images, before performing gradient descent on the density field (Fig. 7).

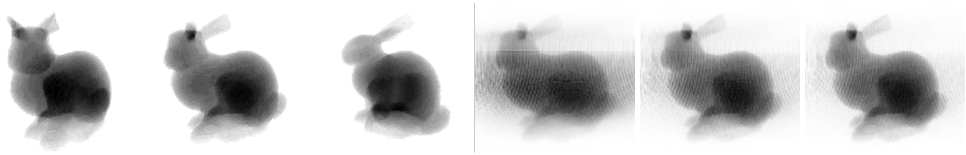


Figure 7: Volume rendering of bunny shaped density field. **Left:** 3 (of the 7) target images. **Right:** optimized images of the middle bunny after iteration 2, 50, 100. [Reproduce: **python3 volume_renderer.py**]

## 4 Building Differentiable Simulators with Robust Gradients

DiffSim makes it possible to easily implement and experiment with various physical simulations. Although some provide reliable and useful gradients, others present difficulties during backpropagation. A typical physical simulation can have thousands of time steps, and each step may contain thousands of tensor operations (i.e., "layers"). Our layers are deeper than a typical convolutional neural network, while wider than a recurrent neural network. At the same time the non-linearities introduced by the physical systems are different than the typical element-wise non-linearities.

Just as in deep neural network training, obtaining robust gradients in physical simulation that guide optimization is not always easy. The most notable pitfall is to assume that *differentiating a well-behaving forward simulation yields a useful gradient version for corresponding inverse problems.* As we will show later, sometimes a good-enough forward simulation can be automatically differentiated into a gradient program that yields completely incorrect gradients of the physical process.

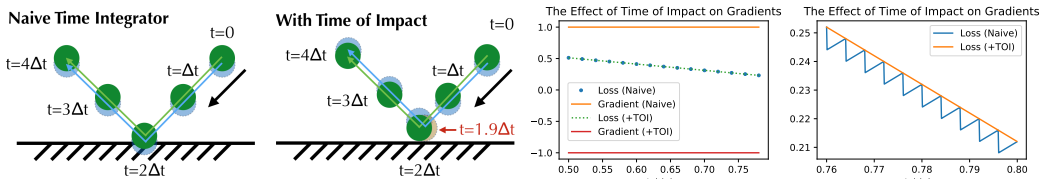



Figure 8: How gradients can go wrong with naive time integrators. For clarity we use a large $\Delta t$ here. **Left:** Since collision detection only happens at multiples of $\Delta t$ ($2\Delta t$ in this case), lowering the initial position of the ball (light blue) leads to a lowered final position. **Middle Left:** By improving the time integrator to support continuous time of impact (TOI), collisions can be detected at any time, e.g. $1.9\Delta t$ (light red). Now the blue ball ends up higher than the green one. **Middle Right:** Although the two time integration techniques lead to almost identical forward results (in practice $\Delta t$ is small), the naive time integrator gives an incorrect gradient of 1, but adding TOI yields the correct gradient. Please see our supplemental video for a better demonstration. [Reproduce: **python3 rigid_body_toi.py**] **Right:** When zooming in, the loss of the naive integrator is decreasing, and the saw-tooth pattern explains the positive gradients. [Reproduce: **python3 rigid_body_toi.py zoom**]

**Time is still discrete and non-differentiable** Most physical simulations rely on time discretization, and gradients of time itself are not considered during automatic differentiation. In short, we are only differentiating the simulation *program* of a time-discretized physical process, instead of the process itself.

Consider the rigid ball example in Fig. 8 (left), where a rigid ball collides with a friction-less ground. Gravity is ignored, and due to conservation of kinetic energy the ball keeps a constant speed even after this elastic collision.

In the forward simulation, using a small $\Delta t$ often leads to a reasonable result, as done in many physics simulators. Lowering the initial ball height will increase the final ball height, since there is less distance to travel before the ball hits the ground and more after (see the loss curves in Fig.8, middle right). However, using a naive time integrator, no matter how small $\Delta t$ is, the evaluated gradient of final height w.r.t. initial height will be $1$ instead of $-1$. This counter-intuitive behavior is due to the fact that time discretization itself is not differentiated by the compiler. Fig. 8 explains this effect in greater detail.

We propose a simple solution of adding continuous collision resolution (see, for example, Redon et al. (2002)), which considers precise time of impact (TOI), to the forward program (Fig. 8, middle left). Although it barely improves the forward simulation (Fig. 8, middle right), the gradient will be corrected effectively (Fig. 8, right). The details of continuous collision detection are in Appendix E. In real-world simulators, we find the TOI technique leads to significant improvement in gradient quality in controller optimization tasks (Fig. 9).

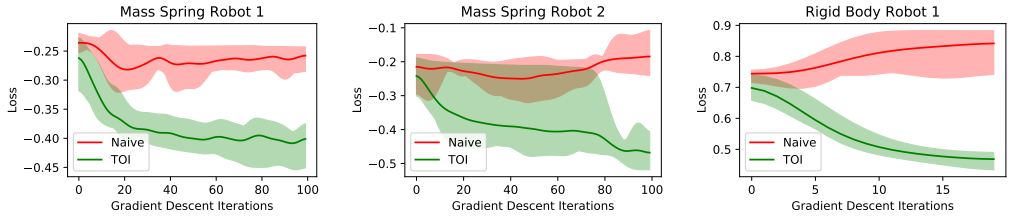



Figure 9: Adding TOI greatly improves gradient and optimization quality. Each experiment is repeated five times. Having TOI or not barely affects forward simulation: in the supplemental video, we show that a robot controller optimized in a simulator with TOI, actually works well in a simulator without TOI. [Reproduce: **python3 [mass_spring/rigid_body.py] [1/2] plot && python3 plot_losses.py**]

**Gradient explosion in simulations with long time horizons** Consider this artificial 1D car example: an open-loop controller provides a constant throttle which translates into acceleration. The

discretized ODE is $v_t = e^{-\Delta t \beta} v_{t-1} + a, x_t = x_{t-1} + v_t$, where $a$ is the controller output, and $\beta$ is a damping factor (Fig. 10, left). Assuming the loss function is the distance travelled and a fixed $\Delta t$ per time step, we change the simulation time by increasing the number of time steps.

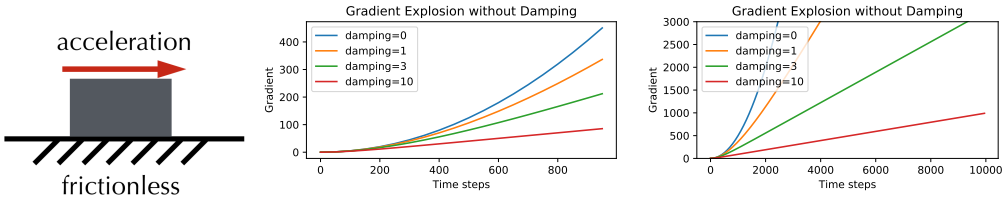



Figure 10: **Left:** A slider moving right with constant acceleration and no friction. **Middle:** Without damping, the gradient of distance w.r.t. acceleration grows quadratically with respect to the number of time steps $n$, i.e. $\partial L / \partial a = 1/2 \cdot (n \Delta t)^2$. **Right:** Even with damping, gradients will grow linearly with simulation time. [Reproduce: **python3 gradient_explosion.py [large]**]

As shown in Fig. 10, the gradient magnitude grows quadratically. In practice, physical systems often have damping, and energy injected by NNs will not last forever. In this case, controller gradients will be proportional to the number of time steps (Fig. 10, right). "Damping" does not have to be explicit. A lot of physical processes intrinsically introduce energy losses, such as friction and inelastic collisions in `rigid_body`, and numerical viscosity in `wave` and `smoke`. **The rule of thumb: whenever you double the simulation time, halve your learning rate to keep the gradient at the same magnitude.**

In reality, controllers are usually closed-loop and the positive feedback of controller outputs may make the gradient magnitude even more unpredictable. Practically, we have found adding damping to the simulator stabilizes gradients. Intuitively, with damping, the actuation will have an exponentially decaying influence, limiting the gradients from propagating back. In Appendix F, we additionally discuss initialization, local minima, and discontinuity issues in physical simulation.

**Future work** Despite the aforementioned gradient issues, we remain optimistic about differentiable simulators. We raise the following questions for potentially meaningful future directions:

- How do we combine differentiable simulators with actor-critic methods in reinforcement learning, so that value networks provide smooth long-term gradients and differentiable simulators provide accurate short-term gradients?
- Existing physical simulation models have been largely built for forward simulation, but what are good models for backpropagation? I.e., are there models that lie in a sweet spot between being physically accurate during forward simulation, and being stably differentiable during backpropagation?
- How do we introduce exploration mechanisms for controller optimization with differentiable simulators? Again, could ideas from reinforcement learning (e.g. solutions to "exploitation v.s. exploration") be applied to optimization approaches via differentiable physical systems?

## 5 Related Work

**Differentiable programming** The recent rise of deep learning has motivated the development of differentiable programming libraries for deep NNs, most notably auto-differentiation frameworks such as Theano (Bergstra et al., 2010), TensorFlow (Abadi et al., 2016) and PyTorch (Paszke et al., 2017). These frameworks usually provide *coarse-grained* operations (e.g., convolutions) on *dense* tensors as building blocks, which are "atoms" of these programming systems and cannot be further customized. However, physical simulation requires complex and customizable operations due to the intrinsic computational irregularity. Using the aforementioned frameworks, programmers have to compose these coarse-grained basic operations into desired complex operations. Doing so often leads to unsatisfactory performance.

Earlier work on automatic differentiation focuses on transforming existing scalar code to obtain derivatives (e.g. Utke et al. (2008), Hascoet & Pascual (2013), Pearlmutter & Siskind (2008)). A recent trend has emerged for modern programming languages to support differentiable function transformations through annotation (e.g. Innes et al. (2019), Wei et al. (2019)). While these frameworks and compilers enable differentiating general programming languages, these transformations often do not target massively parallel architectures such as GPUs when compiling complex kernels, since it involves automatic parallelization of arbitrary code.

Differentiable functional array programming languages such as Halide (Ragan-Kelley et al., 2013; Li et al., 2018b), Autograd (Maclaurin et al., 2015), and JAX (Bradbury et al., 2018) operate on arrays instead of scalars to utilize parallelism, and Halide separates the concerns of storage and computation from actual algorithm design. Enoki (Jakob, 2019) is an SPMD framework also targeted at differentiable simulation. It uses template metaprogramming to transform C++ code into derivatives code, while just-in-time compiling the transformed code into PTX assembly. Instead of operating on arrays that are immutable, DiffSim uses an imperative style to make porting existing physical simulation algorithms easier, while offering fine-grained control such as checkpointing.

**Differentiable Physical Simulators** Building differentiable simulators for robotics and machine learning has recently increased in popularity. Without differentiable programming, Battaglia et al. (2016), Chang et al. (2016) and Mrowca et al. (2018) used NNs to approximate the physical process and used the NN gradients as the approximate simulation gradients. Degrave et al. (2016) and de Avila Belbute-Peres et al. (2018b) used Theano and PyTorch respectively to build differentiable rigid body simulators. Schenck & Fox (2018) differentiates position-based fluid using custom CUDA kernels. Popović et al. (2000) used a differentiable rigid body simulator for manipulating physically based animations. The ChainQueen differentiable elastic object simulator (Hu et al., 2019b) implements forward and gradient versions of continuum mechanics in hand-written CUDA kernels, leading to performance that is two orders of magnitude higher than a pure TensorFlow implementation. 3D deep learning also often incorporates differentiable rendering operations (OpenDR (Loper & Black, 2014), N3MR (Kato et al., 2018), redner (Li et al., 2018a), Mitsuba 2 (Nimier-David et al., 2019)) for modelling the effects of geometry and appearance of objects on the images.

## 6 Conclusion

We have presented DiffSim, a new differentiable programming language designed specifically for building high performance differentiable physical systems for deep learning. We have productively used the system to build 10 different simulators and integrated them into deep learning systems. Benchmarks show higher performance with our language than existing tools. Finally, we discuss why a naive automatically differentiated physical program may not provide useful gradients. We hope our programming language, our differentiable simulators, and our discussions on how to build simulators with robust gradients can inspire future work on combining differentiable physical simulation and deep learning.

## Bibliography

Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pp. 265–283, 2016.

Peter W. Battaglia, Razvan Pascanu, Matthew Lai, Danilo Rezende, and Koray Kavukcuoglu. Interaction networks for learning about objects, relations and physics. 2016.

James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano: A cpu and gpu math compiler in python. In *Proc. 9th Python in Science Conf*, volume 1, pp. 3–10, 2010.

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, and Skye Wanderman-Milne. JAX: composable transformations of Python+NumPy programs, 2018. URL http://github.com/google/jax.

Michael B Chang, Tomer Ullman, Antonio Torralba, and Joshua B Tenenbaum. A compositional object-based approach to learning physical dynamics. *ICLR 2016*, 2016.

Filipe de Avila Belbute-Peres, Kevin Smith, Kelsey Allen, Josh Tenenbaum, and J Zico Kolter. End-to-end differentiable physics for learning and control. In *Advances in Neural Information Processing Systems*, pp. 7178–7189, 2018a.

Filipe de Avila Belbute-Peres, Kevin A Smith, Kelsey Allen, Joshua B Tenenbaum, and J Zico Kolter. End-to-end differentiable physics for learning and control. In *Neural Information Processing Systems*, 2018b.

Jonas Degrave, Michiel Hermans, Joni Dambre, et al. A differentiable physics engine for deep learning in robotics. *arXiv preprint arXiv:1611.01652*, 2016.

Laurent Hascoet and Valérie Pascual. The Tapenade automatic differentiation tool: Principles, model, and specification. 39(3):20:1–20:43, 2013.

Yuanming Hu, Yu Fang, Ziheng Ge, Ziyin Qu, Yixin Zhu, Andre Pradhana, and Chenfanfu Jiang. A moving least squares material point method with displacement discontinuity and two-way rigid body coupling. 37(4):150, 2018.

Yuanming Hu, Tzu-Mao Li, Luke Anderson, Jonathan Ragan-Kelley, and Frédo Durand. Taichi: A language for high-performance computation on spatially sparse data structures. In *SIGGRAPH Asia 2019 Technical Papers*, pp. 201. ACM, 2019a.

Yuanming Hu, Jiancheng Liu, Andrew Spielberg, Joshua B Tenenbaum, William T Freeman, Jiajun Wu, Daniela Rus, and Wojciech Matusik. Chainqueen: A real-time differentiable physical simulator for soft robotics. *Proceedings of IEEE International Conference on Robotics and Automation (ICRA)*, 2019b.

Mike Innes, Alan Edelman, Keno Fischer, Chris Rackauckus, Elliot Saba, Viral B Shah, and Will Tebbutt. Zygote: A differentiable programming system to bridge machine learning and scientific computing. *arXiv preprint arXiv:1907.07587*, 2019.

Wenzel Jakob. Enoki: structured vectorization and differentiation on modern processor architectures, 2019. https://github.com/mitsuba-renderer/enoki.

Hiroharu Kato, Yoshitaka Ushiku, and Tatsuya Harada. Neural 3d mesh renderer. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 3907–3916, 2018.

Stig Larsson and Vidar Thomée. *Partial differential equations with numerical methods*, volume 45. Springer Science & Business Media, 2008.

Tzu-Mao Li, Miika Aittala, Frédo Durand, and Jaakko Lehtinen. Differentiable monte carlo ray tracing through edge sampling. In *SIGGRAPH Asia 2018 Technical Papers*, pp. 222. ACM, 2018a.

Tzu-Mao Li, Michaël Gharbi, Andrew Adams, Frédo Durand, and Jonathan Ragan-Kelley. Differentiable programming for image processing and deep learning in Halide. *ACM Transactions on Graphics (TOG)*, 37(4):139, 2018b.

Matthew M Loper and Michael J Black. Opendr: An approximate differentiable renderer. In *European Conference on Computer Vision*, pp. 154–169. Springer, 2014.

Dougal Maclaurin, David Duvenaud, and Ryan P Adams. Autograd: Effortless gradients in numpy. In *ICML 2015 AutoML Workshop*, volume 238, 2015.

Damian Mrowca, Chengxu Zhuang, Elias Wang, Nick Haber, Li Fei-Fei, Joshua B Tenenbaum, and Daniel LK Yamins. Flexible neural representation for physics prediction. *1806.08047*, 2018.

Merlin Nimier-David, Delio Vicini, Tizian Zeltner, and Wenzel Jakob. Mitsuba 2: A retargetable forward and inverse renderer. *Transactions on Graphics (Proceedings of SIGGRAPH Asia)*, 38 (6), November 2019. doi: 10.1145/3355089.3356498.

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

Barak A. Pearlmutter and Jeffrey Mark Siskind. Reverse-mode AD in a functional framework: Lambda the ultimate backpropagator. *ACM Transactions on Programming Languages and Systems*, 30(2):7:1–7:36, 2008.

Jovan Popović, Steven M Seitz, Michael Erdmann, Zoran Popović, and Andrew Witkin. Interactive manipulation of rigid body simulations. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pp. 209–217. ACM Press/Addison-Wesley Publishing Co., 2000.

Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *SIGPLAN Not.*, 48(6):519–530, jun 2013.

Stéphane Redon, Abderrahmane Kheddar, and Sabine Coquillart. Fast continuous collision detection between rigid bodies. In *Computer graphics forum*, volume 21, pp. 279–287. Wiley Online Library, 2002.

Connor Schenck and Dieter Fox. Spnets: Differentiable fluid dynamics for deep neural networks. *arXiv preprint arXiv:1806.06094*, 2018.

Jos Stam. Stable fluids. In *Siggraph*, volume 99, pp. 121–128, 1999.

Andre Pradhana Tampubolon, Theodore Gast, Gergely Klár, Chuyuan Fu, Joseph Teran, Chenfanfu Jiang, and Ken Museth. Multi-species simulation of porous sand and water mixtures. *ACM Transactions on Graphics (TOG)*, 36(4):105, 2017.

Jean Utke, Uwe Naumann, Mike Fagan, Nathan Tallent, Michelle Strout, Patrick Heimbach, Chris Hill, and Carl Wunsch. Openad/f: A modular open-source tool for automatic differentiation of fortran codes. 34(4):18, 2008.

Yu. M. Volin and G. M. Ostrovskii. Automatic computation of derivatives with the use of the multilevel differentiating technique — I: Algorithmic basis. *Computers and Mathematics with Applications*, 11:1099–1114, 1985.

Jui-Hsien Wang, Ante Qu, Timothy R Langlois, and Doug L James. Toward wave-based sound synthesis for computer animation. *ACM Trans. Graph.*, 37(4):109–1, 2018.

Richard Wei, Marc Rasi Dan Zheng, and Bart Chrzaszcz. Differentiable programming mega-proposal. https://github.com/dan-zheng/swift/blob/differentiable-programming/docs/DifferentiableProgramming.md, 2019. Accessed: 2019-09-25.

R. E. Wengert. A simple automatic derivative evaluation program. *Communications of the ACM*, 7 (8):463–464, aug 1964.

## A BACKGROUND ON THE TAICHI PROGRAMMING LANGUAGE

Taichi is an imperative programming language that delivers both high performance and high productivity on modern hardware. The key design that distinguishes Taichi from other imperative programming languages such as C++/CUDA is the decoupling of computation from data structures. In the Taichi language, multi-level (sparse) voxel grids of various forms can be easily constructed using a data structure mini language, and programmers code algorithms on these sparse data structures as if they are dense, using indices (i.e. `x[i, j, k]`). The Taichi compiler then takes both the sparse data structure and algorithm information to apply performance optimizations. Taichi offers programmers with "parallel-for"s that saturates computing cores of modern processors. According to the benchmark in Hu et al. (2019a), programs written in Taichi are $10.0\times$ shorter and $4.55\times$ faster than hand-optimized baseline implementations, due to better sparse data structure abstraction and the optimizing compiler. For more details, readers are referred to Hu et al. (2019a).

## B COMPILER DESIGN AND IMPLEMENTATION

Existing differentiable programming tools for deep learning are typically centered around large data blobs. For example, in AlexNet, the second convolution layer has size $27 \times 27 \times 128 \times 128$. These tools usually provide users with both low-level operations such as tensor add and mul, and high-level operations such as convolution. The bottleneck of typical deep-learning-based computer vision tasks are convolutions, so the provided high-level operations, with very high arithmetic intensity[2], can fully exploit hardware capability.

However, these high-level operations are "atoms" of such differentiable programming tools, and cannot be further customized. Users often have to use low-level operations to compose their desired high-level operations. This introduces a lot of temporary buffers, and potentially excessive GPU kernel launches. As shown in Hu et al. (2019b), a pure TensorFlow implementation of a complex physical simulator is $132\times$ slower than a CUDA implementation, due to excessive GPU kernel launches and the lack of producer-consumer locality[3].

Unlike existing differentiable programming tools that use arrays as coarse-grained elementary objects, in DiffSim users directly manipulate fine-grained elements in these arrays. A DiffSim program consists of multiple kernels that are just-in-time compiled, and each kernel is composed of (parallel) imperative instructions. The built-in reverse-mode automatic differentiation system is two-scale: we maintain a coarse-grained tape recording kernel launches at run time, and a fine-grained tape for computation inside the kernels at compile time for high performance. These design decisions make DiffSim suitable for building differentiable physics engines. We adopt standard reverse-mode automatic differentiation.

**Primal and adjoint kernels** In DiffSim, (primal) kernels are operators that take as input multiple sparse tensors (e.g., $X, Y$) and output another set of sparse tensors. Mathematically, kernel $f$ has the form

$$f(X_0, X_1, .., X_n) = Y_0, Y_1, \ldots, Y_m.$$

Kernels usually execute uniform operations on sparse tensors. For example, a simple 3D kernel $g$ may do the operation $Z_{ijk} = g(X, Y)_{ijk} = X_{ijk} + \sin(Y_{i,j,k-1})$. All tensors are initialized to be unpopulated (i.e. empty). Taichi automatically populates output tensors ($Z$ in this case) so that its data structure holds all non-zero entries (e.g. voxels in 3D). A computation task typically involves a series of kernels, each taking a subset of previously generated tensors as inputs.

When it comes to differentiable programming, a loss function is defined on the final output tensors. The gradients of the loss function "$L$" with respect to each tensor are stored in *adjoint tensors* and computed via *adjoint kernels*.

The adjoint tensor of (primal) tensor $X_{ijk}$ is denoted as $X^*_{ijk}$. Its entries are defined by $X^*_{ijk} = dL/dX_{ijk}$. At a high level, our automatic differentiation (AD) system transforms a *primal* kernel

[2]FLOPs per byte loaded from/stored to main memory.

[3]The CUDA kernels have much higher arithmetic intensity compared to the TensorFlow computational graph system. In other words, when implementing in CUDA immediate results are cached in registers, while in TensorFlow they are "cached" in main memory.

into its *adjoint* form. Mathematically,

$$\textbf{(primal)}\ f(X_0, X_1, .., X_n) = Y_0, Y_1, \ldots, Y_m$$

$$\Big\downarrow \text{Reverse-Mode Automatic Differentiation}$$

$$\textbf{(adjoint)}\ f^*(X_0, X_1, .., X_n, Y_0^*, Y_1^*, \ldots, Y_m^*) = X_0^*, X_1^*, .., X_n^*.$$

**Differentiating within kernels: The "make_adjoint" pass (reverse-mode AD)** After the first simplification pass[4], which simplifies most branching into "select(cond, x, y)", the "make_adjoint" pass transforms a forward evaluation (primal) kernel into its gradient accumulation ("adjoint") kernel. Note that even nested "if" statements will be transformed into "select"s during simplification. AD is implemented as a transform on the IR, called "make_adjoint". It takes straight-line code directly and operates on the hierarchical intermediate representation (IR) of Taichi. An outer parallel for loop is allowed for the primal kernel. The DiffSim compiler will distribute these parallel iterations onto CPU/GPU threads.

During the "make_adjoint pass", for each SSA statement, a local adjoint variable will be allocated for contribution accumulation. The compiler will traverse the statements in reverse order, and accumulate the gradients to the corresponding adjoint local variable.

An example: 1D array operation $y_i = \sin x_i^2$ has its IR representation as follows:

---

**for** i $\in$ range(0, 16, step 1) **do**
  $\%1 =$ load $x[i]$
  $\%2 =$ mul $\%1, \%1$
  $\%3 = \sin(\%2)$
  store $y[i] = \%3$
**end for**

---

The above primal kernel will be transformed into the following adjoint kernel:

---

**for** i in range(0, 16, step 1) **do**
  // adjoint variables
  $\%1adj =$ alloca $0.0$
  $\%2adj =$ alloca $0.0$
  $\%3adj =$ alloca $0.0$
  // original forward computation
  $\%1 =$ load $x[i]$
  $\%2 =$ mul $\%1, \%1$
  $\%3 = sin(\%2)$
  // reverse accumulation
  $\%4 =$ load $y\_adj[i]$
  $\%3adj$ += $\%4$
  $\%5 = cos(\%2)$
  $\%2adj$ += $\%3adj * \%5$
  $\%1adj$ += $2 * \%1 * \%2adj$
  atomic add $x\_adj[i], \%1adj$
**end for**

---

(Note that for clarity the transformed code is not strictly SSA here. The actual IR has more instructions.) A following simplification pass will simplify redundant instructions generated by the AD pass.

[4]Taichi uses a hierarchical static single assignment (SSA) intermediate representation (IR) as its internal program representation. The Taichi compiler applies multiple transform passes to lower and simplify the SSA IR in order to get high-performance binary code.

**Explicit storage control** Memory consumption and performance directly relates to the scalability of differentiable programs. Reverse-mode AD typically requires recording the whole computational history for backpropagation, leading to much larger memory consumption than traditional forward programs. Therefore, we expose the tensor storage detail to users, which makes it possible to micro-manage memory and possibly trade computation for memory (e.g., "checkpointing"). The draw back of such flexibility is the risk of having the computational history corrupted by the user, although in practice this is not an issue for experienced DiffSim programmers.

## C A FULL DIFFSIM PROGRAM EXAMPLE

Here we briefly introduce the language constructs via a simple minimization problem on vectors $x_i$, with $y_i$ given. The loss function is defined as

$$L(x) = \frac{1}{2}\sum_{i=0}^{n-1}(x_i - y_i)^2.$$

Clearly the solution is $x_i = y_i$. The following code finds this solution using gradient descent:

```
import taichi_lang as ti
import random

# Declare the tensors
n = 32
x = ti.var(dt=ti.f32)
y = ti.var(dt=ti.f32)
L = ti.var(dt=ti.f32)

# Specify the layout of the primal and adjoint tensors
@ti.layout
def data():
  # Place tensors and their adjoints (*.grad)
  ti.root.dense(ti.i, n).place(x, y, x.grad, y.grad)
  ti.root.place(L, L.grad)

@ti.kernel
def reduce():
  for i in range(n):
    ti.atomic_add(L, 0.5 * (x[i] - y[i]) ** 2)

# Initialize the vectors
for i in range(n):
  x[i] = 0.0
  y[i] = random.random()

@ti.kernel
def reset_gradients():
  L.grad[None] = 1
  for i in x:
    x.grad[i] = 0
    y.grad[i] = 0

@ti.kernel
def update():
  for i in x:
    x[i] -= x.grad[i] * 0.1

# Optimize with 100 gradient descent iterations
for k in range(100):
  L[None] = 0
  reduce()
  print('L =', L[None])

  reset_gradients()
  # Call the *adjoint* version of the reduce kernel
  reduce.grad()
  update()

for i in range(n):
  # Now you should approximately have x[i] == y[i]
  print(x[i], y[i])
```

The layout language is from Taichi (Hu et al., 2019a). An alternative way to automatically place the gradient tensors is as follows:

```
@ti.layout
def data():
  ti.root.dense(ti.i, n).place(x, y)
  ti.root.place(L)
  # Automatically place the adjoint tensors immediately next to the primal tensor
  ti.lazy_grad()
```

An efficient layout that ensures consecutive elements in the tensor are also consecutive in memory:

```
@ti.layout
def data():
  for v in [x, y, x.grad, y.grad]:
    ti.root.dense(ti.i, n).place(v)
  ti.root.place(L, L.grad)
```

Generalizing the program to sparse tensors requires using a different two-level sparse data structure:

```
@ti.layout
def data():
  # Use a two-level sparse 1D array for storing x and y
  # A sparse pointer array that points to
  #    blocks of 16 elements of
  #    x_i, y_i and their gradients, is defined:
  ti.root.dense(ti.i, n // 16).pointer().dense(ti.i, 16).place(x, y)
  ti.root.place(L)
  ti.lazy_grad()
```

A more efficient sparse layout:

```
@ti.layout
def data():
  block = ti.root.dense(ti.i, n // 16).pointer()
  for v in [x, y, x.grad, y.grad]:
    block.dense(ti.i, 16).place(v)
  ti.root.place(L, L.grad)
```

In short, all the original data structure nodes from Taichi are supported, for organizing primal tensors (e.g., `x`) and adjoint tensors (e.g., `x.grad`). For quick prototyping, `ti.lazy_grad()` conveniently applies a default data organization for the adjoint tensors. Invoking the adjoint version of a kernel, e.g. `reduce`, requires calling `reduce.grad()`.

## D WAVE SIMULATOR DETAILS

Wang et al. (2018) used the finite different time-domain (FDTD) method (Larsson & Thomée, 2008) to discretize Eqn. 1, yielding an update scheme:

$$u_{t,i,j} = 2u_{t-1,i,j} + (c^2\Delta t^2 + c\alpha\Delta t)(\nabla^2 u)_{t-1,i,j} - p_{t-2,i,j} - c\alpha\Delta t(\nabla^2 u)_{t-2,i,j},$$

where

$$(\nabla^2 u)_{t,i,j} = \frac{-4u_{t,i,j} + u_{t,i,j+1} + u_{t,i,j-1} + u_{t,i+1,j} + u_{t,i-1,j}}{\Delta x^2}.$$

We implemented this wave simulator in DiffSim to simulate shallow water. We used a grid of resolution $128 \times 128$ and 256 time steps. The loss function is defined as

$$L = \sum_{i,j} \Delta x^2 (u_{T,i,j} - \hat{u}_{i,j})^2$$

where $T$ is the final time step, and $\hat{u}$ is the target height field. 200 gradient descent iterations are then used to optimize the initial height field. We set $\hat{u}$ to be the pattern “Taichi", and Fig. 5b shows the unoptimized and optimized wave evolution.

## E FIXING GRADIENTS WITH TIME OF IMPACT AND CONTINUOUS COLLISION DETECTION

Here is a naive time integrator in the mass-spring system example:

```
@ti.kernel
def advance(t: ti.i32):
  for i in range(n_objects):
    s = math.exp(-dt * damping)
    new_v = s * v[t - 1, i] + dt * gravity * ti.Vector([0.0, 1.0])
    old_x = x[t - 1, i]
    depth = old_x[1] - ground_height
    if depth < 0 and new_v[1] < 0:
      # assuming a sticky ground (infinite coefficient of friction)
      new_v[0] = 0
      new_v[1] = 0

    # Without considering time of impact, we assume the whole dt uses new_v
    new_x = old_x + dt * new_v

    v[t, i] = new_v
    x[t, i] = new_x
```

Implementing TOI in this system is relative straightforward:

```
@ti.kernel
def advance_toi(t: ti.i32):
  for i in range(n_objects):
    s = math.exp(-dt * damping)
    old_v = s * v[t - 1, i] + dt * gravity * ti.Vector([0.0, 1.0])
    old_x = x[t - 1, i]
    new_x = old_x + dt * old_v
    toi = 0.0
    new_v = old_v
    if new_x[1] < ground_height and old_v[1] < -1e-4:
      # The 1e-4 safe guard is important for numerical stability
      toi = -(old_x[1] - ground_height) / old_v[1] # Compute the time of impact
      new_v = ti.Vector([0.0, 0.0])

    # Note that with time of impact, dt is divided into two parts,
    #   the first part using old_v, and second part using new_v
    new_x = old_x + toi * old_v + (dt - toi) * new_v

    v[t, i] = new_v
    x[t, i] = new_x
```

In rigid body simulation, the implementation follows the same idea yet is slightly more complex. Please refer to `rigid_body.py` for more details.

## F ADDITIONAL TIPS ON GRADIENT BEHAVIORS

**Initialization matters: flat lands and local minima in physical processes** A trivial example of objective flat land is in `billiards`. Without proper initialization, gradient descent will make no progress since gradients are zero (Fig. 11). Also note the local minimum near $(-5, 0.03)$.

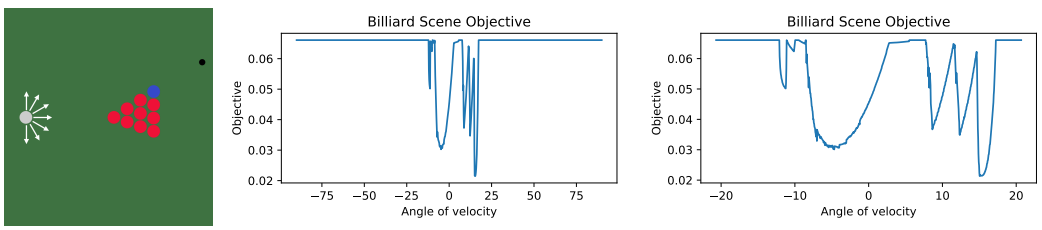



Figure 11: **Left:** Scanning initial velocity in the billiard example. **Middle:** Most initial angles yield a flat objective (final distance between the blue ball and black destination) of $0.065$, since the white ball does not collide with any other balls and imposes no effect on the pink ball via the chain reaction. **Right:** A zoomed-in view of the middle figure. The complex collisions lead to a lot of local minimums. [Reproduce: **python3 billiards.py 1.0/0.23**]

In `mass_spring` and `rigid_body`, once the robot falls down, gradient descent will quickly become trapped. A robot on the ground will make no further progress, no matter how it changes its controller. This leads to a more non-trivial local minimum and zero gradient case.

**Ideal physical models are only "ideal": discontinuities and singularities** Real-world macroscopic physical processes are usually continuous. However, building upon ideal physical models, even in the forward physical simulation results can contain discontinuities. For example, in a rigid body model with friction, changing the initial rotation of the box can lead to different corners hitting the ground first, and result in a discontinuity (Fig. 12). In `electric` and `mass_spring`, due to the $\frac{1}{r^2}$ and $\frac{1}{r}$ terms, when $r \to 0$, gradients can be very inaccurate due to numerical precision issues. Note that $d(1/r)/dr = -1/r^2$, and the gradient is more numerically problematic than the primal for a small $r$. Safeguarding $r$ is critically important for gradient stability.

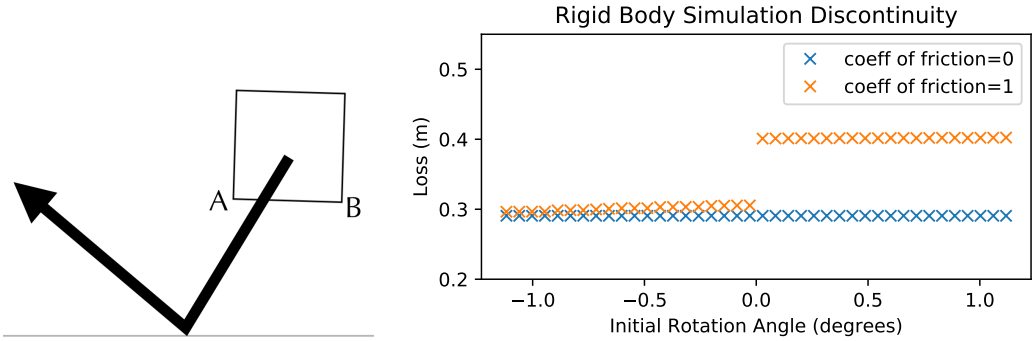



Figure 12: Friction in rigid body with collision is a common source of discontinuity. In this scene a rigid body hits the ground. Slightly rotating the rigid body changes which corner (A/B) hits the ground first, and different normal/friction impulses will be applied to the rigid body. This leads to a discontinuity in its final position (loss=final y coordinate). [Reproduce: **python3 rigid_body_discontinuity.py**] Please see our supplemental video for more details.