Systematically Differentiating Parametric Discontinuities

SAI PRAVEEN BANGARU*, MIT CSAIL
JESSE MICHEL*, MIT CSAIL
KEVIN MU, MIT CSAIL
GILBERT BERNSTEIN, UC Berkeley and MIT CSAIL
TZU-MAO LI, MIT CSAIL
JONATHAN RAGAN-KELLEY, MIT CSAIL

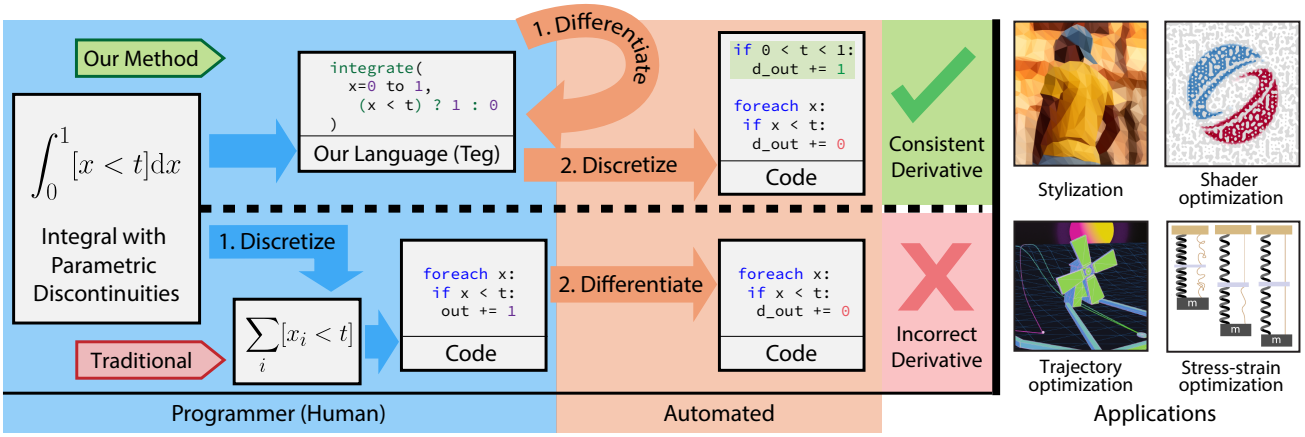


Fig. 1. We propose a language for the automatic differentiation of integrals with discontinuities. Existing auto-diff frameworks require integrals to be discretized into summations prior to differentiation, and therefore lose the derivative contribution from discontinuities. Our method produces a statistically consistent derivative program by introducing integration as a language primitive, which allows us to differentiate discontinuities in continuous space, before discretizing them into summations over discrete samples.

Emerging research in computer graphics, inverse problems, and machine learning requires us to differentiate and optimize parametric discontinuities. These discontinuities appear in object boundaries, occlusion, contact, and sudden change over time. In many domains, such as rendering and physics simulation, we differentiate the parameters of models that are expressed as integrals over discontinuous functions. Ignoring the discontinuities during differentiation often has a significant impact on the optimization process. Previous approaches either apply specialized hand-derived solutions, smooth out the discontinuities, or rely on incorrect automatic differentiation.

We propose a systematic approach to differentiating integrals with discontinuous integrands, by developing a new differentiable programming language. We introduce integration as a language primitive and account for the Dirac delta contribution from differentiating parametric discontinuities in the integrand. We formally define the language semantics and prove the correctness and closure under the differentiation, allowing the generation of gradients and higher-order derivatives. We also build a system, Teg, implementing these semantics. Our approach is widely applicable to a variety of tasks, including image stylization, fitting shader parameters, trajectory optimization, and optimizing physical designs.

CCS Concepts: • **Theory of computation** → Denotational semantics; • **Mathematics of computing** → **Differential calculus**; Stochastic control and optimization; *Probabilistic inference problems*; • **Computing methodologies** → **Computer graphics**; **Visibility**; **Animation**; Computer vision; **Modeling and simulation**.

Additional Key Words and Phrases: Automatic differentiation, differentiable programming, differentiable graphics, differentiable rendering, differentiable physics, domain-specific language.

ACM Reference Format:
Sai Praveen Bangaru, Jesse Michel, Kevin Mu, Gilbert Bernstein, Tzu-Mao Li, and Jonathan Ragan-Kelley. 2021. Systematically Differentiating Parametric Discontinuities. *ACM Trans. Graph.* 40, 4, Article 107 (August 2021), 17 pages. https://doi.org/10.1145/3450626.3459775

*Both authors contributed equally to this research.

Authors' addresses: Sai Praveen Bangaru, MIT CSAIL, Cambridge, MA, sbangaru@mit.edu; Jesse Michel, MIT CSAIL, Cambridge, MA, jmmichel@mit.edu; Kevin Mu, MIT CSAIL, Cambridge, MA, kmu@csail.mit.edu; Gilbert Bernstein, UC Berkeley, Berkeley, CA, MIT CSAIL, Cambridge, MA, gilbo@berkeley.edu; Tzu-Mao Li, MIT CSAIL, Cambridge, MA, tzumao@mit.edu; Jonathan Ragan-Kelley, MIT CSAIL, Cambridge, MA, jrk@csail.mit.edu.

0730-0301/2021/8-ART107
https://doi.org/10.1145/3450626.3459775

1 INTRODUCTION

Automatic differentiation, now indispensable for optimization and machine learning, usually treats discontinuities (e.g., if-else branches) by ignoring them. Doing so is usually correct almost everywhere (in the technical sense). However, the situation is different when computing the derivative of an integral, when discontinuities of the integrand cannot be ignored. We propose a differentiable programming language that can soundly compute derivatives of integrals with discontinuities, producing correct results almost everywhere.

Parametric discontinuities in graphics. Emerging research that combines techniques in computer graphics, inverse problems, and machine learning often requires optimizing discontinuities. We often want to compute the derivatives of integrals of discontinuous functions:

$$\nabla \int f$$

where f is a program that can have parametric discontinuities. Parametric discontinuities are branching expressions containing free parameters (e.g., t in `foreach x: ((x < t) ? 1 : 0)`). The derivative contribution of a parametric discontinuity is a Dirac delta function and usually integrates to a non-zero value. In graphics, these parametric discontinuities arise in rigid-body simulations that exhibit collision and contact phenomena; rendered shapes form silhouettes and shadows; geometry has corners and creases. Both integration and differentiation play critical roles in simulating, rendering, and optimizing these models.

For instance, the variational principle of least action defines physical trajectories as minima of an action integral; which may be applied to synthesizing animations and optimizing robot controllers [Stengel 1994; Witkin and Kass 1988]; rendering is defined as a multi-dimensional integral, and hence inverse rendering involves differentiating that integral [Li et al. 2018a]; and shape optimization uses derivatives of integrated geometric quantities [Hafner et al. 2019]. Combining deep learning models with these problems is only possible if we can compute derivatives through the discontinuities.

Prior work has recognized that ignoring the discontinuities during differentiation can have a significant impact on the optimization. Oftentimes, domain-specific solutions are employed to manually derive the correct derivatives (e.g., [Dyer and McReynolds 1968; Li et al. 2018a]). When adapting to new problems, new derivations are required. Our goal is to instead systematize the domain-specific methods, and develop the foundation of a programming language.

Using our approach and language, we study applications spanning several fields in graphics. We write a 2D differentiable renderer for triangulating images, fit a procedural shader's parameters to a target image, solve frictional contact problems without smoothing, and optimize physical designs with discontinuous properties.

Minimal example. Ignoring discontinuities during differentiation is problematic. Consider the example from Fig. 1: $\frac{\mathrm{d}}{\mathrm{d}t}\int_{x=0}^{1}[x < t]$, where $[x < t] = 1$ if $x < t$ and 0 otherwise. This integral might represent the expected frequency of a t-biased coin flip coming up heads; or the fraction of a pixel covered by a triangle with position t; or the fraction of a time-interval during which a motor is engaged, when it is turned off at time t. If we wish to optimize this t parameter, then we need to take some such derivative.

It is common-place to first *discretize* integrals and only after that to *differentiate* them using automatic differentiation. However, existing auto-diff systems define $\frac{d}{dt}[x < t] = 0$. This is unfortunately incorrect in the presence of integration. The expected frequency of a t-biased coin coming up heads does change with the bias t. This is because the derivative of the step function $[x < t]$ is a *Dirac delta* $\delta(t - x)$, which integrates to a non-zero value when both sides of the branch are visited in the integration domain. In general, the discretization of the differential of an integral is different than the differential of the discretization of an integral (Fig. 1). Ignoring the Dirac delta often leads to suboptimal results – in this example, the optimization will never move from the initial guess.

A systematic solution. We explore a systematic method for optimizing parametric discontinuities, by specifying the semantics of a new differentiable programming language that is provably correct and can generate higher-order derivatives. Our key idea is to differentiate then discretize in our language. By making integration a language primitive and accounting for the Dirac deltas introduced by differentiating parametric discontinuities, our language computes the correct solution:

$$\frac{\mathrm{d}}{\mathrm{d}t}\int_{x=0}^{1}[x < t] \to \int_{x=0}^{1}\delta(t - x) \to [0 < t < 1].$$

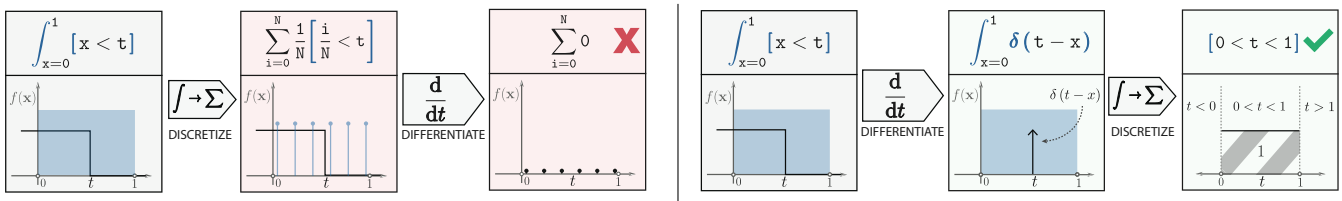
Analytically eliminating Dirac deltas via an enclosing integral – the second step above – is difficult to systematically guarantee for arbitrary expressions inside Dirac deltas. We guarantee the correct treatment of Dirac deltas under suitable conditions. We require that the expression constituting each of the parametric discontinuities be represented only using differentiable, invertible functions (i.e., *diffeomorphisms*). We allow such functions to be explicitly defined by the programmer, or automatically inferred for certain common expression classes such as affine expressions.

In this paper, we prioritize the development of a clean core calculus that captures the interactions among derivatives, integrals, and discontinuities. Our language only supports static loop variables (it is not Turing complete). The symbolic transformation used to eliminate the Dirac deltas are global and may lead to an asymptotic increase in expression size, scaling with the number of discontinuities. We provide a proof of correctness for the language and show that it is closed under the derivative operation, allowing for immediate application of the technique to higher-order derivatives. Finally, we build a system, Teg, that implements the semantics.[1]

In summary, our contributions are:

- We propose a systematic approach to differentiate integrals with discontinuous integrands, by including integration as a language primitive. We define the formal semantics, and prove its correctness and closure under differentiation.
- We implement these semantics in a differentiable programming language, Teg, which supports both forward and reverse-mode automatic differentiation.
- We show novel applications spanning fields in computer graphics. They include a method for fitting parametric discontinuities in inverse shader design, and a frictional contact solver that does not rely on smoothing energy.

[1]The source code of the compiler and application is available at https://github.com/ChezJrk/Teg and https://github.com/ChezJrk/teg_applications.



(a) traditional automatic differentiation (Discretize-then-Differentiate)

(b) our automatic differentiation (Differentiate-then-Discretize)

Fig. 2. 1D Example. Our language, Teg, can automatically differentiate programs with integrals and discontinuities, represented as indicator functions [ ]. (a) Traditional automatic differentiation approaches first discretize then differentiate because they do not have an integral primitive. This leads to incorrect results because it ignores the contribution of the Dirac deltas introduced by differentiating the discontinuities. (b) In contrast, we include integral primitive in our language and account for the contribution of the Dirac deltas. Teg first differentiates then discretizes. In this case, we apply the identity $\int_{x=0}^{1} \delta(\theta - x) = \int_{y=\theta-1}^{\theta} \delta(y) = [\theta - 1 < 0][\theta > 0]$.

To motivate our approach, we describe a few problems and how others have solved them in the past. We only introduce necessary background in the next section and detail additional related work in the section that follows (Sec. 3).

2 MOTIVATION AND BACKGROUND

We motivate the value of optimizing parametric discontinuities with examples. We describe them in our language, Teg, and show how the compiler produces correct code. Formal semantics and the proof of correctness are detailed in Sec. 4.

Notation and frontend. Teg is designed to mirror mathematical equations as closely as possible. Throughout the paper, when we show the code in our language, we use syntax highlighting and a different font to distinguish from the mathematical formulae. For example, a mathematical formula of an integral $\int_{x=0}^{1}[x < 5]x^2$, where $[x < 5] = 1$ if $x < 5$ else 0[2], is expressed as $\int_{x=0}^{1}[\,x < 5\,]x^2$ in our language. We write this code in Teg's Python frontend library as `Teg(0, 1, IfElse(x < 5, 1, 0) * x * x, x)`.

2.1 1D Example

We now more deeply explore the 1D integral parameterized by t from the introduction and Fig. 1:

$$I(t) = \int_{x=0}^{1} [x < t], \tag{1}$$

Many graphics problems are more complex manifestations of this form, such as anti-aliasing [Mitchell and Netravali 1988], global illumination [Kajiya 1986], integration of certain ordinary differential equations, or simulating physics using the variational principle [Hamilton 1834]. The indicator function appears at, for example, object boundaries, occlusion, or sudden change of force over time.

Our goal is to automatically compute the derivative $\frac{dI}{dt}$. We want to use the derivative for optimization, machine learning, or describing physical quantities. Later we show more realistic applications, including differentiable rendering and physics simulation. In this 1D case, we have a closed-form solution: $\frac{dI}{dt} = [0 < t < 1]$.

[2]This "Iversion bracket" syntax comes from APL [Iverson 1962] and was advocated by Donald Knuth [1992].

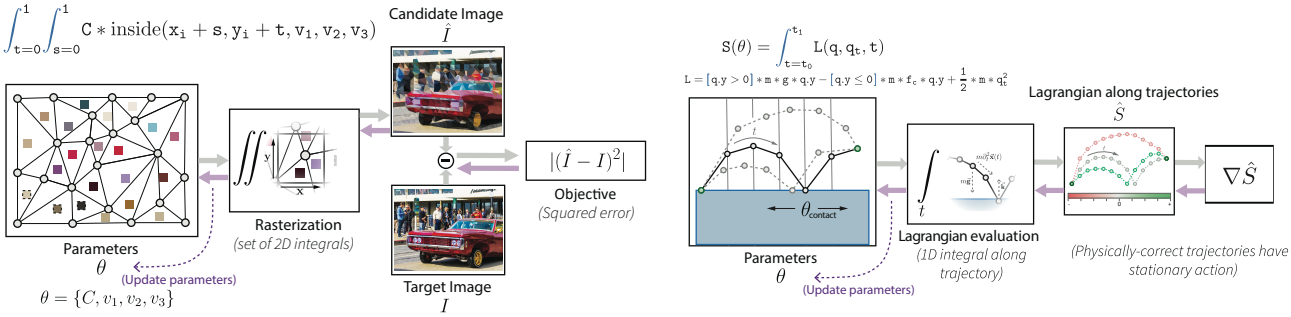
Discretize-then-Differentiate produces incorrect derivatives. Solutions to real-world problems often lack closed-form solutions. Computing derivatives by hand takes up significant time. Therefore, we want to rely on automatic differentiation [Griewank and Walther 2008]. A typical compiler does not have integrals as a primitive. Therefore, to use automatic differentiation, we must first manually discretize the integral into a program. For example, $I(t) \approx \sum_{i=0}^{N}[i/N < t] = I_d(t)$. However, differentiating the program I_d with respect to t gives the incorrect derivative of 0, contradicting our closed-form solution $[0 < t < 1]$. This is because the only dependency of the program to t is through the indicator $[i/N < t]$. The derivative of the indicator gives rise to a *Dirac delta* δ, which describes an infinitesimal size spike at $x/N = t$, but has value zero everywhere else. A delta can only be realized into a number through integration. Since there are no integral primitives in existing automatic differentiation tools, all of them ignore the delta. Fig. 2 shows a visual explanation.

The issue above is not first observed by us, and other researchers proposed alternative solutions. In particular, our approach is closely related to the recent advancement of differentiable rendering [Li et al. 2018a; Ramamoorthi et al. 2007]. Dyer and McReynolds [1968] also derived the derivatives in the optimal control context. These works recognized that we could explicitly account for the Dirac deltas by evaluating the integrals over them. We systematize these approaches and incorporate them into a programming language.

Our approach: Differentiate-then-Discretize. The key idea is to first differentiate then discretize, all inside the language. Similar to previous differentiable rendering works [Li et al. 2018a; Ramamoorthi et al. 2007], we base our compiler passes on the following properties of indicator functions, Dirac deltas δ, and integrals:

$$\begin{aligned} &\frac{\partial}{\partial t}[c(x,t) > 0] = \delta(c(x,t)) * \frac{\partial}{\partial t}c(x,t) \\ &\int_{x=a}^{b} f(R(x))\frac{\partial R(x)}{\partial x} = \int_{u=R(a)}^{R(b)} f(u) \\ &\int_{x=a}^{b} \delta(x) = [a < 0 < b], \end{aligned} \tag{2}$$

where $u = R(x)$ is a variable substitution, or reparametrization. Given an indicator $[c(x,t) > 0]$ under differentiation, we first turn



(a) 2D differentiable rendering

(b) trajectory optimization

Fig. 3. Case studies. Our language can be used for computing many applications where one needs to differentiate integrals with discontinuous integrands. In differentiable rendering (a), each pixel is a 2D anti-aliasing integral. The geometry boundaries introduce discontinuities. The figure shows an example where we fit a triangle mesh with constant color within triangles to a target image using the gradient of loss with respect to the triangle parameters. In physics simulation (b), the variational least action principle predicts motion by minimizing an action integral of the Lagrangian energy over time. The Lagrangian energy can contain discontinuities due to contact, sudden change of force over time, or other physical properties. We can formulate a trajectory optimization problem by finding the path parameters of a bouncing ball by minimizing the action integral. Unlike standard approaches that integrate over ordinary differential equations, we do not need to choose a step size, and the contact point between the ball and the floor can be anywhere. Sec. 6 shows more examples.

it into a delta $\delta(c(x,t))$. We then apply reparametrizations to transform the delta into the form $\delta(x)$. Finally, we eliminate the delta using the last rule above.

Therefore, for a Teg program $I = \int_{x=0}^{x=1} [\,x < t\,]$, the compiler differentiates I by applying the following transformation:

$$\frac{d}{dt}\int_{x=0}^{1}[\,x<t\,] \rightarrow \frac{d}{dt}\int_{x=0}^{1}[\,t-x>0\,] \rightarrow \int_{x=0}^{1}\frac{d}{dt}[\,t-x>0\,]$$
$$\int_{x=0}^{1}\delta(t-x) \rightarrow \int_{u=t-1}^{t}\delta(u) \rightarrow [\,t-1<0\,][\,t>0\,].$$

The first step normalizes the condition into the form $a > 0$. The second step moves the derivative operator inside the integral. We then transform the derivative of indicator functions into a Dirac delta. Next, we apply the reparametrization $u = t - x$. Finally, we eliminate the delta, leading us to the same expression to our closed-form solution $[0 < t < 1]$. Sometimes the final expression can be another integral. Teg evaluates delta-free integral expressions by discretizing them.

Not all deltas $\delta(c(x,t))$ can be easily eliminated, since they must first be reparametrized to the form $\delta(x)$ so that they may then annihilate with an integral. To reparametrize (second rule in Eq. (2)), we need to find the inverse $x = c^{-1}(u,t)$ in order to substitute the variables outside the delta. Automatically deriving such inversion is hard. Our compiler automatically handles an important case when c is affine. In 1D, it takes the form $t_0 x + t_1$, where t_0 and t_1 can depend on other parameters that are not x. For other conditions, we provide a library of diffeomorphisms for users to apply, e.g., a polar-to-Cartesian transformation. Crucially, we allow users to define custom diffeomorphisms (bijectivity can be checked with numerical tests). Custom mappings make our language flexible, while also providing a correctness guarantee by decoupling the derivative transformation from the reparametrization.

In Sec. 4, we formally describe our language, compiler transformation passes, and correctness theorem, dealing with multiple conditions, integrals. We also make sure that our language is closed under differentiation, that is, the differentiated code is still a program expressible in our language. The closure property is crucial for higher-order derivatives. Before that, let us introduce two graphics applications that involve the differentiation of integrals.

2.2 Case Study I: 2D Differentiable Rendering

Rendering computes an image from a given set of geometric shapes and their shading color. Differentiable rendering [Li et al. 2018a; Loper and Black 2014], on the other hand, computes the gradient of pixel color with respect to scene parameters for inverse rendering or machine learning. Previous approaches either apply approximation [Loper and Black 2014], manually derive the gradient [Li et al. 2018a], or use finite differences [Lawonn and Günther 2019]. Here we show how Teg automatically differentiates a 2D rasterization program, outlined in Figure 3a.

Here we study a simplified rendering model, where the geometry is represented as 2D triangles, and the color inside the triangle is constant. We want to fit the geometry and the color to a target image to produce a stylized image [Lawonn and Günther 2019]. For a pixel at (x_i, y_i) and for a triangle with color C and vertices v_1, v_2, v_3, the equation representing the color at that pixel is:

$$I_r = \int_{t=0}^{1}\int_{s=0}^{1} C * \text{inside}(x_i + s, y_i + t, v_1, v_2, v_3), \tag{3}$$

where, for convenience, we define the function inside as the multiplication of three edge equations:

$$\begin{aligned}&\text{inside}(x, y, v_1, v_2, v_3)\\ =&[\,(v_1.y - v_2.y) * x + (v_2.x - v_1.x) * y > v_1.x * v_2.y - v_2.x * v_1.y\,]\\ *&[\,(v_2.y - v_3.y) * x + (v_3.x - v_2.x) * y > v_2.x * v_3.y - v_3.x * v_2.y\,]\\ *&[\,(v_3.y - v_1.y) * x + (v_1.x - v_3.x) * y > v_3.x * v_1.y - v_1.x * v_3.y\,]\end{aligned}$$

The value at each pixel is a sum of the contribution of all of the overlapping triangles.

We want to compute the derivative of the pixel color integral (Program (3)) with respect to the vertices v. Therefore, Teg needs to handle the deltas that arise from differentiating the inside function. Using the same techniques as in the previous subsection, Teg automatically generates the derivative, which are three 1D integrals over the three edges of the parameterized triangle. The main difference is that we now transform multiple integrals. Consider the following double integral with an affine condition:

$$\mathtt{I_2} = \int_{\mathtt{y}=0}^{1}\int_{\mathtt{x}=0}^{1} [\,\mathtt{a}*\mathtt{x}+\mathtt{b}*\mathtt{y}+\mathtt{c}>0\,], \tag{4}$$

and we want to compute the derivative $\frac{\partial \mathtt{I_2}}{\partial \mathtt{c}}$. After transforming the indicator into a Dirac delta during differentiation, our compiler automatically detects the affine condition pattern $\mathtt{a}*\mathtt{x}+\mathtt{b}*\mathtt{y}+\mathtt{c}$ and applies a bijective and rotational reparametrization $\mathtt{x}' = \mathtt{a}*\mathtt{x}+\mathtt{b}*\mathtt{y}$, $\mathtt{y}' = \mathtt{b}*\mathtt{x}-\mathtt{a}*\mathtt{y}$. The compiler generates similar reparametrizations for higher-dimensional spaces. The resulting derivative integral is

$$\frac{\partial \mathtt{I_2}}{\partial \mathtt{c}} = \int_{\mathtt{y}'=\mathtt{B}_\mathtt{y}^\mathtt{l}}^{\mathtt{B}_\mathtt{y}^\mathtt{u}}\int_{\mathtt{x}'=\mathtt{B}_\mathtt{x}^\mathtt{l}}^{\mathtt{B}_\mathtt{x}^\mathtt{u}} \frac{\boldsymbol{\delta}(\mathtt{x}'+\mathtt{c})}{\mathtt{a}^2+\mathtt{b}^2}, \tag{5}$$

where $\mathtt{B}_\mathtt{x}^\mathtt{l}, \mathtt{B}_\mathtt{x}^\mathtt{u}, \mathtt{B}_\mathtt{y}^\mathtt{l}$, and $\mathtt{B}_\mathtt{y}^\mathtt{u}$ are new integral bounds derived from the reparametrization, and $\mathtt{a}^2+\mathtt{b}^2$ is the Jacobian determinant of the transformation. From here, the compiler reparametrizes $\boldsymbol{\delta}(\mathtt{x}'+\mathtt{c})$ into $\boldsymbol{\delta}(\mathtt{x}'')$ by applying $x'' = x' + c$, and eliminates the delta and corresponding integral over x'', leaving a 1D integral. In the triangle case, the same pattern applies with a, b, c being parameters coming from the triangle vertices.

The derivative $\frac{\partial I_r}{\partial v}$ generated by Teg can then be composed with derivatives generated by other automatic differentiation systems such as PyTorch [Paszke et al. 2019] or TensorFlow [Abadi et al. 2015]. Given a loss function $L(I_r)$, we can use the chain rule $\frac{\partial L}{\partial v} = \frac{\partial L}{\partial I_r}\frac{\partial I_r}{\partial v}$. Teg computes $\frac{\partial I_r}{\partial v}$, while other automatic differentiation systems can be used to compute $\frac{\partial L}{\partial I_r}$.

Existing differentiable rendering methods. Recent work has developed different techniques to differentiate the rendering computation. The derivation above is similar to Li et al. [2018a], and is closely related to the Reynolds transport theorem [Li 2019; Li et al. 2020b; Zhang et al. 2020, 2019]. An alternative approach is to apply reparametrizations to remove the discontinuities [Loubet et al. 2019], which turns out to be equivalent to applying divergence theorem on the derivative line integrals [Bangaru et al. 2020]. Other methods approximate the Dirac deltas by using image filters [Loper and Black 2014], or smoothing over discontinuities [Liu et al. 2019].

Teg mechanizes the Dirac delta computation as part of automatic differentiation, which allows us to generalize the approach. For example, in Sec. 6 we show results on optimizing a color model with linear or quadratic interpolation, or even optimizing a texture generated by Perlin noise [Perlin 1985]. We can also apply it to problems outside of differentiable rendering, as we will show next. However, our compiler currently does not support other strategies for dealing with discontinuities, such as smoothing or transforming boundary integrals into area integrals using divergence theorem, nor does it currently scale to millions of objects (Sec. 7).

2.3 Case Study II: Physics-based Animation and Control

Composing derivatives and integrals to form an optimization problem is a common pattern in physics. For didactic purposes, we discuss one of the simplest possible problems: finding the trajectory of a bouncing ball (Fig. 3). We use a piecewise linear model of the trajectory, with parameters θ_b. The variational least action principle [Hamilton 1834] predicts motion by finding the stationary points the following action integral over time:

$$\mathtt{S}(\theta_\mathtt{b}) = \int_{\mathtt{t}=\mathtt{t}_0}^{\mathtt{t}_1} \mathtt{L}(\mathtt{q}, \mathtt{q}_\mathtt{t}, \mathtt{t}), \tag{6}$$

where $\mathtt{q}$ is the 2D position of the ball (parameterized by $\theta_\mathtt{b}$), $\mathtt{q}_\mathtt{t}$ is its time derivative, and $\mathtt{t}$ is time. $\mathtt{L}$ is the Lagrangian of the physics system. In this case, it is composed of the potential energy $-\mathtt{V}$ (whose spatial derivative $-\frac{\partial \mathtt{V}}{\partial \mathtt{q}}$ is the force) and the kinetic energy $\frac{1}{2}\mathtt{m}\mathtt{q}_\mathtt{t}^2$. Intuitively, we find the path parameter θ_b that minimizes a cost function S resulting in a stationary action.

In the case of the bouncing ball, the energy of the system is the gravitation potential when $q.y > 0$, and the contact force f_c pushes the ball back when $q.y \le 0$:

$$\mathtt{L}(\mathtt{q}, \mathtt{q}_\mathtt{t}, \mathtt{t}) = \frac{1}{2}\mathtt{m}*\mathtt{q}_\mathtt{t}^2 - ([\,\mathtt{q.y}>0\,]*\mathtt{m}*\mathtt{g}*\mathtt{q.y} - [\,\mathtt{q.y}\le 0\,]*\mathtt{m}*\mathtt{f_c}*\mathtt{q.y}).$$

For our piecewise linear model of the trajectory q, we consider a sequence of control points, *at varying points in time* rather than at regular temporal intervals. Doing so allows our trajectory model to represent a sudden change in velocity at some arbitrary point and time, which is necessary to exactly capture our idealized inelastic collision. This *direct collocation* trajectory optimization approach [Hargraves and Paris 1987] is different from standard approaches that differentiate forward simulators, usually called the *shooting method*. The standard shooting method requires pre-determining time step sizes for integration, leading to issues when the contact point can be anywhere. Trajectory optimization is used for designing animation or in robotics for planning [Betts 2009; Popović et al. 2000; Witkin and Kass 1988]. In computer graphics, trajectory optimization with contact is usually handled by using a smooth contact model in the first place [Hunt and Crossley 1975], or incorporate inequality constraints and then solve it by smoothing the discontinuities [Mordatch et al. 2012; Todorov 2011], or using sophisticated nonlinear programming solvers [Posa et al. 2014].

In Teg, we can define and differentiate physics problems with discontinuous energy, enabling optimization using gradients or higher-order derivatives. Notice that the derivatives appear both inside the integral ($\mathtt{q}_\mathtt{t}$) and outside (though this can be problematic theoretically, see Sec. 7.2). This physics formulation extends beyond contact modeling. We can model forces that change suddenly over time, and we can model an elastic potential energy $\mathtt{k}\mathtt{q}_\mathtt{t}^2$, where the coefficient k depends on the length of the spring. In Sec. 6, we present additional applications in more detail.

3 RELATED WORK

We now provide more detailed references for related applications, systems, and languages.

Rendering and differentiable rendering. Rendering, physics-based or not, involves solving anti-aliasing integrals [Mitchell and Netravali 1988]. Physics-based rendering further formulates the light transport as an integral equation [Kajiya 1986; Veach 1998]. Analytical derivatives have been derived for diffuse interreflection [Arvo 1994], shadow [Ramamoorthi et al. 2007], and spherical harmonics lighting [Wu et al. 2020]. Recently, there are strong interests in graphics, vision, and machine learning communities to build fully differentiable renderers. Some methods ignore geometry derivatives [Gkioulekas et al. 2013; Nimier-David et al. 2019], some approximate the Dirac delta contributions [de La Gorce et al. 2011; Kato et al. 2018; Loper and Black 2014], some smooth out the discontinuities [Liu et al. 2019], and some apply smooth postprocessing using the geometry buffer [Laine et al. 2020]. Meanwhile, other methods derive the correct derivatives using Dirac deltas [Li et al. 2018a], reparametrization [Loubet et al. 2019], or Reynolds transport theorem [Bangaru et al. 2020; Li 2019; Li et al. 2020b; Roger et al. 2005; Zhang et al. 2020, 2019]. Differentiable renderers have also been used for inverse shader designs [Guo et al. 2020; Shi et al. 2020]. Our language lays the foundation for a programmable differentiable rendering system. We have not yet included rendering-specific abstractions and data structures, which are necessary for applying Teg to large-scale rendering problems.

Variational physics, control, and shape modelling. Calculus of variations – the minimization of integrals over cost functionals – can often predict motion. A prominent example is Hamilton's least action principle [Hamilton 1834]. In graphics, robotics, and optimal control, this formulation has often been used for interpolating animation or planning [Barr et al. 1992; Betts 1998, 2009; Cohen 1992; Popović and Witkin 1999; Witkin and Kass 1988]. The same principle can be used in 2D or higher-dimensional space to find the surface or volume with least cost [Moreton and Séquin 1992; Welch and Witkin 1992], or finding contours in images [Kass et al. 1988]. In trajectory optimization, fitting the trajectory generated by ordinary differential equations is often called the (multiple) shooting method [Geilinger et al. 2020; McNamara et al. 2004; Popović et al. 2000; Twigg and James 2008]. On the other hand, methods that fit trajectory as splines are called the direct collocation method [Hargraves and Paris 1987]. A recent line of work aims to combine neural network controllers with simulators [de Avila Belbute-Peres et al. 2018; Holl et al. 2020; Hu et al. 2020; Um et al. 2020], or directly model system dynamics using neural networks [Chen et al. 2018].

A common strategy to handle object contact is to introduce constraints and solve nonlinear programming problems [Betts 2009; Li et al. 2020a; Mordatch et al. 2012, 2013; Posa et al. 2014]. It is also possible to approximate contact using impulse-based physics [Hu et al. 2020; Mirtich 1996]. We explore an alternative formulation to this problem, by allowing certain discontinuous cost functions to be used in optimization. The mathematics of such formulations has been studied [Dyer and McReynolds 1968]. We further show the connection to differentiable rendering.

Domain-specific languages for computer graphics. Several graphics systems handle integration and differentiation: CONDOR [Kass 1992] proposed an interactive programming interface for solving constrained dynamics with automatic differentiation and interval arithmetic. Aether [Anderson et al. 2017] models Monte Carlo integration, and automatically computes the Jacobian of integral reparametrization. Recent graphics systems often include differentiation operations [Devito et al. 2017; Hu et al. 2020; Jakob 2019; Li et al. 2018b; Nimier-David et al. 2019]. In contrast, we study the interaction between integration, differentiation, and discontinuities.

Automatic differentiation and probabilistic programming. It is possible to automate derivative calculation by applying the chain rule to program expressions [Griewank and Walther 2008; Wengert 1964]. Reverse-mode automatic differentiation [Linnainmaa 1970] derives gradients that can be evaluated at the same time complexity as the original program – much faster than finite differences. There is a revitalized interest in revisiting efficient auto-diff systems in deep learning [Abadi et al. 2015; Bradbury et al. 2018; Paszke et al. 2019; Yu et al. 2014] and probabilistic programming [Bingham et al. 2019; Stan Development Team 2015]. Auto-diff has also gained significant attention in programming languages, where researchers specify system semantics and prove the correctness of their systems [Elliott 2018; Lee et al. 2020; Mazza and Pagani 2021; Pearlmutter and Siskind 2008; Sherman et al. 2021]. We study the particular interaction among the derivative, integral, and discontinuities.[3]

Inala et al. [2018] relax boolean expressions into real numbers through smoothing, and solve numerical problems by combining gradient descent and satisfiability solvers. We instead exploit the structure of integration.

Integration and differentiation are indispensable to Bayesian inference and probabilistic programming [Gehr et al. 2020; Kucukelbir et al. 2015; Lew et al. 2019]. Many probabilistic programming languages handle integrals, but most do not explicitly handle the differentiation of discontinuities. LFPPL [Zhou et al. 2019] studies discontinuities in the context of Hamiltonian Monte Carlo methods. Closest to our work is Lee et al.'s stochastic variational inference work [2018]. They observed when applying the reparametrization trick to non-differentiable models, existing approaches ignore the deltas. They design a language for stochastic variational inference. The language focused on infinite domain integrals with affine discontinuities. We show a more general language with correctness proof, closure, and bounded domain, and generalize the class of discontinuities through diffeomorphisms. Importantly, we show wide applicability of this class of approaches to many graphics problems.

4 SEMANTICS AND CORRECTNESS PROOF OF OUR LANGUAGE

We have motivated and discussed our system Teg. In this section, we formally define a simplified core language $\mathcal{L}$, and prove the correctness of the compiler passes and closure under differentiation.

Our formal semantics focus on forward-mode automatic differentiation. The derivation of reverse-mode is similar to forward-mode, but it requires adding let bindings (intermediate variables) to the core language. We found this to be distracting for the minimal language for our proof. Our practical implementation supports both forward-mode and reverse-mode. We focus on the scalar case in this work. Arrays and tensors are unrolled into scalars in our language.

[3] Sherman et al. [2021] includes an integration primitive and produces a correct, but vacuous result in the presence of discontinuities.

4.1 Preliminaries

A context σ is a partial function from variable names to values, other variable names, or expressions. The empty context $[]$ is undefined for all names ($[](x) = \bot$). A context may be extended $\sigma[x \mapsto v]$, s.t. $\sigma[x \mapsto v](x) = v$ and $\sigma[x \mapsto v](y) = \sigma(y)$ when $x \neq y$. The singleton context $[x \mapsto v]$ is shorthand for an extension of the empty context. $x \in \sigma$ is true when $\sigma(x) \neq \bot$ and $x \notin \sigma$ when $\sigma(x) = \bot$.

We will use contexts passed into mathematical functions as a way of specifying argument bindings without positions (e.g., the value of x is 3, of y is 42, etc.). We will also use contexts by directly applying them to expressions in order to perform variable substitution (e.g. $[x \mapsto y](2 * x + z) = 2 * y + z$).

We will use the notation $\vec{x}$ as shorthand for a finite list of variables $x_1, x_2, \ldots$. The expression $y \in \vec{x}$ means that y is a symbol occuring in that shorthand list.

4.2 Syntax and Denotational Semantics of $\mathcal{L}$

We first present a minimal language $\mathcal{L}$ that demonstrates many of the key features of Teg. We discuss later how to extend the minimal language into our full language Teg. Below is the syntax written in Backus–Naur form:

$$e ::= c \mid x \mid e_1 + e_2 \mid e_1 * e_2 \mid \int_{x=a}^{b} e \mid \left[\phi(\vec{x}) > 0\right] \mid f(e)$$

(i.e., an expression is a constant c, or a variable x, etc.) The denotational semantics describes the behavior of these syntactic forms in terms of mathematical functions. That is, $E[[e]]$ is a function mapping from contexts (holding variable values) to a real number.

$$\begin{aligned}
E[[c]] &= c \\
E[[x]] &= x \\
E[[e_1 + e_2]] &= E[[e_1]] + E[[e_2]] \\
E[[e_1 * e_2]] &= E[[e_1]] * E[[e_2]] \\
E[[\int_{x=a}^{b} e]] &= \int_{x=E[[a]]}^{E[[b]]} E[[e]] \\
E[[\left[\phi(\vec{x}) > 0\right]]] &= [\widehat{\phi_1}(\vec{x}) > 0] \\
E[[f(e)]] &= f(E[[e]])
\end{aligned}$$

In the indicator function $\left[\phi(\vec{x}) > 0\right]$, the vector $\vec{x}$ are the free variables, and ϕ are *invertible* functions drawn from a class of functions that we will discuss later (§4.5). For now, simply note that $\widehat{\phi_1}$ is the component of ϕ that is branched on. We will generally suppress the specification of free variables $\vec{x}$. The bounds expressions a and b are expressions in $\mathcal{L}$ that do not include integrals or indicator functions, and must be independent of variables of integration. In contrast, arguments to invertible functions $\vec{x}$ inside of indicator functions may depend on variables of integration.

The term f corresponds to a built-in function, and f is the mathematical function denoted. We require that f is smooth and that the derivatives are syntactically defined. For example, cosine may be defined as the denotation $E[[\cos(e)]] = \cos(E[[e]])$, and the derivatives are corecursively defined using sin.

4.3 Syntactic Sugar

We briefly address convenient extensions to the minimal language $\mathcal{L}$. Via the built-in mechanism $f(\cdot)$, we add support for division, and other common mathematical functions. More general comparisons $\left[\phi(\vec{x}) > \psi(\vec{x})\right]$ can be reduced to the canonical formulation $\left[\phi(\vec{x}) - \psi(\vec{x}) > 0\right]$. Conjunctions of conditions can be encoded as products of indicator functions, and disjunctions can be represented using the inclusion-exclusion principle.

As presented, $\mathcal{L}$ does not have a way of binding intermediate variables. Our prototype includes let-bindings, although certain parts of derivation (Sec. 4.4) may require inlining these bindings—which can lead to poor compile time in some cases.

4.4 Derivative Application

We define the source-to-source derivative application in terms of the derivative transformation $D[[e]]\sigma$, where the context σ specifies a mapping from variables to be differentiated to their corresponding differential variable names (e.g., $[x \mapsto dx]$).

Taking the derivative of indicator functions produces Dirac deltas, a construct not accounted for in our language $\mathcal{L}$. Therefore, we extend the language to $\mathcal{L}'$ as follows:

$$e' ::= e \mid e' + e' \mid e' * e \mid \int_{x=a}^{b} e' \mid \boldsymbol{\delta}\left(\phi(\vec{x})\right)$$

Note that besides including the Dirac delta $\boldsymbol{\delta}\left(\phi(\vec{x})\right)$, this extension $\mathcal{L}'$ ensures that the resulting expressions are linear in the delta terms, and will also be linear in any differential terms.

Derivative application $D[[e]]\sigma$ follows two steps:

- A lifting derivative, which takes in a program in $\mathcal{L}$ and outputs a program in $\mathcal{L}'$ that may include Dirac deltas.
- Delta elimination takes in a program in $\mathcal{L}'$ and outputs a program in $\mathcal{L}$ that is delta-free. It achieves this by having Dirac deltas either annihilate with the appropriate integrals or be set to zero if their contribution is of measure zero.

We now detail these steps in the derivative application.

4.4.1 Lifting: Forward Derivative. The lifting derivative $F[[\cdot]]\sigma : \mathcal{L} \to \mathcal{L}'$ maps from an expression in the (external) language $\mathcal{L}$ to the derivative expression represented in the internal language $\mathcal{L}'$.

$$\begin{aligned}
F[[x]]\sigma &= dx, \text{ if } x \in \sigma \\
F[[x]]\sigma &= 0, \text{ if } x \notin \sigma \\
F[[c]]\sigma &= 0 \\
F[[e_1 + e_2]]\sigma &= F[[e_1]]\sigma + F[[e_2]]\sigma \\
F[[e_1 * e_2]]\sigma &= F[[e_1]]\sigma * e_2 + e_1 * F[[e_2]]\sigma \\
F[[\int_{x=a}^{b} e]]\sigma &= \int_{x=a}^{b} F[[e]]\sigma \\
&\quad + F[[b]]\sigma * [x \mapsto b]e - F[[a]]\sigma * [x \mapsto a]e \\
F[[\left[\phi(\vec{x}) > 0\right]]]\sigma &= 0 \text{ if } \forall y \in \vec{x}, y \notin \sigma \\
F[[\left[\phi(\vec{x}) > 0\right]]]\sigma &= \boldsymbol{\delta}\left(\phi(\vec{x})\right) * F[[\phi(\vec{x})]]\sigma \text{ if } \exists y \in \vec{x}, \text{ s.t. } y \in \sigma \\
F[[f(e)]]\sigma &= df(e) * F[[e]]\sigma
\end{aligned}$$

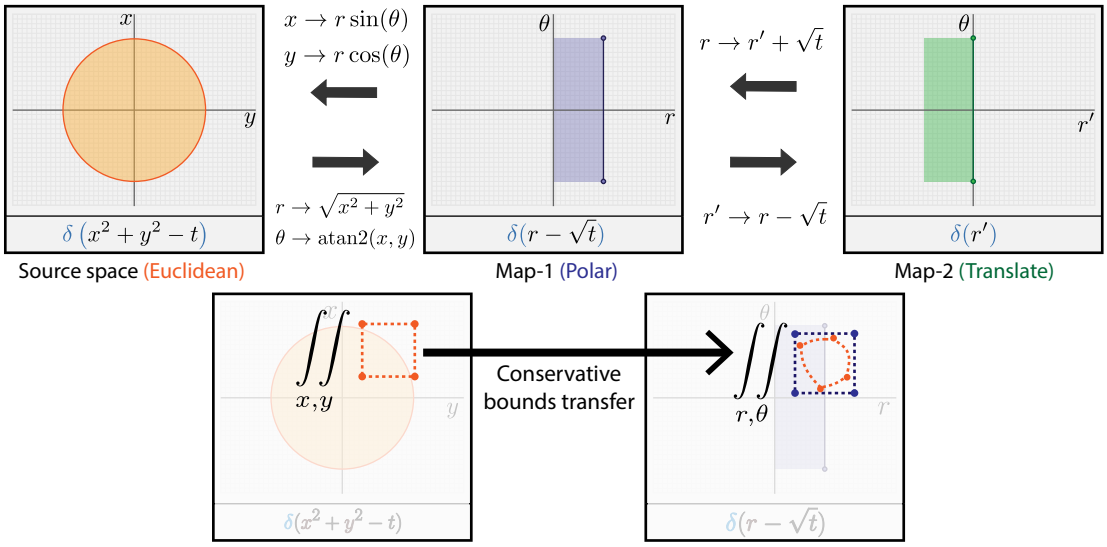


Fig. 4. Delta reparametrization. This flow diagram explains a crucial step in our compiler pass for handling a Dirac delta $\boldsymbol{\delta}(\phi(\vec{x}))$. Our key idea is to reduce this expression to a normal form (e.g., $\boldsymbol{\delta}(y_1)$) through a series of reparameterizations by composing diffeomorphisms. This can be thought of as expressing the expression in a new coordinate system where the expression is exactly coincident with an axis of integration. Here we show an example of reparametrizing a condition representing the area of a circle $\boldsymbol{\delta}(x^2+y^2-t)$. We can transform the coordinates from Cartesian to polar, making it an integral over the angle and radius. Finally we can translate $\boldsymbol{\delta}\left(r-\sqrt{t}\right)$ to obtain the simplified $\boldsymbol{\delta}(r')$, which can be automatically eliminated by Teg.

4.4.2 Lowering: Delta Elimination. To achieve closure back to our original language $\mathcal{L}$, we need to eliminate Dirac deltas using a series of rewrites.

Pass 1: Normalization. To reason about and manipulate programs in $\mathcal{L}'$, we normalize the expressions. Normalization proceeds by exhaustively applying the rewrite rules:

$$e_0 * (e_1 + \boldsymbol{\delta}(\phi) * e_2) \longrightarrow e_0 * e_1 + \boldsymbol{\delta}(\phi) * (e_0 * e_2)$$
$$\int_{x=a}^{b} (e_1 + e_2) \longrightarrow \int_{x=a}^{b} e_1 + \int_{x=a}^{b} e_2,$$

These rewrites distribute multiplication and integration to move Dirac deltas up through the expression until they sit directly inside of integrals—at which point they can be fruitfully manipulated by the subsequent passes. An implementation can be more judicious in the use of the second rule above to reduce code-duplication.

Applying these rewrites results in a normal form:

$$e + \sum_{i=1}^{m}(\boldsymbol{\delta}(\phi_i) * e_i) + \sum_{i=1}^{M}\int_{x_1=a_1}^{b_1}\cdots\int_{x_{n_i}=a_{n_i}}^{b_{n_i}}(\boldsymbol{\delta}(\phi_i) * e_i), \quad (7)$$

where $\sum_{i=1}^{k} e_i$ is syntactic sugar for the expression $e_1 + \ldots + e_k$ and m and M are constants because each term $[\phi > 0]$ contributes at most one discontinuity and there are finitely many such terms in an expression in $\mathcal{L}$. Since the derivative is a linear operator, there is at most one delta in each product.

Pass 2: Delta Reparameterization. To eliminate the deltas, we need to first simplify them by jointly reparameterizing the enclosing integrals (Fig. 4). Without loss of generality, let $\vec{x}$ be all of the variables of integration $x_1, \ldots, x_n$ and $\vec{z}$ are the other variables that parameterize ϕ. The delta reparameterization rewrite is:

$$\int_{x_1=a_1}^{b_1}\cdots\int_{x_n=a_n}^{b_n}\boldsymbol{\delta}(\phi(\vec{z},\vec{x})) * e$$
$$\longrightarrow \int_{y_1=a'_1}^{b'_1}\cdots\int_{y_n=a'_n}^{b'_n}\boldsymbol{\delta}(y_1) * M_\phi * \sigma^{-1} e * |J_\phi|(\vec{z},\vec{y})$$

where the new terms on the right (the substitution σ^{-1} within e, the Jacobian term $|J_\phi|$, the new bounds of integration $(\vec{a'},\vec{b'})$, and the mask M_ϕ) are defined as follows.

As we discuss later (Sec. 4.5), ϕ specifies a map $\widehat{\phi}(\vec{z}) : \vec{x} \to \vec{y}$ whose first output component y_1 is the value being passed to the Dirac delta. As a diffeomorphism, ϕ includes the inverse $\phi^{-1}(\vec{z}) : \vec{y} \to \vec{x}$. From this inverse map we can define σ^{-1} as mapping $[x_i \mapsto \phi_i^{-1}(\vec{z},\vec{y})]$ (for each i), where these latter ϕ^{-1} components are reducible to expressions in our base language $\mathcal{L}$.

$|J_\phi|(\vec{z},\vec{y})$ is the determinant of the Jacobian of $\phi^{-1}(\vec{z})$ evaluated at the base point $\vec{y}$. This Jacobian determinant is reducible to expressions in our base language, just as ϕ^{-1} is.

Finally, the bounds $(\vec{a'},\vec{b'})$ are derived to enclose the image of the original domain of integration. The mask $M_\phi = [\psi_1 > 0]\cdots[\psi_k > 0]$ is derived simultaneously to ensure that only points mapped from within the original domain of integration contribute to the new reparameterized integral. Automatic means of deriving these bounds are discussed later (Sec. 4.5).

Pass 3: Delta Annihilation. We now focus on simplifying a single expression in Eq. 7:

$$\int_{y_1=a_1}^{b_1} \cdots \int_{y_n=a_n}^{b_n} \delta(y_1) * e$$

Our aim is to match the variable of integration with the variable in the delta or to move the delta outside of the integral. The following rewrite rules realize this aim:

$$\int_{y=a}^{b} \delta(y) * e \longrightarrow [a < y < b] * [y \mapsto 0]e$$

$$\int_{y=a}^{b} \delta(x) * e \longrightarrow \delta(x) * \int_{y=a}^{b} e$$

The first rule annihilates the delta with the integral and adds the delta contribution if the discontinuity is inside the integration domain. The second commutes a delta and an integral if the delta expession is independent of the variable of integration. Each delta either annihilates with an integral or is removed in the following pass.

Pass 4: Measure zero destruction. All remaining deltas follow the structure of the second term in Eq. 7:

$$\sum_{i=1}^{m} \delta(\phi_i(\vec{x})) * e$$

We set all of the remaining deltas to zero because they do not depend on an integration variable, and thus are only non-zero over a measure zero support.

$$\delta(\phi(\vec{x})) \longrightarrow 0$$

4.5 Reparameterization

We relied on certain properties of the functions $\phi(\vec{x})$ that occur inside of step functions and Dirac deltas. These functions control the shape of discontinuities via diffeomorphisms (i.e., change-of-coordinates) on the domain of integration.

We rely on the following conventions while defining a diffeomorphism $\phi(\vec{z}, \vec{x})$. We assume that ϕ depends on variables of integration $\vec{x}$, as well as other $\vec{z}$; we write $[\vec{a}, \vec{b}]$ to define a multi-dimensional interval (axis-aligned box) with the given expressions as lower and upper bounds; and we write $[\cdot \vec{\cdot} \cdot > 0]$ to indicate a product of multiple component step functions.

The following five components are required for such a diffeomorphism to be well defined.

(1) *Mapping:* a smooth function $\widehat{\phi}(\vec{z}) : \vec{x} \to \vec{y}$ specified as a collection of expressions e_i in $\mathcal{L}$ not using integration or indicator functions. In particular, $\phi(\vec{z}, \vec{x}) = \widehat{\phi}_1(\vec{z}, \vec{x})$ controls where a step function steps or a Dirac delta has a spike.
(2) *Inverse:* a smooth function $\phi^{-1}(\vec{z}) : \vec{y} \to \vec{x}$ that is the inverse of ϕ, specified as expressions in $\mathcal{L}$, similarly to $\widehat{\phi}$.
(3) *Jacobian:* a smooth function $|J_\phi|(\vec{z}, \vec{y})$ representing the determinant of the Jacobian of ϕ^{-1}, specified as expressions in $\mathcal{L}$, similarly to $\widehat{\phi}$.
(4) *Bounds transfer:* a function $B_\phi(\vec{z}) : (\vec{a}, \vec{b}) \to (\vec{a'}, \vec{b'})$ that specifies safe bounds for the reparameterized integral, i.e. $\forall \vec{x} \in [\vec{a}, \vec{b}] : \widehat{\phi}(\vec{z}, \vec{x}) \in [\vec{a'}, \vec{b'}]$. B_ϕ is specified as expressions in $\mathcal{L}$, similarly to $\widehat{\phi}$.
(5) *Bounds mask:* a function $M_\phi(\vec{a}, \vec{b}, \vec{z}) : (\vec{y}) \to \mathtt{Bool}$, that specifies whether the pre-image of a point $\vec{y}$ lies in the original domain of integration, i.e. $\forall \vec{y} \in B_\phi(\vec{a}, \vec{b}), M_\phi(\vec{a}, \vec{b}, \vec{z}, \vec{y}) \iff \phi^{-1}(\vec{z}, \vec{y}) \subseteq [\vec{a}, \vec{b}]$. M_ϕ is specified as an expression in $\mathcal{L}$, of the form $M_\phi = [\psi_1 > 0] \cdots [\psi_k > 0]$.

While programmers may exploit the ability to specify all five of these components of a diffeomorphism, given (1) and (2) the system can automatically construct (3) the Jacobian by applying automatic differentiation, (4) the bounds-transfer by applying interval arithmetic to $\widehat{\phi}$, and (5) the bounds-mask by $M_\phi(\vec{a}, \vec{b}) = \left[\phi^{-1} - \vec{a} > 0\right]\left[\vec{b} - \phi^{-1} > 0\right]$.

The automatic derivation of M_ϕ is well-defined. First, ϕ^{-1} is a diffeomorphism since $\widehat{\phi}$ is a diffeomorphism. Second, because bounds expressions $\vec{a}$ and $\vec{b}$ do not depend on the variables of integration ($\vec{y}$), offsetting by them is equivalent to offsetting by a constant value (with respect to variation of $\vec{y}$). Thus, the expression $\psi = \phi^{-1} - \vec{a}$ is a diffeomorphism with $\widehat{\psi}(\vec{y}) = \phi^{-1}(\vec{y}) - \vec{a}$ and $\psi^{-1}(\vec{x}) = \widehat{\phi}(\vec{x} + \vec{a})$.

In general, the problem of computing the inverse of functions is uncomputable, and the extension of a function $\phi : \mathbb{R}^n \to \mathbb{R}$ to $\widehat{\phi} : \mathbb{R}^n \to \mathbb{R}^n$ is underspecified. However, in some cases, we can automate even this part of the work. For instance, the case of affine expressions can be treated symbolically and robustly. Appendix A specifies the affine diffeomophism.

4.6 Degeneracies

It is well known [Schwartz 1954] that the products of Dirac deltas, indicator functions, and similar such *distributions* are not always well defined or well-behaved. It is possible to create such ill-defined expressions in our language. Here we characterize this set of ill-defined expressions.

One such problematic expression is the derivative of the integral of the product square of a parameterized indicator function:

$$D[\left[\int_{x=0}^{1} [x > t] * [x > t]\right]][t \mapsto dt]$$

On the one hand, if we think of $[x > t] * [x > t]$ as equivalent to $[x > t \wedge x > t] = [x > t]$, then the result of the derivative (after delta-simplification) should be 1, as in the first example of this paper.

However, the derivative (pass 1) gives

$$\int_{x=0}^{1} \delta(x - t) * [x > t] * dt + \int_{x=0}^{1} \delta(x - t) * [x > t] * dt$$

which following delta annihilation (and some added simplification) becomes:

$$2 * ([t > t] * dt * [0 < t < 1])$$

which is 0 assuming $t > t$ is false, or 2 if $t > t$ is taken to be true.

In general, it is not possible for a compiler to determine whether or not such degeneracies exist, since function equivalence is undecidable. This situation is not that different than the presence of

singularities when using a conventional automatic differentiation system – the automatic derivative of numerically stable code is not necessarily numerically stable. However, we can make certain guarantees about the correctness of our automatic differentiation when we can assume that discontinuities occur in *general position* with respect to each other.

A set of smooth $(n-1)$-manifolds in $\mathbb{R}^n$ lie in general position when the intersection of any k of those manifolds is an $n-k$ manifold. Let $\{\phi_i\}$ be a set of diffeomorphisms on $\mathbb{R}^n$. We say that these $\{\phi_i\}$ are in **general position** at point $x \in \mathbb{R}^n$ if the following is true: let $\{\phi_j\} \subseteq \{\phi_i\}$ be the set of maps s.t. $\widehat{\phi_j}_1(x) = 0$; then the vectors $\{\nabla_x \widehat{\phi_j}_1\}$ are linearly independent. We say that a set of maps $\{\phi_i\}$ are in *general position* if they are in general position at every point $x \in \mathbb{R}^n$. Finally, we say that a program e in $\mathcal{L}$ is in **general position** if the set of diffeomorphisms occuring in any integrand of e are in general position for almost-all values (in the sense of Lebesgue measure) of the free variables $\vec{\mathsf{z}}$ of e.

4.7 Guarantee

Theorem 1. *The derivative of the evaluation and the evaluation of the derivative agree*

$$(D_\gamma E[[e]])\sigma = E[[D[[e]]\gamma]]\sigma$$

almost everywhere assuming that $F[[e]]\gamma$ is in general position.

A proof sketch is included in Appendix B.

5 IMPLEMENTATION

In this section, we briefly describe the implementation and relevant additions to the core semantics presented in Sec. 4.

Language features. For clarity of exposition, our semantics are limited to a minimal set of key primitives. On the other hand, our implementation supports a wider range of common features.

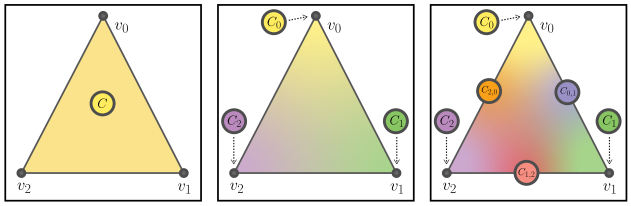
We support let bindings to allow for function abstraction and to avoid unnecessary code replication. Our implementation also allows for the creation and projection of tuples, which can be used for static, fixed-size arrays. Since we do not currently implement integer types, these arrays cannot be dynamically indexed. Returned results are outputted as Python lists allowing for manipulating outputs.

Additionally, we implement reverse-mode differentiation using a standard source-to-source approach. All compiler passes such as those that manipulate Dirac deltas are shared with the forward derivatives as described in detail in the semantics (Sec. 4.4).

Although the implementation lacks looping facilities, it is easy to meta-program Teg in Python to, for example, compute the integral for all of the pixels in an image.

Execution targets. We embed Teg in Python. The Python code can then be lowered to an intermediate representation (IR), where integrals (**Teg**) are discretized to for loops with quadrature. The IR is further converted to a C header file that can be inserted into larger projects, or imported as a Python module. Teg expressions can be evaluated in either NumPy or compiled to CUDA device code.

Numerical validation. We implement a test suite of 95 integral expressions to test both execution targets against finite differences. See the supplementary code for the test cases.



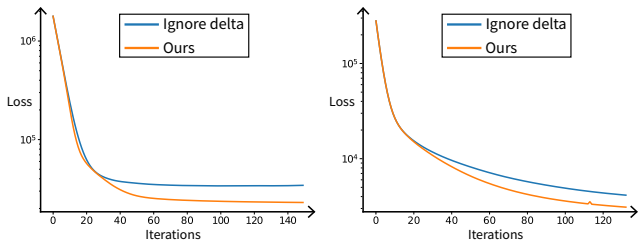
(a) constant color (b) linear color (c) quadratic color

Fig. 5. Color interpolation scheme. We explore three different color interpolation methods for triangulating images: (a) assigning a constant color to a triangle. (b) Linearly interpolating the color from vertices. (c) Quadratically interpolating the color from vertices and edges. In all three settings, each triangle is defined with its own colors, and different triangles do not share color – this creates sharp edges.

Fig. 6. Image triangulation. Given a target image (first column), we optimize the position and color of triangles to fit a stylized version of the image (second column) with different color interpolation schemes (Fig. 5). We initialize the triangles as a grid mesh and set all color to black. We compare to an optimization that ignores the Dirac delta terms of the derivatives (third column) – this is equivalent to using a traditional automatic differentiation system to produce the derivatives. In the constant color case, ignoring the delta term makes the position derivative always zero, so no triangles are moved. Even in the smooth case, ignoring the delta terms lead to artifacts, such as oversmoothing in the linear case and deviation from sharp corners in the quadratic case.

6 APPLICATIONS

We apply Teg to applications spanning several domains in computer graphics and showcase the benefits of automatically accounting for parametric discontinuities during optimization.



(a) linear (b) quadratic

Fig. 7. We compare the loss convergence in the image triangulation task in Fig. 6 between our approach (orange) and an approach that ignore the Dirac delta terms in the derivatives (blue). (a) shows the linear color interpolation case in the second row of Fig. 6, while (b) shows the quadratic color interpolation case in the third row of Fig. 6. Since ignoring the delta derivative leads to a biased estimator, it converges to a worse solution.

6.1 Image triangulation

Given an image, we want to generate a stylized version composed of a triangulated pattern [Lawonn and Günther 2019; Yun 2013]. We can formulate the problem similarly to the 2D differentiable rendering example in Sec. 2.2. Eq. 3 from Sec. 2.2 lays out the rendering model for the constant color image triangulation problem. Here, we further extend the example to have more elaborated shading models. Fig. 5 demonstrates the three different shading methods with increasing parameter complexity:

(1) *Constant.* A triangle is assigned a single color independent of the position (x, y) (Eq. 3).

(2) *Linear.* We define three colors $\mathtt{C_i}$, one for each vertex on the triangle. We then use barycentric interpolation to compute the value at a given continuous (x, y) point. In Teg, this can be expressed as

$$\text{lin\_color} = (\mathtt{w_1 C_1} + \mathtt{w_2 C_2} + \mathtt{w_3 C_3}).$$

where the interpolation weights $\mathtt{w}$ are

$$\mathtt{w_1} = ((\mathtt{x} - \mathtt{v_2.x}) * (\mathtt{y} - \mathtt{v_3.y}) - (\mathtt{x} - \mathtt{v_3.x}) * (\mathtt{y} - \mathtt{v_2.y})) \,/\, \text{norm}$$

$$\text{norm} = (\mathtt{v_1.x} - \mathtt{v_2.x}) * (\mathtt{v_1.y} - \mathtt{v_2.y}) - (\mathtt{v_1.x} - \mathtt{v_3.x}) * (\mathtt{v_1.y} - \mathtt{v_2.y}).$$

$\mathtt{w_2}$ and $\mathtt{w_3}$ are defined similarly.

(3) *Quadratic.* We can further use six colors per triangle, one for each vertex (C_1, C_2, C_3) and edge $(C_{1,2}, C_{2,3}, C_{3,1})$:

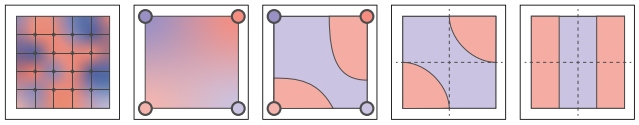
$$\text{quad\_color} = w_1' * C_1 + w_2' C_2 + w_3' C_3 + w_{1,2}' C_{1,2} + w_{2,3}' C_{2,3} + w_{3,1}' C_{3,1},$$

where the interpolation weights w' are defined using the linear interpolation weights w:

$$w_1' = w_1 * (2 * w_1 - 1), w_2' = w_2 * (2 * w_2 - 1), w_3' = w_3 * (2 * w_3 - 1)$$

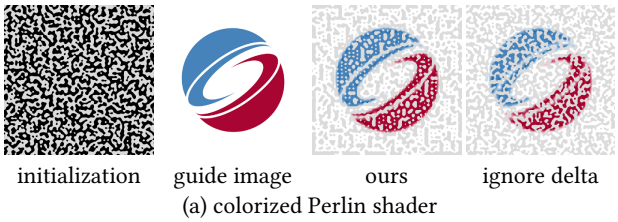
$$w_{1,2}' = 4 * w_1 * w_2, w_{2,3}' = 4 * w_2 * w_3, w_{3,1}' = 4 * w_3 * w_1.$$

The colors C are defined per-triangle and are not shared. This is crucial for representing sharp edges. Unlike the constant color case, the weights in linear and quadratic interpolation depend on the positions of the triangle's vertices. This leads to non-zero gradients even if we ignore the contribution from the Dirac deltas. However, ignoring the deltas generally produces worse results.



(a) Perlin noise (b) bilinear interpolation (c) thresholded (d) scaled and translated (e) hyperbolic coordinates

Fig. 8. Differentiating thresholded Perlin noise textures. Instead of reparameterizing the higher-order polynomials formed by the Perlin noise, we solve a different problem by evaluating the Perlin noise at a discrete grid (a) and reconstruct it using bilinear interpolation (b). We then apply thresholding on the bilinear interpolation (c). The bilinear interpolation still leads to deltas with non-affine arguments. To eliminate the deltas, we apply a diffeomorphism between Cartesian coordinates and hyperbolic coordinates (d). This transforms the deltas into simpler affine conditions (e).



initialization guide image ours ignore delta

(a) colorized Perlin shader

initialization guide image ours ignore delta

(b) two-tone Perlin shader

Fig. 9. Guided Perlin textures. We optimize for the parameters of two shaders based on Perlin noise. The shader in (a) uses the Perlin noise value to decide between using a flat gray color and a low-resolution color map. We keep the decision threshold value fixed and optimize for the color map as well as the noise vectors of the Perlin grid. Ignoring the delta contribution leads to an unchanged noise pattern, while our approach produces a noise pattern that adhere to the logo structure. The shader in (b) uses two colors instead of a color map. We optimize for the grid vectors, the two colors, and the decision threshold value. Without the delta contribution, only the threshold value and colors have non-zero derivatives, which is insufficient to create a structural pattern.

Results. Fig. 6 shows some optimization results with a L^2 loss using Adam [Kingma and Ba 2015], using derivatives automatically generated by Teg. We compare against the gradients obtained when ignoring discontinuities, i.e., the incorrect discretize-then-differentiate approach from Sec. 2.1. We run 150 iterations for both approaches over all images. Fig. 7 shows the convergence plots for both approaches. We compiled to CUDA kernels and run on an RTX 2080 Ti. For the constant fragment case, each full iteration took 0.11s to complete. The more complex linear and quadratic fragment programs took 1.01s and 2.05s per iteration respectively. All reported timings are for a 1200×800 image with approximately 2000

triangles. The constant and quadratic fragment programs had peak memory usage statistics of 177 and 180.8 megabytes respectively.

6.2 Guided Perlin textures

We fit the parameters of a discontinuous procedural shader to a target image. A common pattern in procedural shaders is to threshold Perlin noise [1985] – the noise function is compared to a threshold to decide which color to use for producing textures with segmented regions. The thresholding produces complex discontinuities that require special treatment.

The noise function produced from Perlin noise is at least a 4th-degree polynomial in two variables. Eliminating the resulting delta expression thus requires finding a diffeomorphism from this space to space where the delta expression is affine. Solving the polynomial system in a numerically robust way is challenging. Fortunately, there is an easier formulation that avoids this problem (Fig. 8).

We evaluate the Perlin noise function at discrete positions on a grid. We compute the value at continuous points through bilinear interpolation of the four closest grid points:

$$\begin{aligned}\text{noise} =&\big(\mathtt{N_{0,0}} * (1-\mathtt{x}) + \mathtt{N_{1,0}} * (\mathtt{x})\big) * (1-\mathtt{y}) +\\ &\big(\mathtt{N_{0,1}} * (1-\mathtt{x}) + \mathtt{N_{1,1}} * (\mathtt{x})\big) * (\mathtt{y}),\end{aligned}$$

where $\mathtt{N_{i,j}}$ are values computed by the noise function. Importantly, the continuous noise function is piecewise bilinear, which is neither linear nor affine.

To use thresholded noise to produce a two-color shader, we arrive at the following expression that selects color $\mathtt{C_+}$ if the noise is above the threshold T, and $\mathtt{C_-}$ otherwise:

$$\text{shader} = \mathtt{C_-} + [\,\mathtt{noise} > \mathtt{T}\,] * (\mathtt{C_+} - \mathtt{C_-}).$$

The condition $[\,\mathtt{noise} > \mathtt{T}\,]$ is in the general form $k_{(xy)}xy + k_{(x)}x + k_{(y)}y + k_{(1)}$ that traces an off-axis rectangular hyperbola in x and y. This is not the common affine pattern, so we reparametrize the delta expression through two diffeomorphisms:
(i) scale and translate to move the rectangular hyperbola to the center, $(x, y) \mapsto (x', y')$ (Fig. 8(d)):

$$\mathtt{x'} \to \sqrt{\mathtt{k^{(xy)}}}\,\mathtt{x} + \frac{\mathtt{k^{(y)}}}{\sqrt{\mathtt{k^{(xy)}}}}, \qquad \mathtt{y'} \to \sqrt{\mathtt{k^{(xy)}}}\,\mathtt{y} + \frac{\mathtt{k^{(x)}}}{\sqrt{\mathtt{k^{(xy)}}}}$$

$$\mathtt{x} \to \left(\mathtt{x'} - \frac{\mathtt{k^{(y)}}}{\sqrt{\mathtt{k^{(xy)}}}}\right) / \sqrt{\mathtt{k^{(xy)}}}, \quad \mathtt{y} \to \left(\mathtt{y'} - \frac{\mathtt{k^{(x)}}}{\sqrt{\mathtt{k^{(xy)}}}}\right) / \sqrt{\mathtt{k^{(xy)}}},$$

(ii) convert from Cartesian to hyperbolic coordinates $(x', y') \mapsto (u, v)$ (Fig. 8(e)):

$$\mathtt{u} \to \pm\sqrt{\mathtt{x'y'}}, \quad \mathtt{v} \to \sqrt{\frac{\mathtt{x'}}{\mathtt{y'}}}$$

$$\mathtt{x'} \to \mathtt{uv}, \qquad \mathtt{y'} \to \frac{\mathtt{u}}{\mathtt{v}}.$$

Note that there are two possible values for u. This is because the delta expression in this new space takes the form $\delta\left(\mathtt{u}^2 - \mathtt{c}\right)$, which is equivalent to $\frac{1}{2}\mathtt{c}^{-1}\left(\delta\left(\mathtt{u} - \sqrt{\mathtt{c}}\right) + \delta\left(\mathtt{u} + \sqrt{\mathtt{c}}\right)\right)$. Therefore, there are two hyperbolic spaces, which correspond to each of the two delta expressions in the new space, as shown by Fig. 8(c).

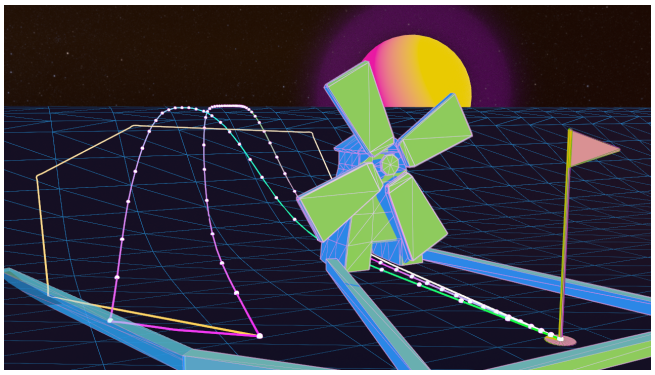
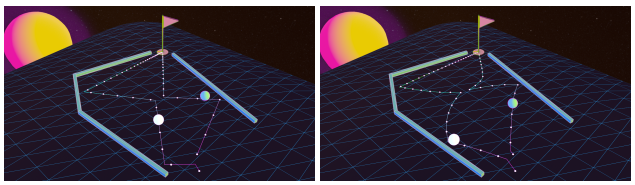


Fig. 10. Minigolf. We use our language to search for hole-in-one trajectories in a minigolf game. Given an initial guess of the trajectory (the orange path), we optimize for path parameters that minimize the action integral over a Lagrangian with friction and contact forces. The walls and the windmill blades are potential contact points where the velocity of the ball is discontinuous. We solve the path using a second-order trust-region based Newton method. The two paths are found using slightly varying setups: one with k = 0.4 hitting no walls and one with k = 0.2 hitting the bottom-left wall.



(a) ours (b) without delta

Fig. 11. Double minigolf. We solve a similar problem to Fig. 10 with two golf balls on a flat surface that can collide with each other. We compare to a solution that ignores the discontinuity derivatives. Our solution obeys the law of reflection. Ignoring the delta contribution from the discontinuities lead to non-physical behaviors that violate Newton's first law.

Results. Fig. 9 shows the optimization results with an L^2 loss using Adam [Kingma and Ba 2015], using the shader discussed here with a modified version of a spatially varying color map $\mathtt{C_+(x, y)}$ instead of the uniform color $\mathtt{C_+}$. We compare against the gradients obtained when ignoring the derivative contribution of discontinuities. The noise structure cannot be optimized when the deltas contributions are ignored. We run the optimization for 300 iterations for both approaches. On an RTX 2080 Ti, each iteration took 0.033s and used 135 megabytes of memory for 400 × 400 noise and guide images, and a 40 × 40 Perlin vector grid.

6.3 Trajectory optimization with contact

Consider a minigolf task (Fig. 10 and 11), where we want to hit a hole-in-one from a starting position, while hitting a pre-specified sequence of walls along the way. We want to find a path $\mathtt{q(t)}$ for the ball satisfying its equations of motion and additional boundary conditions, including: $\mathtt{q(t_0)}$ is at its start location and $\mathtt{q(t_1)}$ is at the hole, for start and end times $\mathtt{t_0}$ and $\mathtt{t_1}$. However, this problem

is complicated because of discontinuities in the velocity trajectories at points of contact, and contact times that are not known a priori.

Using the methodology introduced in Sec. 2.3, we can optimize the trajectory in Teg, without resorting to complicated machinery such as function smoothing or non-linear programming. We use a Lagrangian energy with a frictional term [Bateman 1931]:

$$\mathtt{L} = \left(\frac{1}{2}\mathtt{m}\mathtt{q}_\mathtt{t}^2 - \mathtt{V}(\mathtt{q})\right)\exp(\frac{\mathtt{k}}{\mathtt{m}}\mathtt{t}),$$

where $\mathtt{m}$ is the mass of the ball, $\mathtt{q}_\mathtt{t}$ is the time derivative of the position $\mathtt{q}$, and $\mathtt{k}$ is the frictional coefficient. The position, $\mathtt{q}(\mathtt{t})$, is parametrized as a linear spline. The potential energy, $\mathtt{V}$, contains the gravitational force $\mathtt{m} * \mathtt{g} * \mathtt{q}.\mathtt{y}$. Following Sec. 2.3, we can incorporate elastic contact via a high potential within the barriers $\mathtt{m} * \mathtt{C} * \mathtt{H}_\mathtt{c}(\mathtt{q})$, where $\mathtt{H}_\mathtt{c}$ is a function that is positive only inside the barrier, such that the contact force is only applied when the object interpenetrates a wall (C is set to a large number). We can also have walls that move with time – the windmill in Fig. 10 is one such example, the blade of the windmill is blocking the golf ball only at a certain timeframes. Our solution paths thus correspond to finding stationary points of the action $\mathtt{S} = \int \mathtt{L}$. We initialize the optimizer with a non-physical path that contacts the input wall sequence, so that the system finds a physical path with the desired contact behavior.

Due to the size of the scenes and to avoid numerical instability, we parameterize $\mathtt{q}(\mathtt{t})$ via a set of generalized coordinates that implicitly constrain the path to contact our given input walls. Note that this choice of coordinates only simplifies the optimization; our problem is still the same – we want to solve for where and when the contact points are, wherein velocities are discontinuous.

We set up two scenes for experimentation: in one scene we have a hill and a windmill, and in the other scene we make two golf balls collide with each other but ask both of them to reach the goal. For the windmill scene we set the friction coefficient to $\mathtt{k} = 0.2$ and 0.4 for two different runs, and for the double minigolf scene we set $\mathtt{k} = 0.2$. Fig. 10 and 11 show the results. We use a second-order trust-region based Newton method implemented in scipy for optimization. We perform a coarse-to-fine optimization strategy by starting with a low-resolution trajectory, then gradually refining it to higher resolutions. The final resolution is around 10 control vertices between each collision. The method converges quickly with the tolerance of 10^{-8} to 10^{-12}. In our setting, the collision events need to be known a priori, but the collision position can be anywhere. We also compare to an optimization that ignore the derivatives coming from the discontinuities in Fig. 11. It clearly converges to a non-physical result, showing the importance of incorporating the Dirac delta terms. The optimization uses a single thread, takes (on average) 1.17s per iteration and uses 779.7 megabytes of memory on an Intel Core i9-9900K.

6.4 Optimizing a discontinuous bungee

Now we discuss optimization of a physical design in the presence of discontinuities. Suppose we want to design a spring system for bungee jump (Fig. 12). The motion of a spring is traditionally modeled using Hooke's law $m\ddot{x} = mg - kx$, where m is mass, g is gravity, and k is the spring constant, which models the stiffness of the spring. The term kx assumes that the material does not deform or lock and

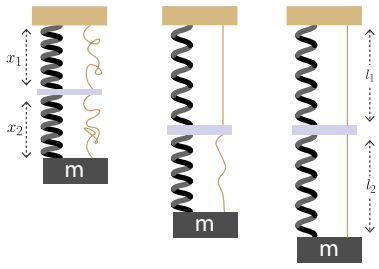
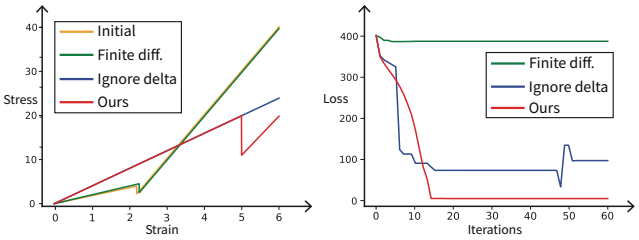


Fig. 12. Optimizing a discontinuous bungee.



(a) Stress-strain curves (b) Loss convergence

Fig. 13. Bungee results: We show the the stress-strain curves and loss for optimizations using finite differences, ignoring discontinuities, and using Teg. The discontinuous stress-strain curve leads to Dirac delta during differentiation. This makes it difficult to tune finite difference step sizes and ignoring the deltas lead to suboptimal results. Our final loss is 18 times lower than ignoring deltas.

is not part of a composite. For example, metal may bend, fracture, or exhibit structural transitions under extreme heat or cold [Hibbeler 2000; Lin et al. 2017; Tabin et al. 2016]. Composite materials such as rebar concrete (steel-reinforced concrete) exhibit non-linear stress-strain curves corresponding to the transitions between materials [Hibbeler 2000]. Woven materials such as yarn may lock producing discontinuities in strain. In these cases, the kx term is replaced by a stress function $s(x)$ that is discontinuous, making the system analytically intractable. A natural way to estimate the solution is by using numerical methods.

We study an idealized series of bungee cords that deform. After deformation, a string prevents further extension of the spring. We jointly minimize the time and acceleration of a person connected to this bungee-string system. We optimize the spring constants k_1, k_2 and the lengths of the strings l_1, l_2 (Fig. 12). We add the hard constraint that the person does not hit the ground to prevent death.

Derivation. With two bungees-string links, the stress function is:

$$s(x) = \begin{cases} k_1x_1 + k_2x_2 \text{ if } x_1 \le l_1, x_2 \le l_2 \\ \alpha k_1 l_1 + k_2(x - l_1) \text{ if } x_1 > l_1, x_2 < l_2 \\ \alpha k_2 l_2 + k_1(x - l_2) \text{ if } x_1 < l_1, x_2 > l_2 \\ g \text{ if } x_1 \ge l_1, x_2 \ge l_2 \end{cases}$$

Table 1. Parameter and loss values for the initialization (Init), finite differences (Finite Diff), not accounting for the Dirac deltas (No Deltas), and accounting for the Dirac deltas (With Deltas).

	Init	Finite Diff	No Deltas	With Deltas
k_1	1.00	1.15	2.72	2.58
k_2	10.0	10.01	7.53	8.92
l_1	2.00	2.05	3.91	3.88
l_2	10.00	10.01	9.99	10.02
Loss	417.49	402.72	101.53	5.50

where α is the plastic deformation factor of the bungees, and x_1 and x_2 are the individual displacements of two springs that sum up to the total displacement x: $x = x_1 + x_2$. Also $k_1 x_1 = k_2 x_2$, since the springs are massless and the force at the bottom of the first spring must balance the force at the top of the second spring. The corresponding second-order autonomous differential equation:

$$m\ddot{x} = mg - s(x)$$

cannot be solved by analytical means due to the discontinuities. If we let $v = \frac{dx}{dt}$ then since $\ddot{x} = \frac{dv}{dt} = v\frac{dv}{dx}$, we have:

$$mv\mathrm{d}v = (mg - s(x))\mathrm{d}x$$

Dividing by m, integrating both sides of the equation assuming the system is initially at rest, and solving for v gives:

$$\mathsf{v(x)} = \left(2 * \int_{\mathsf{z=1}}^{\mathsf{x}} \mathsf{g} - \frac{\mathsf{s(z)}}{\mathsf{m}}\right)^{-1/2},$$

where l is lowest acceptable height in the trajectory (right above the ground). Assuming the system is initially at position 0, we solve for time by substituting $v = \frac{dx}{dt}$ and integrating after isolating $\mathrm{d}t$ to one side of the equation. Thus, we find the time it takes for the person to fall is:

$$\mathsf{t} = \int_{\mathsf{x=l}}^{\mathsf{u}} \left(2 * \int_{\mathsf{z=0}}^{\mathsf{x}} \mathsf{g} - \frac{\mathsf{s(z)}}{\mathsf{m}}\right)^{-1/2},$$

where u is the initial height above the ground. We bound the total displacement by constraining the velocity to be 0 at l which is just above the ground. The final constrained optimization problem is:

$$\min_{k_1,k_2,l_1,l_2} \mathsf{t} + \mathsf{a(l)}^2 \text{ such that } \mathsf{v(l)} = 0,$$

where $\mathsf{a(x)}$ is the acceleration at position x. In words, the goal is to minimize the time to fall plus the squared acceleration given that the final velocity just above the ground is 0. We use the trust region constrained algorithm with BFGS approximated Hessian implemented in scipy to optimize k_1, k_2, l_1, and l_2.

Experiment. We now detail the experimental results for optimizing the parameters of the bungee-string system. The system starts 5 meters above the ground and we initialize it with arbitrarily chosen parameters. These parameters do not satisfy the constraint that $\mathsf{v(l)} = 0$. We set the deformation factor $\alpha = 0.2$ and let the bungee bottom out just above the ground at 10^{-5}.

We compare to derivatives computed by finite difference or automatic differentiation that ignore the Dirac deltas. Fig. 13a depicts the different stress curves for each of these parameter assignments. Fig. 13b shows the loss during optimization. Before iteration 50, the ignore delta solution is infeasible, whereas ours is feasible by iteration 15. Table 1 includes the final parameter assignments and their corresponding loss. The computation took 0.048s per iteration and used 145 megabytes of memory on an i9-9900K.

7 LIMITATIONS AND FUTURE WORK

We have shown that our approach to handling parametric discontinuities is applicable to problems in graphics and physical simulation. We now detail the limitations on the expressivity and implementation of our approach and how they may be addressed in the future.

7.1 Non-Smooth Builtins and Changes of Coordinates

In practice, we use many functions that are not defined everywhere (division, trigonometric functions) and violate our theoretical assumptions of smoothness; such functions with singularities and asymptotes may also be numerically unstable near such non-smooth regions. This is often acceptable for applications, but often leads to difficult kinds of numerical debugging. Such challenges are familiar for graphics programmers, but the reparameterization machinery we present here can make such problems even harder to debug. It would be helpful to figure out better software engineering tools for analyzing automatically differentiated code, especially in the presence of additional code transformations.

7.2 First-Class Derivatives: Inside of Integrals

Our guarantee of correctness does not consider integrals of derivatives. In particular, applying our differentiation to some expression (without the context of integration) will eliminate Dirac deltas arising from discontinuities even if we later place the resulting differentiated expression inside of an integral. However, keeping Dirac deltas around instead would allow users to form expressions that are non-linear in those delta functions (e.g., products of deltas).

Still, there are problems that our system does not support, yet a particular treatment of the Dirac deltas in question leads to a desirable result. For example, the action integral $\int L(t, x, x_t)$ where x_t is the time derivative of position, contains the kinetic energy $\frac{1}{2}mx_t^2$. This is not a problem for any physical trajectory, since all such trajectories are C0 continuous (otherwise objects would instantaneously teleport). However, in the case of our examples, x_t is not C0 continuous, and so x_t^2 is not properly in general position.

Nonetheless, our method appears to work suitably for the physics problems we investigated. Further investigation into the underlying mathematics of distributions, and the corresponding automatic differentiation compilers, is warranted to help ensure reliably correct behavior for products of distributions.

7.3 Tensors Manipulation

The proposed semantics and implementation lacks the facility for tensor manipulations such as indexing, block computations, etc. Instead, data is implicitly unrolled and processed as scalar values. Implementation of these additional constructs is important for applying our language to problems in domains including deep learning. Furthermore, there are important considerations with respect to the interactions between tensor operations and other language interactions, specifically, derivatives of integrals with discontinuities.

7.4 Performance

For an expression e of size n with m-many deltas, our automatic differentiation can end up duplicating most of e as many as m times. Furthermore, we can only perform the code transformations (e.g., normalization and reparameterization) that constitute our differentiation as whole-program transformations. For instance, deep-learning frameworks such as TensorFlow or PyTorch expose composable layers using the chain rule as their modularity principle. It is unclear if and how the derivatives of integrals we consider in this paper can be made fully modular and composable in a similar way. However, the methods considered in this paper can be used to write differentiable layers so long as the integrals in question are wholly contained within a single layer.

7.5 Approximations other than Integral Discretization

Many other approximate operations other than integral discretization are not commutative with differentiation. For example, approximating a function using a piecewise constant function makes the derivative of the approximation ill-behaved. Extending our idea to general function approximation is an interesting direction for future work.

8 CONCLUSIONS

We explore a systematic way to solve graphics and physical simulation problems that involve differentiation, integration, and parametric discontinuities. We formalize the semantics of a new programming language and implement these semantics in Teg. In the same way that automatic differentiation frameworks, such as TensorFlow and PyTorch, made implementation of machine learning algorithms accessible, we believe our differentiable programming language makes a significant first step towards making the implementation of differentiable graphics systems accessible.

ACKNOWLEDGMENTS

We thank Fredo Durand for the discussions of the idea in the early stage and proofreading, Ante Qu for tips on handling friction, Paul Zhang for his discussions on image triangulation and diffeomorphisms, Joshua Fishman and Tao Du for their advice on physical simulation methods, Samuel Tenka for his insightful discussions of distribution theory, and Luke Anderson for his detailed proof-reading. This research was funded under DARPA agreement HR00112090017.

REFERENCES

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems.

Luke Anderson, Tzu-Mao Li, Jaakko Lehtinen, and Frédo Durand. 2017. Aether: An embedded domain specific sampling language for Monte Carlo rendering. *ACM Trans. Graph. (Proc. SIGGRAPH)* 36, 4 (2017), 1–16.

James Arvo. 1994. The Irradiance Jacobian for Partially Occluded Polyhedral Sources. In *SIGGRAPH*. 343–350.

Sai Praveen Bangaru, Tzu-Mao Li, and Frédo Durand. 2020. Unbiased warped-area sampling for differentiable rendering. *ACM Trans. Graph. (Proc. SIGGRAPH Asia)* 39, 6 (2020), 1–18.

Alan H Barr, Bena Currin, Steven Gabriel, and John F Hughes. 1992. Smooth interpolation of orientations with angular velocity constraints using quaternions. *Comput. Graph. (Proc. SIGGRAPH)* 26, 2 (1992), 313–320.

Harry Bateman. 1931. On dissipative systems and related variational principles. *Physical Review* 38, 4 (1931), 815.

John T Betts. 1998. Survey of numerical methods for trajectory optimization. *Journal of guidance, control, and dynamics* 21, 2 (1998), 193–207.

John T. Betts. 2009. *Practical Methods for Optimal Control and Estimation Using Nonlinear Programming* (2nd ed.). Cambridge University Press, USA.

Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D. Goodman. 2019. Pyro: Deep Universal Probabilistic Programming. *J. Mach. Learn. Res.* 20, 1 (2019), 973–978.

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

Ricky T. Q. Chen, Yulia Rubanova, Jesse Bettencourt, and David K Duvenaud. 2018. Neural Ordinary Differential Equations. In *Advances in Neural Information Processing Systems*, Vol. 31. 6571–6583.

Michael F Cohen. 1992. Interactive spacetime control for animation. *Comput. Graph. (Proc. SIGGRAPH)* (1992), 293–302.

J.F. Colombeau. 1984. *New Generalized Functions and Multiplication of Distributions.* North-Holland.

Filipe de Avila Belbute-Peres, Kevin Smith, Kelsey Allen, Josh Tenenbaum, and J. Zico Kolter. 2018. End-to-End Differentiable Physics for Learning and Control. In *Advances in Neural Information Processing Systems*, Vol. 31. 7178–7189.

Martin de La Gorce, David J Fleet, and Nikos Paragios. 2011. Model-based 3D hand pose estimation from monocular video. *IEEE Trans. Pattern Anal. Mach. Intell.* 33, 9 (2011), 1793–1805.

Zachary Devito, Michael Mara, Michael Zollhöfer, Gilbert Bernstein, Jonathan Ragan-Kelley, Christian Theobalt, Pat Hanrahan, Matthew Fisher, and Matthias Niessner. 2017. Opt: A Domain Specific Language for Non-Linear Least Squares Optimization in Graphics and Imaging. *ACM Trans. Graph.* 36, 5 (2017), 171:1–171:27.

P.A.M. Dirac. 1981. *The Principles of Quantum Mechanics.* Clarendon Press.

Peter Dyer and SR McReynolds. 1968. On optimal control problems with discontinuities. *J. Math. Anal. Appl.* 23, 3 (1968), 585–603.

Conal Elliott. 2018. The Simple Essence of Automatic Differentiation. *International Conference on Functional Programming* (2018).

Timon Gehr, Samuel Steffen, and Martin Vechev. 2020. λPSI: exact inference for higher-order probabilistic programs. In *Programming Language Design and Implementation.* 883–897.

Moritz Geilinger, David Hahn, Jonas Zehnder, Moritz Bächer, Bernhard Thomaszewski, and Stelian Coros. 2020. ADD: Analytically Differentiable Dynamics for Multi-Body Systems with Frictional Contact. *ACM Trans. Graph. (Proc. SIGGRAPH Asia)* 39, 6 (2020).

Ioannis Gkioulekas, Shuang Zhao, Kavita Bala, Todd Zickler, and Anat Levin. 2013. Inverse Volume Rendering with Material Dictionaries. *ACM Trans. Graph. (Proc. SIGGRAPH Asia)* 32, 6 (2013), 162:1–162:13.

Andreas Griewank and Andrea Walther. 2008. *Evaluating Derivatives.* Society for Industrial and Applied Mathematics.

Yu Guo, Miloš Hašan, Lingqi Yan, and Shuang Zhao. 2020. A Bayesian Inference Framework for Procedural Material Parameter Estimation. *Comput. Graph. Forum (Proc. Pacific Graphics)* 39, 7 (2020), 255–266.

Christian Hafner, Christian Schumacher, Espen Knoop, Thomas Auzinger, Bernd Bickel, and Moritz Bächer. 2019. X-CAD: Optimizing CAD Models with Extended Finite Elements. *ACM Trans. Graph. (Proc. SIGGRAPH Asia)* 38, 6 (2019).

William Rowan Hamilton. 1834. XV. On a general method in dynamics; by which the study of the motions of all free systems of attracting or repelling points is reduced to the search and differentiation of one central relation, or characteristic function. *Philosophical transactions of the Royal Society of London* 124 (1834), 247–308.

Charles R Hargraves and Stephen W Paris. 1987. Direct trajectory optimization using nonlinear programming and collocation. *Journal of guidance, control, and dynamics* 10, 4 (1987), 338–342.

R.C. Hibbeler. 2000. *Mechanics of Materials.* Prentice Hall.

Philipp Holl, Nils Thuerey, and Vladlen Koltun. 2020. Learning to Control PDEs with Differentiable Physics. In *International Conference on Learning Representations.*

Yuanming Hu, Luke Anderson, Tzu-Mao Li, Qi Sun, Nathan Carr, Jonathan Ragan-Kelley, and Frédo Durand. 2020. DiffTaichi: Differentiable Programming for Physical Simulation. *International Conference on Learning Representations* (2020).

K. H. Hunt and F. R. E. Crossley. 1975. Coefficient of Restitution Interpreted as Damping in Vibroimpact. *Journal of Applied Mechanics* 42, 2 (1975), 440–445.

Jeevana Priya Inala, Sicun Gao, Soonho Kong, and Armando Solar-Lezama. 2018. REAS: combining numerical optimization with SAT solving. *arXiv* (2018).

Kenneth E. Iverson. 1962. *A Programming Language.* John Wiley & Sons, Inc.

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

James T. Kajiya. 1986. The Rendering Equation. *Comput. Graph. (Proc. SIGGRAPH)* 20, 4 (1986), 143–150.

Michael Kass. 1992. CONDOR: Constraint-Based Dataflow. *Comput. Graph. (Proc. SIGGRAPH)* 26, 2 (1992), 321–330.

Michael Kass, Andrew Witkin, and Demetri Terzopoulos. 1988. Snakes: Active contour models. *Int. J. Comput. Vision* 1, 4 (1988), 321–331.

Hiroharu Kato, Yoshitaka Ushiku, and Tatsuya Harada. 2018. Neural 3D Mesh Renderer. In *Computer Vision and Pattern Recognition*. IEEE, 3907–3916.

Diederick P Kingma and Jimmy Ba. 2015. Adam: A method for stochastic optimization. In *International Conference on Learning Representations*.

Donald E Knuth. 1992. Two notes on notation. *The American Mathematical Monthly* 99, 5 (1992), 403–422.

Alp Kucukelbir, Rajesh Ranganath, Andrew Gelman, and David M. Blei. 2015. Automatic Variational Inference in Stan. In *Advances in Neural Information Processing Systems*. 568–576.

Samuli Laine, Janne Hellsten, Tero Karras, Yeongho Seol, Jaakko Lehtinen, and Timo Aila. 2020. Modular primitives for high-performance differentiable rendering. *ACM Trans. Graph. (Proc. SIGGRAPH Asia)* 39, 6 (2020), 1–14.

Kai Lawonn and Tobias Günther. 2019. Stylized Image Triangulation. In *Computer Graphics Forum*, Vol. 38. Wiley Online Library, 221–234.

Wonyeol Lee, Hangyeol Yu, Xavier Rival, and Hongseok Yang. 2020. On Correctness of Automatic Differentiation for Non-Differentiable Functions. In *Advances in Neural Information Processing Systems*.

Wonyeol Lee, Hangyeol Yu, and Hongseok Yang. 2018. Reparameterization gradient for non-differentiable models. In *Advances in Neural Information Processing Systems*. 5553–5563.

Alexander K Lew, Marco F Cusumano-Towner, Benjamin Sherman, Michael Carbin, and Vikash K Mansinghka. 2019. Trace types and denotational semantics for sound programmable inference in probabilistic languages. *Proc. ACM Program. Lang.* 4, POPL (2019), 1–32.

Minchen Li, Zachary Ferguson, Teseo Schneider, Timothy Langlois, Denis Zorin, Daniele Panozzo, Chenfanfu Jiang, and Danny M Kaufman. 2020a. Incremental potential contact: Intersection-and inversion-free, large-deformation dynamics. *ACM Trans. Graph. (Proc. SIGGRAPH)* (2020).

Tzu-Mao Li. 2019. *Differentiable Visual Computing*. Ph.D. Dissertation. Massachusetts Institute of Technology. Advisor(s) Durand, Frédo.

Tzu-Mao Li, Miika Aittala, Frédo Durand, and Jaakko Lehtinen. 2018a. Differentiable Monte Carlo Ray Tracing through Edge Sampling. *ACM Trans. Graph. (Proc. SIGGRAPH Asia)* 37, 6 (2018), 222:1–222:11.

Tzu-Mao Li, Michaël Gharbi, Andrew Adams, Frédo Durand, and Jonathan Ragan-Kelley. 2018b. Differentiable programming for image processing and deep learning in Halide. *ACM Trans. Graph. (Proc. SIGGRAPH)* 37, 4 (2018), 139:1–139:13.

Tzu-Mao Li, Michal Lukáč, Gharbi Michaël, and Jonathan Ragan-Kelley. 2020b. Differentiable Vector Graphics Rasterization for Editing and Learning. *ACM Trans. Graph. (Proc. SIGGRAPH Asia)* 39, 6 (2020), 193:1–193:15.

Peng Lin, Yonggang Hao, Baoyou Zhang, Shuzhi Zhang, and Jun Shen. 2017. Strain rate sensitivity of Ti-22Al-25Nb (at.alloy during high temperature deformation. *Materials Science and Engineering: A* (2017).

Seppo Linnainmaa. 1970. *The representation of the cumulative rounding error of an algorithm as a Taylor expansion of the local rounding errors*. Master's thesis. Univ. Helsinki.

Shichen Liu, Tianye Li, Weikai Chen, and Hao Li. 2019. Soft Rasterizer: A Differentiable Renderer for Image-based 3D Reasoning. *International Conference on Computer Vision* (2019).

Matthew M. Loper and Michael J. Black. 2014. OpenDR: An Approximate Differentiable Renderer. In *European Conference on Computer Vision*, Vol. 8695. ACM, 154–169.

Guillaume Loubet, Nicolas Holzschuch, and Wenzel Jakob. 2019. Reparameterizing discontinuous integrands for differentiable rendering. *ACM Trans. Graph. (Proc. SIGGRAPH Asia)* 38, 6 (2019), 228.

Damiano Mazza and Michele Pagani. 2021. Automatic differentiation in PCF. *Proceedings of the ACM on Programming Languages* 5 (2021), 1–27.

Antoine McNamara, Adrien Treuille, Zoran Popović, and Jos Stam. 2004. Fluid control using the adjoint method. *ACM Trans. Graph. (Proc. SIGGRAPH)* 23, 3 (2004), 449–456.

Brian Vincent Mirtich. 1996. *Impulse-based dynamic simulation of rigid body systems*. University of California, Berkeley.

Don P Mitchell and Arun N Netravali. 1988. Reconstruction filters in computer-graphics. *Comput. Graph. (Proc. SIGGRAPH)* 22, 4 (1988), 221–228.

Igor Mordatch, Emanuel Todorov, and Zoran Popović. 2012. Discovery of Complex Behaviors through Contact-Invariant Optimization. *ACM Trans. Graph. (Proc. SIGGRAPH)* 31, 4 (2012).

Igor Mordatch, Jack M Wang, Emanuel Todorov, and Vladlen Koltun. 2013. Animating human lower limbs using contact-invariant optimization. *ACM Trans. Graph.* 32, 6 (2013), 1–8.

Henry P Moreton and Carlo H Séquin. 1992. Functional optimization for fair surface design. *Comput. Graph. (Proc. SIGGRAPH)* 26, 2 (1992), 167–176.

Merlin Nimier-David, Delio Vicini, Tizian Zeltner, and Wenzel Jakob. 2019. Mitsuba 2: A retargetable forward and inverse renderer. *ACM Trans. Graph. (Proc. SIGGRAPH Asia)* 38, 6 (2019), 1–17.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems*. 8024–8035.

Barak A. Pearlmutter and Jeffrey Mark Siskind. 2008. Reverse-mode AD in a Functional Framework: Lambda the Ultimate Backpropagator. *Trans. Program. Lang. Syst.* 30, 2 (2008), 7:1–7:36.

Ken Perlin. 1985. An image synthesizer. *Comput. Graph. (Proc. SIGGRAPH)* 19, 3 (1985), 287–296.

Jovan Popović, Steven M Seitz, Michael Erdmann, Zoran Popović, and Andrew Witkin. 2000. Interactive manipulation of rigid body simulations. In *SIGGRAPH*. 209–217.

Zoran Popović and Andrew Witkin. 1999. Physically based motion transformation. In *Comput. Graph. (Proc. SIGGRAPH)*. 11–20.

Michael Posa, Cecilia Cantu, and Russ Tedrake. 2014. A direct method for trajectory optimization of rigid bodies through contact. *The International Journal of Robotics Research* 33, 1 (2014), 69–81.

Ravi Ramamoorthi, Dhruv Mahajan, and Peter Belhumeur. 2007. A First-order Analysis of Lighting, Shading, and Shadows. *ACM Trans. Graph.* 26, 1 (2007), 2.

Maxime Roger, Stéphane Blanco, Mouna El Hafi, and Richard Fournier. 2005. Monte Carlo estimates of domain-deformation sensitivities. *Physical review letters* 95, 18 (2005), 180601.

L. Schwartz. 1950. *Théorie des distributions*. Number v. 2 in Actualités scientifiques et industrielles. Hermann.

L. Schwartz. 1954. Sur l'impossibilité de la multiplication des distributions. *C. R. Acad. Sci. Paris* (1954).

Benjamin Sherman, Jesse Michel, and Michael Carbin. 2021. λ_S: Computable semantics for differentiable programming with higher-order functions and datatypes. *Proc. ACM Program. Lang.* 5, POPL, Article 3 (2021), 31 pages.

Liang Shi, Beichen Li, Miloš Hašan, Kalyan Sunkavalli, Tamy Boubekeur, Radomir Mech, and Wojciech Matusik. 2020. MATch: Differentiable Material Graphs for Procedural Material Capture. *ACM Trans. Graph. (Proc. SIGGRAPH Asia)* 39, 6 (2020), 1–15.

Stan Development Team. 2015. *Stan Modeling Language Users Guide and Reference Manual, Version 2.9.0*. http://mc-stan.org/

Robert F Stengel. 1994. *Optimal control and estimation*. Courier Corporation.

J. Tabin, B. Skoczen, and J. Bielski. 2016. Strain localization during discontinuous plastic flow at extremely low temperatures. *International Journal of Solids and Structures* (2016).

Emanuel Todorov. 2011. A convex, smooth and invertible contact model for trajectory optimization. In *International Conference on Robotics and Automation*. IEEE, 1071–1076.

Christopher D. Twigg and Doug L. James. 2008. Backward Steps in Rigid Body Simulation. *ACM Trans. Graph. (Proc. SIGGRAPH)*, Article 25 (2008).

Kiwon Um, Robert Brand, Yun Fei, Philipp Holl, and Nils Thuerey. 2020. Solver-in-the-Loop: Learning from Differentiable Physics to Interact with Iterative PDE-Solvers. *Advances in Neural Information Processing Systems* (2020).

Eric Veach. 1998. *Robust Monte Carlo Methods for Light Transport Simulation*. Ph.D. Dissertation. Stanford University. Advisor(s) Guibas, Leonidas J.

William Welch and Andrew Witkin. 1992. Variational surface modeling. *Comput. Graph. (Proc. SIGGRAPH)* 26, 2 (1992), 157–166.

R. E. Wengert. 1964. A Simple Automatic Derivative Evaluation Program. *Commun. ACM* 7, 8 (1964), 463–464.

Andrew Witkin and Michael Kass. 1988. Spacetime constraints. *ACM Trans. Graph. (Proc. SIGGRAPH)* 22, 4 (1988), 159–168.

Lifan Wu, Guangyan Cai, Shuang Zhao, and Ravi Ramamoorthi. 2020. Analytic spherical harmonic gradients for real-time rendering with many polygonal area lights. *ACM Trans. Graph. (Proc. SIGGRAPH)* 39, 4 (2020), 134.

Dong Yu, Adam Eversole, Mike Seltzer, Kaisheng Yao, Oleksii Kuchaiev, Yu Zhang, Frank Seide, Zhiheng Huang, Brian Guenter, Huaming Wang, Jasha Droppo, Geoffrey Zweig, Chris Rossbach, Jie Gao, Andreas Stolcke, Jon Currey, Malcolm Slaney, Guoguo Chen, Amit Agarwal, Chris Basoglu, Marko Padmilac, Alexey Kamenev, Vladimir Ivanov, Scott Cypher, Hari Parthasarathi, Bhaskar Mitra, Baolin Peng, and Xuedong Huang. 2014. *An Introduction to Computational Networks and the Computational Network Toolkit*. Technical Report. Microsoft Research.

Dofl Y.H. Yun. 2013. DMesh, Triangulation Image Generator. http://dmesh.thedofl.com/ Accessed: 2021-01-26.

Cheng Zhang, Bailey Miller, Kai Yan, Ioannis Gkioulekas, and Shuang Zhao. 2020. Path-space Differentiable Rendering. *ACM Trans. Graph. (Proc. SIGGRAPH)* 39, 6 (2020), 143:1–143:19.

Cheng Zhang, Lifan Wu, Changxi Zheng, Ioannis Gkioulekas, Ravi Ramamoorthi, and Shuang Zhao. 2019. A differential theory of radiative transfer. *ACM Trans. Graph.*

(Proc. SIGGRAPH Asia) 38, 6 (2019), 227.

Yuan Zhou, Bradley J. Gram-Hansen, Tobias Kohn, Tom Rainforth, Hongseok Yang, and Frank Wood. 2019. LF-PPL: A Low-Level First Order Probabilistic Programming Language for Non-Differentiable Models. In *International Conference on Artificial Intelligence and Statistics (AISTATS)*, Vol. 89. PMLR, 148–157.

A AFFINE DIFFEOMORPHISMS

We syntactically pattern-match affine expressions and automatically generate the appropriate mappings. Assuming the affine expression is of the pattern $\mathtt{c_0x_0} + \mathtt{c_1x_1} + \cdots \mathtt{c_nx_n}$ (the translation term does not affect the reparametrization), we can think of this as a dot product between the expression vector $\vec{\mathtt{c}} = [\mathtt{c_0}, \mathtt{c_1}, \cdots \mathtt{c_n}]$ and the variable vector $\vec{\mathtt{x}}$. In a similar manner, we can pose our target expression as a dot product: $\widehat{\phi}_1(\vec{\mathtt{z}}, \vec{\mathtt{y}}) = \mathtt{y_1} = [1, 0, \cdots 0] \cdot \vec{\mathtt{y}}$.

Our approach to finding such a transformation is to apply a symbolic *rotation matrix* R to $\vec{\mathtt{x}}$ such that R satisfies the mapping $[1, 0, \cdots 0] \mapsto [\mathtt{c_0}, \mathtt{c_1}, \cdots \mathtt{c_n}]$.

We arrive at the following quintuplet for our affine diffeomorphism:

(1) *Mapping:* $\widehat{\phi}(\vec{\mathtt{z}}, \vec{\mathtt{y}}) = \left|\vec{\mathtt{c}}\right| \mathbf{R}\vec{\mathtt{y}}$, where

$$\mathtt{R_{ij}} = \left\{ \begin{array}{ll} c_j^{'}, & \text{for } i = 0 \\ -c_i^{'}, & \text{for } j = 0 \\ \delta_{i,j} - \frac{c_i^{'} \cdot c_j^{'}}{1 + c_0^{'}}, & \text{for } 0 \leq n \leq 1 \end{array} \right\}$$

and $\mathtt{c}_\mathtt{i}^{'}$ denotes the normalized expressions $\mathtt{c}_\mathtt{i}^{'} = \mathtt{c_i} / \left|\vec{\mathtt{c}}\right|$

(2) *Inverse:* $\phi^{-1}(\vec{\mathtt{z}}, \vec{\mathtt{y}}) = \left|\vec{\mathtt{c}}\right|^{-1} \mathbf{R}^{\mathsf{T}} \vec{\mathtt{y}}$. The inverse of a rotation transformation is also the transpose.

(3) *Jacobian:* $|\mathtt{J}_\phi|(\vec{\mathtt{z}}, \vec{\mathtt{y}}) = \left|\vec{\mathtt{c}}\right|^{-1}$. Although the compiler can automatically derive an equivalent expression, we can use the properties of our rotation transformation to simplify the resulting program. The Jacobian adjustment term in this case is the norm of the vector of coefficients.

(4) *Bounds transfer:* a function $\mathtt{B}_\phi(\vec{\mathtt{z}}) : (\vec{\mathtt{a}}, \vec{\mathtt{b}}) \rightarrow (\vec{\mathtt{a}'}, \vec{\mathtt{b}'})$. This expression can be computed through applying interval arithmetic. The expression for the general affine case resolves to the following:

$$\mathtt{B}_\phi^{(\mathtt{j})}(\vec{\mathtt{z}}) = (\mathbf{R}\vec{\mathtt{u}}^{(\mathtt{j})}, \mathbf{R}\vec{\mathtt{v}}^{(\mathtt{j})}), \text{ where}$$

$$\mathtt{u}_\mathtt{i}^{(\mathtt{j})} = \left\{ \begin{array}{ll} a_i & \text{for } R_{ij} > 0 \\ b_i & \text{for } R_{ij} \leq 0 \end{array} \right\}$$

$$\mathtt{v}_\mathtt{i}^{(\mathtt{j})} = \left\{ \begin{array}{ll} b_i & \text{for } R_{ij} > 0 \\ a_i & \text{for } R_{ij} \leq 0 \end{array} \right\}$$

Essentially, we construct expressions for the extreme values for each target-space variable by selecting between the upper or lower bound for each source-space variable depending on the sign of the corresponding coefficient in $\mathbf{R}$. This constructs a set of bounds that are guaranteed to enclose the integration domain.

(5) *Bounds mask:* a function $\mathtt{M}_\phi(\vec{\mathtt{z}})$. The bounds mask is automatically constructed by the compiler, using the process discussed previously in Sec. 4.5.

B PROOF SKETCH OF LANGUAGE CORRECTNESS

Proof. In order to reason about the correctness of our transformations, we must extend the denotational semantics of $\mathcal{L}$ to include $\boldsymbol{\delta}(\cdot)$. This can be done through any suitable choice of a mathematical theory of distributions [Colombeau 1984; Dirac 1981; Schwartz 1950]. This argument will remain agnostic to this choice.

Given our general position assumption, $(D_\gamma E[[e]])\sigma = E[[F[[e]]\gamma]]\sigma$ will hold everywhere except for in some $n-2$ sub-manifold of the free variables of e. This property is true recursively for all sub-expressions of e.

Proceeding on to the Delta Elimination passes, normalization (Pass 1) will preserve equality. The first of our two normalization rewrites is sound because the expression being distributed cannot contain a Dirac delta (by linearity of the $\mathcal{L}'$ grammar). The second of the two rewrites is sound by linearity of integration. Again by the linearity of $\mathcal{L}'$, we must reach the desired normal form.

Reparameterization (Pass 2) is a direct application of change-of-coordinates, and therefore preserves the meaning of the program. As a diffeomorphism, the change of coordinates must preserve the general position condition.

Finally, Delta Annihilation replaces our "volume" integral over n dimensions with a "surface" integral over $n-1$ dimensions. This change of integration domain is sound by the sifting property of the Dirac delta (i.e. $\int_x \delta(x) f(x) = f(0)$). Furthermore, thanks to our general position assumption, the diffeomorphisms for the remaining step/indicator-functions can only evaluate (i.e. $\widehat{\phi}_1$) to 0 on a $n-2$ dimensional sub-manifold of our new domain of integration, which is a measure zero sub-set of the integration domain. As a consequence, this integral is well-defined regardless of the underlying theory of distributions, and produces the expected result.

Finally, the remaining deltas are no longer contained within integrals (since they have either been factored out during Delta Annihilation, or were already outside thanks to normalization). The ϕ maps inside these deltas are zero on at most a measure zero subset of the domain of the free variables. This is fine, since we only claim correctness *almost everywhere*, like most automatic differentiation systems' claim without integrals [Mazza and Pagani 2021].

□