Distributions for Compositionally Differentiating Parametric Discontinuities

JESSE MICHEL, Massachusetts Institute of Technology, USA
KEVIN MU, University of Washington, USA
XUANDA YANG, University of California, San Diego, USA
SAI PRAVEEN BANGARU, Massachusetts Institute of Technology, USA
ELIAS ROJAS COLLINS, Massachusetts Institute of Technology, USA
GILBERT BERNSTEIN, University of Washington, USA
JONATHAN RAGAN-KELLEY, Massachusetts Institute of Technology, USA
MICHAEL CARBIN, Massachusetts Institute of Technology, USA
TZU-MAO LI, University of California, San Diego, USA

Computations in physical simulation, computer graphics, and probabilistic inference often require the differentiation of discontinuous processes due to contact, occlusion, and changes at a point in time. Popular differentiable programming languages, such as PyTorch and JAX, ignore discontinuities during differentiation. This is incorrect for *parametric discontinuities*—conditionals containing at least one real-valued parameter and at least one variable of integration. We introduce Potto, the first differentiable first-order programming language to soundly differentiate parametric discontinuities. We present a denotational semantics for programs and program derivatives and show the two accord. We describe the implementation of Potto, which enables separate compilation of programs. Our prototype implementation overcomes previous compile-time bottlenecks achieving an 88.1x and 441.2x speed up in compile time and a 2.5x and 7.9x speed up in runtime, respectively, on two increasingly large image stylization benchmarks. We showcase Potto by implementing a prototype differentiable renderer with separately compiled shaders.

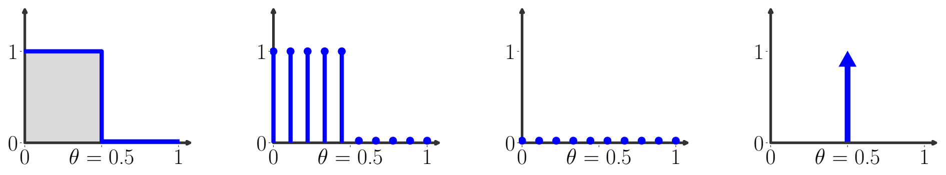
CCS Concepts: • **Theory of computation** → **Denotational semantics**; *Operational semantics*; • **Computing methodologies** → *Rendering*; • **Mathematics of computing** → Functional analysis.

Additional Key Words and Phrases: Differentiable Programming, Denotational Semantics, Differentiable Rendering, Distribution Theory, Probabilistic Programming

ACM Reference Format:
Jesse Michel, Kevin Mu, Xuanda Yang, Sai Praveen Bangaru, Elias Rojas Collins, Gilbert Bernstein, Jonathan Ragan-Kelley, Michael Carbin, and Tzu-Mao Li. 2024. Distributions for Compositionally Differentiating Parametric Discontinuities. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 126 (April 2024), 30 pages. https://doi.org/10.1145/3649843

Authors' addresses: Jesse Michel, Massachusetts Institute of Technology, Cambridge, USA, jmmichel@csail.mit.edu; Kevin Mu, University of Washington, Seattle, USA, kmu0@cs.washington.edu; Xuanda Yang, University of California, San Diego, San Diego, USA, xuanday@ucsd.edu; Sai Praveen Bangaru, Massachusetts Institute of Technology, Cambridge, USA, sbangaru@mit.edu; Elias Rojas Collins, Massachusetts Institute of Technology, Cambridge, USA, erojasc@mit.edu; Gilbert Bernstein, University of Washington, Cambrdige, USA, gilbo@cs.washington.edu; Jonathan Ragan-Kelley, Massachusetts Institute of Technology, Cambridge, USA, jrk@csail.mit.edu; Michael Carbin, Massachusetts Institute of Technology, Cambridge, USA, mcarbin@csail.mit.edu; Tzu-Mao Li, University of California, San Diego, San Diego, USA, tzli@ucsd.edu.

ACM 2475-1421/2024/4-ART126
https://doi.org/10.1145/3649843



(a) Integral of step. (b) Discretized integral. (c) Standard AD deriv. (d) Potto deriv.

Fig. 1. Differentiate Before You Discretize. (a) the integral (shaded) $\int_0^1 [x \leq \theta]\, dx = \theta = 0.5$. (b) shows a discretization of integral as in an implementation in a language without integration. (c) shows the derivative of the integral as computed by Potto, which integrates a Dirac delta spike at θ and returns 1. (d) depicts that standard AD differentiates the discretized program and incorrectly returns 0.

1 INTRODUCTION

Automatic differentiation (AD) is the automated computation of the derivative of a function given just the definition of the function itself. AD has been applied in many domains such as computer graphics and vision [Li et al. 2018], robotics [Bangaru et al. 2021; Hu et al. 2020], and probabilistic inference [Lee et al. 2018] for optimization and uncertainty quantification. Many of these computations are conceived as automatically differentiating continuous functions, however, discontinuous functions also arise naturally.

Discontinuities arise in computer graphics due to object boundaries, occlusion, and sharp changes of color. In robotics and physical simulation, computations can model contact, which causes a discontinuous change in the velocity of an object. In probabilistic inference, the model can have discontinuities. For example, a time-series modeling problem can have a discontinuous change in behavior at a point in time.

Figure 1a illustrates an example of a discontinuity. A discontinuity in a function is a point at which the left and right limits approach different values. In many applications, the occurrence of an integral can introduce a *parametric discontinuity*. A parametric discontinuity is a discontinuity specified by a condition whose value depends on at least one parameter and at least one real-valued variable of integration. The shaded region depicts $f(\theta) = \int_0^1 [x \leq \theta]\, dx$, where the Iverson bracket $[P]$ is 1 if the proposition P holds and is 0 otherwise. Since there is a variable of integration x and parameter θ, the discontinuity $[x \leq \theta]$ is a parametric discontinuity.

State of the Art. Popular AD tools ignore discontinuities during differentiation [Abadi et al. 2015; Bradbury et al. 2018; Paszke et al. 2019]. Ignoring discontinuities that are not parametric discontinuities is correct almost everywhere [Lee et al. 2020]. However, ignoring parametric discontinuities produces incorrect results. In optimization, this leads to slower convergence or even divergence [Bangaru et al. 2021; Lee et al. 2018; Li et al. 2018].

In standard AD systems, when applications include integrals, the typical strategy is to discretize the integral by evaluating the integrand at samples and summing the result. Standard AD then differentiates this discretized program. Figure 1b shows the discretization of the integral in Figure 1a and Figure 1c that it returns 0 because the derivative of each sample is 0 [Abadi et al. 2015; Bradbury et al. 2018; Paszke et al. 2019]. This is incorrect.

Potto. Figure 1d depicts that the correct derivative of the integrand is instead the *Dirac delta distribution* $\delta(\theta - x)$ that integrates to 1 if θ lies in the domain of integration: $D_\theta \int_0^1 [x \leq \theta]\, dx = [0 < \theta < 1]$. We present a differentiable programming language, Potto, that uses *distributions* to differentiate integrals with parametric discontinuities (Teodorescu et al. [2013] Section 1.3.7). Distributions are a generalization of functions that can represent the derivatives of a discontinuous function. In particular, derivatives of Potto programs denote distributions.

We extend the sampling approach in standard AD to additionally sample at parametric discontinuities, where derivatives are non-zero resulting in a correct estimate of the derivative. As a result, Potto supports compositional evaluation and therefore, separate compilation, which was not possible in prior work, Teg [Bangaru et al. 2021].

We provide an example of using Potto for probabilistic inference (risk minimization) and implement both a 2D and a 3D differentiable renderer. Potto significantly improves compile time and is slower in runtime for smaller programs and faster on larger programs (with more discontinuities and more complex discontinuities, i.e., a larger expression in the condition) when compared to Teg [Bangaru et al. 2021]. We find an 88.1x and 441.2x speed up in compile time and a 2.5x and 7.9x speed up in runtime respectively on two increasingly large image stylization benchmarks.

In this paper, we present the following contributions:

- We introduce Potto[1], a language for distribution programming, that is the first to differentiate parametric discontinuities, while supporting compositional evaluation (Section 2). We provide a mathematical introduction to *distribution theory* (Section 3).
- We define the syntax of a core language, Langur (Section 4). A Langur term is an integrand and a Langur program is the integral of a term.
- We provide a type system defining well-formed terms (Section 5). We present the denotational semantics of Langur terms (Section 6) and their derivativess (Section 7). We provide a type system and denotational semantics for programs and their derivatives (Section 8). We prove that the derivative of the denotation of a program is equivalent to the (distributional) derivative denotation of that program.
- We prove that the operational semantics of Langur supports compositional evaluation and therefore, separate compilation (Section 9).
- We implement a 3D renderer in the surface language, Potto[2], and use separate compilation to efficiently swap among shaders. We compare Potto with Teg [Bangaru et al. 2021] on differentiable rendering tasks and find that it significantly improves compilation times, a bottleneck in Teg, and that it is slower in runtime on small programs, but it is faster on larger programs (Section 10).

With Potto, we expand the scope of differentiable programming languages to account for parametric discontinuities. Potto is a first-order language that supports separate compilation, leading to better performance in workflows involving many small changes to a larger program. We envision that our theoretical development and programming language design will lead to more expressive differentiable programming languages that better serve application domains such as computer graphics, robotics, and probabilistic inference.

2 EXAMPLE: RISK MINIMIZATION

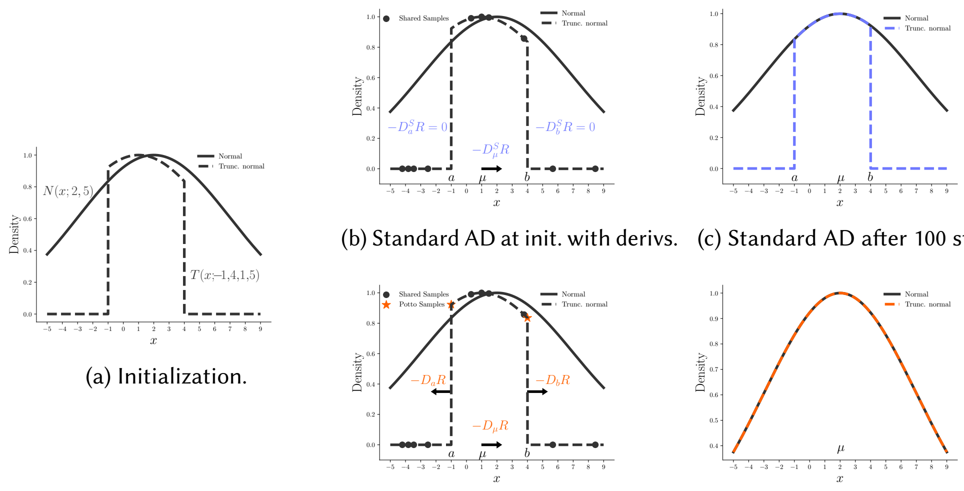
Risk minimization is a fundamental problem in machine learning (Chapter 1.2 of Vapnik [1998]). We present a pedagogical example where the goal is to find parameters θ that minimize the *risk*:

$$R(h_\theta) = \int_l^u \ell(h_\theta(x),\, g(x))\, \mathrm{d}x, \tag{1}$$

where the squared error loss $\ell(h_\theta(x), g(x)) = (h_\theta(x) - g(x))^2$, the bounds are $l = -10$, $u = 10$, the parameters $\theta = [a, b, \mu]$, and $h_\theta(x) = \mathcal{N}(x; \mu, 5)[a \leq x \leq b]$ is the (unnormalized) truncated normal density, and $g(x) = \mathcal{N}(x; 2, 5)$ is the normal density.

[1]We provide the prototype implementation at https://github.com/divicomp/potto.

[2]We provide the code used to produce the applications at https://github.com/divicomp/potto_applications.



(a) Initialization.

(b) Standard AD at init. with derivs.

(c) Standard AD after 100 steps.

(d) Potto at init. with derivs.

(e) Potto after 100 steps.

Fig. 2. At initialization (a), the truncated normal density is far from the normal density. The descent direction in standard AD (b) is defined as $-D^S$ and is correct for μ, but is 0 and therefore incorrect for the truncation points a and b. The black dots in (b) and (d) are shared samples that standard AD and Potto use to estimate the risk R. Standard AD (c) optimizes μ but fails to optimize a and b. Potto (d) samples at a and b (orange stars) to account for the parametric discontinuities and has the same descent direction for μ as standard AD. Potto converges (e) to the desired curve by moving the mean right and widening the truncation points.

Recall that a *parametric discontinuity* is a conditional containing one or more real-valued variables (of integration) and parameters in the condition. The two parametric discontinuities in $R(h_\theta)$ arise due to $a \leq x$ and $x \leq b$ because the parameters a and b are compared to the variable x.

The task is to automatically identify the optimal truncation points a, b and mean μ. At the minimal θ, the risk $R(h_\theta)$ will have parameters $a \mapsto -10$, $b \mapsto 10$, and $\mu \mapsto 2$.

Optimization using AD. A standard approach to solving $\arg\min_\theta R(h_\theta)$ is to use gradient descent, which requires taking the gradient of R with respect to θ and then updating θ in the direction of descent according to the gradient. Modern AD systems enable one to write R as a program and rely on the system to automatically generate the gradient of R.

2.1 Differentiating Parametric Discontinuities

We present the problem of differentiating parametric discontinuities in the context of risk minimization and compare standard AD techniques with Potto.

Standard AD. Figure 2a depicts the (unnormalized) truncated normal and normal densities. Figure 2b shows the descent direction (black arrow) at initialization $-D^S$ for standard AD. We use the notation D^S to denote the derivative computed by standard AD. The descent direction $-D^S_\mu R$ is correct, which is 0 and therefore incorrect for truncation points a and b. Figure 2c illustrates that gradient descent can optimize the mean, but cannot optimize the truncation points.

Figure 2d depicts the descent direction computed by Potto. Potto correctly computes the derivative for the truncation points by sampling at a and b (orange stars) to account for the parametric discontinuities. At initialization, Potto computes the same descent direction for μ as standard AD. Figure 2e shows that the optimization is successful, moving the desired curve by moving the mean right and widening the truncation points.

A standard AD algorithm as implemented in common AD frameworks [Abadi et al. 2015; Paszke et al. 2019] computes the gradient of risk by applying the assumption that the derivative of the integral equals the integral of the derivative. However,

$$D_\theta \int_{-10}^{10} (h_\theta(x) - g(x))^2 \, \mathrm{d}x \neq \int_{-10}^{10} D_\theta (h_\theta(x) - g(x))^2 \, \mathrm{d}x,$$

recalling that $\theta = [a, b, \mu]$, $h_\theta(x) = \mathcal{N}(x; \mu, 5)[a \leq x \leq b]$, and $g(x) = \mathcal{N}(x; 2, 5)$. However, the derivative and integral typically do not commute in the presence of discontinuities and therefore, theoretically, the meaning of the resulting expression is not well-defined. Figure 2c illustrates that when used within a gradient-based optimization algorithm, the resulting derivative does not result in the algorithm materially optimizing the parameters.

To show what goes wrong, we continue the derivation. In the next step of differentiating the risk, we follow the power rule and chain rule to produce:

$$D_\theta((h_\theta(x) - g(x))^2) = 2(h_\theta(x) - g(x)) D_\theta(h_\theta(x) - g(x)).$$

By linearity of the derivative and since g does not depend on θ, we have that $D_\theta(h_\theta(x) - g(x)) = D_\theta h_\theta(x)$. Standard AD ignores the parametric discontinuities in h_θ and makes the following step:

$$D_\theta h_\theta(x) \neq D_\mu \mathcal{N}(x; \mu, 5) \cdot [a \leq x \leq b].$$

On its own, the derivative above is correct at every point except $x = a$ and $x = b$, where the (standard AD) derivative of the program is zero, but is undefined in math. However, because the conditional is integrated over, the derivative is incorrect everywhere, not just at two points.

Automatic Differentiation in Potto. Figure 2e shows the result of applying Potto to calculate the gradient of `risk` to be used in gradient descent. During the optimization, the truncation points `a` and `b` widen and the mean of the truncated normal density shifts toward the normal density.

Using distribution theory [Teodorescu et al. 2013], we show that the derivative of the integral is the integral of the *parametric distributional derivative* of the integrand lifted to a *distribution* as long as it satisfies a mild integrability condition and a *transversality condition*. Informally, these conditions ensure the integral is well-defined and that the discontinuities do not coincide with each other, because in general, the product of distributions is not well-defined [Schwartz 1954]. For example, we can not have a product of two indicator functions $[x < c][x < d]$ where the discontinuities are equal ($c = d$).

Returning to our running example, the following equality holds by definition (Definition 3.8):

$$D_\theta \int_{-10}^{10} l(h_\theta(x), g(x)) \, \mathrm{d}x = \int_{-10}^{10} \partial_\theta T_l(h_\theta(x), g(x)) \, \mathrm{d}x,$$

where the operator T lifts l to a distribution and therefore has a parametric distributional derivative.

In the case of risk minimization, the squaring $[a \leq x \leq b]^2$ results in the parametric discontinuities coinciding and therefore not satisfying the transversality condition. However, we can rewrite the integrand so that there are no products of coincident discontinuities. Specifically, we expand the square and use the identity that the functions $[a \leq x \leq b]^2 = [a \leq x \leq b]$ are equal: $l(h_\theta(x), g(x)) = \mathcal{N}^2(x; \mu, 5)[a \leq x \leq b] - 2h_\theta(x)g(x) + g^2(x)$. Now that the integrand is written to satisfy the transversality condition, we can apply the product rule (Lemma 7.1).

We simplify the integrand to:

$$l(h_\theta(x), g(x)) = h_\theta(x)(\mathcal{N}(x; \mu, 5) - 2g(x)) + g^2(x) \tag{2}$$

and then lift it to a distribution. Since the transversality condition is satisfied, the rules for the parametric distributional derivatives match the rules of calculus, except for the derivative of discontinuous functions. We show this new derivative rule in detail.

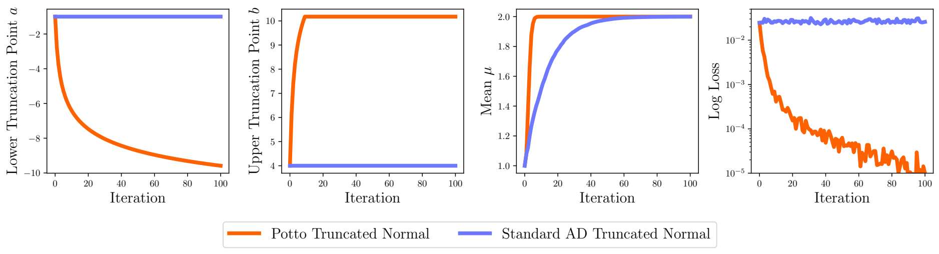


Fig. 3. The first two images show that standard AD (blue) has zero gradient for the truncation points, so they remain at their initialization $a = -1$ and $b = 4$. Potto (orange) accounts for the parametric discontinuities optimizing the truncation points to their optimal values $a = -10$ and $b = 10$. The third image shows that the mean converges to the optimum $\mu = 2$ about 10x faster for Potto than standard AD. The fourth image shows that the loss for Potto is orders of magnitude lower than Standard AD.

The parametric distributional derivative of $T_{h_\theta}(x)$ is:

$$\partial_\theta T_{h_\theta}(x) = \underbrace{(\partial_\mu T_{\mathcal{N}})(x;\mu,5)T_g(x;a,b)}_{1} + (\underbrace{\delta(x-a)\cdot D_a(x-a)}_{2} + \underbrace{\delta(b-x)\cdot D_b(b-x)}_{3})\cdot \underbrace{\mathcal{N}(x;\mu,5)}_{4},$$

where $g(x;a,b) = [a \le x \le b]$. The first term (1) is the same as in standard AD: the piecewise sum of the function's pieces. Introducing the three terms that follow makes the integral of the derivative correct. The product of terms (2) and (4) accounts for the parametric discontinuity at a and the product of terms (3) and (4) accounts for parametric discontinuity at b. The parametric distributional derivative introduces a *Dirac delta distribution*, $\delta(\cdot)$, for each differentiated parametric discontinuity. Informally, the Dirac delta distribution is an infinite spike when the conditional is true and zero everywhere else and satisfies the property that: $\partial_\theta([f(x,\theta) \ge 0]) = \delta(f(x,\theta))D_\theta(f(x,\theta))$.

The arrows in Figure 2a depict the directions of the derivative of the risk with respect to the parameters a (terms 2 and 4), b (terms 3 and 4), and μ (term 1) that Potto computes. We have that $\partial_a([x \ge a]) = \partial_a([x - a \ge 0]) = \delta(x-a)D_a(x-a)$. The derivative $D_a(x-a) = -1$. As a result, $\partial_a h_\theta(x) = -\delta(x-a)\mathcal{N}(x;\mu,5)$. Putting these results together, we have that:

$$\int_{-10}^{10} \partial_a T_{h_\theta}(x)\,\mathrm{d}x = \int_{-10}^{10} -\delta(x-a)\mathcal{N}(x;\mu,5)\,\mathrm{d}x = -\mathcal{N}(a;\mu,5)\cdot[-10 \le a \le 10], \tag{3}$$

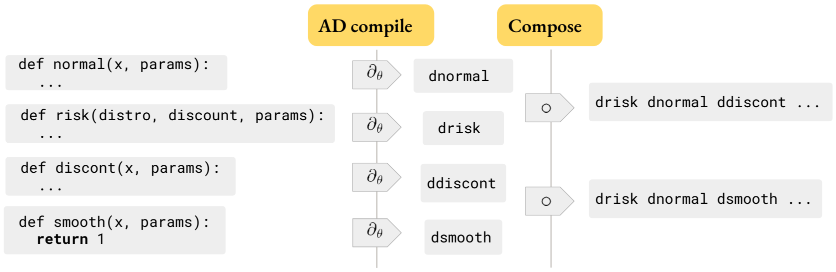
where the last equality follows from the *sifting property* (Teodorescu et al. [2013] Equation 1.47):

$$\int_{-x_l}^{x_u} \delta(x-c)f(x)\,\mathrm{d}x = f(c)[x_l \le c \le x_u]. \tag{4}$$

We conclude that an infinitesimal perturbation to the parameter a shifts it to the right, decreasing the area under the truncated normal by $\mathcal{N}(a;\mu,5)$.

Standard AD versus Potto Results. Standard AD and Potto use *Monte Carlo integration* to estimate the integral in the derivative of the risk. Both systems average the evaluation of the integrand at 50 uniformly random samples from $[-10,10]$. Potto implements the sifting property by sampling and evaluating at the points at the parametric discontinuities.

Figure 3 depicts the changes in each parameter using standard AD. The derivative at the truncation points is zero for standard AD (blue), so the loss remains nearly constant because only `mu` is optimized. Potto (orange) more rapidly approaches the minimum and achieves a loss that is over three orders of magnitude lower than the loss from standard AD.



```
def normal(x, params):
  ...
def risk(distro, discount, params):
  ...
def discont(x, params):
  ...
def smooth(x, params):
  return 1
```

Fig. 4. Potto supports separate compilation, which enables the reuse of the derivative of risk.

2.2 Separate Compilation

The Potto prototype implementation supports differentiation of parametric discontinuities and *separate compilation*. Previous work, Teg [Bangaru et al. 2021] supported the differentiation of parametric discontinuities, but not separate compilation.

Problem as a Program. In Potto, we can implement risk minimization as follows:

```
1  # functions.po
2  def normal(x: Var, mu: Param):
3    return exp(-0.5 * ((x - mu) / 5)^2)
4  def discont(x: Var, a: Param, b: Param):
5    return 1 if a <= x <= b else 0
```

The type declaration `x: Var` in Lines 2 and 4 specify that `x` is a variable of integration and similarly, parameters μ, `a`, and `b` are declared with type **`Param`**. Lines 2-3 declare a function representing the normal density, which is a bell-shaped curve, where the peak is at μ and the width is controlled by σ, modeled by the equation $e^{-\frac{1}{2}(\frac{x-\mu}{\sigma})^2}$. In Line 3, the parameter μ is `mu` and σ is 5. Lines 4-5 declare a function modeled by $[a \leq x \leq b]$ with parametric discontinuities at a and b.

```
1  # risk.po
2  from functions import normal
3  def risk(distro, discont, a: Param, b: Param, mu: Param):
4    return integral ([-10, 10]) (let pdf = distro x mu in let target = normal x 2
5      in pdf*(discont x a b)*(pdf-2*target)+target^2) dx
```

The function defined in `risk.po` implements the risk function specified by Equation 1 with the integrand implemented as in Equation 2. Lines 4-5 define an integral representing the area under the curve defined by the integrand from `x=-10` to `10`. The first argument to the **`integral`** specifies the domain of integration, the second argument is the integrand, and the third argument **`d`**`x` declares the variable `x`. The let bindings define a probability density function, `pdf`, depending on a parameter `mu` and the target density `target` as the normal density centered at 2.

```
1  # main.po
2  from functions import_deriv normal, discont as dnormal, ddiscont
3  params: list[Param] = [-1, 4, 1]
4  step_size = 400
5  for i in range(params):
6    (a, b, mu), (da, db, dmu) = params, one_hot(i, 3)
7    params[i] -= step_size*(drisk dnormal ddiscont (a, da) (b, db) (mu, dmu))[1]
```

The `main.po` file does a single step of gradient descent for the parameters of the truncated normal density. Line 2 imports the derivative of the normal and truncated normal densities from `densities.po`. The **`import_deriv`** specifies the functions to differentiate and **`as`** gives a name to the derivative.

Line 6 unpacks the parameters in `params` and declares new parameters. The `one_hot(i, n)` command returns a list of length n that is one at index i and zero otherwise. The decrement on Line 7 implements gradient descent with a step size of 400. For example, to differentiate with respect to the parameter `a`, we set the infinitesimals `da, db, dmu` to `one_hot(0, 3)`, which is `(1, 0, 0)`. Since `drisk` returns a pair of the evaluation of risk and its derivative, we extract the derivative by indexing.

Teg. Teg [Bangaru et al. 2021] is implemented as a series of rewrites that are applied exhaustively to a given program. For consistency, we use a similar syntax to represent Teg programs as we did for Potto programs.

For example, a Teg program that arises in the computation of risk minimization is:

```
integral ([-10, 10]) -delta(x - a)*normal(x, mu) dx,
```

which matches up with the left-hand side of Equation 3 and denotes that expression. The **`delta`** syntax denotes the Dirac delta distribution. To evaluate this expression, Teg applies a rewrite rule that eliminates Dirac deltas by applying the sifting property (Equation 4), producing the program:[3]

```
-normal(a, mu) if -10 < a < 10 else normal(a, mu),
```

which matches the right-hand side Equation 3. Teg evaluates the integral-free and delta-free program to a number, in the same way any other compiler would.

Potto. Potto collects samples from the integral and the derivative of the discontinuities in the integrand (i.e. the Dirac deltas) and then evaluates the program at a given sample for each file (`densities.po` and `risk.po`).

In the example above, Potto samples the point `x=a`. Potto makes no changes to the code and evaluates the integrand directly at `x=a` as follows. First, Potto accepts the sample if it is inside the range $[-10, 10]$ and rejects the sample otherwise. If accepted, Potto then evaluates the delta at `a` and returns one because `a - a` equals zero (otherwise it returns zero). Potto multiplies one by the `normal` density evaluated at `a`, producing the same result as Teg.

Separate Compilation. In risk minimization, different approximating densities change the quality of results. Figure 4 shows how a user can replace the truncated normal density with a normal density and compose both it (and its derivative) with the risk function (and its derivative). Ideally, the user should be able to interchange the approximating densities by replacing the density (and its derivative) rather than differentiating the risk function again.

Support for *separate compilation* achieves this by enabling the division of a program into distinct source files, differentiating each individually, and composing the derivatives together to form an executable. This process enables efficient code reuse. The challenge is to support separate compilation of derivatives for programs with parametric discontinuities.

Potto supports separate compilation by directly sampling discontinuities and evaluating the program at a given sample for each file. Because Teg relies on a rewrite rule to apply the sifting property, it requires that code for the integral and integrand are within the same file, preventing separate compilation.

Results. We present timing results in our prototype implementation of Potto. Differentiating the risk separately from the normal density and truncated normal density takes 50% less time than differentiating the composition of the risk function with the normal density and with the truncated normal density (as averaged over three runs on an Intel i9-9980HK).

[3]This is a simplification of the actual Teg rewrite rules. Teg applies a change of variables `y=x-a` and then applies the sifting property to the resulting program. However, Teg could easily be extended to support this implementation.

Techniques from Teg [Bangaru et al. 2021] can be used in addition to Potto as a static analysis that eliminates Dirac deltas when the code for an integral and integrand are within the same file. However, Potto significantly improves compile time and tends to be slower in runtime on smaller programs and faster on larger programs (with more discontinuities and more complex discontinuities, i.e., a larger expression in the condition).

For risk minimization, we find that Potto (with separate compilation) compiles about 377x faster than Teg, and that the runtime of Potto is about 27x slower than Teg. On workloads in computer graphics (Section 10), we find that compilation time is a barrier to common workflows and that Potto has better runtime performance than Teg on larger programs (see Figure 15).

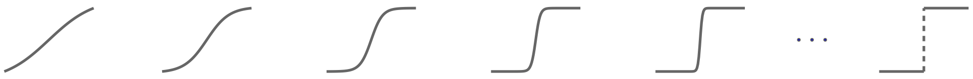
3 DISTRIBUTIONS

We now provide a brief mathematical introduction to *distributions* (Teodorescu et al. [2013] Definition 1.21) and *distributional derivatives* (Teodorescu et al. [2013] Equation 1.184).

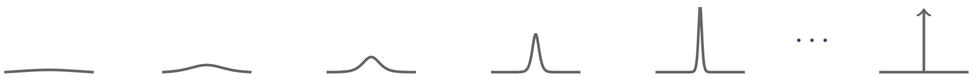
The derivative of a discontinuous function does not exist in calculus. Distributions are a generalization of functions that give meaning to derivatives of discontinuous functions.

We present an example of differentiating the Heaviside step function $H(x) = [x \geq 0]$, which has a jump discontinuity at $x = 0$. One way to approach this problem is to note that if a sequence of smooth functions $f_n(x)$ converges to a smooth function $f(x)$ as n goes to infinity, then the sequence of derivatives of $D_x f_n(x)$ converges to the derivative $D_x f(x)$ as n goes to infinity.[4] We know that this reasoning will fail for $H(x)$ because it is discontinuous, but we will see what happens anyway.

The sequence of smooth functions defined by $H_n(x) = \frac{1}{1+e^{-nx}}$ converges to $H(x)$:



The derivative of $H_n(x)$ is $D_x H_n(x) = \delta_n(x) = \frac{ne^{-nx}}{(1+e^{-nx})^2}$ and let us see what it converges to:



The arrow in the rightmost figure indicates an infinite spike at $x = 0$. This sequence diverges, since there is no real-valued function that represents the limit of an infinite spike at x=0

This is similar to how a convergent sequence of rational numbers may not converge to a rational number (e.g., Leibniz formula for π is a sequence $S_n = \sum_{k=0}^{n} \frac{(-1)^k}{2k+1}$ that approaches $\frac{\pi}{4}$). However, every convergent sequence approaches a real number. Similar to how real numbers generalize rational numbers, distributions generalize functions. In the case above, the sequence $\delta_n(x)$ converges to the *Dirac delta distribution* $\delta(x)$.

The Dirac Delta Distribution. Informally, $\delta(x)$ is infinite at $x = 0$ and 0 everywhere else. Additionally, the integral over the real line of each $\delta_n(x)$ and $\delta(x)$ is one. We can formalize the Dirac delta as a distribution using the *sifting property*: for every *test function* ϕ,

$$\int_{\mathbb{R}} \delta(x)\phi(x)\,\mathrm{d}x := \phi(0). \tag{5}$$

Formalism. A *distribution* $u \in \mathcal{D}'(\mathbb{R}^n)$ maps *test functions* $\phi \in \mathcal{D}(\mathbb{R}^n)$ to real numbers, where we write $u[\phi] =: \int_{\mathbb{R}^n} u(x)\phi(x)\,\mathrm{d}x$. To define test functions, we need the following definition.

[4] Weak convergence is defined as $\lim_{k\to\infty} \int_{\mathbb{R}^n} f_k(x)\phi(x)\,\mathrm{d}x = \int_{\mathbb{R}^n} f(x)\phi(x)\,\mathrm{d}x$ for every smooth function with compact support ϕ (Teodorescu et al. [2013], Definition 1.22).

Definition 3.1. A function $f : \mathbb{R}^n \to \mathbb{R}$ has *compact support* if it is 0 outside a closed and bounded set. △

For example, $f(x) = [-1 < x < 1]$ has compact support (it is 0 everywhere outside $[-1, 1]$), but $f(x) = 1$ does not have compact support.

Definition 3.2. A function $\phi : \mathbb{R}^n \to \mathbb{R}$ is a *test function* $\phi \in \mathcal{D}(\mathbb{R}^n)$ if it is smooth (infinitely differentiable) and has compact support. △

For example, the constant function $\phi(x) = 0$ and the bump function $\phi(x) = [-1 < x < 1]e^{-\frac{1}{1-x^2}}$ are test functions. On the other hand, $f(x) = [-1 < x < 1]$ is not smooth and $f(x) = 1$ does not have compact support, so both are not test functions.

Distributions are designed to be a generalization of functions where the regular rules of calculus—such as integration by parts, changes of variables, and derivatives—still hold.

Definition 3.3. A *distribution* $u \in \mathcal{D}'(\mathbb{R}^n)$ is a *continuous linear functional* on the space of test functions $\mathcal{D}(\mathbb{R}^n)$. Concretely, a distribution $u \in \mathcal{D}'(\mathbb{R}^n)$ if it[5]:

(1) is *continuous*: if a sequence $(\phi_i)_{i\geq 1} \in \mathcal{D}(\mathbb{R}^n)$ approaches $\phi \in \mathcal{D}(\mathbb{R}^n)$ in $\mathcal{D}(\mathbb{R}^n)$ as $i \to \infty$ then the sequence of integrals $\left(\int_{\mathbb{R}^n} u(x)\phi_i(x)\,\mathrm{d}x\right)_{i\geq 1}$ approaches $\int_{\mathbb{R}^n} u(x)\phi(x)dx$ as $i \to \infty$.
(2) is *linear*: for every $a, b \in \mathbb{R}$ and $\phi_1, \phi_2 \in \mathcal{D}(\mathbb{R}^n)$ we have $\int_{\mathbb{R}^n} u(x)(a\phi_1(x) + b\phi_2(x))\,\mathrm{d}x = a\int_{\mathbb{R}^n} u(x)\phi_1(x)\,\mathrm{d}x + b\int_{\mathbb{R}^n} u(x)\phi_2(x)\,\mathrm{d}x$.
(3) is a *functional*: for every $\phi \in \mathcal{D}(\mathbb{R}^n)$ we have $\int_{\mathbb{R}^n} u(x)\phi(x)\,\mathrm{d}x \in \mathbb{R}$. △

We define changes of variables in terms of diffeomorphisms. Figure 5 depicts an example of a diffeomorphism $\hat{\Psi}$ that transforms a 2D grid using the sigmoid function.

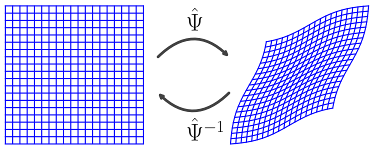


Fig. 5. The depicted diffeomorphism $\hat{\Psi}$ tranforms a 2D grid using the sigmoid function and has inverse $\hat{\Psi}^{-1}$ (adapted from del Toro Streb and Alexandrov [2009]).

Definition 3.4. A C^k*-diffeomorphism* $\hat{\Psi} : \mathbb{R}^n \to \mathbb{R}^n$ is a k-times differentiable, invertible function. △

Every C^k-diffeomorphism $\hat{\Psi}$ (with $k \geq 1$) satisfies the change of variable formula:

$$\int_{\mathbb{R}^n} u(\hat{\Psi}(x))\phi(x)\,\mathrm{d}x = \int_{\mathbb{R}^n} u(y)\frac{\phi(\hat{\Psi}^{-1}(y))}{\det|D\hat{\Psi}(y)|}\,\mathrm{d}y \qquad (6)$$

for every $u \in \mathcal{D}'(\mathbb{R}^n)$ and $\phi \in \mathcal{D}(\mathbb{R}^n)$. Examples of C^∞-diffeomorphisms are $\hat{\Psi}(x) = x+1$, which has inverse $\hat{\Psi}^{-1}(y) = y-1$ and $\hat{\Psi}(x, y) = (x+y, x-y)$, which has inverse $\hat{\Psi}^{-1}(w, z) = (\frac{w+z}{2}, \frac{w-z}{2})$. Some functions that are not diffeomorphisms are $f(x, y) = x+y$ because it is not invertible (it has signature $\mathbb{R}^2 \to \mathbb{R}$ rather than $\mathbb{R}^2 \to \mathbb{R}^2$) and $f(x) = x^2$ because it is not one-to-one. A diffeomorphism can depend on a parameter. For example, $\hat{\Psi}_a(x) = a - x$ has an inverse $\hat{\Psi}_a^{-1}(y) = a - y$ for every $a \in \mathbb{R}$.

Definition 3.5. The *distributional derivative* ∂_x of $u \in \mathcal{D}'(\mathbb{R}^n)$ satisfies:

$$\int_{\mathbb{R}^n} \partial_x u(x)\phi(x)\,\mathrm{d}x := -\int_{\mathbb{R}^n} u(x)D_x\phi(x)\,\mathrm{d}x \qquad (\forall\phi \in \mathcal{D}(\mathbb{R}^n)). \qquad (7)$$

△

[5]We follow Teodorescu et al. [2013] Definition 1.21, which uses a definition of continuity that is generally termed sequential continuity, but is sufficient for our purposes.

This definition is inspired by integration by parts, where the boundary term is 0 because ϕ has compact support.

We want to be able to lift a function to a distribution so that it can then be differentiated. *Locally integrable* functions can be lifted to distributions.

Definition 3.6. A function is *locally integrable* if it is integrable on every compact set of its domain. △

Every integrable function (f such that $\int_{\mathbb{R}^n} |f(x)|\, \mathrm{d}x < \infty$) is locally integrable, but not every locally integrable function is integrable. For example, for every compact set K the integral $\int_K 1\, \mathrm{d}x$ is finite, but $\int_{\mathbb{R}^n} 1\, \mathrm{d}x$ is infinite. So the $f(x) = 1$ is locally integrable, but not integrable. Every probability density function (PDF) defined over the reals (e.g., the uniform PDF and the normal PDF) is integrable because it integrate to 1 and therefore locally integrable.[6]

Definition 3.7. A *regular distribution* T_f is a distribution that is equal to the (Lebesgue) integral over a locally integrable function $f(x)$:

$$\int_{\mathbb{R}^n} T_f(x)\phi(x)\, \mathrm{d}x := \int_{\mathbb{R}^n} f(x)\phi(x)\, \mathrm{d}x \qquad (\forall \phi \in \mathcal{D}(\mathbb{R}^n)). \tag{8}$$

△

The Dirac delta is not locally integrable because it is not a real-valued function and it is not a regular distribution because no locally integrable function satisfies the sifting property (Teodorescu et al. [2013] page 19).

Derivative of Heaviside is Delta. We now revisit the example of the Heaviside step function $H(x) = [x \geq 0]$. It is not differentiable at $x = 0$, but it is locally integrable and can be lifted to a regular distribution T_H with the distributional derivative:

$$\int_{\mathbb{R}} \partial_x T_H(x)\phi(x)\, \mathrm{d}x \overset{(7)}{=} -\int_{\mathbb{R}} T_H(x) D_x\phi(x)\, \mathrm{d}x \overset{(8)}{=} -\int_{\mathbb{R}} H(x) D_x\phi(x)\, \mathrm{d}x$$

Continuing, we apply the definition of the Heaviside step function, and then simplify the integral using the fact that ϕ has compact support:

$$-\int_{\mathbb{R}} H(x) D_x\phi(x)\, \mathrm{d}x = -\int_{[0,\infty]} D_x\phi(x)\, \mathrm{d}x = -(\phi(\infty) - \phi(0)) = \phi(0) \overset{(5)}{=} \int_{\mathbb{R}} \delta(x)\phi(x)\, \mathrm{d}x.$$

We can thus conclude that: $\partial_x T_H(x) = \delta(x)$.

Our Desiderata. We need a notion of derivative that applies to discontinuous integrands. Furthermore, classical theory severely limits products of distributions [Schwartz 1954], preventing $T_H(x)\delta(x)$ from being a valid distribution.

For example, the distributional derivative of a program with nested conditionals and coincident conditions such as `(1 if x > a else 0) if x > a else 0` may not satisfy the product rule.[7] Our theory provides a sufficient condition for the product rule to hold (See Appendix B.1). For example, the derivative of `(1 if x > a else 0) if x > a + 1 else 0` satisfies the product rule because the conditions (`x > a + 1` and `x > a`) are distinct as the case for numerically stable programs (see Section 7.3.2).

[6]This result is important because it means probabilistic programs interpreted as integrators [Shan and Ramsey 2017] also define distributions. Note that while all Radon measures as integrators define distributions, not all distributions denote Radon measures (e.g. $\partial_x \delta$).

[7]We expect that $T_H(x-a) \cdot T_H(x-a) = T_H(x-a)$. The derivative of both sides is $2\delta(x-a)H(x-a) = \delta(x-a)$, which means $H(0) = 1/2$. However, repeating this for $T_H(x-a)^3 = T_H(x-a)$ gives $H(0) = 1/3$, which is a contradiction.

Our Solution. We resolve these problems by formalizing *parametric distributions* and *parametric distributional derivatives.* A *parametric distribution* $u_\theta : \mathbb{R}^m \to \mathcal{D}'(\mathbb{R}^n)$ is a family of distributions over free parameters $\theta \in \mathbb{R}^m$.

Definition 3.8. The *parametric distributional derivative* ∂_θ of a parametric distribution u_θ is:

$$\int_{\mathbb{R}^n} \partial_\theta u_\theta(x)\phi(x)\,\mathrm{d}x := D_\theta \int_{\mathbb{R}^n} u_\theta(x)\phi(x)\,\mathrm{d}x \qquad (\forall \phi \in \mathcal{D}(\mathbb{R}^n)).$$

as long as the derivative of the integral on the right-hand side exists. △

4 SYNTAX

In this section, we introduce syntax for the core language, Langur, for implementing Potto programs. Langur is a *first-order language*. All terms are of base type **real** and Langur has first-order let-bindings. We describe how the surface language maps to the core language at the end of Section 9.

The grammar for Langur terms is:

$$\begin{aligned} t, s, r &::= c \mid x \mid z \mid t + s \mid t \cdot s \mid \textbf{ifge0}\ t\ \textbf{then}\ s\ \textbf{else}\ r \\ &\quad\mid \textbf{ifge0}\ \lfloor\Psi\rfloor\ \textbf{then}\ s\ \textbf{else}\ r \mid \textbf{let}\ z = t\ \textbf{in}\ s \\ p &::= \textbf{int}\ t\ \mathbf{d}(x_1, \ldots, x_n) \qquad x_1, \ldots, x_n \in \textsc{Vars} \quad z \in \textsc{Params} \quad \textsc{Vars} \cap \textsc{Params} = \emptyset. \end{aligned}$$

We use the metavariables t, s, and r for terms and the metavariable p for programs.

4.1 Terms

We now briefly describe the syntax of Langur terms.

Arithmetic Primitives. Langur has standard arithmetic primitives: constants c, variables $x_1, \ldots, x_n \in$ Vars, parameters $z \in$ Params where z is a metavariable and arithmetic operators $+, \cdot$.

Conditionals. Conditionals **ifge0** t **then** s **else** r are such that t must be variable-free (it can contain parameters z) and have $t \geq 0$ implicitly (in OCaml, **if** $t \geq 0$ **then** s **else** r). For example, in the case study, there is one variable, x, and there are three free parameters a, b, and μ. The condition t must be free of variables, preventing parametric discontinuities.

Diffeomorphic Conditionals. Diffeomorphic conditionals, **ifge0** $\lfloor\Psi\rfloor$ **then** s **else** r, have differentiable, invertible conditions, $\lfloor\Psi\rfloor$, defined outside the core language. For example, the truncation points $a \leq x \leq b$ in Line 6 of `distributions.po` are specified by nesting the diffeomorphic conditionals: $\hat{\Psi}_1(x, a) = x - a$ and $\hat{\Psi}_2(x, b) = b - x$ because if $x - a \geq 0$ then $a \leq x$ and if $b - x \geq 0$ then $x \leq b$. The arguments to Ψ are all n variables and m free parameters (we make this choice to simplify the presentation and could easily modify the language so that the number of variables could range from 1 to n and the number of parameters could range from 1 to m).

4.2 Programs

The term **int** t $\mathbf{d}(x_1, \ldots, x_n)$ represents integration of a term t over $\mathbb{R}^n$ with respect to variables $x_1, \ldots, x_n$. Programs can be differentiated with respect to parameters $z \in$ Params.

5 TYPE SYSTEM

Figure 6 presents a type system that characterizes when well-formed terms denote real functions.

5.1 Types and Type Contexts

Langur has **real** as its only type. Langur lacks arrow types because it is a first-order language.

$$\frac{}{\Delta;\Gamma \vdash c : \textbf{real}} \qquad \frac{x : \textbf{real} \in \Delta}{\Delta;\Gamma \vdash x : \textbf{real}} \qquad \frac{z : \textbf{real} \in \Gamma}{\Delta;\Gamma \vdash z : \textbf{real}} \qquad \frac{\Delta;\Gamma \vdash t : \textbf{real} \quad \Delta;\Gamma \vdash s : \textbf{real}}{\Delta;\Gamma \vdash t + s : \textbf{real}}$$

$$\frac{\Delta;\Gamma \vdash t : \textbf{real} \quad \Delta;\Gamma \vdash s : \textbf{real}}{\Delta;\Gamma \vdash t \cdot s : \textbf{real}} \qquad \frac{\cdot;\Gamma \vdash t : \textbf{real} \quad \Delta;\Gamma \vdash s : \textbf{real} \quad \Delta;\Gamma \vdash r : \textbf{real}}{\Delta;\Gamma \vdash (\textbf{ifge0}\ t\ \textbf{then}\ s\ \textbf{else}\ r) : \textbf{real}}$$

$$\frac{\Psi : [\![\Delta]\!] \times [\![\Gamma]\!] \overset{\text{diffeo}}{\to} [\![\Delta]\!] \quad \Delta;\Gamma \vdash s : \textbf{real} \quad \Delta;\Gamma \vdash r : \textbf{real}}{\Delta;\Gamma \vdash (\textbf{ifge0}\ \lfloor\Psi\rfloor\ \textbf{then}\ s\ \textbf{else}\ r) : \textbf{real}} \qquad \frac{\cdot;\Gamma \vdash t : \textbf{real} \quad \Delta;\Gamma, z : \textbf{real} \vdash s : \textbf{real}}{\Delta;\Gamma \vdash \textbf{let}\ z = t\ \textbf{in}\ s : \textbf{real}}$$

Fig. 6. The type system specifying well-formed terms.

The typing judgments are expressed in terms of two type contexts: one for variables Δ and one for parameters Γ. We distinguish contexts for variables so that we can easily state when a statement does or does not contain variables. A type context is a mapping from variable names to types:

$$\Delta ::= \cdot \mid x : \textbf{real}, \Delta \qquad \Gamma ::= \cdot \mid z : \textbf{real}, \Gamma$$

The contexts Δ and Γ are disjoint so that variables and parameters do not overlap. We use syntactic sugar $x : \textbf{real} := x : \textbf{real}, \cdot$ for simplicity.

5.2 Terms

The typing judgment $\Delta;\Gamma \vdash t : \textbf{real}$ indicates that the term t is of type **real** given that each of the variables in Δ and parameters in Γ are of type **real**.

Arithmetic Primitives. The typing rules for arithmetic primitives operate over **real**s and are standard. For both sums and products if the arguments are **real** then the result is **real**.

Conditionals. The typing rule for conditionals **ifge0** t **then** s **else** r states that a conditional is of type **real** if the condition term t is **real** and variable-free and if the terms s and r are **real**.

Diffeomorphic Conditionals. To have variables arise in the conditionals, we have to use diffeomorphic conditionals **ifge0** $\lfloor\Psi\rfloor$ **then** s **else** r. Using the isomorphism between variable names and ordered inputs $\vec{x} \mapsto \hat{\Psi}(\vec{x}, \vec{z})$ is a C^1*-diffeomorphism* as in Definition 3.4.

The programmer defines a diffeomorphism in the surface language, Potto, as a pair of tuples, where the first tuple specifies Potto code to compute the diffeomorphism and the second tuple specifies its inverse. Future work could automate (piecewise) inversion, which is a long-standing and challenging problem studied in both the programming languages and graphics communities [Anderson et al. 2017; Lutz 1986; Matsuda and Wang 2020].

Let Bindings. The **let** $z = t$ **in** s primitive introduces a fresh variable z into the typing context Γ. The term t must be free of variables. For example, the type system accepts **let** $z = z_1 + z_2$ **in** $z + z$ if $z_1 : \textbf{real}, z_2 : \textbf{real} \in \Gamma$ and rejects **let** $z = x$ **in** z.

6 DENOTATIONAL SEMANTICS OF TERMS

The denotational semantics, $[\![t]\!](\rho, \gamma)$, maps a well-typed term $\Delta;\Gamma \vdash t : \textbf{real}$ to a *simply decomposable function.* Informally, the class of simply decomposable functions are piecewise-differentiable functions with finitely many piecewise-invertible discontinuities. Using the isomorphism between mappings from variable or parameter names to real numbers $[\![\Delta]\!] \cong \mathbb{R}^n$ and $[\![\Gamma]\!] \cong \mathbb{R}^m$, where the sizes of the contexts $|\Delta| = n$ and $|\Gamma| = m$, we can write $\widehat{[\![t]\!]} : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}$. The following semantics are standard and serve as scaffolding for the following section.

$$[\![t]\!] : [\![\Delta]\!] \times [\![\Gamma]\!] \to \mathbb{R} \qquad [\![c]\!](\rho,\gamma) = c \qquad [\![x]\!](\rho,\gamma) = \rho(x) \qquad [\![z]\!](\rho,\gamma) = \gamma(z)$$

$$[\![t + s]\!](\rho,\gamma) = [\![t]\!](\rho,\gamma) + [\![s]\!](\rho,\gamma) \qquad [\![t \cdot s]\!](\rho,\gamma) = [\![t]\!](\rho,\gamma) \cdot [\![s]\!](\rho,\gamma)$$

$$[\![\textbf{ifge0 } t \textbf{ then } s \textbf{ else } r]\!](\rho,\gamma) = \begin{cases} [\![s]\!](\rho,\gamma) & \text{if } [\![t]\!](\cdot,\gamma) \geq 0 \\ [\![r]\!](\rho,\gamma) & \text{otherwise} \end{cases}$$

$$[\![\textbf{ifge0 } \lfloor\Psi\rfloor \textbf{ then } s \textbf{ else } r]\!](\rho,\gamma) = \begin{cases} [\![s]\!](\rho,\gamma) & \text{if } \pi_0\Psi(\rho,\gamma) \geq 0 \\ [\![r]\!](\rho,\gamma) & \text{otherwise} \end{cases}$$

$$[\![\textbf{let } z = t \textbf{ in } s]\!](\rho,\gamma) = [\![s]\!](\rho,\gamma[z \mapsto [\![t]\!](\cdot,\gamma)]) \qquad \mathbb{R}^n \cong \textsc{Vars} \to \mathbb{R} \qquad \mathbb{R}^m \cong \textsc{Params} \to \mathbb{R}$$

Fig. 7. The denotational semantics $[\![t]\!](\rho,\gamma)$ for terms.

6.1 Types, Type Contexts, and Value Contexts

The only type in Langur is **real**, and it denotes real numbers: $[\![\textbf{real}]\!] = \mathbb{R}$. Type contexts Γ and Δ denote mappings to real numbers and are defined by:

$$[\![\cdot]\!] = \{\{\}\} \qquad [\![x : \textbf{real}, \Delta]\!] = (x \mapsto [\![\textbf{real}]\!]) \sqcup [\![\Delta]\!] \qquad [\![z : \textbf{real}, \Gamma]\!] = (z \mapsto [\![\textbf{real}]\!]) \sqcup [\![\Gamma]\!]$$

The empty context denotes the empty function and the denotation of a non-empty context denotes a disjoint union of functions.

The value context $\rho : \textsc{Vars} \to \mathbb{R}$ maps from variable names to real numbers. The value context $\gamma : \textsc{Params} \to \mathbb{R}$ maps from parameters to real numbers. We say that $\rho \in [\![\Delta]\!]$ if for every x in the domain of Δ, we have $\rho(x) \in [\![\Delta(x)]\!]$. In words, for every variable in the typing context, its value is an element of its type. We define $\gamma \in [\![\Gamma]\!]$ analagously.

6.2 Terms

Now that we have defined the denotation of each type, we give a denotation to each term in Figure 7. The denotation of a term $\Delta; \Gamma \vdash t : \textbf{real}$ is a function from the denotation of the free variables $[\![\Delta]\!] \times [\![\Gamma]\!]$ to the denotation of the type $[\![\textbf{real}]\!]$. As a result, we provide inputs ρ, γ to the denotation of a term, where $\rho \in [\![\Delta]\!]$ and $\gamma \in [\![\Gamma]\!]$.

We discuss only notable cases.

Arithmetic Primitives. For convenience, we define syntactic sugar for subtraction of terms using multiplication and addition: $[\![t_1 - t_2]\!](\rho,\gamma) := [\![t_1]\!](\rho,\gamma) + (-1) \cdot [\![t_2]\!](\rho,\gamma)$.

Conditionals. For conditionals **ifge0** t **then** s **else** r, variables are not allowed in the condition t to prevent parametric discontinuities.

Diffeomorphic Conditionals. For the conditional **ifge0** $\lfloor\Psi\rfloor$ **then** s **else** r, the first dimension $\pi_0\Psi : [\![\Delta]\!] \times [\![\Gamma]\!] \to \mathbb{R}$ corresponds to the condition, the other dimensions serve to make Ψ a diffeomorphism (Definition 3.4). We require that the condition is a diffeomorphism in order to give meaning to the derivative.

Let Bindings. The denotation of a let binding **let** $z = t$ **in** s is the denotation of s with the variable z bound to the denotation of t.

6.3 Results

In this section, we prove results that serve primarily as a scaffolding for results in the following sections. We show that the denotation of a term is a *simply decomposable function.*

Definition 6.1. A function $f : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}$ is a C^k *simple discontinuity* if $f(\vec{x}, \vec{z}) = [\Phi(\vec{x}, \vec{z}) \geq 0]$, where either (1) $\Phi(\vec{x}, \vec{z})$ is constant in $\vec{x}$ or (2) $\Phi(\vec{x}, \vec{z}) = \pi_0 \hat{\Psi}(\vec{x}, \vec{z})$, where $\vec{x} \mapsto \hat{\Psi}(\vec{x}, \vec{z})$ is a C^k diffeomorphism for all $\vec{z}$. △

Definition 6.2. A function $g : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}$ is C^k *simply decomposable* if $g(\vec{x}, \vec{z})$ is a finite sum of a finite product of C^k simple discontinuities times a k-times differentiable function in $\vec{z}$. Concretely,

$$g(\vec{x}, \vec{z}) = \sum_{i \in I} \left(\prod_{j \in J_i} [\Phi_{ij}(\vec{x}, \vec{z}) \geq 0] \right) g_i(\vec{x}, \vec{z})$$

where I and all J_i are sets of finitely many indices, each g_i is differentiable in $\vec{z}$, and each $[\Phi_{ij}(\vec{x}, \vec{z}) \geq 0]$ is a C^k simple discontinuity. △

For simplicity, we call C^1 simply decomposable functions just simply decomposable. For example, $[\sin(x) \geq 0]$ is not simply decomposable because $\sin(x)$ is not invertible and cannot be decomposed into finitely many invertible pieces.

Theorem 6.1. *Every well-typed term* $\Delta; \Gamma \vdash t : \textbf{\textit{real}}$ *denotes a simply decomposable function* $[\![t]\!]$. «

Proof. We provide a full proof in Appendix H. □

7 DENOTATIONAL SEMANTICS OF DERIVATIVES OF TERMS

A simply decomposable function can be lifted to a *regular distribution* as defined in Equation 8. While simply decomposable functions $[\![t]\!]$ do not necessarily have a well-defined derivative, they do have a well-defined *distributional derivative* $(\!|t|\!)$.

7.1 Types, Type Contexts, and Value Contexts

Types now denote distributions $(\!|\textbf{real}|\!) = \mathcal{D}'(\mathbb{R})$. The denotation of the typing context for variables remains the same: $(\!|\Delta|\!) = [\![\Delta]\!]$. The typing context for parameters, Γ, has new bindings for the infinitesimal perturbation:

$$(\!|\cdot|\!) = \{\{\}\} \qquad\qquad (\!|z : \textbf{real}, \Gamma|\!) = \{z \mapsto [\![\textbf{real}]\!]\} \sqcup \{z' \mapsto (\!|\textbf{real}|\!)\} \sqcup (\!|\Gamma'|\!),$$

where the parameter z is a real number and the infinitesimal perturbation z' is a distribution.

The meaning of the derivative of the empty context is still the empty context. In a nonempty context, there are twice as many parameters because each parameter has an associated infinitesimal. For instance, the total derivative of $f(x, y) = xy$ is $Df(x, y, x', y') = x'y + xy'$ with infinitesimals x', y'. We can then, for example, recover the partial derivative D_x by setting $x' = 1$ and $y' = 0$.

The value contexts ρ and γ remain as before, but γ' is a mapping from parameters to distributions representing the values of the infinitesimal perturbations to the parameters.

7.2 Terms

Figure 8 presents the denotational semantics for derivatives of terms as distributions $(\!|t|\!)(\rho, \gamma, \gamma') \in \mathcal{D}'(\mathbb{R}^n)$, where $\rho \in (\!|\Delta|\!)$, the real values for parameters, γ, and the distributions for infinitesimal perturbations, γ', are such that $(\gamma, \gamma') \in (\!|\Gamma|\!)$, and the number of variables $n = |\Delta|$. We write the parametric distributional derivative with respect to γ as ∂_γ.

Arithmetic Primitives. The derivative denotation of a sum is the sum of the derivative denotations. The denotation of the derivative of a product is the denotation of the derivative of the first term times the denotation of the second term lifted to a distribution plus the derivative of the second term times the denotation of the first term lifted to a distribution.

$$(\!|t|\!) : (\!|\Delta|\!) \times (\!|\Gamma|\!) \to \mathcal{D}'(\mathbb{R}^n) \qquad (\!|c|\!)(\rho,\gamma,\gamma') = 0 \qquad (\!|x|\!)(\rho,\gamma,\gamma') = 0$$

$$(\!|z|\!)(\rho,\gamma,\gamma') = \gamma'(z) \qquad (\!|t+s|\!)(\rho,\gamma,\gamma') = (\!|t|\!)(\rho,\gamma,\gamma') + (\!|s|\!)(\rho,\gamma,\gamma')$$

$$(\!|t \cdot s|\!)(\rho,\gamma,\gamma') = (\!|t|\!)(\rho,\gamma,\gamma') \cdot T_{[\![s]\!]}(\rho,\gamma) + (\!|s|\!)(\rho,\gamma,\gamma') \cdot T_{[\![t]\!]}(\rho,\gamma)$$

$$(\!|\textbf{ifge0 } t \textbf{ then } s \textbf{ else } r|\!)(\rho,\gamma,\gamma') = \begin{cases} (\!|s|\!)(\rho,\gamma,\gamma') & \text{if } [\![t]\!](\cdot,\gamma) \geq 0 \\ (\!|r|\!)(\rho,\gamma,\gamma') & \text{otherwise} \end{cases}$$

$$(\!|\textbf{ifge0 } \lfloor\Psi\rfloor \textbf{ then } s \textbf{ else } r|\!)(\rho,\gamma,\gamma') = f(\rho,\gamma,\gamma') + \delta(\pi_0\Psi(\rho,\gamma)) \cdot (D_\gamma\pi_0\Psi)(\rho,\gamma,\gamma') \cdot T_{[\![s]\!]-[\![r]\!]}(\rho,\gamma)$$
$$\text{where } f(\rho,\gamma,\gamma') = \begin{cases} (\!|s|\!)(\rho,\gamma,\gamma') & \text{if } \pi_0\Psi(\rho,\gamma) \geq 0 \\ (\!|r|\!)(\rho,\gamma,\gamma') & \text{otherwise} \end{cases}$$

$$(\!|\textbf{let } z = t \textbf{ in } s|\!)(\rho,\gamma,\gamma') = (\!|s|\!)(\rho, \gamma[z \mapsto [\![t]\!](\cdot,\gamma)], \gamma'[z' \mapsto (\!|t|\!)(\cdot,\gamma,\gamma')])$$

Fig. 8. The denotational semantics for derivatives of terms $(\!|t|\!)(\rho,\gamma,\gamma')$.

Conditionals. A conditional **ifge0** t **then** s **else** r may not be differentiable at the boundary of the condition $[\![t]\!](\rho,\gamma) = 0$, so we only guarantee differentiability almost everywhere.

Diffeomorphic Conditionals. The derivative of a diffeomorphic conditional **ifge0** $\lfloor\Psi\rfloor$ **then** s **else** r is expressed in terms of the Dirac delta distribution $\delta(\hat{\Psi}(\vec{x},\vec{z}))$, which intuitively is zero everywhere except along $\hat{\Psi}(\vec{x},\vec{z}) = 0$, where it approaches infinity. We can only formally define $\delta(\hat{\Psi}(\vec{x},\vec{z}))$ because $\vec{x} \mapsto \hat{\Psi}(\vec{x},\vec{z})$ is a diffeomorphism for every $\vec{z} \in \mathbb{R}^m$ (see Equation 6).

To see that the derivative rule is correct, we first encode the denotation in terms of the Heaviside step function $H(x) := [x \geq 0]$:

$$[\![\textbf{ifge0 } \lfloor\Psi\rfloor \textbf{ then } s \textbf{ else } r]\!](\rho,\gamma) = \begin{cases} [\![s]\!](\rho,\gamma) & \text{if } \pi_0\Psi(\rho,\gamma) \geq 0 \\ [\![r]\!](\rho,\gamma) & \text{otherwise} \end{cases}$$
$$= H(\pi_0\Psi(\rho,\gamma)) \cdot [\![s]\!](\rho,\gamma) + (1 - H(\pi_0\Psi(\rho,\gamma))) \cdot [\![r]\!](\rho,\gamma).$$

Recall from Section 3 that the Dirac delta satisfies: $\delta(x) = D_x H(x)$. By the product and chain rules, we have that the derivative of the first summand is:

$$(\partial_\gamma T_{H_\Psi \cdot [\![s]\!]})(\rho,\gamma,\gamma') = \delta(\pi_0\Psi(\rho,\gamma)) \cdot (D_\gamma\pi_0\Psi)(\rho,\gamma,\gamma') \cdot T_{[\![s]\!]}(\rho,\gamma) + T_{H_\Psi}(\rho,\gamma) \cdot (\!|s|\!)(\rho,\gamma,\gamma'),$$

where $H_\Psi(\rho,\gamma) = H(\pi_0\Psi(\rho,\gamma))$ and the derivative of the second summand is:

$$(\partial_\gamma T_{(1-H_\Psi)\cdot[\![r]\!]})(\rho,\gamma,\gamma') = -\delta(\pi_0\Psi(\rho,\gamma)) \cdot (D_\gamma\pi_0\Psi)(\rho,\gamma,\gamma') \cdot T_{[\![r]\!]}(\rho,\gamma) + T_{1-H_\Psi}(\rho,\gamma) \cdot (\!|r|\!)(\rho,\gamma,\gamma').$$

Adding the two together and combining terms we have that:

$$(\!|\textbf{ifge0 } \lfloor\Psi\rfloor \textbf{ then } s \textbf{ else } r|\!) = \delta(\pi_0\Psi(\rho,\gamma)) \cdot (D_\gamma\pi_0\Psi)(\rho,\gamma,\gamma') \cdot (T_{[\![s]\!](\rho,\gamma)} - T_{[\![r]\!](\rho,\gamma)})$$
$$+ T_{H_\Psi}(\rho,\gamma) \cdot (\!|s|\!)(\rho,\gamma,\gamma') + T_{1-H_\Psi}(\rho,\gamma) \cdot (\!|r|\!)(\rho,\gamma,\gamma').$$

We can convert the sum of Heavisides back to a piecewise function and a difference of regular distributions into a regular distribution of the difference, producing the desired result.

Let Bindings. Let expressions follow the chain rule. In particular, the derivative of a let expression **let** $z = t$ **in** s is the derivative denotation of s evaluated at the infinitesimal perturbation defined by the derivative denotation of t.

7.3 Results

We now prove that the denotational semantics is sound: for every term, the derivative denotation of a term equals the derivative of the denotational term. To do so, we extend the theory.

Definition 7.1. A parametric distribution $u_\theta \in \mathbb{R}^m \to \mathcal{D}'(\mathbb{R}^n)$ is a C^k *simple distribution* if it has a density $g_\theta(\vec{x})$, where $\vec{x} \in \mathbb{R}^n$, that is C^k simply decomposable for all $\theta \in \mathbb{R}^m$. △

Proposition 7.1. *For every well-typed term $\Delta; \Gamma \vdash t$: **real**, if the $[\![t]\!]$ is locally integrable, then it can be lifted to a distribution $T_{[\![t]\!]}$, that is a C^1 simple distribution.* «

Proof. Since t is a locally integrable function, it can be lifted to a distribution $T_{[\![t]\!]}$. By Theorem 6.1, the density $[\![t]\!]$ is simply decomposable, so, $T_{[\![t]\!]}$ is a simple distribution. □

7.3.1 The Transversality Condition. In 1D, the derivative is not defined when there are discontinuities that correspond to the same sets at equality. For instance, the (parametric) distributional derivative of $[x \le \theta][x \le \theta]$ violates the product rule[7] due to a fundamental restriction of distribution theory [Schwartz 1954]. In higher dimensions, degeneracies only occur when the zero sets of $\hat{\Psi}$ are not *transverse.* So, two submanifolds of $\mathbb{R}^n$ are *transverse* when they are not tangent to each other at any point of their intersection. Such a point of tangency creates a degeneracy similar to the 1D case of colocated discontinuities.

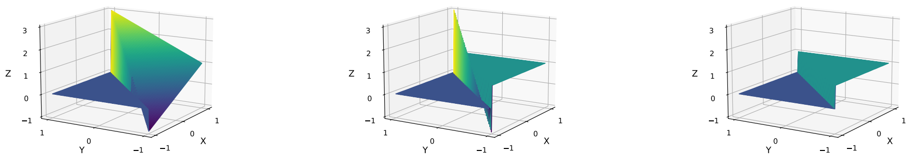
Definition 7.2. A family of submanifolds $\mathcal{M}$ in $\mathbb{R}^n$ are *mutually transverse* when, for any subfamily $\mathcal{M}_I \subseteq \mathcal{M}$ and any point x in the intersection of those submanifolds $\mathcal{M}_I$, the normal spaces of $\mathcal{M}_I$ at x are linearly independent. △

We specialize this definition to $\mathbb{R}^n$, and therefore use normal space rather than the tangent space for simplicity. For example, two planes in $\mathbb{R}^3$ that only intersect along a line are transverse. If they do not intersect at all, then they are (vacuously) transverse. However, when the two planes coincide, i.e. they are the same plane, their intersection is not transverse as the normals for the planes are equivalent. The following examples are not mutually transverse: two osculating circles and three lines in $\mathbb{R}^2$ that intersect at the same point.

The diffeomorphic conditional **ifge0** $\lfloor \Psi \rfloor$ **then** s **else** r has a condition represented by the diffeomorphism $\vec{x} \mapsto \hat{\Psi}(\vec{x}, \vec{z})$. The points along the zero-level sets of the diffeomorphisms are exactly those values for which $\pi_0 \hat{\Psi}(\vec{x}, \vec{z}) = 0$. For example, if $\hat{\Psi}(x, a) = a - x$, the zero-level set is $x = a$.

The derivative of a diffeomorphic conditional introduces a Dirac delta located along the zero-level set of the diffeomorphism. Figure 9 presents such an example. The initial function (9a) has a zero-level set of the diffeomorphism that introduces a ridge representing the Dirac delta in the derivative (9b), which is not present when the delta is not included (9c). For a product of diffeomorphic conditionals, if the corresponding zero-level sets are not transverse, then the denotation of the conditionals is degenerate. For example, if $[x \ge a][x \ge a]$ are equivalent $[x \ge a]$ (which is the case for functions) then the derivative double-counts the jump $2[x \le a]\delta(a - x)$ versus $\delta(a - x)$.

7.3.2 Justification for the Transversality Condition. Terms can often be written in a way that satisfies the transversality condition. For example, in math, some rewrites are: replacing $[x \ge a]^2$ with $[x \ge a]$ or replacing $\sin([x \ge a])$ with $[x \ge a] \sin(1)$ prior to differentiation. Practical cases that cannot be written this way typically pose problems beyond differentiability, such as numerical instability. For instance, the construction of a quadrilateral from two triangles violates the transversality condition because the two triangles have a coincident edge. Operations on the quadrilateral, such as rotation and translation, lead to numerical errors because edges that are supposed to be coincident are instead distinct.



(a) Initial (b) Langur derivative (c) Standard AD derivative

Fig. 9. (a) depicts the integrand $[y < x + \theta](1 + \theta + x + y)$, where $\theta = 0$. (b) shows its derivative $[y < x] + \delta(x - y)(1 + x + y)$ at $\theta = 0$, where the ridge along $y = x$ represents the Dirac delta that we integrate over. (c) is the derivative computed with Standard AD $[y < x]$, which is missing the Dirac delta.

Such numerical instabilities in geometry cause problems such as *light leaks*, where light from behind a mesh shines through an object due to a crack and *missed collisions*, where an object fails to collide with another object and passes through instead [Woop et al. 2013].

So the violation of transversality poses practical challenges beyond that of differentiability. As a result, graphics engines generally encode geometry as a partition of space (each edge is only represented once). Hence, although practical computations can be represented as programs that violate transversality, it is often desirable to implement the computation in an alternative way that restores transversality [Dalstein et al. 2014].

If the transversality condition fails, the program may not have a well-defined distributional derivative (e.g., **ifge0** $\lfloor \Psi \rfloor$ **then** (**ifge0** $\lfloor \Psi \rfloor$ **then** 1 **else** 0) **else** 0). In this case, the interpreter will return results that we do not assign a meaning to. Similarly, Pytorch [Paszke et al. 2019] and Tensorflow [Abadi et al. 2015] produce results at discontinuous points, although the derivative is ill-defined at these points.

7.3.3 Terms as Parametric Distributions. With the formalisms of parametric distributions and the transversality condition in place, we present two key results: (1) the derivative of the denotation of a term is the derivative denotation of the integrand (soundness) and (2) the derivative of a term is a parametric distribution that is the sum of a C^0 simply decomposable function and Dirac deltas multiplied by C^0 simply decomposable functions in Figure 10.

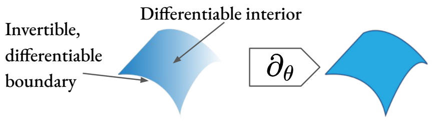


Fig. 10. A well-typed term denotes a C^0 *simply decomposable function* (left). Its derivative D_θ (right) is a sum of a *simple distribution* (constant interior color) and Dirac deltas multiplied by C^0 simply decomposable functions (the boundary lines).

Proposition 7.2. *For every $(\gamma, \gamma') \in (\!|\Gamma|\!)$, if $\Delta; \Gamma \vdash t$: **real** then $(\!|t|\!)(\rho, \gamma, \gamma') \in \mathcal{D}'(\mathbb{R}^n)$ whenever the zero level-set of all diffeomorphisms in the derivative are mutually transverse.* «

Proof. The interesting case is products. We have that $(\!|t \cdot s|\!)(\rho, \gamma, \gamma') = (\!|t|\!)(\rho, \gamma, \gamma')T_{[\![s]\!]} + (\!|s|\!)(\rho, \gamma, \gamma')T_{[\![t]\!]}$ is a distribution. Here we use the transversality side condition (Theorem B.2). □

Definition 7.3. Let $f, g : \mathbb{R}^n \to \mathbb{R}$ be given. We say that $f(\vec{x}) = g(\vec{x})$ *for almost every* $\vec{x}$ if $\{x \in \mathbb{R}^n \mid f(\vec{x}) \neq g(\vec{x})\}$ has Lebesgue measure zero. △

Lemma 7.1 (Derivative correctness for terms). *Let $(\gamma, \gamma') \in (\!|\Gamma|\!)$ be given. Assume that the zero level-set of all diffeomorphisms in the derivative are mutually transverse. For every $\Delta; \Gamma \vdash t$: **real**, the derivative denotation and derivative commute:*

$$(\!|t|\!)(\rho, \gamma, \gamma') = (\partial_\gamma T_{[\![t]\!]})(\rho, \gamma, \gamma')$$

for almost every γ. «

$$\frac{x_1 : \mathbf{real}, \ldots, x_n : \mathbf{real}; \Gamma \vdash t : \mathbf{real}}{\cdot; \Gamma \vdash \mathbf{int}\ t\ \mathbf{d}(x_1, \ldots, x_n) : \mathbf{real}}$$

Fig. 11. The type system specifying well-typed programs.

$$[\![\mathbf{int}\ t\ \mathbf{d}(x_1, \ldots, x_n)]\!] : [\![\Delta]\!] \times [\![\Gamma]\!] \to \mathbb{R} \qquad [\![\mathbf{int}\ t\ \mathbf{d}(x_1, \ldots, x_n)]\!](\cdot, \gamma) = \int_{\mathbb{R}^n} [\![t]\!](\rho, \gamma)\, \mathrm{d}\rho$$

$$(\!|\mathbf{int}\ t\ \mathbf{d}(x_1, \ldots, x_n)|\!) : (\!|\Delta|\!) \times (\!|\Gamma|\!) \to \mathbb{R} \qquad (\!|\mathbf{int}\ t\ \mathbf{d}(x_1, \ldots, x_n)|\!)(\cdot, \gamma, \gamma') = \int_{\mathbb{R}^n} (\!|t|\!)(\rho, \gamma, \gamma')\, \mathrm{d}\rho$$

Fig. 12. The denotational semantics of programs.

Proof. We provide the full proof in Appendix I. □

We now characterize the semantic domain of the derivatives of well-typed terms.

Theorem 7.1. *For every well-typed term $\Delta; \Gamma \vdash t : \mathbf{real}$, if the $[\![t]\!]$ is locally integrable and the zero level-set of all diffeomorphisms in the derivative are mutually transverse then $(\!|t|\!)(\rho, \gamma, \gamma')$ is a sum of a C^0 simple distribution and a finite sum of delta distributions multiplied by C^0 simply decomposable functions. Concretely, using the isomorphism between variable names and ordered variables, $(\!|t|\!)(\rho, \gamma, \gamma')$ can be written as the distribution:*

$$u(\vec{x}, \vec{z}, \vec{z}') = f(\vec{x}, \vec{z}, \vec{z}') \cdot T_1(\vec{x}) + \sum_i \sum_k h_{ik}(\vec{x}, \vec{z}, \vec{z}') \cdot \delta(\Phi_{ik}(\vec{x}, \vec{z})), \tag{9}$$

where f and h_{ik} for all i and k are C^0 simply decomposable and T_1 is the lifting of the one function $1(\vec{x})$ into a distribution (Definition 3.7). «

Proof. It is sufficient to prove the theorem for $\partial_\gamma T_{[\![t]\!]}$ (Lemma 7.1 and Lemma 7.1 cover the other cases). Theorem 6.1 gives us that $[\![t]\!]$ is C^1 simply decomposable. We can then take the derivative $D_{\vec{z}}$ of each of the summands $g_i(\vec{x}, \vec{z}) \prod_j \left[\Phi_{ij}(\vec{x}, \vec{z}) \geq 0\right]$ from the definition of simply decomposable. Using Lemma 7.1 to apply the product rule, we find that the derivative is:

$$u(\vec{x}, \vec{z}, \vec{z}') = \sum_i \Big(f_i(\vec{x}, \vec{z}, \vec{z}') \cdot T_1(\vec{x}) + \sum_k h_{ik}(\vec{x}, \vec{z}, \vec{z}') \cdot \delta(\Phi_{ik}(\vec{x}, \vec{z})) \Big),$$

where i and k are in finite sets and the functions $f_i(\vec{x}, \vec{z}, \vec{z}') = (D_{\vec{z}} g_i)(\vec{x}, \vec{z}, \vec{z}') \prod_j \left[\Phi_{ij}(\vec{x}, \vec{z}) \geq 0\right]$ and $h_{ik}(\vec{x}, \vec{z}, \vec{z}') = (D_{\vec{z}} \Phi_{ik})(\vec{x}, \vec{z}, \vec{z}') g_i(\vec{x}, \vec{z}) \prod_{j \neq k} \left[\Phi_{ij}(\vec{x}, \vec{z}) \geq 0\right]$ are C^0 simply decomposable. We can use linearity and set $f(\vec{x}, \vec{z}, \vec{z}') = \sum_i f_i(\vec{x}, \vec{z}, \vec{z}')$, which is simply decomposable. □

8 DENOTATIONAL SEMANTICS OF PROGRAMS AND THEIR DERIVATIVES

Langur programs are the integrals and the derivative of integrals of Langur terms. We discuss integrability and prove a soundness theorem that shows that the derivative denotation of a program equals the derivative of the denotation of that program.

8.1 Type System

Figure 11 depicts the typing rules for Langur programs. The syntax $\mathbf{int}\ t\ \mathbf{d}(x_1, \ldots, x_n)$ introduces variables $x_1, \ldots, x_n$ into the typing context for t. A well-typed program integrates over a well-typed term with no free variables, preventing nested integration.

8.2 Denotational Semantics

Figure 12 depicts the denotational semantics for Langur programs and their derivatives. The integral primitive denotes n-dimensional Lebesgue integration. We use the notation $\mathrm{d}\rho$ to introduce variables $x_1, \ldots, x_n$ in the integrand. The denotation of the derivative of an integral is the integral of the distributional derivative of the integrand.

8.3 Results

We now briefly discuss integrability and the correctness of the derivative.

Recall from Section 2 that a program in the surface language has a compact domain of integration specified by the constant bounds of integration. Its encoding in the core language results in an integrand with compact support.

For example, a program in the surface language `integral ([-10, 10]) x dx` has a compact domain of integration $[-10, 10]$. In the core language, this program can be encoded as:

$$\textbf{int}\ (\textbf{ifge0}\ x + 10\ \textbf{then}\ 1\ \textbf{else}\ 0) \cdot (\textbf{ifge0}\ 10 - x\ \textbf{then}\ 1\ \textbf{else}\ 0) \cdot x\ \mathbf{d}x,$$

where the integrand has compact support $[-10, 10]$.

As a result of this property, we can show that programs implemented in the surface language have a well-defined denotation in the core language.

Proposition 8.1. *A well-typed program $\Delta; \Gamma \vdash \textbf{\textit{int}}\ t\ \boldsymbol{d}(x_1, \ldots, x_n) : \textbf{\textit{real}}$ has a well-defined denotation $[\![\textbf{\textit{int}}\ t\ \boldsymbol{d}(x_1, \ldots, x_n)]\!](\cdot, \gamma)$ if the integrand $[\![t]\!](\rho, \gamma)$ has compact support.* «

Proof. By Theorem 6.1, the integrand denotes a simply decomposable function. Splitting the domain of integration into these finitely many pieces, we see that each of the integrals is bounded because it is the integral of a continuous function over compact support. □

Proposition 8.2. *The derivative denotation $(\!|\textbf{\textit{int}}\ t\ \boldsymbol{d}(x_1, \ldots, x_n)|\!)(\cdot, \gamma, \gamma')$ of a well-typed program $\Delta; \Gamma \vdash \textbf{\textit{int}}\ t\ \boldsymbol{d}(x_1, \ldots, x_n) : \textbf{\textit{real}}$ is well-defined if the integrand $[\![t]\!](\rho, \gamma)$ has compact support.* «

Proof. Theorem 7.1 shows that the integrand is a C^0 simple distribution plus a finite sum of Dirac deltas and scaled by C^0 simply decomposable functions each of which has compact support. The sum of Dirac deltas is bounded because the integrand is bounded. Since the integral of a continuous function with compact support is bounded, and there are finitely many continuous pieces, the result is bounded. □

Derivative Correctness. The derivative of the denotation equals the derivative denotation:

Theorem 8.1 (Derivative correctness). *Let $(\gamma, \gamma') \in (\!|\Gamma|\!)$ be given. For every well-typed program $\Delta; \Gamma \vdash \textbf{\textit{int}}\ t\ \boldsymbol{d}(x_1, \ldots, x_n) : \textbf{\textit{real}}$, the denotation and derivative commute:*

$$(\!|\textbf{\textit{int}}\ t\ \boldsymbol{d}(x_1, \ldots, x_n)|\!)(\cdot, \gamma, \gamma') = (D_\gamma [\![\textbf{\textit{int}}\ t\ \boldsymbol{d}(x_1, \ldots, x_n)]\!])(\gamma, \gamma')$$

where the equivalence above is almost everywhere equivalence, and we assume that the program $[\![p]\!]$ and its derivative $(\!|\textbf{\textit{int}}\ t\ \boldsymbol{d}(x_1, \ldots, x_n)|\!)(\cdot, \gamma, \gamma')$ are well-defined and that the zero level-set of all diffeomorphisms in the derivative are mutually transverse. «

Proof. We provide the full proof in Appendix J. □

9 SEPARATE COMPILATION

In this section, we discuss an interpreter for the core language Langur, formalize separate compilation, and relate the surface language Potto to Langur.

The operational semantics for programs follows from the denotational semantics (Appendix D). We now formalize the relationship between the two.

Operationally, we can commute the derivative and the integral (Figure 12):

$$(\!|\textbf{int}\ t\ \mathbf{d}(x_1, \ldots, x_n)|\!)(\cdot, \gamma, \gamma') = \int_{\mathbb{R}^n} (\!|t|\!)(\rho, \gamma, \gamma')\, \mathrm{d}\rho = \int_{\mathbb{R}^n} u(\vec{x}, \vec{z}, \vec{z}')\, \mathrm{d}\vec{x},$$

where u is a parametric distribution and the last equality applies the isomorphism between mappings from variable or parameter names to ordered real numbers.

Then if $(\!|t|\!)$ satisfies the conditions of the Theorem 7.1:

$$\begin{aligned}\int_{\mathbb{R}^n} u(\vec{x},\vec{z},\vec{z}')\,\mathrm{d}\vec{x} &= \int_{\mathbb{R}^n} f(\vec{x},\vec{z},\vec{z}')\cdot T_1(\vec{x}) + \sum_i\sum_k h_{ik}(\vec{x},\vec{z},\vec{z}')\cdot\delta(\Phi_{ik}(\vec{x},\vec{z}))\,\mathrm{d}\vec{x} && \text{by Thm 7.1}\\ &= \int_{\mathbb{R}^n} f(\vec{x},\vec{z},\vec{z}')\,\mathrm{d}\vec{x} + \sum_i\sum_k\int_{\mathbb{R}^n} h_{ik}(\vec{x},\vec{z},\vec{z}')\,\mathrm{d}\delta(\Phi_{ik}(\vec{x},\vec{z}))(\vec{x}) && \text{by linearity}\end{aligned}$$

where f and h_{ik} for all i and k are C^0 simply decomposable functions, and T_1 is the one function lifted to a distribution, and $\delta(\Phi_{ik}(\vec{x},\vec{z}))$ is the Dirac delta distribution on the first line and the Dirac measure on the second line (it is one if $\{\vec{x}\in\mathbb{R}^n \mid \Phi_{ik}(\vec{x},\vec{z})=0\}$ and zero otherwise).

We can then implement the operational semantics for derivatives of Langur programs with Monte Carlo integration over each of the measures as is standard in probabilistic programming languages. The operational semantics for derivatives provides an implementation for sampling and point evaluation at a sample (Appendix E). This is analagous to `sample` and `score` in probabilistic programming [Staton 2017]. We summarize some interesting cases.

Diffeomorphic Conditionals. The derivative of a diffeomorphic conditional introduces a Dirac delta distribution. To implement `sample`, Langur must draw unbiased samples satisfying the equality $\pi_0\Psi(x_1,...,x_n) = 0$. For example, if **ifge0** $\lfloor\Psi\rfloor$ **then** 1 **else** 0 has a diffeomorphism $\hat{\Psi}(x) = a - x$ with inverse $\hat{\Psi}^{-1}(y) = a - y$ in the condition. The distribution derivative of **ifge0** $\lfloor\Psi\rfloor$ **then** 1 **else** 0 is $\delta(a-x)$. In math, we would calculate the derivative by applying a change of variables $u = a - x$ (Definition 6) and applying the sifting property $\int(a-u)\delta(u)du = a$ (Equation 5). This is equivalent to sampling the point $x = a$ and evaluating the rest of the integrand at $x = a$. Langur implements `sample` by returning $x \mapsto a$ and implements `score`, by evaluating the rest of the integrand at $x = a$.

Let Bindings. For let bindings **let** $z = t$ **in** s, by the type system for diffeomorphisms, the parameter z cannot be an input to a diffeomorphism and as a result, samples generated from the derivative of s do not depend on t. As such, Langur can separately sample from the measures denoted by t and s and combine the results. This is key to enabling separate compilation. To evaluate **let** $z = t$ **in** s, we need only evaluate t at given sample points and then bind each value to z in s and evaluate s.

Compositional Evaluation and Separate Compilation. We consider relations taking in an environment E and a term of a grammar G and returning either nothing (it is partial) or a value. If it returns a value, we write it as $(E,t)Rv$. We consider values that are either (real) numbers or collections of numbers (such as disjoint unions of pairs of numbers). This is critical because passing an expression to a function violates compositional evaluation

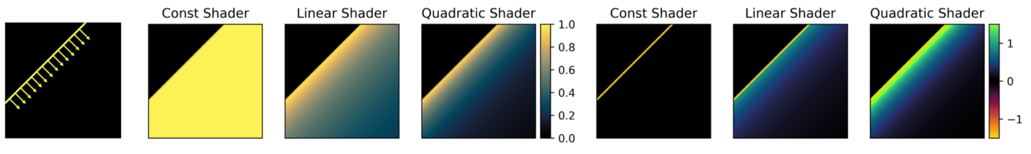
A hypothesis H is a predicate defined by a function $h(v_1,\ldots,v_l)$. A function such as $h(E,v_1,\ldots,v_l)$ depends only on $v_1,\ldots,v_l$, E, and mathematical functions (e.g., f,Ψ, $\leq$, $=$).

We consider an operational semantics as a set of relations $\mathcal{R}$ over a grammar G. Let R be a metavariable representing one of a fixed set of relations.

Definition 9.1. An operational semantics supports *compositional evaluation* if, for every inference rule, the conclusion depends only on the values generated by the hypotheses. Formally, each inference rule can be written in the following form: $\frac{(E,t_1)Rv_1 \quad \ldots \quad (E,t_l)Rv_l \quad H_1 \quad \ldots \quad H_q}{(E,t)Rg(E,v_1,\ldots,v_l)}$. △

Theorem 9.1. *The operational semantics supports compositional evaluation.* «

The proof follows by inspection of the operational semantics ($\mathcal{R} = \{\Downarrow_N, \Downarrow_N^d, \Uparrow_N^d\}$), where the case of let binding is the key case. Langur achieves compositional evaluation by evaluating subterms before combining results in every rule.



(a) Scene (b) Shaders change how light attenuates (c) The derivative w.r.t. light position

Fig. 13. We use a differentiable renderer implemented in Potto turn scenes containing a single line of light pointing diagonally downward with different light attenuations into images. Swapping between different shaders is a critical part of the design process. In Potto, programs and their derivatives can be separately compiled and composed to efficiently swap among shaders.

The mechanism of building separate compilation from compositional operational semantics is well studied [Cardelli 1997]. Separate compilation is important in programming and specifically in computer graphics where many small changes are made to program parts in the design process. Langur separately compiles program parts and simply evaluates them at different points to produce results rather than requiring global program manipulations as in Teg [Bangaru et al. 2021].

Example. We now discuss the Potto program from Section 2 (code blocks 1, 2, 3). We briefly discuss this conversion from the Potto to the Langur. Recall from the type system that variables and parameters are passed in the environment.

The `risk(distro, a, b, mu)` becomes the integral on the last line because we have applied the function trunc_normal. We take the liberty of using the names normal, μ, and trunc_normal for parameters rather than z_1, z_2, and z_3 for clarity.

We need a little more machinery to introduce `risk` and therefore gain the benefits of separate compilation. We introduce a finite looping construct that is syntactic sugar and is unrolled. Using the same argument as for let binding, we can separately compile the body of the loop and iteration.

10 EVALUATION

In this section, we evaluate the prototype programming language, Potto, and implement a differentiable renderer with *swappable shaders*, meaning that each shader can be separately compiled and therefore efficiently interchanged. We describe implementation details and optimizations on top of those presented in the operational semantics in Appendix G. We compare Potto and Teg on a renderer from Teg and two synthetic examples as well [Bangaru et al. 2020].

10.1 A Differentiable Renderer with Swappable Shaders

We implement a *differentiable renderer* [de La Gorce et al. 2011; Li et al. 2018, 2020; Loper and Black 2014; Zhao et al. 2020]. Domains ranging from autonomous driving to robotics to computer-generated imagery, use differentiable renderers to recognize the 3D shapes of cars, signs, and pedestrians, to reconstruct a 3D scene for a robot, and to capture the 3D properties of actors' faces.

A *renderer* is a program that takes in the geometry and color of each object in the scene and outputs an image. Renderers are built out of programs called *shaders*. A differentiable renderer computes the change in the color of a pixel with respect to changes in parameters, such as the location or color of an object. These derivatives are useful for optimization.

Figure 13a depicts the scene: a line of light shining diagonally downward. The color of a single pixel is the average of light within the pixel area:

$$f(c,s) = \int_0^1 \int_0^1 s(x,y,c)\,[x+y+c \geq 0]\ \mathrm{d}x\mathrm{d}y. \tag{10}$$

We use the convention that the origin is in the top left corner and the y-axis points down. The half-plane $[x + y + c \geq 0]$ is the shader determining whether a point is illuminated. The function $s : \mathbb{R}^3 \rightarrow \mathbb{R}$ is the shader that determines the brightness as a function of x, y, and c.

The goal is to optimize the parameter c so that the renderer generates a pixel with color a. We use gradient descent to minimize the loss function $L(c, s) = (f(c, s) - a)^2$ with derivative $D_cL(c, s) = 2(f(c, s) - a)D_cf(c, s)$. A differentiable renderer computes the derivative $D_cf(c, s)$.

Differentiating Parametric Discontinuities. The leftmost image in Figure 13b depicts the constant shader $s(x, y, c) = 1$, which a half-plane. The leftmost image in Figure 13c depicts its derivative, which is a line along the boundary of the half-plane because it is the change resulting from perturbing c in $f(c, s)$. The half-plane shifts diagonally downward, making the boundary the only region with nonzero derivative. The linear shader $s(x, y, c) = (\sqrt{2}(x + y + c) + 2)^{-1}$ and quadratic shader $s(x, y, c) = (\sqrt{2}(x + y + c) + 2)^{-2}$ have the same boundary contribution to the derivative, but also have a nonzero interior derivative.

The derivative of the renderer decomposes into the interior and boundary contributions:

$$D_cf(c, s) = \int_S \underbrace{\partial_c s(x, y, c)[x + y + c \geq 0]}_{\text{interior}} + \underbrace{s(x, y, c)\delta(x + y + c)}_{\text{boundary}} \, \mathrm{d}(x, y). \tag{11}$$

The contribution of the Dirac delta distribution δ shows up as the diagonal yellow line in the derivative of all three shaders. We provide a derivation in Appendix A.

Automatic Differentiation of Parametric Discontinuities. A naïve implementation would discretize the integral to a sum and use automatic differentiation to compute the derivative of the discretization. The resulting program would approximate the interior term in Equation 11, but ignore the boundary term, producing an incorrect result. We implement the renderer specified in Equation 10:

```
1  # renderer.po
2  from half_plane import cond
3  def renderer(c: Param, s):
4     return integral ([0,1],[0,1]) s(x,y,c)*(1 if cond(x,y,c) else 0) d(x,y)
```

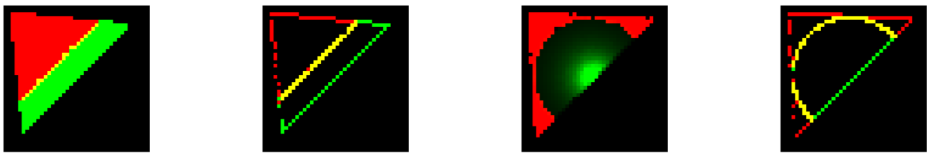
Line 2 imports the diffeomorphism from the `half_plane.diffeo` file. It takes in variables x, y and parameter c, and returns an affine combination of the three, representing a 2D rotation parameterized by c. This file can be implemented as a collection of Potto programs that specify the forward and inverse transformations.

```
# color_shaders.po
def const_shader(x: Var,y: Param,c: Param): return 1
def lin_shader(x: Var,y: Param,c: Param): return 1/(sqrt(2)*(x+y+c)+2)
def quad_shader(x: Var,y: Param,c: Param): return 1/(sqrt(2)*(x+y+c)+2)^2
```

The three shaders are first-order functions that model how quickly light attenuates. The `const_shader` corresponds to no attenuation—intensity is invariant to the distance from the light. While in `lin_shader` and `quad_shader`, the attenuation is linear and quadratic, respectively.

```
# main.po
from color_shaders import_deriv const_shader, lin_shader, quad_shader
     as dconst_shader, dlin_shader, dquad_shader
from renderer import_deriv renderer as drenderer
for dshader in [dconst_shader, dlin_shader, dquad_shader]:
  print((drenderer (-2, 1) dshader)[1])
```

In `main.po`, we compose the derivative of the renderer with the derivative of each of the shaders to compute the pixel color. We evaluate the resulting expression at a base point `c=-2` with infinitesimal



(a) Depth shader (b) Depth shader deriv. (c) Thresh. Lambert (d) Thresh. Lam. deriv.

Fig. 14. Discontinuous shaders at a 40×40 resolution. We display the sparsity pattern of the derivative shaders. For example, the derivative shader is red, if the derivative is nonzero in the red channel.

`dc=1`, producing a number representing the derivative. The program then does a step of gradient descent to find the value of c resulting in a pixel most similar to the pixel in a given image.

Separate Compilation. Potto separately compiles the derivative of the renderer and shaders, making it possible to compile different program parts to different hardware (e.g., the CPU and GPU). As a result, Potto compiles the derivative of the renderer and each shader once rather than needing to compile the derivative of the renderer and shaders three times. Since calculating the derivative of the renderer is so time-consuming, it results in a 2.78x speedup.

10.2 A Ray tracing Differentiable Renderer with Swappable Shaders

A *ray tracer* is a renderer that computes the color of each pixel in the image by shooting rays from a camera to identify which objects are visible and what color they have. The ray tracer shoots rays into the scene and computes the color at the intersection of the ray with the geometry [Shirley 2020]. The subset of parameters that control the color, reflectiveness, and other properties of surfaces reside in the shader programs. Changing shader parameters changes the stylization of the scene.

Differentiable rendering [Li et al. 2018; Loper and Black 2014] is useful for solving inverse problems. For example, an artist might draw a 2D image and want a 3D scene that, when viewed from a particular angle, looks as similar to their drawing as possible. Gradient-based optimization can automatically set these shader parameters.

A production renderer has many shaders that artists can select to achieve a desired effect [Perlin 1985]. Rendering often involves large geometries within the scene, making it important to be able to quickly swap out shaders without recompiling the whole rendering engine.

As a result, engines typically allow for separate compilation of the renderer from the component shaders. However, boundary sampling techniques are either limited to affine discontinuities [Li et al. 2018] or do not support separate compilation [Bangaru et al. 2021]. To solve inverse problems with multiple shaders without incurring this overhead, it is imperative to be able to swap out shaders and their derivatives in the derivative of a renderer.

Each pixel in the rendered image is a rectangular cell in an imaginary screen in front of the camera. Its color is the average (integral) of the light rays that pass through the cell.

Swappable Shaders. We would like to swap between different discontinuous shaders, often called toon shaders based on their cartoon stylized look. Often these come from standard physical shaders that are thresholded. Our example shaders are 1) a *z-depth shader*, which gives different colors to objects based on how far away from the camera they are, and 2) a *thresholded Lambert shader*, which models the reflectance of a matte surface under a point light, which sits in front of the triangle and has linear intensity falloff [Lake et al. 2000].

Figure 14 depicts images produced by a Potto implementation of a differentiable ray tracer. The derivative of Figure 14a is Figure 14b and the derivative of Figure 14c is Figure 14d.

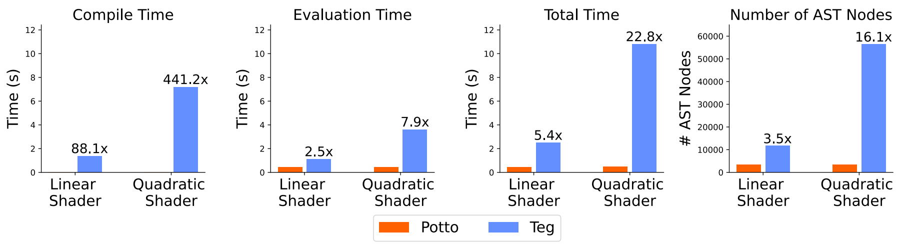


Fig. 15. A bar chart, where smaller is better, comparing Potto to Teg on an image stylization. Potto is so much faster in compile time that the bars on not visible. Compile time was the bottleneck in Teg.

The scene consists of a single triangle, where the upper left corner is closer to the camera than the others. We differentiate with respect to the vertex positions of the triangle as it expands in space, as well as a threshold parameter in the shaders. The upper left corner comes towards the camera, while the other corners spread outward.

The z-depth shader has a discontinuity at a fixed depth in space. The triangle is colored red if it is close enough to the camera, and green otherwise. As the triangle expands, more black pixels along its border become colored and more of the triangle becomes red.

The thresholded Lambert shader has a discontinuity on the triangle surface based on the amount of light from the point light source that is received. When the light value is less than a certain threshold, the triangle is colored red. As the triangle expands, the transition curve moves around on the triangle.

10.3 Comparison to Teg

We compare to Teg [Bangaru et al. 2021] on an image stylization task and two microbenchmarks.

Methodology. Teg provides three backends: a vectorized Python implementation using NumPy, a C backend, and a CUDA backend. We use the vectorized NumPy backend because it most closely matches our implementation. We also observed overhead from starting the C compiler that made evaluation in the generated C code slower than in NumPy on these benchmarks. We run all evaluations using a desktop with an Intel i9-10900k 10-core CPU and 64GB of memory.

10.3.1 Renderer for Image stylization. Bangaru et al. [2021] presents a renderer for image stylization in Figures 5 and 6. Figure 15 shows the (derivative) compilation time, evaluation time, total time, and AST size for linear and quadratic shader stylization.

Teg creates duplicate expressions during compilation, leading to increased compile time, evaluation time, total time, and code size. Teg does not perform common sub-expression elimination (same for Potto), so the evaluation time increases. Separate compilation in Potto results in smaller ASTs and faster compilation and evaluation.

10.3.2 Microbenchmarks. We design a set of small benchmarks to compare Potto and Teg in terms of compile time (to calculate the derivative), evaluation time, total time (both compilation and evaluation), and code size. We report the geometric mean across n runs and set $n = 10$ for the first benchmark and $n = 3$ for the second due to time constraints.

Increasing the Number of Parametric Discontinuities. We study how the performance of Potto and Teg scale as we increase the number of parametric discontinuities to be differentiated. The program takes a function `f` and a number `n` representing the number of Heavisides and produces an expression evaluating the sum of Heavisides multiplied by the function. We bind `f` to programs first-order functions `a+x` and `a-x`. The number `n` controls the number of Heavisides and therefore the number of Dirac deltas in the derivative.

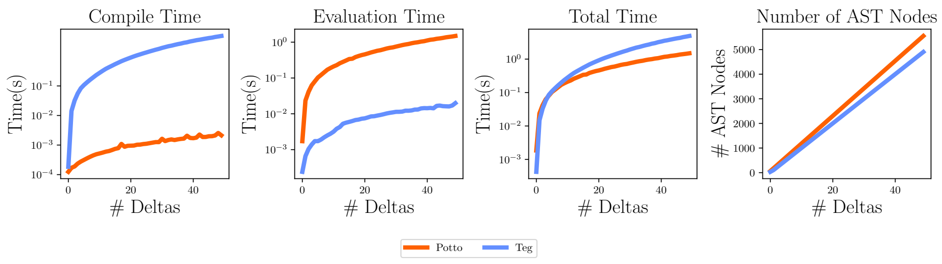


Fig. 16. A comparison of how Potto and Teg scale with the number of discontinuities.

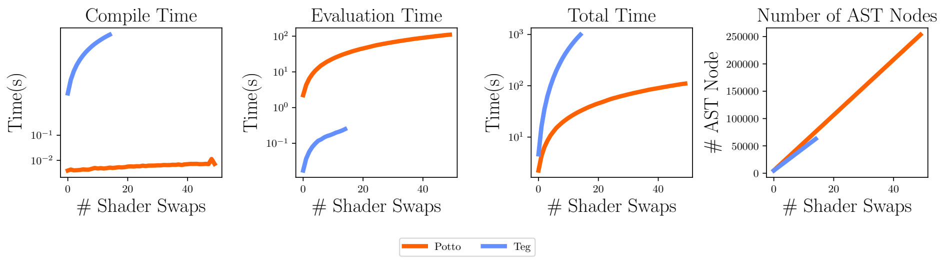


Fig. 17. A comparison of how Potto and Teg scale with the number of swaps between shaders.

Figure 16 depicts the trends as the number of Dirac deltas increases. Potto compiles code faster than Teg, which is a result of the global program transformations needed in Teg. Potto has slower runtime performance than Teg, which is due to Potto doing dynamic tracing and the fact that it is written in pure Python. Potto takes less total time to compute the result. Potto and Teg have similar AST sizes that grow linearly in the number of deltas.

Separate Compilation. We study the impact of separate compilation by building an expression with 90 parametric discontinuities representing the geometry of a scene and scaling the number of times we swap between shaders n. Figure 17 depicts the trends as the number of shader swaps increases. Teg must repeatedly compile the whole expression every time we swap the shader, leading to a slow compile time that reaches our timeout of 20 minutes at 15 swaps. On the other hand, since Potto can separately compile the two modules, the compilation cost increases slowly. Potto needs about a minute at 50 swaps.

11 RELATED WORK

We extend distribution theory to differentiate under integrals. Our implementation draws on concepts from differentiable and probabilistic programming. For applications, we build on a body of research in differentiable rendering and probabilistic inference.

11.1 Distribution Theory

Distribution theory was formalized by Schwartz [1950]. We adapt the theory to address the product of distributions problem [Schwartz 1954]. Recent programming languages work uses distribution theory to (equationally) reason about derivatives of discontinuous programs in a higher-order language [Azevedo de Amorim and Lam 2022]. However, they do not extend directly the theory, which poses a challenge for reasoning about nested conditionals, preventing applications as simple as rendering a triangle. Their lack of diffeomorphic conditional means that the implementation is similarly restricted.

11.2 Sampling Along Discontinuities

Early work hand-coded samplers for discontinuities [Li et al. 2018]. Concurrent work automatically handled affine discontinuities with applications to stochastic variational inference Lee et al. [2018]. Recent work also differentiated parametric discontinuities using an integration primitive [Bangaru et al. 2021]. The paper claims to support higher-order derivatives, but this is not the case (diffeomorphisms are often not preserved under differentiation). We provide a formal foundation that encompasses their work and support additional language features useful in graphics (e.g., first-order functions and separate compilation).

11.3 Smoothing Out Discontinuities

Rather than directly sampling discontinuities, one can smooth them out [Chaudhuri and Solar-Lezama 2010; Laine et al. 2020; Liu et al. 2019]. This typically leads to approximate results and requires additional parameters [Inala et al. 2018; Yang et al. 2022]. We focus on consistent, unbiased estimation of derivatives, but are interested in the practical trade-offs between these approaches. Recent work showed that unbiased estimation is possible for smoothed out discontinuities, but the work is yet to be systematized [Bangaru et al. 2020].

Automatic Differentiation. Automatic differentiation has been a known technique for quite some time [Wengert 1964]. Interest in developing efficient AD systems has grown significantly [Abadi et al. 2015; Bradbury et al. 2018; Paszke et al. 2019; Pearlmutter and Siskind 2008; Yu et al. 2014]. Researchers have also developed theoretical techniques for proving correctness [Abadi and Plotkin 2019; Elliott 2018; Lee et al. 2020; Mazza and Pagani 2021; Sherman et al. 2021]. Sherman et al. [2021] study a language with derivatives and integrals but returns vacuous results when differentiating parametric discontinuities.

Probabilistic Programming. Saheb-Djahromi [1978] and Kozen [1981] performed early work on the semantics of probabilistic programs. Recent work develops efficient probabilistic programming languages [Bingham et al. 2019; Cusumano-Towner et al. 2019; Stan Development Team 2015]. Compositionality is also of interest in these languages. For example, Cusumano-Towner et al. [2019] introduces a generative function interface that is compositional and should similarly support separate compilation. We believe distributional semantics are also relevant to other inference tasks [Gehr et al. 2020; Lew et al. 2019; Shan and Ramsey 2017; Zhou et al. 2019].

Recent work, ADEV provides a framework for reasoning about the composition of gradient estimators and providing proofs of correctness [Lew et al. 2023]. It lacks support for differentiating parametric discontinuities as discussed in their related work section in the subsection “AD of Languages with Integration." Our paper provides such a sound gradient estimator that could potentially compose with other estimators using an appropriate extension of their framework. This would require changing their type system to admit programs with parametric discontinuities beyond discrete distributions that can either be enumerated or estimated via REINFORCE.

12 CONCLUSIONS

Parametric discontinuities arise in applications spanning computer graphics [Li et al. 2018], robotics [Hu et al. 2020], and probabilistic programming [Lee et al. 2018]. We design a theory providing a semantic model for differentiating parametric discontinuities that arise in these applications. Using the insight from our theory, we implement a system that can separately compile programs, allowing us to build a differentiable renderer with swappable shaders. In the future, we hope that differentiable programming languages will support differentiation of parametric discontinuities to better serve application domains.

ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers whose feedback guided us to simplify and clarify the key ideas of the paper. Thank you to our shepherd Michael Arntzenius who gave multiple rounds of extensive and technical feedback as well as helped shape the organization and presentation of the paper. We would also like to thank the PC chairs Alex Potanin and Bor-Yuh Evan Chang for their support and involvement throughout the process.

Thank you to Ellie Cheng, Eric Atkinson, Tian Jin, Logan Weber, Cambridge Yang, Alex Renda, Charles Yuan, Mattieu Huot, Alex Lew, Feras Saad, and Teodoro Collin for their feedback.

This work was supported in-part by the National Science Foundation (CCF-1751011, 2238839), the Sloan Foundation, and SRC JUMP 2.0 (CoCoSys).

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.

Martín Abadi and Gordon D. Plotkin. 2019. A Simple Differentiable Programming Language. Principles of Programming Languages (2019).

Luke Anderson, Tzu-Mao Li, Jaakko Lehtinen, and Frédo Durand. 2017. Aether: An Embedded Domain Specific Sampling Language for Monte Carlo Rendering. (2017).

Pedro H. Azevedo de Amorim and Christopher Lam. 2022. Distribution Theoretic Semantics for Non-Smooth Differentiable Programming. *arXiv e-prints* (2022).

Sai Bangaru, Jesse Michel, Kevin Mu, Gilbert Bernstein, Tzu-Mao Li, and Jonathan Ragan-Kelley. 2021. Systematically Differentiating Parametric Discontinuities. (2021).

Sai Praveen Bangaru, Tzu-Mao Li, and Frédo Durand. 2020. Unbiased warped-area sampling for differentiable rendering. *Special Interest Group on Computer Graphics and Interactive Techniques in Asia* (2020).

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. *Journal of Machine Learning Research* (2019).

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

Luca Cardelli. 1997. Program Fragments, Linking, and Modularization *(Principles of Programming Languages).*

Swarat Chaudhuri and Armando Solar-Lezama. 2010. Smooth Interpretation. In *Programming Language Design and Implementation.*

Brian Conrad. 2006. Math 396. Stokes' theorem with corners. https://math.stanford.edu/~conrad/diffgeomPage/handouts/stokescorners.pdf. Accessed: 2022-11-10.

Marco F. Cusumano-Towner, Feras A. Saad, Alexander K. Lew, and Vikash K. Mansinghka. 2019. Gen: A General-Purpose Probabilistic Programming System with Programmable Inference. In *Programming Language Design and Implementation.*

Boris Dalstein, Rémi Ronfard, and Michiel van de Panne. 2014. Vector Graphics Complexes. *ACM Trans. Graph.* (2014).

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.* (2011).

Erik del Toro Streb and Oleg Alexandrov. 2009. Diffeomorphism of a square — Wikipedia, The Free Encyclopedia. https://commons.wikimedia.org/wiki/File:Diffeomorphism_of_a_square.svg [Online; accessed 3-March-2024].

J.J. Duistermaat and J.A.C. Kolk. 2004. *Multidimensional Real Analysis II.* Cambridge University Press.

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.*

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).

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

Dexter Kozen. 1981. Semantics of probabilistic programs. *J. Comput. System Sci.* (1981).

Samuli Laine, Janne Hellsten, Tero Karras, Yeongho Seol, Jaakko Lehtinen, and Timo Aila. 2020. Modular primitives for high-performance differentiable rendering. *Special Interest Group on Computer Graphics and Interactive Techniques in Asia* (2020).

Adam Lake, Carl Marshall, Mark Harris, and Marc Blackstein. 2000. Stylized Rendering Techniques for Scalable Real-Time 3D Animation. In *International Symposium on Non-Photorealistic Animation and Rendering*. 8 pages.

John M. Lee. 2012. *Introduction to Smooth Manifolds, 2nd Ed.* Springer-Verlag.

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

Wonyeol Lee, Hangyeol Yu, and Hongseok Yang. 2018. Reparameterization Gradient for Non-Differentiable Models. In *Neural Information Processing Systems*.

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. (2019).

Alexander K. Lew, Mathieu Huot, Sam Staton, and Vikash K. Mansinghka. 2023. ADEV: Sound Automatic Differentiation of Expected Values of Probabilistic Programs. (2023).

Tzu-Mao Li, Miika Aittala, Frédo Durand, and Jaakko Lehtinen. 2018. Differentiable Monte Carlo Ray Tracing through Edge Sampling. *Special Interest Group on Computer Graphics and Interactive Techniques in Asia* (2018).

Tzu-Mao Li, Michal Lukáč, Gharbi Michaël, and Jonathan Ragan-Kelley. 2020. Differentiable Vector Graphics Rasterization for Editing and Learning. *Special Interest Group on Computer Graphics and Interactive Techniques in Asia* (2020).

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*.

Christopher Lutz. 1986. Janus: a time-reversible language.

Kazutaka Matsuda and Meng Wang. 2020. Sparcl: A Language for Partially-Invertible Computation. (2020).

Damiano Mazza and Michele Pagani. 2021. Automatic differentiation in PCF. Principles of Programming Languages (2021).

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 *Neural Information Processing Systems*.

Barak A. Pearlmutter and Jeffrey Mark Siskind. 2008. Reverse-mode AD in a Functional Framework: Lambda the Ultimate Backpropagator. *Transactions on Programming Languages and Systems* (2008).

Ken Perlin. 1985. An image synthesizer. *Comput. Graph. (Proc. SIGGRAPH)* (1985).

Walter Rudin. 1953. *Principles of mathematical analysis*.

N. Saheb-Djahromi. 1978. Probabilistic LCF. In *Mathematical Foundations of Computer Science 1978*.

Laurent Schwartz. 1950. *Théorie des distributions*.

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

Chung-chieh Shan and Norman Ramsey. 2017. Exact Bayesian Inference by Symbolic Disintegration. In *Priniciples of Programming Languages*.

Benjamin Sherman, Jesse Michel, and Michael Carbin. 2021. λ_S: Computable Semantics for Differentiable Programming with Higher-Order Functions and Datatypes. (2021).

Peter Shirley. 2020. Ray Tracing in One Weekend. https://raytracing.github.io/books/RayTracingInOneWeekend.html

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

Sam Staton. 2017. Commutative semantics for probabilistic programming. In *European Symposium on Programming*.

Petre Teodorescu, Wilhelm Kecs, and Antonela Toma. 2013. *Distribution Theory: With Applications in Engineering and Physics*. https://doi.org/10.1002/9783527653614

Vladimir N. Vapnik. 1998. *Statistical Learning Theory*. Wiley-Interscience.

R. E. Wengert. 1964. A Simple Automatic Derivative Evaluation Program. *Commun. ACM* (1964).

Sven Woop, Carsten Benthin, and Ingo Wald. 2013. Watertight Ray/Triangle Intersection. *Journal of Computer Graphics Techniques (JCGT)* (2013).

Yuting Yang, Connelly Barnes, Andrew Adams, and Adam Finkelstein. 2022. Aδ: Autodiff for Discontinuous Programs - Applied to Shaders.

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.

Shuang Zhao, Wenzel Jakob, and Tzu-Mao Li. 2020. Physics-Based Differentiable Rendering: From Theory to Implementation. In *SIGGRAPH Courses*.

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*.