i.e., its derivative is the Dirac-delta. Here, AD can correctly determine the derivative at any $x \neq 0$, but its value of zero prohibits the use of gradient descent and does not provide any information about the jump discontinuity. Evidently, this situation remains even if the derivative is averaged across different sample points in the parameter space (cf. Fig. 1, dotted line). More generally, from the literature on infinitesimal perturbation analysis (IPA) [4] it is known that for stochastic discontinuous programs, simple averaging of the pathwise derivatives leads to a biased gradient estimator. One solution is to calculate the gradient of a smooth approximation of H (cf. Fig. 1, dashed line).

The need for differentiating discontinuous functions currently arises in many practical applications, such as neurosymbolic programming [5], program synthesis [6], agent-based simulation [7], and inverse rendering [8]. Thus, the challenge of obtaining gradients over discontinuities has been tackled from several angles: interpolation [7], stochastic (Monte Carlo) estimation [9], and smoothing over discrete randomness [10].

Here, we explore two novel approaches for providing smoothed gradients of imperative programs involving branching control flow. Using our tool DiscoGrad, problems involving *parameter-dependent control flow* formulated in the C++ language (cf. Section 4.5 for a brief description of supported language constructs) can be automatically differentiated to determine their smoothed gradients, enabling the use of gradient-based methods for local optimization. As observed in the training of neural networks with backpropagation, suitable gradient descent algorithms are capable of finding high-quality solutions, even for non-convex functions [11]. Accordingly, our methods assume neither smoothness nor convexity and we evaluate the performance of our proposed estimators on four high-dimensional, discontinuous optimization problems. Our main contributions are:

1. We show how the existing technique of Smooth Interpretation (SI) [12], a form of abstract interpretation, can be combined with AD to obtain gradients across discontinuities in Section 4.1.

2. We provide a clear description of the assumptions made in SI's probabilistic execution of a program and their effects on the output's fidelity in Section 4.2.

3. We propose a novel gradient estimator that avoids SI's assumptions by a combination of AD and Monte Carlo sampling in Section 4.4.

4. We present DiscoGrad[1], our tool to automatically translate C++ programs to an efficiently smoothed, differentiable counterpart using our proposed smoothing methods and other existing gradient estimators in Section 4.5.

5. We provide an extensive evaluation of the estimator's execution times, gradient fidelity and optimization progress against existing sampling-based schemes for local optimization such as REINFORCE [13] and non-gradient based, global optimization methods (genetic algorithm, simulated annealing) in Section 5.

[1] Available at `https://github.com/philipp-andelfinger/DiscoGrad`.

In the following sections, we introduce AD, the smoothing of gradients, and SI (Section 2) and review the related literature (Section 3). Section 4 presents our main results. Finally, we carry out an extensive evaluation (Section 5), concluding with final remarks and future directions (Section 6).

2 Background

In the following, we outline the established work on differentiating programs, with a focus on programs involving branching control flow. Starting from automatic differentiation as the base technique for differentiating programs, we introduce stochastic smoothing as well as smooth interpretation.

2.1 Automatic Differentiation

Automatic differentiation (AD) is a method to compute partial derivatives of computer programs [2, 3]. Treating a program $\mathcal{P}$ as a composition of mathematical functions $\mathcal{P} = f_1 \circ f_2 \circ \cdots \circ f_n$, AD repeatedly applies the chain rule $f_i' = (f_{i+1} \circ f_{i+2})' \cdot f_{i+2}'$ to calculate the program's derivative $\mathcal{P}'$ from the inputs f_n. The well-known backpropagation algorithm [1] widely used in machine learning is a special case of AD.

The literature distinguishes two approaches: *Forward-mode* AD propagates derivative information throughout the (forward) executions of the program by augmenting the involved variables v with a so-called tangent value $\dot{v}$. After each invocation of a mathematical function, this value is updated according to the function's derivative and the chain rule. Thus, at any given point in the execution, the tangent value can be interpreted as the partial derivative $\partial f_j(\mathbf{x})/\partial x_i$ of the operations up to the current point j wrt. the component x_i of the input vector $\mathbf{x}$.

In contrast, *reverse-mode* AD records the arithmetic operations and values involved in the program's forward execution in a so-called *tape* and computes the partial derivatives in a subsequent step by traversing the tape in reverse. While forward-mode AD calculates the partial derivatives of a single input variable wrt. all output variables throughout a single program execution, reverse-mode AD calculates the derivatives of all inputs wrt. a single output based on a single traversal of the tape following the program's termination. The pathwise gradients computed by AD are exact to machine precision. This is in contrast to finite differences methods, whose fidelity depends on finding an appropriate step size.

Many programs encountered in optimization problems involve input-dependent control flow, which typically introduces jump discontinuities. Unfortunately, as AD's purely arithmetic view of a single forward execution of a program cannot account for alternative control flow paths, it produces gradients of limited utility for optimization for such programs (cf. Fig. 1). For stochastic programs, averaging across pathwise derivatives, known as infinitesimal perturbation analysis (IPA), produces an unbiased estimator only if the program permits the exchange of expectation and differentiation [14]:

$$\nabla \mathbb{E}[\mathcal{P}(\mathbf{x})] = \mathbb{E}[\nabla \mathcal{P}(\mathbf{x})]$$

Programs involving input-dependent discontinuities typically violate this condition. For deterministic programs, this form of smoothing can also be used by perturbing the input vector

$\mathbf{x}$ with random noise. Sampling-based estimators applicable to the discontinuous case are discussed in Sections 3.1 and 5.2. As described next, the expected value can alternatively be obtained by a symbolic probabilistic execution.

2.2 Smooth Interpretation

Smooth interpretation (SI) [12] is a method to smooth the output of programs involving discontinuities, making them more amenable to numerical optimization using black-box approaches such as Nelder and Mead's method [15]. Building on abstract interpretation [16, 17] and probabilistic program semantics [18], SI executes a program $\mathcal{P}$ according to a smoothed semantics that approximates the convolution of the program output with a Gaussian kernel $f_{\mathbf{x},\Sigma}$:

$$\widetilde{\mathcal{P}}(\mathbf{x}) \coloneqq \int_{\mathbf{y}\in\mathbb{R}^n} \mathcal{P}(\mathbf{y}) f_{\mathbf{x},\Sigma}(\mathbf{y}) d\mathbf{y}. \tag{1}$$

Here, $\mathbf{x}$ is the program's n-dimensional input vector and Σ a diagonal covariance matrix determining the amount of smoothing.

In SI, each originally scalar input variable $x_i \in \mathbf{x}$ is substituted by a Gaussian random variable X_i with mean $\mu_{x_i} = x_i$ and a configurable standard deviation σ_{x_i} sometimes referred to as the smoothing factor. Now, the arithmetic operations specified in the program operate on and generate random variables. As an approximation, the distribution of any operation's output is in practice again represented by a Gaussian characterized by its mean and standard deviation. Thus, the output variables of a smooth interpretation are, just like the inputs, Gaussian random variables. The result of the interpretation, i.e., of the approximate convolution of the Gaussian with the program at the current input, are the expectations of the output variables.

A key aspect of SI is the handling of conditional branches, as encountered in the form of `if-else` statements. The smoothed semantics require both possible paths to be executed and weighted according to the distribution of the variables involved in the branching condition. This leads to two different distributions for some of the variables. Hence, each variable is represented as a mixture distribution, each element of which represents a Gaussian approximation of the distribution resulting from one of the program's (sequences of) branches. To limit the number of elements of each mixture distribution, a "Restrict" algorithm combines the results of branches in a way that minimizes the deviation from the original overall mixture distributions of the variables.

In general, the exact convolution of a program with a Gaussian is intractable. The approximations made by SI, e.g., assuming the program state to be a Gaussian mixture and limiting it to a finite size through Restrict, enable a practical application of the method and will be further explored in Section 4.2.

3 Related Work

Rooted in the field of non-smooth optimization [19], the (gradient-based) optimization of discontinuous programs has recently seen major interest across many domains, for example machine learning [20], computer graphics [8] and optimal control [9]. Besides gradient-free approaches such as genetic algorithms or the Nelder-Mead method [15], the state of the art in non-smooth optimization includes bundle methods, which augment the subgradient method through the exploitation of past subgradient information [21] and gradient sampling methods exploiting piecewise differentiability [22]. In contrast, we consider smoothed gradients of problems that, while piecewise differentiable, typically provide only zero-vector gradients (cf. Fig. 1). For these, pathwise estimates based on pathwise gradients, as in IPA [4], are insufficient.

In the following, we discuss existing work on differentiating *across* branching control flow directly related to ours. Broader overviews of gradient estimation techniques are given in [23] and [24].

3.1 Sampling-Based Gradient Estimation

Based on the conditional Monte Carlo method for variance reduction, *smoothed perturbation analysis* (SPA) obtains an unbiased gradient estimate through conditional expectations [14]. By choosing suitable problem variables (called characterization) to condition on, the calculation of the expected value is effectively separated into continuous parts, allowing for the interchange of the expectation and differentiation operations (cf. the end of 2.1). While SPA has been widely applied to differentiate discontinuous problems such as certain discrete-event simulations, its applicability is limited by the need to manually determine a suitable characterization to condition on for the problem at hand. For an overview of the many variations of (S)PA refer to [23] (Section 9) and the references therein.

The *REINFORCE* estimator, commonly employed in reinforcement learning, exploits the differentiation rule of the logarithm to eliminate the need for calculating gradients of the program [13]. The gradient is calculated from the plain program output, multiplied by the log-derivatives of the program-specific probability density (cf. Section 5.2). REINFORCE is thus also referred to as the log-derivative trick, likelihood ratio estimator or score function estimator. Similar to the information conditioned on in SPA, the probability density and its log-derivative are problem-specific.

Some problem-independent (black-box) estimators are given in [25], therein referred to as *gradient-free oracles*. The authors analyze the convergence of a scheme introduced in Chapter 3.4 of [26], which estimates the descent direction through directional derivatives, calculated by a randomized version of finite differences where each input dimension is perturbed simultaneously. We will make use of the first of these gradient-free oracles for comparison in the evaluation, where it is also briefly introduced in Section 5.2. Due to the lack of a commonly used name for this estimator, we abbreviate it as PGO (Polyak's Gradient-Free Oracle). A similar construction using non-directional finite differences is proposed in [9].

We note that sampling-based schemes for stochastic programs, like SPA and REINFORCE, can also be applied to the case of deterministic objective functions by introducing artificial perturbations to the program inputs. If these perturbations are sampled from a normal distribution, their estimates approach the gradient of the convolution integral from Eq. (1) as the number of samples approaches infinity. In other words, the gradient estimators, just like SI, approximate the convolution of the gradient with a Gaussian.

3.2 Combination of Sampling and AD

Some recent works propose combinations of sampling-based methods with AD rather than finite differences.

In [10], an unbiased SPA estimator was derived for programs involving discrete randomness and integrated with the AD process in the Julia package `StochasticAD`. This allows for the automatic smooth differentiation of programs that sample from discrete probability distributions with parameters depending on the program inputs. In contrast to our work, this approach does not consider input-dependent discrete control flow.

A recent use of forward-mode AD is found in the "forward gradient", which is determined by sampling over directional derivatives [27]. While their work does not consider differentiation across discontinuities, our methods share the use of forward-mode AD. As we will see in Section 5.3, forward-mode AD incurs only a tolerable overhead in our benchmark problems, while allowing us to efficiently obtain intermediate partial derivatives and avoiding reverse-mode AD's linear dependence of the memory consumption on the program length.

Finally, a sampling-based method proposed in a recent preprint [28] achieves differentiability by applying a static degree of smoothing at each branch and systemically visiting all control flow paths whose probability is within machine precision. Their approach shares with SI the challenge of scaling to problems with non-trivial numbers of branches without introducing biases, the effects of which on SI's fidelity are detailed in Section 4.2 and quantified in Section 5.

3.3 Differentiable Programming Languages and Neurosymbolic Programming

Differentiable programming languages offer semantics that allow for a sound calculation of gradients across entire, typically functional, programs. Abadi presented operational and denotational semantics for a functional language that includes a construct for reverse-mode AD [29]. Discontinuities are ruled out by assuming that constructs such as conditional branches are substituted by smooth approximations by the user.

Some recent languages treat discontinuities natively. Sherman et al. presented semantics for a functional language that covers non-differentiable functions, but requires continuity [30]. The functional language ADEV [31], which targets differentiable probabilistic programming, allows discontinuities to only depend on the program's stochasticity, not on the parameters. This is the same condition that is satisfied after applying the reparametrization trick [32]. A more general approach for handling discontinuities is taken in the functional language by Amorim et al. [33], which relies on distribution theory to soundly express the contribution of discontinuities to the gradient. The resulting integrals are approximated by Monte Carlo sampling. In contrast to these works focused on language semantics, we propose and study concrete gradient estimators that approximate gradients of smoothed imperative programs, specifically targeting the case where discontinuities depend on the parameters.

Among the use cases of differentiable programming languages is the gradient-based *synthesis of symbolic programs*, which offers an alternative to traditional combinatorial program search. The search over symbolic programs is achieved by interpreters that employ continuous relaxations to enable the computation of gradients of the program's output wrt. its parameters, which may represent numerical constants, instructions, or the registers to operate on [6, 34–36]. *Neurosymbolic programming* [5, 37] extends this idea towards programs combining symbolic and neural building blocks. In a recent work, a generalization of the REINFORCE estimator was used to differentiate across symbolic program executions in order to determine parameters leading to control flow paths that adhere to a safety criterion [38]. Although the focus of our own work is on the parameter synthesis for existing programs, the proposed gradient estimators may benefit synthesis approaches as well.

3.4 Domain-Specific Approaches

Methods for gradient smoothing are proposed and applied in many contexts, motivated by a myriad of goals. Here, we discuss relevant publications from the popular fields of neurosymbolic programming, program synthesis, differentiable rendering and simulation-based optimization.

Some recent work achieves smoothing by weighted averaging of variable values across branches as a basis for combining neural networks with traditional algorithms [39, 40], for parameter estimation across agent-based simulations [7], and for antialiasing [41]. In contrast to SI and unbiased black-box estimators, these works lack a well-defined probabilistic semantics and thus do not offer a clear interpretation of the smoothed output.

An alternative approach common in simulation-based optimization is to sample a model's input-output relation to generate a *surrogate model* [42]. Depending on the type of surrogate, e.g., a neural network, the resulting model may be smooth and differentiable. A similar approach has been taken in *systems security* for gradient-based fuzzing [43]. Surrogate models are typically fitted to input-output samples in a black-box fashion, after which gradient estimates are made without involvement of the original model. In contrast, the gradient estimators proposed in our work operate on the original program, making use of its internal structure.

Finally, the field of computer graphics has also shown broad interest in the differentiation of discontinuous programs. Differentiable rendering [8] aims at determining partial derivatives of pixel values with respect to scene parameters, enabling applications such as inverse rendering, i.e., determining scene parameters that best fit real-world image data. Modern rendering techniques are typically based on Monte Carlo sampling of light rays through three-dimensional scenes. The integrals approximated in this manner may carry discontinuities related to the visibility of objects in the scene. This problem can be solved either by explicitly sampling edges that cause the discontinuities [44], or more scalably by applying problem-specific reparametrizations to the objective function so that the position of discontinuities becomes independent of the parameters [45]. An overview of Monte Carlo techniques for differentiable rendering is given by Zeltner et al. [46].

In contrast to the above works, the gradient estimators proposed and evaluated in the remainder of our article target generic imperative programs without reliance on domain-specific problem properties.

4 Smooth Automatic Differentiation

Many optimization problems are naturally formulated as imperative programs. However, the existing work on smooth differentiation lacks a method that 1. focuses on imperative programs involving conditional branching on the input values and 2. makes use of exact pathwise derivatives as determined via AD. In this central section, we will show two possible candidates. First, we provide a common framework for the problem of calculating the smoothed gradient of discontinuous programs where the conditional control flow depends on the input vector. By treating SI as a special case of this framework, its integration with AD becomes straight forward, providing our first (deterministic) estimator. However, this approach is encumbered by the strong assumptions of SI. We explore ways to relax these through information gained by AD, but find that the benefits of these improvements are dwarfed by the effect of SI's restricted representation of the probabilistic program states. Thus, our second (sampling-based) candidate is an AD-powered Monte Carlo approach operating under much lighter assumptions, often yielding significantly more accurate gradient estimates.

4.1 Approach

In our derivations, we consider optimization problems expressed as imperative programs $\mathcal{P}\colon \mathbb{R}^n \to \mathbb{R}$ mapping n input variables to a single output value. Thus, $\mathcal{P}$ is typically a piecewise function. We further assume that discontinuities only arise through branch and loop conditions, noting that common discontinuous functions such as the absolute value function can be rewritten using conditional branching.

As shown in [12], the smoothing of $\mathcal{P}$ with a (multivariate) Gaussian kernel $f_{\mathbf{x},\Sigma}$ can be expressed in terms of a convolution

$$\widetilde{\mathcal{P}}(\mathbf{x}) \coloneqq \int_{\mathbf{y}\in\mathbb{R}^n} \mathcal{P}(\mathbf{y}) f_{\mathbf{x},\Sigma}(\mathbf{y}) d\mathbf{y} = \mathbb{E}_{X\sim\mathcal{N}(\mathbf{x},\Sigma)}[\mathcal{P}(X)]\,, \tag{2}$$

where $\mathbf{x}$ is the program's n-dimensional input vector and Σ a diagonal covariance matrix determining the amount of smoothing. This form of smoothing is sometimes also called the (generalized) Weierstrass transform. By the law of the unconscious statistician, the convolution Eq. (2) can be regarded as taking the expected value of the program's output distribution when executed on normally distributed random variables $X \sim \mathcal{N}(\mathbf{x}, \Sigma)$, see also [23], Section 4. It is important to note that in our case randomness is artificially introduced by moving from $\mathbf{x}$ to X, i.e., our derivations are also applicable to deterministic programs.

A possible strategy to automatically calculate the smoothed gradient $\widetilde{\nabla}\mathcal{P}(\mathbf{x}) \coloneqq \nabla\widetilde{\mathcal{P}}(\mathbf{x}) = \nabla_{\mathbf{x}}\mathbb{E}[\mathcal{P}(X)]$ is to exchange the expectation and gradient operators, as is done in IPA. While this enables a close and automatic approximation, e.g., through a Monte Carlo approach and AD, the equality $\nabla_{\mathbf{x}}\mathbb{E}[\mathcal{P}(X)] = \mathbb{E}[\nabla_{\mathbf{x}}\mathcal{P}(X)]$ only holds if $\mathcal{P}$ is continuous (precise conditions are given in [47]).

In the case of imperative programs with control flow depending on $\mathbf{x}$, $\mathcal{P}$ is not generally continuous, requiring an alternative approach. We observe that the control flow partitions the program's (discontinuous) output $\mathcal{P}(\mathbf{x})$ into several continuous parts. In the probabilistic execution context, we can isolate these continuous parts by conditioning the distribution of $\mathcal{P}(X)$ on the execution path $p \in \{1, \ldots, N\}$. More precisely, let the random variable $\mathfrak{P}$ reflect which control flow path p was taken in the execution of $\mathcal{P}$. Then, using the law of total expectation, we can decompose the integral Eq. (2) into a sum over expectations of its path-specific outputs:

$$\begin{aligned}\widetilde{\mathcal{P}}(\mathbf{x}) &= \mathbb{E}_{X\sim\mathcal{N}(\mathbf{x},\Sigma)}[\mathcal{P}(X)] \\ &= \mathbb{E}_{\mathfrak{P}}\big[\mathbb{E}_X[\mathcal{P}(X)|\mathfrak{P}]\big] \\ &= \sum_{p=1}^{N} \mathbb{P}(\mathfrak{P}=p)\mathbb{E}_X[\mathcal{P}(X)|\mathfrak{P}=p]\,.\end{aligned} \tag{3}$$

In practice, $\mathfrak{P}$ is defined in terms of the conjunction of branching conditions on X encountered along each control flow path p. We note that the idea of using conditional expectations to obtain a smooth objective is very similar to SPA (cf. the overview in [24]). Here, however, the conditioning on $\mathfrak{P}$ is not sufficient to exclusively rely on the IPA estimate of $\mathbb{E}_X[\mathcal{P}(X)|\mathfrak{P}=p]$, because by the definition of $\mathfrak{P}$, the probability $\mathbb{P}(\mathfrak{P}=p)$ still depends on X. Considering Eq. (3), the smoothed gradient of the program is given by:

$$\begin{aligned}\widetilde{\nabla}_{\mathbf{x}}\mathcal{P}(X) &= \nabla_{\mathbf{x}}\mathbb{E}_{\mathfrak{P}}\big[\mathbb{E}_{X\sim\mathcal{N}(\mathbf{x},\Sigma)}[\mathcal{P}(X)|\mathfrak{P}]\big] \\ &= \nabla_{\mathbf{x}}\sum_{p=1}^{N}\mathbb{P}(\mathfrak{P}=p)\mathbb{E}_X[\mathcal{P}(X)|\mathfrak{P}=p] \\ &= \sum_{p=1}^{N}\big(\nabla_{\mathbf{x}}\mathbb{P}(\mathfrak{P}=p)\big)\mathbb{E}_X[\mathcal{P}(X)|\mathfrak{P}=p] \\ &\quad + \sum_{p=1}^{N}\mathbb{P}(\mathfrak{P}=p)\big(\nabla_{\mathbf{x}}\mathbb{E}_X[\mathcal{P}(X)|\mathfrak{P}=p]\big).\end{aligned} \tag{4}$$

Taking into account the sum and chain rules of differentiation, this requires determining the gradients of $\mathbb{P}(\mathfrak{P}=p)$ and $\mathbb{E}[\mathcal{P}(X)|\mathfrak{P}=p]$ wrt. $\mathbf{x}$, with the latter equivalent to the expectation of the pathwise gradient, which can be obtained by IPA and AD. The gradient of the probability of taking a certain path is more difficult to calculate automatically. In the general case where it does not depend on $\mathbf{x}$, it is always $\mathbf{0}$ and can be omitted (and Eq. (4) coincides with the IPA estimator). However, in our case the probability of a taking a branch is influenced by the branching condition, which depends on $\mathbf{x}$, yielding a non-zero gradient vector. The second problem with estimating Eq. (4) is the number of possible paths N, which grows exponentially with the number of branches, leading to an exponential explosion of summation terms.

SI handles both problems by assuming that $\mathcal{P}(X)$ follows a Gaussian mixture distribution of maximum size $M \ll N$. The smoothed execution then involves propagating the first two moments $\mathbf{x}$ and Σ of X and descendant variables through the program. Fig. 2 (left and center) showcases this on a simple example program. At branches, the mixture then naturally arises (for each variable) from the two new possible distributions of the `then` and `else` cases. The weights of the two new mixture elements are calculated by evaluating the cumulative normal density parametrized with the two known moments of the input distribution. To ensure that at most M control paths are

	Program	State	AD Operations
1	`input x;`	weight w: $\{X \sim \mathcal{N}(\hat{\mu}_X \coloneqq \mathbf{x}, \hat{\sigma}_X^2)\}$	$\dot{\mu}_X = 1$
2	`y = 3 * x;`	w: $\{X \sim \mathcal{N}(\hat{\mu}_X, \hat{\sigma}_X^2), Y \sim \mathcal{N}(3 * \hat{\mu}_X, \square_1)\}$	$\dot{\mu}_Y = 3 * \dot{\mu}_X = 3$
3	`y = y + 1;`	w: $\{X \sim \mathcal{N}(\hat{\mu}_X, \hat{\sigma}_X^2), Y \sim \mathcal{N}(3\hat{\mu}_X + 1, \square_2)\}$	
4	`z = 0;`	w: $\{X \sim \mathcal{N}(\hat{\mu}_X, \hat{\sigma}_X^2), Y \sim \mathcal{N}(3\hat{\mu}_X + 1, \square_2), Z = 0\}$	$\dot{z} = 0$
5	`if (y < 0) {`	$w_1 \coloneqq \Phi(\frac{\mu_Y - 0}{\sigma_Y})$: $\{\dots\}, w_2 \coloneqq 1 - \Phi(\frac{\mu_Y - 0}{\sigma_Y})$: $\{\dots\}$	$\dot{w}_1 = \varphi(\frac{\mu_Y}{\sigma_Y})\dot{\mu}_Y$
6	`z = 1;`	w_1: $\{X \sim \dots, Y \sim \dots, Z = 1\}, w_2$: $\{\dots\}$	
7	`}`	join the two weighted branches into: w_1: $\{X \sim \dots, Y \sim \dots, Z = 1\}$, w_2: $\{X \sim \dots, Y \sim \dots, Z = 0)\}$	
8	`output z;`	$\mathbb{E}[Z] = w_1 * 1 + w_2 * 0$	$\partial\mathbb{E}[Z]/\partial\hat{\mu}_X = \dot{w}_1 * 1$

The variance $\square_1$ can be calculated on demand (e.g. in line 5) as $\sigma_Y^2 = (\dot{\mu}_Y)^2 \hat{\sigma}_X^2$ using the tangents or immediately using an approximation disregarding correlations; analogously for $\square_2$.

Figure 2: *Example program (left) execution showcasing the probabilistic semantics of SI (center) and their integration with forward-mode AD (right). Only relevant AD operations are shown. The tangents $\dot{v}$ denote the (generally partial) derivative $\partial v/\partial\hat{\mu}_X$ wrt. the mean μ of the normally distributed random variable $X \sim \mathcal{N}(\hat{\mu}_X, \hat{\sigma}_X^2)$. The hat $\hat{}$ symbol indicates an input value. Upon initialization, the (here scalar) input vector $\mathbf{x}$ is taken as the mean $\hat{\mu}_X$; φ and Φ denote the normal distribution's probability and cumulative density functions respectively. Note that in this example the pathwise derivative is 0, but through the combination of SI and AD, the derivative wrt. the branching condition is obtained.*

carried along, selected mixture elements are merged (cf. Section 4.3). Thus, both the calculations leading to $\mathbb{P}(\mathfrak{P} = p)$ and $\mathbb{E}[\mathcal{P}(X)|\mathfrak{P} = p]$, which coincide with the weight and mean of the mixture element corresponding to p, are smooth functions of the program input. We make use of this property by differentiating SI's approximations of the per-path weight and output via AD. The right-hand side of Figure 2 shows the differentiation by the example of forward-mode AD, which tracks the derivatives wrt. the first moments μ_X of X. We now briefly consider some interesting opportunities opened up by this integration of AD.

4.2 Relaxing SI's Assumptions

The method of SI [12] imposes several major assumptions upon programs, trading off fidelity for execution speed:

1. *No interdependencies*: The inputs to any operation are assumed to be independent normal distributions.
2. *Everything is Gaussian*: The output of any operation is assumed to follow a normal distribution, which is not true in the general case, even assuming 1). As a consequence, within a particular branch, the distribution of a variable X_i (and its dependent variables) contains values excluded by the branching condition $X_i \leq c$.
3. *No truncation at branches*: When splitting the state at a branching point, the resulting two distributions are approximated by scaling the weight of the Gaussian mixture elements. Thus, their means and standard deviations remain unaltered, where in reality the results are the lower and upper tails of Gaussian distributions, i.e., truncated Gaussians.
4. *Fixed-size state representation*: The program state across all possible control flow paths, whose number is exponential in the number of branches, is approximated by a fixed-size Gaussian mixture.

While these assumptions permit a reasonable approximation of small programs with mostly affine operations, they cause substantial deviations for larger programs. When integrating AD as described above, the resulting gradient estimates can thus become inaccurate and noisy (cf. Section 5). In the following, we briefly sketch how the additional information available via AD makes it possible to relax assumptions 1 to 3, but show that the effects of 4 dominate the error.

Improvements regarding assumptions 1 and 2 concern operations of the form $C = h(A, B)$. We restrict this example to binary operations, as the n-ary case is analogous. As shown in Figure 2, SI determines the mean μ_C and variance σ_C^2 of C from $A \sim \mathcal{N}(\mu_A, \sigma_A^2)$ and $B \sim \mathcal{N}(\mu_B, \sigma_B^2)$:

$$\begin{aligned} \mu_C &= h(\mu_A, \mu_B) \\ \sigma_C^2 &= \left(\frac{\partial\mu_C}{\partial\mu_A}\right)^2 \sigma_A^2 + \left(\frac{\partial\mu_C}{\partial\mu_B}\right)^2 \sigma_B^2 \end{aligned} \tag{5}$$

This is a standard result of uncertainty propagation (UP) [48] and exact if h is an affine function and A and B are independent. Linear dependencies (correlations) between A and B can be accounted for by using the information obtained by AD (cf. also the explanation in Fig. 2):

$$\sigma_C^2 = \sum_{i=1}^{n} \left(\frac{\partial \mu_C}{\partial \mu_{X_i}} \right)^2 \sigma_{X_i}^2, \tag{6}$$

where n is the number of inputs and the gradient is provided by AD (see Appendix B for a derivation). Intuitively, this calculates the variance of C from the variance of the inputs X, based on the transformations captured by the gradient since the start of the program and leading to g, which implicitly accounts for the covariance between A and B. Closer and automatically differentiable approximations of the distributions resulting from non-affine operations could be determined using higher-order Taylor approximations [41], at the cost of additional computational overhead.

Lifting Assumption 2 poses the largest challenge, as the assumption of Gaussian distributions for all variables is the main enabler of an easy integration with AD. To exclude invalid intervals from the variables' distributions on each path state, interval bounds could be carried along with the mean and variance for each mixture element [38]. Moving beyond Gaussian distributions would require the tracking and updating of higher-order moments or explicit representations of the distributions' shapes.

Assumption 3 can be lifted by calculating the truncated Gaussian distributions resulting from a branch and approximating them by their first two moments. This requires obtaining the dependence of every variable on the branching condition to determine the point of truncation for each variable, which may be approximated by first-order dependencies determined via AD.

To explore the effects of the enhancements on SI's fidelity, we carried out preliminary experiments with correlation-preserving variance calculation via UP and approximating the truncation at branches. However, we found that the error incurred by restricting the state representation to a small maximum number of paths (cf. Assumption 4) dwarfs the benefits of these enhancements. Figure 3 showcases this on a simple synthetic program (cf. Appendix C) by comparing the combination of the original SI with AD, its combination with UP (cf. Eq. (6)), and an unbiased stochastic approximation of the exact convolution. While UP improves the fidelity of the derivative to the convolution, restricting the number of tracked paths introduces significant jumps and deviations from the reference. The erratic results are caused by the decision which paths to merge, which is not smooth with respect to the program inputs: for a small change in the input value, an entirely different set of merging decisions may be made. As a consequence of this observation, and since the state restriction dominates SI's computational cost, we abstain from exploring the above enhancements further and instead focus on the Restrict [12] algorithm.

4.3 State Restriction Strategies

Tracking the effects of every possible branch in a program execution quickly leads to a state explosion that renders the execution of non-trivial programs intractable. Thus, SI employs an algorithm to merge branches, enabling the restriction of the state size to a user-defined limit M. The restriction is achieved by identifying and subsequently merging two elements of the Gaussian mixture, such that the cost defined as the deviation from the original overall distribution is minimized, which involves calculations of new means and standard deviations. As noted in [12], this algorithm is optimal in the sense that it minimizes the deviation from the overall original mixture. However, the algorithm is computationally intensive, requiring an iteration across the variables of all combinations of path states to determine a pair of paths to merge. Further, merging two dissimilar paths can result in high-variance mixture elements and unreachable program states even according to a strict probabilistic semantics. For example, refer to line 7 in Fig. 2. If the two states were merged, this would result in $Z \sim \mathcal{N}(0.5, 0.5)$ assuming that $w_1 = w_2 = 0.5$. In other words, Z's two possible crisp integer values are merged into a single Gaussian, which can severely affect subsequent operations. Here, we explore three alternate heuristics with different tradeoffs in fidelity and computational cost.

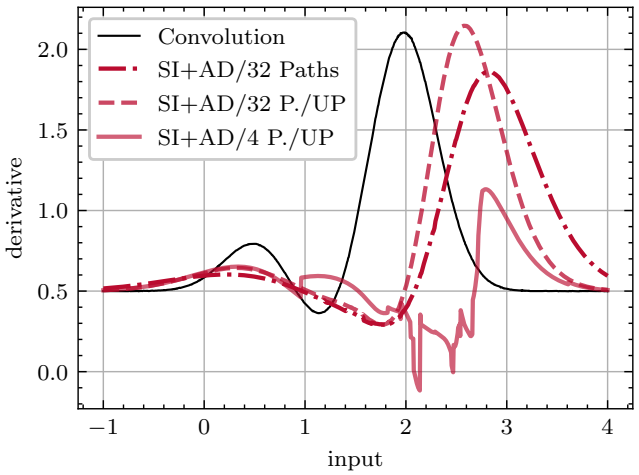


Figure 3: *Comparison of gradient fidelity wrt. the convolution with the original SI proposal with 32 tracked control flow paths and with correlation-preserving variance calculation using AD (uncertainty propagation, UP). When reducing the number of tracked paths to a small subset, as would be required for larger programs, the assumption of a fixed size mixture dominates the error. The non-smooth merging of mixture elements causes the gradient to jump or even assume the wrong sign, which is problematic for gradient descent.*

In the Restrict algorithm, the cost of merging two variables is determined by the difference in the mixture element's moments and by the paths' weights. To avoid merging dissimilar states, we can select paths to merge solely on the moments and ignore the path weights. We refer to this strategy as Ignore Weights (IW).

On the other hand, by only considering the paths' weights, the expensive pair-wise comparisons among the paths' states can be avoided. In this strategy, which we refer to as Weights Only (WO), the weight is used as a proxy for each variables' contribution to the gradient. Assuming sufficiently similar mean values across paths, the weight is a good indicator of a mixture elements' contribution to the final expected value

Eq. (4). Most importantly, since every variable on a given path shares the same weight, a merge decision is reduced to determining the pair of paths with the lowest weights. A more radical strategy that aims for improved performance and low variance at the same time is to discard (Di) paths with the lowest weights entirely, avoiding the merging of variables. The disadvantage of this strategy is that value and gradient information from discarded paths is lost entirely, whereas merging of path states retains the available gradient information across all paths, albeit in an aggregated form.

Although IW, WO, and Di are suboptimal in the theoretical sense, we will see in our evaluation (cf. Section 5) that the decrease in overhead and/or variance obtained through these strategies can lead to faster optimization progress than the strategy proposed by Chaudhuri et. al., which we abbreviate as Ch.

4.4 Monte Carlo Approach to Smooth Differentiation

The assumption of Gaussian distributions and restricting the state to a small subset of possible paths (cf. assumption 4) cause the results of SI to deviate from the exact convolution of the program's output with a Gaussian kernel. As a consequence, SI's estimations of the program's output and its gradient may be significantly biased. In the following, we present an alternative approximation of the probabilistic program semantics based on Monte Carlo sampling and AD.

The key idea is to revisit the decomposition of the program's convolution with a Gaussian into a sum over the control flow paths. From Eq. (4) it follows that the partial derivative wrt. dimension k of the mean $\mathbf{x}$ for a program with N control flow paths is given by:

$$\begin{aligned}\frac{\partial \mathcal{P}(X)}{\partial x_k} &= \sum_{p=1}^{N} \frac{\partial \mathbb{E}[Y_p]\, w_p}{\partial x_k} \\ &= \sum_{p=1}^{N} \frac{\partial \mathbb{E}[Y_p]}{\partial x_k} w_p + \sum_{p=1}^{N} \mathbb{E}[Y_p]\, \frac{\partial w_p}{\partial x_k}.\end{aligned} \tag{7}$$

For readability, we abbreviate the probability of taking the control flow path p with the weight $w_p \equiv \mathbb{P}(\mathfrak{P} = p)$ and the expected value of the output distribution conditioned on p as $\mathbb{E}[Y_p] \equiv \mathbb{E}[Y|\mathfrak{P} = p]$.

We now consider the approximation using Monte Carlo sampling, i.e., through repeated execution of the program on inputs drawn from the input distribution. By definition, the output distribution Y_p of an individual path p for fixed $\mathbf{x}$ does not depend on branch conditions. Averaging across the pathwise derivatives of samples restricted to path p using the indicator function $\mathbf{1}$ leads to the following unbiased estimator:

$$\frac{\partial \mathbb{E}[Y_p]}{\partial x_k} = \lim_{S\to\infty} \frac{1}{n_p} \sum_{s=1}^{S} \frac{\partial y_s}{\partial x_{k,s}} \mathbf{1}_{\mathfrak{P}(\mathbf{x}_s)=p}, \tag{8}$$

where S is the number of samples, the $x_{k,s}$ are sampled from $X_k \sim \mathcal{N}(x_k, \sigma^2)$, y_s and $\mathfrak{P}(\mathbf{x}_s)$ are the output and chosen control flow path when running $\mathcal{P}$ on the sample $\mathbf{x}_s$, and n_p is the number of samples on path p.

Each sample takes exactly one of the N paths, and the weight w_p of a path p is its probability of being taken, i.e., as S tends to infinity, n_p/S approaches w_p. Hence, a sampling-based form of the first summation in Eq. (7) is simply:

$$\begin{aligned}\sum_{p=1}^{N} \frac{\partial \mathbb{E}[Y_p]}{\partial x_k} w_p &= \lim_{S\to\infty} \sum_{p=1}^{N} \frac{1}{n_p} \sum_{s=1}^{S} \frac{\partial y_s}{\partial x_{k,s}} w_p \mathbf{1}_{\mathfrak{P}(\mathbf{x}_s)=p} \\ &= \lim_{S\to\infty} \frac{1}{S} \sum_{s=1}^{S} \frac{\partial y_s}{\partial x_{k,s}}.\end{aligned} \tag{9}$$

The remaining difficulty lies in the term $\partial w_p/\partial x_k$ from Eq. (7), which depends on the distribution of the branch conditions and their sensitivities to the program inputs. Given a branch statement of the form "if (C <= d)", with d a constant, we refer to the random variable $B \coloneqq C - d$ as the branch condition. The branch condition is true, i.e., the branch is taken, with probability $\mathbb{P}(B \leq 0)$, which is the cumulative distribution function of B evaluated at 0. In SI, this probability is estimated based on the parameters of the assumed Gaussian distribution of B, and its derivative can thus be determined transparently via AD. To determine the derivative in the general case, where B may depend arbitrarily on $\mathbf{x}$, let g be the function that describes the dependence of B on $\mathbf{x}$, i.e.: $g(\mathbf{x}) \coloneqq B$. Using the chain rule, we have:

$$\frac{\partial F_{g(\mathbf{x})}}{\partial x_k} = \frac{\partial F_{g(\mathbf{x})}}{\partial g(\mathbf{x})} \frac{\partial g}{\partial x_k}. \tag{10}$$

Here, $\partial F_{g(\mathbf{x})}/\partial g(\mathbf{x}) = f_{g(\mathbf{x})} = f_B$ is the PDF of the condition, which we can estimate based on the samples, e.g., via kernel density estimation. The second term $\frac{\partial g}{\partial x_k}$ is the derivative of the branch condition wrt. x_k. In a sampling-based regime, the value of this term at 0 can be approximated by averaging the exact AD derivatives for realizations in a neighborhood $[-\delta, \delta]$. Using the approximation of the product from Eq. (10) and denoting the sampling-based estimate of the branch conditions' PDF as $\tilde{f}_B$, we arrive at the following Monte Carlo estimator for the derivative of the path weight of the "true" case at a branch encountered on path p:

$$\frac{\partial w_p}{\partial x_k} \approx \frac{-\tilde{f}_B(0)}{S} \sum_{s=1}^{S} \frac{\partial g}{\partial x_{k,s}}(\mathbf{x}_s) \mathbf{1}_{\mathfrak{P}(\mathbf{x}_s)=p, |g(\mathbf{x}_s)|<\delta} \tag{11}$$

and analogously with positive sign in the "false" case. We refer to this estimator as the DiscoGrad Gradient Oracle (DGO).

The derivation above assumes that the program behaves deterministically across the samples. However, since the smoothed gradients permit simple averaging, stochasticity can be introduced across estimation passes.

Each sample $\mathbf{x}_s$ may encounter several branches along each dimension k. For sufficiently large δ, each $x_{k,s}$ may thus appear in the summation of Eq. (11) for multiple branches. This situation is not accounted for since the contribution of each individual sample only captures the output's derivative on its encountered control flow path without explicitly considering the (transitive) effects of taking alternative paths. We treat this case heuristically by assigning an affected sample the path weight derivative wrt. the relevant dimension of the branch with the most equal distribution of samples between $[-\delta, 0]$ and $(0, \delta]$.

```
1  const int num_inputs = 1;
2  #include "include/discograd.hpp"
3
4  adouble _DiscoGrad_f(DiscoGrad<num_inputs> &dg,
5                       aparams &p) {
6    // user code
7    sdouble x({p[0], dg.get_variance()}), y;
8    if (x < 0) y = 0; else y = 1;
9    return y.expectation();
10 }
11
12 int main(int argc, char **argv) {
13   DiscoGrad<num_inputs> dg(argc, argv);
14   DiscoGradFunc<num_inputs> func(_DiscoGrad_f);
15   dg.estimate(func);
16   return 0;
17 }
```

(a) Original program using the DiscoGrad API.

DiscoGrad

```
7  ...
8  si_stack.prepare_branch(x < 0);
9  { si_stack.enter_if(); y = 0; }
10 { si_stack.enter_else(); y = 1; }
11 ...
```

(b) Smoothed branch for the DGSI backend.

```
7  ...
8  dg.prepare_branch(x - 0);
9  if (x < 0) y = 0; else y = 1;
10 ...
```

(c) Smoothed branch for the DGO backend.

Figure 4: *Example DiscoGrad program (a) and the smoothed versions of the contained branch for SI (b) and DGO (c).*

For the problems considered in our evaluation, we found that it sufficed to set δ to ∞, indicating that the benefit of collecting more samples per dimension and branch outweighs the bias in the pathwise derivative introduced by choosing a larger neighborhood. Even so, deeply nested branches can lead to only small numbers of samples being observed at each branch. This problem can be mitigated by translating nested branches to sequential branches, which was straight forward for the problems considered in Section 5.

4.5 DiscoGrad: Smooth Differentiation of C++ Programs

DiscoGrad is a tool to translate programs written in a subset of C++ to a smooth representation in order to execute them according to an approximate probabilistic semantics and to estimate the smooth programs' gradient. The tool is comprised of two main parts: a set of header-based back-ends that implement AD, SI, and AD-guided Monte Carlo gradient estimation on one hand, and source-to-source transformations implemented via the LLVM compiler toolchain to generate estimator-specific code that makes use of the respective back-end. To allow for a meaningful evaluation of execution times, the code was carefully profiled and optimized using standard techniques such as early returns, avoiding unnecessary copy operations, and minimizing dynamic memory allocation.

Fig. 4a depicts a basic DiscoGrad program that implements the Heaviside function. In the main function, instances of `DiscoGrad` and `DiscoGradFunc` are created as interfaces to the chosen estimator's backend. In this example, the user function `_DiscoGrad_f()` initializes smooth variables of type `sdouble` using an input mean value and variance, branches on the smooth variable `x`, and returns the resulting expectation of `y`. The listed program is the input to the smoothing transformation, which is applied to any user function prepended with the string `_DiscoGrad_`. After compilation, the smoothed program outputs the expectation returned by the user function along with its gradient.

At present, beside crisp code, which remains unmodified, DiscoGrad's smoothing transformation supports mathematical operations and assignments on any combination of crisp and smooth variables, conditional branching, loops, functions on smooth variables, references and pointers to smooth variables as well as simple uses of containers. Among the features currently not implemented are smooth versions of the ternary operator, switch statements, and global variables. DiscoGrad's features and limitations are documented in our repository[2], where the full source code and the programs used in the evaluation in Section 5.1 can be accessed.

4.5.1 AD Implementation

DiscoGrad includes an implementation of forward-mode AD based on operator overloading. At first appearance, reverse-mode AD might seem preferable, since it covers the common case of differentiating programs mapping large numbers of inputs to a single output in a single reverse pass. However, forward-mode AD provides two key benefits in our problem setting: firstly, its memory consumption is independent of the number of operations carried out by the program. In contrast, reverse-mode AD maintains a tape in memory that grows linearly with the number of operations. Furthermore, in combination with SI, the memory consumption for the tape multiplies with the number of tracked control flow paths. Secondly, forward-mode AD allows us to determine a variable's derivatives with respect to the inputs at any time throughout a program's execution without further cost, in contrast to the reverse passes that would be needed with reverse-mode AD. This allows us to efficiently determine the derivatives of branch conditions to the inputs in our implementation of the Monte Carlo estimator described in Section 4.4.

In our implementation, a variables' tangents (partial derivatives) with respect to the inputs are carried along as arrays, allowing for compiler vectorization of the tangent operations[3]. To exploit the frequent case of variables carrying at most one non-zero tangent, full tangent arrays are allocated lazily only once required. A pool of tangent arrays is maintained to avoid frequent explicit memory allocations and deallocations when variables are created or destroyed. While naive forward-mode AD incurs a slowdown factor equivalent to the input dimension, we will see in Section 5.3 that these simple implementation-level design decisions suffice to reduce the AD overhead to a more tolerable level.

[2] `https://github.com/philipp-andelfinger/DiscoGrad`

[3] We verified that in the code generated by Debian clang, version 11.0.1-2, the tangent operations were translated to AVX instructions.

4.5.2 SI implementation

Executing a program according to the probabilistic semantics of SI deviates from a regular "crisp" execution in two main regards: firstly, when encountering a branching statement, both the `then` and the `else` case is visited. Doing so repeatedly generates an exponential number of *path states* representing the variables' values resulting from different branch sequences, which is restricted to a configurable maximum number to constrain the memory consumption and execution time. Secondly, the mathematical, logical, and comparison operations of the original program are widened to operate on all present path states. On each path state, operations originally carried out on scalars are executed on Gaussian distributions represented by their first two moments. DiscoGrad takes a similar approach to the original implementation of SI in the EULER tool [49], but allows for the use of smoothed variables with C++ features such as containers and references, integrates SI with AD, and implements the state restriction strategies presented in Section 4.3.

The key idea is for the source-to-source transformation to flatten the program's control flow across all branches so that all branch bodies are visited and to delegate the management of the variable states in the different control flow paths to our back-end library. In the back-end, the program state is managed by an instance of type `SiStack`, which holds the path states that are active at the programs' scopes, with the state in the currently visited scope (e.g., the `then`-body of an `if-else` statement) at the top (cf. Fig. 4b). The type `sdouble` (smooth double) overloads the operations defined for `double`, carries them out for all active paths in the current scope, and substitutes originally crisp mathematical operations by operations on the moments of Gaussians. Each path state's weight is a floating-point variable subject to differentiation via our AD implementation. Similarly, the mean and optionally also the variance of each Gaussian value representing a mixture element are differentiable variables.

Given an `if-else` statement with a condition `l <= r` where at least one of `l`, `r` evaluates to type `sdouble`, DiscoGrad determines on each active path state p the conditional probability $\mathbb{P}_p(l - r \leq 0)$ of entering the branch. The other inequality operators are handled analogously. For each existing path state, two new path states representing the `then` and `else` cases are created with the new weights determined by multiplying the conditional probability and its complement with the original path's weight. Paths with weights below a configurable threshold (set to 10^{-20} in the evaluation) are discarded due to their negligible impact on the program's output and to avoid arithmetic underflow. Smooth loop constructs are supported by exiting a loop once no path state with sufficient probability enters another iteration.

At a branch, DiscoGrad first halves the number of active paths using one of the state restriction strategies Ch, IW, WO, and Di described in Section 4.3 to make it possible to generate new paths. As we will see in our evaluation, the overhead incurred by this step is decisive for the overall execution time under SI. For the computationally expensive Ch strategy, we cache previously computed merge costs among the path states as long as they remain unchanged and maintain a priority queue to efficiently select the next pair of states to be merged.

4.5.3 DiscoGrad Oracle (DGO) Implementation

Our Monte Carlo estimator DGO (cf. Section 4.4) estimates the gradient of the smoothed program based on a series of runs on perturbed inputs. Since the individual samples follow the control flow of the original crisp program, an implementation of DGO is vastly more lightweight compared to SI. Here, the type `sdouble` maps to a single scalar floating-point number differentiable via our AD implementation, without any probabilistic semantics.

The source-to-source transformation simply prepends each `if-else` statement with a condition the form `l <= r` by a call that passes $l - r$ to our back-end, and analogously for the remaining inequality operators (cf. Fig. 4c). For each sample, the condition values in a neighborhood $[-\delta, \delta]$ and their derivatives are gathered in order to estimate the weight derivative according to Section 4.4. The mean of the conditions' gradients is computed on the fly via AD, as is the overall pathwise gradient of the sample. Having collected the branch conditions, we estimate the probability density of the condition values at each branch using kernel density estimation with a Gaussian kernel.

Subsequently, we iterate over the branches encountered by each sample to assign the sample the path weight derivatives along its path. Finally, the pathwise gradients and the path weight derivatives are combined according to the summation of Equation Eq. (7) to yield the DGO estimate of the gradient.

5 Evaluation

Bringing everything together, we perform an extensive empirical evaluation of the proposed gradient estimators with respect to their execution times and fidelity, comparing them to two other estimators. We combine them with the Adam gradient descent procedure to solve optimization problems and also compare against global optimization techniques.

In all experiments, we distinguish two types of replications. A single run of a program at a given solution is a *microreplication*. For stochastic programs, several microreplications are carried out, averaging the partial derivatives across microreplications. A *macroreplication* is a replication of an entire experiment (optimization, parameter sweep) starting from an initial solution and spanning a series of microreplications. When multiple macroreplications are executed, all estimators are configured with the same sequence of starting solutions across macroreplications.

All measurements of execution times and optimization progress were carried out on a machine equipped with a 16-core AMD Ryzen 9 7950X processor and 64 GiB RAM running Debian GNU/Linux 11, running at most 16 processes in parallel.

5.1 Selected Non-Smooth Optimization Problems

Based on well-known optimization problems from the literature and practical applications, we implemented four evaluation problems dominated by discontinuous control flow. An overview of the problems is provided in Table 1.

5.1.1 Traffic Lights

The TRAFFIC dxd problem is a deterministic macrosimulation of a *road network* using a simple form of the cell transmission

Problem	Randomness	Parameters	Objective Function
TRAFFIC dxd	Deterministic	d^2 traffic light offsets with $d \in \{2, 5, 10, 20, 40\}$	Traffic flow
AC	Stochastic	82 neural network weights	Loss determined by deviation from target temperature and energy cost
HOTEL	Stochastic	56 products' booking limits	Revenue
EPIDEMICS	Stochastic	Recovery rate, initial infection probability, 100 location-specific infection probabilities	Mean squared error wrt. reference over steps and locations

Table 1: *Overview of the benchmark problems.*

model [50] covering a two-dimensional grid of four-way intersections with dimensions $d \times d$ over d time steps. Vehicles are represented as *populations* (vehicle counts) per lane. At each time step, d new *vehicles* are created at the northern and/or western border of the grid, each with a general movement direction to the opposite border interrupted by rare predetermined turns. The traffic flow at each intersection is organized by a signal that alternates between green and red phases of two time steps each for the horizontal and vertical lanes, allowing at most one vehicle per step and lane to advance to the next intersection. The parameters of this problem are the d^2 traffic light *phase offsets* for each intersection and the objective is to maximize the total *number of intersections passed* by the vehicles throughout the simulation. Here, discontinuities arise from the discrete switching of the traffic signals.

5.1.2 Air Conditioning

The second problem, taking inspiration from [38] and later referred to as AC, considers the *optimal control* of an air conditioning unit by a *neural network*. An insulated room with a single window is simulated over 10 time steps. Over time, the *temperature* of the room gradually approaches the *outside temperature* according to the room's *insulation*. With a probability of 5% per step, the window is opened, decreasing the insulation drastically. The task of the AC is to keep the temperature of the room as close to a chosen *target* as possible by deciding on its on/off state and the cooling power (2 neural network outputs), given the target, previous, and new temperature together with its previous action (5 neural network inputs). Each time the AC activates, an *energy penalty* is incurred. For each simulation, the initial, target, and outside temperature, as well as the insulation are chosen randomly to force the network to generalize. Considering the feedback from the previous' time steps inputs and outputs, this problem can be viewed as training a recurrent neural network. The problem parameters are the 82 weights of the one layer deep network and the objective is the minimization of the loss function defined as the sum of the average mean squared error over time and the energy penalties. Discontinuities arise both from the on/off state of the AC and the discrete randomness of the window opening event.

5.1.3 Hotel Booking

The third problem is a revenue maximization problem from the SimOpt benchmark suite [51, 52] and considers the *optimal booking* of a hotel. The hotel offers 100 rooms via 56 "products" reflecting the guests' arrival days within a week, the lengths of their stays, and two different rates: a rack rate and a discount rate. Over the course of one week, guests arrive and request products according to per-product Poisson processes with predefined rates. The number of bookings is restricted by a per-product *booking limit*. Whenever a requested product's booking limit is greater than zero, the guest can be accommodated, and the booking limits for all products are decremented to account for the reduced room availability on the days covered by the product. Here, the goal is to maximize the revenue by adjusting the products' interdependent booking limits to the guests' arrivals. The parameters are the 56 booking limits, each with an upper bound of 100. Discontinuities are caused by the discrete decisions whether a guest's request for a product can be satisfied.

5.1.4 Disease Spread

The final problem is an *agent-based* SIR (susceptible, infected, recovered) simulation of *disease spread* similar to [7]. Over 25 time steps, a population of 200 individuals moves on an undirected *graph topology* generated as a random geometric graph with 100 nodes (locations) and average degree ≈ 7. Initially, agents are infected with a certain probability. Upon infection, a recovery time is scheduled with a delay drawn from an exponential distribution. Throughout the simulation, agents move along predetermined paths along the edges in the graph, being infected by their neighbors at the same location and recovering at their scheduled time. The probability of being infected depends on a per-location coefficient. The 102 parameters of this problem are the initial infection probability, the mean recovery time relative to the simulation end time, and the location-specific infection probabilities. The objective is to fit a previously generated progression of the epidemic by minimizing the mean squared error between the distribution of agent states at each location and the states recorded in the reference trajectory. Discontinuities arise due to the discrete randomness in the infections, which occur via Bernoulli trials, as well as the discrete recovery event.

5.2 Gradient Estimators

The objective of the evaluation is to determine the utility of our gradient estimators DGSI and DGO for solving the optimization problems defined in the previous section by gradient descent. We chose the popular Adam optimizer due to its well-known applicability to noisy gradient estimates [53]. As points of reference, we employ the PGO [25] and REINFORCE (RF) [13] estimators. Smooth gradients are calculated by sampling over a set of normally distributed random perturbations of the program parameters, making REINFORCE also applicable to deterministic problems such as TRAFFIC (cf. Appendix A for a complete derivation). More formally, we use the following reference estimators to obtain the smoothed

gradient $\widetilde{\nabla}_{\mathbf{x}}$:

$$\widetilde{\nabla}_{\mathbf{x}}^{\texttt{PGO}}\,\mathcal{P}(\mathbf{x}) = \lim_{S\to\infty}\frac{1}{S}\sum_{s=1}^{S}\frac{\mathcal{P}(\mathbf{x}+\sigma\mathbf{u}_s)-\mathcal{P}(\mathbf{x})}{\sigma}\mathbf{u}_s \quad (12)$$

and

$$\widetilde{\nabla}_{\mathbf{x}}^{\texttt{RF}}\,\mathcal{P}(\mathbf{x}) = \lim_{S\to\infty}\frac{1}{S}\sum_{s=1}^{S}\frac{\mathcal{P}(\mathbf{x}+\sigma\mathbf{u}_s)}{\sigma}\mathbf{u}_s, \quad (13)$$

where $\mathbf{u}_s \sim \mathcal{N}(\mathbf{0},\mathbf{I})$ are iid. variates of the standard multivariate normal distribution and σ is the smoothing factor (i.e., standard deviation). Notice how this formulation of REINFORCE, where the stochasticity is introduced by random perturbations, is very similar to PGO.

We note that, for stochastic problems, REINFORCE may also directly exploit the problem-specific stochasticity, but only if the log-derivative is known. As our problems allow arbitrary probability distributions, the log-derivative is difficult to obtain in general. Further, our proposed mechanisms and PGO work fully automatically, i.e., in a black-box setting. We thus evaluate REINFORCE in the same setting. Additionally, PGO can be combined with a random search as described in [25, 26]. The random gradient-free search algorithm described therein can be viewed as only taking one sample from PGO per step of descent. In this evaluation, we limit our scope to performing *gradient* descent, leaving the random search and possible interesting integrations with the Adam optimizer as future work (cf. Section 6).

This leaves us with four optimization procedures. For brevity, we only show the estimator and number of samples (or tracked control flow paths), as they are all combined with Adam, for example "PGO/100" for PGO estimator with 100 samples or "DGSI/Di/4" for our SI implementation with four tracked control flow paths, using the Discard restriction strategy (cf. Table 2).

In the following, we perform three types of evaluation. First, we evaluate the scalability in terms of computation time and memory (Section 5.3) and the gradient error (Section 5.4). Testing the practical impacts of the former two, we conclude with an evaluation of the optimization performance (Section 5.5). For the evaluation of optimization performance we also compare two popular global optimization algorithms that would typically be applied to solve our non-smooth and non-convex problems: a standard *genetic algorithm* (GA) with elitism (as provided by the `pyeasyga` module[4]) and *simulated annealing* (SA) (by porting the version from the Ensmallen library[5] to Python). Our measurements of execution time and optimization progress over time exclude process startup times to avoid disadvantaging the existing baseline approaches without AD, which are typically faster and thus more strongly impacted by startup times.

5.3 Execution Time

All of the AD-based gradient estimators incur an overhead over a crisp program execution without AD. Here, we evaluate the scaling of the estimators' wall-clock execution time

[4] `https://github.com/remiomosowon/pyeasyga`

[5] `https://ensmallen.org/docs.html#simulated-annealing-sa`

Name	Estimator	References
DGSI	DiscoGrad implementation of our combination of SI with AD using the	Section 4.1 ff. and [12]
DGIS/Di	Discard restriction strategy (discard branches based on weight).	
DGIS/WO	Weights Only restriction strategy (merge highest-weighted branches).	
DGIS/IW	Ignore Weights restriction strategy (merge only based on moments).	
DGO	DiscoGrad implementation of our Monte Carlo estimator.	Section 4.4
PGO	Polyak's Gradient-Free Oracle.	Eq. (12); [25], [26]
RF	DiscoGrad version of REINFORCE.	Eq. (13); [13]

Table 2: *Overview of evaluated gradient estimators. All estimators are combined with the Adam optimizer.*

with the number of samples or paths. Each measurement was repeated one hundred times, resulting in 95% confidence intervals smaller than 7% of any of the shown averages.

Figure 5 shows the slowdown of different configurations of the estimators over a single crisp program execution without AD, normalized to the number of paths or samples and rounded to two significant digits. Each value can be interpreted as the slowdown per path or sample.

The overhead for the IPA estimator comprises only the cost of AD and the negligible cost of random number generation for the perturbations. In the EPIDEMICS, HOTEL, and TRAFFIC problem, we see the benefit of the simple sparsity optimization in our AD implementation. In these problems, similar to the extreme case of the Heaviside function shown in Fig. 1, the program's smoothed gradient depends mostly on the branches taken, with only limited arithmetic on variables that directly depend on the input parameters. As a consequence, most variables do not carry a tangent value and the slowdown factor remains far below the input dimension, which would be the expected slowdown with naive forward-mode AD. When increasing the number of samples, the reuse of tangent vectors (cf. Section 4.5.1) allows the overhead per sample to gradually diminish. In contrast, in the AC problem, the output has a non-zero pathwise gradient with respect to the neural network coefficients. Hence, most intermediate variables carry tangents with respect to all inputs and the slowdown becomes more pronounced. Still, due to vectorization of the AD operations, the slowdown for IPA with 1 000 samples is only about a quarter of the input dimension of 82.

Our Monte Carlo estimator DGO involves a configurable number of AD-enabled program executions, additional bookkeeping at branches, and kernel density estimations. We see that accordingly, the slowdown is somewhat larger than IPA's. However, we observe sublinear scaling behavior when increasing the number of samples. Since branch condition values are stored and operated on per branch, the impact of the base costs for the per-branch operations diminishes with larger numbers of samples. As with IPA, the AC problem, which involves more input-dependent arithmetic operations, entails higher overhead.

SI incurs the cost for AD and carrying along Gaussian distributions across several control flow paths, as well as for restricting the state according to the chosen strategy. Depending on the program and state restriction strategy, the number of control flow paths can fluctuate and may not always saturate the configured upper bound. Hence, we normalize to the *ef-*

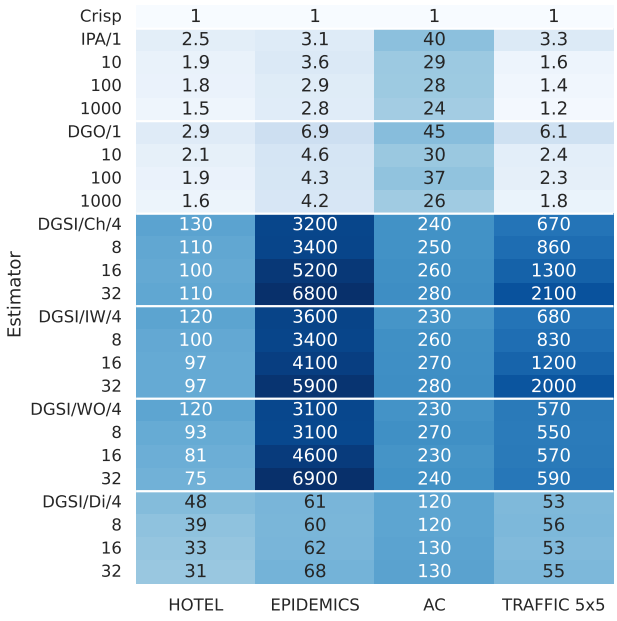


Figure 5: *Slowdown of the different gradient estimators compared to a crisp execution without AD, normalized to reflect the slowdown per sample (Monte Carlo estimators) or path (DGSI).*

fective number of control flow paths throughout the program's execution, which we define as the average number of paths active when encountering a branch. Since the state restriction is applied before branches and dominates SI's execution time, normalizing to the number of paths at that point captures the main cost of the different SI estimators. As expected, the slowdown of SI is much larger compared to the other estimators. The restriction strategies based on merging paths (Ch, IW, WO) are up to two orders of magnitude more expensive than "discard" (Di). The cost of selecting the next paths to be merged is negligible for WO, while it is quadratic in the number of paths for Ch and IW. For all three of Ch, IW, and WO, the cost for the subsequent merging of paths is linear in the number of variables. The results for the EPIDEMICS problem, which uses the largest number of variables of the considered problems, show the resulting enormous overhead of the merging-based strategies, with only modestly better scaling using WO. The DGSI estimators based on merging incurred the lowest overhead in the HOTEL and AC problems, which is a consequence of their comparatively smaller number of variables and branches.

Overall, substantial slowdown is observed with all of the smoothing estimators, ranging from a factor of about two to several orders of magnitude compared to a single crisp execution. In particular, the overhead of DGSI with the restriction strategies based on merging paths is likely to put many real-world applications out of reach. Whether each estimator's overhead can be justified depends on the fidelity of the calculated gradients and the resulting progress in parameter synthesis problems, which we evaluate in the next sections.

5.4 Gradient Fidelity

In this section, we provide empirical measurements of the estimated smoothed gradient fidelity in terms of the mean absolute error, which is defined in the dimension k for the input vector $\mathbf{x}$ as:

$$MAE(g,k) = \frac{1}{S}\sum_{s=1}^{S}\left|\widetilde{\nabla}_k^g\,\mathcal{P}(\mathbf{x}_s) - \widetilde{\nabla}_k\,\mathcal{P}(\mathbf{x}_s)\right|, \qquad (14)$$

where $\mathbf{x}_1,\ldots,\mathbf{x}_S$ is a sequence of sample points and $\widetilde{\nabla}_k^g$ indicates the k-th component of the smoothed gradient estimate of the estimator $g \in \{\text{DGSI, DGO, PGO, RF}\}$, i.e., the partial derivative $\partial\mathcal{P}(\mathbf{x}_s)/\partial x_{k,s}$. To retrieve the MAE in dimension k, we uniformly sample from these partial derivatives in a problem-specific range in dimension k around an optimal value for $\mathbf{x}$, as determined by optimization.

Evaluating this error is challenging, as for large problems it is expensive to calculate the exact smoothed gradient baseline $\widetilde{\nabla}\mathcal{P}(\mathbf{x}_s)$. Thus, we use a large number of samples (5×10^5) of the unbiased PGO to produce a baseline with maximum 95% confidence intervals of 0.003 (AC), 0.014 (TRAFFIC 2x2), 0.033 (TRAFFIC 5x5), 0.425 (HOTEL) and 7.3 (EPIDEMICS). The wide maximum confidence intervals for the HOTEL and EPIDEMICS baselines can be attributed to some large partial derivatives in these problems (cf. Fig. 8). To reduce the computation time, we evaluate all problems in a deterministic setting by configuring a fixed seed and only consider the first ≤ 25 partial derivatives.

Fig. 6 shows an overview of the MAE wrt. the respective dimensions of the AC, TRAFFIC, HOTEL and EPIDEMICS problems. The MAE, as indicated by the cell color intensity, is the average of the MAE defined in Eq. (14) over the first $k=1,\ldots,25$ dimensions of each problem. The first result is that, with some exceptions, the error of all sampling-based estimators decreases with the number of samples. However, the estimators differ in how much samples they need to obtain the same fidelity, with our application of the REINFORCE estimator requiring orders of magnitude more samples than PGO and DGO. Overall, our DGO estimator slightly outperforms the PGO estimation in terms of sample efficiency, although in some scenarios a bias prohibits further improvement with the number of samples. The DGSI estimation is also very close to the baseline in many cases. Especially on smaller problems such as TRAFFIC 2x2, DGSI exhibits a competitive error, delivering good results even with only 8 tracked paths. As expected, the Di restriction strategy is less accurate than the more expensive Ch. Additionally, the error varies drastically between problems for all estimators.

A more granular view of these findings is depicted in Figures 7, 8 and 9, which show a comparison of selected partial derivatives as line plots. The vertical bar in each plot represents the optimal value, around which the samples of the partial derivative were taken. In Fig. 7 it can be seen that for relatively low-dimensional problems, DGO and DGSI deliver almost perfect results, and the optimum could be identified at the parameter values where the partial derivatives cross 0. From Fig. 8 it is evident that some problem dimensions pose greater challenges to smoothed derivative estimation than others. For example, the estimates wrt. the location-specific

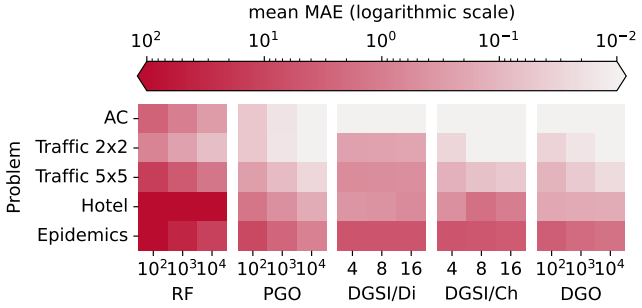


Figure 6: *Error in the gradient estimate over different problems and estimator parametrizations (lower is better). Each cell reflects the mean absolute error (MAE) wrt. to the baseline (PGO/500 000), averaged over the first ≤ 25 model dimensions. The lowest error is provided by PGO and DGO, with the best result for REINFORCE magnitudes higher. For DGSI, a consistent decrease of the error with the number of paths can generally not be observed. In some cases, this is also true for DGO, signaling a potential bias in the estimate.*

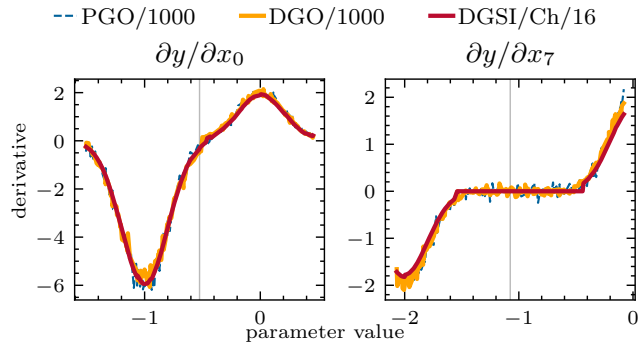


Figure 7: *Partial derivatives wrt. the signal offsets of the TRAFFIC 5x5 problem, as calculated by different gradient estimates. Here and in the following figures, the centered vertical bar indicates the optimal value around which the samples were taken. In this fairly easy problem, all estimators deliver accurate partial derivatives.*

infection probabilities x_4 and x_6 are much noisier than wrt. the relative recovery time x_1; the estimates wrt. some of the dimensions seem to be biased for DGO and DGSI. Additionally, DGSI delivers a gradient that is significantly smaller than the baseline and sometimes (erroneously) zero. This can be attributed to the fixed size of the state tracked by SI, which necessarily results in a loss of smoothed derivative information for sufficiently large problems. Interestingly, in the AC problem, the gradients delivered by DGSI are also much smaller than the baseline, but can still capture the trend very well (cf. Fig. 9). On this problem, DGO is vastly less noisy than PGO, which can be attributed to the use of the exact pathwise derivative.

To conclude, we observe that with some exceptions where the gradient estimates are biased, DGO delivers accurate results. Where it can exploit the pathwise derivative, the results exhibit less variance than the baseline PGO estimate. DGSI is competitive in terms of the MAE on smaller problems with low dimensionality, beating every other estimate on TRAFFIC 2x2, and obtains less noisy derivatives than the sampling-based estimators in these cases. On larger problems, it incurs a bias, but can often still capture the underlying trend. The effects of the observed differences in fidelity on the estimators' utility for gradient descent is evaluated in the next section.

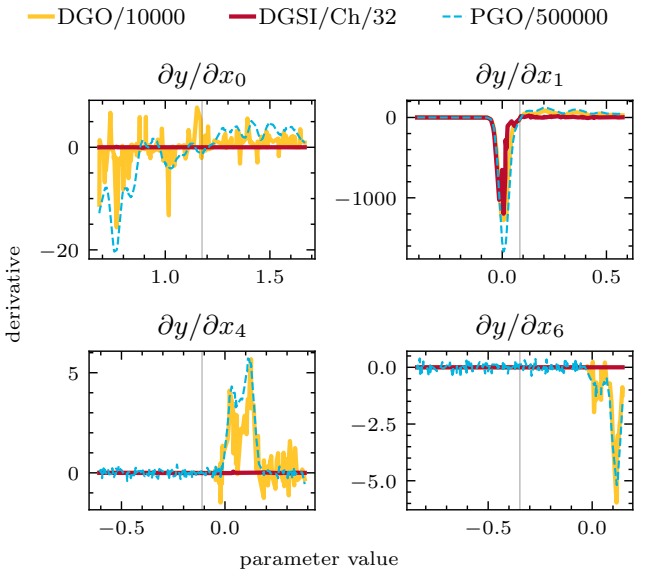


Figure 8: *Partial derivatives wrt. the recovery rate (x_0), initial infection probability (x_1) and location-specific infection probabilities x_4 and x_6 of the EPIDEMICS model, as calculated by different gradient oracles. Using the PGO estimator with 500 000 samples as an unbiased baseline, the fidelity varies drastically among estimators, but also the problem dimensions. In particular, DGSI produces derivatives that are much too small.*

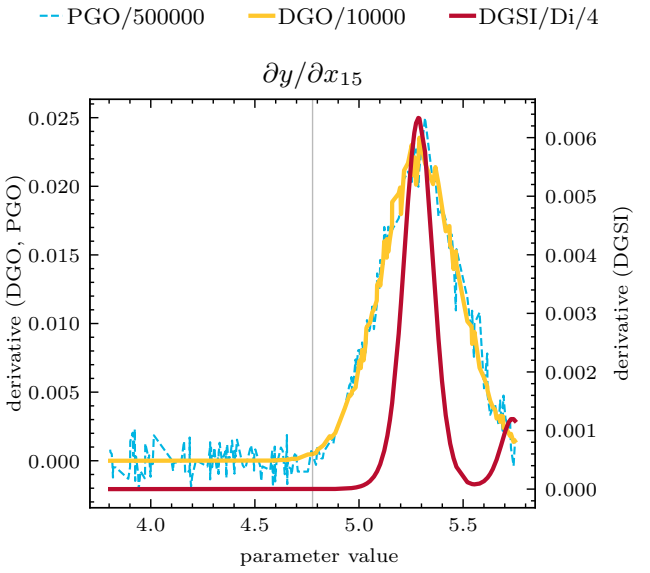


Figure 9: *Partial derivatives wrt. neural network parameters of the AC problem, as calculated by different gradient oracles. In this problem, the DGO and DGSI estimators can profit from their accurate pathwise gradient, while PGO exhibits a lot of noise. The DGSI estimator is severely biased and thus plotted on a separate scale (right), but is able to capture the trends correctly.*

5.5 Optimization Performance

The optimization progress using the gradient-based approaches hinges on a suitable choice of the input standard deviation and the learning rate of the Adam optimizer. Varying these two hyperparameters affects the degree to which the computed gra-

dients accord with the original function on one hand, the ability to escape local minima on the other hand. Here, we identified for each problem one combination (σ_0, η_0) of standard deviation and learning rate where good progress was made with all estimators. An automated hyperparameter sweep was then carried out covering three levels for each hyperparameter, covering all combinations of $\sigma_0 \cdot (\frac{1}{2}, 1, 2)$ and $\eta_0 \cdot (\frac{1}{2}, 1, 2)$. As additional hyperparameters, we further varied the number of samples used by the stochastic gradient estimators, and for DGSI the number of paths and the path restriction strategy. Across all problems and optimization methods, we carried out 3 310 macroreplications, resulting in a total CPU time of about 3 085 hours.

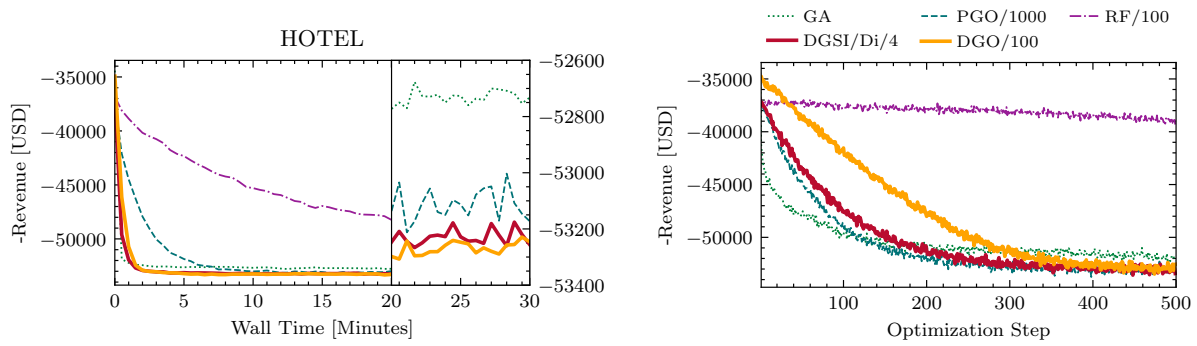
Where not stated otherwise, we show for each estimator the results of the hyperparameter combination that yielded the best solution at the end of the time budget. At each solution determined via a smoothing estimator, we carried out an additional evaluation using the *crisp* program. The plots show the results of the crisp evaluation to ensure comparability of the solution qualities even if the smoothed program deviates from the crisp one. After observing a premature convergence of simulated annealing (SA) to low-quality solutions, we decreased its hyperparameter λ, which determines the relative decrease in temperature per step, from its default value of 10^{-3} to 10^{-5}. Nevertheless, due to a lack of significant progress, the results for the SA are excluded from most plots.

Our plots show the optimization progress over wall-clock time and optimization steps. Each data point in our results is the average of five macroreplications carried out for the respective combination of problem, estimator, and hyperparameters. While the progress over time is the main concern for practical purposes, the progress over steps indicates the strides made when disregarding differences in execution time. For the gradient-based estimators and SA, one step represents an evaluation of the objective function at one solution across the configured number of paths or samples for the smoothing estimators. For the genetic algorithm (GA), one step represents an update from one generation to the next, which involves 50 function evaluations, one for each population-member.

Figure 10 shows the optimization progress for the HOTEL problem. Apart from SA and REINFORCE, all methods converge to a similar revenue of about 53 200 within the time budget of thirty minutes. The fastest convergence is achieved by the GA, albeit to a slightly worse solution than the best-performing methods. Comparing the stochastic estimators, PGO performed best with 1 000 samples, in contrast to 100 samples with DGO. Figure 10b, which shows the first 500 optimization steps, indicates that the DGO's higher variance with only 100 samples leads to less progress per step compared to PGO/1000. DGSI performed best with the "discard" (Di) restriction strategy and with four paths, also converging to roughly the same solution quality as DGO and PGO. Inspecting the solutions, we observed that all three of these methods arrived at similar final parameter combinations.

In the EPIDEMICS problem (cf. Figure 11), PGO/100 and particularly the GA outperform the other methods. We have seen in Section 5.4 that DGO and PGO both struggled to accurately estimate the gradient for this problem, in which there is a complex interplay between the initial infection probability, the recovery rate, and the per-location infection probabilities. Here, GA converges extremely quickly, identifying a solution that is reached by PGO/100 only at the end of the time budget. Considering the progress over the first 50 steps, we see that DGO makes larger strides than all other gradient estimators, indicating that that its slower progress over time is a result of its higher execution time rather than lower-fidelity gradient estimates. With DGSI, the convergence both over time and steps is too slow to be competitive.

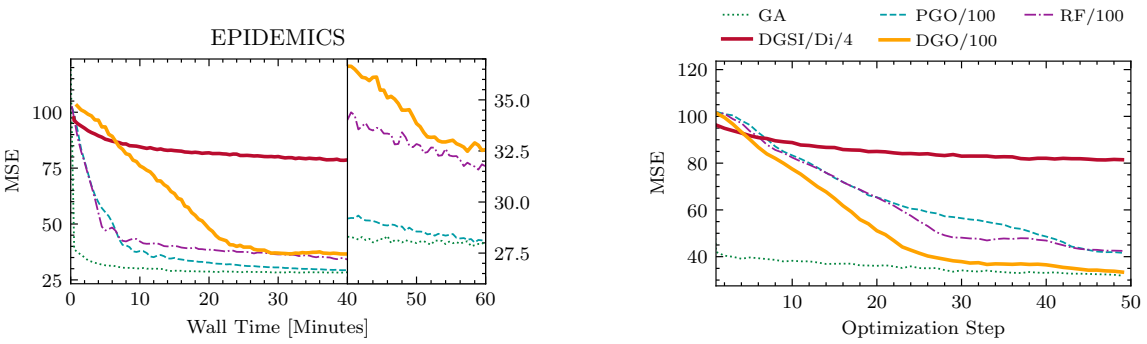
Of our problems, AC is the only one in which some of the partial derivatives are non-zero in the crisp case. Hence, the classical IPA estimator can be applied, albeit without capturing the discontinuities generated by the decision whether cooling is activated in a time step. As Figure 12 shows, all methods apart from SA were able to reduce the cost function to below 2.4, with the best solutions obtained by PGO/100 and DGSI with the IW strategy and eight paths. IPA/1000 made very little initial progress, but approaches the other methods' results at the end of the time budget. Studying the AC controller's behavior, all of the solutions obtained by the listed methods activate the cooling whenever the temperature is higher than the target. However, the best two solutions identified by PGO and DGSI result in a more careful selection of the degree of cooling according to the current insulation and the energy cost. Here, in contrast to the other problems, DGSI benefits from the existence of non-zero pathwise gradients, for which AD delivers exact values.



(a) Progress over time, y-axis zoomed for the last 10 min.

(b) Progress over the first 500 optimization steps.

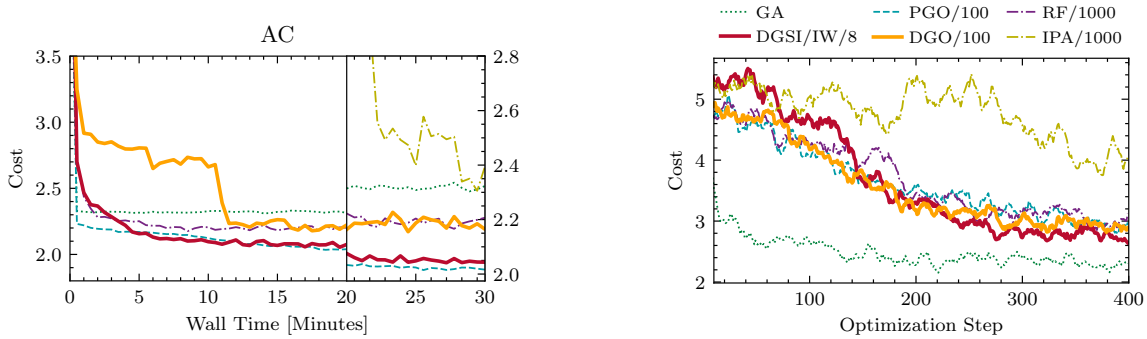
Figure 10: *Optimization progress of the best-performing parametrization of each estimator for the HOTEL problem over optimization steps.*



(a) Progress over time, y-axis zoomed for the last 20 min.

(b) Progress over the first 50 optimization steps.

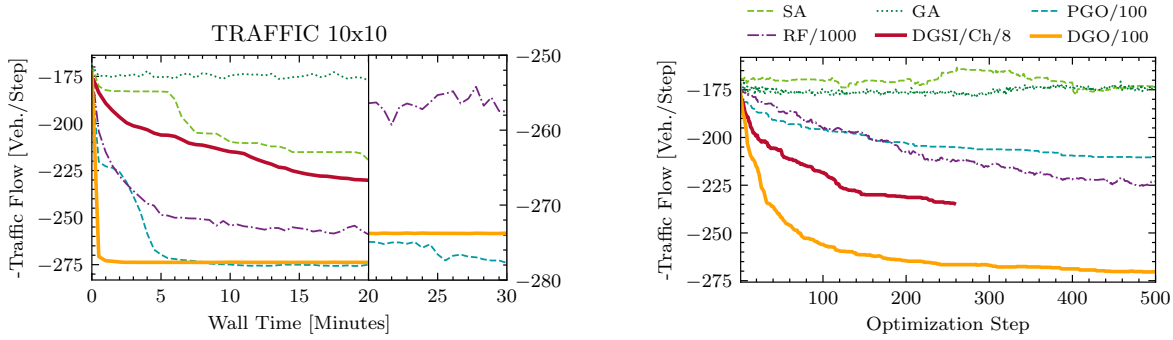
Figure 11: *Optimization progress of the best-performing parametrization of each estimator for the EPIDEMICS problem.*



(a) Progress over time, y-axis zoomed for the last 10 min.

(b) Progress over the first 800 optimization steps.

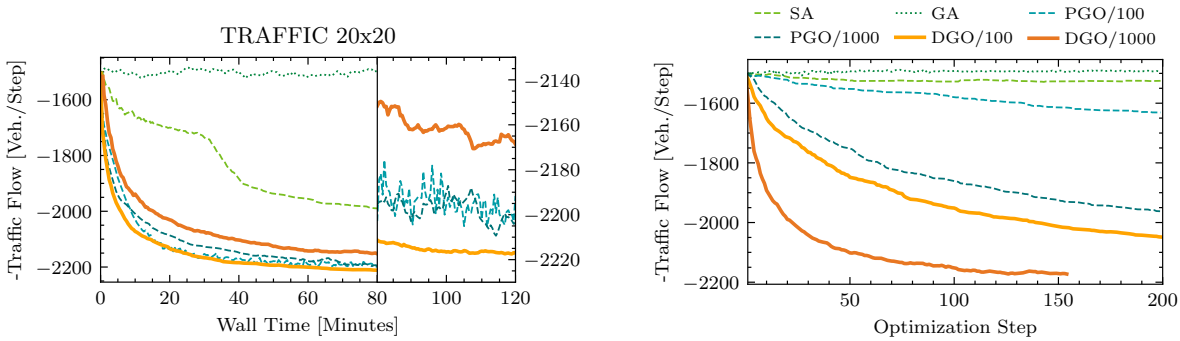
Figure 12: *Optimization progress of the best-performing parametrization of each estimator for the AC problem.*



(a) Progress over time, y-axis zoomed for the last 10 min.

(b) Progress over the first 500 optimization steps.

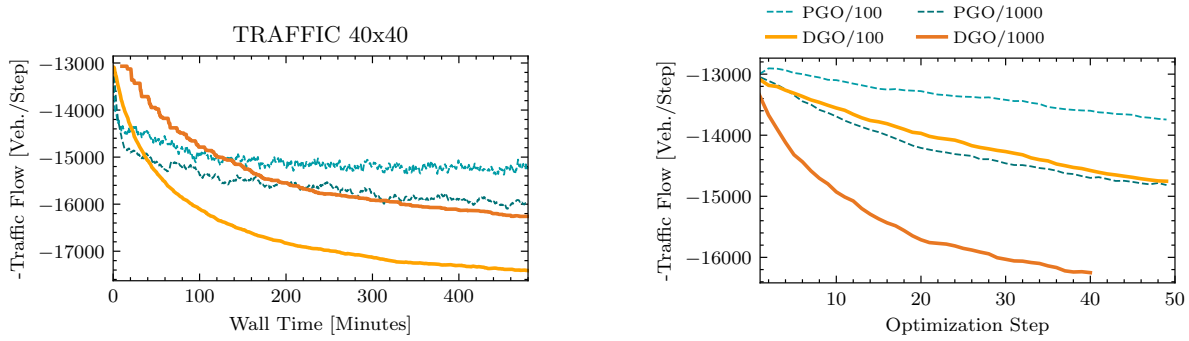
Figure 13: *Optimization progress of the best-performing parametrization of each estimator for the TRAFFIC 10x10 problem.*



(a) Progress over time, y-axis zoomed for the last 40 min.

(b) Progress over the first 200 optimization steps.

Figure 14: *Optimization progress of the best-performing parametrization of each estimator for the TRAFFIC 20x20 problem.*



(a) Progress over time, y-axis zoomed for the last 40 min.

(b) Progress over the first 50 optimization steps.

Figure 15: *Optimization progress of the DGO and DGO estimators for the TRAFFIC 40x40 problem.*

Finally, we consider the TRAFFIC problem at the three scales of 10x10, 20x20, and 40x40 resulting in 100, 400, and 1 600 decision variables. Figure 13 shows the results for the 10x10 grid. In accordance with the fidelity results from Section 5.4, where we have seen that DGO produces highly accurate gradients for this problem, the fastest convergence over the optimization steps by far is achieved by DGO/100, which still holds when plotted over optimization steps. Here, GA was not able to improve beyond the initial solution. In contrast to SA's results in the other problems, it is able to make substantial progress within the time budget, while still not being competitive with the best-performing methods. REINFORCE is once again outperformed by PGO/100. DGSI with the Ch strategy behaved somewhat similarly to SA, but having completed less than 300 steps was unable to obtain a competitive solution.

Similar trends are observed in Figure 14 for our largest problem TRAFFIC 20x20. We omit DGSI and REINFORCE, which did not make substantial progress. Here, DGO/100 provides the fastest convergence and the best solution, slightly better than PGO/100 and PGO/1000. DGO/1000 exhibits vastly faster convergence over steps, but finished only about 150 steps within the time budget.

Since PGO and DGO consistently outperformed the other methods in the TRAFFIC problem, we limited the computationally intensive experiment on the 40x40 grid to these two estimators with their respective best-performing hyperparameter combination from the 20x20 experiment. Figure 15 shows that for this problem with 1 600 decision variables, DGO/100 vastly outperforms both PGO/100 and PGO/1000 over time, and DGO/1000 over optimization steps. Here, DGO benefits from its use of AD to separate the effects of the individual in-

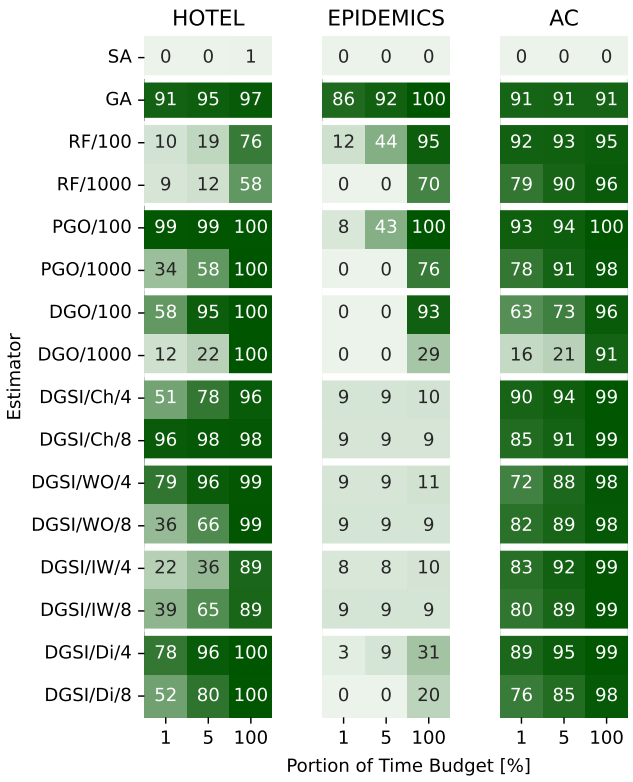


Figure 16: *Overview of the optimization progress over the time budgets for the HOTEL AC and EPIDEMICS problems.*

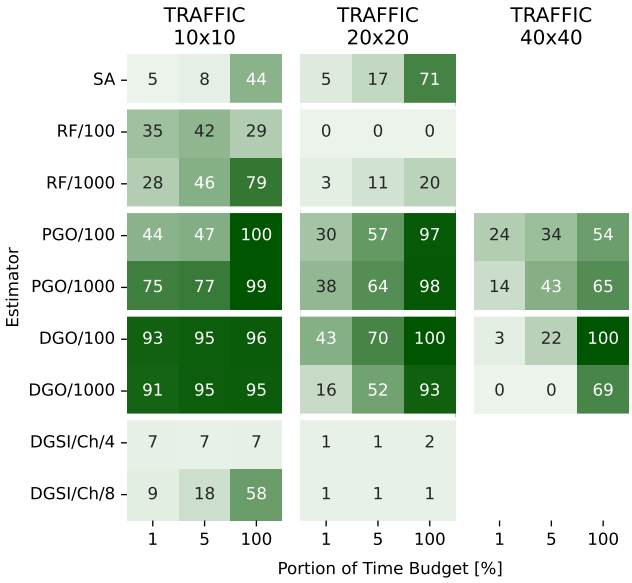


Figure 17: *Overview of the optimization progress over the time budgets for the TRAFFIC problem.*

put dimensions, whereas PGO must rely on the scalar program output alone.

The optimization progress measurements are summarized in the heatmaps shown in Figures 16 and 17, which indicate each method's progress relative to the best improvement over the initial solution made by any method. For the traffic problems, we only show the methods that made significant progress for at least one problem size.

In summary, DGSI has proven to swiftly determine high-quality solutions for the HOTEL and AC problems, in which the effects of choosing alternate branches are less extreme than in the other problems. Particularly good results were seen for AC which is the only of the considered problems in which the pathwise gradient is non-zero. Generally, the stochastic estimators PGO and DGO delivered the most reliably high-quality solutions within the time budget. While our AD-based estimator DGO showed outstanding performance particularly in the TRAFFIC problem with large numbers of input dimensions, the existing estimator PGO has the benefit of being applicable to existing programs without instrumentation.

6 Conclusion

Although an evaluation of gradient estimators targeting a problem domain as broad as parameter synthesis must necessarily be limited in scope, we identify several trends in our results.

The objective functions of the considered optimization problems are discontinuous and non-convex. Nevertheless, local search based on gradient descent consistently outperformed global search via a genetic algorithm or simulated annealing. Our results indicate that both for stochastic and non-stochastic problems, likely due to the combination of noisy estimates with Adam, the local search is able to escape local minima sufficiently to swiftly identify high-quality solutions. Future work dedicated to more extensive benchmarking could consider more sophisticated global optimization methods such as recent trust region-based algorithms [54] or metaheuristics [55].

Each of the studied gradient estimators comes with its own tradeoffs. The estimators that combine smooth interpretation and automatic differentiation (DGSI) incur a substantial cost in execution time, depending on the state restriction strategy and the number of control flow paths carried along. Notably, we saw that even if the fundamental approximations made by smooth interpretation were lifted, the need to combine or discard intermediate state severely impacts the gradient fidelity. Accordingly, DGSI excels at problems with only limited branching or where the effects of branches are relatively constrained, i.e., non-chaotic problems. Future efforts to improve the gradient estimation via smooth interpretation should thus focus on robust state restriction strategies.

Our proposed estimator using automatic differentiation and Monte Carlo sampling (DGO) is vastly less computationally expensive and avoids the various approximations made in smooth interpretation. A key limitation is the need to obtain a sufficient number of samples at each branch, which may require many replications in the presence of deeply nested branches. In the considered problems, where nested branching could be reformulated into sequential branching, the fidelity of DGO was always among the best of the considered estimators. DGO's combination with Adam provided competitive convergence behavior for all problems, vastly outperforming its closest competitor in our highest-dimensional problem. Our tool DiscoGrad offers an efficient implementation of DGSI and DGO to automate the estimations via DGSI and DGO for programs written in C++.

Considering the existing estimators, a remarkable observation is that Polyak's Gradient-Free Oracle (PGO), which does not require AD, exhibited low execution times and provided good results in all experiments. We thus consider PGO and the closely related gradient-free algorithm Nesterov Random Search [25, 56] promising alternatives to global search across high-dimensional parameter spaces, even for non-convex problems.

In our experiments, we carried out a hyperparameter sweep to identify suitable combinations of learning rates, degrees of smoothing, and numbers of samples. Since these hyperparameters interact, our future work will include an exploration of scheduling algorithms that jointly select combinations of these hyperparameters and adjust them throughout the optimization process.

Acknowledgments

We extend our thanks to Marian Zuska for implementing the HOTEL optimization problem in DiscoGrad. Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation), grant no. 497901036.

References

[1] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.

[2] Andreas Griewank, David Juedes, and Jean Utke. Algorithm 755: ADOL-C: a package for the automatic differentiation of algorithms written in C/C++. *ACM Transactions on Mathematical Software (TOMS)*, 22(2):131–167, 1996.

[3] Charles C Margossian. A review of automatic differentiation and its efficient implementation. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 9(4):e1305, 2019.

[4] Yu-Chi Ho and Christos Cassandras. A new approach to the analysis of discrete event dynamic systems. *Automatica*, 19(2):149–167, 1983.

[5] Jennifer J Sun, Megan Tjandrasuwita, Atharva Sehgal, Armando Solar-Lezama, Swarat Chaudhuri, Yisong Yue, and Omar Costilla-Reyes. Neurosymbolic programming for science. *arXiv preprint arXiv:2210.05050*, 2022.

[6] Alexander L Gaunt, Marc Brockschmidt, Rishabh Singh, Nate Kushman, Pushmeet Kohli, Jonathan Taylor, and Daniel Tarlow. Terpret: A probabilistic programming language for program induction. *arXiv preprint arXiv:1608.04428*, 2016.

[7] Philipp Andelfinger. Towards differentiable agent-based simulation. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 2022.

[8] Hiroharu Kato, Deniz Beker, Mihai Morariu, Takahiro Ando, Toru Matsuoka, Wadim Kehl, and Adrien Gaidon. Differentiable rendering: A survey. *arXiv preprint arXiv:2006.12057*, 2020.

[9] Hyung Ju Terry Suh, Tao Pang, and Russ Tedrake. Bundled gradients through contact via randomized smoothing. *IEEE Robotics and Automation Letters*, 7(2):4000–4007, 2022.

[10] Gaurav Arya, Moritz Schauer, Frank Schäfer, and Chris Rackauckas. Automatic differentiation of programs with discrete randomness. *arXiv preprint arXiv:2210.08572*, 2022.

[11] Anna Choromanska, Mikael Henaff, Michael Mathieu, Gérard Ben Arous, and Yann LeCun. The loss surfaces of multilayer networks. In *Artificial Intelligence and Statistics*, pages 192–204. PMLR, 2015.

[12] Swarat Chaudhuri and Armando Solar-Lezama. Smooth interpretation. *ACM Sigplan Notices*, 45(6):279–291, 2010.

[13] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8:229–256, 1992.

[14] Wei-Bo Gong and Yu-Chi Ho. Smoothed (conditional) perturbation analysis of discrete event dynamical systems. *IEEE Transactions on Automatic Control*, 32(10):858–866, 1987.

[15] J. A. Nelder and R. Mead. A simplex method for function minimization. *The Computer Journal*, 7(4):308–313, 1965.

[16] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[17] Michael JA Smith. Probabilistic abstract interpretation of imperative programs using truncated normal distributions. *Electronic Notes in Theoretical Computer Science*, 220(3):43–59, 2008.

[18] David Monniaux. Abstract interpretation of probabilistic semantics. In *International Static Analysis Symposium*, pages 322–339. Springer, 2000.

[19] Manlio Gaudioso, Giovanni Giallombardo, and Giovanna Miglionico. Essentials of numerical nonsmooth optimization. *Annals of Operations Research*, 314(1):213–253, 2022.

[20] Felix Petersen. Learning with differentiable algorithms. *arXiv preprint arXiv:2209.00616*, 2022.

[21] Marko Mäkelä. Survey of bundle methods for nonsmooth optimization. *Optimization Methods and Software*, 17(1):1–29, 2002.

[22] James V Burke, Frank E Curtis, Adrian S Lewis, Michael L Overton, and Lucas EA Simões. Gradient sampling methods for nonsmooth optimization. *Numerical Nonsmooth Optimization: State of the Art Algorithms*, pages 201–225, 2020.

[23] Michael C. Fu. Chapter 19: Gradient Estimation. In Shane G. Henderson and Barry L. Nelson, editors, *Simulation*, volume 13 of *Handbooks in Operations Research and Management Science*, pages 575–616. Elsevier, 2006.

[24] Pierre L'Ecuyer. An overview of derivative estimation. In *Proceedings of the 23rd Conference on Winter Simulation*, WSC '91, page 207–217, USA, 1991. IEEE Computer Society.

[25] Yurii Nesterov and Vladimir Spokoiny. Random gradient-free minimization of convex functions. *Foundations of Computational Mathematics*, 17:527–566, 2017.

[26] B.T. Polyak. *Introduction to Optimization*. Optimization Software, New York, 1987.

[27] Atılım Güneş Baydin, Barak A Pearlmutter, Don Syme, Frank Wood, and Philip Torr. Gradients without backpropagation. *arXiv preprint arXiv:2202.08587*, 2022.

[28] Sebastian Christodoulou and Uwe Naumann. Differentiable programming: Efficient smoothing of control-flow-induced discontinuities. *arXiv preprint arXiv:2305.06692*, 2023.

[29] Martín Abadi and Gordon D Plotkin. A simple differentiable programming language. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–28, 2019.

[30] Benjamin Sherman, Jesse Michel, and Michael Carbin. λ_s: computable semantics for differentiable programming with higher-order functions and datatypes. *Proceedings of the ACM on Programming Languages*, 5(POPL):1–31, 2021.

[31] Alexander K Lew, Mathieu Huot, Sam Staton, and Vikash K Mansinghka. Adev: Sound automatic differentiation of expected values of probabilistic programs. *Proceedings of the ACM on Programming Languages*, 7(POPL):121–153, 2023.

[32] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[33] Pedro H Azevedo de Amorim and Christopher Lam. Distribution theoretic semantics for non-smooth differentiable programming. *arXiv preprint arXiv:2207.05946*, 2022.

[34] John K Feser, Marc Brockschmidt, Alexander L Gaunt, and Daniel Tarlow. Differentiable functional program interpreters. *arXiv preprint arXiv:1611.01988*, 2016.

[35] Matko Bošnjak, Tim Rocktäschel, Jason Naradowsky, and Sebastian Riedel. Programming with a differentiable forth interpreter. In *International Conference on Machine Learning*, pages 547–556. PMLR, 2017.

[36] Alexander L Gaunt, Marc Brockschmidt, Nate Kushman, and Daniel Tarlow. Differentiable programs with neural libraries. In *International Conference on Machine Learning*, pages 1213–1222, 2017.

[37] Swarat Chaudhuri, Kevin Ellis, Oleksandr Polozov, Rishabh Singh, Armando Solar-Lezama, Yisong Yue, et al. Neurosymbolic programming. *Foundations and Trends in Programming Languages*, 7(3):158–243, 2021.

[38] Chenxi Yang and Swarat Chaudhuri. Safe neurosymbolic learning with differentiable symbolic execution. *arXiv preprint arXiv:2203.07671*, 2022.

[39] Ameesh Shah, Eric Zhan, Jennifer J. Sun, Abhinav Verma, Yisong Yue, and Swarat Chaudhuri. Learning differentiable programs with admissible neural heuristics. *CoRR*, abs/2007.12101, 2020.

[40] Felix Petersen, Christian Borgelt, and Oliver Deussen. Algonet: Cinf smooth algorithmic neural networks. *CoRR abs/1905.06886*, 2019.

[41] Yuting Yang and Connelly Barnes. Approximate program smoothing using mean-variance statistics, with application to procedural shader bandlimiting. In *Computer Graphics Forum*, volume 37, pages 443–454. Wiley Online Library, 2018.

[42] Russell R Barton. Tutorial: metamodeling for simulation. In *2020 Winter Simulation Conference (WSC)*, pages 1102–1116. IEEE, 2020.

[43] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. Neuzz: Efficient fuzzing with neural program smoothing. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 803–817. IEEE, 2019.

[44] Tzu-Mao Li, Miika Aittala, Frédo Durand, and Jaakko Lehtinen. Differentiable monte carlo ray tracing through edge sampling. *ACM Transactions on Graphics (TOG)*, 37(6):1–11, 2018.

[45] Guillaume Loubet, Nicolas Holzschuch, and Wenzel Jakob. Reparameterizing discontinuous integrands for differentiable rendering. *ACM Transactions on Graphics (TOG)*, 38(6):1–14, 2019.

[46] Tizian Zeltner, Sébastien Speierer, Iliyan Georgiev, and Wenzel Jakob. Monte carlo estimators for differential light transport. *ACM Transactions on Graphics (TOG)*, 40(4):1–16, 2021.

[47] Pierre L'ecuyer. A unified view of the IPA, SF, and LR gradient estimation techniques. *Management Science*, 36(11):1364–1383, 1990.

[48] KK Benke, S Norng, NJ Robinson, LR Benke, and TJ Peterson. Error propagation in computer models: analytic approaches, advantages, disadvantages and constraints. *Stochastic Environmental Research and Risk Assessment*, 32(10):2971–2985, 2018.

[49] Swarat Chaudhuri and Armando Solar-Lezama. EULER: A system for numerical optimization of programs. In *International Conference on Computer Aided Verification*, pages 732–737. Springer, 2012.

[50] Carlos F Daganzo. The cell transmission model: A dynamic representation of highway traffic consistent with the hydrodynamic theory. *Transportation Research Part B: Methodological*, 28(4):269–287, 1994.

[51] D. J. Eckman, S. G. Henderson, S. Shashaani, and R. Pasupathy. SimOpt. `https://github.com/simopt-admin/simopt`, 2021.

[52] David J Eckman, Shane G Henderson, and Sara Shashaani. Simopt: A testbed for simulation-optimization experiments. *INFORMS Journal on Computing*, 35(2):495–508, 2023.

[53] Diederik P Kingma and Jimmy Ba. Adam: a method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[54] Sara Shashaani, Fatemeh S Hashemi, and Raghu Pasupathy. ASTRO-DF: A class of adaptive sampling trust-region algorithms for derivative-free stochastic optimization. *SIAM Journal on Optimization*, 28(4):3145–3176, 2018.

[55] Ilhem Boussaïd, Julien Lepagnot, and Patrick Siarry. A survey on optimization metaheuristics. *Information Sciences*, 237:82–117, 2013.

[56] Jeffrey Larson, Matt Menickelly, and Stefan M Wild. Derivative-free optimization methods. *Acta Numerica*, 28:287–404, 2019.

A REINFORCE for Discrete Programs

The following is a derivation of the REINFORCE estimator for deterministic programs $\mathcal{P}$, which works by perturbing the input vector $\mathbf{x}$ with Gaussian noise. Through the log-derivative trick, REINFORCE is defined as:

$$\nabla_\theta \mathbb{E}_{X\sim f_\theta(y)}[\mathcal{P}(X)] = \mathbb{E}_{X\sim f_\theta(y)}[\mathcal{P}(X)\,\nabla_\theta \log f_\theta(X)]\,, \tag{15}$$

where f is a density with parameter θ. In our case, we perturb the input vector $\mathbf{x}$ to $\mathcal{P}$ with Gaussian noise to obtain a random variable $X\sim\mathcal{N}(\mathbf{x},\Sigma)$ where $diag(\Sigma)=\sigma^2$, so $\theta\equiv\mathbf{x}$ (cf. Eq. (2)). Thus, we only need to derive the following gradient of $f_{\mathbf{x},\sigma}$, the normal density with mean $\mathbf{x}$ and variance σ^2:

$$\begin{aligned}
\nabla_{\mathbf{x}} \ln f_{\mathbf{x},\sigma}(y) &= \nabla_{\mathbf{x}} \ln\left(\frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{y-\mathbf{x}}{\sigma}\right)^2}\right)\\
&= \nabla_{\mathbf{x}} \ln\left(\frac{1}{\sigma\sqrt{2\pi}}\right) + \nabla_{\mathbf{x}} \ln\left(e^{-\frac{1}{2}\left(\frac{y-\mathbf{x}}{\sigma}\right)^2}\right)\\
&= \nabla_{\mathbf{x}} \ln e^{-\frac{1}{2}\left(\frac{y-\mathbf{x}}{\sigma}\right)^2}\\
&= -\nabla_{\mathbf{x}} \frac{1}{2}\left(\frac{y-\mathbf{x}}{\sigma}\right)^2\\
&= -\nabla_{\mathbf{x}} \frac{(y-\mathbf{x})^2}{2\sigma^2}\\
&= -\frac{2(y-\mathbf{x})}{2\sigma^2}\\
&= \frac{y-\mathbf{x}}{\sigma^2}.
\end{aligned} \tag{16}$$

Using this, we can approximate Eq. (15) by Monte Carlo sampling:

$$\mathbb{E}_{X\sim\mathcal{N}(\mathbf{x},\sigma^2)}[\mathcal{P}(X)\,\nabla_{\mathbf{x}} \log f_{\mathbf{x},\sigma}(X)] = \frac{1}{S}\sum_{s=1}^{S} \mathcal{P}(\mathbf{x}_s)\frac{\mathbf{x}_s-\mathbf{x}}{\sigma^2}, \tag{17}$$

where $\mathbf{x}_s,\ s\in\{1,\dots,S\}$ are iid. variates of X. To make the fact that X is an "artificial" random variable obtained by perturbing the input vector $\mathbf{x}$ more explicit, it is convenient to reparametrize and redefine $X \coloneqq \mathbf{x}+\sigma U$ for $U\sim\mathcal{N}(\mathbf{0},\mathbf{1})$. Applying this substitution to the previous equation yields:

$$\begin{aligned}
\nabla_{\mathbf{x}}\mathbb{E}_X[\mathcal{P}(X)] &= \frac{1}{S}\sum_{s=1}^{S} \mathcal{P}(\mathbf{x}+\sigma\mathbf{u}_s)\frac{\mathbf{x}+\sigma\mathbf{u}_s-\mathbf{x}}{\sigma^2}\\
&= \frac{1}{S}\sum_{s=1}^{S} \mathcal{P}(\mathbf{x}+\sigma\mathbf{u}_s)\frac{\mathbf{u}_s}{\sigma},
\end{aligned} \tag{18}$$

where $u_s,\ s\in\{1,\dots,S\}$ are iid. variates of U. This leads to the REINFORCE estimator shown in Eq. (13).

B Implicit Covariance through Automatic Differentiation

In this appendix, we show how AD wrt. the program inputs can be used to implicitly account for the covariance when applying the standard uncertainty propagation formula. This

technique is also applied by the uncertainties-cpp library[6]. For clarity, we assume the common case of a binary function on two intermediate variables a_1 and a_2 that depend on each other via a single input variable x. The derivation can easily be adapted to cases with multiple inputs and n-ary operations. For the former, one needs to include the covariance between the components of the input vector (which in our case is always 0) in Eq. (22) below, for the latter, one needs to consider $h(a_1, \ldots, a_n)$.

Let the two intermediate variables $a_1 = f(x)$ and $a_2 = g(x)$ be linear functions f and g of the input variable x. The variances of a_2 and a_2 are computed from the input variance σ_x according to [48] to first order as

$$\sigma^2_{f(x)} = \left(\frac{df(x)}{dx}\right)^2 \sigma_x^2 \tag{19}$$

and

$$\sigma^2_{g(x)} = \left(\frac{dg(x)}{dx}\right)^2 \sigma_x^2. \tag{20}$$

Because of the linearity of f and g, this is exact. If we apply a function h on a_1 and a_2 and want to calculate the resulting uncertainty in form of the variance, we need to account for their covariance [48]:

$$\begin{aligned}\sigma^2_{h(a_1,a_2)} \approx &\left(\frac{\partial h(a_1,a_2)}{\partial a_1}\right)^2 \sigma^2_{a_1} + \left(\frac{\partial h(a_1,a_2)}{\partial a_2}\right)^2 \sigma^2_{a_2} \\ &+ 2\left(\frac{\partial h(a_1,a_2)}{\partial a_1}\right)\left(\frac{\partial h(a_1,a_2)}{\partial a_2}\right)\mathrm{Cov}(a_1,a_2).\end{aligned} \tag{21}$$

However, instead of tracking the covariance of each variable explicitly, the following equation can be used, which makes use of the derivative wrt. the input variable x:

$$\sigma^2_{h(a_1,a_2)} \approx \left(\frac{dh(a_1,a_2)}{dx}\right)^2 \sigma_x^2. \tag{22}$$

Under the usual assumption of linearity in h, this expression is equivalent to Eq. (21). One can see this in our binary case by applying the chain rule, as used in AD, and expanding using the binomial theorem:

$$\begin{aligned}\sigma^2_{h(a_1,a_2)} &\approx \left(\frac{\partial h(a_1,a_2)}{\partial a_1}\frac{da_1}{dx} + \frac{\partial h(a_1,a_2)}{\partial a_2}\frac{da_2}{dx}\right)^2 \sigma_x^2 \\ &= \left(\frac{\partial h(a_1,a_2)}{\partial a_1}\frac{da_1}{dx}\right)^2 \sigma_x^2 \\ &+ \left(\frac{\partial h(a_1,a_2)}{\partial a_2}\frac{da_2}{dx}\right)^2 \sigma_x^2 \\ &+ 2\left(\frac{\partial h(a_1,a_2)}{\partial a_1}\frac{da_1}{dx}\right)\left(\frac{\partial h(a_1,a_2)}{\partial a_2}\frac{da_2}{dx}\right)\sigma_x^2.\end{aligned}$$

We observe that, given the definition of $\sigma^2_{f(x)}$ in Eq. (19), we can substitute $(da_1/dx)^2\sigma_x^2$ with $\sigma^2_{a_1}$ (analogously for a_2):

$$\begin{aligned}&= \left(\frac{\partial h(a_1,a_2)}{\partial a_1}\right)^2 \sigma^2_{a_1} + \left(\frac{\partial h(a_1,a_2)}{\partial a_2}\right)^2 \sigma^2_{a_2} \\ &+2\left(\frac{\partial h(a_1,a_2)}{\partial a_1}\frac{\partial a_1}{\partial x}\right)\left(\frac{\partial h(a_1,a_2)}{\partial a_2}\frac{\partial a_2}{\partial x}\right)\sigma_x^2.\end{aligned}$$

The last term of the sum can be brought into the form of Equation 21 by considering a first order approximation of the covariance[7] $\mathrm{Cov}(a_1, a_2)$ as

$$\mathrm{Cov}(f(x), g(x)) \approx \frac{df(x)}{dx}\sigma_x \frac{dg(x)}{dx}\sigma_x = \frac{da_1}{dx}\frac{da_2}{dx}\sigma_x^2. \tag{23}$$

Substitution then leads to Eq. (21). Thus, assuming linear dependencies, AD can be used to avoid the need to explicitly track the covariance term.

C Synthetic Example

Listing 1: *Program used to generate Fig. 3.*

```
const int num_inputs = 1;
#include "include/discograd.hpp"
double thresholds[]{
    0.270522,  0.897051,  -0.217456, -0.123758,
    0.644134,  0.839213,  -0.352062, -0.399846,
    0.998247,  0.933284,  0.910108,  -0.365462,
    -0.079554, 0.743998,  0.076774,  -0.790476,
    -0.715160, -0.835860, 0.642159,  0.311679,
    0.868213,  -0.435599, -0.222482, -0.085613,
    -0.873302, -0.916008, -0.059836, 0.879541,
    -0.168570, -0.054479, -0.674992, -0.253402
};

adouble _DiscoGrad_synthetic_test(
    DiscoGrad<num_inputs> &_discograd, aparams &p) {
  sdouble x({p[0], _discograd.get_variance()});
  sdouble y = x / 2.0; // correlate y with x
  sdouble v = x - y;   // here, y is correlated with x,
                       // necessitating DEA
  // this loop 'simulates' repeated branching behavior
  for (int i = 0; i < 32; ++i) {
    if (v < thresholds[i]) {
      // branches depend on previous branches
      v -= thresholds[i];
    }
  }
  return v.expectation();
}

int main(int argc, char **argv) {
  DiscoGrad<num_inputs> dg(argc, argv);
  DiscoGradFunc<num_inputs> func(
      _DiscoGrad_synthetic_test);
  dg.estimate(func);
  return 0;
}
```

[6] https://www.giacomopetrillo.com/software/uncertainties-cpp/doc/html/classuncertainties_1_1_u_real.html

[7] https://www.giacomopetrillo.com/software/uncertainties-cpp/doc/html/classuncertainties_1_1_u_real.html