Smooth Interpretation [*]

Swarat Chaudhuri
Pennsylvania State University
swarat@cse.psu.edu

Armando Solar-Lezama
MIT
asolar@csail.mit.edu

Abstract

We present *smooth interpretation*, a method for systematic approximation of programs by smooth mathematical functions. Programs from many application domains make frequent use of discontinuous control flow constructs, and consequently, encode functions with highly discontinuous and irregular landscapes. Smooth interpretation algorithmically attenuates such irregular features. By doing so, the method facilitates the use of numerical optimization techniques in the analysis and optimization of programs.

Smooth interpretation extends to programs the notion of *Gaussian smoothing*, a popular signal-processing technique that filters out noise and discontinuities from a signal by taking its convolution with a Gaussian function. In our setting, Gaussian smoothing executes a program P according to a probabilistic semantics. Specifically, the execution of P on an input $\mathbf{x}$ after smoothing is as follows: (1) Apply a Gaussian perturbation to $\mathbf{x}$—the perturbed input is a random variable following a normal distribution with mean $\mathbf{x}$. (2) Compute and return the *expected output* of P on this perturbed input. Computing the expectation explicitly would require the execution of P on all possible inputs, but smooth interpretation bypasses this requirement by using a form of symbolic execution to approximate the effect of Gaussian smoothing on P.

We apply smooth interpretation to the problem of *synthesizing optimal control parameters* in embedded control applications. The problem is a classic optimization problem: the goal here is to find parameter values that minimize the error between the resulting program and a programmer-provided behavioral specification. However, solving this problem by directly applying numerical optimization techniques is often impractical due to discontinuities and "plateaus" in the error function. By "smoothing out" these irregular features, smooth interpretation makes it possible to search the parameter space efficiently by a local optimization method. Our experiments demonstrate the value of this strategy in synthesizing parameters for several challenging programs, including models of an automated gear shift and a PID controller.

Categories and Subject Descriptors D.2.2 [*Software Engineering*]: Design Tools and Techniques; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages; G.1.6 [*Numerical Analysis*]: Optimization; G.1.0 [*Numerical Analysis*]: Approximation.

General Terms Theory, Design, Verification

Keywords Program Smoothing, Continuity, Parameter Synthesis, Abstract Interpretation, Operational Semantics

[*] The research in this paper was supported by the MIT CSAI Lab and the National Science Foundation (CAREER Award #0953507).

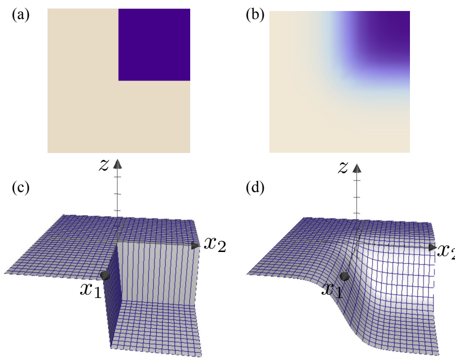


Figure 1. (a) A crisp image. (b) Gaussian smoothing. (c) $Error$: "$\mathtt{z} := 0;$ $\mathtt{if}$ $(\mathtt{x}_1 > 0 \wedge \mathtt{x}_2 > 0)$ $\mathtt{then}\,\mathtt{z} := \mathtt{z} - 2$". (c) Gaussian smoothing of $Error$.

1. Introduction

It is accepted wisdom in software engineering that the dynamics of software systems are inherently discontinuous, and that this makes them fundamentally different from analog systems whose dynamics are given by smooth mathematical functions. Twenty-five years ago, Parnas [17] attributed the difficulty of engineering reliable software to the fact that "the mathematical functions that describe the behavior of [software] systems are not continuous." His argument for using logical methods in software analysis was grounded in the fact that logic, unlike classical analysis, can handle discontinuous systems.

In the present paper, we show that while typical software systems may indeed represent highly nonsmooth functions, the *approximation* of program semantics by smooth functions is plausible and of potential practical value. In particular, we show that such approximations facilitate the use of local numerical optimization techniques in the analysis of control programs manipulating floating-point data. Numerical optimization is usually infeasible for real-world programs because the search spaces derived from them are full of "plateaus" and discontinuities, two bêtes noires of local search algorithms. However, we show that both of these problems can be overcome if numerical search is made to operate on *smooth approximations* of programs.

Gaussian smoothing Smooth approximations of programs can be defined in many ways. Our definition is inspired by the literature on computer vision and signal processing, where numerical optimization is routinely used on noisy real-world signals, and often runs into problems as well. In signal processing, a standard solution to these problems is to preprocess a signal using *Gaussian smoothing* [19], an elementary technique for filtering out noise and discontinuities from a signal by taking its convolution with a Gaussian function. The result of Gaussian smoothing is a smooth, comparatively well-behaved signal—for example, applying Gaussian smoothing to the image in Figure 1-(a) results in an image as in Figure 1-(b). Local optimization is now applied more profitably to this smoothed signal.

We show that a similar strategy can enable the use of numerical search techniques in the analysis of programs. Our first contribution is to introduce a notion of Gaussian smoothing for programs. In our setting, a Gaussian smoothing transform is an interpreter that takes a program P whose inputs range over $\mathbb{R}^K$, and executes it on an input $\mathbf{x}$ according to the following nontraditional semantics:

1. Perturb $\mathbf{x}$ using a Gaussian probability distribution with mean $\mathbf{0}$. The perturbed input is a random variable following a normal distribution with mean $\mathbf{x}$.
2. Execute the program P on this perturbed input. The output is also a random variable—compute and return its expectation.

If $[\![P]\!]$ is the denotational semantics of P, the semantics used by Gaussian smoothing is captured by the following function $\overline{[\![P]\!]}$:

$$\overline{[\![P]\!]}(\mathbf{x}) = \int_{\mathbf{y}\in\mathbb{R}^K} [\![P]\!](\mathbf{y})\ f_{\mathbf{x},\sigma}(\mathbf{y})\ d\mathbf{y}. \tag{1}$$

where $f_{\mathbf{x},\sigma}$ is the density function of the random input obtained by perturbing $\mathbf{x}$.

Note that $\overline{[\![P]\!]}$ is obtained by taking the *convolution* of $[\![P]\!]$ with a Gaussian function. Thus, the above definition is consistent with the standard definition of Gaussian smoothing for signals and images. To see the effect of smoothing defined this way, let P be

$$\mathtt{z} := 0;\ \mathtt{if}\ (\mathtt{x}_1 > 0 \land \mathtt{x}_2 > 0)\ \mathtt{then}\ \mathtt{z} := \mathtt{z} - 2$$

where $\mathtt{z}$ is an output variable and $\mathtt{x}_1$ and $\mathtt{x}_2$ are input variables. The semantics of P is graphed in Figure 1-(c). Smoothing P attenuates its discontinuities, resulting in a program with the semantic map shown in Figure 1-(d).

Smooth interpretation The main challenge in program smoothing is to algorithmically perform Gaussian smoothing on a program—i.e., for a program P and an input $\mathbf{x}$, we want to compute the convolution integral in Equation 1. Unfortunately, solving this integral *exactly* is not possible in general: P is a complex imperative program, and its Gaussian convolution is unlikely to have a closed-form representation. Consequently, we must seek approximate solutions. One possibility is to discretize—or sample—the input space of P and compute the convolution numerically. This approach, while standard in signal processing, just does not work in our setting. The problem is that there is no known way to sample the input space of P while guaranteeing coverage of the executions that significantly influence the result of smoothing (a similar problem afflicts approaches to program testing based on random sampling). This means that approximations of $\overline{[\![P]\!]}(\mathbf{x})$ using discretization will usually be hopelessly inaccurate. Consequently, we compute the integral using approximate symbolic methods.

As our integral involves programs rather than closed-form mathematical functions, the symbolic methods we use to solve it come from abstract interpretation and symbolic execution [5, 11, 14, 18, 20] rather than computer algebra. Our algorithm—called *smooth interpretation*—uses symbolic execution to approximate the probabilistic semantics defined earlier.

The symbolic states used by smooth interpretation are probability distributions (more precisely, each such state is represented as a *Gaussian mixture distribution*; see Section 3). Given an input $\mathbf{x}$, the algorithm first generates a symbolic state representing a normal distribution with mean $\mathbf{x}$. Recall that this is the distribution of the random variable capturing a Gaussian perturbation of $\mathbf{x}$. Now P is symbolically executed on this distribution, and the expected value of the resulting distribution is the result of smooth interpretation.

The approximations in smooth interpretation are required because what starts as a simple Gaussian distribution can turn arbitrarily complex through the execution of the program. For example, conditional branches introduce conditional probabilities; assignments can lead to pathological distributions defined in terms of Dirac delta functions, and join points in the program require us to compute the *mixture* of the distributions from each incoming branch. All of these situations require smooth interpretation to approximate highly complex distributions with simple Gaussian mixture distributions.

Despite these approximations, smooth interpretation effectively smooths away the discontinuities of the original program. By tuning the standard deviation of the input distribution, one can control not just the extent of smoothing, but also the amount of information lost by the approximations performed by smooth interpretation.

Parameter synthesis Finally, we show that smooth interpretation allows easier use of numerical optimization techniques in program analysis. The concrete problem that we consider is *parameter synthesis*, where the goal is to automatically instantiate unknown program parameters such that the resultant program matches a behavioral specification. This problem is especially important for programs controlling cyber-physical systems. The dynamics of such systems often depend crucially on certain numerical parameters referenced in their controllers—e.g., the behavior of a system controlled by a PID controller is affected in complex ways by the values of the latter's control parameters. At the same time, correct values of these parameters are often difficult to determine from high-level insights.

In our formulation, parameter synthesis is a problem of numerical optimization. An instance here consists of a control program P with unknown parameters, a model of the physical system that it controls, and a specification defining the desired trajectories of the cyber-physical system on various initial conditions. We define a function $Error$ that maps each instantiation $\mathbf{x}$ of the control parameters to real value that captures the deviation of the observed and specified trajectories of the system on the test inputs. Our goal is to find $\mathbf{x}$ such that $Error(\mathbf{x})$ is minimized.

In theory, the above optimization problem can be solved by a local, nonlinear search technique like the Nelder-Mead simplex method [15], which starts with an arbitrary value for the parameters, constructs a simplex around the point, and iteratively shifts the simplex based on a local criterion. The appeal of this method is that it is applicable in principle to discontinuous functions. In practice, such search fails in our context as the function $Error$ can be not only discontinuous but *extremely discontinuous*. Consider a program of the form

$$\mathbf{for}(\mathtt{i} := 0; \mathtt{i} < \mathtt{N}; \mathtt{i} := \mathtt{i}+1)\{\mathtt{if}\ (\mathtt{x} < 0)\{\ldots\}\}.$$

Here the branch inside the loop introduces a potential discontinuity; sequential composition allows for the possibility of an exponential number of discontinuous regions. Secondly, like any other local search method, the Nelder-Mead method suffers from "plateaus" and "troughs", i.e., regions in the function's landscape consisting of suboptimal local minima. An example of such a failure scenario is shown in Figure 1-(c). Here, if the local search starts with a point that is slightly outside the quadrant $(\mathtt{x}_1 > 0) \land (\mathtt{x}_2 > 0)$, it will be stuck in a region where the program output is far from the desired global minimum. In more realistic examples, $Error$ will have many such suboptimal regions.

However, this difficulty can often be overcome if the search runs on a smoothed version of $Error$ (Figure 1-(d)). The sharp discontinuities and flat plateaus of Figure 1-(c) are now gone, and any point reasonably close to the quadrant $(\mathtt{x_1} > 0) \wedge (\mathtt{x_2} > 0)$ has a nonzero downhill gradient. Hence, the search method is now able to converge to a value close to the global minimum for a much larger range of initial points.

The above observation is exploited in an algorithm for parameter synthesis that combines Nelder-Mead search with smooth interpretation. The algorithm tries to reconcile the conflicting goal of smoothing away the hindrances to local optimization, while also retaining the core features of the expression to be minimized. The algorithm is empirically evaluated on a controller for a gear shift, a thermostat controller, and a PID controller controlling a wheel. Our method can successfully synthesize parameters for these applications, while the Nelder-Mead method alone cannot.

Summary of contributions and Organization Now we list the main contributions of this paper and the sections where they appear:

- We introduce Gaussian smoothing of programs. (Section 2)
- We present an algorithm—smooth interpretation—that uses symbolic execution to approximately compute the smoothed version of a program. (Section 3)
- We demonstrate, via the concrete application of parameter synthesis, that smooth interpretation facilitates the use of numerical search techniques like Nelder-Mead search in program analysis and synthesis. (Section 4)
- We experimentally demonstrate the value of our method using three control applications. (Section 5)

2. Gaussian smoothing of programs

In this section, we introduce Gaussian smoothing of programs. We start by fixing, for the rest of the paper, a simple language of imperative programs. Programs in our language maintain their state in K real-valued variables named $\mathtt{x_1}$ through $\mathtt{x_k}$. Expressions can be real or boolean-valued. Let E and B respectively be the nonterminals for real and boolean expressions, op stand for real addition or multiplication, and m be a real constant. We have:

$$\begin{array}{lll} E & ::= & \mathtt{x_i} \mid m \mid op(E_1, E_2) \\ B & ::= & E_1 > E_2 \mid B_1 \wedge B_2 \mid \neg B \end{array}$$

Programs P are now given by:

$$\begin{array}{lll} P & ::= & \mathtt{skip} \mid \mathtt{z_i} := E \mid P \,;\, P \mid \mathtt{while}\ B\ \{\ P\ \} \\ & & \mid \mathtt{if}\ B\ \mathtt{then}\ P\ \mathtt{else}\ P. \end{array}$$

2.1 Crisp semantics

Now we give a traditional denotational semantics to a program. To distinguish this semantics from the "smoothed" semantics that we will soon introduce, we refer to it as the *crisp* semantics.

Let us first define a *state* of a program P in our language as a vector $\mathbf{x} \in \mathbb{R}^K$, where $\mathbf{x}(i)$, the i-th component of the vector, corresponds to the value of the variable $\mathtt{x_i}$. The crisp semantics of each real-valued expression E appearing in P is a map $[\![E]\!] : \mathbb{R}^K \to \mathbb{R}$ such that $[\![E]\!](\mathbf{x})$ is the value of E at the state $\mathbf{x}$. The crisp semantics of a boolean expression B is also a map $[\![B]\!] : \mathbb{R}^K \to \mathbb{R}$; we have $[\![B]\!](\mathbf{x}) = 0$ if B is false at the state $\mathbf{x}$, and 1 otherwise. Finally, for $m \in \mathbb{R}$, $\mathbf{x}[i \mapsto m]$ denotes the state $\mathbf{x}'$ that satisfies $\mathbf{x}'(i) = m$, and agrees with $\mathbf{x}$ otherwise.

Using these definitions, we can now express the crisp semantics of P. For simplicity, we assume that *each subprogram of P (including P itself) terminates on all inputs*.

Definition 1 (Crisp semantics)**.** Let P' be an arbitrary subprogram of P. The crisp semantics of P' is a function $[\![P']\!] : \mathbb{R}^K \to \mathbb{R}^K$ defined as follows:

- $[\![\mathtt{skip}]\!](\mathbf{x}) = \mathbf{x}$.
- $[\![\mathtt{x_i} := E]\!](\mathbf{x}) = \mathbf{x}[i \mapsto [\![E]\!](\mathbf{x})]$
- $[\![P_1; P_2]\!](\mathbf{x}) = [\![P_2]\!]([\![P_1]\!](\mathbf{x}))$.
- $[\![\mathtt{if}\ B\ \mathtt{then}\ P_1\ \mathtt{else}\ P_2]\!](\mathbf{x}) =$
  $[\![B]\!](\mathbf{x}) \cdot [\![P_1]\!](\mathbf{x}) + [\![\neg B]\!](\mathbf{x}) \cdot [\![P_2]\!](\mathbf{x})$.
- Let $P' = \mathtt{while}\ B\ \{\ P_1\ \}$. Then we have

$$[\![P']\!](\mathbf{x}) = \mathbf{x} \cdot [\![\neg B]\!](\mathbf{x}) + [\![B]\!](\mathbf{x}) \cdot [\![P']\!]([\![P_1]\!](\mathbf{x})). \quad \square$$

Note that $[\![P']\!](\mathbf{x})$ is well-defined as P' terminates on all inputs.

If $[\![P]\!](\mathbf{x}) = \mathbf{x}'$, then $\mathbf{x}'$ is the *output* of P on the *input* $\mathbf{x}$.

2.2 Smoothed semantics and Gaussian smoothing

Let us now briefly review Gaussian (or normal) distributions. Recall, first, the probability density function for a random variable Y following a 1-D Gaussian distribution:

$$f_{\mu,\sigma}(y) = (1/(\sqrt{2\pi}\sigma))\, e^{-(y-\mu)^2/2\sigma^2}. \tag{2}$$

Here $\mu \in \mathbb{R}$ is the mean and $\sigma > 0$ the *standard deviation*.

A more general setting involves random variables $\mathbf{Y}$ ranging over vectors of reals. This is the setting in which we are interested. We assume that $\mathbf{Y}$ has K components (i.e., $\mathbf{Y}$ ranges over states of P), and that these components are independent variables. In this case, the joint density function of $\mathbf{Y}$ is a K-D Gaussian function

$$f_{\mu,\sigma}(\mathbf{y}) = \prod_{i=1}^{K} f_{\mu(i),\sigma(i)}(\mathbf{y}(i)) \tag{3}$$

Here $\sigma \in \mathbb{R}^K > \mathbf{0}$ is the *standard deviation*, and $\mu \in \mathbb{R}^K$ is the *mean*. The *smoothed semantics* of P can now be defined as a smooth approximation of $[\![P]\!]$ in terms of $f_{\mu,\sigma}$.

Definition 2 (Smoothed semantics)**.** Let $\beta > 0$, $\sigma = (\beta, \dots, \beta)$, and let the function $f_{\mathbf{x},\sigma}$ be defined as in Equation (3). The *smoothed semantics* of P with respect to β is the function $\overline{[\![P]\!]}_\beta : \mathbb{R}^K \to \mathbb{R}^K$ defined as follows:

$$\overline{[\![P]\!]}_\beta(\mathbf{x}) = \int_{\mathbf{y} \in \mathbb{R}^K} [\![P]\!](\mathbf{y})\ f_{\mathbf{x},\sigma}(\mathbf{y})\ d\mathbf{y}. \quad \square$$

The function $\overline{[\![P]\!]}_\beta(\mathbf{x})$ is the *convolution* of the function $[\![P]\!]$ and a K-D Gaussian with mean $\mathbf{0}$ and standard deviation $\sigma = (\beta, \dots, \beta)$. The constant β is said to be the *smoothing parameter*. When β is clear from the context, we often denote $\overline{[\![P]\!]}_\beta$ by $\overline{[\![P]\!]}$.

Smoothed semantics is defined not only at the level of the whole program, but also at the level of individual expressions. For example, the smoothed semantics of a boolean expression B is

$$\overline{[\![B]\!]}_\beta(\mathbf{x}) = \int_{\mathbf{y} \in \mathbb{R}^K} [\![B]\!](\mathbf{y}) f_{\mathbf{x},\sigma}(\mathbf{y})\ d\mathbf{y}$$

where $\sigma = (\beta, \dots, \beta)$.

We use the phrase "*Gaussian smoothing* of P (with respect to β)" to refer to the interpretation of P according to the smoothed semantics $\overline{[\![\circ]\!]}_\beta$. Note that Gaussian smoothing involves computing the convolution of $[\![P]\!]$ and a Gaussian function—thus, our terminology is consistent with the standard definition of Gaussian smoothing in image and signal processing. The following examples shed light on the nature of Gaussian smoothing.

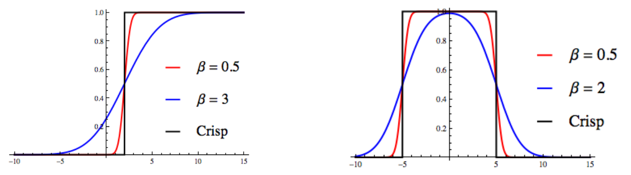


Figure 2. (a) A sigmoid. (b) A bump.

Example 1. Consider the boolean expression $B : (\mathtt{x}_0 - a) > 0$, where $a \in \mathbb{R}$. We have:

$$\overline{[\![B]\!]}_\beta(x) = \int_{-\infty}^{\infty} [\![P]\!](y) f_{x,\beta}(y)\, dy = 0 + \int_a^\infty f_{x,\beta}(y)\, dy$$
$$= \int_0^\infty \frac{1}{\sqrt{2\pi}\beta}\, e^{-(y-x+a)^2/2\beta^2} dy = \frac{1+\mathrm{erf}(\frac{x-a}{\sqrt{2}\beta})}{2}$$

where x ranges over the reals, and erf is the Gauss error function.

Figure 2-(a) plots the crisp semantics $[\![B]\!]$ of B with $a = 2$, as well as $\overline{[\![B]\!]}_\beta$ for smoothing with Gaussians with standard deviations $\beta = 0.5$ and $\beta = 3$. While $[\![B]\!]$ has a discontinuous "step," $\overline{[\![B]\!]}_\beta$ is a smooth S-shaped curve, or a *sigmoid*. As we decrease β, the sigmoid $\overline{[\![B]\!]}_\beta$ becomes steeper and steeper, and at the limit, approaches $[\![B]\!]$.

Along the same lines, consider the boolean expression $B' : a < \mathtt{x}_0 < c$, where $a, c \in \mathbb{R}$. We can show that

$$\overline{[\![B']\!]}_\beta(x) = \frac{\mathrm{erf}(\frac{c-x}{\sqrt{2}\beta}) + \mathrm{erf}(\frac{x-a}{\sqrt{2}\beta})}{2}.$$

The functions $[\![B']\!]$ and $\overline{[\![B']\!]}_\beta$, with $a = -5$, $c = 5$, and $\beta = 0.5$ and 2, are plotted in Figure 2-(b). Once again, discontinuities are smoothed, and as β decreases, $\overline{[\![B']\!]}$ approaches $[\![B]\!]$. □

Example 2. Let us now consider the following program P:

$$\mathtt{if}\ \mathtt{x}_0 > 0\ \mathtt{then\ skip\ else}\ \mathtt{x}_0 := 0.$$

It can be shown that for all $x \in \mathbb{R}$,

$$\overline{[\![P]\!]}_\beta(x) = x\,\frac{1+\mathrm{erf}(\frac{x}{\sqrt{2}\beta})}{2} + \frac{\beta e^{-x^2/2\beta^2}}{2}. \quad \square$$

We note that the smoothed semantics of a program P can be formulated as the composition of the smoothed semantics of P with respect to 1-D Gaussians. This follows directly from our assumption of independence among the variables in the input distribution. Specifically, we have: $f_{\mathbf{x},\sigma} = f_{\mathbf{x}(1),\beta}\ f_{\mathbf{x}(2),\beta} \dots f_{\mathbf{x}(K),\beta}$, where $f_{\mathbf{x}(i),\beta}$ is a 1-D Gaussian with mean $\mathbf{x}(i)$ and standard deviation β.

Let us denote by $\overline{[\![P]\!]}_{i,\beta}$ the smoothed semantics of P with respect to the i-th variable of the input state, while all other variables are held constant. Formally, we have

$$\overline{[\![P]\!]}_{i,\beta}(\mathbf{x}) = \int_{y \in \mathbb{R}} [\![P]\!](\mathbf{x}')\ f_{\mathbf{x}(i),\beta}(\mathbf{x}')\ dy$$

where f is as in Equation 2, and $\mathbf{x}'$ is such that $\mathbf{x}'(i) = y$, and for all $j \neq i$, $\mathbf{x}'(j) = \mathbf{x}(j)$. It is now easy to see that:

Theorem 1. $\overline{[\![P]\!]}_\beta(\mathbf{x}) = (\overline{[\![P]\!]}_{1,\beta} \circ \dots \circ \overline{[\![P]\!]}_{K,\beta})(\mathbf{x})$.

2.3 Smoothed semantics: a structural view

As mentioned earlier, Definition 2 matches the traditional signal-processing definition of Gaussian smoothing. In signal processing, however, signals are just data sets, and convolutions are easy to compute numerically. The semantics of programs, on the other hand, are defined structurally, so a structural definition of smoothing is much more desirable. Our structural definition of Gaussian smoothing is probabilistic, and forms the basis of our smooth interpretation algorithm defined in Section 3.

In order to provide an inductive definition of $\overline{[\![P]\!]}_\beta$, we view the smoothed program as applying a transformation on random variables. Specifically, the smoothed version $\overline{P}$ of P performs the following steps on an input $\mathbf{x} \in \mathbb{R}^K$:

1. Construct the (vector-valued) random variable $\mathbf{Y}_\mathbf{x}$ with density function $f_{\mathbf{x},\sigma}$ from Equation (3). Note that $\mathbf{Y}_\mathbf{x}$ has mean $\mathbf{x}$ and standard deviation $\sigma = (\beta, \dots, \beta)$. Intuitively, the variable $\mathbf{Y}_\mathbf{x}$ captures the result of *perturbing* the state $\mathbf{x}$ using a Gaussian distribution with mean $\mathbf{0}$ and standard deviation σ.
2. Apply the transformation $\mathbf{Y}'_\mathbf{x} = [\![P]\!](\mathbf{Y}_\mathbf{x})$. Note that $\mathbf{Y}'_\mathbf{x}$ is a random variable; intuitively, it captures the output of the program P when executed on the input $\mathbf{Y}_\mathbf{x}$. Observe, however, that $\mathbf{Y}'_\mathbf{x}$ is not required to be Gaussian.
3. Compute and return the expectation of $\mathbf{Y}'_\mathbf{x}$.

One can see that the smoothed semantics $\overline{[\![P]\!]}_\beta$ of P is the function mapping $\mathbf{x}$ to the above output, and that this definition is consistent with Definition 2. With this model in mind, we now define a probabilistic semantics that lets us define $\overline{[\![P]\!]}_\beta$ structurally. The key idea here is to define a semantic map $[\![P]\!]^\#$ that models the effect of P on a random variable. For example, in the above discussion, we have $\mathbf{Y}'_\mathbf{x} = [\![P]\!]^\#(\mathbf{Y}_\mathbf{x})$. The semantics is very similar to Kozen's probabilistic semantics of programs [13]. Our smooth interpretation algorithm implements an approximation of this semantics.

Preliminaries For the rest of the section, we will let $\mathbf{Y}$ be a random vector over $\mathbb{R}^K$. If the probability density function of $\mathbf{Y}$ is $h_\mathbf{Y}$, we write $\mathbf{Y} \sim h_\mathbf{Y}$. In an abuse of notation, we will define the probabilistic semantics of a boolean expression B as a map $[\![B]\!]^\# : \mathbb{R}^K \to [0, 1]$:

$$[\![B]\!]^\#(\mathbf{Y}) = \mathsf{Prob}_\mathbf{Y}[[\![B]\!](\mathbf{Y}) = 1]. \tag{4}$$

Assignments Modeling assignments is a challenge because assignments can introduce dependencies between variables, which can often lead to very pathological distribution functions. For example, consider a program with two variables $\mathtt{x}_0$ and $\mathtt{x}_1$, and consider the semantics of the assignment $\mathtt{x}_0 := \mathtt{x}_1$.

$$[\![\mathtt{x}_0 := \mathtt{x}_1]\!]^\#(\mathbf{Y}) = \mathbf{Y}'$$

After the assignment, the probability density $h_{\mathbf{Y}'}(x_0, x_1)$ of $\mathbf{Y}'$ will have some peculiar properties. Specifically, $h'_Y(x_0, x_1) = 0$ for any $x_0 \neq x_1$, yet it's integral over all $\mathbf{x}$ must equal one. The only way to satisfy this property is to define the new probability density in terms of the Dirac δ function. The Dirac δ is defined formally as a function which satisfies $\delta(x) = 0$ for all $x \neq 0$, and with the property that $\int_{-\infty}^{\infty} \delta(x) f(x)\, dx = f(0)$; informally, it can be thought of as a Gaussian with an infinitesimally small σ.

Now we define the semantics of assignment as follows:

$$[\![\mathtt{x}_\mathtt{i} := E]\!]^\#(\mathbf{Y}) = \mathbf{Y}' \sim h_{\mathbf{Y}'} \tag{5}$$

where $h_{\mathbf{Y}'}$ is defined by the following integral.

$$h_{\mathbf{Y}'}(\mathbf{x}') = \int_{\mathbf{x} \in \mathbb{R}^K} D(E, i, \mathbf{x}, \mathbf{x}') \cdot h_\mathbf{Y}(\mathbf{x}) d\mathbf{x}$$

Here, $h_\mathbf{Y}$ is the density of $\mathbf{Y}$, and the function $D(E, i, \mathbf{x}, \mathbf{x}')$ above is a product of deltas $\delta(\mathbf{x}(0) - \mathbf{x}'(0)) \cdot \ldots \cdot \delta([\![E]\!](\mathbf{x}) - \mathbf{x}'(i)) \cdot \ldots \cdot \delta(\mathbf{x}(K-1) - \mathbf{x}'(K-1))$, which captures the condition that all variables must have their old values, except for the i^{th} variable, which must now be equal to $[\![E]\!](\mathbf{x})$.

From this definition, we can see that the probability density for $\mathbf{Y}' = [\![\mathtt{x}_0 := \mathtt{x}_1]\!]^{\#}(\mathbf{Y})$ will equal $h_{\mathbf{Y}'}(\mathbf{x}')$

$$\begin{aligned} &= \int_{x_0\in\mathbb{R}}\int_{x_1\in\mathbb{R}} \delta(\mathbf{x}(1) - \mathbf{x}'(0))\cdot\delta(\mathbf{x}(1)-\mathbf{x}'(1))\cdot h_{\mathbf{Y}}(\mathbf{x})d\mathbf{x} \\ &= \int_{x_0\in\mathbb{R}} \delta(\mathbf{x}(0)' - \mathbf{x}'(1))\cdot h_{\mathbf{Y}}(\mathbf{x}(0),\mathbf{x}'(0))d\mathbf{x}(0) \\ &= \delta(\mathbf{x}(0)' - \mathbf{x}'(1))\cdot \int_{x_0\in\mathbb{R}} h_{\mathbf{Y}}(\mathbf{x}(0),\mathbf{x}'(0))d\mathbf{x}(0)\end{aligned}$$

It is easy to see from the properties of the Dirac delta that the solution above has exactly the properties we want.

Conditionals Defining the semantics of conditionals and loops will require us to work with conditional probabilities. Specifically, let B be a boolean expression. We use the notation $(\mathbf{Y} \mid B)$ to denote the random variable with probability distribution

$$h_{\mathbf{Y}|B}(\mathbf{x}) = \begin{cases} \frac{h_{\mathbf{Y}}(\mathbf{x})}{[\![B]\!]^{\#}} & \text{if } [\![B]\!](\mathbf{x}) = 1 \\ 0 & \text{otherwise.}\end{cases} \quad (6)$$

Intuitively, the distribution of $(\mathbf{Y} \mid B)$ is obtained by restricting the domain of the distribution of $\mathbf{Y}$ to the points where the boolean condition B is satisfied. Thus, $(\mathbf{Y} \mid B)$ is a random variable following a *truncated distribution*. For example, if $\mathbf{Y}$ follows a 1-D normal distribution and $[\![B]\!]$ equals $(x > 0)$, then the density function of $(\mathbf{Y} \mid B)$ is a *half-Gaussian* which has the shape of a Gaussian for $x > 0$, and equals 0 elsewhere. Note that the density function of $(\mathbf{Y} \mid B)$ is normalized so that its integral still equals 1.

Conditionals also require us to define the notion of *mixture* of random variables to model the joining of information about the behavior of the program on two different branches. Consider two vector-valued random variables $\mathbf{Y_1}$ and $\mathbf{Y_2}$ such that $\mathbf{Y_1} \sim h_1$ and $\mathbf{Y_2} \sim h_2$. We define the *mixture* of $\mathbf{Y_1}$ and $\mathbf{Y_2}$ with respect to a constant $v \geq 0$ to be a random variable $\mathbf{Y_1} \sqcup_v \mathbf{Y_2}$ with the following density function:

$$h(\mathbf{x}) = v\cdot h_1(\mathbf{x}) + (1-v)\cdot h_2(\mathbf{x}). \quad (7)$$

By combining all of the above concepts, we can now give a concise definition of the probabilistic semantics of programs.

Definition 3 (Probabilistic semantics of programs)**.** Let P' be an arbitrary subprogram of P (including, possibly, P itself), and let $\mathbf{Y}$ be a random vector over $\mathbb{R}^K$.

The probabilistic semantics of P' is a partial function $[\![P']\!]^{\#}$ defined by the following rules: $[\![P']\!]^{\#}$ is as follows:

- $[\![\mathtt{skip}]\!]^{\#}(\mathbf{Y}) = \mathbf{Y}$.
- $[\![\mathtt{x_i} := E]\!]^{\#}(\mathbf{Y}) = Y'$ defined by Equation (5).
- $[\![P_1; P_2]\!]^{\#}(\mathbf{Y}) = [\![P_2]\!]^{\#}([\![P_1]\!]^{\#}(\mathbf{Y}))$.
- $[\![\mathtt{if}\ B\ \mathtt{then}\ P_1\ \mathtt{else}\ P_2]\!]^{\#}(\mathbf{Y}) =$
  let $v = [\![B]\!]^{\#}(\mathbf{Y})$ in
  $([\![P_1]\!]^{\#}(\mathbf{Y} \mid B)) \sqcup_v ([\![P_2]\!]^{\#}(\mathbf{Y} \mid \neg B))$.
- Let $P' = \mathtt{while}\ B\ \{\ P_1\ \}$; let us also set

$$\mathbf{Y_1} = [\![P_1]\!]^{\#}(\mathbf{Y} \mid B).$$

  For all $j \geq 0$, let us define a map:

$$[\![P']\!]^{\#}_j(\mathbf{Y}) = \begin{cases} \mathbf{Y} & \text{if } j = 0 \\ \text{let } v = [\![B]\!]^{\#}(\mathbf{Y}) \text{ in} & \\ \quad ([\![P']\!]^{\#}_{j-1}(\mathbf{Y_1})) \sqcup_v \mathbf{Y} & \text{otherwise.}\end{cases}$$

  Now we define: $[\![P']\!]^{\#}(\mathbf{Y}) = \lim_{j\to\infty}[\![P']\!]^{\#}_j$. □

Of particular interest in the above are the rules for branches and loops. Suppose P' equals "$\mathtt{if}\ B\ \mathtt{then}\ P_1\ \mathtt{else}\ P_2$," and suppose $[\![P']\!]^{\#}(\mathbf{Y}) \sim h'$. Now consider any $\mathbf{x} \in \mathbb{R}^K$. We note that an output of P' in the neighborhood of $\mathbf{x}$ could have arisen from either the true or the false branch of P'. These two "sources" are described by the distribution functions of the variables $[\![P_1]\!]^{\#}(\mathbf{Y} \mid B)$ and $[\![P_2]\!]^{\#}(\mathbf{Y} \mid \neg B)$. The value $h(\mathbf{x})$ is then the sum of these two "sources," weighted by their respective probabilities. This intuition directly leads to the expression in the rule for branches.

The semantics of loops is more challenging. Let P' now equal $\mathtt{while}\ B\ \{\ P_1\ \}$. While each approximation $[\![P']\!]^{\#}_j(\mathbf{Y})$ is computable, the limit $[\![P']\!]^{\#}(\mathbf{Y})$ is not guaranteed to exist. While it is possible to give sufficient conditions on P' under which the above limit exists, developing these results properly will require a more rigorous, measure-theoretic treatment of probabilistic semantics than what this paper offers. Fortunately, our smooth interpretation algorithm does not depend on the existence of this limit, and only uses the approximations $[\![P']\!]^{\#}_j(\mathbf{Y})$. Therefore, in this paper, we simply *assume* the existence of the above limit for all P' and $\mathbf{Y}$. In other words, $[\![P']\!]^{\#}$ is always a total function.

Smoothed semantics The smoothed semantics of P is now easily defined in terms of the probabilistic semantics:

Definition 4 (Smoothed semantics)**.** Let $\beta > 0$, $\mathbf{x} \in \mathbb{R}^K$, and let $\mathbf{Y} \sim f_{\mathbf{x},\sigma}$, where $\sigma = (\beta, \ldots, \beta)$ and $f_{\mathbf{x},\sigma}$ is as in Definition 2. Further, suppose that for all subprograms P' of the program P, $[\![P']\!]^{\#}$ is a total function. The smoothed semantics of the program P is defined as

$$\overline{[\![P]\!]}_\beta(\mathbf{x}) = \mathsf{Exp}[[\![P]\!]^{\#}(\mathbf{Y})]. \quad \Box$$

2.4 Properties of smoothed semantics

Before finishing this section, let us relate our smoothed semantics of programs to the traditional, crisp semantics. We note that as β becomes smaller, $\overline{[\![P]\!]}_\beta$ approaches $[\![P]\!]$. The one exception to the above behavior involves "isolated" features of $[\![P]\!]$ defined over measure-0 subsets of the input space. For example, suppose P represents the function "$\mathtt{if}\ x = 0\ \mathtt{then}\ 1\ \mathtt{else}\ 0$"—here, the P outputs 1 only within a measure-0 subset of $\mathbb{R}$. In this case, $\overline{[\![P]\!]}_\beta$ evaluates to 0 on all inputs, for every value of β.

Observe, however, that in this case, the set of inputs x for which $\overline{[\![P]\!]}_\beta(x)$ and $[\![P]\!](x)$ disagree has measure 0. This property can be generalized as follows. For functions $f : \mathbb{R}^K \to \mathbb{R}^K$ and $g : \mathbb{R}^K \to \mathbb{R}^K$, let us say that f and g agree *almost everywhere*, and write $f \approx g$, if the set of inputs x for which $f(x) \neq g(x)$ has measure 0. We can prove the following "limit theorem":

Theorem 2. $\lim_{\beta\to 0}\overline{[\![P]\!]}_\beta \approx [\![P]\!]$.

The same property can be restated in terms of the probabilistic semantics. To do so, we will have to relate our probabilistic semantics with our crisp semantics. This can be done by studying the behavior of the probabilistic semantics when the input distribution has all its mass concentrated around a point $\mathbf{x_0}$. Such a random variable can be modeled by a Dirac delta, leading to the following theorem.

Theorem 3. *Let $\mathbf{Y}$ be a random vector with a distribution:*

$$h(\mathbf{x}) = \delta(\mathbf{x} - \mathbf{x_0})$$

Then the variable $[\![P]\!]^{\#}(\mathbf{Y})$ follows the distribution

$$h'(\mathbf{x}) = \delta(\mathbf{x} - [\![P]\!](\mathbf{x_0}))$$

Informally speaking, the random vector $\mathbf{Y}$ is "equivalent" to the (non-random) vector $\mathbf{x_0}$. The above theorem states that $[\![P]\!]^{\#}(\mathbf{Y})$ is also "equivalent," in the same sense, to the output $[\![P]\!](\mathbf{x_0})$. Theorem 2 can now be obtained using the relationship between our probabilistic and smoothed semantics.

3. Smooth interpretation

The last section defined Gaussian smoothing in terms of a probabilistic semantics based on transformations of random variables. The semantics of Definition 3, however, do not lend themselves to algorithmic implementation. The immediate problem is that many of the evaluation rules require complex manipulations on arbitrary probability density functions. More abstractly, the smoothed semantics of Definition 3 essentially encodes the precise behavior of the program on all possible inputs, a losing proposition from the point of view of a practical algorithm. This section describes four approximations that form the basis of our practical smooth interpretation algorithm. The approximations focus on four operations that introduce significant complexity in the representation of density functions. These approximations are guided by our choice of representation for the probability density functions; arguably the most important design decision in our algorithm.

- **Conditional probabilities:** The use of conditional probabilities allows the probability density functions to encode the path constraints leading to a particular point in the execution. Path constraints can be arbitrarily complex, so having to encode them precisely as part of the representation of the state poses a challenge for any practical algorithm.
- **Mixture:** The mixture procedure combines the state of two different paths in the program without any loss of information. This poses a problem because it means that by the end of the execution, the probability density function encodes the behavior of the program on all possible paths.
- **Update:** The semantics of update involve complex integrals and Dirac deltas; a practical smoothing algorithm needs a simple update procedure.
- **Loop approximation:** The probabilistic semantics defines the behavior of loops as the limit of an infinite sequence of operations. In order to make smooth interpretation practical, we must be able to efficiently approximate this limit.

3.1 Random variables as Gaussian mixtures

The first important design decision in the smooth interpretation algorithm is that each random variable is approximated by a random variable following a *Gaussian mixture distribution*. The density function $h_{\mathbf{Y}}$ of a variable following a Gaussian mixture distribution has the following form:

$$h_{\mathbf{Y}} = \sum_{i=0..M} w_i \cdot \prod_{j=0..K} f_{\mu_i(j),\beta_i(j)} = \sum_{i=0..M} w_i \cdot f_{\mu_i,\beta_i} \quad (8)$$

where f_{μ_i,β_i} is the density function of a Gaussian with mean $\mu_i \in F^K$ and standard deviation $\beta_i \in \mathbb{R}^K$.

The above equation represents a distribution with M distinct components, where each component is a weighted Gaussian in K independent variables. The weights w_i must add up to 1 to ensure that the function above is a legal probability density function.

By assuming that all density functions have the form above, our system can represent a random variable very compactly as a list of triples of the form

$$h_{\mathbf{Y}} = [(w_0, \mu_0, \beta_0), \ldots, (w_M, \mu_M, \beta_M)]$$

where $\mu_i, \beta_i \in \mathbb{R}^K$ for all i.

Conditional probabilities The probabilistic semantics makes extensive use of conditional probabilities. Unfortunately, if a random variable $\mathbf{Y}$ has a density function of the form defined in Equation (8), the variable $\mathbf{Y}|B$ for some Boolean expression B is most likely not going to have that form. Our solution is to find an approximation of $\mathbf{Y}|B$ that matches our canonical form. Let $h_{\mathbf{Y}} = [m_0, m_1, \ldots m_N]$ where $m_i = (w_i, \mu_i, \beta_i)$ is a Gaussian component. Our goal is to find a density function $h_{\mathbf{Y}|B} = [m'_0, m'_1, \ldots m'_N]$ that best approximates the probability density of $\mathbf{Y}|B$. To simplify the problem, we require that $m'_i = (t_i, \mu_i, \beta_i)$.

With a bit of calculus, it is easy to see that the solution is to let

$$t_i = w_i \cdot \frac{[\![B]\!]^{\#}(\mathbf{Y}_i)}{[\![B]\!]^{\#}(\mathbf{Y})},$$

where $\mathbf{Y}_i$ follows the Gaussian distribution with mean μ_i and standard deviation β_i. This definition still requires us to compute $[\![B]\!]^{\#}(\mathbf{Y}_i)$, but it is easy to do so approximately at runtime. Example 1 showed how to compute this exactly for the case when B is of the form $(\mathtt{x_i} < a)$ or $(a < \mathtt{x_i} < c)$. Other cases can be handled heuristically by rules such as

$$\begin{array}{lcl} [\![B_1 \wedge B_2]\!]^{\#}(\mathbf{Y}_i) & = & [\![B_1]\!]^{\#}(\mathbf{Y}_i) \cdot [\![B_2]\!]^{\#}(\mathbf{Y}_i). \\ [\![\neg B]\!]^{\#}(\mathbf{Y}_i) & = & 1 - [\![B]\!]^{\#}(\mathbf{Y}_i). \end{array}$$

As for computing $[\![B]\!]^{\#}(\mathbf{Y})$, it turns out we do not have to. If we observe the rules in Definition 3, we see that every time we use $\mathbf{Y}|B$, the resulting density function gets multiplied by a factor $[\![B]\!]^{\#}(\mathbf{Y})$. Therefore, we can perform a few simple adjustments to the evaluation rules of Definition 3 to avoid having to renormalize the representation of the distribution after every step. Skipping normalization has other benefits besides efficiency. Specifically, the total weight of the un-normalized representation of the distribution gives us a good estimate of the probability that the execution will reach a particular program point. This becomes useful when dealing with loop termination, as we will see further ahead.

Mixture The mixture operation is easy to implement accurately on our chosen representation. Given two random variables $\mathbf{X} \sim h_{\mathbf{X}}$ and $\mathbf{Y} \sim h_{\mathbf{Y}}$ with

$$\begin{array}{l} h_{\mathbf{X}} = [(w_0^{\mathbf{X}}, \mu_0^{\mathbf{X}}, \beta_0^{\mathbf{X}}), \ldots, (w_M^{\mathbf{X}}, \mu_M^{\mathbf{X}}, \beta_M^{\mathbf{X}})] \\ h_{\mathbf{Y}} = [(w_0^{\mathbf{Y}}, \mu_0^{\mathbf{Y}}, \beta_0^{\mathbf{Y}}), \ldots, (w_N^{\mathbf{Y}}, \mu_N^{\mathbf{Y}}, \beta_N^{\mathbf{Y}})], \end{array}$$

the mixture $\mathbf{X} \sqcup_v \mathbf{Y}$ has the probability density function shown below. It simply contains the components of both of the initial distributions weighted by the appropriate factor. Note that the result below assumes normalized representations for $h_{\mathbf{X}}$ and $h_{\mathbf{Y}}$, if they are unnormalized, the scaling factor is already implicit in the weights, and all we have to do is concatenate the two lists.

$$[(v{\cdot}w_0^{\mathbf{X}}, \mu_0^{\mathbf{X}}, \beta_0^{\mathbf{X}}), ..., (v{\cdot}w_M^{\mathbf{X}}, \mu_M^{\mathbf{X}}, \beta_M^{\mathbf{X}}), ((1{-}v){\cdot}w_0^{\mathbf{Y}}, \mu_0^{\mathbf{Y}}, \beta_0^{\mathbf{Y}}), ...]$$

One drawback of this representation is that it can grow significantly in the process of applying mixtures. Every time we mix a random variable with a representation of size M with one of size N, the result is a random variable with a representation of size $M+N$. Performing mixtures in the process of evaluating the probabilistic semantics leads to an explosion in the number of components; by the end of the program, the total number of components will be equal to the number of possible paths through the program, an enormous number even for modestly sized programs.

To prevent this explosion, our interpreter establishes a bound N, so if a mixture causes the number of components to exceed N, the result is approximated with a representation of size N. The approximation is performed according to the pseudocode in Algorithm 1.

In this algorithm, the *merge* operation takes two components in the distribution $h_{\mathbf{X}}$ and replaces them with a single component computed according to the following function.

Algorithm 1 $Restrict(h_{\mathbf{X}}, N)$

```
Input: Density function
        $h_{\mathbf{X}} = [(w_0^{\mathbf{X}}, \mu_0^{\mathbf{X}}, \beta_0^{\mathbf{X}}), \ldots, (w_M^{\mathbf{X}}, \mu_M^{\mathbf{X}}, \beta_M^{\mathbf{X}})]$
    of size $M$ needs to be reduced to size $N < M$.
1: while $Size(h_{\mathbf{X}}) > N$ do
2:     find pair $(w_i^{\mathbf{X}}, \mu_i^{\mathbf{X}}, \beta_i^{\mathbf{X}}), (w_j^{\mathbf{X}}, \mu_j^{\mathbf{X}}, \beta_j^{\mathbf{X}}) \in h_{\mathbf{X}}$ that
       minimizes $cost((w_i^{\mathbf{X}}, \mu_i^{\mathbf{X}}, \beta_i^{\mathbf{X}}), (w_j^{\mathbf{X}}, \mu_j^{\mathbf{X}}, \beta_j^{\mathbf{X}}))$
3:     $h_{\mathbf{X}}$ := replace components $i$ and $j$ in $h_{\mathbf{X}}$ with
       $merge((w_i^{\mathbf{X}}, \mu_i^{\mathbf{X}}, \beta_i^{\mathbf{X}}), (w_j^{\mathbf{X}}, \mu_j^{\mathbf{X}}, \beta_j^{\mathbf{X}}))$
4: end while
5: return $h_{\mathbf{X}}$
```

$$
merge((w^a, \mu^a, \beta^a), (w^b, \mu^b, \beta^b)) = (w', \mu', \beta')
$$
$$
\text{where} \quad \begin{array}{rcl} w' & = & w^a + w^b \\ \mu' & = & (\mu^a \cdot w^a + \mu^b \cdot w^b)/(w^a + w^b) \\ \beta' & = & \frac{w^a \cdot (\beta^a + 2\|\mu^a - \mu'\|^2) + w^b \cdot (\beta^b + 2\|\mu^b - \mu'\|^2)}{w^a + w^b} \end{array}
$$

The mean is computed as one would expect, as the weighted average of the two components to be merged; the new variance will grow in proportion to how far the old means were from the mean of the merged component. The definition of merge is optimal: it produces the best possible approximation of the two components.

The algorithm merges in a greedy fashion, always trying to merge components that will cause the smallest possible change to the distribution $h_{\mathbf{X}}$. The cost of a merge is computed according to the following function.

$$
cost((w^a, \mu^a, \beta^a), (w^b, \mu^b, \beta^b)) = w^a \|\mu^a - \mu'\| + w^b \|\mu^b - \mu'\|
$$

where $\mu' = (\mu^a \cdot w^a + \mu^b \cdot w^b)/(w^a + w^b)$. The cost is an estimate of how much information will be lost when the two components are merged into one. The algorithm as stated is quite inefficient in that it does a lot of repeated work, but it is simple to state and implement, and for the values of N we used for our experiments, it was reasonably efficient.

As we have stated before, each component in the density function $h_{\mathbf{X}}$ carries information of the behavior of the program on a particular path. The *Restrict* operation has the effect of summarizing the behavior of multiple paths in a single component. The result is an algorithm with very selective path sensitivity; paths with very high probability, or whose behavior is very different from the others are analyzed very accurately, while paths with similar behaviors are merged into a single summary component. The price paid for this selective path sensitivity is a small amount of discontinuity in the smooth semantics. This is because a small change in one component can cause components to merge differently. In Section 5, we will explore this effect in the context of one of our benchmarks.

Update Our interpreter implements assignments according to the following rule: If $\mathbf{X} \sim h_{\mathbf{X}} = [q_0, q_1, \ldots q_N]$ where $q_i = (w_i, \mu_i, \beta_i)$, then

$$
[\![x_j = E]\!]^{\#}(\mathbf{X}) = \mathbf{X}'
$$

where $\mathbf{X}' = [q'_0, q'_1, \ldots q'_N]$, and $q'_i = (w_i, \mu_i[j \mapsto [\![E]\!](\mu_i)], \beta_i)$.

From the rule for assignments in our probabilistic semantics, it is possible to determine that the above definition is optimal under the constraint that β_i remain unchanged. We could have gotten a more accurate definition if we allowed β_i to change, but the added accuracy would have been at the expense of greater complexity.

Loop approximation A final approximation has to do with loop termination. Consider a program P of form `while` B { ... }, and recall the approximations $[\![P]\!]_j^{\#}$ defined in Definition Definition 3. Our interpreter uses $[\![P]\!]_j^{\#}(\mathbf{Y})$, for a sufficiently high j, as an approximation to the idealized output $[\![P]\!]^{\#}(\mathbf{Y})$. An advantage of this approach is that $[\![P]\!]_j^{\#}(\mathbf{Y})$ is defined for all j, even when the limit in the definition of $[\![P]\!]^{\#}(\mathbf{Y})$ does not exist.

To select a suitable value of j, we apply a simple heuristic. Suppose $\mathbf{Y}'$ is the unnormalized random variable obtained by executing the body of the loop j times on the input random variable $\mathbf{Y}$. Our system monitors the weight of $\mathbf{Y}'$. Because our representations are unnormalized, the weight of $\mathbf{Y}'$ is an estimate of how much an execution with j iterations will contribute to the end solution. Further, as weights get multiplied along paths, an execution with $j' > j$ loop iterations will contribute strictly less than $\mathbf{Y}'$. When the weights associated with $\mathbf{Y}'$ become sufficiently small (less than 10^{-8} in our implementation), we determine that an execution with j or more iterations has nothing to contribute to the output of the smooth interpreter, and take $[\![P]\!]_j^{\#}$ as an approximation of $[\![P]\!]^{\#}$.

The above strategy essentially amounts to a dynamic version of loop unrolling, where the amount of unrolling is based on the probability that the loop will iterate a certain number of iterations. Once that probability falls below threshold, the unrolling is terminated.

Example 3. Let us consider the following program P, previously seen in Example 2:

$$
\texttt{if } \texttt{x}_0 > 0 \texttt{ then skip else } \texttt{x}_0 := 0
$$

We sketch the smooth interpretation of P on the input 0.5 and $\beta = 1$. Here, the input random variable $\mathbf{Y}$ has the representation $h_{\mathbf{Y}} = [(1, 0.5, 1)]$. To propagate it through P, we must first compute $[\![\texttt{x}_0 > 0)]\!]^{\#}(\mathbf{Y})$. This is done following Example 1—the result is approximately 0.69. Thus, the random variables $\mathbf{Y}_1$ and $\mathbf{Y}_2$ propagated into the true and the false branch of P respectively have densities $h_{\mathbf{Y}_1} = [(0.69, 0.5, 1)]$ and $h_{\mathbf{Y}_2} = [(0.31, 0.5, 1)]$.

Now we have $[\![\texttt{skip}]\!]^{\#}(\mathbf{Y}_1) = \mathbf{Y}_1$ and $[\![\texttt{x}_0 := 0]\!]^{\#}(\mathbf{Y}_2) = [(0.31, 0, 1)]$. The estimation of $\mathbf{Y}_2 = [\![P]\!]^{\#}(\mathbf{Y})$ thus has the representation $h_{\mathbf{Y}_2} = [(0.69, 0.5, 1), (0.31, 0, 1)]$, and the output of the smooth interpretation is $0.69 \times 0.5 = 0.345$.

Now, let P' equal $\texttt{if } \texttt{x}_0 > 5 \texttt{ then skip else } \texttt{x}_0 := \texttt{x}_0 + 5$ and consider the program $P; P'$. To do smooth interpretation of this program, we must propagate $h_{\mathbf{Y}_2}$ through P'. The resulting distribution $h_{\mathbf{Y}_3}$ will equal:
$[(0.69, 0.5, 1), (0.31, 0, 1), (2.3 \cdot 10^{-6}, 5.5, 1), (9.8 \cdot 10^{-8}, 5, 1)]$,
Now, if our limit of components equals 3, then the *Restrict* operation will merge the last two components, which together have such a small weight that the effect of the merge on the accuracy of the solution will be negligible.

4. Parameter synthesis

In this section, we introduce the *parameter synthesis* problem for embedded control programs as a concrete application of smooth interpretation.

4.1 Problem definition

Motivating example: Thermostat Let us consider a simple cyber-physical system: a thermostat that controls the temperature of a room by switching on or off a heater. The program controlling the thermostat is shown in Figure 3-(a). The program repeatedly reads in the temperature of the room using a routine `readTemp`. If this temperature is above a certain threshold `tOff`, the heater is switched off; if it is below a threshold `tOn`, the heater is switched on. We assume that the time gap between two successive temperature readings equals a known constant `dt`. Also, the differential equations

(a) Source code of controller:

```
tOff := ??; tOn := ??;
repeat {
  temp := readTemp();
  if (isOn() and temp > tOff)
      switchHeater (Off);
  else if (not isOn() and temp < tOn)
      switchHeater (On);     }
```

(b) Temperature variation in warming phase:

$$\frac{d}{dt} temp = -k \cdot temp + h$$

(c) Temperature variation in cooling phase:

$$\frac{d}{dt} temp = -k \cdot temp$$

Figure 3. Parameter synthesis in a thermostat

that govern the change of room temperature during the warming or cooling phase are known from the laws of physics and the characteristics of the heater. These are as in Figure 3-(b) and Figure 3-(c).

What we do not know are the values of the thresholds `tOn` and `tOff`. These thresholds are the *control parameters* of the thermostat. For the room's temperature to be controlled desirably, these parameters must be instantiated properly. At the same, the correct values of these parameters do not easily follow from high-level programming insights. Consequently, we aim to *synthesize* these values automatically.

Let a *trajectory* be a finite sequence of real values. An instance of our synthesis problem consists of the system $\mathcal{S}$ described above (consisting of the control program, the model of temperature variation, as well as the constant `dt`), and a *specification* $Spec$ consisting of a finite set of *reference trajectories*. Intuitively, the i-th value $\tau[i]$ in a reference trajectory τ is the *desired* temperature of the room during the i-th loop iteration. In more complex applications, a trajectory may need to be defined as a sequence of *observable program states*—for brevity, we avoid this generalization.

Let us now fix an instantiation of the parameters `tOn` and `tOff` with real values. Now consider any reference trajectory τ; the first value in τ corresponds to the *initial condition* of τ and is denoted by $Init(\tau)$. Now let us consider an execution of the system from a point where the room is at temperature $Init(\tau)$ (note that this execution is deterministic). Let τ_{Obs} be the sequence of temperature readings returned by the first $N = |\tau|$ calls to `readTemp`. This sequence captures the observable behavior of the system under the present instantiation of the control parameters and the same initial condition as τ, and is known as the *observed trajectory* of $\mathcal{S}$. We refer to the distance (defined for this example as the L_2-distance) between τ and τ_{Obs} as the *τ-error* between $\mathcal{S}$ and $Spec$. The *error* between $\mathcal{S}$ and $Spec$ is the sum of τ-errors between $\mathcal{S}$ and $Spec$ over all $\tau \in Spec$ (note that the this error is a function of `tOn` and `tOff`). The goal of parameter synthesis is to find values for the control parameters such that the error between $\mathcal{S}$ and $Spec$ is minimized.

Observe that we do not aim to match the observed and reference trajectories of the system exactly. This is because in our setting, where parameters range over real domains, a problem may often lack exact solutions but have good approximate ones. When this happens, a good synthesis technique should find the best approximate answer rather than report that the program could not be made to match the specification.

Parameter synthesis as numerical optimization More generally, an instance of the parameter synthesis problem consists of a physical system with known characteristics, a program with uninitialized parameters that controls it, the rates at which the program invokes its sensors and actuators, and a reference trajectory defining the desired observable behavior of the system. The goal of parameter synthesis is once again to minimize the distance between the reference and observed trajectories. A rigorous definition of the problem would require us to formally define the semantics of cyber-physical systems—we omit this for brevity and also because we do not solve the parameter synthesis problem directly. Instead, we reduce the problem to a problem of numerical optimization.

```
float singleTrajError (float tOn, float tOff,
        trajectory τ) {
  float temp := Init(τ);
  float Err := 0;
  bool isOn = false;
  for (i := 0; i < N; i := i + 1) {
    Err := Err + (temp − τ[i])^2;
    if(isOn){ temp := temp + dt * K * (h - temp); }
    else{ temp := temp * (1 - K * dt); }
    if (isOn and temp > tOff)
        isOn := false;
    else if (not isOn and temp < tOn)
        isOn := true;
  }
  return √Err; }
```

Figure 4. Parameter synthesis in a thermostat (contd.)

We illustrate the reduction using the thermostat example. Consider the procedure `singleTrajError` in Figure 4. This procedure is obtained by "weaving" the dynamics of the room's temperature into the code for the controller (we take `dt` to be an approximation of an infinitesimal timestep), and recording the L_2-distance between the observed and reference trajectories in a special variable `Err`. The input variables of the routine are `tOn` and `tOff`—the control parameters that we want to synthesize. Now let $Error$ be a function returing the sum of the return values of `singleTrajError` for $\tau \in Spec$ (`tOn` and `tOff` are fixed), and consider the problem of finding an initial assignment to the inputs of $Error$ such that the value it returns is minimized. It is easy to see that assuming small `dt`, an optimal solution to this problem is also an optimal solution to the parameter synthesis problem for the system $\mathcal{S}$, and vice-versa.

Generalizing, let P be any program with a single real-valued output variable `out`. The *output minimization problem* for P is to find an input $\mathbf{q}$ to P such that the value of the output q' (we have $q' = [\![P]\!](\mathbf{q})$) is minimized. The parameter synthesis problem for embedded control applications manipulating floating-point data can be reduced to the output minimization problem for programs such as P.

4.2 Algorithm

Algorithm 2 SMOOTHED-NELDER-MEAD$(\mathcal{F}, \eta, \epsilon)$

```
1:  repeat
2:      Randomly select starting point $\mathbf{p} \in \mathbb{R}^K$
3:      Select large $\beta > 0$.
4:      repeat
5:          $\mathbf{p} :=$ NELDER-MEAD(SMOOTH$(\mathcal{F}, \beta), \mathbf{p}, \eta, \epsilon)$
6:          $\beta :=$ new value smaller than old value of $\beta$.
7:      until $\beta < \epsilon$
8:      if $\mathcal{F}(\mathbf{p}) < \mathcal{F}(\mathbf{bestp})$ then $\mathbf{bestp} := \mathbf{p}$
9:  until timeout
10: return $\mathbf{bestp}$
```

Our approach to the output minimization problem (and hence, parameter synthesis) is based on local optimization. Specifically, we use *Nelder-Mead simplex search* [15], a derivative-free local optimization technique that is implemented in the GNU Scientific Library (GSL) [10]. Let us say we want to minimize a function $\mathcal{F} : \mathbb{R}^K \to \mathbb{R}$. To do so using Nelder-Mead search, we issue a call $\textsc{Nelder-Mead}(\mathcal{F}, \mathbf{p}, \eta, \epsilon)$, where $\mathbf{p} \in \mathbb{R}^K$ is an *initial point*, $\eta > 0$ is a *step size*, and $\epsilon > 0$ is a *stopping threshold*. On this call, the search algorithm starts by constructing a *simplex* around the point $\mathbf{p}$ (the maximum distance of the points in this simplex from $\mathbf{p}$ is a defined by η). This simplex is now iteratively shifted through the search landscape. More precisely, in each iteration, the algorithm uses a simple geometrical transformation to update the simplex so that $\mathcal{F}$ has "better" values at the extrema of the simplex. After each iteration, the value of $\mathcal{F}$ at the "best" vertex of the simplex is recorded. The search terminates when we reach an approximate fixpoint—i.e., when the best value of $\mathcal{F}$ between two iterations differs by less than ϵ. At this point the simplex is contracted into one point, which is returned as a local minimum.

Naive and smoothing-based algorithms A simple way to solve parameter synthesis using Nelder-Mead search would be to generate the function $Error$, then call $\textsc{Nelder-Mead}(Error, \mathbf{p_{in}}, \eta, \epsilon)$ with suitable η and ϵ; the function could even be called repeatedly from a number of different starting points $\mathbf{p_{in}}$ to increase the likelihood of hitting a global minima. Unfortunately, $Error$ is only available as an imperative program, and in most cases of interest to us, proves to be too ill-behaved to be minimized using Nelder-Mead search.

For example, consider again our thermostat, where $Error$ is a function of `tOn` and `tOff`. In Figure 5-(a), we plot the value of $Error$ as `tOn` and `tOff` vary—this plot is the "search landscape" that the search algorithm navigates. In Figure 5-(b), we plot the magnitude of the gradient of $Error$ vs `tOn` and `tOff`. The plots are "thermal" plots, where lighter colors indicate a high value. Now note the highly irregular nature of the search landscape. In particular, note the black squares in the bottom left corner of the gradient plot: these represent "plateaus" where the gradient is 0 but the value of $Error$ is far from minimal. It is therefore not a surprise that empirically, Nelder-Mead search fails to solve the parameter synthesis problem even in this simple-looking example.

The problem of plateaus and discontinuities goes away with the introduction of smoothing. Consider the algorithm for *smoothed Nelder-Mead search* shown in Algorithm 2. This time, local search operates on the smoothed version of $Error$—we start with a highly smooth search landscape, then progressively reduce the value of the smoothing parameter β to increase the accuracy of the analysis. Of course, the accuracy of smoothing and the tradeoffs of accuracy and scalability also depend on the path-sensitivity of smooth interpretation. We do not believe that there is a single policy of path-sensitivity that will be successful for all applications; therefore, we allow the user to play with various strategies in different benchmark applications.

Let us now consider the thermostat application again. Let $K = 1.0$, $h = 150.0$, and $dt = 0.5$. Also, let us use a partially path-sensitive smooth interpretation where we track only up to 2 paths simultaneously—i.e., an abstract state has at most 2 components. Figures 5-(c)-(h) depict the values of $Error$ and its gradient under this smooth interpretation, for various values of β. Note how the gradient smooths out into a uniform slope as β increases—empirically, this causes the search to converge rapidly.

5. Evaluation

In this section, we describe a series of experiments that we performed to evaluate the benefits of smoothed Nelder-Mead search

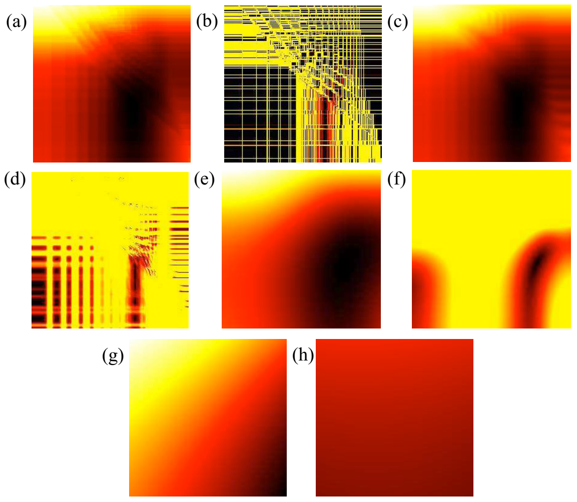


Figure 5. (a) $Error$ vs. $(\texttt{tOn}, \texttt{tOff})$: no smoothing. (b) Magnitude of gradient of $Error$: no smoothing. (c) $Error$: $\beta^2 = 0.5$. (d) Gradient plot: $\beta^2 = 0.5$. (e) $Error$: $\beta^2 = 5$. (f) Gradient plot: $\beta^2 = 5$. (g) $Error$: $\beta^2 = 50$. (h) Gradient plot: $\beta^2 = 50$.

in parameter synthesis. We used an implementation of numerical search available in the GNU Scientific Library (GSL). Our evaluation centers on three questions that are central to the viability of our approach:

1. Do typical control applications actually exhibit discontinuities that would be significant enough to require smoothing?
2. If discontinuities do exist, are they eliminated by smooth interpretation? We are particularly interested in how the approximations made by our algorithm affect the quality of smoothing.
3. Does smoothing actually make a difference for parameter synthesis?

To answer these questions, we analyzed three applications: the thermostat controller introduced in Section 4, a gearbox controller, and a PID controller with brakes. We have already discussed the thermostat application in some detail; also, it is less "real-world" than the other two applications. Consequently, in this section we focus on the gearbox and the PID controller.

5.1 Gearbox

The gearbox benchmark models the gearbox of a car. Here we have five gears, the i-th of which ($1 \leq i \leq 5$) has an associated efficiency curve of the form

$$\alpha(i, v) := \frac{1}{1 + (v - p_i)^2/25}.$$

Here v represents the current velocity of the car, and $\langle p_1, \ldots, p_5 \rangle = \langle 5, 15, 25, 40, 60 \rangle$ are known parameters. Note that at gear i, the gearbox achieves maximum efficiency at $v = p_i$. At gear i, the velocity of the car changes according to the equation

$$dv/dt = v \cdot \alpha(i, v) + u.$$

where u is a known constant. When the gearbox is disengaged, the velocity of the car follows the equation

$$dv/dt = -(v^2 \cdot drag)$$

where $drag$ is a known constant. We assume that $u = 5.0$ and $drag = 0.0005$.

```
while(t < T) {
  if (gear > 0)
        v := v + dt*($\alpha$(gear, v) * v + 5.0);
   else  v := v - dt*(v * v * drag);
  if (gear = 1 and v $\geq s_1$) {
       gear := 0; nxt := 2; w := 0.8;  }
  if (gear = 2 and v $\geq s_2$) {
       gear := 0, nxt := 3, w := 0.8;  }
  if (gear = 3 and v $\geq s_3$) {
       gear := 0; nxt := 4; w := 0.8; }
  if (gear = 4 and v $\geq s_4$) {
       gear := 0, nxt := 5, w := 0.8; }
  if (w < 0.0 and gear = 0)
       gear := nxt;
  t := t + dt;
  w := w - dt;  }
```

Figure 6. The gearbox benchmark (s_1, s_2, s_3, and s_4 are the control parameters)

The controller for the gearbox decides when to shift from one gear to the next. The system only allows consecutive gear transitions—i.e., from gear i, we can only shift to gear $(i + 1)$. For $i = 1$ to 4, we have a control parameter s_i whose value is the velocity at which gear i is released and gear $(i + 1)$ is engaged. There is a second complicating factor: gear changes are not instantaneous. In our model, each shift takes 0.8 seconds—thus, for 0.8 seconds after gear i is released, the gearbox stays disengaged.

We define a trajectory of the system to be the sequence of values of the velocity v at the beginning of loop iterations. We want to make the system reach a target velocity v_{target} as soon as possible; accordingly, we use a single reference trajectory of the form $\langle 0, v_{target}, v_{target}, \ldots \rangle$. Our goal is to synthesize values for the control parameters s_1-s_4 such that the L_2-distance between the observed and reference trajectories is minimized. As in the thermostat example, we write a program that folds the dynamics of the physical component (car) into the source code of the controller—it is this program that we analyze. The main loop of this benchmark is shown in Figure 6 (we have omitted the lines computing the error).

Smooth interpretation in this application requires us to run the program on Gaussian distributions corresponding to the variables s_1-s_4. This proves to be highly challenging. First, the number of paths in the system is $\Omega(2^{T/dt})$, making completely path-sensitive smooth interpretation infeasible. At the same time, indiscriminate merging of distributions propagated along different paths will lead to inaccurate smoothing, causing the minima of the smoothed program to become meaningless. In particular, the delay in the shifting of gears makes a certain level of path-sensitivity essential. If the system loses track of the concrete values of variables nxt or w, it risks losing track of how much time has elapsed since the gear got disengaged, or of what gear should be engaged next. Thus, the values of these variables must be tracked precisely across iterations.

We studied this benchmark using our implementation of smoothed Nelder-Mead search (based on the gsl library). We assumed that $\mathtt{dt} = 0.1$ and $\mathtt{T} = 20.0$. We limited the number of distinct components of the distribution to 80, a very small number compared to the astronomical number of paths in the program.

For our first experiment, we ran Nelder-Mead search with a number of initial values for the s_i, and with various degrees of smoothing. Since one of our goals was to understand the effect of β on Nelder-Mead search, all the experiments were run keeping β constant, rather than progressively reducing it as we did in Algorithm 2. The results are summarized in the plots in Figure 8. The lines in each figure show how the solution evolves with each iteration of Nelder-Mead search; the solid region at the bottom of the graph shows how the error value changes with each iteration. Each row in the figure corresponds to a different initial value $s_1 = s_2 = s_3 = s_4 = s_{ini}$. The first column, with $\beta = 0.0001$, corresponds to the case where there is virtually no smoothing; the second column involves moderate smoothing, and the last column involves a huge amount of smoothing. The correct settings are $s_1 = 14, s_2 = 24, s_3 = 40, s_4 = 65$.

The first two rows show clearly the effect of smoothing on the ability of Nelder-Mead search to find a solution. Without smoothing, the solver is stuck in a plateau where the error remains constant after every try. By contrast, the smoothed version of the problem quickly finds a good, if not necessarily optimal, solution. Note that from some starting points, the method is able to find a correct solution even in the absence of smoothing. In fact, in the third row of Figure 8, the solution found without smoothing is actually better than the solution found with smoothing.

To understand these effects better, we ran a series of experiments to help us visualize the error as a function of the different s_i. The results are shown in Figure 7. Illustrating a function of a four dimensional parameter space is tricky; for these plots, we held all the parameters constant at their optimal value and we plotted the error as a function of one of the parameters. For example, in Figure 7(a), we held s_2, s_3 and s_4 constant and plotted the error as a function of s_1 for different values of β.

The unsmoothed functions all have a very smooth region close to the middle, with big plateaus and discontinuities closer to the edges. This explains why for some cases, the unsmoothed function was able to converge to a correct solution, but for others it was completely lost. If the Nelder-Mead search starts in the smooth region, it easily converges to the correct solution, but if it starts in a plateau, it is unable to find a solution. When we do apply smoothing, the results are dramatic; even small degrees of smoothing completely eliminate the discontinuities and cause the plateaus to have a small amount of slope. This helps the Nelder-Mead search method to "see" the deep valleys at the end of the plateau.

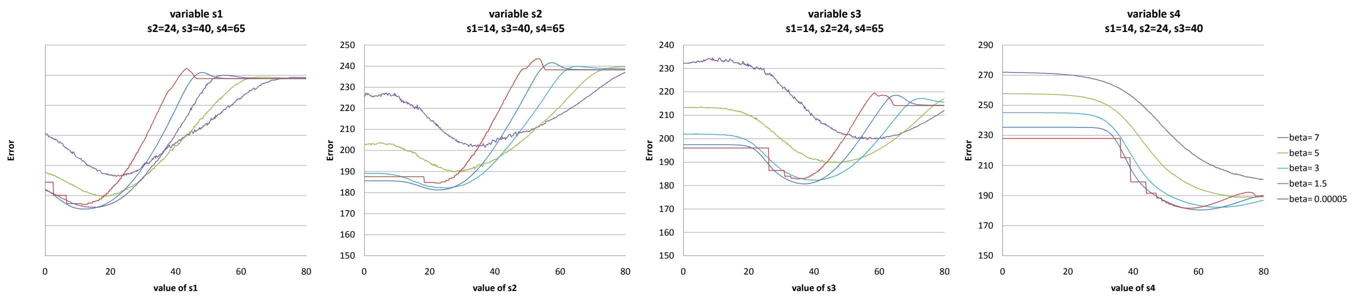
A final observation from these plots is that while smoothing eliminates discontinuities and plateaus, the smoothed function can still have sub-optimal local minima. Additionally, smoothing can change the position of the actual minima, so the minima of the smoothed function may be different from the minima of the original function. This explains why in Figure 8 with starting value 30, the unsmoothed function produced a better solution than the smoothed version. The solution to this problem is to start with a very smooth function and then progressively reduce the degree of smoothing so that the most accurate solution can be found.

5.2 PID Controller with a brake

In this problem, we want to synthesize parameters for a Proportional-Integral-Derivative (PID) controller for a wheel. To make the problem more interesting, we have considered a version of the problem where the wheel has brakes. The brake allows for a much finer control of the motion, and consequently a more effective controller. At the same time, they lead to a significant increase in the size of the parameter space and the number of control-flow paths to be considered, making parameter synthesis much more difficult.

The code for the benchmark, obtained by combining the dynamics of the wheel with the code for the controller, is shown in Figure 9. The parameters we want to synthesize are s_1, s_2, and s_3 (the coefficients of the proportional, derivative, and integral components of the controller), and b_1-b_8 (parameters in the Boolean expression deciding when to apply the brake). Known constants include $\mathtt{dt} = 0.1$, $\mathtt{target} = \pi$, $\mathtt{inertia} = 10.0$, and $\mathtt{decay} = 0.9$.

As for the specification, we want to ensure that after exactly T seconds, the wheel reaches its target position of $\mathtt{target} = \pi$. The



(a) (b) (c) (d)

Figure 7. *Error* in the gearbox benchmark as a function of (a) s_1, (b) s_2, (c) s_3, (d) s_4

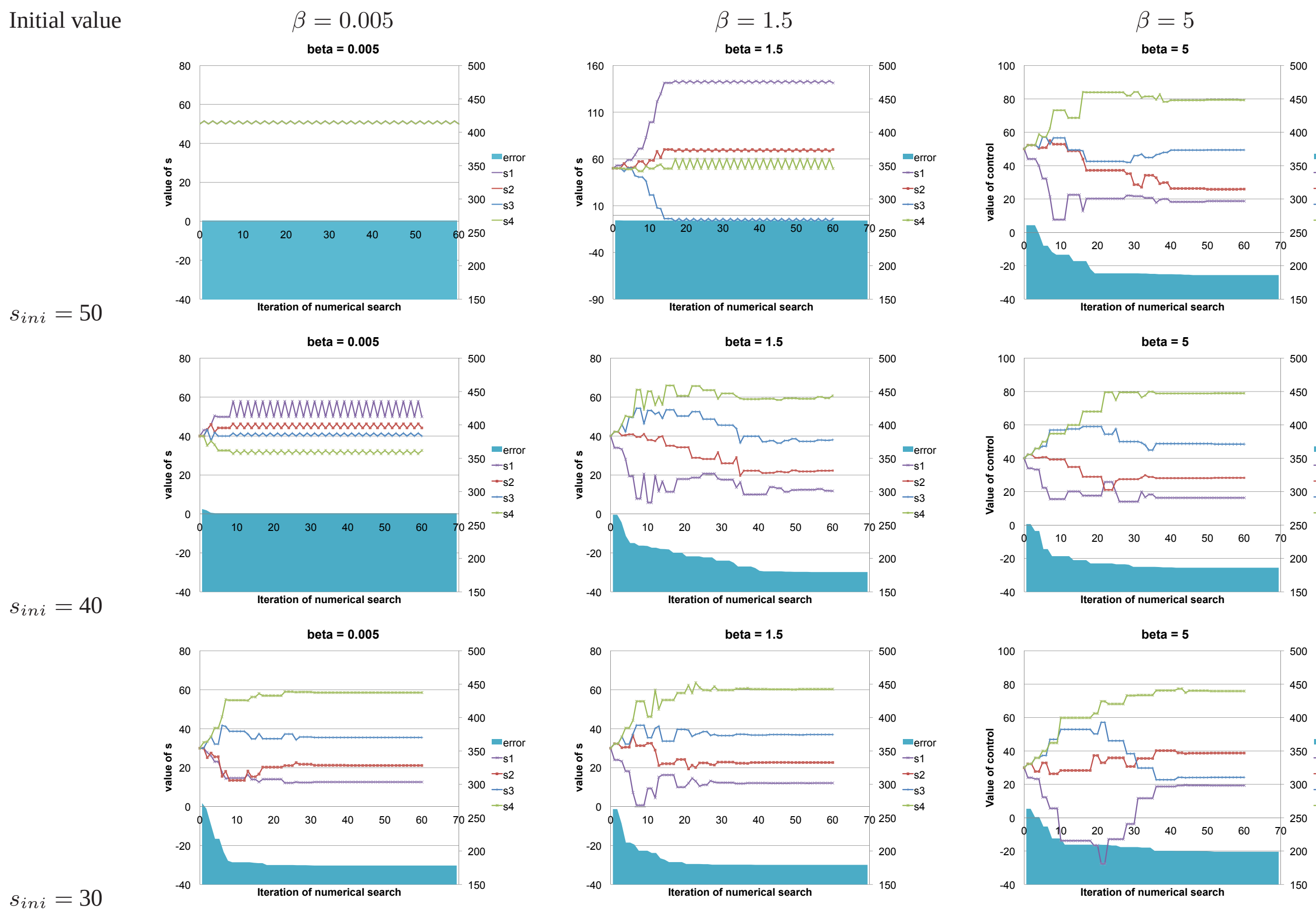


Figure 8. Effect of Nelder-Mead search on the control parameters of gearbox

precise error function is defined as follows:

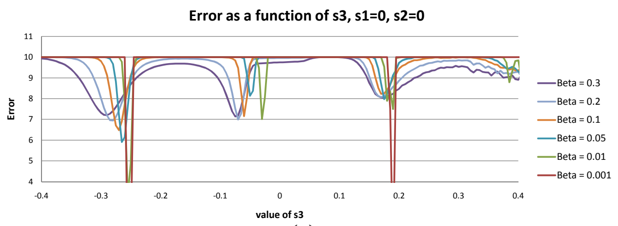
$$Error(s_1, s_2, s_3, b_1, \ldots, b_8) =$$
$$\quad \texttt{if}\ (\texttt{target} - \epsilon < \texttt{ang} < \texttt{target} + \epsilon)\ \texttt{then}\ 0\ \texttt{else}\ 10,$$

where $\epsilon = 0.00001$ for all our experiments. A notion of trajectories corresponding to this error function is easily defined—we skip the details.

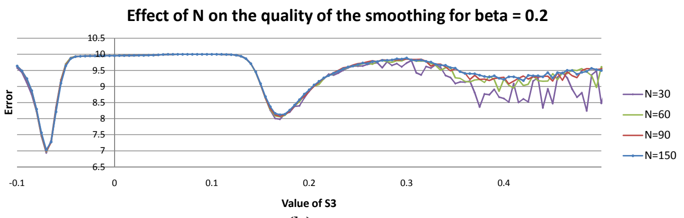
This function *Error* is highly discontinuous. Within a small subset of the input space, it evaluates to 0—everywhere else, it evaluates to 10. Such functions lead to the worst-case scenario for a Nelder-Mead search algorithm as the latter is stuck in a plateau. On the other hand, smooth execution really shines in this is example. Smoothing creates a gradient that Nelder-Mead search can follow, and allows the algorithm to find optimal parameters from a variety of starting points.

To illustrate the effect of smoothing, consider Figure 10(a), which shows *Error* as a function of s_3 with all other controls held constant. Once again, the effect of smoothing is dramatic. When β is very small, the error function only has two deep groves, and it is zero everywhere else. As we increase the value of β, the deep groves turn into gentle slopes.

Two interesting features are worth pointing out in Figure 10(a). First, as in the previous benchmark, smoothing has the effect of shifting the minima slightly. In fact, in this case, we see a new local minimum appear around -0.1. If this appears strange, remember that the plot is only showing a slice of a multi-dimensional plot.



(a)



(b)

Figure 10. (a) Error function for PID controller as a function of s_3 for different values of β. (b) Error function as a function of s_3 for different extents of path-sensitivity (N is the maximum number of components per symbolic state)

```
while(t < T) {
  if($b_1$ * d + $b_2$ > 0 and $b_3$ * v + $b_4$ > 0 or
     $b_5$ * d + $b_6$ > 0 and $b_7$ * v + $b_8$ > 0) {
        if(v > 0)
            brakev := -1;
        else  brakev := 1;
  }
  else brakev := 0;
  d := dist(ang, target);
  torq := $s_0$ * d + $s_1$ * v + $s_2$ * id + brakev;
  id := id * decay + d; // id: integral of distance
  oldv := v;
 // velocity v: derivative of distance
  v  := v + (torq / inertia) * dt;
  ang := ang + ( v + oldv)/2 * dt;
  if (ang > 2 * $\pi$)
    ang := ang - 2 * $\pi$;
  else if (ang < 0)
    ang := ang + 2 * $\pi$;
}
```

Figure 9. PID controller with a brake

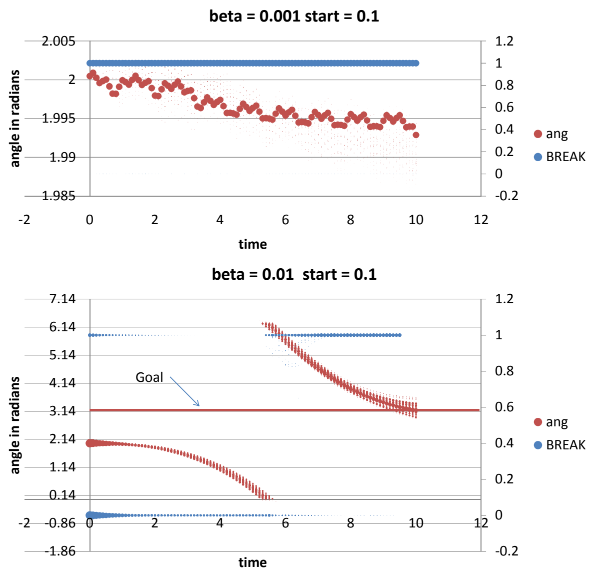


Figure 11. Execution of the PID controller with the best parameters found with β=0.01 and 0.0001

What is happening is that smoothing is allowing us to observe a local minimum in a different dimension.

The second feature is the "noise" to the right of the plot, particularly for the most-smoothed curve ($\beta = 0.3$). This noise, it turns out, is an artifact of the approximation that we make with the $Restrict$ operation to limit the number of components in our representation of the distribution (see Section 3). These plots were all generated by setting the maximum number of components of a distribution to $N = 90$. As we mentioned in Section 3, this $Restrict$ operation is the only approximation we make that is capable of introducing discontinuities.

In the plot in Figure 10, we can observe the effect of setting N to higher or lower values. In this plot, N is the number of states maintained by the analysis. Notice that in most of the graph, the value of N doesn't really matter; the plot is very smooth regardless. When s_3 starts becoming more positive, however, the system starts to become unstable, so path sensitivity begins to matter more and more. Notice that for $N = 30$, the loss of information due to merging of paths causes major discontinuities in the function. Increasing N up to 150 virtually eliminates these effects, since now the system can maintain enough path information to avoid building up approximation errors. Of course, higher values of N come at the cost of scalability.

Finally, Figure 11 shows the effect that smoothing has on our final goal of synthesizing parameters. The plots show the behavior of the solution found by Nelder-Mead search after 90 iterations, with and without smoothing. The solution found without smoothing is essentially useless; in this solution, the wheel starts with an angle of 2 radians, and drifts slowly without much direction. The blue line in the chart shows the brake, which in this solution is engaged the whole time. In other words, without smoothing, Nelder-Mead search was completely unable to find an acceptable solution. By contrast, when we applied smoothing with $\beta = 0.01$ (which is actually fairly large, given that the unit here is a radian), the system was able to find a very good solution. The wheel starts at two radians, and rotates in a very controlled manner towards its goal of π radians. Also, the brake (blue line) is mostly disengaged until around 6 time units, when the brake suddenly becomes engaged, helping the wheel reach the desired position exactly at the desired time.

Overall, we can see that for these benchmarks, discontinuities are a real obstacle to solving the parameter synthesis problem with numerical optimization techniques. Smoothing is able to eliminate these discontinuities, making parameter synthesis possible. Additionally, the most significant effect of our approximations are the small discontinuities introduced by the $Restrict$ operation.

6. Related work

So far as we know, the present paper is the first to develop a notion of smoothing for programs. While the idea of modeling software by smooth mathematical functions was previously considered by DeMillo and Lipton [6] in a brief note, their only technical result was a simple proof that every discrete transition system can be captured by a smooth function. Neither did DeMillo and Lipton identify an application of this view of program semantics.

However, Gaussian smoothing is ubiquitous in signal and image processing [19], and smooth approximations of boolean functions is a well-studied topic in theoretical computer science [16]. The idea of using smooth approximations to improve the performance of numerical methods like gradient descent is well-known in the domain of neural networks [3]. The basic unit of a neural network is a perceptron which has several real-valued inputs, and outputs 1 if and only if a weighted sum of these inputs is above a certain threshold. In multilayer perceptrons, comparison of the weighted sum with a threshold is replaced with the application of a sigmoid function, making learning more efficient. At a high level, our strategy is similar, as we also replace conditionals with sigmoids. The difference is that in our setting, smoothing is tied to a specific probabilistic semantics of programs, and the extent of smoothing at different points in the program are globally related by this semantics.

As for smooth interpretation, it is related to a line of recent work on probabilistic abstract and operational semantics [11, 14, 18] that builds on classic work on abstract interpretation [5] and probabilistic semantics [13]. In particular, our work is related to Smith's work [20] on abstract interpretation using truncated normal distributions. There are several important differences between Smith's approach and ours—in particular, not being interested in verification of probabilistic safety properties, we do not use a collecting semantics or offer a notion of probabilistic soundness.

The problem of tuning real-valued system parameters is a classic problem in systems theory. In particular, the hybrid systems community has studied the problem [7, 9, 12] in the context of embedded control applications such as ours. In their approach to the problem, a cyber-physical system is typically modeled as a hybrid automaton [1]; analysis approaches include simulation, symbolic reachability analysis, and counterexample-guided abstraction-refinement. To the best of our knowledge, none of this prior work frames parameter synthesis as a problem in numerical optimization, or uses smoothing or path-insensitivity.

Related efforts on program synthesis includes the Sketch system for combinatorial program sketching [21, 22], the ALisp system [2], and the Autobayes system [8] for synthesis of Bayesian classifiers. Like our approach to parameter synthesis, these approaches aim to produce a program satisfying a specification given a partial program conveying the high-level insight of a solution. However, none of these systems use a notion of program approximation akin to Gaussian smoothing.

7. Conclusion

In this paper, we have introduced a notion of Gaussian smoothing of programs, and presented an implementation of the smoothing transform based on symbolic execution. Using the concrete problem of parameter synthesis and three embedded control applications, we have demonstrated that smoothing facilitates the use of numerical search techniques in the analysis of embedded control programs.

The development of probabilistic and smoothed semantics in this paper was fairly informal. We leave a rigorous study of the mathematical properties of smoothed semantics, as well as the benefits of program smoothing to numerical methods, for future work. A second thread of future work will study the interplay of program smoothing with static analysis. Approximations obtained from smoothing are more accurate if smoothing is applied only on regions of the input space where the program behaves in a discontinuous manner. It may be possible to use recent results on *continuity analysis* of programs [4] to statically identify these regions. Finally, we plan to expand our technique for parameter synthesis into a method that can synthesize discrete as well as real-valued program parameters and expressions. Such an approach will integrate the present approach with the sketching approach to program synthesis [21, 22].

References

[1] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theor. Comput. Sci.*, 138(1):3–34, 1995.

[2] D. Andre and S. Russell. State abstraction for programmable reinforcement learning agents. In *AAAI/IAAI*, pages 119–125, 2002.

[3] C. Bishop. *Neural Networks for Pattern Recognition*. 1995.

[4] S. Chaudhuri, S. Gulwani, and R. Lublinerman. Continuity analysis of programs. In *POPL*, 2010.

[5] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, 1977.

[6] R. DeMillo and R. Lipton. Defining software by continuous, smooth functions. *IEEE Transactions on Software Engineering*, 17(4):383–384, 1991.

[7] A. Donzé, B. Krogh, and A. Rajhans. Parameter synthesis for hybrid systems with an application to Simulink models. In *HSCC*, 2009.

[8] B. Fischer and J. Schumann. AutoBayes: A system for generating data analysis programs from statistical models. *Journal of Functional Programming*, 13(03):483–508, 2003.

[9] G. Frehse, S. Jha, and B. Krogh. A counterexample-guided approach to parameter synthesis for linear hybrid automata. In *HSCC*, pages 187–200, 2008.

[10] B. Gough. GNU Scientific Library Reference Manual. 2009.

[11] S. Gulwani and G. Necula. Discovering affine equalities using random interpretation. In *POPL*, 2003.

[12] T. Henzinger and H. Wong-Toi. Using HyTech to synthesize control parameters for a steam boiler. In *Formal Methods for Industrial Applications*, pages 265–282, 1995.

[13] D. Kozen. Semantics of probabilistic programs. *J. Comput. Syst. Sci.*, 22(3):328–350, 1981.

[14] D. Monniaux. Abstract interpretation of probabilistic semantics. In *SAS*, pages 322–339, 2000.

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

[16] N. Nisan and M. Szegedy. On the degree of Boolean functions as real polynomials. *Computational Complexity*, 4(4):301–313, 1994.

[17] D. Parnas. Software aspects of strategic defense systems. *Communications of the ACM*, 28(12):1326–1335, 1985.

[18] A. Di Pierro and H. Wiklicky. Probabilistic abstract interpretation and statistical testing. In *PAPM-PROBMIV*, pages 211–212, 2002.

[19] J. Russ. *The image processing handbook*. CRC Press, 2007.

[20] M. Smith. Probabilistic abstract interpretation of imperative programs using truncated normal distributions. *Electron. Notes Theor. Comput. Sci.*, 220(3):43–59, 2008.

[21] A. Solar-Lezama, R. M. Rabbah, R. Bodík, and K. Ebcioglu. Programming by sketching for bit-streaming programs. In *PLDI*, pages 281–294, 2005.

[22] A. Solar-Lezama, L. Tancau, R. Bodik, V. Saraswat, and S. Seshia. Combinatorial sketching for finite programs. In *ASPLOS '06*, 2006.