Optimization and Engineering (2021) 22:217–245
https://doi.org/10.1007/s11081-020-09507-w

RESEARCH ARTICLE

How to catch a lion in the desert: on the solution of the coverage directed generation (CDG) problem

Raviv Gal[1] · Eldad Haber[2] · Brian Irwin[2] · Bilal Saleh[1] · Avi Ziv[1]

Received: 13 September 2019 / Revised: 22 April 2020 / Accepted: 22 April 2020 /
Published online: 26 May 2020

Abstract
The testing and verification of a complex hardware or software system, such as modern integrated circuits found in everything from smartphones to servers, can be a difficult process. One of the most difficult and time-consuming tasks a verification team faces is reaching coverage closure, or hitting all events in the coverage space. *Coverage-directed-generation* (CDG), or the automatic generation of tests that can hit hard-to-hit coverage events, and thus provide coverage closure, holds the potential to save verification teams significant simulation resources and time. In this paper, we propose a new approach to the CDG problem by formulating the CDG problem as a noisy derivative free optimization problem. However, this formulation is complicated by the fact that derivatives of the objective function are unavailable, and the objective function evaluations are corrupted by noise. We solve this noisy optimization problem by utilizing techniques from direct optimization coupled with a robust noise estimator, and by leveraging techniques from inverse problems to estimate the gradient of the noisy objective function. We demonstrate the efficiency and reliability of this new approach through numerical experiments with a noised quadratic function and an abstract model of part of IBM's *NorthStar* processor, a superscalar in-order processor designed for servers.

Keywords Hardware verification · Coverage directed generation · Derivative free optimization · Statistical parameter estimation · Inverse problems

1 Introduction

Verification of a complex hardware or software system, such as modern *integrated circuits* (ICs), can be a challenge. In principle, one would like to test every state or event that the system can reach, and observe that the system functions as intended. However, for complex systems, this is impossible, as the number of possible states is

✉ Brian Irwin
birwin@eoas.ubc.ca

Extended author information available on the last page of the article

so large that it is impractical to test each state individually. To this end, it is common to define a large, but finite, random set of tests or *test instances*, also referred to as *test stimuli*, that are drawn from the distribution of all possible tests, and apply them to the *design-under-test* (DUT) to be tested.

This paper targets verification environments that utilize *biased random stimuli generators* to generate test stimuli. The stimuli generator uses *test templates* as its input. The test templates bias the test stimuli generation toward targeted areas and features of the verified design. A test template comprises a set of parameters, or directives, where each parameter is a set of weight-value pairs. We refer to the output stimuli of the random stimuli generator as a test instance.

Even with a smart choice of test stimuli, one may have great difficulty hitting a number of key events to be tested. These events are often referred to as *hard-to-hit* events. This is because the mapping from test parameters to events is unknown, and can be highly nontrivial. Therefore, one of the most difficult and time-consuming tasks a verification team faces is reaching coverage closure, or, in other words, hitting all coverage events, including hard-to-hit events. Understanding why certain events are difficult to hit, and how they can be hit, requires both verification expertise and a deep understanding of the design under test. Moreover, generating test instances that hit such events is often an iterative trial and error process that consumes significant simulation resources and verification team time. Therefore, it is desirable to have an automatic solution for improving the probability of hitting hard-to-hit events.

Coverage-directed-generation (CDG), or the automatic generation of test instances, is a concept that has long been on the wish list of verification teams, and the target of a vast amount of research. Many techniques have been proposed to tackle the CDG problem, ranging from formal methods, via AI algorithms, to data analytics and machine learning techniques (see Mishra and Dutt 2002; Nativ 2001; Fine and Ziv 2003 and references within). For microprocessors, the simplest of these techniques involves exciting all the functions described in the data-sheet (Mishra and Dutt 2002). More advanced techniques, such as those in Fine and Ziv (2003), use the response of the processor to inputs to build a model to help predict which inputs will improve the probability of hitting hard-to-hit events. Almost all these techniques employ some form of statistical sampling. However, these techniques did not mature to be widely used in industry for various reasons, including the scalability of the solution, difficulty in applying it, and the quality of the proposed solution. As a result, reaching coverage closure remains almost entirely a manual process.

The goal of this work is to propose a new approach for the solution of the CDG problem and increasing the probability of hitting hard-to-hit events. Finding how to hit a low probability event in a large space is sometimes humorously referred to as finding "how to catch a lion in the desert", originated by the seminal paper of Pétard (1938). We propose a method that solves the problem by minimizing a cost function that increases the probability of hitting the hard-to-hit event(s). We show that such an approach can lead to an efficient solution of the problem, especially if it is coupled with a robust and efficient optimization algorithm.

The rest of the paper is organized as follows. In Sect. 2, we give a mathematical background to the proposed approach. In Sect. 3, we discuss solution techniques for

the problem. These techniques are based on direct optimization methods coupled with a robust noise estimator. In Sect. 4, we describe the main experimental environment used to test the proposed approach. In Sect. 5, we perform a number of experiments that demonstrate the efficiency of our approach, and we summarize the paper in Sect. 6.

2 Mathematical background

Let us mathematically formalize the testing process. Throughout the rest of the paper in general, bold letters, such as $\mathbf{t}$, represent vectors. Non-bold letters, such as θ, represent scalar quantities. Subscripts represent elements of a vector, such as s_k, and superscripts are used to represent a single vector in a group of vectors, such as $\mathbf{d}^l$.

Let $\theta(\mathbf{t})$ denote a random variable, referred to as a *test instance* of *test template* $\mathbf{t}$, and representing a test to be run by the DUT. Using *directives*, the test template $\mathbf{t}$ can be represented as a vector $\mathbf{t} = [\mathbf{d}^1, \mathbf{d}^2, \ldots, \mathbf{d}^L]$ composed by concatenating L *directive weight vectors* $\mathbf{d}^l, l = 1, 2, \ldots, L$. As a result, $\mathbf{t}$ is M-dimensional where $dim(\mathbf{t}) = M = \sum_{l=1}^{L} dim(\mathbf{d}^l)$. The directive weight vectors $\mathbf{d}^l$ parametrize each directive, and each $\mathbf{d}^l$ is normalized to present a probability distribution. The space of all possible test templates is denoted by $\mathscr{T}$, and is also known as the *test templates skeleton*. It is important to note that, while each test instance $\theta(\mathbf{t})$ is random, the directives and the test templates are **not**. The directives and test templates are deterministic parameters that define the random space and control the distribution of the test instances.

In the testing and verification process, the main goal is to hit every event in the *coverage space* $\mathscr{C} = \{c_1, c_2, \ldots, c_K\}$, or space of all events. Given a test instance $\theta(\mathbf{t})$, chosen from a probability space defined by the vector $\mathbf{t} \in \mathscr{T}$, one runs a simulation to obtain a random vector

$$\mathbf{s}(\theta) = [s_1, s_2, \ldots, s_K], \quad s_k \in \{0, 1\} \quad \forall k, \quad k = 1, 2, \ldots, K$$

that is defined as a *hit coverage* vector. The entries of the hit coverage vector $\mathbf{s}$ are binary. If a particular event in the coverage space was hit by the specific test instance θ, the entry of the corresponding index in $\mathbf{s}$ is 1, and it is 0 otherwise.

Clearly, since the test instances are generated randomly in a manner dependent on the parameters of the test template $\mathbf{t}$, the vector $\mathbf{s}$ is also random and depends on $\mathbf{t}$. To this end, let

$$\mathbf{e}(\mathbf{t}) = \mathbb{E}_{\mathbf{s}}\left[\mathbf{s}(\theta(\mathbf{t}))\right] \tag{1}$$

be the expected value of the hit coverage vector $\mathbf{s}$, and let

$$\mathbf{e}_N(\mathbf{t}) = \frac{1}{N} \sum_{i=1}^{N} \mathbf{s}^i \tag{2}$$

be the empirical expectation of the hit coverage vector estimated using N test instances generated from test template $\mathbf{t}$. Note that while $s_k \in \{0, 1\}\, \forall k$, the vector $\mathbf{e} = [e_1, e_2, \ldots, e_K]$ and its empirical values are *real*. The k-th value in $\mathbf{e}(\mathbf{t})$, e_k, represents the probability of hitting the event c_k using a test instance θ generated according to the distribution defined by $\mathbf{t}$. To compute the empirical expectation $\mathbf{e}_N(\mathbf{t})$ of a hit coverage vector, given a test template $\mathbf{t}$, we can run N simulations, obtain $\mathbf{s}^i, i = 1, 2, \ldots, N$ hit coverage vectors, and average them. Clearly, such a process is computationally expensive, especially if we are to estimate $\mathbf{e}(\mathbf{t})$ accurately. To demonstrate the above definitions, let us consider the following simple, but concrete example.

Example 1: Testing the multiplication of two numbers Assume that we build a calculator that can compute the product of two numbers in the interval [0, 1]. In a test instance, we need to randomly pick two numbers within the interval and compute their product. In this case, we have $L = 2$ directive weight vectors, $\mathbf{d}^1$ and $\mathbf{d}^2$, that define how we choose each of the two numbers. For simplicity, in this case we assume that $\mathbf{d}^1 = \mathbf{d}^2 = \mathbf{d}$, and therefore the test template $\mathbf{t}$ is just the single directive weight vector $\mathbf{t} = \mathbf{d}$. Next, we choose the parametrization of the test template, which defines the numbers in the interval [0, 1]. For simplicity, we split the interval [0, 1] into M equally sized increments. Formally, we assume that $\mathbf{t} = [t_1, t_2, \ldots, t_M]$ are the probabilities of choosing a number in the interval $[0, 1/M], (1/M, 2/M], \ldots, (1 - 1/M, 1]$.

Recall the space $\mathscr{C}$ is the space of all events, or coverage space. Let us define $K = M$ different events that correspond to the k cases that the output of the multiplication falls into one of the intervals $[0, 1/K], (1/K, 2/K], \ldots, (1 - 1/K, 1]$. Now, consider choosing the probability density parameterized by $\mathbf{t}$. One tempting choice is to simply use the uniform distribution, setting $t_m = 1/M, m = 1, 2, \ldots, M$. Clearly, this choice leads to less than optimal sampling of the coverage space. For this case, it is easy see that

$$e_1 \gg e_K$$

If we further refine the intervals in the $\mathscr{T}$ and $\mathscr{C}$ spaces by letting $K \to \infty$, then the likelihood of hitting an event that is at the right edge (close to 1) will approach 0, and therefore using a uniform distribution may not lead to a complete sampling of the coverage space, and we may end up with some unhit events. Understanding this problem allows one to choose a sampling routine that gives a higher probability to numbers that are close to 1, and improve the probability of sampling the whole coverage space.

The above multiplication example can be clearly analyzed to obtain an optimal sampling scheme. However, in practice, this is very frequently not the case. The system under test may be highly nonlinear. In this case, one typically performs some probing of the space by randomly testing a number of sampling schemes, and then tries to improve the coverage and sample as intelligently as possible. However, as previously discussed, hitting a hard-to-hit event may be difficult and require manual and labor intensive processes. Our goal is to improve over such processes by *automatically* increasing the probability of hitting hard-to-hit events.

Obtaining $\mathbf{e}_N(\mathbf{t})$ from $\mathbf{t}$ is an unknown function that is dictated by the simulator, and can be written as

$$\mathbf{e}_N(\mathbf{t}) = \mathbf{e}(\mathbf{t}) + \boldsymbol{\omega}(\mathbf{t}) \tag{3}$$

Here, $\boldsymbol{\omega}(\mathbf{t})$ is a noise vector that depends on $\mathbf{t}$. This noise vector $\boldsymbol{\omega}$ gets a different value every time we compute $\mathbf{e}_N$, giving us a noisy realization of the expected value of the hit coverage vector.

Let us define the target event(s), $\mathbf{e}^{\text{tar}}(\mathbf{t}) = \mathbf{P}^\top \mathbf{e}(\mathbf{t})$. Depending on the specific problem, $\mathbf{e}^{\text{tar}}$ can be a vector or a number. For example, if we only want to hit the k-th event, we can define $\mathbf{P}^\top$ as the k-th row of the identity matrix. In some cases, we aim to increase the probability of hitting a group of hard-to-hit events, and in this case $\mathbf{P}^\top$ corresponds to a few rows of the identity matrix. Maximizing the probability of hitting the events in $\mathbf{e}^{\text{tar}}$ can now be formulated as the simple optimization problem

$$\max_{\mathbf{t}} \left\{ \phi(\mathbf{t}) = \mathbf{1}^\top \mathbf{e}^{\text{tar}}(\mathbf{t}) = \mathbb{E}_{\mathbf{s}}\left[\mathbf{1}^\top \mathbf{P}^\top \mathbf{s}(\theta(\mathbf{t}))\right] \right\} \tag{4}$$

where $\mathbf{1}$ is a vector of all ones.

There are a number of problems when attempting to solve the maximization problem defined by Eq. 4. First, we do not have access to the objective function directly. The objective function can only be evaluated up to some unknown noise. Second, this noise is not necessarily stationary. That is, every time the objective function is called, a different noise vector is generated, and, on top of that, the noise level $\|\boldsymbol{\omega}\|$ can be different for different values of $\mathbf{t}$. Third, for a fixed $\mathbf{t}$, the noise corrupting the measurement of $\mathbf{e}(\mathbf{t})$ is likely different for each entry. In other words, the noise level is likely different for each event e_k. Fourth, a critical difference between this problem and the common problem of minimization under the expectation is that for the canonical stochastic programming problem, the random variable is drawn from a *fixed* distribution. Here, the distribution is parameterized by $\mathbf{t}$, and therefore, as we change the values of the parameters we optimize, we obtain a different distribution with a possibly different noise signature. To illustrate the above, we continue with our discussion of Example 1.

Example 1: Testing the multiplication of two numbers-continued We choose $M = K = 100$ segments, and choose the entries of $\mathbf{t}$ to grow quadratically in the interval [0, 1], and normalize such that they sum to 1. This implies that we have a higher probability of choosing larger numbers compared with smaller numbers. Given this test template, we compute the empirical hit coverage vector, $\mathbf{e}_N(\mathbf{t})$, for $N = 10^p, p = \{2, 3, 4, 5, 6\}$. The results are plotted in Fig. 1.

The results demonstrate how noisy the function can be when the number of realizations is small, and how the probability converges as the number of samples grows. Also note how low the probability of choosing numbers close to 1 is, even when $\mathbf{t}$ is chosen to grow quadratically. To find a $\mathbf{t}$ that further improves the probability of hitting the rightmost element in the empirical hit coverage vector $\mathbf{e}_N$, by setting $\mathbf{1}^\top \mathbf{P}^\top = [0, \ldots, 0, 1]$, one can compute an objective function that maximizes the probability of hitting the rightmost element.

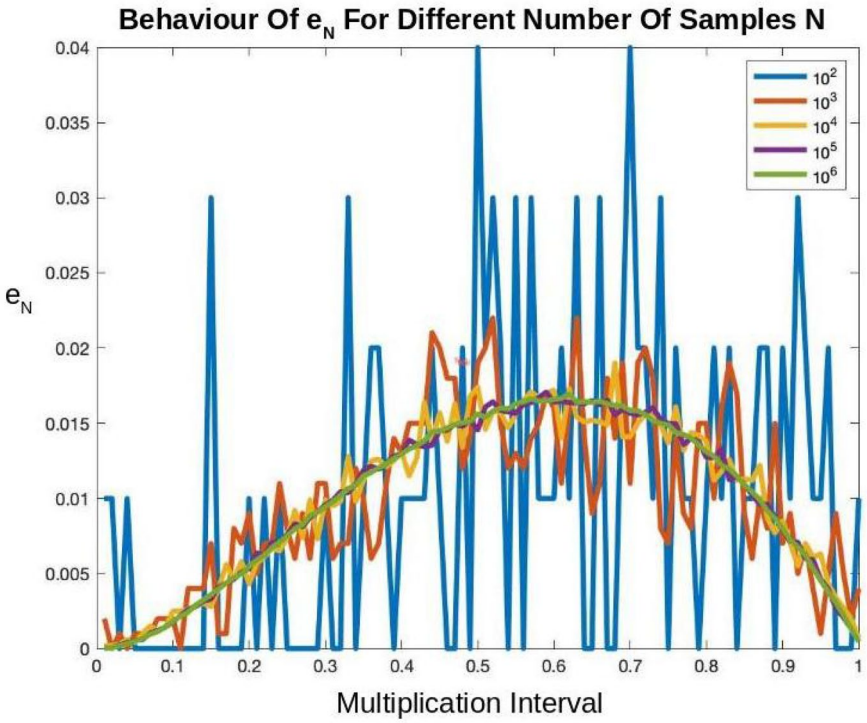

Fig. 1 Evaluation of $\mathbf{e}_N$ for the values $N = 10^p, p = \{2, 3, 4, 5, 6\}$ on the simple two number multiplication model problem. We choose $M = K = 100$ segments, and choose the elements of $\mathbf{t}$ to grow quadratically in the interval [0, 1], and normalize such that they sum to 1. Note how noisy the function can be when the number of realizations is small, and how the probability converges as the number of samples grows. Also note the low probability of hitting events at either end (close to 0 or 1) of the multiplication interval

3 Solution techniques

The main problem under consideration here, the CDG problem, can be formulated as a derivative free optimization (DFO) problem where the objective function under consideration is noisy. The topic has been considered by many authors, such as Kelley, Nocedal, and Scheinberg, using different techniques, ranging from stochastic methods (Conn et al. 2009), direct search methods (Kelley 2011), and gradient based methods (Berahas et al. 2019). In this paper, we experiment with three optimization techniques: an implicit filtering based technique, a steepest descent based technique, and a Broyden–Fletcher–Goldfarb–Shanno (BFGS) based technique. While we have directly used an implicit filtering technique, we have modified existing steepest descent and BFGS techniques from non-noisy unconstrained optimization in order to better deal with the noise in the problem. Below, we describe the algorithmic framework we used to solve the problem.

3.1 Optimization problem setup

Given an objective function $f(\mathbf{t})$, we decompose it as

$$f(\mathbf{t}) = \phi(\mathbf{t}) + \omega(\mathbf{t}) \tag{5}$$

We assume that $\phi(\mathbf{t})$ is a smooth function, and that $\omega(\mathbf{t})$ is noise. The noise ω is assumed to be uncorrelated with zero mean, and some unknown standard deviation σ. We assume that the standard deviation $\sigma(\mathbf{t})$ changes slowly with respect to $\mathbf{t}$.

For the CDG problem, we are unable to obtain the derivatives of ϕ with respect to $\mathbf{t}$, and therefore we turn to DFO methods. While there are many DFO methods, we turn our attention to local methods that are based on the numerical estimation of the gradient. Such methods have been studied extensively in the last 30 years (Rios and Sahinidis 2013), yielding successful software packages such as MCS, TOMLAB/LGO, and NEWUOA (Huyer and Neumaier 1999; Pintér 1996; Powell 2006) (also see references within).

However, when experimenting with the problem, we found that standard approaches based on gradient estimation methods fail or work poorly when the noise level is high. To explain this problematic observation, we first review the standard approach to such problems. A typical algorithm for such problems is composed of the following steps.

1. Evaluate the function f, its gradient ∇f, and its approximate Hessian $\mathbf{B} \approx \nabla^2 f$.
2. Compute a descent direction $\mathbf{z}$.
3. Update the solution using some relaxed line search or trust region method.

Function and gradient evaluations are typically done using finite differences. Let us review the process at some depth. Assume that we would like to compute the directional derivative of $f(\mathbf{t})$ in the direction $\mathbf{v}$. It is common to use a central finite difference approach computing

$$\mathbf{v}^\top \nabla f(\mathbf{t}) \approx \frac{f(\mathbf{t} + h\mathbf{v}) - f(\mathbf{t} - h\mathbf{v})}{2h} \tag{6}$$

which gives

$$\mathbf{v}^\top \nabla f(\mathbf{t}) \approx \mathbf{v}^\top \nabla \phi(\mathbf{t}) + \frac{\bar{\omega}}{2h} + h^2 N_{res}(\phi(\mathbf{t}), \mathbf{v}) \tag{7}$$

where $\bar{\omega}$ is a random variable generated by combining the zero mean errors in the function evaluations, and $N_{res}(\phi(\mathbf{t}), \mathbf{v})$ is the nonlinear residual. It is evident that the approximation for $\mathbf{v}^\top \nabla f(\mathbf{t})$ is polluted with two types of errors. The first type of error, corresponding to the second term in Eq. 7, is the error due to the noisy estimation of the function, and the second type of error, corresponding to the third term in Eq. 7, is an error due to the nonlinearity of $\phi(\mathbf{t})$. Unfortunately, these error terms have contradicting behaviours. While the second term in Eq. 7 requires as large an h as possible to reduce the error, the third requires a small h to obtain the same goal.

In some cases, when the noise is small, and it is possible to obtain an estimate of the magnitude of the nonlinear residual, one can balance these terms, choosing

$$h = \left(\frac{\sigma}{2\bar{N}_{res}} \right)^{\frac{1}{3}}$$

where $\bar{N}_{res}$ is an estimate of the nonlinear residual N_{res}. This approximation can be used in order to obtain a reasonable estimate of the gradient. However, even with this choice of approximation, the estimate of the gradient may not be sufficiently accurate.

Furthermore, estimating the noise and the nonlinear errors can be computationally difficult, and require additional function evaluations using different sized stencils. Such work was proposed in Moré and Wild (2011). However, even with an optimal stencil size, the noise can still be significant (see, for example, Fig. 1). Indeed, even for the optimal h (assuming that both $\bar{N}_{res}$ and σ are known), the error corrupting $\mathbf{v}^\top \nabla f(\mathbf{t})$ scales as $\sigma^{\frac{2}{3}}$, which only marginally improves the problem presented by the noise for large values of σ.

In this work, we introduce a different approach to the optimization problem. Rather than estimating the noise by further function evaluations, we view the problem as a statistical inverse problem, where the solution has to be evaluated from noisy data. In the next subsection, we show how to use standard techniques from inverse problems to estimate the behaviour of the objective function f, and its gradient ∇f.

3.2 Function and gradient approximation as statistical parameter estimation

Let us provide a different interpretation of the process of evaluating the gradient of a noisy function. Let us consider a general linear model of the form

$$f(\mathbf{t} + h\mathbf{v}) = \bar{\phi}(\mathbf{t}) + h\mathbf{v}^\top \mathbf{g} + \omega(\mathbf{t}) \tag{8}$$

with $\|\mathbf{v}\| = 1$ and ω being noise. Here, $\mathbf{g}$ is an unknown vector that is to be computed from the values of the objective function in points around $\mathbf{t}$. Note that this linear approximation is **not** necessarily the Taylor expansion. It can be any linear model that approximates the function for a given step size h and direction $\mathbf{v}$. Clearly, for smooth functions, as $h \to 0$, the approximation converges to the Taylor expansion when no noise is present.

Now, assume that we have n directions, $\mathbf{v}^1, \ldots, \mathbf{v}^n$. Using these directions, we obtain the following set of n equations

$$\begin{pmatrix} f_1 \\ \vdots \\ f_n \end{pmatrix} = \begin{pmatrix} 1 & h\mathbf{v}^{1\top} \\ 1 & \cdots \\ 1 & h\mathbf{v}^{n\top} \end{pmatrix} \begin{pmatrix} \bar{\phi} \\ \mathbf{g} \end{pmatrix} + \begin{pmatrix} \omega_1 \\ \vdots \\ \omega_n \end{pmatrix} \tag{9}$$

which we rewrite as the simple linear system

$$\mathbf{f} = \mathbf{V}\hat{\mathbf{g}} + \boldsymbol{\omega} \tag{10}$$

where $\hat{\mathbf{g}} = [\bar{\phi}, \mathbf{g}^\top]^\top$. Estimating $\hat{\mathbf{g}}$ from the noisy data $\mathbf{f}$ is a corner stone of statistical inverse problems (Tenorio 2017). It is therefore straight forward to use inverse problems techniques for the estimation of the average function value, $\bar{\phi}$, and gradient $\mathbf{g}$.

As is pointed out in Eq. 7 for the case of finite differences, a smaller h does not necessarily give a better gradient estimate due to the error introduced by the noisy evaluation of the function, and a similar observation also applies in the general linear model case here. By comparing the general linear model and a Taylor expansion, we get

$$f(\mathbf{t} + h\mathbf{v}^i) = \bar{\phi}(\mathbf{t}) + h\mathbf{v}^{i\top}\mathbf{g} + \omega_i = \phi(\mathbf{t}) + h\mathbf{v}^{i\top}\nabla\phi(\mathbf{t}) + O(h^2)$$

and noting that if $\bar{\phi}(\mathbf{t}) \to \phi(\mathbf{t})$ as $h \to 0$, then $\mathbf{v}^{i\top}\mathbf{g} - \mathbf{v}^{i\top}\nabla\phi(\mathbf{t}) \propto \frac{\omega_i}{h}$, and a smaller h does not necessarily cause $\mathbf{g} \to \nabla\phi$. However, it is also important to note that $\mathbf{g}$ is a locally "averaged" approximation to the gradient, and is **not** intended to be a *pointwise* estimator of the gradient in general. In other words, $\mathbf{g}$ in the general linear model is intended to quantify how ϕ changes "on average" in a local region as we move away from the current point. The key observation is that shrinking h still has the potential to magnify the corrupting effect of noise when estimating $\mathbf{g}$. This effect is not present when there is no noise, and it should be somewhat unsurprising that this can happen, given that finite difference approaches can be written as special cases of Eq. 10. To see that finite difference approaches can be written as special cases of Eq. 10, observe that finite difference estimation using forward differences is the special case of the system in Eq. 10 that can be explicitly written as

$$\begin{pmatrix} f_1 \\ \vdots \\ f_n \end{pmatrix} = \begin{pmatrix} 1 & & \\ \vdots & & h\mathbf{I} \\ 1 & & \end{pmatrix} \begin{pmatrix} f_0 \\ \mathbf{g} \end{pmatrix} + \begin{pmatrix} \omega_1 \\ \vdots \\ \omega_n \end{pmatrix}$$

where $\mathbf{I}$ is the $n \times n$ identity matrix here, and f_0 is known. Also note that the finite difference approach requires $n = M$, whereas our general linear model approach generalizes to cases where $n \neq M$.

Proceeding, we further assume that we have some prior estimate of $\hat{\mathbf{g}}$, $\hat{\mathbf{g}}_0$. If no such estimate is available, then we can choose $\hat{\mathbf{g}}_0 = \mathbf{0}$. Such an estimate can be obtained if we know something about the function f, or if we computed $\hat{\mathbf{g}}$ at a nearby point. For example, if $\hat{\mathbf{g}}$ was computed during a previous iteration, we can use this value from the previous iteration as $\hat{\mathbf{g}}_0$. A new estimate of $\hat{\mathbf{g}}$ can be obtained by solving the following ridge regression type minimization problem

$$\min_{\hat{\mathbf{g}}} \left\{ \frac{1}{2}\|\mathbf{V}\hat{\mathbf{g}} - \mathbf{f}\|_2^2 + \frac{\alpha}{2}\|\hat{\mathbf{g}} - \hat{\mathbf{g}}_0\|_2^2 \right\} \tag{11}$$

Given the regularization parameter, α, the problem has the closed form solution

$$\hat{\mathbf{g}}_\alpha = (\mathbf{V}^\top\mathbf{V} + \alpha\mathbf{I})^{-1}(\mathbf{V}^\top\mathbf{f} + \alpha\hat{\mathbf{g}}_0) \tag{12}$$

The regularization parameter α is chosen based on the noise level. When the noise level is unknown, as in our problem, the Generalized Cross Validation (GCV)

method can be used to choose α, and obtain an unbiased estimate of the noise level (Golub et al. 1979). This is done by minimizing the GCV function for this problem

$$\mathrm{GCV}(\alpha) = \frac{\|(\mathbf{I} - \mathbf{A}(\alpha))(\mathbf{f} - \mathbf{V}\hat{\mathbf{g}}_0)\|_2^2}{[\mathrm{trace}(\mathbf{I} - \mathbf{A}(\alpha))]^2} \tag{13}$$

where

$$\mathbf{A}(\alpha) = \mathbf{V}(\mathbf{V}^\top\mathbf{V} + \alpha\mathbf{I})^{-1}\mathbf{V}^\top.$$

Minimizing the GCV function in Eq. 13 in 1D can be done using a bisection method (Burden and Faires 2010).

Regarding the choice of the directions $\mathbf{v}^i$, as long as the regularization parameter $\alpha > 0$, the minimizer of Eq. 11 is unique. As a result, one is able to estimate the gradient in the underdetermined case using our linear model approach, even in the extreme situation where $n = 1$. It is important to note, however, that without reuse of the previous gradient estimate $\mathbf{g}_0$, which is equivalent to setting $\alpha = 0$, the underdetermined system may have infinitely many solutions. Hence, it should be relatively easy to see that choosing a very small number of directions n could potentially be problematic because as $\alpha \to 0$, $(V^TV + \alpha I)$ approaches a singular matrix, and the new gradient estimates may behave increasingly erratically. This is not as much of an issue if α is not close to 0, and in the overdetermined case. In the overdetermined case V^TV is likely a full rank, positive definite matrix, and thus $(V^TV + \alpha I)$ is likely invertible even if $\alpha = 0$. Thus, a transition in behaviour happens approximately when V is square, and so once the number of directions n is approximately the number of variables M or greater, the general linear model should give a gradient estimate depending noticeably less on previous estimates.

Note that choosing the directions $\mathbf{v}^i$ randomly with $n \geq M$ leads to a set of directions that span the space with high probability, even though the random $\mathbf{v}^i$ are likely not orthogonal. A lack of orthogonality is not necessarily bad, as the original choice of coordinate system can be arbitrary, and is not necessarily a natural coordinate system for the problem. Using random directions can help alleviate any issues arising from the specific choice of coordinate system, as demonstrated by Implicit Filtering (Kelley 2011), Simultaneous Perturbation Stochastic Approximation (SPSA) (Spall 1987, 1992, 1998) and Random Directions Stochastic Approximation (RDSA) (Prashanth et al. 2017) type methods. As we will later see in the numerical experiments section, the values obtained using the above approach can provide a significant advantage compared to the simple finite difference approximations employed in classical noisy optimization approaches.

3.3 Solution algorithms

Below, we present our solution algorithms in pseudocode. Algorithm 1 outlines the gradient based steepest descent technique and the BFGS approximation. As a comparison to this approach, we use implicit filtering as presented in Kelley (2011), outlined in Algorithm 2. The implicit filtering algorithm does not require any gradient

approximations, and only relies on the noisy values of f itself. Our algorithm adapts standard descent methods for non-noisy problems by using the GCV estimated $\bar{\phi}$ and $\mathbf{g}$ in place of the values of f and ∇f at each point, and performs a simple line search procedure.

A few comments regarding GCV are in order. First, minimizing the GCV function is not, in general, a computationally cheap process. For the CDG problem, function evaluation is very expensive, and the number of variables does not exceed the few thousands. In this case, investing some work to obtain the best direction possible is justified. However, for problems where function evaluation is cheap, one may not find our approach attractive. Nonetheless, efficient ways to minimize the GCV function that use stochastic trace estimators can make the process of solving the problem relatively fast. Here we have used the technique proposed in Golub and von Matt (1997) to obtain the solution of the problem using Krylov space decomposition.

Algorithm 1 Gradient Based Descent Technique

procedure DESCENT TECHNIQUE
  % Initialize minimization algorithm
  $iter \leftarrow 0$ % Iteration count
  $\mu_{ls} \leftarrow 10$ % Initialize line search parameter
  $ls_{break} \leftarrow$ False % Line search break flag
  $\mathbf{t} \leftarrow \mathbf{t}_{init}$
  $\mathbf{t}_{opt} \leftarrow \mathbf{t}_{init}$
  $\bar{\phi}_{opt} \leftarrow \infty$
  Evaluate $f(\mathbf{t}+h\mathbf{v})$ in n random directions $\mathbf{v}$
  Estimate $\bar{\phi}$ and $\mathbf{g}$ by solving 12 using GCV
  Approximate the inverse Hessian $\mathbf{B}^{-1}$ (in the case of BFGS) or set $\mathbf{B}^{-1} = \mathbf{I}$
  while True **do**
    $iter \leftarrow iter + 1$
    $\mathbf{t}_{old} \leftarrow \mathbf{t}$
    $\bar{\phi}_{old} \leftarrow \bar{\phi}$
    $\mathbf{g}_{old} \leftarrow \mathbf{g}$
    $lsIter \leftarrow 1$ % Line search iteration count
    while True **do**
      $\mathbf{t} \leftarrow \mathbf{t}_{old} - \mu_{ls}\mathbf{B}^{-1}\mathbf{g}_{old}$
      Evaluate $f(\mathbf{t}+h\mathbf{v})$ in n directions $\mathbf{v}$
      Estimate $\bar{\phi}$, $\mathbf{g}$, and average noise level $||\omega||$
      if $\bar{\phi} < \bar{\phi}_{old} + 2||\omega||$ **then**
        if $\bar{\phi} < \bar{\phi}_{opt}$ **then**
          $\mathbf{t}_{opt} \leftarrow \mathbf{t}$
          $\bar{\phi}_{opt} \leftarrow \bar{\phi}$
          break
      $\mu_{ls} \leftarrow \mu_{ls}/2$ % Shrink line search parameter
      $lsIter \leftarrow lsIter + 1$
      if $lsIter >$ Max. # line search iterations **then**
        $ls_{break} \leftarrow$ True % Line search break
        break
    if $lsIter = 1$ **then**
      $\mu_{ls} \leftarrow 2\mu_{ls}$ % Expand line search parameter
    % Check algorithm termination conditions
    if $ls_{break} =$ True **then**
      break
    if $iter >$ Max. # iterations **then**
      break

Algorithm 2 Implicit Filtering Based Technique

```
procedure IMPLICIT FILTERING TECHNIQUE
    % Initialize minimization algorithm
    iter ← 0 % Iteration count
    h ← h_init % Stencil size
    t ← t_init
    t_opt ← t_init
    f_opt ← ∞
    while True do
        iter ← iter + 1
        Evaluate f(t + hv) in n random directions v
        Find the minimum of the f(t + hv) values f*
        Find the direction v* corresponding to f*
        t* = t + hv*
        if f* < f_opt then
            t = t*
            t_opt = t
            f_opt = f*
        else
            h = h/2 % Refine stencil size
        % Check algorithm termination conditions
        if iter > Max. # iterations then
            break
        if h < Minimum Stencil Size then
            break
```

Before presenting the results from numerical experiments using our approach, we first describe the system used for the main numerical experiments: an abstract model of part of IBM's NorthStar processor.

4 The northstar pipeline

As a lightweight experimental environment, we employ a high-level software model of the two arithmetic pipes of the NorthStar superscalar in-order processor and the dispatch unit, also used in Fine and Ziv (2003). The NorthStar processor, also known as the RS64-II or PowerPC A50, was released by IBM in the late 1990s, featuring a RISC instruction set architecture (Borkenhagen and Sorino 1999). The high-level software model consists of two main components. First, a biased random stimuli generator that generates programs, and second, a software simulator of the NorthStar processor's dispatch unit and two arithmetic pipes that executes the randomly generated programs.

The NorthStar has two pipes, one simple and one complex (see Fig. 2). Each of the pipes comprises three stages: data fetch, execution, and write-back. One of the pipes, the simple pipe, handles only simple instructions, such as *add*. The other pipe, the complex pipe, handles complex instructions, such as *mul*. The complex pipe can also handle simple instructions when the simple pipe is busy. The model supports five types of instructions: simple instructions *Sim*, three types of complex

instructions Cm_1, Cm_2, Cm_3 that differ in the time they spend in the execution stage (1, 2, and 3 cycles respectively), and *Nop*, which represents all instructions that are not executed in the arithmetic pipes. The actual execution time can be longer due to data dependencies between instructions. To maintain simplicity, we assume the processor has only eight registers and instructions use one source and one target register. In addition, the processor has a condition register *CR*, which some instructions read from and write to. In each cycle, up to two instructions are fetched, according to the instruction's type and the state of the pipes.

Test templates $\mathbf{t}$ for the NorthStar software model are defined by four directive weight vectors $\mathbf{t} = [\mathbf{d}^1, \mathbf{d}^2, \mathbf{d}^3, \mathbf{d}^4]$, and control the distribution that the biased random stimuli generator component generates random programs from. Each directive weight vector defines a probability distribution. The first directive weight vector $\mathbf{d}^1 = IW = [W_{Nop}, W_{Sim}, W_{Cm_1}, W_{Cm_2}, W_{Cm_3}]$ contains instruction set selection weights, and controls the mnemonic of the generated instructions. The second and third directives affect the behaviour of the source and target registers. The second directive weight vector $\mathbf{d}^2 = SW = [W_{S_0}, \ldots, W_{S_7}]$ contains source register weights, and the third directive weight vector $\mathbf{d}^3 = TW = [W_{T_0}, \ldots, W_{T_7}]$ contains target register weights. The fourth directive weight vector $\mathbf{d}^4 = CW = [W_{C_0}, W_{C_1}]$ controls the conditional register. Thus, one can express a test template as the $M = 23$ entry vector $\mathbf{t} = [IW, SW, TW, CW]$.

The coverage space $\mathscr{C}$ is a cross-product (Piziali 2004) of the instructions in stage 0 of the complex and simple pipes (5 and 2 possible values respectively), two indicators for whether stage 1 of each pipe is occupied, and an indicator for whether the instruction in S_1 is using the conditional register. An event is defined by assigning values to each coordinate. For example, the event $(C_2, Sim, 0, 0, 0)$ means that C_2 and *Sim* are hosted at stage 0 of the complex and simple pipes, stage 1 of both pipes is not occupied, and the conditional register is not used. Clearly, the size of the coverage space size is

$$K = |C_0 Inst \times S_0 Inst \times C_1 Used \times S_1 Used \times S_1 CR| = |5 \times 2 \times 2 \times 2 \times 2| = 80.$$

However, out of this space, only 54 events are legal. For example, the 8 events spanned by the subspace $(Sim, Nop, *, *, *)$, where * indicates a wildcard that can be any value, are illegal because if S_0 is free, then the simple instruction should have been fetched into the simple pipe. During simulation, coverage is tracked for a time interval of 100 cycles, starting at cycle 10. An event is considered hit by the test instance if it was hit at least once during this time interval.

5 Numerical experiments

In this section, we illustrate how our linear model based gradient estimation approach can outperform finite differences on a simple noisy function, and we compare the performance of the implicit filtering, steepest descent, and BFGS techniques numerically using the NorthStar pipeline simulator described above in Sect. 4.

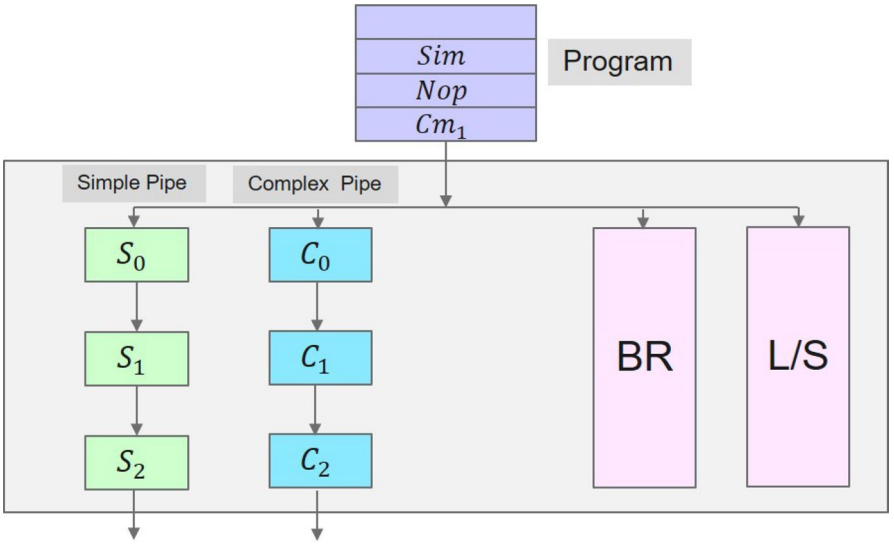

Fig. 2 Schematic of the simulated NorthStar pipeline. There are two pipes of 3 stages, one simple pipe *S* and one complex pipe *C*. In addition, L/S represents the processor's load store unit, and BR the branch prediction unit

5.1 Noisy 2-D quadratic function

To illustrate the performance of our general linear model based approach compared to finite differences in a noisy environment, we compared the performance of gradient descent using various gradient estimators on the 2-D quadratic function $f(x, y) = x^2 + y^2$ with standard normally distributed $\mathcal{N}(0, 1)$ random noise added. For simplicity, we used the constant step size $\mu = 0.001$, the initialization point of $(x_0, y_0) = (10, -10)$, and the stencil size $h = 0.001$.

Practically speaking, making h too small can cause the optimization procedure to become very sensitive to noise as well as roughness in the objective surface. As a result, a very small h can be undesirable. On the other hand, making h too big can make it difficult for the optimization procedure to locate sharp minima in the objective surface. Basically, the choice of h controls the level of smoothing, and a very small h likely does not smooth enough, while a very large h is likely to smooth too much. The choice of $h = 0.001$ is somewhat arbitrary, but it is meant to be an intermediate value that is not too large or too small.

Figure 3 compares the progress of our linear model approach with gradient descent using exact gradients for 1000 iterations. Similarly, Fig. 4 shows the evolution of the true value of $f(x, y)$ during gradient descent using forward finite differences and central finite differences to approximate the gradient.

We now make a few important observations. First, as expected, the behaviour of the finite difference based approximate gradients can be very erratic in the presence of noise. Figure 4 illustrates this (note the larger range of y-axis values compared to Fig. 3). Second, the linear model based approach exhibits an ability to handle

the noise in such a way that performance is not as severely degraded as in the finite difference case. Third, increasing the number of directions n appears to improve the performance of the linear model approach, which is also expected given the discussion in Sect. 3.2.

5.2 NorthStar initial exploration

Proceeding to the NorthStar environment, as an initial exploration of $\mathbf{e}(\mathbf{t})$ for the NorthStar, we first ran 5000 random test templates drawn from $\mathscr{T}$ according to the Dirichlet distribution $Dir(1)$. Using these 5000 test templates, we hit all events in the coverage space $\mathscr{C}$ at least once. We also found the hardest event to hit to be event $c_{hard} = (C_2, Nop, 0, 1, 0)$. The single best test template hit event c_{hard} with probability $p(c_{hard}) = 0.15$. Based on applied domain knowledge, it was deduced that the test template defined by $IW = (0.5, 0.2, 0, 0.3, 0)$, $SW = TW = (1, 0, 0, 0, 0, 0, 0, 0)$, and $CR = (1, 0)$ would yield the best chance of hitting event c_{hard}.[1] This test template

$$\mathbf{t} = [0.5, 0.2, 0, 0.3, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0]$$

will give high weights to *Nop*, *Sim*, and Cm_2, will create dependencies between the source and target registers, and will not use the condition register *CR*. Experimentally, by averaging over 100,000 runs of the simulator, this template was observed to yield a hit probability of $p(c_{hard}) = 0.4$. Below, we continue to use values averaged over 100,000 runs of the simulator to define a high quality estimate, or "true" value of $p(c_{hard})$. However, as we show below, both the implicit filtering and steepest descent based techniques are able to *automatically* discover test templates that achieve $p(c_{hard}) = 0.4$ or close to 0.4 with a modest budget of total runs of the NorthStar simulator.

5.3 Event c_{hard} objective function

As was the case when analyzing how to maximize the probability of hitting the rightmost element in the empirical hit coverage vector for the two number multiplication simulator in Sect. 2, we can again choose $\mathbf{P}^\top$ to be a single row of the identity matrix. Specifically, $\mathbf{P}^\top$ is now the row of the identity matrix corresponding to c_{hard}. To avoid explicity enforcing the constraint that the directive weight vectors *IW*, *SW*, *TW*, *CW* define properly normalized probability distributions, and thus solving a constrained optimization problem, we instead pass the values obtained from the optimization algorithms through the standard softmax function to ensure valid probability distributions before passing them to the program generator component of the NorthStar software model.

In Fig. 5, we plot the objective function for maximizing $p(c_{hard})$ sliced over two random directions $\mathbf{y}_1$ and $\mathbf{y}_2$, for $N = 10$ and $N = 1000$ simulator runs per point respectively. The uniform test template $\mathbf{t}_{uni}$, defined by

[1] There are many other templates with different values of *SW* and *TW* that achieve the same probability.

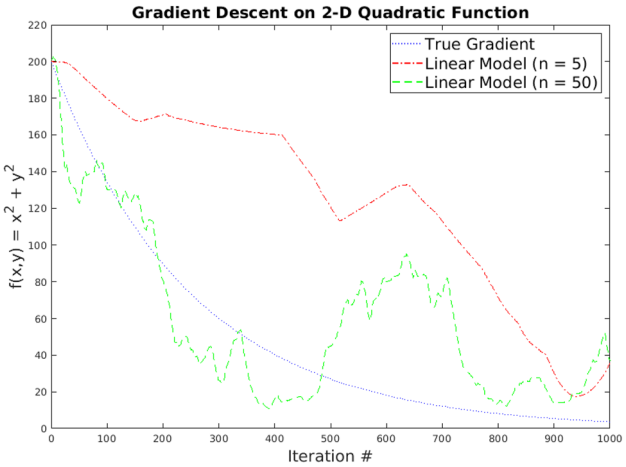

Fig. 3 Exact function values $f(x, y)$ from running gradient descent on the noisy quadratic function $f(x, y) + \mathcal{N}(0, 1)$ using a constant step size of $\mu = 0.001$ and our linear model approach to approximate the gradient. n directions of length $h = 0.001$ were sampled uniformly at random each iteration. The blue curve shows the performance of gradient descent using exact gradients in this situation. The initialization point for all runs was $(x_0, y_0) = (10, -10)$

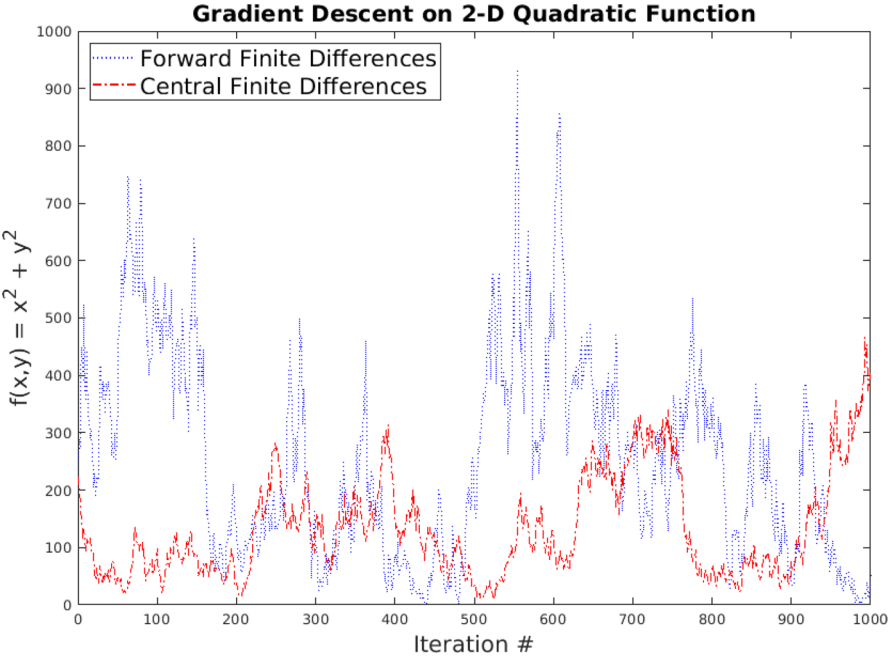

Fig. 4 Exact function values $f(x, y)$ from running gradient descent on the noisy quadratic function $f(x, y) + \mathcal{N}(0, 1)$ using a constant step size of $\mu = 0.001$ and finite differences with stencil size $h = 0.001$ to approximate the gradient. The initialization point for all runs was $(x_0, y_0) = (10, -10)$

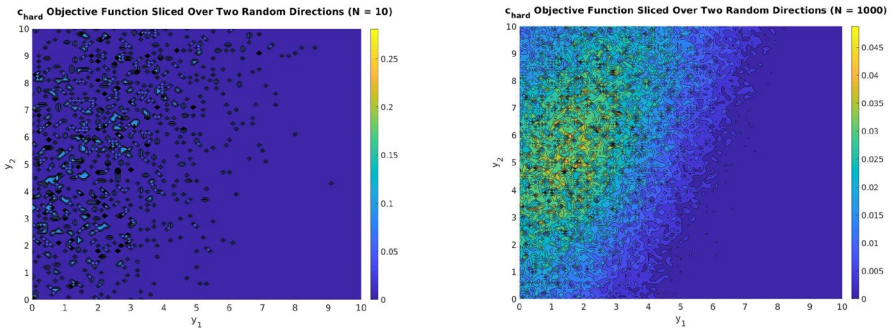

Fig. 5 The landscape of the objective function for maximizing the probability of hitting event c_{hard} computed over two random directions $\mathbf{y}_1$ and $\mathbf{y}_2$. Note the many local maxima, and the objective function's overall non-convexity. Also, note the confusing effects of noise, such as overestimating probabilities when the number of samples N is small

$$
\begin{aligned}
IW &= (0.2, 0.2, 0.2, 0.2, 0.2) \\
SW &= TW = (0.125, 0.125, 0.125, 0.125, 0.125, 0.125, 0.125, 0.125) \\
CW &= (0.5, 0.5)
\end{aligned}
$$

defines the origin in Fig. 5, and is the starting point for all the optimization experiments in the following subsections. Once again, evaluating the objective function first consists of passing a vector in $\mathbb{R}^{23}$ through standard softmax functions over the 1st–5th, 6th–13th, 14th–21st, and 22nd–23rd components. After using the standard softmax function to ensure the 4 directive weight vectors define valid probability distributions, we pass the template defined by these 4 directive weight vectors to the biased random stimuli generator, and track the coverage of the generated random programs over 100 cycles. Note the many local minima, the objective function's overall non-convexity, and how increasing the number of simulator runs per point does not substantially alleviate the non-convexity.

5.4 Implicit filtering technique

We use implicit filtering to maximize $p(c_{hard})$, which is equivalent to minimizing $-p(c_{hard})$. Table 1 shows the results of a typical successful run of the implicit filtering algorithm. N denotes the number of simulator runs we use to estimate $\mathbf{e}_N(\mathbf{t})$ at each template, and n is the number of random directions $\mathbf{v}$. The algorithm was set to terminate after 50 iterations, or the stencil size h decreased below 1e−3. Overall, with a modest budget of 15,000 total simulations,[2] we are able to automatically get within 0.01 of the best hit probability of $p(c_{hard}) = 0.4$. Additionally, we also present the value of $\bar{\phi}$ estimated by fitting the regularized linear model defined by Eq. 11 at each iteration. Figure 6 compares the behaviour of f^*, $\bar{\phi}$, and the "true"

[2] $(24 \text{ Iterations}) \times \left(25 \frac{\text{Points}}{\text{Iteration}}\right) \times \left(25 \frac{\text{Simulations}}{\text{Point}}\right) = 15{,}000 \text{ Simulations}.$

Table 1 Summary of a successful run of the implicit filtering algorithm

I	f^*	$\bar{\phi}$	Update $\mathbf{t}_{opt}$?	h	$p(c_{hard})$
Implicit filtering history ($N = 25$, $n = 25$, $h_{init} = 50$)					
1	0	0	True	50	0.016
2	0	0	False	50	0
3	0.160	7.15e−5	True	25	0
4	0.080	8.62e−5	False	25	0.099
5	0.160	0.043	False	12.5	0.099
6	0.320	0.032	True	6.25	0.101
7	0.400	0.063	True	6.25	0.142
8	0.280	0.066	False	6.25	0.339
9	0.440	0.245	True	3.125	0.343
10	0.520	0.227	True	3.125	0.325
11	0.600	0.814	True	3.125	0.341
12	0.440	0.264	False	3.125	0.383
13	0.520	0.297	False	1.5625	0.385
14	0.560	0.270	False	7.8125e−1	0.382
15	0.640	0.350	True	3.90625e−1	0.388
16	0.520	0.390	False	3.90625e−1	0.363
17	0.560	0.346	False	1.953125e−1	0.364
18	0.480	0.346	False	9.765625e−2	0.363
19	0.520	0.346	False	4.8828125e−2	0.360
20	0.560	0.370	False	2.44140625e−2	0.365
21	0.520	0.335	False	1.220703125e−2	0.365
22	0.560	0.353	False	6.103515625e−3	0.363
23	0.520	0.353	False	3.0517578125e−3	0.364
24	0.480	0.353	False	1.52587890625e−3	0.362

Table 1 (continued)

Summary of final results
Total # of Simulations = 15,000
$IW_{opt} = [0.5790, 0.2010, 0, 0.2151, 0.0049]$
$SW_{opt} = [0, 1, 0, 0, 0, 0, 0, 0]$
$TW_{opt} = [1, 0, 0, 0, 0, 0, 0, 0]$
$CW_{opt} = [1, 0]$
$f_{opt} = 0.64$, $\bar{\phi}_{opt} = 0.35$, $p_{opt}(c_{hard}) = 0.39$

The algorithm is able to *automatically* get within 0.01 of the best c_{hard} hit probability of 0.4 using a modest budget of 15,000 total simulations. The optimization was initialized at the uniform template $\mathbf{t}_{uni}$, and was set to terminate after 50 iterations were exceeded, or the stencil size shrank below 1e−3. I is the iteration number

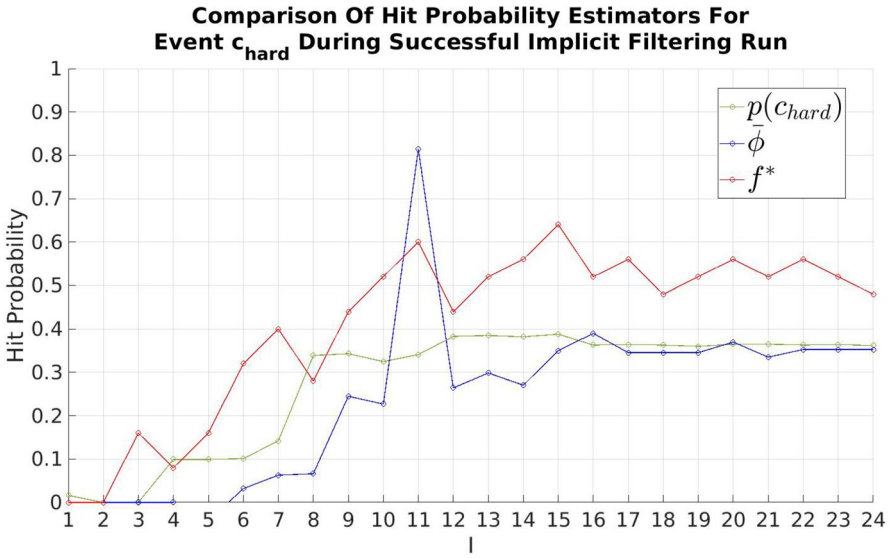

Fig. 6 Visualizing the evolution of the hit probability estimators from Table 1 during the successful implicit filtering run. $p(c_{hard})$ denotes the reference "true" value, which is calculated by averaging over 100,000 runs of the simulator at a given template $\mathbf{t}$. The x-axis I is the iteration number. Note how the implicit filtering estimate f^* consistently overestimates the probability, whereas $\bar{\phi}$ generally underestimates the probability, but then converges to the "true" value as the algorithm progresses

value of $p(c_{hard})$ averaged over 100,000 simulator runs. It is worth noting that the fitted $\bar{\phi}$ value appears to be a better estimator of $p(c_{hard})$ than the values of f^*, which is not unexpected given it incorporates information from more runs of the simulator, and nearby points.

Table 1 only shows a typical successful run of implicit filtering. The algorithm can also unsuccessfully terminate yielding templates achieving $p_{opt}(c_{hard}) = 0$. As a result, we experiment with the expected performance of the algorithm for different parameter values in Table 2. We also investigate the tradeoff between the number of samples N used to estimate $\mathbf{e}_N(\mathbf{t})$ at each template, and the number of random directions n, at each iteration. For each set of N and n values, we ensemble results over 25 independent runs, and, as in Table 1, $h_{init} = 50$, and the algorithm was set to terminate after 50 iterations, or the stencil size decreased below 1e−3.

Overall, we see that the implicit filtering technique is not always very reliable. This is exemplified by the "Failures" column in Table 2, which shows that even for relatively large per iteration budgets, the algorithm can still fail to *ever* hit the event c_{hard}. Specifically, a failure is defined as the algorithm terminating at a test template that has a $p_{opt}(c_{hard}) = 0$, or in other words, even after averaging over 100,000 simulations at that template, event c_{hard} was *never* hit. As expected, Table 2 shows

Table 2 Results of the implicit filtering based optimization technique compared over fixed per iteration budgets

N	n	$\bar{I}$	$s^2[I]$	$\overline{f_{opt}}$	$s^2[f_{opt}]$	$\overline{p_{opt}(c_{hard})}$	$s^2[p_{opt}(c_{hard})]$	max $\{p_{opt}(c_{hard})\}$	Failures
Per iteration budget = 100									
5	20	17.2	0.4	0.040	0.027	0.015	0.005	0.367	$\frac{23}{25}$
10	10	17.2	0.8	0.048	0.028	0.022	0.006	0.335	$\frac{23}{25}$
20	5	17.4	1.8	0.062	0.031	0.033	0.009	0.352	$\frac{22}{25}$
Per iteration budget = 625									
5	125	18.3	3.8	0.344	0.225	0.104	0.022	0.365	$\frac{16}{25}$
25	25	18.5	5.7	0.192	0.083	0.104	0.025	0.377	$\frac{17}{25}$
125	5	17.3	2.0	0.017	0.007	0.014	0.005	0.337	$\frac{24}{25}$
Per iteration budget = 1250									
5	250	18.1	2.7	0.320	0.227	0.105	0.025	0.375	$\frac{17}{25}$
10	125	19.4	5.1	0.444	0.164	0.183	0.029	0.396	$\frac{11}{25}$
25	50	19.5	10.4	0.259	0.106	0.143	0.032	0.394	$\frac{15}{25}$
50	25	17.6	2.8	0.066	0.033	0.041	0.013	0.388	$\frac{22}{25}$
125	10	17.7	4.5	0.058	0.026	0.044	0.015	0.394	$\frac{22}{25}$
250	5	17.4	3.2	0.014	0.005	0.012	0.004	0.299	$\frac{24}{25}$
Per iteration budget = 2500									
5	500	18.5	2.0	0.600	0.250	0.207	0.031	0.395	$\frac{10}{25}$
10	250	20.0	5.3	0.576	0.166	0.230	0.027	0.390	$\frac{8}{25}$
25	100	20.5	11.4	0.379	0.119	0.201	0.034	0.390	$\frac{11}{25}$
50	50	18.9	10.0	0.154	0.064	0.101	0.027	0.376	$\frac{18}{25}$
100	25	18.5	9.8	0.102	0.043	0.073	0.022	0.386	$\frac{20}{25}$
250	10	17.6	3.9	0.036	0.016	0.031	0.011	0.390	$\frac{23}{25}$
500	5	17.4	2.3	0.035	0.014	0.029	0.010	0.375	$\frac{23}{25}$
Per iteration budget = 5000									
5	1000	19.7	1.1	0.912	0.077	0.281	0.010	0.389	$\frac{2}{25}$
10	500	20.8	4.0	0.748	0.113	0.285	0.018	0.395	$\frac{4}{25}$
25	200	20.8	5.8	0.539	0.080	0.276	0.021	0.393	$\frac{5}{25}$
50	100	21.3	12.5	0.404	0.081	0.246	0.030	0.389	$\frac{8}{25}$
100	50	20.7	13.7	0.294	0.072	0.204	0.034	0.392	$\frac{11}{25}$
200	25	19.0	12.8	0.131	0.046	0.103	0.029	0.394	$\frac{18}{25}$
500	10	18.6	11.5	0.084	0.030	0.074	0.023	0.397	$\frac{20}{25}$
1000	5	18.1	7.0	0.048	0.017	0.043	0.014	0.392	$\frac{21}{25}$

Statistics are calculated over 25 independent runs for each combination of n and N, where a bar represents the sample average, and $s^2[\cdot]$ represents the sample variance. N is equivalent to simulator runs, and n is the number of directions in which we choose new test templates $\mathbf{t}$. I is the number of iterations to termination, which occurs when the stencil size parameter reaches less than $h = 0.001$. A failure occurs when the algorithm terminates at a template with a "true" probability $p_{opt}(c_{hard}) = 0$

the chances of a failure happening are reduced when the per iteration budget is increased, and the number of random directions n is increased. As a general trend, trading off simulation runs N for random directions n, given a fixed per iteration budget, is beneficial for the performance of implicit filtering. Only for very small values of N, such as $N = 5$, does this trend appear to break down. As it is undesirable for how many different parameter choices the algorithm fails more than half the time. We now investigate if a gradient-based algorithm performs better overall.

5.5 Steepest descent technique

Now, we use Algorithm 1 to maximize $p(c_{hard})$, which is again equivalent to minimizing $-p(c_{hard})$. Table 3 shows the results of a typical successful run of the steepest descent algorithm. The algorithm was set to terminate after 50 iterations, or the line search break flag was set after 10 consecutive line search failures. The line search parameter μ_{ls} was initialized to 10. Overall, with a modest budget of 21,875 total simulations,[3] we are able to automatically get within 0.09 of the best hit probability of $p(c_{hard}) = 0.4$. Note that the term "total iterations" refers to all the iterations requiring computations, including the failed line searches. For example, iteration 12 in Table 3 contributed 4 total iterations, as iteration 12 required 4 line search iterations.

Figure 7 compares the behaviour of $\bar{\phi}$, and the "true" value of $p(c_{hard})$. In general, $\bar{\phi}$ tracks $p(c_{hard})$ closely, but with a tendency to vary more slowly. This is because of the averaging effect of the algorithm.

Similar to Table 2, Table 4 investigates the expected performance of Algorithm 1 for different parameter values. Like with Table 2, for each set of N and n values, we ensemble over 25 independent runs, and, as in Table 3, $h = 5$ and μ_{ls} is initialized to 10, and the algorithm was set to terminate after 50 iterations, or the line search break flag was set after 10 consecutive line search failures.

Overall, the gradient based steepest descent technique appears much more reliable than the implicit filtering technique. Algorithm 1 almost always terminates at a template that at least hits the event c_{hard} a minimum of once in 100,000 simulation runs. It is also worth noting that in most cases $\bar{\phi}$ underestimates $p(c_{hard})$, sometimes by a large margin of up to almost 0.2. The authors conjecture this may be due to a relatively large choice of h, which is not refined during Algorithm 1. The effects of a relatively large h should be especially pronounced if the optima are rather sharp, which given domain knowledge, is not unlikely for this problem. However, as with the implicit filtering technique, the steepest descent algorithm's performance strongly benefits from trading off N for n, given a fixed per iteration budget. For both algorithms, it appears that in general, coarsely sampling many points is preferable to sampling a few points with high accuracy at each point.

[3] $(35 \text{ Total Iterations}) \times (25 \frac{\text{Points}}{\text{Iteration}}) \times (25 \frac{\text{Simulations}}{\text{Point}}) = 21{,}875 \text{ Simulations}.$

Table 3 Summary of a successful run of the steepest descent algorithm (Algorithm 1)

I	$\bar{\phi}$	$\|\mathbf{g}\|$	$\|\omega\|$	μ_{ls}	*Update* $\mathbf{t}_{opt}$*?*	$p(c_{hard})$
Steepest descent history ($N = 25, n = 25, h = 5$)						
1.1	0.018	0.0358	1.24e−3	10	True	0.017
2.1	0.029	0.1149	1.75e−4	20	True	0.022
3.4	0.030	0.0483	2.22e−4	5	True	0.022
4.1	0.027	0.0336	3.37e−3	5	False	0.022
5.1	0.027	0.0336	7.73e−3	10	False	0.025
6.1	0.021	0.0231	5.10e−3	20	False	0.028
7.1	0.021	0.0218	7.31e−3	40	False	0.040
8.1	0.026	0.0249	8.59e−3	80	False	0.062
9.1	0.030	0.0221	1.01e−2	160	False	0.088
10.1	0.034	0.0452	3.49e−3	320	True	0.027
11.3	0.145	0.1747	4.68e−3	160	True	0.333
12.4	0.147	0.1313	3.49e−2	20	True	0.339
13.1	0.155	0.1118	1.70e−2	20	True	0.331
14.3	0.241	0.2345	2.20e−4	10	True	0.312
15.10	0.031	0.2225	5.89e−4	1.953125e−2	False	0.314

Summary of final results

Total # of Simulations = 21,875

$IW_{opt} = [0.4377, 0.1642, 0.2597, 0.1368, 0.0015]$

$SW_{opt} = [0, 0, 0, 0, 1, 0, 0, 0]$

$TW_{opt} = [0, 1, 0, 0, 0, 0, 0, 0]$

$CW_{opt} = [1, 0]$

$\bar{\phi}_{opt} = 0.24$, $p_{opt}(c_{hard}) = 0.31$

The algorithm is able to *automatically* get within 0.09 of the best c_{hard} hit probability of 0.4 using a modest budget of 21,875 total simulations. The optimization was initialized at the uniform template $\mathbf{t}_{uni}$, and was set to terminate after 50 main iterations were exceeded, or after 10 consecutive line search failures. I is the iteration number, formatted so the number after the decimal point represents the final line search iteration for the given main iteration

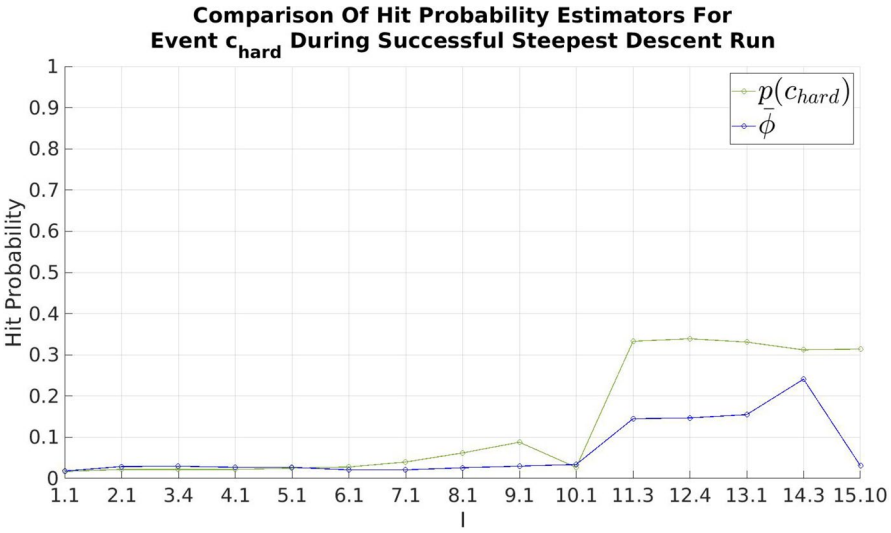

Fig. 7 Visualizing the evolution of the hit probability estimators from Table 3 during the successful run of the steepest descent algorithm (Algorithm 1). $p(c_{hard})$ denotes the reference "true" value, which is calculated by averaging over 100,000 runs of the simulator at a given template $\mathbf{t}$. The x-axis shows the iteration number, where the value after the decimal point is the final line search iteration number for the given main iteration. Note how $\bar{\phi}$ understimates the "true" probability, especially towards the end of the run

5.6 BFGS technique

Finally, following the framework of Algorithm 1, we use BFGS to minimize $-p(c_{hard})$. Specifically, we use a limited-memory implementation of the BFGS method, also referred to as L-BFGS. To compute the BFGS directions, we use the L-BFGS two-loop recursion detailed on p. 225 of Nocedal and Wright (1999). We set the initial inverse Hessian approximation to be a scaled version of the identity matrix, where the scaling factor is given by Eq. 9.6 on p. 226 of Nocedal and Wright (1999). As a result, a back tracking line search starting with $\mu_{ls} = 1$ at each iteration, and refining by a factor of two for each line search failure, was employed. However, we set the memory value, m, for our L-BFGS implementation to $m = 100$, which was almost always greater than the number of iterations before termination. As a result, almost all of the time our L-BFGS implementation was equivalent to a standard BFGS implementation.

Of the three algorithms we tested, the L-BFGS implementation had the most difficulty obtaining test templates achieving close to $p(c_{hard}) = 0.4$, and is not competitive with either implicit filtering or steepest descent. As with Tables 2 and 4 Table 5 investigates the expected performance of BFGS for different parameter values. Like with Table 4, for each set of N and n values, we ensemble over 25 independent runs, $h = 5$, and the algorithm was set to terminate after 50 iterations, or the line search break flag was set after 10 consecutive line search failures.

Whereas Tables 2 and 4 show that with a budget of 625 simulations per iteration, $n = 25$, and $N = 25$, the implicit filtering and steepest descent techniques on

Table 4 Results of the steepest descent based optimization technique compared over fixed per iteration budgets

N	n	$\overline{I_t}$	$s^2[I_t]$	$\overline{\bar{\phi}_{opt}}$	$s^2[\bar{\phi}_{opt}]$	$\overline{p_{opt}(c_{hard})}$	$s^2[p_{opt}(c_{hard})]$	max $\{p_{opt}(c_{hard})\}$	Failures
Per Iteration Budget = 100									
5	20	82.5	290.2	0.164	0.013	0.124	0.016	0.375	$\frac{0}{25}$
10	10	81.7	1.04e3	0.041	7.13e−4	0.077	0.010	0.372	$\frac{3}{25}$
20	5	93.8	168.6	0.020	2.41e−4	0.022	0.002	0.165	$\frac{12}{25}$
Per Iteration Budget = 625									
5	125	79.6	2.1	0.180	7.27e−5	0.355	0.001	0.397	$\frac{0}{25}$
25	25	47.7	645.9	0.337	0.085	0.163	0.017	0.367	$\frac{0}{25}$
125	5	94	246.0	0.020	1.47e−4	0.018	0.001	0.159	$\frac{11}{25}$
Per Iteration Budget = 1250									
5	250	79.7	2.0	0.183	3.48e−5	0.375	3.44e−4	0.399	$\frac{0}{25}$
10	125	80.3	2.3	0.182	4.00e−5	0.362	8.63e−4	0.400	$\frac{0}{25}$
25	50	80.5	3.0	0.181	2.07e−4	0.340	0.001	0.384	$\frac{0}{25}$
50	25	43.5	646.3	0.211	0.024	0.151	0.019	0.394	$\frac{0}{25}$
125	10	71.2	951.2	0.061	0.003	0.118	0.011	0.379	$\frac{0}{25}$
250	5	98.0	336.5	0.019	8.95e−5	0.036	0.004	0.218	$\frac{10}{25}$
Per Iteration Budget = 2500									
5	500	80.2	1.8	0.185	1.40e−5	0.381	2.15e−4	0.402	$\frac{0}{25}$
10	250	79.6	1.8	0.185	2.18e−5	0.381	2.00e−4	0.400	$\frac{0}{25}$
25	100	79.2	2.5	0.184	3.33e−5	0.358	0.001	0.397	$\frac{0}{25}$
50	50	79.4	1.8	0.183	1.04e−4	0.338	0.002	0.398	$\frac{0}{25}$
100	25	44.2	585.1	0.250	0.043	0.189	0.017	0.382	$\frac{0}{25}$
250	10	80.0	1.76e3	0.054	0.002	0.120	0.012	0.344	$\frac{0}{25}$
500	5	101.5	99.0	0.019	8.10e−5	0.044	0.006	0.331	$\frac{7}{25}$
Per Iteration Budget = 5000									
5	1000	80.6	1.6	0.185	1.01e−5	0.385	2.02e−4	0.400	$\frac{0}{25}$
10	500	80.1	3.3	0.185	1.16e−5	0.387	8.26e−5	0.402	$\frac{0}{25}$
25	200	80.0	2.4	0.186	2.28e−5	0.378	5.03e−4	0.402	$\frac{0}{25}$
50	100	80.0	2.1	0.182	4.65e−5	0.364	7.69e−4	0.399	$\frac{0}{25}$
100	50	79.9	1.6	0.185	1.43e−4	0.354	0.002	0.395	$\frac{0}{25}$
200	25	46.7	442.7	0.207	0.012	0.177	0.018	0.376	$\frac{0}{25}$
500	10	89.1	1.65e3	0.057	0.001	0.126	0.015	0.384	$\frac{2}{25}$
1000	5	99.9	326.2	0.027	4.83e−4	0.052	0.005	0.213	$\frac{7}{25}$

Statistics are calculated over 25 independent runs for each combination of n and N, where a bar represents the sample average, and $s^2[\cdot]$ represents the sample variance. N is equivalent to simulator runs, and n is the number of directions in which we choose new test templates $\mathbf{t}$. I_t is the number of total iterations to termination, which includes all line searches. Termination occurs after 50 iterations, or 10 consecutive failed line searches. A failure occurs when the algorithm terminates at a template with a "true" probability $p(c_{hard}) = 0$

Table 5 Results of the L-BFGS based optimization technique compared over fixed per iteration budgets

N	n	$\overline{I_t}$	$s^2[I_t]$	$\overline{\bar{\phi}_{opt}}$	$s^2[\bar{\phi}_{opt}]$	$\overline{p_{opt}(c_{hard})}$	$s^2[p_{opt}(c_{hard})]$	max $\{p_{opt}(c_{hard})\}$	Failures
Per iteration budget = 100									
5	20	32.7	423.9	0.032	4.27e−4	0.017	4.37e−7	0.019	$\frac{0}{25}$
10	10	45.8	497.4	0.023	1.51e−4	0.017	1.08e−4	0.061	$\frac{1}{25}$
20	5	33.2	682.2	0.012	8.69e−5	0.017	2.56e−6	0.024	$\frac{0}{25}$
Per iteration budget = 625									
5	125	45.0	407.1	0.018	1.39e−4	0.022	4.13e−4	0.119	$\frac{0}{25}$
25	25	37.0	255.1	0.072	0.001	0.018	3.02e−5	0.044	$\frac{0}{25}$
125	5	57.5	187.3	0.016	5.15e−5	0.016	1.14e−4	0.059	$\frac{2}{25}$
Per iteration budget = 1250									
5	250	51.5	232.4	0.019	1.46e−4	0.025	3.79e−4	0.112	$\frac{0}{25}$
10	125	51.0	244.1	0.030	0.002	0.048	0.005	0.270	$\frac{0}{25}$
25	50	49.0	215.7	0.028	9.34e−4	0.032	0.001	0.169	$\frac{0}{25}$
50	25	35.0	361.3	0.065	0.003	0.030	0.004	0.337	$\frac{0}{25}$
125	10	51.7	315.7	0.018	1.54e−4	0.018	1.36e−4	0.059	$\frac{1}{25}$
250	5	53.9	275.1	0.018	1.75e−4	0.016	1.34e−4	0.050	$\frac{3}{25}$
Per iteration budget = 2500									
5	500	44.9	288.2	0.027	4.52e−4	0.041	0.002	0.203	$\frac{0}{25}$
10	250	46.5	282.4	0.026	3.87e−4	0.035	0.001	0.142	$\frac{0}{25}$
25	100	52	367.7	0.019	2.37e−4	0.026	7.01e−4	0.150	$\frac{0}{25}$
50	50	47.6	353.2	0.025	4.45e−4	0.031	8.01e−4	0.129	$\frac{0}{25}$
100	25	34.4	223.3	0.045	0.001	0.026	6.83e−4	0.144	$\frac{0}{25}$
250	10	51.2	423.3	0.018	3.91e−5	0.017	5.79e−5	0.045	$\frac{1}{25}$
500	5	58.5	135.6	0.014	5.50e−5	0.015	4.59e−5	0.026	$\frac{2}{25}$
Per iteration budget = 5000									
5	1000	47.6	531.8	0.016	3.44e−6	0.022	1.98e−5	0.032	$\frac{0}{25}$
10	500	46.3	313.0	0.028	0.001	0.046	0.007	0.368	$\frac{0}{25}$
25	200	47.1	424.9	0.023	4.27e−4	0.033	0.001	0.161	$\frac{0}{25}$
50	100	47.4	391.3	0.016	2.28e−5	0.022	9.76e−5	0.065	$\frac{0}{25}$
100	50	47.8	452.7	0.017	8.17e−5	0.025	2.77e−4	0.094	$\frac{0}{25}$
200	25	31.2	215.4	0.062	0.013	0.018	1.35e−6	0.021	$\frac{0}{25}$
500	10	48.0	212.9	0.020	2.84e−4	0.021	1.86e−4	0.071	$\frac{2}{25}$
1000	5	58.6	345.5	0.015	2.56e−5	0.017	4.55e−5	0.037	$\frac{1}{25}$

Statistics are calculated over 25 independent runs for each combination of n and N, where a bar represents the sample average, and $s^2[\cdot]$ represents the sample variance. N is equivalent to simulator runs, and n is the number of directions in which we choose new test templates $\mathbf{t}$. I_t is the number of total iterations to termination, which includes all line searches. Termination occurs after 50 iterations, or 10 consecutive failed line searches. A failure occurs when the algorithm terminates at a template with a "true" probability $p(c_{hard}) = 0$

average achieve $p_{opt}(c_{hard}) = 0.104$ and $p_{opt}(c_{hard}) = 0.163$ respectively, Table 5 shows the L-BFGS method only achieves $p_{opt}(c_{hard}) = 0.018$ on average. However, as with the previous two algorithms, there is still a noticeable benefit from trading off N for n, given a fixed per iteration budget, and increasing the per iteration budget can improve performance. The L-BFGS technique also fails much less frequently than the implicit filtering technique. Overall though, the ensemble results suggest this method is inferior to the steepest descent based approach, and that the standard BFGS technique may need further modifications to handle the noise in this problem.

6 Summary and conclusions

In this paper, we have proposed three algorithms for solving the coverage directed generation problem, all based on the key observation that the problem can be posed as derivative free optimization of a noisy objective function. By applying techniques from statistical parameter estimation and inverse problems, including the generalized cross validation technique, we are able to generate quality estimates of the gradient of a noisy objective function. With these gradient estimates, we are able to build algorithms that adapt the steepest descent and BFGS techniques from non-noisy continuous optimization.

The algorithm based on gradient descent, on average, empirically outperforms a simple, but sometimes surprisingly effective, implicit filtering based approach. Numerical experiments with a high-level software model of part of IBM's NorthStar processor show that both the implicit filtering and steepest descent techniques are economical in terms of the total number of simulations required for them to be effective, and how to best choose parameters given a fixed per iteration budget of simulations. Furthermore, all our algorithms are relatively easily parallelized in practice, as the repeated simulations at a single point N can be carried out in parallel, and this can further be done in parallel for the n points along the random directions, with the only major bottleneck being the work required during the decision to update to the next template. We suspect that the use of inverse problems based techniques for gradient estimation can be further extended to the evaluation of Hessians and in other contexts where the function and gradients are noisy, and this will be investigated in the future.

Acknowledgements EH and BI's work is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC). RG, BS, and AZ's work is supported by IBM.

Compliance with ethical standards

Conflict of interest The authors declare that they have no conflict of interest.

References

Berahas AS, Byrd RH, Nocedal J (2019) Derivative-free optimization of noisy functions via quasi-newton methods. SIAM J Optim 29:965–993

Borkenhagen J, Sorino S (1999) 4th generation 64-bit PowerPC-compatible commercial processor design. IBM Server Group White Paper

Burden RL, Faires JD (2010) Numerical analysis. Cengage learning

Conn A, Scheinberg K, Vicente L (2009) Introduction to derivative-free optimization. SIAM, Philadelphia

Fine S, Ziv A (2003) Coverage directed test generation for functional verification using Bayesian networks. In: Design automation conference

Golub GH, von Matt U (1997) Generalized cross-validation for large-scale problems. J Comput Graph Stat 1:1–34

Golub GH, Heath M, Wahba G (1979) Generalized cross-validation as a method for choosing a good ridge parameter. Technometrics 21:215–223

Huyer W, Neumaier A (1999) Global optimization by multilevel coordinate search. J Glob Optim 14:331–355

Kelley C (2011) Implicit filtering. SIAM, Philadelphia

Mishra P, Dutt N (2002) Automatic functional test program generation for pipelined processors using model checking. In: 7th annual IEEE international workshop on high-level design validation and test, pp 99–103

Moré JJ, Wild SM (2011) Estimating computational noise. SIAM J Sci Comput 33:1292–1314

Nativ G, Mittermaier S, Ur S, Ziv A (2001) Cost evaluation of coverage directed test generation for the IBM mainframe. In: Proceedings of the 2001 international test conference, pp 793–802

Nocedal J, Wright S (1999) Numerical optimization. Springer, New York

Pétard H (1938) A contribution to the mathematical theory of big game hunting. Am Math Monthly 45:446–447

Pintér JD (1996) Global optimization in action. Springer, New York

Piziali A (2004) Functional verification coverage measurement and analysis. Springer, New York

Powell M (2006) The NEWUOA software for unconstrained optimization without derivatives. In: Pillo GD, Roma M (eds) Large-scale nonlinear optimization. Springer, Boston, pp 255–297

Prashanth LA, Bhatnagar S, Fu M, Marcus S (2017) Adaptive system optimization using random directions stochastic approximation. IEEE Trans Autom Control 62(5):2223–2238

Rios LM, Sahinidis NV (2013) Derivative-free optimization: a review of algorithms and comparison of software implementations. J Glob Optim 56:1247–1293

Spall JC (1987) A stochastic approximation technique for generating maximum likelihood parameter estimates. In: 1987 American control conference, pp 1161–1167

Spall JC (1992) Multivariate stochastic approximation using a simultaneous perturbation gradient approximation. IEEE Trans Autom Control 37(3):332–341. https://doi.org/10.1109/9.119632

Spall JC (1998) An overview of the simultaneous perturbation method for efficient optimization. Johns Hopkins APL Tech Dig 19(4):482–492

Tenorio L (2017) An introduction to data analysis and uncertainty quantification for inverse problems. SIAM, New York

Affiliations

Raviv Gal[1] · Eldad Haber[2] · Brian Irwin[2] · Bilal Saleh[1] · Avi Ziv[1]

Raviv Gal
RAVIVG@il.ibm.com

Eldad Haber
haber@eoas.ubc.ca

Bilal Saleh
BILAL@il.ibm.com

Avi Ziv
AZIV@il.ibm.com

[1] IBM Research Laboratory in Haifa, Haifa, Israel

[2] Department of Earth and Ocean Science, The University of British Columbia, Vancouver, BC, Canada