Manuscript submitted to
AIMS' Journals
Volume **X**, Number **0X**, XX **200X**

doi:10.3934/xx.xxxxxxx

pp. **X–XX**

SMART GRADIENT - AN ADAPTIVE TECHNIQUE FOR IMPROVING GRADIENT ESTIMATION

Esmail Abdul Fattah*, Janet Van Niekerk and Håvard Rue

CEMSE Division, King Abdullah University of Science and Technology, Kingdom of Saudi Arabia

Abstract. Computing the gradient of a function provides fundamental information about its behavior. This information is essential for several applications and algorithms across various fields. One common application that requires gradients are optimization techniques such as stochastic gradient descent, Newton's method and trust region methods. However, these methods usually require a numerical computation of the gradient at every iteration of the method which is prone to numerical errors. We propose a simple limited-memory technique for improving the accuracy of a numerically computed gradient in this gradient-based optimization framework by exploiting (1) a coordinate transformation of the gradient and (2) the history of previously taken descent directions. The method is verified empirically by extensive experimentation on both test functions and on real data applications. The proposed method is implemented in the `R` package `smartGrad` and in C++.

1. **Introduction.** Gradients are one of the oldest constructs of modern mathematics and often form the basis for many optimization problems. Moreover, gradients are used in various function expansions like the Taylor series expansion, in tangent plane construction and in various real-life engineering challenges such as building ramps, roads and buildings, amongst others. The gradient of a mathematical function f, at the input $\boldsymbol{x}$, is denoted by $\nabla f(\boldsymbol{x})$. It can be mathematically interpreted as a rate of disposition based on the disposition of $\boldsymbol{x}$ or graphically as the slope of the tangent plane at $\boldsymbol{x}$ [6]. Often though, the analytical form of $\nabla f(\boldsymbol{x})$ is unknown or computationally intensive to evaluate and hence the popularity of numerical gradients methods. Newton's and quasi-Newton methods [3] are well-known frameworks in optimization that use (numerical) gradients to get the descent directions.

Consider the problem of minimizing a twice differentiable continuous function f, where $f(\boldsymbol{x}) : \mathbb{R}^n \to \mathbb{R}$, is convex, i.e. satisfies,

$$f((1-\lambda)\boldsymbol{x} + \lambda\boldsymbol{y}) \leq (1-\lambda)f(\boldsymbol{x}) + \lambda f(\boldsymbol{y})$$

for $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^n$ and $\lambda \in [0, 1]$. Assume this (unconstrained) optimization problem is solvable, where its optimum $\boldsymbol{x}^*$ exists and is unique. In this scenario, a necessary and a sufficient condition for $\boldsymbol{x}^*$ to be optimal is

$$\nabla f(\boldsymbol{x}^*) = \boldsymbol{0}.$$

2020 *Mathematics Subject Classification.* Primary: 03-08; Secondary: 62-08.

Key words and phrases. Adaptive Technique, Gradient Estimation, Numerical Gradient, Optimization, Vanilla Gradient Descent.

* Corresponding author: Esmail Abdul Fattah, email: esmail.abdulfattah@kaust.edu.sa.

Here, $\nabla f(\boldsymbol{x})$ is composed of the function's partial derivatives, that describe the rates of change in multiple dimensions. More generally we consider the directional derivative of a function. Given a continuous function g with its first order partial derivatives, the directional derivative [10] of g at $\boldsymbol{x} \in \mathbb{R}^n$ in the direction of unit vector $\boldsymbol{u}$ is

$$D_{\boldsymbol{u}} g(\boldsymbol{x}) = \frac{\partial g}{\partial x_1} u_1 + \frac{\partial g}{\partial x_2} u_2 + \ldots + \frac{\partial g}{\partial x_n} u_n.$$

The gradient vector can be reformulated as a combination of directional derivatives based on the canonical basis $\{\boldsymbol{e}_1, \ldots, \boldsymbol{e}_n\}$, where every element of $\boldsymbol{e}_i$ is zero except for the i^{th} position being 1, as in formula (1), that will follow shortly.

Common methods for solving optimizations problems rely on iterative techniques which generate iterations that converge to the optimum $\boldsymbol{x}^*$. The iterations are constructed from the general iterative update equation,

$$\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} + \alpha_k \boldsymbol{d}^{(k)},$$

with step size α_k in the direction of the vector $\boldsymbol{d}^{(k)}$. The choice of the directions $\{\boldsymbol{d}_k\}_{k=1}^n$ depends on the utilized method [5]:

1. Gradient Descent: $\boldsymbol{d}^{(k)} = -\nabla f(\boldsymbol{x}^{(k)})$.
2. Quasi Newton's Method: $\boldsymbol{d}^{(k)} = -\boldsymbol{B}_k \nabla f(\boldsymbol{x}^{(k)})$ where $\boldsymbol{B}_k$ is an estimate of the Hessian at $\boldsymbol{x}^{(k)}$, $\nabla^2 f(\boldsymbol{x}^{(k)})$.
3. Newton's Method: $\boldsymbol{d}^{(k)} = -\nabla^2 f(\boldsymbol{x}^{(k)})^{-1} \nabla f(\boldsymbol{x}^{(k)})$.

The choice of the step size $\{\alpha^{(k)}\}$ is obtained using either exact or inexact line search [6]. Although second order methods are already good enough as they utilize Hessian information, here we focus on improving the first order methods such as gradient descent and quasi-newton methods that include symmetric rank 1 method and BFGS [3].

We summarize and relate four popular gradient computational frameworks next [2].

1. **Exact gradient with the canonical basis:** Based on directional derivatives, the gradient can be computed using the canonical basis $\{\boldsymbol{e}_1, \ldots, \boldsymbol{e}_n\}$,

$$\nabla f(\boldsymbol{x}) = \Big(D_{\boldsymbol{e}_1} f(\boldsymbol{x}), D_{\boldsymbol{e}_2} f(\boldsymbol{x}), \ldots, D_{\boldsymbol{e}_n} f(\boldsymbol{x}) \Big). \tag{1}$$

2. **Exact gradient with a non-canonical basis:** Here, the gradient is computed based on the directional derivatives using a set of directions obtained from $\{\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n\}$. For a non-singular matrix $\boldsymbol{G} = [\boldsymbol{v}_1 | \boldsymbol{v}_2 | \ldots | \boldsymbol{v}_n]$, then

$$\nabla_{\boldsymbol{v}} f(\boldsymbol{x}) = \boldsymbol{G}^{-T} \nabla h(\boldsymbol{\varphi}) \Big|_{\boldsymbol{\varphi}=0} \text{ where } h(\boldsymbol{\varphi}) = f(\boldsymbol{x} + \boldsymbol{G}\boldsymbol{\varphi}). \tag{2}$$

3. **Inexact gradient in canonical basis: Vanilla Gradient (VG).** The gradient of the objective function is computed numerically as an estimate to the exact gradient with the canonical basis, such as using finite-difference methods,

$$\tilde{\nabla} f(\boldsymbol{x}) \approx \nabla f(\boldsymbol{x}). \tag{3}$$

$$\tilde{\nabla} f(\boldsymbol{x}) = \begin{cases} \text{forward finite difference,} \\ \text{backward finite difference,} \\ \text{central finite difference,} \\ \vdots \end{cases}$$

4. **Inexact gradient in non-canonical basis:** The gradient is computed using a general basis instead of the canonical basis,

$$\tilde{\nabla}_{\boldsymbol{v}} f(\boldsymbol{x}) = \boldsymbol{G}^{-T} \tilde{\nabla} h(\boldsymbol{\varphi})\Big|_{\boldsymbol{\varphi}=0} \approx \nabla f(\boldsymbol{x}) \text{ where } h(\boldsymbol{\varphi}) = f(\boldsymbol{x} + \boldsymbol{G}\boldsymbol{\varphi}). \tag{4}$$

For $\boldsymbol{G}^T = \boldsymbol{I}_n$, (2) reduces to (1), and (4) reduces to the Vanilla Gradient in (3) where the partial derivatives are directly computed along the vectors of the canonical basis.

A natural question is if we can construct $\boldsymbol{G}$ in (4) that results in a more accurate Vanilla Gradient. As an illustration, we estimate the gradient of the two-dimensional Rosenbrock function [1] $f(\boldsymbol{x}) = (1-x_1)^2 + 100 \times (x_2 - x_1^2)^2$, see Figure 1, using various bases. We use the central difference method of first order of step length 10^{-3} to get the gradient estimate at $\boldsymbol{x}^{(0)} = (-0.29, 0.4)^T$. As for the directions, we start with the unit vectors $\boldsymbol{e}_1 = (1,0)^T$ and $\boldsymbol{e}_2 = (0,1)^T$ then we keep rotating with an angle of $\pi/10^3$ until most of the directions are explored. The mean square error for the estimated gradient is calculated for each set of directions, and at each direction the magnitude of the gradient is calculated. The results are presented in Figure 1.

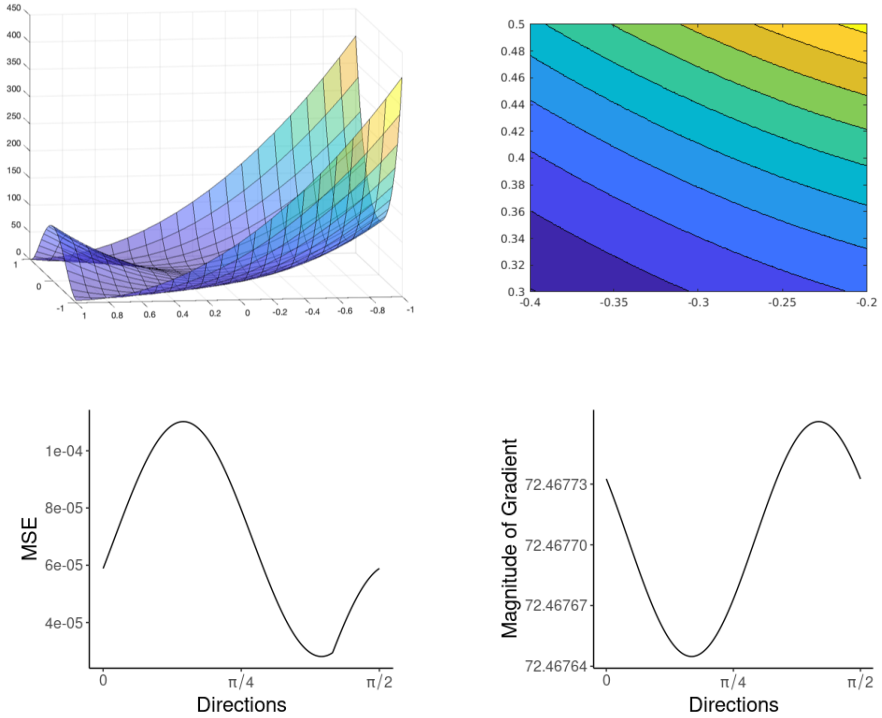

FIGURE 1. 2-dimensional Rosenbrock Function and a contour plot around the point p (-0.29,0.4). At the same point p, the MSE and magnitude of gradient are plotted using different directions.

From Figure 1, we observe opposite trends between the error in the gradient estimates and the magnitude of the gradient itself: the MSE in the gradient estimate is low when the magnitude of the gradient is high. Although such a claim doesn't hold in general, we found empirically it holds on most functions we tested for the

average MSE throughout iterations, and this is sufficient for our proposed method. This observation, suggests the obvious approach, here explained in dimension 2 for simplicity, to improve the gradient estimates at $\boldsymbol{x}$,

1. Find the direction $\boldsymbol{d}$ that maximize the change in the function itself.
2. Estimate the gradient in direction $\boldsymbol{d}$ and the direction orthogonal to $\boldsymbol{d}$.
3. Do a linear transformation for the estimates we get in 2 to obtain the estimates of the Vanilla Gradient as in (4).

We are moving around the same point: to get the descent direction we estimate the gradient and to estimate the gradient we use direction where there is a high change in the objective value f, and this reduction is exactly what we want for the descent direction. This means that the directions we should use for estimating the gradient are exactly what we want to obtain by computing gradients. Within an iterative optimization algorithm, we have access to $\boldsymbol{x}$ at most recent iterations, $\boldsymbol{x}^{(k)}, \boldsymbol{x}^{(k-1)}, \ldots$, then the most recent change in $\boldsymbol{x}$ can be used as the (surrugate) direction $\boldsymbol{d}^{(k)} = \boldsymbol{x}^{(k)} - \boldsymbol{x}^{(k-1)}$. In dimension n, we would need to use the n most recent differences as surrogate directions. These directions should still be relevant, and hence we assume a low or moderate dimension of $\boldsymbol{x}$.

In this paper, we propose in Section 2 a method for constructing the matrix $\boldsymbol{G}$ adaptively . In Section 3, we demonstrate how this construction results in improving the accuracy of a numerical gradient using test functions and Section 4 is for some applications including the R-INLA package. The paper is concluded by a discussion in Section 5.

2. Methodology.

The proposed method is based on using the prior information we have about the descent directions in previous iterations, $\{\boldsymbol{x}^{(i)}\}_{i=k-n}^{k}$ for improving the estimated numerical gradient $\tilde{\nabla} f(\boldsymbol{x})$. The difference between $\boldsymbol{x}^{(i)}$ and $\boldsymbol{x}^{(i-1)}$, $i = k, \ldots k-n$, indicates the direction where the objective function is highly reduced at that position. We then use these n differences as (surrogate) directions to estimate the gradient.

At iteration k, the n most recent differences can be used and the n directions have the following form,

$$\boldsymbol{d}^{(k)} = \frac{\Delta \boldsymbol{x}^{(k)}}{\left\| \Delta \boldsymbol{x}^{(k)} \right\|}$$

where $\Delta \boldsymbol{x}^{(k)} = \boldsymbol{x}^{(k)} - \boldsymbol{x}^{(k-1)}$. To get a unique contribution from each direction $\boldsymbol{d}^{(k)}$, we subtract all the components of $\boldsymbol{d}^{(k)}$ that are in the same direction of $\{\boldsymbol{d}^{(j)}\}_{j=k-n+1}^{k-1}$, then normalize it. We can do this by expressing these directions in terms of projectors,

$$\tilde{\boldsymbol{d}}^{(k)} = \boldsymbol{P}_{\perp \tilde{\boldsymbol{d}}^{(k-1)}} \boldsymbol{P}_{\perp \tilde{\boldsymbol{d}}^{(k-2)}} \cdots \boldsymbol{P}_{\perp \tilde{\boldsymbol{d}}^{(k-n+1)}} \boldsymbol{d}^{(k)}$$

where $\boldsymbol{P}_{\perp \tilde{\boldsymbol{d}}^{(k)}} = I - \tilde{\boldsymbol{d}}^{(k)} (\tilde{\boldsymbol{d}}^{(k)})^T$. We apply this orthogonalization to the n given directions using the Modified Gram-Schmidt (MGS) orthogonalization [7], which is an algorithm used to compute an orthonormal basis $\{\tilde{\boldsymbol{d}}_1^{(k)}, \tilde{\boldsymbol{d}}_2^{(k)}, \ldots, \tilde{\boldsymbol{d}}_n^{(k)}\}$ that spans the same subspace as the original vectors, i.e,

$$\texttt{Span}(\{\boldsymbol{d}_1^{(k)}, \boldsymbol{d}_2^{(k)}, \ldots, \boldsymbol{d}_n^{(k)}\}) = \texttt{Span}(\{\tilde{\boldsymbol{d}}_1^{(k)}, \tilde{\boldsymbol{d}}_2^{(k)}, \ldots, \tilde{\boldsymbol{d}}_n^{(k)}\}).$$

Some tiny noise to the $\boldsymbol{x}$ differences was added to avoid singularities. This is as an adjustment when at least two directions are identical.

The matrix $\tilde{\boldsymbol{G}}^{(k)}$ represents the orthogonal directions at iteration k,

$$\tilde{\boldsymbol{G}}^{(k)} = \left[\tilde{\boldsymbol{d}}_1^{(k)} | \tilde{\boldsymbol{d}}_2^{(k)} | \dots | \tilde{\boldsymbol{d}}_n^{(k)}\right],$$

and is the MGS of matrix $\boldsymbol{G}^{(k)}$,

$$\boldsymbol{G}^{(k)} = \left[\Delta \boldsymbol{x}^{(k)} | \boldsymbol{d}_1^{(k-1)} | \boldsymbol{d}_2^{(k-1)} | \dots | \boldsymbol{d}_{n-1}^{(k-1)}\right].$$

We use matrix $\tilde{\boldsymbol{G}}^{(k)}$ to estimate the gradient at $\boldsymbol{x}^{(k)}$, using (4),

$$\tilde{\nabla}_{\tilde{\boldsymbol{d}}} f^{(k)}(\boldsymbol{x}^{(k)}) = \tilde{\boldsymbol{G}}^{(k)^{-T}} \tilde{\nabla}_{\boldsymbol{e}} h^{(k)}(\boldsymbol{\varphi})\Big|_{\boldsymbol{\varphi}=0} \text{ where } h(\boldsymbol{\varphi}) = f(\boldsymbol{x}^{(k)} + \tilde{\boldsymbol{G}}^{(k)}\boldsymbol{\varphi}). \tag{5}$$

We start with $\tilde{\boldsymbol{G}}^{(0)} = \boldsymbol{I}_n$. This transformation (5) is a rotation/reflection using a different basis, and the length of the gradient is invariant due to the property of orthogonal matrices. The simplicity of this technique is represented by its easy implementation. The gradient is just computed in a different coordinate system using a simple reparamaterization of function f and a matrix-vector multiplication. We label this new approach by Smart Gradient.

Simple example for illustration Assume we have two iterations representing two $\boldsymbol{x}$ positions of a two-dimensional function,

$$\boldsymbol{x}^{(0)} = (1.78, 2.82)^T, \boldsymbol{x}^{(1)} = (1.89, 4.62)^T, \boldsymbol{x}^{(2)} = (11.54, 4.15)^T, \dots$$

At each position k, we form $\boldsymbol{G}^{(k)}$. This matrix is initialized as the identity matrix, so at the first position $k = 0$ the first direction used is $\boldsymbol{d}_1^{(0)} = (1, 0)^T$ and the second $\boldsymbol{d}_2^{(0)} = (0, 1)^T$. When reaching the second position, we get a new direction $\Delta \boldsymbol{x}^{(1)} = \boldsymbol{x}^{(1)} - \boldsymbol{x}^{(0)}$, so matrix $\boldsymbol{G}^{(1)}$ is constructed as follow:

$$\boldsymbol{G}^{(1)} = \left[\boldsymbol{x}^{(1)} - \boldsymbol{x}^{(0)} | \boldsymbol{d}_1^{(0)}\right] = \begin{pmatrix} 0.11 & 1 \\ 1.80 & 0 \end{pmatrix}$$

This matrix is orthonormalized using the MGS algorithm, such that the columns of $\boldsymbol{G}^{(1)}$, $\boldsymbol{g}_1^{(1)}$ and $\boldsymbol{g}_2^{(1)}$, are updated to form $\tilde{\boldsymbol{G}}^{(1)}$ by columns $\tilde{\boldsymbol{d}}_1^{(1)}$ and $\tilde{\boldsymbol{d}}_2^{(1)}$,

$$\tilde{\boldsymbol{d}}_1^{(k)} = \frac{\Delta \boldsymbol{x}^{(k)}}{\left\|\Delta \boldsymbol{x}^{(k)}\right\|} \text{ and } \tilde{\boldsymbol{d}}_2^{(k)} = \frac{\Delta \boldsymbol{x}^{(k-1)} - \left(\boldsymbol{g}_1^{(k)} \cdot \Delta \boldsymbol{x}^{k-1}\right) \dfrac{\Delta \boldsymbol{x}^{(k-1)}}{\left\|\Delta \boldsymbol{x}^{(k-1)}\right\|}}{\left\|\Delta \boldsymbol{x}^{(k-1)} - \left(\boldsymbol{g}_1^{(k)} \cdot \Delta \boldsymbol{x}^{(k-1)}\right) \dfrac{\Delta \boldsymbol{x}^{(k-1)}}{\left\|\Delta \boldsymbol{x}^{(k-1)}\right\|}\right\|},$$

where $\boldsymbol{g}_i^{(k)}$ is the ith column of $\boldsymbol{G}^{(k)}$, and

$$\tilde{\boldsymbol{G}}^{(1)} = \begin{pmatrix} 0.0601 & 0.9981 \\ 0.9981 & -0.0610 \end{pmatrix}.$$

At the next position, $\boldsymbol{x}_2$, the same procedure for the two updates are computed,

$$\boldsymbol{G}^{(2)} = \left[\boldsymbol{x}^{(2)} - \boldsymbol{x}^{(1)} | \boldsymbol{d}_1^{(1)}\right] = \begin{pmatrix} 9.65 & 0.0610 \\ -0.47 & 0.9981 \end{pmatrix}$$

then after MGS,

$$\tilde{\boldsymbol{G}}^{(2)} = \begin{pmatrix} 0.9989 & 0.0486 \\ -0.0486 & 0.9989 \end{pmatrix}$$

The columns of $\tilde{\boldsymbol{G}}^{(2)}$ are orthogonal unit vectors, and they are used as the new set of axes in an orthogonal coordinate system in this unitary transformation of the gradient. For each formed $\tilde{\boldsymbol{G}}^{(k)}$, the gradient is estimated using (5). See Figure 2

for the plotted directions used to compute Vanilla Gradient and Smart Gradient (the first two iterations).

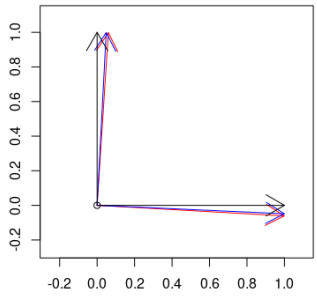

FIGURE 2. Directions used to compute Vanilla Gradient (black) - Smart Gradient (iteration 1 - red) and Smart Gradient (iteration 2 - blue).

3. Numerical Experiments.

3.1. Test Functions.

In this section, some experiments are carried out to compare the performance of using the proposed approach. We chose to compare our approach with another numerical gradient estimator based on the first order central differences.

For each test function, we use `optim` function in `stats` package in R with the BFGS method [3] to get the optimum, and the average squared difference (MSE) between the exact gradient (calculated analytically) and the estimated gradient (using Vanilla Gradient and the Smart Gradient) is calculated at each iteration. The experiment is repeated 100 times with a random initial value.

One of the popular test problems for unconstrained optimization is the Extended Rosenbrock Function, which is unimodal, differentiable and has an optimum in a narrow parabolic valley, see Figure 1. Another one is the Extended Freudenstein Roth Function, and the equations for both functions are here:

1. Extended Rosenbrock Function:

$$f(\boldsymbol{x}) = \sum_{i=1}^{n/2} 100(x_{2i} - x_{2i-1}^2)^2 + (1 - x_{2i-1})^2 \tag{6}$$

2. Extended Roth Freudenstein Function:

$$\begin{aligned} f(\boldsymbol{x}) = \sum_{i=1}^{n/2} &\Big(-13 + x_{2i-1} + x_{2i}(x_{2i}(5 - x_{2i}) - 2)\Big)^2 \\ &+ \Big(-29 + x_{2i-1} + x_{2i}(x_{2i}(x_{2i} + 1) - 14)\Big)^2 \end{aligned} \tag{7}$$

Comparison results are summarized in Table (1) showing an improvement (ratio of the errors of VG and SG) of 2.5 for dimension 5 and at least 3.5 for the other two higher dimensions in Rosenbrock function.

In Figure 3, the two estimated gradients started with almost the same MSE in the first some iterations, then gradually the MSE of the Smart Gradient decreases

$\boldsymbol{x}$ dimension	Average MSE Vanilla Gradient	Average MSE Smart Gradient	Improvement
Extended Rosenbrock Function			
5	2.60e-04	1.04e-04	2.5
10	2.71e-04	0.78e-04	3.47
25	2.80e-04	0.49e-04	5.71
Extended Freudenstein Roth Function			
5	2.26e-04	1.39e-04	1.63
10	2.78e-04	1.42e-04	1.96
25	2.84e-04	1.25e-04	2.27

TABLE 1. The average MSE when using Vanilla and Smart Gradient approaches compared to the exact gradient for different dimensional Rosenbrock and Roth functions

showing a clear improvement, and it stays lower than the MSE of the Vanilla Gradient till the end of the iterations. Matrix $\boldsymbol{G}^{(k)}$ needs at least n iterations to be filled properly with informative directions, and improvements of the rounding off errors become evident after the number of iterations exceeds the dimension of $\boldsymbol{x}$. This matrix continues rolling up for different k until the optimum is found.

It is clear that the gradient is calculated more accurately by Smart Gradient for all the functions in this optimization framework, as the prior information we have about the the descent directions boosts this accuracy. We examined the use of Smart Gradients on range of different test functions, all showing similar behaviour and overall improvement to the examples reported.

3.2. **Smart Hessian.** The Smart Gradient technique can be extended and applied on Hessians. Assume for a continuous function $f(\boldsymbol{x})$, the second partial derivatives exit, we can estimate the Hessian $\nabla^2 f^{(k)}(\boldsymbol{x}^{(k)})$ of this objection function at iteration k based on some directions $\{\tilde{\boldsymbol{d}}_1^{(k)}|\tilde{\boldsymbol{d}}_2^{(k)}|\dots|\tilde{\boldsymbol{d}}_n^{(k)}\}$ as we did for gradient in formulas (4) and (5),

$$\tilde{\nabla}^2_{\tilde{\boldsymbol{d}}} f^{(k)}(\boldsymbol{x}^{(k)}) = \tilde{\boldsymbol{G}}^{(k)^{-T}} \tilde{\nabla}^2 h^{(k)}(\boldsymbol{\varphi})\Big|_{\boldsymbol{\varphi}=0} \tilde{\boldsymbol{G}}^{(k)^{T}} \text{ where } h(\boldsymbol{\varphi}) = f(\boldsymbol{x}^{(k)} + \tilde{\boldsymbol{G}}^{(k)}\boldsymbol{\varphi}) \quad (8)$$

One important application to this Smart Hessian is computing the Hessian of a function at its mode. This adaptive technique can help describing better the curvature of a function at its optimum, using the last n descent directions.

4. Applications.

4.1. **Optimization of a Target Function.** In this example, we use Smart Gradient technique to get the optimum of different dimensional Extended Rosenbrock Function using `optim` function in `stats` package in R, and compare it with Vanilla Gradient. We compute the two gradient with different step-size, in the range of $[10^{-8} - 10^{-1}]$, using first order central difference, and we plot the results as the function of MSE of the converged optimum $\boldsymbol{x}^*$ and the true optimum value $\mathbf{1}$, see Figure 4.

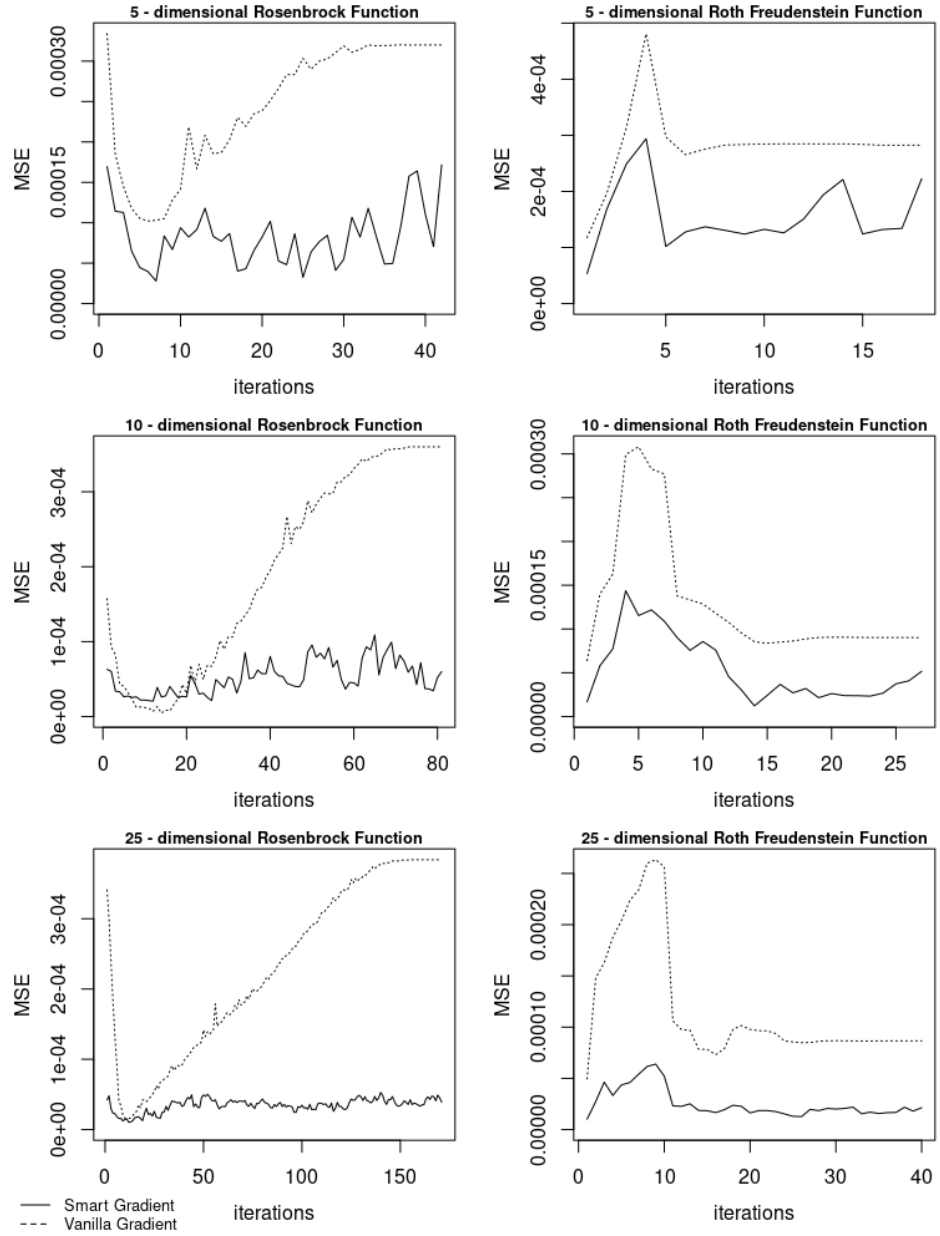

FIGURE 3. MSE for Smart and Vanilla Gradients at each iteration using Different Dimensional Rosenbrock and Roth Functions.

Also, we present in Figure 4 the computational time needed to reach the optimum when using Smart Gradient compared to Vanilla Gradient, as function of the step-size. Smart Gradient shows a higher computational cost compared to Vanilla gradient. However, this cost is still small compared to the accuracy gained, given that $\boldsymbol{x}$ has moderate dimension. The MSE of the converged $\boldsymbol{x}^*$ still has better overall accuracy.

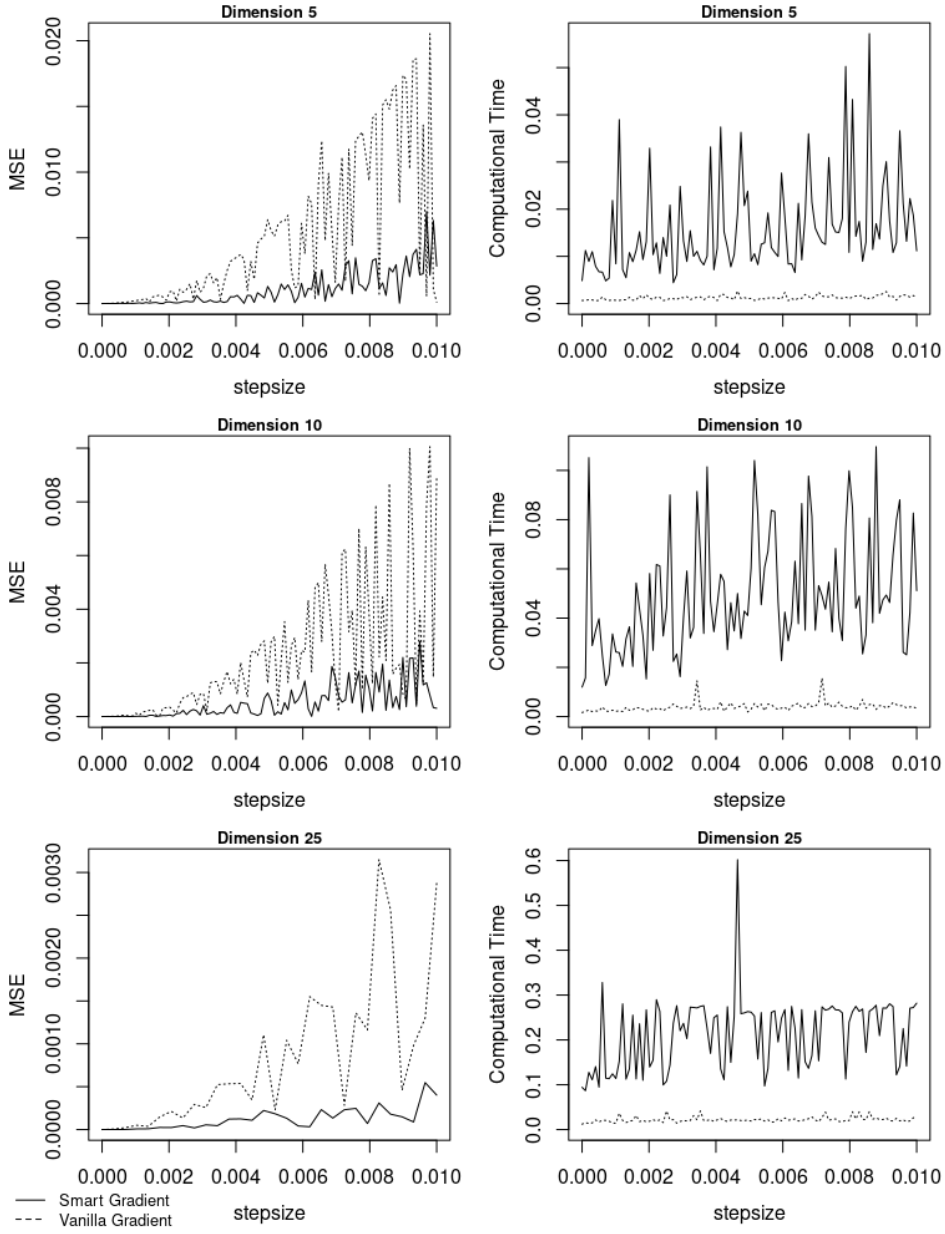

FIGURE 4. MSE and computational time (to reach optimum) as the function of step-size for different dimensional Rosenbrock Function.

4.2. **Hierarchical Optimization of a Hyperparamater.** A motivating application of the Smart Gradient technique is approximate Bayesian inference of a latent Gaussian model using Integrated Nested Laplace Approximation (INLA) [8], [9]. INLA uses combinations of analytical approximations and numerical integration to obtain approximate posterior distributions of the paramaters.

4.2.1. *Brief overview of INLA.* Latent Gaussian models (LGMs) form a class of regression models that include generalized linear models (GLM) like Poisson regression, generalized additive mixed models (GAMM), temporal and spatial models, amongst many others. An LGM is defined for data $\boldsymbol{y}$, a Gaussian latent field $\boldsymbol{z}$ and hyperparameters $\boldsymbol{x}$,

$$\begin{aligned} \boldsymbol{y}|\boldsymbol{z},\boldsymbol{x} &\sim \pi(h(\boldsymbol{z}),\boldsymbol{x}) \\ \boldsymbol{z}|\boldsymbol{x} &\sim \mathcal{N}(\boldsymbol{0},\boldsymbol{Q}^{-1}(\boldsymbol{x})) \\ \boldsymbol{x} &\sim \pi(\boldsymbol{x}), \end{aligned} \tag{9}$$

where $\pi(.,.), \pi(.)$ are density functions, h is a link function, and $\boldsymbol{Q}(\boldsymbol{x})$ is the precision matrix of the latent field $\boldsymbol{z}$.
The main goal of INLA is to approximate the posteriors of $\boldsymbol{z}$ and $\boldsymbol{x}$, i.e. $\pi(\boldsymbol{z}|\boldsymbol{y})$ and $\pi(\boldsymbol{x}|\boldsymbol{y})$, respectively. INLA performs these approximations in two steps - firstly, optimize $\pi(\boldsymbol{x}|\boldsymbol{y})$ and then use the optimum value of $\boldsymbol{x} = \boldsymbol{x}^*$ to optimize $\pi(\boldsymbol{z}|\boldsymbol{y},\boldsymbol{x})$ to find $\boldsymbol{z}^*$, and subsequently approximate $\pi(\boldsymbol{z}|\boldsymbol{y})$ with numerical integration. Note that the dimension of $\boldsymbol{z}$ is often large $(10^3 - 10^5)$ but that of $\boldsymbol{x}$ is often small $(0-20)$. The BFGS method [3] is used to estimate the optimum value of $\boldsymbol{x}$. The iterative nature of INLA and the size of $\boldsymbol{z}$ prohibits exact function evaluations.
Assume $\boldsymbol{x}$ has a prior distribution $\pi(\boldsymbol{x})$, we use the marginal posterior of $\boldsymbol{x}$ to find the mode $\boldsymbol{x}^*$,

$$\pi(\boldsymbol{x}|\boldsymbol{y}) = \frac{\pi(\boldsymbol{z},\boldsymbol{x}|\boldsymbol{y})}{\pi(\boldsymbol{z}|\boldsymbol{y},\boldsymbol{x})} = \frac{\pi(\boldsymbol{x})\pi(\boldsymbol{z}|\boldsymbol{x})\pi(\boldsymbol{y}|\boldsymbol{z})}{\pi(\boldsymbol{z}|\boldsymbol{y},\boldsymbol{x})}.$$

INLA method constructs the approximate $\tilde{\pi}(\boldsymbol{x}|\boldsymbol{y})$ by replacing the full conditional of $\boldsymbol{z}$: $\pi(\boldsymbol{z}|\boldsymbol{y},\boldsymbol{x})$ by its Gaussian approximation, then we write the Laplace approximation of $\pi(\boldsymbol{x}|\boldsymbol{y})$, and evaluate it at the mode $\boldsymbol{z}^*$,

$$\boldsymbol{x}^* \approx \underset{\boldsymbol{x}}{\operatorname{argmin}} \; -\Big(\log\pi(\boldsymbol{y}|\boldsymbol{z}^*,\boldsymbol{x}) + \log\pi(\boldsymbol{z}^*|\boldsymbol{x}) + \log\pi(\boldsymbol{x}) - \log\pi(\boldsymbol{z}^*|\boldsymbol{y},\boldsymbol{x})\Big). \tag{10}$$

The gradient of the objective function in INLA applications is not exact and needs to be estimated numerically. A cheap and efficient technique is Smart Gradient, that shows a great improvement in the MSE with small added cost. Assume for example the dimension of the hyperparamater $\boldsymbol{x}$ is 5 and that of the latent field is 10^3. Adding 5^2 and 5^3 operations to estimate the gradient at every iteration has essentially small cost compared to (1) the gained accuracy of this gradient and (2) the high number of operations needed at every iteration for the inner optimization to get $\boldsymbol{z}^*$.

4.2.2. *Probit regression example.* Probit regression is used when we have binary data and the goal is to model the probability of a success (a value of 1). To illustrate the Smart Gradient technique in a complex optimization problem we consider a continuously indexed spatial probit regression model,

$$\begin{aligned} y_i|\boldsymbol{p} &\sim \text{Bin}(p_i) \\ \text{logit}(p_i)|\beta_0,\beta_1,\boldsymbol{w} &= \beta_0 + \beta_1 v_i + \boldsymbol{A}\boldsymbol{w}, \end{aligned} \tag{11}$$

where $\boldsymbol{p}$ is the vector of probabilities, $\boldsymbol{v}$ is a vector of covariates, $\boldsymbol{w}$ is a Gaussian spatial random field and $\boldsymbol{A}$ is projector matrix that links the spatial field with the data. The fixed effects β_0 and β_1 follow a weakly informative Gaussian distribution a priori, with constant precision $1e^{-3}$. The Gaussian spatial field is assumed to have

a Matérn correlation structure with smoothness parameter ν, scaling parameter κ and marginal variance σ_w^2 such that $\boldsymbol{w} \sim N(\boldsymbol{0}, \boldsymbol{R})$, with the entries of $\boldsymbol{R}$ as,

$$R_{ij} = \frac{\sigma_w^2}{2^{\nu-1}\Gamma(\nu)}(\kappa||\boldsymbol{s}_i - \boldsymbol{s}_j||)^\nu K_\nu(\kappa||\boldsymbol{s}_i - \boldsymbol{s}_j||) \tag{12}$$

for locations $\boldsymbol{s}_i$ and $\boldsymbol{s}_j$, where $K_\nu(.)$ is the modified Bessel function of the second kind and $||.||$ is the Euclidean distance.

The continuously indexed Matérn spatial field $\boldsymbol{w}$ is defined as the weak solution of the following Stochastic Partial Differential Equation (SPDE) [4],

$$(\kappa^2 - \Delta)^{\frac{\alpha}{2}}\boldsymbol{w}(\boldsymbol{s}) = \boldsymbol{W}(\boldsymbol{s}), \tag{13}$$

with $\alpha = \nu + 1$ and where $\boldsymbol{W}(\boldsymbol{s})$ is a white noise Gaussian stochastic process in space (i.e. $\mathbb{R}^2$) with unit variance, and Δ is the Laplacian operator. The parameters are reparameterized for orthogonality and interpretibility to a range parameter $\rho = \sqrt{8\nu}/\kappa$ and marginal variance $\sigma_w^2 = \Gamma(\nu)/\Gamma(\nu+1)4\pi\kappa^{2\nu}$.
The solution $\boldsymbol{w}(\boldsymbol{s})$ to (13) is found through the finite element method by discretizing the continuous space into triangles (which forms a mesh) and mapping the observed locations with a projector matrix $\boldsymbol{A}$, to this mesh composed of triangles.

For m triangles in the mesh, the latent field is $\boldsymbol{z} = \{\beta_0, \beta_1, \boldsymbol{w}\}$ and of dimension $m + 2$. We use the unit square as a spatial domain and the accompanying mesh is illustrated in Figure 5. Note that we extend the mesh beyond the study area to avoid any boundary effects. Independent Gaussian priors with precisions τ_{β_0} and τ_{β_1} for β_0 and β_1, respectively, results in a latent Gaussian prior with precision matrix

$$\boldsymbol{Q}(\boldsymbol{x}) = \begin{pmatrix} \tau_{\beta_0} & 0 & 0 \\ 0 & \tau_{\beta_1} & 0 \\ 0 & 0 & \boldsymbol{R}^{-1}(\boldsymbol{x}) \end{pmatrix},$$

where $\boldsymbol{x} = (\log\sigma_w, \log\rho)$.

We simulate a sample of size 1000, with $\beta_0 = 1/2, \beta_1 = 1, \sigma_w^2 = 0.5$, $\nu = 0.5$ and $\kappa = 10$ and represent the observed locations in Figure 5. Now we perform Bayesian inference for this spatial probit regression model using the `inla()` function. In this inferential procedure, we need to find the modes of the hyperparameters $\kappa, \tau_w = \sigma_w^{-2}$, so that the marginals of the elements of the Gaussian latent field β_0 and β_1 can be estimated. We use (10) to find $\boldsymbol{x}^*$.

When the data is generated, we specify the true $\boldsymbol{x}$ to be $(-\log\sqrt{2}, -\log 5)$ and we use the BFGS method with first order central difference to reach the maximum. We use `INLA` package in R to fit the model using Vanilla and Smart gradients. The absolute mean error is calculated in each case, and the number of calls to reach $\boldsymbol{\theta}^*$ is stored. We repeated the experiment 1000 times, for Vanilla Gradient the average absolute mean error is 0.26224 and the average number of calls is 220, whereas for Smart Gradient the average error is 0.26110 and average calls is 206. When using Smart Gradient, the optimum is reached with less number of calls of the objective function and with an improvement in the optimum value.

5. **Discussion and further considerations.** Gradients are an important tool that is used in various applied mathematical and statistical methods. This wide use of gradients necessitates the search for techniques that improve its estimation.

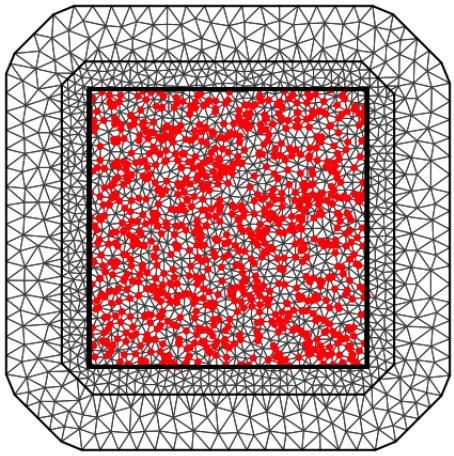

FIGURE 5. Triangulated mesh with 1000 locations on the unit square

Here, we presented a simple framework that enhances the accuracy of gradient estimation within optimization context using the most recent differences between $\boldsymbol{x}$ positions as directions to compute the gradient.

In general, using more precise gradient will lead to a better and more robust performance for gradient based optimization methods. For a moderate number of dimension, say ≤ 25, Smart Gradient method shows improvement in its accuracy with essentially small cost. One useful application for the presented approach is the Hierarchical optimization. That is the case when evaluating the function at two points or more to compute the gradient is costly because of the inner optimization, and using Smart Gradient with its added small cost to gain more accuracy is worth it.

The proposed method is naturally extended to improve the estimate of the Hessian through the attained descent directions. It is implemented in the accompanying R package `smartGrad` available on the github at `esmail-abdulfattah/Smart-Gradient`. Additionally, `smartGrad` can be used to enhance a user-defined gradient formula through the function `makeSmart`, see Appendix A. C++ code is also available on github.

References

[1] Besag, J. (1975). Statistical analysis of non-lattice data. *Journal of the Royal Statistical Society: Series D (The Statistician)*, 24(3):179–195.

[2] Depner, J. S. and Rasmussen, T. C. (2016). *Hydrodynamics of Time-periodic Groundwater Flow: Diffusion Waves in Porous Media*, volume 224. John Wiley & Sons.

[3] Fletcher, R. (1988). Practical methods of optimization.

[4] Lindgren, F., Rue, H., and Lindström, J. (2011). An explicit link between gaussian fields and gaussian markov random fields: the stochastic partial differential equation approach. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 73(4):423–498.

[5] Nocedal and dan Stephen J. Wright, J. (2020). Numerical optimization, 2nd edition.

[6] Nocedal, J. and Wright, S. J. (1999). Numerical optimization.

[7] Picheny, V., Wagner, T., and Ginsbourger, D. (2013). A benchmark of kriging-based infill criteria for noisy optimization. *Structural and Multidisciplinary Optimization*, 48:607–626.

[8] Rue, H., Martino, S., and Chopin, N. (2009). Approximate bayesian inference for latent gaussian models by using integrated nested laplace approximations. *Journal of The Royal Statistical Society Series B-statistical Methodology*, 71:319–392.

[9] Rue, H., Riebler, A., Sørbye, S., Illian, J., Simpson, D. P., and Lindgren, F. (2016). Bayesian computing with inla: A review.

[10] Thomas, G., Weir, M. D., Hass, J., and Giordano, F. (2005). Thomas' calculus early transcendentals (11th edition) (thomas series).

Appendix A. smartGrad Installation and Examples.

A.1. Installation.

```
library("devtools")
install_github("esmail-abdulfattah/Smart-Gradient",
                subdir = "smartGrad")
```

A.2. **makeSmart Function.** To illustrate how to use this function, we use the Extended Rosenbrock function of dimension n. It has global minimum at $\boldsymbol{x}^* = \mathbf{1}$. We estimate the gradient of this objective function f using a simple central difference method, with step size 10^{-3}.

```
myfun <- function(x) {
  res <- 0.0
  for(i in 1:(length(x)-1))
    res <- res + 100*(x[i+1] - x[i]^2)^2 + (1-x[i])^2
  return(res)
}

mygrad <- function(fun,x){
  h = 1e-3
  grad <- numeric(length(x))
  for(i in 1:length(x)){
    e = numeric(length(x))
    e[i] = 1
    grad[i] <- (fun(x+h*e) - fun(x-h*e))/(2*h)
  }
  return(grad)
}
```

A user can change a numerical gradient function `mygrad` to a SMART numerical gradient function `mySmartgrad`, and then this SMART function can be used instead. We use the `optim` function from `stats` package with `BFGS` algorithm.

```
library("stats")
library("smartGrad")
x_dimension = 5
x_initial = rnorm(x_dimension)
result <- optim(par = x_initial,
                        fn = myfun,
                        gr = makeSmart(fn = myfun,gr = mygrad),
                        method = c("BFGS"))
```

Received xxxx 20xx; revised xxxx 20xx.

E-mail address: esmail.abdulfattah@kaust.edu.sa
E-mail address: janet.vanniekerk@kaust.edu.sa
E-mail address: haavard.rue@kaust.edu.sa