BENCHMARKING ITERATIVE OPTIMIZATION ALGORITHMS

A MASTERS THESIS

SUBMITTED ON THE 5 DAY OF MAY, 2020

TO THE DEPARTMENT OF COMPUTATIONAL SCIENCE

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

OF THE SCHOOL OF SCIENCE AND ENGINEERING

TULANE UNIVERSITY

FOR THE DEGREE OF MASTER OF SCIENCE

BY

Elliot Hill

Elliot Hill

Approved: [illegible]

James Hyman, Ph.D.

Director

Tong Wu

Tong Wu, Ph.D.

This work is dedicated to the friends, family, and mentors who have helped me along my journey through academia. I want to give a special thanks to Natasha Navejar, James "Mac" Hyman, Tong Wu, and Douglas and Rebecca Hill for supporting me in the development of this work.

List of Tables

List of Figures

Contents

1 Comparing optimization algorithms

Scientists, engineers, and mathematicians have developed hundreds of algorithms for solving numerical optimization problems [21, 20, 22, 16, 18, 17]. With such a diversity of methods available, researchers are left to choose which method will be the most effective for their applications. However, not all optimization algorithms work equally well on all problems [26]. Thus researchers must be prudent in which algorithms they choose for a given optimization problem to achieve satisfactory results in an efficient and timely manner. The issue is that comparing algorithms across a range of problems that are representative of common optimization problems is difficult and time-consuming.

1.1 The challenge of comparing optimization methods

Outwardly, the task of comparing whether one algorithm performs better than another seems like a simple task. It would appear that all we need to do is run a select number of algorithms on a problem of interest and compare each algorithm's performance. However, performance is a multifaceted subject, and we can assess it by multiple metrics, such as computational cost, convergence rate, and many more measures [1]. Further complicating matters is that ”no free lunch” theorems [26] imply that an algorithm with better performance on one class of problems necessitates worse performance on another class of problems. Therefore, the performance of an optimization algorithm depends on the problem it is solving. As such, what performance metric we use to compare algorithms can alter our conclusions of which algorithm is better than another. Because of the looseness of the term ”performance,” we must take careful consideration to obtain fair and unbiased comparisons of methods [1]. Without consistent and objective experiments to test comparable performance metrics, comparing the performance of algorithms can be difficult and subjective. Consequently, comparing optimization algorithms through benchmarking experiments is a significant research problem and provides researchers with the necessary information to choose an appropriate algorithm for their applications.

1.2 Optimization benchmarking libraries

Due to the tediousness and difficulty of comparing optimization algorithms on a large variety of problems, researchers have developed software libraries for automating the benchmarking process [11, 13, 3, 8]. These libraries usually have suites of test functions and provide different quantitative and qualitative performance metrics to evaluate algorithms. The primary goal of such software is to efficiently test algorithms across hundreds or even thousands of test problems and aggregate their performance data into summary statistics that can we can use to compare algorithms.

1.3 Comparing iterative optimization methods

One class of optimization methods that have had a considerable and lasting impact on science, engineering, and applied mathematics is iterative methods. Iterative methods begin with an initial solution and repeatedly update that solution, with the hope that the sequence of iterates that are generated by the algorithm will converge to a satisfactory solution [16].

In this work, we compare the prototypical iterative methods, gradient descent (also known as steepest descent) and the BFGS (Broyden, Fletcher, Goldfarb, Shanno) algorithm, one of the original quasi-Newton methods. Our goal is to determine which types of problems the methods succeed or fail on. Benchmarking these prototypical methods will not only allow us to predict which kinds of problems these algorithms will perform well on, but also give us insight into when and why they underperform so that we can design new algorithms that remediate the deficiencies of these algorithms.

Though optimization practitioners do not often use gradient descent and quasi-Newton methods in their original formulation, these methods form the basis for many of the improved variants in current use. For example, algorithms derived from gradient descent, like stochastic gradient descent, have become a popular choice for solving many statistical and machine learning optimization problems [23, 4].

We begin this work by laying out the basic theory required to implement optimization algorithms in section 2, followed by the formulation and discussion of the methods we benchmarked in section 3. We then describe the types of problems that we benchmarked each optimization method on and some of their difficulties in section 4. In section 5, we give the details of the benchmarking software we used and describe the types of results it provides. We outline the experimental procedure we conducted using the benchmarking software in section 6, and we compiled the results of our experiments in section 7. We conclude this work with a discussion of our results and their practical implications in sections 8 and 9.

2 Optimization background

Optimization is the maximization or minimization of a function, where the global minimum is denoted

$$\min_{\boldsymbol{x}} f(\boldsymbol{x}),$$

where $f : \mathbb{R}^n \to \mathbb{R}$ is a continuous function referred to as the objective function, and $\boldsymbol{x} \in \mathbb{R}^n$ is a real vector with n real elements. Finding the global optimum of a nonconvex function is typically an NP-hard problem. Therefore, in this work we consider the task of finding local minimums of f. Our goal is to find a *local minimizer*, $\boldsymbol{x}^*$ such that

$$f(\boldsymbol{x}^*) \leq f(\boldsymbol{x}) \quad \text{for all } \boldsymbol{x} \text{ near } \boldsymbol{x}^*.$$

Often there are constraints on the problem we wish to solve, however the problems we discuss here are unconstrained, so there is no conditions on $\boldsymbol{x}$ and f is defined for all $\boldsymbol{x}$. Though [18, 16, 17] are the primary references used for sections 2 and 3, for consistency we tend to stick to the notation of [18].

2.1 Scaling

Before we begin solving an optimization problem, an important step is to scale $\boldsymbol{x}$. We scale $\boldsymbol{x}$ because large differences in the scale of variables can cause f to be more sensitive to changes in some variables than others. Poor scaling often manifests in $\boldsymbol{x}^*$ lying in the center of a narrow, elliptical valley in the objective function. Additionally, if we measured variables in different units, then they could have a large difference in their magnitudes. Algorithms that fail to account for poor scaling tend to converge slowly in these valleys.

While some algorithms are scale-invariant, many others, are not [18]. Thus, it is usually best practice to scale variables before applying an algorithm.

2.2 Overview of algorithms

The class of numerical optimization algorithms we investigate is called iterative methods. Iterative methods start at an initial solution and iteratively improve on that solution until a stopping criterion is satisfied. The primary difference in the behavior of different iterative methods is how they move from one iterate to the next. Differences in the update of the iterate may include using more or less information about the function at the current point, or incorporating previous information from previous iterates into the update step. Though there are countless variations of iterative methods, in general, we want our algorithms to have have certain properties to ensure they perform well.

2.2.1 Properties of a useful algorithm

Though the performance of an algorithm is problem dependent, we seek algorithms that are:

- **Efficient**: we prefer algorithms with low computational cost (be it floating-point operations or memory use) over algorithms with high computational cost
- **Robust**: we want algorithms to perform well on a broad class of problems
- **Accurate**: we want our algorithm to converge to a valid solution with a high degree of precision
- **Scalable**: We would like our algorithms to perform well in both high and low dimensional spaces
- **Interpretable**: Ideally, we favor algorithms whose behavior is easy to understand, because it makes debugging and improving them less difficult

Often, when we design optimization algorithms, the balance of these properties is a zero-sum game. For example, an algorithm with high accuracy may have low interpretability, or an algorithm with high efficiency might scale poorly into higher dimensions.

2.3 Line search methods

The class of optimization algorithms we study in this work are called line search methods [16]. For line search methods, the next iteration is given by

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \alpha_k \boldsymbol{p}_k, \tag{1}$$

where the step length, α_k, is the distance an iterate will travel along the search direction, $\boldsymbol{p}_k$ if $||\boldsymbol{p}_k|| = 1$. In the methods we address, the search direction takes the form

$$\boldsymbol{p}_k = -\boldsymbol{B}_k^{-1} \nabla f_k, \tag{2}$$

where $\boldsymbol{B}_k$ is a symmetric, nonsingular matrix. In the line search strategy, the search direction of iterate k, $\boldsymbol{p}_k$, is solved for first, then the step length, α_k is found afterward by solving the one-dimensional minimization subproblem

$$\alpha* = \min_{\alpha > 0} f(\boldsymbol{x}_k + \alpha_k \boldsymbol{p}_k). \tag{3}$$

We call solving for the optimal α^* an exact line search. However, in practice, we often do not want to solve for the optimal step length because it can be prohibitively expensive if we find it at each iteration. Instead, we perform an inexact line search, which looks for a step length that satisfies conditions that promote general movement towards a minimum without incurring the cost of a performing an exact line search.

2.3.1 Wolfe conditions

To perform an inexact line search we require conditions for choosing an α that approximately solves 3. The Wolfe conditions ([25]) are a set of conditions that α is commonly required to satisfy when choosing the current iterate's step length. The first condition is sufficient decrease, also known as the Armijo condition. Sufficient decrease is satisfied if

$$f(\boldsymbol{x}_k + \alpha \boldsymbol{p}_k) \leq f(\boldsymbol{x}_k) + c_1 \alpha_k \nabla f_k^T \boldsymbol{p}_k, \tag{4}$$

where the constant $c_1 \in (0, 1)$. This condition requires that changes in f be proportional to the step length and directional derivative, $\nabla f_k^T \boldsymbol{p}_k$. If α_k is small enough the sufficient decrease condition will always be satisfied, however the convergence of the algorithm will also be slow for small α_k, thus we require another condition to penalize conservative values of α_k.

The second Wolfe condition is the sufficient curvature condition, which stipulates that α_k satisfy

$$\nabla f(\boldsymbol{x}_k + \alpha_k \boldsymbol{p}_k)^T \boldsymbol{p}_k \geq c_2 \nabla f_k^T \boldsymbol{p}_k, \tag{5}$$

where $c_2 \in (c_1, 1)$ is a constant. The sufficient curvature condition is met when $\nabla f_k^T \boldsymbol{p}_k$ is large and negative, which indicates that moving in this direction will give a significant decrease in f, however if $\nabla f_k^T \boldsymbol{p}_k$ is small then f will not change much along this direction, and therefore we should end the line search in this direction.

2.3.2 Backtracking

In practice, we can drop the second Wolfe condition (5) by using a strategy called backtracking. In backtracking, we choose the next step length by testing (4) for some initial step length, and if the reduction in f does not satisfy the sufficient decrease condition, we contract the step length by some factor ρ

until the sufficient decrease condition is met. We provide the pseudo-code in algorithm 1.

Algorithm 1 Backtracking pseudo-code

```
Choose $\alpha_{\text{initial}} > 0$
$\rho \in (0, 1)$
$c \in (0, 1)$
Set $\alpha = \alpha_{\text{initial}}$
while $f(\boldsymbol{x}_k + \alpha \boldsymbol{p}_k) \leq f(x_k) + c\alpha \nabla f_k^T p_k$ do
    $\alpha = \rho\alpha$
end while
Terminate with $\alpha_k = alpha$
```

2.3.3 Cubic polynomial line search

While backtracking is an improvement over a fixed step size, we can do better than simply reducing α repeatedly by some constant factor ρ as in algorithm 1. Instead, we can make educated guesses for the optimal α by using polynomial interpolation. We layout the necessary information to implement a cubic line search, and the details can be found in [16] and [18])

We would like to start a line search by modeling

$$\xi(\alpha) = f(\boldsymbol{x}_k - \alpha \boldsymbol{p})$$

with a cubic polynomial. However, initially we only have three pieces of information: $\xi(0) = f(\boldsymbol{x}_k)$, $\xi'(0) = \nabla f(\boldsymbol{x}_k)^T \boldsymbol{p} < 0$, and $\xi(1) = f(\boldsymbol{x}_k + \boldsymbol{p})$, which is only enough information to form a quadratic model

$$q(\alpha) = \xi(0) + \alpha\xi'(0) + \alpha^2(\xi(1) - \xi(0) - \xi'(0)).$$

If we let the initial $\alpha \in [\rho_{\text{low}}, \rho_{\text{high}}] \subset (0, 1)$ then the global minimum of q is given by

$$\alpha^* = \frac{-\xi'(0)}{2(\xi(1) - \xi(0) - \xi'(0))}$$

If α fails to satisfy (4), using $\xi(0)$, $\xi'(0)$, α_i and α_{i-1} (the most recent values of α that did not satisfy 4), we can construct a cubic polynomial

$$q(\alpha) = \xi(0) + \xi'(0)\alpha + c_2\alpha^2 + c_3\alpha^3$$

The local minimum of which is given by

$$\alpha^* = \frac{-c_2 + \sqrt{c_2^2 - 3c_3\xi'(0)}}{3c_3}$$

Usually a line search terminates when an α is found that satisfies (4), however in practice, it can take many reductions to find a suitable α. Therefore, in the implementation of a cubic line search, once we have chosen α^* we check it against the conditions

$$\alpha_{\text{new}} = \begin{cases} \rho_{\text{low}}, & \alpha^* \leq \rho_{\text{low}} \\ \alpha^*, & \rho_{\text{low}} < \alpha^* < \rho_{\text{high}} \\ \rho_{\text{high}}, & \alpha^* \geq \rho_{\text{high}} \end{cases}$$

This is called *safeguarding* and prevents the line search from stagnating [16].

2.4 The gradient and Hessian

In this work, we investigate algorithms that choose their search direction by using higher order information of f, namely the gradient and Hessian of f. The gradient is a vector of n partial derivatives defined as

$$\nabla f(\boldsymbol{x}) = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n}\right)^T.$$

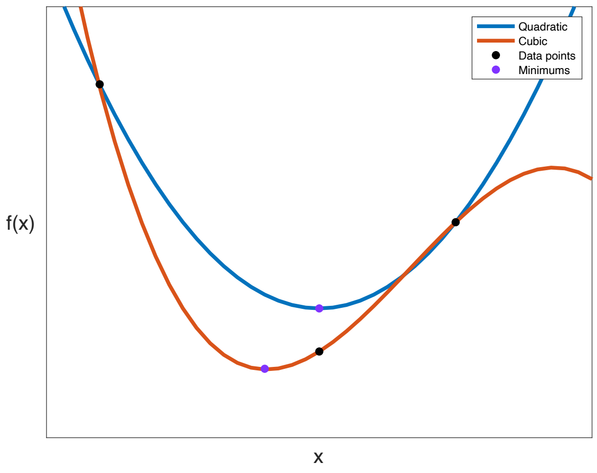


Figure 1: In a cubic polynomial line search, a quadratic is tested first using two points and a derivative. If the minimum of the quadratic does not satisfy the sufficient decrease condition, a cubic polynomial is fit by using the previous information and the minimum of the quadratic. Then, the minimum of the cubic polynomial is solved for.

The Hessian, $\nabla^2 f(\boldsymbol{x})$, is a symmetric matrix of second partial derivatives of f and is defined as

$$\nabla^2 f(\boldsymbol{x})_{ij} = \frac{\partial^2 f}{\partial x_i \partial x_j} \qquad \text{for } i = 1, ..., n, \ j = 1, ..., n.$$

Informally, the gradient gives the direction in which $f(\boldsymbol{x})$ increases most rapidly, and the Hessian tells us the local curvature of $f(\boldsymbol{x})$, that is, the rate of change of the gradient.

2.4.1 Finite difference approximation

In order to use a gradient, we must compute it. If we are lucky, there is an analytical gradient for our objective function, such as in the case of linear least squares, but in practice obtaining an analytical derivative is difficult. Therefore, we often need to estimate the gradient numerically, which can be accomplished by a finite difference approximation [10]. In one dimension, the second-order central difference approximation for the derivative is given by

$$\nabla f(x_i) \approx \frac{f(x_{i+1}) - f(\boldsymbol{x}_{i-1})}{2h}$$

where h is the distance between each x_i. This scheme of approximating derivatives with function values generalizes to higher dimensions and is the method we use for calculating numerical gradients.

2.5 Solutions and stopping criteria

In the optimization of f, we require a way to tell if a point is a local minimizer so that we can terminate an algorithm when it has reached a satisfactory solution. If we assume that f is twice continuously differentiable, then we can examine the gradient and Hessian of $f(\boldsymbol{x}^*)$ to determine if $\boldsymbol{x}^*$ is a local minimizer. For $\boldsymbol{x}^*$ to be a local minimizer, the following two conditions must hold for a local minimizer

Theorem 2.1 (First-order necessary conditions)**.** *If $\boldsymbol{x}^*$ is a local minimizer and f is continuously differentiable in an open neighborhood of $\boldsymbol{x}^*$, then $\nabla f(\boldsymbol{x}^*) = 0$.*

Theorem 2.2 (Second-order necessary conditions)**.** *If $\boldsymbol{x}^*$ is a local minimizer of f and $\nabla^2 f$ exists and is continuous in an open neighborhood of $\boldsymbol{x}^*$, then $\nabla f(\boldsymbol{x}^*) = 0$ and $\nabla^2 f(\boldsymbol{x}^*)$ is positive semidefinite.*

Theorem 2.1 tells us if $\boldsymbol{x}^*$ is a stationary point, that is $\nabla f(\boldsymbol{x}^*) = 0$, and theorem 2.2 tells use whether $\boldsymbol{x}^*$ is a local minimizer of f. For example, if the Hessian of $f(\boldsymbol{x}^*)$ is positive-semidefinite then $\boldsymbol{x}^*$ is a minimum, whereas if it is negative semidefinate then it is a maximum.

3 Optimization methods

We begin our investigation of iterative optimization methods by considering one of the simplest descent methods, namely, gradient descent. We then discuss some of the problems with the method. Following gradient descent, we explore Newton's method, which improves on gradient descent by correcting some of its deficiencies. Lastly, we cover a quasi-Newton method, the BFGS (Broyden, Fletcher, Goldfarb, Shanno) algorithm, which emulates properties of Newton's method but avoids some of its shortcomings.

3.1 Gradient descent

Gradient descent, also known as steepest descent, chooses the search direction at each iteration that has the largest negative change in f. The idea is intuitive; if we wish to reach a minimum quickly, we should proceed in the direction that f decreases most rapidly. We begin with the derivation of the gradient descent search direction used in the algorithm's update step and then discuss some of the properties of the algorithm.

To derive steepest descent search direction we can expand with a Taylor series about the point $\boldsymbol{x}_k$

$$f(\boldsymbol{x}_k + \alpha \boldsymbol{p}) = f(\boldsymbol{x}_k) + \alpha \boldsymbol{p}^T \nabla f(\boldsymbol{x}_k) + \frac{1}{2}\alpha^2 \boldsymbol{p}^T \nabla f(x_k + t\boldsymbol{p})\boldsymbol{p}, \qquad \text{for some } t \in (0, \alpha).$$

This tells us that the rate of change of f in the search direction $\boldsymbol{p}$ at the point $\boldsymbol{x}_k$ is $\boldsymbol{p}^T \nabla f(\boldsymbol{x}_k)$. Then the unit direction that has the largest rate of negative change is the solution to

$$\min_{p} \boldsymbol{p}^T \nabla f(\boldsymbol{x}_k), \quad \text{subject to } ||\boldsymbol{p}|| = 1.$$

Because $\boldsymbol{p}^T \nabla f(\boldsymbol{x}_k) = ||\boldsymbol{p}^T|| ||\nabla f(\boldsymbol{x}_k)|| \cos(\theta) = ||\nabla f(\boldsymbol{x}_k)|| \cos(\theta)$ we obtain the minimizer when $\cos(\theta) = -1$ and $\boldsymbol{p} = -\nabla f(\boldsymbol{x}_k) || \nabla f(\boldsymbol{x}_k)||$.

The search direction that gradient descent uses is

$$\boldsymbol{p} = -\nabla f(\boldsymbol{x}_k),$$

which gives the update step

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - \nabla f(\boldsymbol{x}_k).$$

This is equal to (1) when $\boldsymbol{B}_k$ is the identity matrix, I. We provide the psuedo-code for gradient descent in algorithm 2.

Algorithm 2 Gradient descent pseudo-code

```
Choose $x_0$
Choose tolerance
k = 0
while $||\nabla f_(\boldsymbol{x}_k)||$ > tolerance do
    $\boldsymbol{p} = -\nabla f_(\boldsymbol{x}_k)$
    find $\alpha_k$ with a line search
    $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \alpha_k \boldsymbol{p}_k$
    $k = k + 1$
end while
```

3.1.1 Properties of gradient descent

Intuitively, the direction of steepest descent seems like the best direction to move in if we want to minimize a function as fast as possible. However, there are scenarios when the direction of the largest negative partial derivative (the gradient) is a lousy search direction. Consider when the objective function is a shallow, wide surface with low curvature. In this case, a small derivative will decrease our step size when we would rather have larger step sizes. Alternatively, if the objective function is steep with high curvature, a large derivative may give large step sizes that are too large and cause instability [24]. The underlying problem is that the gradient scales with f, thus gradient descent requires a way to moderate the step size (such as a line search) at each iteration to ensure good performance [16]. If we choose proper step lengths, Gradient descent has a linear convergence rate [17].

The following theorem (provided in [18]) can give us some insight into how we can expect gradient descent to perform on ill-conditioned problems.

Theorem 3.1. *Suppose that $f : \mathbb{R}^n \to \mathbb{R}$ is twice continuously differentiable, and that gradient descent with exact line search converges to a point $\boldsymbol{x}^*$ where $\nabla^2 f(\boldsymbol{x}^*)$ is positive definite. Let r be a scalar that satisfies*

$$r \in (\frac{\lambda_n - \lambda_1}{\lambda_n + \lambda_1}, 1).$$

where λ_1 and λ_n are the smallest and largest eigenvalues of $\nabla^2 f(\boldsymbol{x}^*)$, respectively. Then $\forall k$ sufficiently large

$$f(\boldsymbol{x}_{k+1}) - f(\boldsymbol{x}^*) \leq r^2[f(\boldsymbol{x}_k) - f(\boldsymbol{x}^*)].$$

This theorem tells us that the convergence of gradient descent depends on the ratio of the largest to the smallest eigenvalue of the Hessian of the objective function. Heuristically, what this tells us is that if the ratio is large, the curvature at the point $\boldsymbol{x}_k$ is highly ellipsoidal. If the ratio is substantial, then the gradient direction can lie nearly orthogonal to the minimum, and thus steps in that direction will progress towards the minimum exceptionally slowly. We can see this phenomenon in the zig-zag search path of figure 3.1.1.

In theorem 3.1, it is assumed that an exact line search is used. Therefore, it is an optimistic estimate of convergence, and inexact line searches (which we used for our experiments) may have even slower convergence. Though, the underlying cause is different, ill-conditioning and poor scaling cause similar pathological behavior for algorithms like gradient descent. This is because, in both scenarios, the objective function tends to be ellipsoidal and decreases the efficiency of the gradient as a search direction.

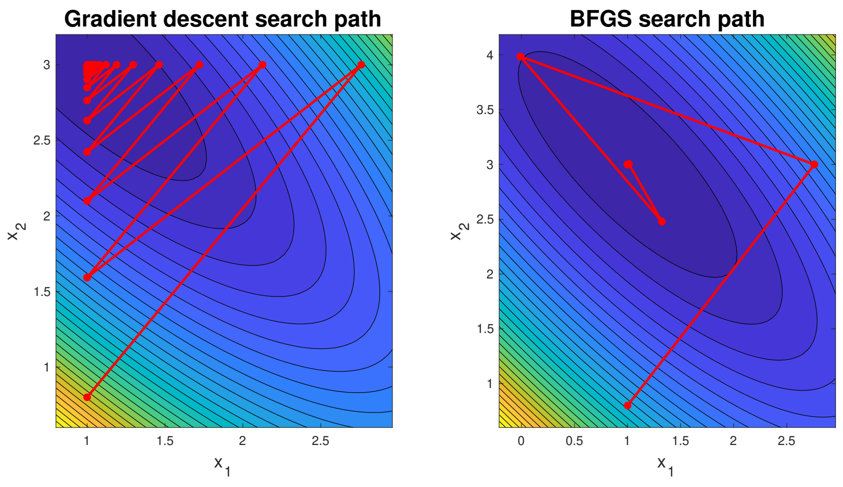


Figure 2: The search paths of the iterates for gradient descent and BFGS. Notice that the ellipsoidal contours of the objective function cause the gradient descent search direction to be nearly orthogonal to the minimum and slows convergence, whereas the BFGS algorithm accounts for the rate of change of the gradient and converges in fewer steps.

Historically, gradient descent has not favored over more advanced methods because of its comparatively slower theoretical convergence rate [16, 18]. However, recently variants of gradient descent, such as stochastic gradient descent ([21]), have seen a resurgence in popularity for large scale applications due to the relatively inexpensive computational cost of the algorithm [4].

3.2 Newton's method

The idea behind Newton's method is to account for the local curvature of f by using second-order information contained in the Hessian, $\nabla^2 f$. By incorporating the Hessian into the update step, we can correct both the search direction and step length by accounting for the rate of change of the gradient.

To derive the method, we begin by assuming that $\boldsymbol{x}_k$ is the argument that minimizes the local quadratic model of f, which we denote m. We can express m as a second-order Taylor expansion about $\boldsymbol{x}$.

$$m(\boldsymbol{x}) = f(\boldsymbol{x}_k) + \nabla f(\boldsymbol{x}_k)^T(\boldsymbol{x} - \boldsymbol{x}_k) + \frac{1}{2}(\boldsymbol{x} - \boldsymbol{x}_k)^T \nabla^2 f(\boldsymbol{x}_k)(\boldsymbol{x} - \boldsymbol{x}_k) \tag{6}$$

If we take the gradient of our local model m and set it equal to zero we get

$$\nabla m(\boldsymbol{x}) = \nabla f(\boldsymbol{x}_k) + \nabla^2 f(\boldsymbol{x}_k)(\boldsymbol{x} - \boldsymbol{x}_k) = 0$$

Thus, the system we need to solve for Newton's method is

$$\nabla^2 f(\boldsymbol{x}_k)(\boldsymbol{x}^{(k+1)} - \boldsymbol{x}_k) = -\nabla f(\boldsymbol{x}_k) \tag{7}$$

Which leads to

$$\begin{aligned} 0 &= \nabla f(\boldsymbol{x}_k) + \nabla^2 f(\boldsymbol{x}_k)\boldsymbol{x}^{(k+1)} - \nabla^2 f(\boldsymbol{x}_k)\boldsymbol{x}_k \\ \nabla^2 f(\boldsymbol{x}_k)\boldsymbol{x}^{(k+1)} &= -\nabla f(\boldsymbol{x}_k) + \nabla^2 f(\boldsymbol{x}_k)\boldsymbol{x}_k \end{aligned}$$

Solving for the next iteration $\boldsymbol{x}^{(k+1)}$ we obtain the Newton update step

$$\boldsymbol{x}^{(k+1)} = \boldsymbol{x}_k - \nabla^2 f(\boldsymbol{x}_k)^{-1}\nabla f(\boldsymbol{x}_k). \tag{8}$$

Though the problem is formulated this way, we always try to avoid computing the inverse of the Hessian $\nabla^2 f(\boldsymbol{x}_k)^{-1}$. Instead we solve the system (7) for the shift vector $(\boldsymbol{x}^{(k+1)} - \boldsymbol{x}_k)$.

The second term in the Newton update step (8) is

$$\alpha \boldsymbol{p}_k = -(\nabla^2 f(\boldsymbol{x}_k))^{-1}\nabla f(\boldsymbol{x}_k).$$

In a sense, this vector shifts the current point $\boldsymbol{x}$ by the direction and step size that minimizes the error in the local quadratic model of f at $\boldsymbol{x}$.

3.2.1 Properties of Newton's method

Newton's method has a quadratic convergence rate when an iterate is near local minimums; however, when the current iterate is far away from a local minimum, there is no guarantee that the Newton direction will be a descent direction [18]. One of the benefits of Newton's method is that it has an implicit step size, $\alpha = 1$, which obviates the need to find a new step size at each iteration as we did in gradient descent. In practice, the biggest shortcoming of Newton's method is that we must solve a linear system (7) at each iteration, which is expensive to compute in high dimensions.

3.3 Quasi-Newton methods

Quasi-Newton methods were developed to improve the search direction of gradient descent and reduce the computational cost of Newton's method. Instead of evaluating the full Hessian (as in Newton's method), we construct an approximation of the Hessian, that we iteratively update using changes in the gradient. Like gradient descent, Quasi-newton methods use only function and gradient evaluations to compute the current iterate. The aim is to replicate the fast convergence properties of Newton's method by incorporating second-order information of f without incurring the cost of solving the Newton equation (7) at each iteration. The critical step in these algorithms is updating the Hessian approximation, so we devote the following subsection to describing the formula that allows us to accomplish this update.

3.3.1 Sherman-Morrison formula

The Sherman-Morrison formula computes the sum an an invertible matrix $\boldsymbol{A}$ and a rank-one matrix $\boldsymbol{u}\boldsymbol{v}^T$.

$$(\boldsymbol{A} + \boldsymbol{u}\boldsymbol{v}^T)^{-1} = \boldsymbol{A}^{-1} - \frac{\boldsymbol{A}^{-}1\boldsymbol{u}\boldsymbol{v}^T\boldsymbol{A}^{-1}}{1 + \boldsymbol{v}\boldsymbol{A}^{-1}\boldsymbol{u}}. \tag{9}$$

If we already know the inverse of the matrix $\boldsymbol{A}$, then this formula allows us to correct the inverse by a rank-one update. This update takes $O(n^2)$ operations, as opposed to the $O(n^3)$ operations that are required to compute the inverse of A. We shall see that this formula is central computing the update step of quasi-Newton methods in an efficient manner.

3.3.2 The BFGS algorithm

In what follows, we provide the necessary information required to understand and implement the BFGS [5, 6] algorithm. The update step of the BFGS algorithm has the same structure of the other iterative methods we have discussed, only now the search direction is

$$\boldsymbol{p} = \boldsymbol{B}_k \nabla f_k$$

where $\boldsymbol{B}_k$ is an approximation of the Hessian, ∇f_k. The fundamental step in the BFGS algorithm is the update of $\boldsymbol{B}_k$; therefore, we provide its derivation here.

We can derive the BFGS Hessian update by using a quadratic model of f at iterate x_k

$$m_k(\boldsymbol{p}) = f_k + \nabla f_k^T \boldsymbol{p} + \frac{1}{2}\boldsymbol{p}^T \boldsymbol{B}_k \boldsymbol{p},$$

where $\boldsymbol{B}_k$ is symmetric positive definite. The minimizer $\boldsymbol{p}_k$ of this model is

$$\boldsymbol{p}_k = -\boldsymbol{B}_k^{-1} \nabla f_k,$$

where $\boldsymbol{p}_k$ is the search direction of (1). As in Newton's method, we avoid computing the inverse of $\boldsymbol{B}_k$ and instead solve the system

$$\boldsymbol{B}_k \boldsymbol{p}_k = -\nabla f_k.$$

In order to measure the curvature in the previous step we construct a quadratic model at the next iteration as

$$m_{k+1}(\boldsymbol{p}) = f_{k+1} + \nabla f_{k+1}^T \boldsymbol{p} + \frac{1}{2}\boldsymbol{p}^T \boldsymbol{B}_{k+1}\boldsymbol{p}.$$

For $\boldsymbol{B}_{k+1}$ to be a good approximation of the Hessian, we require that the gradient of m_{k+1} is a good approximation of the gradient of f at x_k and x_{k+1}, which we state as

$$\nabla m_{k+1}(-\alpha_k p_k) = \nabla f_{k+1} - \alpha_k \boldsymbol{B}_{k+1}\boldsymbol{p}_k = \nabla f_k.$$

This gives

$$\alpha_k \boldsymbol{B}_{k+1}\boldsymbol{p}_k = \nabla f_{k+1} - \nabla f_k.$$

Using the simplifying notation

$$\boldsymbol{s}_k = \boldsymbol{x}_{k+1} - \boldsymbol{x}_k = \alpha_k \boldsymbol{p}_k, \qquad \boldsymbol{y}_k = \nabla f_{k+1} - \nabla f_k,$$

we arrive at

$$\boldsymbol{B}_{k+1}\boldsymbol{s}_k = \boldsymbol{y}_k, \tag{10}$$

where $\boldsymbol{s}_k$ is the displacement between the current and previous iterate and $\boldsymbol{y}_k$ is the difference between the current and previous gradients. Equation (10) is called the secant equation. The BFGS update step is derived by imposing conditions on $\boldsymbol{B}_{k+1}^{-1}$ rather than $\boldsymbol{B}_{k+1}$ such that the secant equation becomes

$$\boldsymbol{B}_{k+1}^{-1}\boldsymbol{y}_k = s_k.$$

The secant equations necessitates that $\boldsymbol{B}_{k+1}^{-1}$ maps $\boldsymbol{s}_k$ into $\boldsymbol{y}_k$ which is occurs when

$$\boldsymbol{s}_k^T \boldsymbol{y}_k > 0, \tag{11}$$

which is called the *curvature condition.* If the curvature condition is met the secant equation has an infinite number of solutions for $\boldsymbol{B}_{k+1}$ [18]. To make the solution to unique, we stipulate that the difference between $\boldsymbol{B}_{k+1}$ and $\boldsymbol{B}_k$ is minimized under the Frobenius norm. Stated mathematically, we say

$$\min_{\boldsymbol{B}^{-1}} ||\boldsymbol{B}^{-1} - \boldsymbol{B}_k^{-1}||,$$

subject to the constraints that (11) and $\boldsymbol{B}^{-1} = (\boldsymbol{B}^{-1})^T$. The unique solution to (11) is given by

$$\boldsymbol{B}_{k+1}^{-1} = (\boldsymbol{I} - \rho_k \boldsymbol{s}_k \boldsymbol{y}_k^T)\boldsymbol{B}_k^{-1}(\boldsymbol{I} - \rho_k \boldsymbol{s}_k \boldsymbol{y}_k^T) + \rho_k \boldsymbol{s}_k \boldsymbol{y}_k^T, \tag{12}$$

where $\rho_k = \frac{1}{\boldsymbol{y}_k^T \boldsymbol{s}_k}$. Finally, using the Sherman-Morrison formula (9) this expression becomes

$$\boldsymbol{B}_{k+1} = \boldsymbol{B}_k - \frac{\boldsymbol{B}_k \boldsymbol{s}_k \boldsymbol{s}_k^T \boldsymbol{B}_k}{s_k^T \boldsymbol{B}_k \boldsymbol{s}_k} + \frac{\boldsymbol{y}_k \boldsymbol{y}_k^T}{\boldsymbol{y}_k^T \boldsymbol{s}_k}.$$

The second and third terms in this expression are rank-one matrices, therefore, $\boldsymbol{B}_k$ undergoes a rank-two update. The reason that we update the inverse of $\boldsymbol{B}_k^{-1}$ instead of recalculating the inverse is that it only requires $O(n^2)$ operations rather than the $O(n^3)$ operations that would be required to recompute the inverse. We provide the pseudo-code for BFGS in algorithm (3).

3.3.3 Properties of BFGS

The BFGS algorithm enjoys super-linear convergence [18]. However, this fast convergence comes with the extra cost of updating the Hessian approximation with the Sherman-Morrison formula (9). Though the Hessian approximation saves computations, it is still an $O(n^2)$ operation that we perform at each iterate. Therefore, we might predict that as n grows, the expense of this approximation outweighs the benefit of its fast convergence.

One advantage of the BFGS algorithm is that if $\boldsymbol{B}_k$ is a poor approximation of the local curvature, the Hessian update tends to correct itself in future iterates. However, this self-correcting property is only exhibited by the method if we use a line search with Wolfe conditions (4) to choose the step size [18].

Algorithm 3 BFGS pseudo-code

```
Choose $x_0$
Choose tolerance
Choose $H_0$
k = 0
**while** $||\nabla f_(\boldsymbol{x}_k)|| >$ tolerance **do**
    $\boldsymbol{p} = -\boldsymbol{B}_k^{-1}\nabla f_(\boldsymbol{x}_k)$
    find $\alpha_k$ with a line search
    $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \alpha_k \boldsymbol{p}_k$
    $\boldsymbol{s}_k = \boldsymbol{x}_{k+1} - \boldsymbol{x}_k$
    $\boldsymbol{y}_k = \nabla f_(\boldsymbol{x}_{k+1}) - \nabla f_(\boldsymbol{x}_k)$
    update $\boldsymbol{B}_k^{-1}$ using (12)
    $k = k + 1$
**end while**
```

4 Optimization problems

To perform a fair and useful benchmarking experiment, we need to test algorithms on a batch of test problems that are representative of the kinds of issues we would face in practical optimization problems. To this end, the COCO platform provides the Black-Box Optimization Benchmarking (BBOB) suite of 24 noiseless test functions [15] that the COCO developers broadly classified into five categories:

- separable functions
- functions with good or moderate conditioning
- unimodal functions with poor conditioning
- multi-modal functions with an adequate global structure
- multi-modal functions with a weak global structure

We describe the characteristics of these problem classes in the following sections. The global optimum is in $[-4, 4]^d$ for the majority of the test problems. In the following sections, we provide a general description of each of these 5 test function groups. All test functions are defined everywhere in $\mathbb{R}^d$, and their global optimum is in $[-5, 5]^d$.

4.1 Separable functions

Separable functions are functions that can be expressed as a composition of single variable functions. This implies solving a separable problem can be broken down into d one-dimensional optimization problems where each dimension can be optimize individually. For instance, the sphere function is separable

$$f(\boldsymbol{x}) = ||\boldsymbol{z}||^2 + f_{\text{opt}}, \tag{13}$$

where the location of the optimum is $\boldsymbol{x}^{\text{opt}}$, $f_{\text{opt}} = f(\boldsymbol{x}^{\text{opt}})$, and $\boldsymbol{z} = \boldsymbol{x} - \boldsymbol{x}^{\text{opt}}$. Presumably, the group of separable functions should be the easiest to solve.

4.2 Functions with reasonable, moderate, and poor conditioning

Conditioning is an essential property of numerical problems. Generally, a problem is well-conditioned if small perturbations in the data do not have a strong effect on the problem's solution. Conversely, the solution to ill-conditioned problems can be significantly affected by small changes in the data. Poor conditioning can introduce numerical errors [19] as well as stall the convergence [18] of many algorithms. The BBOB suite includes problems with reasonable, moderate, and poor conditioning.

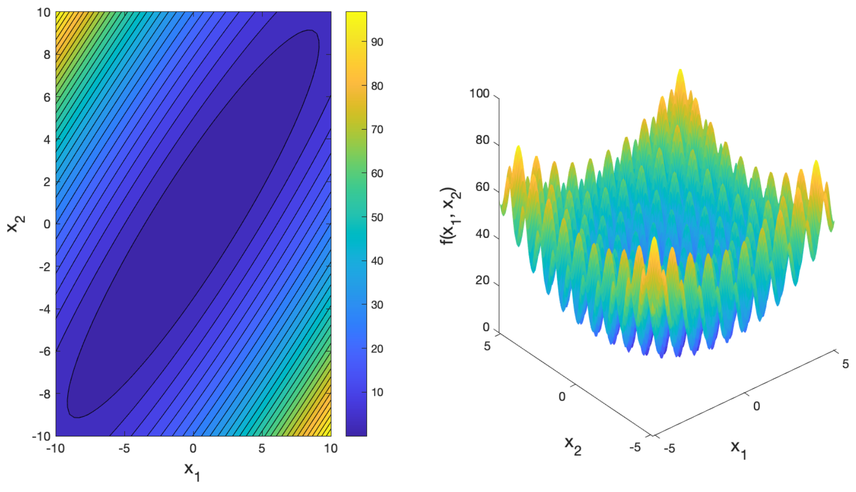


Figure 3: An ill-conditioned function (left) and a highly multimodal function (right). Notice that the contours of the left plot are highly ellipsoidal and thus the gradient would be a poor search direction. In the right plot, most descent algorithms would git immediate stuck in one of the many local minimums.

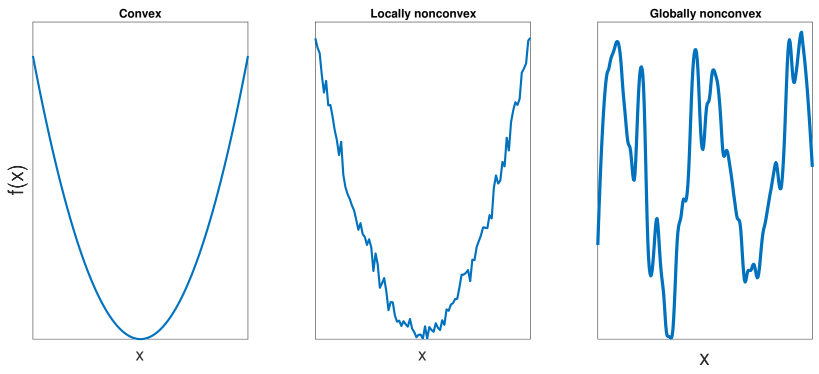


Figure 4: The robustness of an algorithm can be determined by testing it on problems with local and global structure, as well as problems without structure.

4.3 Multi-modal functions with adequate global structure

In the context of optimization, the modality refers to the number of minima the objective function has. For example, the sphere function (13) is unimodal, whereas the Rastrigin function

$$f(x) = 10\left(d - \sum_{i=1}^{d} \cos(2\pi z_i) + ||\boldsymbol{z}||^2 + f_{\text{opt}}\right). \tag{14}$$

has 10^d local minima.

One of the general classes of problems in the BBOB suite is multi-modal functions with an adequate global structure. These functions tend to have a convex global structure, but their local structure is more irregular (see figure 4).

4.4 Multi-modal functions with weak global structure

The last class of problems in the BBOB suite is multi-modal functions with a weak global structure. In this case, the problems are nonconvex and irregular on both a local and global level, though some semblance of global convexity is often maintained.

5 Optimization benchmark software

The Comparing Continuous Optimizers (COCO, [13]) is a black-box benchmarking software for numerical optimization algorithms that provides automated testing of algorithms on suites of test problems and produces evaluation metrics that allow the comparison of methods.

5.1 Terminology

We now present the terminology for the COCO platform that we use throughout the remainder of this work.

- **function**: the objective function with input space $n \geq 2$.
- **instance**: a function that has been translated or shifted by some transformation. The COCO platform tests algorithms on multiple randomized instances of a function to make performance metrics more robust [15]
- **problem**: a particular function instance. A problem is solved when an algorithm has reached the smallest target value (e.g., 10^-8)
- **target value**: particular f-values that algorithms are compared at for a given problem. Typically, smaller target values are harder to reach.
- **runtime**: the number of function evaluations required to reach a target value for a given problem instance

5.2 Evaluation metrics

The simplest way to compare methods is to compare the number of function, gradient, or Hessian evaluations in their search. Additionally, algorithms are only comparable if they reach similar values of f before terminating. Hence, the need for standardized ways to compare algorithms.

5.2.1 Fixed-target and fixed-cost measures

Fixed-target measures compare how long each algorithm takes to reach a specific target value for a function. In contrast, fixed-cost measures allow each algorithm to have a constant number of function evaluations. Then the lowest function value reached by each algorithm is compared (see figure 5.2.1).

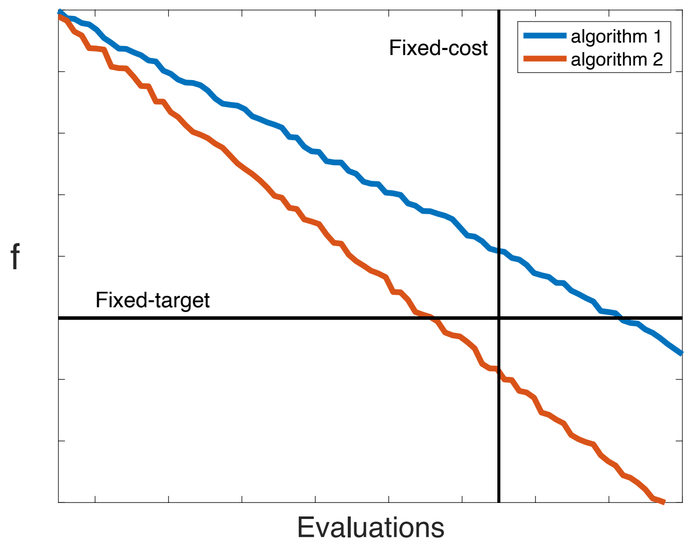


Figure 5: Comparison of fixed-cost and fixed-target scenarios. For fixed-cost measures, algorithms are compared based on the function value reached after a fixed number of function evaluations. For fixed-target measures, algorithms are compared after they reach a fixed target value of the function. Algorithms are compared at the points where they intersection the fixed-cost or fix-value line.

The COCO developers advocate fixed-target experiments [13], and all evaluation metrics provided by the COCO platform are fixed-target metrics.

5.2.2 Runtime

The single performance metric that COCO provides (and aggregates into plots and tables) is the algorithm runtime, as measured by function evaluations [12]. The developers of COCO advocate for this runtime as their sole performance metric because it is

- we typically generate higher-order information (gradients and Hessians) from function evaluations
- independent of the hardware and software used for experiments
- easily interpretable
- independent of the function we are evaluating
- easy to aggregated into summary statistics

Though simple, runtime does not take into account other overheads that the algorithm may incur beyond function evaluations. Thus, a major assumption is that the primary computational expense of an optimization algorithm is the number of function evaluations.

5.2.3 Expected running time

The COCO library also provides the expected running time (ERT, CITE Price 1997 and Auger 2005). ERT is a fixed-target measure and is defined as the expected number of evaluations that are required to reach a target function value for a particular problem. ERT is calculated as

$$\begin{aligned} ERT(f_{target}) &= RT_s + \frac{1-p_s}{p_s} RT_{us} \\ &= \frac{p_s RT_s + (1-p_s) RT_{us}}{p_s} \\ &= \frac{\text{number of } FE_s(f_{best} \text{ in all trials} \geq f_{target})}{\text{number of successful trials}} \end{aligned}$$

where RT_s is successful trials and RT_{us} is unsuccessful trials. A trial is successful if it reaches f_{target} and f_{best} is the lowest value of f reached. The fraction of successful trials is p_s. ERT is dependent on the termination criteria of an algorithm. ERT is a single measure from a sample set of N_{trial} optimization runs. To get an estimate of the uncertainty of the ERT of an algorithm, COCO uses bootstrapping [9] to generate dispersion measures.

5.2.4 Problem instances and independent restarts

To increase the robustness of comparison metrics, COCO creates variations of a particular test function by applying transformations (such as rotations or shifts) to the function. These function varients are called problem instances. In doing so, summary statistics can be computed by aggregating data from each algorithm run on each problem instance.

In practice, all optimization algorithms must terminate. However, if an algorithm that we are benchmarking terminates, we can restart the algorithm from an independent point in the domain to try and reach a lower function value. Independent restarts to not affect the runtime calculation of an algorithm for a given problem instance because we only compare the number of function evaluations it takes algorithms to reach the same target value. However, independent restarts do increase the chance of reaching more target values and therefore allow us to compare different algorithms more reliably.

5.2.5 Empirical cumulative distribution functions

An empirical cumulative distribution function (ECDF) is a distribution function that is associated with an empirical measure of a sample. ECDFs are step functions, and for each N data points in the sample, the function increases by $\frac{1}{N}$. In the context of the COCO software, we use ECDFs to report the proportion of problems solved within a specified number of function evaluations, where we plot the total budget used on the x-axis. Each value on the x-axis is the fraction of runtimes that are less than or equal to that value. After m trial runs of an algorithm on a set number of instances of a test problem, the software records the number of function evaluations needed to reach each target value. Then, the number of targets reached for a given run is recorded (see figure 5.2.5 for an illustration of this process).

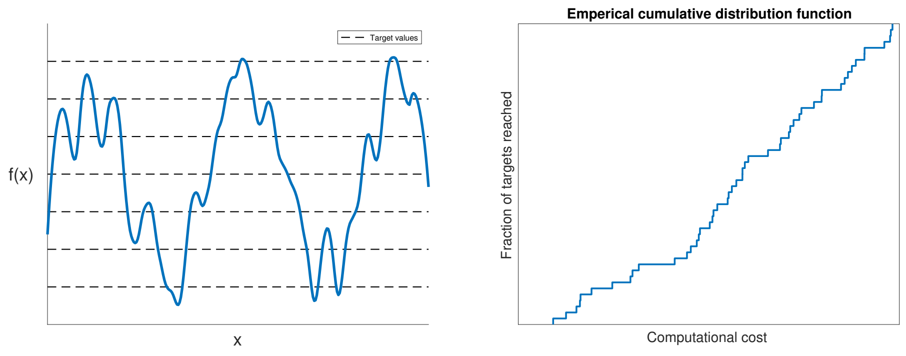


Figure 6: Am illustrative example of how an empirical cumulative distribution function (ECDF) is used with the COCO software to tabulate the number of function evaluations required to reach target values for a given algorithm. ECDFs tell us the relationship between computational cost (number of function evaluations) and performance (fraction of targets reached). The results are aggregated across each function, dimension, and problem instance.

6 Benchmark experiments

We now provide the specific setup we used for the benchmark experiments. Table 6.1 has a full list of parameters for each of the algorithms used in the experiments.

The benchmarking software we use to compare algorithms is the Comparing Continuous Optimizers (COCO, [13]) library. We compare the performance of gradient descent and BFGS on the single objective suite of small scale (1-40 dimensions) test functions. We also benchmarked a random search of the domain to establish a baseline. We conducted the benchmarking experiments in MATLAB ver. R2019b. The gradient descent, BFGS, cubic polynomial line search algorithms were taken from [16]. We used our implementation of finite differences.

6.1 Cubic polynomial line search parameters

The step length, α, for both the gradient descent and quasi-Newton method was chosen via a cubic polynomial line search with sufficient decrease (Armijo) condition. As recommended in [16], to prevent extremely long step lengths, the initial α for each line search was determined by

$$\alpha_0 = \min\left(1, \frac{100}{1 + ||\nabla f(x)||}\right).$$

This check also protects the step length against poor scaling. Additionally, we used the safeguarding procedure described in section 2.3.3, but limited the number of reductions in α to 10 reductions, and terminated the run otherwise.

Algorithm	**Parameter**	**Description**	**Value**
Finite difference	h	step size between each $\boldsymbol{x}_i$	10^{-4}
Cubic line search	ρ_{high}	upper bound for α	0.5
	ρ_{low}	lower bound for α	0.1
	$\alpha_{\max}$	maximum allowed α	1
	c_1	algorithmic parameter for (4)	10^{-4}
	maxback	max number of backtracks	10
Gradient descent	tolerance	termination criteria	10^{-5}
	max iterations	termination criteria	d $\times$ budget multiplier
	$\boldsymbol{x}_0$	initial solution	random vector
BFGS	tolerance	termination criteria	10^{-5}
	max iterations	termination criteria	d $\times$ budget multiplier
	$\boldsymbol{x}_0$	initial solution	random vector
	$\boldsymbol{H}_0$	initial Hessian	$\boldsymbol{I}$

Table 1: Parameter values for each algorithm used in the experiments. Budget multiplier is a factor that increases the allowed number of function evaluations per run and d is the problem dimension.

6.2 COCO software parameters

We now list out the parameters used for the experiments that are specific to the COCO platform and are independent of the specific optimization algorithm that we benchmarked. The function evaluation budget that each optimization algorithm was given for each problem instance was (budget multiplier) $\times\, d - (CFE)$, where d is the dimension of the problem, the budget multiplier was 100, and CFE is the cumulative function evaluations used so far before a restart. There were 51 function target values for each test problem chosen between 10^2 and 10^{-8}.

We benchmarked each optimization algorithm for dimensions 2, 3, 5, 10, 20, and 40 for each of the 24 functions in the COCO noiseless test function suite. The number of instances of each test function was 15. A total of 2160 problem instances were tested for each algorithm. On each instance of the 24 test functions, there were 51 function target values evenly distributed on a log scale between 10^2 and 10^{-8}. The number of allowed independent algorithm restarts was 10^9.

7 COCO benchmark results

We present the results of our experiments on the 24 objective functions in the Black-Box Optimization-Benchmarking (BBOB) test suite.

7.1 Algorithm performance aggregated by problem dimension

When we assimilated the performance of each algorithm for all 24 test functions across all functions dimensions tested (2, 3, 5, 10, 20, and 40) and 15 problem instances, BFGS reached a larger fraction of the 51 target values than gradient descent for each dimension at the maximum number of function evaluations (figure 8). However, a larger fraction of gradient descent runs reached smaller target values in fewer function evaluations compared to BFGS (figure 3).

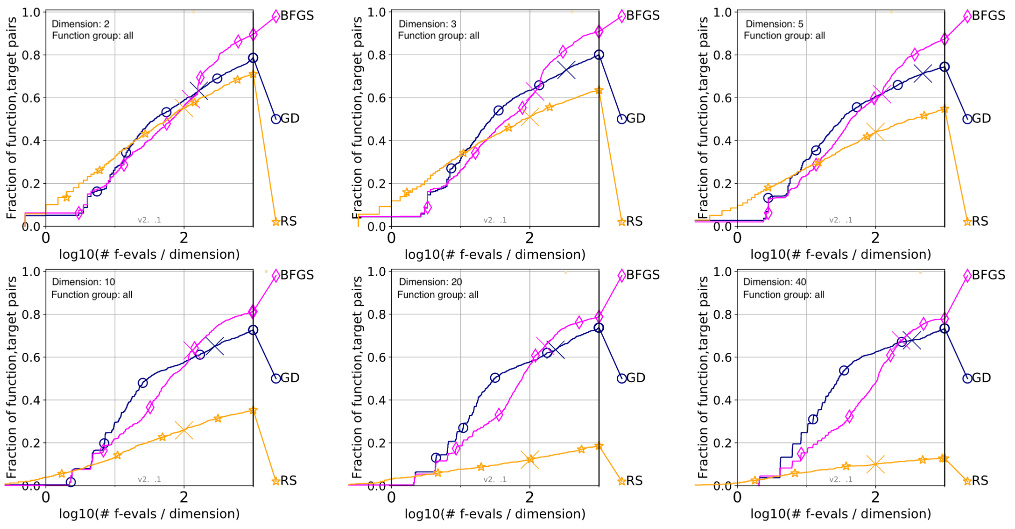
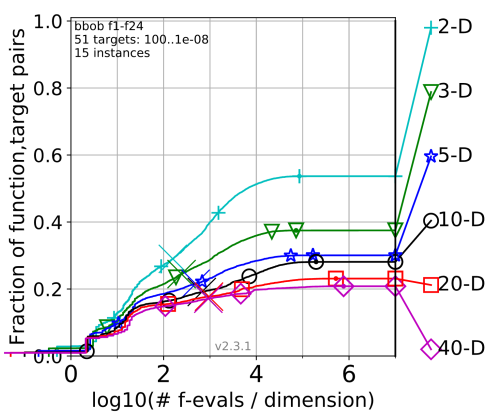


Figure 7: Bootstrapped empirical cumulative distribution functions of aggregated results across all test problems, problem instances, and dimensions: 2, 3, 5, 10, 20, and 40. Notice that gradient descent reaches a larger fraction of target values in fewer function evaluations than BFGS, but with more function evaluations, BFGS reaches a larger fraction of target values



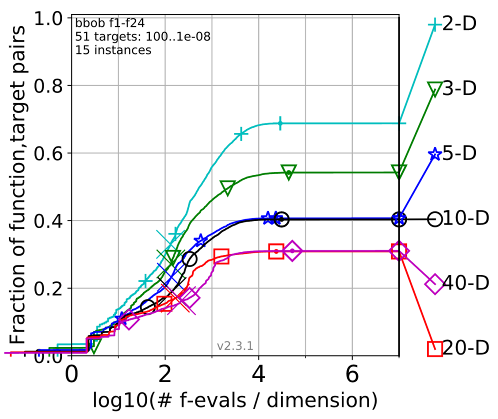


Figure 8: Bootstrapped empirical cumulative distribution functions of the runtime (function evaluations divided by dimension) versus fraction of 51 logspaced function target values (from 10^2 to 10^{-8}) reached for gradient descent and BFGS for all 24 test functions in the BBOB test suite and all dimensions. Notice that the fraction of function target values reached decays with the dimension of the problem.

7.2 Algorithm performance aggregated by test function type

BFGS outperformed gradient descent for the smallest target value reached for each problem group except the moderate conditioning group, where they reached a roughly equal fraction of target values. Additionally, gradient descent outperformed BFGS in the multimodel with a global structure problem group (see figure 9). The trend of gradient descent reaching a higher fraction of target values when the number of function evaluations is moderate or low held for each function group. BFGS did significantly exceed gradient descent on problems with poor conditioning. However, for the other function groups, BFGS reached a roughly equal number of target values as gradient descent at the maximum number of function evaluations, except on multimodel problems with global structure, where gradient descent outperformed BFGS.

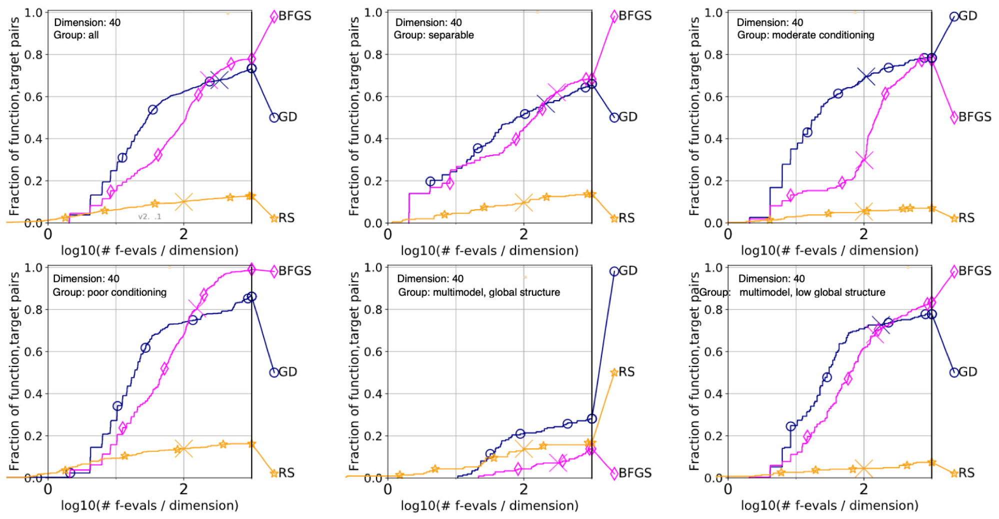


Figure 9: All 24 test functions in the small-scale noiseless COCO test suite divided into six groups based on the properties of the test problem. The dimension of each test function was 40. Notice the performance of the algorithms depends on the class of functions

8 Discussion

The results of our benchmarking experiments show that there are quantitative differences between the performance of the gradient descent and BFGS methods. We found that The performance of the methods varied across dimensions, problem class, efficiency, and accuracy of the final solution. The following subsections discuss the details of these results and offer recommendations for researchers when choosing between optimization methods.

8.1 Comparison of gradient descent and BFGS

Across all dimensions and all problem groups, BFGS achieved a higher level of accuracy than gradient descent, for larger function evaluations budgets 3. However, gradient descent reached less accurate function targets more often in fewer function evaluations. This result implies that, if we have a limited function evaluation budget for a particular application, gradient descent will perform better by providing a more accurate solution for a smaller number of evaluations.

Hence, we recommend that when researchers must choose between gradient descent and BFGS, if they require and can afford highly accurate solutions, use BFGS. However, if we have a constraint on the number of function evaluations and only require a decent solution, use gradient descent. For example, in many statistical and machine learning applications, function evaluations can become extremely expensive when the number of model parameters grows large [4]. Furthermore, we often do not want highly accurate solutions because they can lead to model overfitting [27], so gradient descent would be preferred over BFGS. This trend of cheap approximate solutions for gradient descent is likely the reason that it has risen in popularity for statistical and machine learning

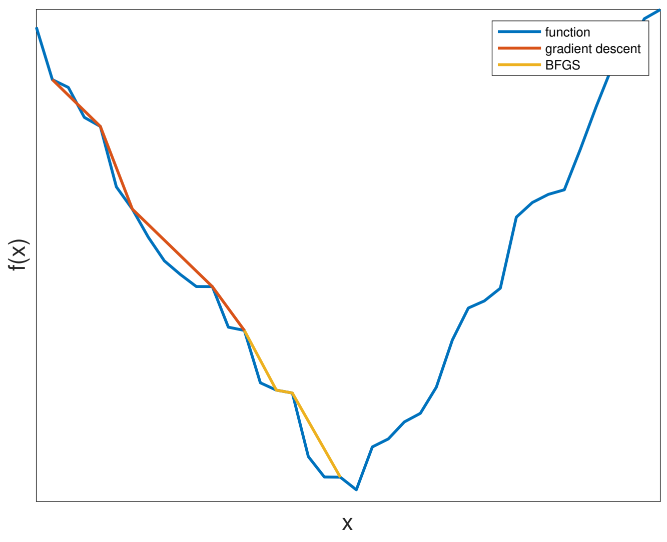


Figure 10: A proposed hybrid gradient descent-BFGS scheme for decreasing the number of function evaluations required to reach accurate solutions by starting an optimization run with gradient descent then switching to BFGS when near a minimum.

applications recently.

This efficiency-accuracy trade-off between gradient descent and BFGS suggests an alternative strategy for obtaining an accurate solution in fewer function evaluations. That is, begin a run with gradient descent for an approximate solution, then switch to BFGS afterward by starting from the final $\boldsymbol{x}_k$ of the sequence of gradient descent iterates. This would provide a more accurate solution while saving computations by not using BFGS for the whole run (see figure 10 for an illustrative example).

A general, and unsurprising, result was that a smaller fraction of each algorithm instances reached fewer target values as the dimension of the problem increased (figures 8 and 3). This outcome was particularly noticeable for the random search algorithm. What was surprising, however, was that random search performed almost as well as the more sophisticated algorithms when the problem dimension was low (e.g. 2 or 3, see figure 3). Nevertheless, as the dimension grew, and consequently, the search space, we can see that the strategy of choosing a random search direction broke down rapidly.

8.2 Algorithm performance on function groups

Gradient descent and BFGS performed best on the separable functions, like the sphere function, which generally only had one or a few minima. In contrast, their worst performance was on multimodel functions such as the Rastrigin function with its abundance of local minima (figure 9). This behavior is expected for these descent algorithms because they are designed to converge if an iterate is in the neighborhood of a local minimum. Additionally, the stopping criteria (terminate if tolerance $< ||\nabla f||$) that we used for both algorithms guarantees that, if it finds a local minimizer, it will not explore beyond it.

Gradient descent reached considerably more target values than BFGS on the multimodel functions with global structure. The likely reason is that when the BFGS iterate begins to approach a local minimum, because of its use of the Hessian approximation to choose the search direction and step size, it follows the curvature of f more accurately than gradient descent and is thus pulled into the local minimum more rapidly. Though the gradient descent iterate still finds local minima readily, it is indifferent to the local curvature in f, since it neglects second-order information in its update step. Therefore it is more likely not to be as attracted towards local minima as the BFGS iterate is and may even pass by them on its way to a lower minimum.

For the group of problems with poor conditioning, BFGS performed far better than gradient descent, nearly reaching all target function values for the given budget 9. When the conditioning was moderate, the smallest function target value reached was similar for both gradient descent and BFGS. However, gradient descent reached all other (larger) targets in fewer function evaluations. This result suggests that for problems with moderate conditioning of up to at least 40 dimensions, gradient descent may be preferable to BFGS because it is less computationally expensive.

8.3 Random search and the explore-exploit trade-off

Optimization can be roughly divided into two phases, an exploration phase and an exploitation phase [7]. The goal of the exploration phase is to search for novel regions in the search space to find promising values of f. In contrast, the goal of the exploitation phase is to search in the neighborhood of the promising regions found by the exploration phase to find locally optimal values of f. A trade-off arises because we must divide our budget of function evaluations between exploring or exploiting. If we spend too much of our budget in the exploration phase, we often get diminishing returns on promising regions of the search space. However, if we do not spend enough of our budget in the exploration phase when we start the exploitation phase, we will likely converge to a poor local solution that is far away from the global optimum.

We can see the effects of the explore-exploit trade-off in expected runtime tables in the appendix A. Random search often hit the large (and presumably easy) target values in far shorter average run times than BFGS and gradient descent. The likely reason is that in the exploration phase when we only desire a course grain estimate of the search space, we gain more benefit from the cheapness of randomly sampling the search space. Again, as in the ECDF results, we can also see that in high dimensions, we get less payoff from random search because the search space explodes exponentially due to the ”curse of dimensionally” [2].

9 Conclusion

By benchmarking the quintessential iterative optimization methods, namely gradient descent and BFGS, we were able to provide empirical data on what classes of problems the methods succeed and failed on as well as how they performed in comparison to one another. Generally, BFGS obtained highly accurate solutions but required more function evaluations, whereas gradient descent found adequate solutions in fewer function evaluations, but often failed to obtain highly accurate solutions. In light of this finding, we proposed strategies for finding more accurate solutions in fewer function evaluations by using a hybrid gradient descent-BFGS scheme. This work demonstrates that benchmarking optimization algorithms can provide actionable insights for researchers when they must choose between iterative optimization algorithms.

A Appendix

A.1 Notation

Symbol	Description
f	The objective function we wish to minimize
n	Dimension of f
$\boldsymbol{x}$	Vector in $\mathbb{R}^n$ we optimize to minimize f
$\boldsymbol{x}_0$	Initial solution for an iterative algorithm
$\boldsymbol{x}^*$	a local minimizer of f
k	Iteration counter
∇f	Gradient of the objective function
$\nabla^2 f$	Hessian of the objective function
$\boldsymbol{p}$	search direction of an iterate
α	Step length of an iterate
α^*	Minimum of a line search
ξ	Polynomial model used in line search
ξ'	Derivative of ξ
ρ	Factor that reduces α in a line search
ρ_{high}	Upper bound for α in cubic line search reduction
ρ_{low}	Lower bound for α in cubic line search reduction
$\alpha_{\max}$	Maximum allowed α for line search
c_1	Algorithmic parameter for (4)
h	Step size between each $\boldsymbol{x}_i$ in finite difference
$\boldsymbol{I}$	The identity matrix
$\boldsymbol{A}$	An invertible matrix
$\boldsymbol{s}_k$	Difference between $\boldsymbol{x}_{k+1}$ and $\boldsymbol{x}_k$
$\boldsymbol{y}_k$	Difference between the current and next gradient
$\boldsymbol{B}$	SPD matrix that approximates the Hessian

Table 2: Symbol definitions used in this thesis.

A.2 COCO supporting information

For a full list of all of the COCO problems in the BBOB test suite visit:

https://coco.gforge.inria.fr/downloads/download16.00/bbobdocfunctionsdef.pdf

For the details of the COCO platform visit:

http://coco.lri.fr/downloads/download15.03/bbobdocexperiment.pdf

A.3 Average runtime tables

#FEs/D	0.5	1.2	3	10	50	#succ
f6, 40-D	*6.3e+5*:50	*4.0e+5*:82	*4.0e+4*:127	*4.0e+2*:734	*6.3e+1*:2121	15/15
GD	3.7 (0.8)	2.7 (1)	**2.5** (0.7)	**3.2** (2)$^{\star 3}$	**5.2** (2)	15/15
BFGS	**3.3** (1)	**2.4** (1)	3.4 (1)	13 (9)	6.6 (3)	15/15
RS	20 (52)	111 (150)	∞	∞	∞*4000*	0/15

#FEs/D	0.5	1.2	3	10	50	#succ
f7, 40-D	*1.6e+3*:35	*1.0e+3*:106	*6.3e+2*:165	*2.5e+2*:489	*2.5e+1*:2987	15/15
GD	4649 (3716)	∞	∞	∞	∞*4195*	0/15
BFGS	1085 (1620)	∞	∞	∞	∞*4068*	0/15
RS	**34** (57)$^{\star 3}$	**272** (205)$^{\star 2}$	∞	∞	∞*4000*	0/15

#FEs/D	0.5	1.2	3	10	50	#succ
f8, 40-D	*1.0e+5*:85	*6.3e+4*:111	*4.0e+4*:125	*2.5e+3*:430	*6.3e+1*:2106	15/15
GD	**1.8** (0)	**1.4** (0)	**1.3** (0.2)$^{\star 2}$	**0.78** (0.1)$^{\star 4}$	11 (0.8)	15/15
BFGS	3.7 (6)	8.7 (10)	14 (12)	10 (1)	**6.6** (3)	15/15
RS	∞	∞	∞	∞	∞*4000*	0/15

#FEs/D	0.5	1.2	3	10	50	#succ
f9, 40-D	*2.5e+2*:676	*1.6e+2*:865	*1.0e+2*:1397	*6.3e+1*:1896	*4.0e+1*:2180	15/15
GD	**1.1** (0.1)$^{\star 4}$	**1.1** (0.1)$^{\star 4}$	**1.6** (4)$^{\star 2}$	7.6 (27)	6.8 (23)	15/15
BFGS	8.2 (4)	7.3 (3)	4.9 (1)	**4.4** (3)	**4.2** (2)	15/15
RS	∞	∞	∞	∞	∞*4000*	0/15

#FEs/D	0.5	1.2	3	10	50	#succ
f10, 40-D	*1.0e+7*:44	*6.3e+6*:80	*2.5e+6*:126	*2.5e+5*:408	*6.3e+3*:2376	15/15
GD	3.7 (0.9)	**2.2** (0.5)	**2.7** (2)	**4.7** (2)	52 (29)	15/15
BFGS	24 (30)	14 (15)	12 (11)	7.6 (7)	**3.1** (1)	15/15
RS	**3.1** (5)	18 (19)	463 (524)	∞	∞*4000*	0/15

#FEs/D	0.5	1.2	3	10	50	#succ
f11, 40-D	*1.0e+4*:22	*2.5e+3*:52	*2.5e+2*:432	*1.6e+2*:887	*1.6e+1*:2204	15/15
GD	6.9 (2)	3.5 (2)	∞	∞	∞*4e5*	0/15
BFGS	12 (7)	6.2 (3)	**3.6** (0.6)$^{\star 3}$	**2.7** (3)$^{\star 3}$	**13** (18)	3/15
RS	**1.9** (2)$^{\star 3}$	**2.2** (2)	∞	∞	∞*4000*	0/15

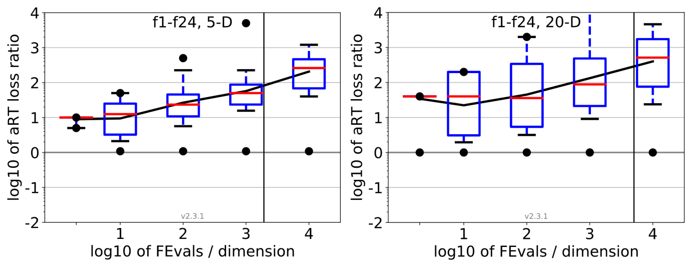
#FEs/D	0.5	1.2	3	10	50	#succ
f12, 40-D	*2.5e+8*:54	*1.6e+8*:218	*1.0e+8*:284	*1.0e+7*:424	*4.0e+1*:2479	15/15
GD	**5.8** (4)$^{\star 2}$	**1.6** (0.5)$^{\star 2}$	**1.3** (0.7)$^{\star 4}$	**1.1** (0.5)$^{\star 4}$	53 (216)	11/15
BFGS	106 (100)	29 (23)	24 (16)	20 (10)	**14** (15)	7/15
RS	∞	∞	∞	∞	∞*4000*	0/15

#FEs/D	0.5	1.2	3	10	50	#succ
f13, 40-D	*2.5e+3*:85	*1.6e+3*:121	*1.6e+3*:121	*6.3e+1*:429	*1.0e+1*:2029	15/15
GD	**2.0** (0)	**1.4** (0)$^{\star 4}$	**1.4** (0)$^{\star 4}$	**2.0** (0.3)$^{\star 4}$	**0.81** (0.6)$^{\star 2}$	15/15
BFGS	2.9 (1)	3.4 (2)	3.4 (1)	7.7 (0.4)	2.4 (0.2)	15/15
RS	∞	∞	∞	∞	∞*4000*	0/15

#FEs/D	0.5	1.2	3	10	50	#succ
f14, 40-D	*6.3e+1*:34	*4.0e+1*:137	*2.5e+1*:176	*4.0e+0*:438	*1.0e-3*:2207	15/15
GD	5.4 (1)	**1.6** (0.3)	**1.6** (0.2)	1.7 (0.5)	21 (4)	15/15
BFGS	**5.0** (0.0)	1.7 (0.3)	1.6 (0.2)	**1.3** (0.3)*	**1.2** (0.1)$^{\star 4}$	15/15
RS	87 (140)	∞	∞	∞	∞*4000*	0/15

Table 3: Average runtimes of gradient descent (GD), BFGS, and random search (RS) to reach 5 equidistant target values on a log scale for functions with poor conditions (f6-f9) and multimodal with global structure (f10-f14). #succ is the number of runs that reached the smallest target value. #FEs/D is the number of function evaluations divided by the dimension of the problem. The top row are reference values of the average runtime of the best algorithm at the BOBB 2009 conference [14].

Gradient descent



BFGS

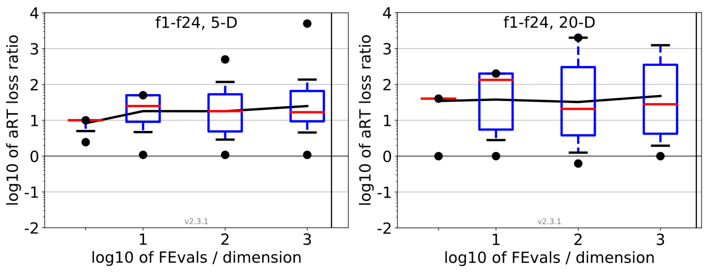


Figure 11: BFGS and Gradient descent average runtime (aRT) loss ratio over function evalutations (FEvals/dimension) on a log scale. The loss ratio is the ratio between the number of function evaluations for an algorithm and the ERT of the best algorithm from the BBOB 2009 conference [14]. The trend line is the geometric mean and the vertical line is the maximum number of function evaluations in a single run. Notice that the loss ratio increases with the number of function evaluations for gradient descent, where as it is relatively constant for BFGS.

References

[1] Vahid Beiranvand, Warren Hare, and Yves Lucet. Best practices for comparing optimization algorithms. *Optimization and Engineering*, 18(4):815–848, 2017.

[2] Richard Bellman. Dynamic programming. *Science*, 153(3731):34–37, 1966.

[3] Stephen C Billups, Steven P Dirkse, and Michael C Ferris. A comparison of large scale mixed complementarity problem solvers. *Computational Optimization and Applications*, 7(1):3–25, 1997.

[4] Léon Bottou, Frank E Curtis, and Jorge Nocedal. Optimization methods for large-scale machine learning. *Siam Review*, 60(2):223–311, 2018.

[5] Charles G Broyden. The convergence of a class of double-rank minimization algorithms: 2. the new algorithm. *IMA journal of applied mathematics*, 6(3):222–231, 1970.

[6] Charles George Broyden. The convergence of a class of double-rank minimization algorithms 1. general considerations. *IMA Journal of Applied Mathematics*, 6(1):76–90, 1970.

[7] Matej Črepinšek, Shih-Hsi Liu, and Marjan Mernik. Exploration and exploitation in evolutionary algorithms: A survey. *ACM computing surveys (CSUR)*, 45(3):1–33, 2013.

[8] Ferenc Domes, Martin Fuchs, Hermann Schichl, and Arnold Neumaier. The optimization test environment. *Optimization and Engineering*, 15(2):443–468, 2014.

[9] B. Efron and R.J. Tibshirani. *An Introduction to the Bootstrap*. Chapman & Hall/CRC Monographs on Statistics & Applied Probability. Taylor & Francis, 1994.

[10] Bengt Fornberg. Generation of finite difference formulas on arbitrarily spaced grids. *Mathematics of computation*, 51(184):699–706, 1988.

[11] Nicholas IM Gould, Dominique Orban, and Philippe L Toint. Cutest: a constrained and unconstrained testing environment with safe threads for mathematical optimization. *Computational Optimization and Applications*, 60(3):545–557, 2015.

[12] Nikolaus Hansen, Anne Auger, Dimo Brockhoff, Dejan Tušar, and Tea Tušar. Coco: Performance assessment. *arXiv preprint arXiv:1605.03560*, 2016.

[13] Nikolaus Hansen, Anne Auger, Olaf Mersmann, Tea Tusar, and Dimo Brockhoff. Coco: A platform for comparing continuous optimizers in a black-box setting. *arXiv preprint arXiv:1603.08785*, 2016.

[14] Nikolaus Hansen, Anne Auger, Raymond Ros, Steffen Finck, and Petr Pošík. Comparing results of 31 algorithms from the black-box optimization benchmarking bbob-2009. In *Proceedings of the 12th Annual Conference Companion on Genetic and Evolutionary Computation*, GECCO '10, page 1689–1696, New York, NY, USA, 2010. Association for Computing Machinery.

[15] Nikolaus Hansen, Steffen Finck, Raymond Ros, and Anne Auger. Real-parameter black-box optimization benchmarking 2009: Noiseless functions definitions, 2009.

[16] Carl T Kelley. *Iterative methods for optimization*. SIAM, 1999.

[17] David G Luenberger, Yinyu Ye, et al. *Linear and nonlinear programming*, volume 2. Springer, 1984.

[18] Jorge Nocedal and Stephen Wright. *Numerical optimization*. Springer Science & Business Media, 2006.

[19] Alfio Quarteroni, Riccardo Sacco, and Fausto Saleri. *Numerical mathematics*, volume 37. Springer Science & Business Media, 2010.

[20] Luis Miguel Rios and Nikolaos V Sahinidis. Derivative-free optimization: a review of algorithms and comparison of software implementations. *Journal of Global Optimization*, 56(3):1247–1293, 2013.

[21] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[22] Jan A Snyman. *Practical mathematical optimization*. Springer, 2005.

[23] Suvrit Sra, Sebastian Nowozin, and Stephen J Wright. *Optimization for machine learning*. Mit Press, 2012.

[24] Richard Sutton. Two problems with back propagation and other steepest descent learning procedures for networks. In *Proceedings of the Eighth*

Annual Conference of the Cognitive Science Society, 1986, pages 823–832, 1986.

[25] Philip Wolfe. Convergence Conditions for Ascent Methods. *SIAM Review*, 11(2):226–235, 1969. Publisher: Society for Industrial and Applied Mathematics.

[26] David H Wolpert and William G Macready. No free lunch theorems for optimization. *IEEE transactions on evolutionary computation*, 1(1):67–82, 1997.

[27] Yuan Yao, Lorenzo Rosasco, and Andrea Caponnetto. On early stopping in gradient descent learning. *Constructive Approximation*, 26(2):289–315, 2007.

B Biography

In his free time, Elliot enjoys thinking about how to improve the scientific method, be it through methodological, heuristic, or technological improvements to the tools and practices used by researchers. Academically and professionally, he is interested in machine learning, optimization, and data science. In his personal life, Elliot enjoys hiking, cooking, and writing clean code.