TROPHY: Trust Region Optimization Using a Precision Hierarchy

Richard J. Clancy[1,2], Matt Menickelly[1], Jan Hückelheim[1], Paul Hovland[1], Prani Nalluri[1,3], and Rebecca Gjini[1,4]

[1] Argonne National Laboratory, Lemont, IL 60439, USA
[2] University of Colorado, Boulder, CO 80309, USA
[3] Rice University, Houston TX 77005, USA
[4] University of California, San Diego, CA 92093, USA

Abstract. We present an algorithm to perform trust-region-based optimization for nonlinear unconstrained problems. The method selectively uses function and gradient evaluations at different floating-point precisions to reduce the overall energy consumption, storage, and communication costs; these capabilities are increasingly important in the era of exascale computing. In particular, we are motivated by a desire to improve computational efficiency for massive climate models. We employ our method on two examples: the CUTEst test set and a large-scale data assimilation problem to recover wind fields from radar returns. Although this paper is primarily a proof of concept, we show that if implemented on appropriate hardware, the use of mixed-precision can significantly reduce the computational load compared with fixed-precision solvers.

1 Introduction

Optimization methods are used in many applications, including engineering, science, and machine learning. The memory requirements and run time for different methods have been studied extensively and determine the problem sizes that can be run on existing hardware. Similarly, the energy consumption of each method determines its cost and carbon footprint, which is a growing concern [1].

With the desire to incorporate more data into models and ever-increasing computational power, problem scales have grown as well. To improve efficiency, modern computers tightly integrate graphical processing units (GPUs) and other accelerators. Many of these units natively support data types of differing precision to lessen the storage and computational load. Previous work has found significant differences in the overall energy consumption for double- and single-precision computations [2,3]. Server-level products such as NVIDIA Tensor cores in V100 GPUs show 16$\times$ improvement over traditional double precision [4].

Such gains come at a cost, however. Classical algorithms such as the Gram–Schmidt process are well known to suffer from loss of orthogonality and numerical instability due to limited precision [5]. In an effort to ameliorate algorithmic issues with accuracy and stability, there has been a flurry of activity using mixed precision. These methods utilize multiple data types in a principled fashion to reduce the computational burden without sacrificing accuracy. A few of the many applications are tomographic reconstruction [6], seismic modeling [7], and neural network training [8–11]. Mixed-precision methods have been used generically to minimize the cost of linear algebra methods [4], iterative schemes [12], and improved finite element solvers [13].

There are a variety of auto-tuning algorithms that attempt to identify variables within a program that can safely be cast in a lower precision, while satisfying some accuracy constraint [14–18]. Our method proposed does not attempt to identify low precision candidates nor do we try satisfying accuracy constraints. All computation within the objective/gradients are performed in the lowest precision possible and only increase after it is deemed necessary for the solver to proceed.

A recent paper by Gratton and Toint [19] illustrates potential savings in an optimization setting via variable-precision trust region (TR) methods. We investigate the ideas proposed in their work but with an important difference. In particular, their algorithm (TR1DA) requires access to an approximate objective, $\bar{f}(\mathbf{x}_k, \omega_{f,k})$, and gradient, $\bar{\mathbf{g}}(\mathbf{x}_k, \omega_{g,k})$, where $\omega_{f,k}$ and $\omega_{g,k}$ are uncertainty parameters (for the kth iterate $\mathbf{x}_k$) that satisfy

$$|\bar{f}(\mathbf{x}_k, \omega_{f,k}) - f(\mathbf{x}_k)| \leq \omega_{f,k} \qquad \text{and} \qquad \frac{\|\bar{\mathbf{g}}(\mathbf{x}_k, \omega_{g,k}) - \mathbf{g}(\mathbf{x}_k)\|}{\|\bar{\mathbf{g}}(\mathbf{x}_k, \omega_{g,k})\|} \leq \omega_{g,k}.$$

Their error model requires user specified absolute error bounds on function and gradient values; such bounds are difficult to realize in practice as computational complexity grows for reasons such as catastrophic cancellation and accumulated round-off error. Our focus here is on designing an algorithm that performs well without assumptions on the output error when using lower precision.

In this paper, we introduce TROPHY (**T**rust **R**egion **O**ptimization using a **P**recision **H**ierarch**Y**), a mixed-precision TR method for unconstrained optimization. We provide practically verifiable conditions intended to determine whether the error related to a current precision level may be interfering with the dynamics of the TR algorithm. If the conditions are not satisfied, we increase the precision level until they are. Our goal is to lighten the computational load without sacrificing accuracy of the final solution. By using a limited-memory, symmetric rank-1 update (L-SR1) to the approximate Hessian, the method is suitable for large-scale, high-dimensional problems. We compare the method with a standard TR method—supplied with access to either a single- or double-precision evaluation of the function and gradient—on the Constrained and Unconstrained Testing Environment with safe threads (CUTEst) test problem collection [20] and on a large-scale weather model based on the PyDDA software package [21].

Since computational, storage, and communication savings are based on hardware implementations of different precision types rather than assumed theoretical values, our primary metric for comparison will be adjusted function evaluations rather than time. Simply put, adjusted function evaluations discount computations performed in lower-precision levels. The goal here is to provide a proof of concept for computational gains attainable by exploiting variable precision in TR methods. In practice, improvements in energy consumption, time, communication, and memory must be realized through optimized hardware which is beyond the scope of this paper.

2 Background

Consider the unconstrained minimization of a differentiable function $f : \mathbb{R}^n \to \mathbb{R}$,

$$\min_{\mathbf{x}\in\mathbb{R}^n} f(\mathbf{x}). \tag{1}$$

We are motivated by problems where the objective and its derivatives are expensive to calculate as is typical for large-scale computing. In this paper we focus on the TR framework, but could have studied line-search methods instead such as L-BFGS, which is a popular quasi-Newton method distributed in SciPy [22]. However, it is remarkably simpler to illustrate the effect of error on the quality of models within a TR method; that is likely the reason TR methods were employed in [19]. In the following subsections we give an overview of the general framework for TR methods and describe the model function used in our algorithm.

2.1 Trust Region Methods

Trust region methods are iterative algorithms used for numerical optimization. At each iteration (with the counter denoted by k), a model function $m_k : \mathbb{R}^n \to \mathbb{R}$ is built around the incumbent point or iterate, $\mathbf{x}_k$, such that $m_k(\mathbf{0}) = f(\mathbf{x}_k)$ and $m_k(\mathbf{s}) \approx f(\mathbf{x}_k + \mathbf{s})$. The model, m_k, is intended to be a "good" local model of f on the *trust region*, $\{\mathbf{s} \in \mathbb{R}^n : \|\mathbf{s}\| \leq \delta_k\}$ for $\delta_k > 0$. We refer to δ_k as the *trust region radius.* A *trial step*, $\mathbf{s}_k$, is then computed via a(n approximate) solution to the *trust region subproblem*,

$$\mathbf{s}_k \;=\; \underset{\|\mathbf{s}\|\leq\delta_k}{\operatorname{argmin}} \;\; m_k(\mathbf{s}), \tag{2}$$

for $\mathbf{s} \in \mathbb{R}^n$. By an approximate solution to the TR subproblem (2), we mean that one requires the *Cauchy decrease condition* to be satisfied:

$$f(\mathbf{x}_k) - m_k(\mathbf{s}_k) \geq \frac{\mu}{2} \min\left\{\delta_k, \frac{\|\mathbf{g}_k\|}{C}\right\}, \tag{3}$$

where μ and C are constants and $\mathbf{g}_k = \nabla m(\mathbf{x}_k)$. A common choice for m_k is a quadratic Taylor expansion, namely, $m_k(\mathbf{s}) = f(\mathbf{x}_k) + \mathbf{g}_k^T\mathbf{s} + \frac{1}{2}\mathbf{s}^T\nabla^2 f(\mathbf{x}_k)\mathbf{s}$. In practice, $\nabla^2 f(\mathbf{x}_k)$ is typically replaced with a (quasi-Newton) approximation.

Having computed $\mathbf{s}_k$, the standard TR method then compares the true decrease in the function value, $f(\mathbf{x}_k) - f(\mathbf{x}_k + \mathbf{s}_k)$, with the decrease predicted by the model, $m_k(\mathbf{0}) - m_k(\mathbf{s}_k)$. In particular, one computes the quantity

$$\rho_k = \frac{f(\mathbf{x}_k) - f(\mathbf{x}_k + \mathbf{s}_k)}{m_k(\mathbf{0}) - m_k(\mathbf{s}_k)}. \tag{4}$$

If ρ_k is sufficiently positive ($\rho_k > \eta_{\text{good}}$ for fixed $\eta_{\text{good}} > 0$), then the algorithm accepts $\mathbf{x}_k + \mathbf{s}_k$ as the incumbent point $\mathbf{x}_{k+1}$ and may possibly increase the TR radius $\delta_k < \delta_{k+1}$ (if $\rho_k > \eta_{\text{great}}$ for fixed $\eta_{\text{great}} \geq \eta_{\text{good}}$). This scenario is called a *successful iteration.* On the other hand, if ρ_k is not sufficiently positive or is negative ($\rho_k < \eta_{\text{good}}$), then the incumbent point stays the same, $\mathbf{x}_{k+1} = \mathbf{x}_k$, and we set $\delta_{k+1} < \delta_k$. For the experiments below, we chose $\eta_{\text{good}} = 10^{-5}$ and $\eta_{\text{great}} = 0.10$. This process is iterated until a stopping criterion is met, e.g., when the gradient norm $\|\nabla f(\mathbf{x}_k)\|$ is below a given tolerance. Under mild assumptions, TR methods asymptotically converge to stationary points of $f(\mathbf{x})$ [23].

2.2 Model Function

The model function, m_k, must be specified for a TR algorithm. Popular choices include linear or quadratic approximations of the objective using Taylor series or interpolation methods; the latter are often employed in derivative-free optimization [24]. Since many applications of interest are high-dimensional or have costly objective and derivative functions, it is difficult if not impossible to compute and/or store the Hessian matrix for use in quadratic TR models with memory requirement scaling as $\mathcal{O}(n^2)$. A common technique that exploits derivative information while keeping the cost low is to use *curvature pairs* given by $\mathbf{s}_k$ and $\mathbf{y}_k = \nabla f(\mathbf{x}_k + \mathbf{s}_k) - \nabla f(\mathbf{x}_k)$. After each successful iteration, the curvature pairs are used to update the current approximate Hessian denoted by $\mathbf{H}_k$. These updates employ secant approximations of second derivatives. Common update rules include BFGS, DFP, and SR1 [25].

In this work we use a limited-memory symmetric rank-1 update (L-SR1) to the approximate Hessian. This update rule requires the user to set a memory parameter that specifies a number of secant pairs to use in the approximate Hessian. Since we require only a matrix-vector product and not the explicit Hessian, we can implement a matrix-free version reducing the storage cost to $\mathcal{O}(n)$. Thus, our TR subproblem is

$$\mathbf{s}_k = \underset{\|\mathbf{s}\| \leq \delta_k}{\operatorname{argmin}} \ \mathbf{s}^T \nabla f(\mathbf{x}_k) + \frac{1}{2}\mathbf{s}^T \mathbf{H}_k \mathbf{s}, \tag{5}$$

which we recast and approximately solve using the Steihaug conjugate gradient method implemented in [26][Appendix B.4].

In the next section we describe the dynamic precision framework and present criteria for when precision should switch. We then are prepared to give a formal statement of TROPHY. In Section 4 we describe the problems on which we have tested TROPHY, and in Section 5 we discuss the results of our experiments.

3 Method

We assume access to a hierarchy of arithmetic precisions for the evaluation of both $f(\mathbf{x})$ and $\nabla f(\mathbf{x})$, but the direct (infinite-precision) evaluation of $f(\mathbf{x}), \nabla f(\mathbf{x})$ is unavailable. We formalize this slightly by supposing we are given oracles that compute $f^p(\mathbf{x}), \nabla f^p(\mathbf{x})$ for $p \in \{0, \ldots, P\}$. With very high probability, given a uniform distribution on all possible inputs $\mathbf{x}$, the oracles satisfy the inequalities

$$|f^p(\mathbf{x}) - f(\mathbf{x})| > |f^{p+1}(\mathbf{x}) - f(\mathbf{x})|, \quad \|\nabla f^p(\mathbf{x}) - \nabla f(\mathbf{x})\| > \|\nabla f^{p+1}(\mathbf{x}) - \nabla f(\mathbf{x})\|.$$

For a tangible example, if intermediate calculations involved in the computation of $f(\mathbf{x})$ can be done in half, single, or double precision, then we can denote $f^0(\mathbf{x})$, $f^1(\mathbf{x})$, and $f^2(\mathbf{x})$ as the oracles using only half, single, or double, respectively.

To build on the generic TR method described in Section 2, we must specify when and how to switch precision. We can identify two additional difficulties presented in the multiple-precision setting. First, it is currently unclear how to compute ρ_k in (4) since our error model assumes we have no access to an oracle that directly computes $f(\cdot)$. Second, because models m_k typically use function and gradient information provided by $f(\cdot)$ and $\nabla f(\cdot)$, we must specify how to construct models using lower precision oracles.

For the first of these two issues, we make a practical assumption that **the highest level of precision available to us should be treated as if it were infinite precision.** Although this is a theoretically poor assumption, virtually all computational optimization makes it implicitly; algorithms are analyzed over the real numbers but are typically implemented using floating point arithmetic (often double). Thus, in the notation we have developed, the optimization problem we actually aim to solve is not (1) but

$$\min_{\mathbf{x}\in\mathbb{R}^n} f^P(\mathbf{x}), \tag{6}$$

so that the ρ-test in (4) is replaced with

$$\rho_k = \frac{f^P(\mathbf{x}_k) - f^P(\mathbf{x}_k + \mathbf{s}_k)}{m_k(\mathbf{0}) - m_k(\mathbf{s}_k)} = \frac{\text{ared}_k}{\text{pred}_k}. \tag{7}$$

The values ared and pred were introduced to denote "actual reduction" and "predicted reduction", respectively. We note that computing (7) still entails two evaluations of the highest-precision oracle, $f^P(\cdot)$, which is exactly what we hoped to avoid by using mixed-precision. Our algorithm avoids the cost of full-precision evaluations by dynamically adjusting the precision level $p_k \in \{0, \ldots, P\}$ between iterations so that in the kth iteration, ρ_k is approximated by

$$\tilde{\rho}_k = \frac{f^{p_k}(\mathbf{x}_k) - f^{p_k}(\mathbf{x}_k + \mathbf{s}_k)}{m_k(\mathbf{0}) - m_k(\mathbf{s}_k)} = \frac{\text{ered}_k}{\text{pred}_k}, \tag{8}$$

introducing ered to denote "estimated reduction". To update p_k, we are motivated by a strategy similar to one employed in [27] and [28]. We introduce a variable θ_k that is not initialized until the end of the first unsuccessful iteration and set $p_0 = 0$. When the first unsuccessful iteration is encountered, we set

$$\theta_k \leftarrow |\text{ared}_k - \text{ered}_k|\,. \tag{9}$$

Notice that we must incur the cost of two evaluations of $f^P(\cdot)$ following the first unsuccessful iteration in order to compute ared_k. From that point on, θ_k is involved in a test triggered on every unsuccessful iteration (in which the TR radius is sufficiently small) to determine whether the precision level, p_k, should be increased. We compute θ_k and test for precision when $\delta_k < \varDelta_{\text{prec}}$. The value $\varDelta_{\text{prec}}$ is set to be a length scale where numerical imprecision is a concern.

Introducing a predetermined *forcing sequence* $\{r_k\}$ satisfying $r_k \in [0, \infty)$ for all k and $\lim\limits_{k\to\infty} r_k = 0$, and fixing a parameter $\omega \in (0,1)$, we check on any unsuccessful iteration whether

$$\theta_k^\omega \leq \eta \min\{\text{pred}_k, r_k\}\,, \tag{10}$$

where $\eta = \min\{\eta_{\text{good}}, 1 - \eta_{\text{great}}\}$. If (10) does not hold, then we increase $p_{k+1} = p_k + 1$ and again update the value of θ_k according to (9) (thus incurring two more evaluations of $f^P(\cdot)$). The reasoning behind the test in (10) is that if (the unknown) ρ_k in (7) satisfies $\rho_k \geq \eta$, then

$$\eta \leq \rho_k = \frac{\text{ared}_k}{\text{pred}_k} \leq \frac{|\text{ared}_k - \text{ered}_k| + \text{ered}_k}{\text{pred}_k} \approx \frac{\theta_k + \text{ered}_k}{\text{pred}_k} = \frac{\theta_k}{\text{pred}_k} + \tilde{\rho}_k. \quad (11)$$

Thus, for the practical test (8) to be meaningful, we need to ensure that $\theta^k/\text{pred}_k < \eta$, which is what (10) attempts to enforce. The use of ω and the forcing sequence in (10) is designed to ensure that we eventually do not tolerate error, since (11) involves an approximation due to the estimate θ_k. The forcing sequence would likely be necessary to guarantee convergence for theoretical analysis, but is not critical to the performance of a practical algorithm, and was not employed in our implementation. For concreteness, if a forcing sequence were employed, one might consider a slowly decaying sequence such as $r_k = 1/\sqrt{k}$.

It remains to describe how we deal with our second identified difficulty, the construction of m_k in the absence of evaluations of $f(\cdot)$ and $\nabla f(\cdot)$. As is frequently done in trust region methods, we will employ quadratic models of the form

$$m_k(\mathbf{s}) = f_k + \mathbf{g}_k^\top \mathbf{s} + \frac{1}{2}\mathbf{s}^\top \mathbf{H}_k \mathbf{s}. \quad (12)$$

Having already defined rules for the update of p_k through the test (10), we take in the kth iteration $f_k = f^{p_k}(\mathbf{x})$ and $\mathbf{g}_k = \nabla f^{p_k}(\mathbf{x})$. In theory, we require $\mathbf{H}_k$ to be any Hessian approximation with a spectrum bounded above and below uniformly for all k. In practice, we update $\mathbf{H}_k$ via L-SR1 updates [29]. By implementing a reduced-memory version, we need not store an explicit approximate Hessian, thus greatly reducing the memory cost and significantly accelerating the matrix-vector products in our model. Pseudocode for TROPHY is provided in Algorithm 1.

4 Test Problems and Implementations

Our initial implementation of TROPHY is written in Python. To validate the algorithm, we focus on a well-known optimization test suite and a problem relating to climate modeling. In all cases, the algorithms terminate when one of the following conditions are met: (1) the first-order condition is satisfied, namely, $\|\nabla f^P(\mathbf{x}_k)\| < \epsilon_{\text{tol}}$; (2) the TR radius is smaller than machine precision, namely, $\delta_k < \epsilon_{\text{machine}}$; or (3) the first two conditions have not been met after some maximum number of iterations. Condition 1 is a success whereas conditions 2 and 3 are failed attempts. We describe the problem setup and implementation considerations in the current section and then discuss results in Section 5.

4.1 CUTEst

Our first example used the CUTEst set [20], which is well known within the optimization community and offers a variety of problems that are challenging to solve. Each problem is given in a Standard Input Format [30] file that is passed to a decoder from which Fortran subroutines are generated. The problems can be built directly by using single or double precision, making the set useful for mixed-precision comparison.

Algorithm 1: TROPHY

```
Initialize $0 < \eta_{\text{good}} \leq \eta_{\text{great}} < 1$, $\omega \in (0,1)$, $\gamma_{\text{inc}} > 1$, $\gamma_{\text{dec}} \in (0,1)$,
  $\Delta_{\text{prec}} \in (0,1)$, forcing seq. $\{r_k\}$.
Choose initial $\delta_0 > 0$, $\mathbf{x}_0 \in \mathbb{R}^n$.
$\theta_0 \leftarrow 0, p_k \leftarrow 0, k \leftarrow 0, \text{failed} \leftarrow \text{FALSE}$
while some stopping criterion not satisfied do
    Construct model $m_k$.
    (Approximately solve) (2) to obtain $\mathbf{s}_k$.
    Compute $\tilde{\rho}_k$ as in (8).
    if $\tilde{\rho}_k > \eta_{good}$ (successful iteration) then
        $\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k + \mathbf{s}_k$.
        if $\tilde{\rho}_k > \eta_{great}$ (very successful iteration) then
            $\delta_{k+1} \leftarrow \gamma_{\text{inc}}\delta_k$.
        end
    else
        if not failed then
            Compute $\theta_k$ as in (9).
            failed $\leftarrow$ TRUE.
        end
        if (10) holds or $\delta_k \geq \Delta_{prec}$ then
            $\delta_{k+1} \leftarrow \gamma_{\text{dec}}\delta_k$.
        else
            $p_{k+1} \leftarrow p_k + 1$.
            Compute $\theta_k$ as in (9).
            $\delta_{k+1} \leftarrow \delta_k$.
        end
        $\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k$.
    end
    $k \leftarrow k + 1$.
end
```

Python Implementation of CUTEst: The PyCUTEst package [31] serves as an interface between Python and CUTEst's Fortran source code. The problems are compiled via the interface; then Python scripts are generated and cached for subsequent function calls. Although CUTEst natively supports both single- and double-precision evaluations, at the time of writing, the single-precision implementations are concealed from the PyCUTEst API.

To access single-precision evaluations, we used PREDUCER [32], a Python script written to compare the effect of round-off errors in scientific computing. PREDUCER parses Fortran source code and downcasts double data types to single. To allow for its use in existing code, all single data types are recast to double after function/gradient evaluation but before returning to the calling program. Because of the overhead associated with casting operations, we do not expect improvements in computational time. However, performance gains in terms of both accuracy and a reduction in the number of adjusted function calls for an iterative algorithm should be realized. We built the double-precision functions, too, and wrapped both functions to pass as a unified handle to TROPHY.

For the subset of unconstrained problems with dimension less than or equal to 100, we ran TROPHY with single/double switching along with the same TR method using only single or double precision. Our first-order stopping criterion was $\|\nabla f^P(\mathbf{x}_k)\| < 10^{-5}$, and the maximum number of allowable iterations was 5,000. We show results for problems solved by at least one TR in Section 5.

Julia Implementation of CUTEst: The Julia programming language supports variable-precision floating-point data types. More precisely, it allows users to specify the number of bits used in the mantissa and expands memory for the exponent as necessary to avoid overflow. This is in contrast to the IEEE 754 standard that uses 11 (5), 24 (8), and 53 (11) bits for the mantissa (exponent) of half-, single-, and double-precision floats, respectively. In practice, one can assign enough bits for the exponent to avoid dynamic reallocation.

To exploit variable precision, we hand-coded several of the unconstrained CUTEst objectives in Julia and then computed gradients with forward-mode automatic differentiation (AD) using the ForwardDiff.jl package [33]. We wrote a Julia port that allows us to call this code from Python for use in TROPHY. We opted to use forward-mode AD for its ease of implementation. In all cases used, the hand-coded Julia objective and AD gradient were compared against the Fortran implementation and found to be accurate.

We compared TROPHY with TR methods using a fixed precision of half, single, and double precision (11, 24, and 53 bits, respectively). We used the same first-order condition of $\|\nabla f(\mathbf{x}_k)\| < 10^{-5}$ but allowed this implementation to run only for 1,000 iterations. For TROPHY, we used several precision-switching sets: $\{24, 53\}$, $\{11, 24, 53\}$, $\{8, 11, 17, 24, 53\}$, and $\{8, 13, 18, 23, 28, 33, 38, 43, 48, 53\}$. The third set of precisions was motivated by the number of mantissa bits in bfloat16, fp16, fp24, fp32, and fp64, respectively. The last set increased the number of bits in increments of 5 up to double-precision.

4.2 Multiple Doppler Radar Wind Retrieval:

We also looked at a data assimilation problem for retrieving wind fields for convective storms from Doppler radar returns. Shapiro and Potvin [34, 35] proposed a method for doing so that optimizes a cost functional based on vertical vorticity, mass continuity, field smoothness, and data fidelity, among others. Although the function calls are fairly simple, the wind field must be reconciled on a 3-D grid over space, each with an x, y, and z component. For a $39 \times 121 \times 121$ grid, there are $1,712,997$ variables. Therefore, reducing computational, storage, and communication costs where possible is paramount.

Our work centered on the PyDDA package [21], which was written to solve the aforementioned problem. We amended the code in two significant ways. First, to improve efficiency, we rewrote portions of the code to use JAX, an automatic differentiation package using XLA that exploits efficient computation on GPUs [36]. Since JAX natively supports single precision on CPUs and can be recast to half and double as desired, it nicely serves as a proof of concept on a real application. Second, we modified the solver to use TROPHY rather than the SciPy implementation of L-BFGS.

Once again, we compared TROPHY against single- and double-precision TR methods. TROPHY switched among half, single, and double precision. To avoid overflow initially for half-precision, we warm started the algorithm by providing it with the tenth iterate from the double-precision TR method, i.e., $\mathbf{x}_{10}$. We perturbed this initial iterate 10 times and used the perturbed vectors as the initial guess for each algorithm (including double TR). We measured the average performance when solving each problem to different first-order conditions: $\|\nabla f(\mathbf{x}_k)\| < 10^{-3}$ and $\|\nabla f(\mathbf{x}_k)\| < 10^{-6}$. The maximum number of allowable iterations was 10,000.

5 Experimental Results

We display results across the CUTEst set using data and performance profiles [37, 38]. For a given metric, performance profiles help determine how a set of solvers, $\mathcal{S}$, performs over a set of problems, $\mathcal{P}$. The value $v_{ij} > 0$ denotes a particular metric (say, the final gradient norm) of the jth solver on problem i. We can then consider the performance of each solver in relation to the solver that performed best, that is, the one that achieved the smallest gradient norm. The *performance ratio* is defined as

$$r_{ij} = \frac{v_{ij}}{\min_j \{v_{ij}\}}. \tag{13}$$

Smaller values of r_{ij} are better since they are closer to optimal. The performance ratio was set to ∞ if the solver failed to solve the problem. We can evaluate the performance of a solver by asking what percentage of the problems are solved within a fraction of the best. This is given by the *performance profile*,

$$h_j(\tau) = \frac{\sum_{i=1}^{N} \mathcal{I}_{\{r_{ij} \le \tau\}}}{N},$$

where $N = |\mathcal{P}|$ (the cardinality of $\mathcal{P}$) and $\mathcal{I}_{\{A\}}$ is the indicator function such that $\mathcal{I}_{\{A\}} = 1$ if A is true and 0 otherwise. Hence, better solvers have profiles that are above and to the left of the others.

Motivated by the computational models in [2] and [3], we assume that the energy efficiency of single precision is between 2 and 3.6 times higher than double precision [39, 40]. The storage and communication have less optimistic savings since we expect the cost of both to scale linearly with the number of bits used in the mantissa. Accordingly, we focus primarily on the model where half- and single-precision evaluations cost 1/4 and 1/2 that of a double evaluation, respectively. This gives a conservative estimate for energy cost and a favorable one for execution time. For a given problem and solver, we define *adjusted calls*:

$$\text{Adj. calls} \;=\; \sum_{p \,\in\, \{0,1,\ldots,P\}} \frac{(\#\text{ bits for prec. } p) \times \;(\#\text{ func. calls at prec. } p)}{\#\text{ bits in prec. } P}. \tag{14}$$

Figures 1 and 2 show performance profiles for the Python and Julia implementations of CUTEst, respectively. All CUTEst problems had their first-order tolerance set to 10^{-5}. Working from right to left in both images, we can see that

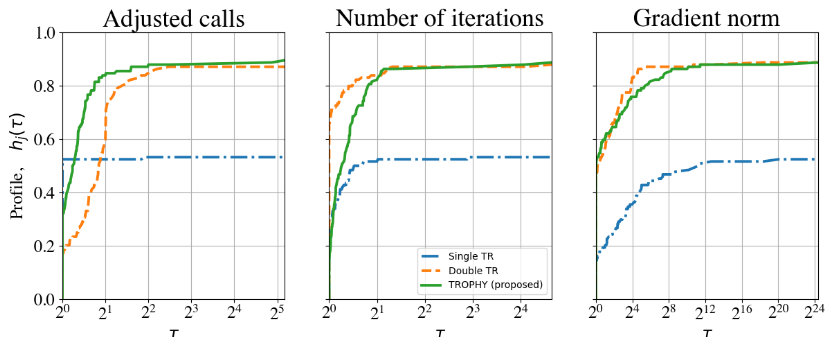

Fig. 1: Performance profiles for Python implementation of unconstrained CUTEst problems of dimension < 100 solved to first-order tolerance of 10^{-5}. Standard single and double TR methods compared against TROPHY using single/double switching.

the first-order condition is steady across methods provided that double-precision evaluations are ultimately available to the solver. When limited to half (11 bits) or single (24 bits), the performance suffers, and a number of problems cannot be solved. For the number of iterations in Python, we see that TROPHY and the double TR method perform comparably. The Julia implementation shows that the iterations count suffers when using low precision or TROPHY with many precision levels available for switching. This behavior is expected for low precision since the solver may never achieve the desired accuracy and hence runs longer, and for TROPHY since each precision switch requires a full iteration. For example, if 10 precision levels are available, TROPHY will take at least 10 iterations to complete. This limits the usefulness of the method on small to medium problems and problems where the initial iterate is close to the final solution. As anticipated, TROPHY shows a distinct advantage for adjusted calls. The one exception is when there are many precision levels to cycle through, for the same reason as above. Although the initial iterate might be close to optimal, the algorithm must still visit all precision levels before breaking. The fact is made worse since each time the precision switches, two evaluations at the highest precision are required. Using two or three widely spaces precision levels yields strong results for the CUTEst set.

Table 1: Average performance over ten initializations for single-precision TR, double-precision TR, and TROPHY on PyDDA wind retrieval example. Adjusted calls indicate improved computational efficiency. Half, single, and double costs are 1/4, 1/2, and 1 for linear and 1/16, 1/4, and 1 for quadratic adjustments, respectively. Problem solved to $\|\nabla f(x_{\text{final}})\|_2 < 10^{-3}$ above.

Tolerance $\|\nabla f\| < 10^{-3}$	Half calls	Single calls	Double calls	**Adj. calls (linear)**	**Adj. calls (quad.)**	f_{final}	$\|\nabla f_{\text{final}}\|$
Single	-	3411	-	**1706**	**853**	5.3×10^{-3}	9.5×10^{-4}
Double	-	-	1877	**1877**	**1877**	4.5×10^{-3}	9.1×10^{-4}
TROPHY	465	1898	6	**<u>1071</u>**	**<u>510</u>**	4.7×10^{-3}	9.4×10^{-4}

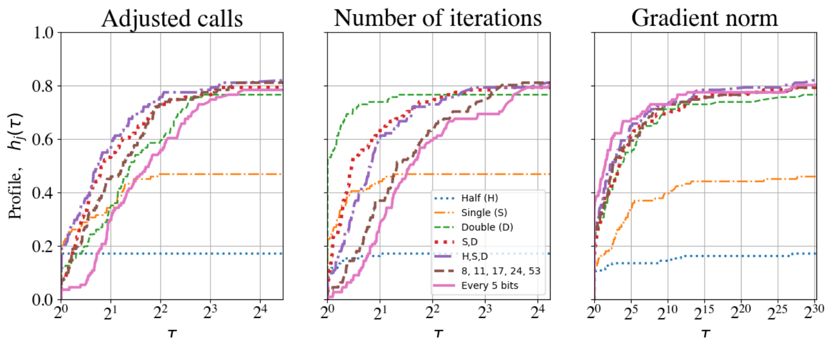

Fig. 2: Performance profiles for Julia implementation of unconstrained CUTEst problems of dimension < 100 solved to first-order tolerance of 10^{-5}. Half, single, and double are standard TR methods using corresponding precision. "S,D", "H,S,D", "8,11,17,24,53", and "Every 5 bits" are TROPHY implementations using different precision regimes. "Every 5 bits" starts at 8 bits and increases to 53 bits in increments of 5 bits. A finer precision hierarchy does not imply better performance.

Table 2: Problem solved to $\|\nabla f(x_{\text{final}})\|_2 < 10^{-6}$ accuracy. The single-precision TR method failed to converge with the TR radius falling below machine precision.

Tolerance $\|\nabla f\| < 10^{-6}$	Half calls	Single calls	Double calls	**Adj. calls (linear)**	**Adj. calls (quad.)**	f_{final}	$\|\nabla f_{\text{final}}\|$
Single	-	∞	-	**FAIL**	**FAIL**	9.9×10^{-7}	3.9×10^{-6}
Double	-	-	5283	**5283**	**5283**	1.8×10^{-7}	9.6×10^{-7}
TROPHY	465	7334	601	**<u>4384</u>**	**<u>2464</u>**	1.9×10^{-7}	9.7×10^{-8}

The wind retrieval example shows similar results. The switching criteria used here differs slightly from the one presented in (10). Specifically, a baseline θ_{p_k} is set after the first failed iteration at the current precision level. The model predicted reduction (pred_k) is compared to the baseline θ_{p_k} for successive failures. If pred_k is small compared to the baseline θ_{p_k}, then the precision is increased. We do not expect this to significantly change the qualitative results or behavior of the method for this test case. We included two "adjusted call" columns: one for a linear decay adjustment (memory and communication as above) and the other for quadratic decay (reduction in energy consumption). We originally iterated until $\|\nabla f(\mathbf{x}_k)\| < 10^{-6}$ but observed that the single TR method failed to converge. Consequently, we loosened the stopping criterion as far as possible while maintaining the correct qualitative behavior of the solution. TROPHY outperformed the standard (double-precision TR) method in all cases, reducing the number of adjusted calls by 17% to 73%.

Our results show a promising reduction in the relative cost over naive single or double TR solvers. We expect that for many problems where function evaluations dominate linear algebra costs for the TR subproblem, our time to solve will greatly benefit from the method.

6 Conclusion and Future Work

In this paper we introduced TROPHY, a TR method that exploits variable-precision data types to lighten the computational burden of expensive function/gradient evaluations. We illustrated proof of concept for the algorithm by implementing it on the CUTEst set and PyDDA. The full benefit of our work has not yet been realized. We look forward to implementing similar tests on hardware that can realize the full benefit of lower energy consumption and reduced memory/communication costs and ultimately shorten the time to solution. This will be especially beneficial for large scale climate models.

We would also like to incorporate mixed precision into line-search methods given their popularity in quasi-Newton solvers. By incorporating the same ideas into highly optimized algorithms such as the SciPy implementation of L-BFGS, we could easily deploy mixed precision to a wide population, dramatically reducing computational loads. Although TR methods are, computationally speaking, more appropriate for expensive-to-evaluate objectives, there is no reason the same ideas cannot be extended if practitioners prefer them.

Acknowledgements: We gratefully acknowledge the support by the Applied Mathematics activity within the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research Program, under contract number DE-AC02-06CH11357, and the computing resources provided on Swing, a high-performance computing cluster operated by the Laboratory Computing Resource Center at Argonne National Laboratory.

References

1. K. Hao, "Training a single AI model can emit as much carbon as five cars in their lifetimes," *MIT Technology Review*, 2019.
2. D. Molka, D. Hackenberg, R. Schöne, and M. S. Müller, "Characterizing the energy consumption of data transfers and arithmetic operations on x86- 64 processors," in *International conference on green computing*, pp. 123–133, IEEE, 2010.
3. G. Kestor, R. Gioiosa, D. J. Kerbyson, and A. Hoisie, "Quantifying the energy cost of data movement in scientific applications," in *2013 IEEE international symposium on workload characterization*, pp. 56–65, IEEE, 2013.
4. A. Abdelfattah, H. Anzt, E. G. Boman, E. Carson, T. Cojean, J. Dongarra, A. Fox, M. Gates, N. J. Higham, X. S. Li, *et al.*, "A survey of numerical linear algebra methods utilizing mixed-precision arithmetic," *The International Journal of High Performance Computing Applications*, p. 10943420211003313, 2021.
5. G. H. Golub and C. F. Van Loan, *Matrix computations.* Johns Hopkins University Press, Baltimore, MD, 1996.
6. N. Doucet, H. Ltaief, D. Gratadour, and D. Keyes, "Mixed-precision tomographic reconstructor computations on hardware accelerators," in *2019 IEEE/ACM 9th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*, pp. 31–38, IEEE, 2019.
7. T. Ichimura, K. Fujita, T. Yamaguchi, A. Naruse, J. C. Wells, T. C. Schulthess, T. P. Straatsma, C. J. Zimmer, M. Martinasso, K. Nakajima, *et al.*, "A fast scalable implicit solver for nonlinear time-evolution earthquake city problem on low-ordered unstructured finite elements with artificial intelligence and transprecision

computing," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 627–637, IEEE, 2018.
8. X. Jia, S. Song, W. He, Y. Wang, H. Rong, F. Zhou, L. Xie, Z. Guo, Y. Yang, L. Yu, *et al.*, "Highly scalable deep learning training system with mixed-precision: Training imagenet in four minutes," *preprint arXiv:1807.11205*, 2018.
9. P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, *et al.*, "Mixed precision training," in *International Conference on Learning Representations*, 2018.
10. N. Wang, J. Choi, D. Brand, C.-Y. Chen, and K. Gopalakrishnan, "Training deep neural networks with 8-bit floating point numbers," in *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, pp. 7686–7695, 2018.
11. Z. Carmichael, H. F. Langroudi, C. Khazanov, J. Lillie, J. L. Gustafson, and D. Kudithipudi, "Performance-efficiency trade-off of low-precision numerical formats in deep neural networks," in *Proceedings of the Conference for Next Generation Arithmetic 2019*, pp. 1–9, 2019.
12. R. Strzodka and D. Göddeke, "Mixed precision methods for convergent iterative schemes," *EDGE*, vol. 6, pp. 23–24, 2006.
13. D. Göddeke, R. Strzodka, and S. Turek, "Performance and accuracy of hardware-oriented native-, emulated-and mixed-precision solvers in FEM simulations," *International Journal of Parallel, Emergent and Distributed Systems*, vol. 22, no. 4, pp. 221–256, 2007.
14. W.-F. Chiang, M. Baranowski, I. Briggs, A. Solovyev, G. Gopalakrishnan, and Z. Rakamarić, "Rigorous floating-point mixed-precision tuning," in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, (New York, NY, USA), p. 300–315, Association for Computing Machinery, 2017.
15. S. Graillat, F. Jézéquel, R. Picot, F. Févotte, and B. Lathuilière, "Auto-tuning for floating-point precision with discrete stochastic arithmetic," *Journal of computational science*, vol. 36, p. 101017, 2019.
16. H. Guo and C. Rubio-González, "Exploiting community structure for floating-point precision tuning," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 333–343, 2018.
17. H. Menon, M. O. Lam, D. Osei-Kuffuor, M. Schordan, S. Lloyd, K. Mohror, and J. Hittinger, "Adapt: Algorithmic differentiation applied to floating-point precision tuning," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 614–626, IEEE, 2018.
18. C. Rubio-González, C. Nguyen, H. D. Nguyen, J. Demmel, W. Kahan, K. Sen, D. H. Bailey, C. Iancu, and D. Hough, "Precimonious: Tuning assistant for floating-point precision," in *SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pp. 1–12, 2013.
19. S. Gratton and P. L. Toint, "A note on solving nonlinear optimization problems in variable precision," *Computational Optimization and Applications*, vol. 76, no. 3, pp. 917–933, 2020.
20. N. I. Gould, D. Orban, and P. L. Toint, "CUTEst: a constrained and unconstrained testing environment with safe threads for mathematical optimization," *Computational Optimization and Applications*, vol. 60, no. 3, pp. 545–557, 2015.
21. R. Jackson, S. Collis, C. Potvin, and T. Munson, "PyDDA: A Pythonic direct data assimilation framework for wind retrievals," *Journal of Open Research Software*, vol. 8(1), 2020.

22. P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, *et al.*, "SciPy 1.0: fundamental algorithms for scientific computing in Python," *Nature Methods*, vol. 17, no. 3, pp. 261–272, 2020.
23. A. R. Conn, N. I. Gould, and P. L. Toint, *Trust region methods.* SIAM, 2000.
24. J. Larson, M. Menickelly, and S. M. Wild, "Derivative-free optimization methods," *Acta Numerica*, vol. 28, pp. 287–404, 2019.
25. J. Nocedal and S. Wright, *Numerical optimization.* Springer Science & Business Media, 2006.
26. A. S. Berahas, M. Jahani, and M. Takác, "Quasi-Newton methods for deep learning: Forget the past, just sample," *preprint arXiv:1901.09997*, vol. 16, 2019.
27. M. Heinkenschloss and L. Vicente, "Analysis of inexact trust-region sqp algorithms," *SIAM J. on Optimization*, vol. 12, pp. 283–302, 02 2002.
28. D. P. Kouri, M. Heinkenschloss, D. Ridzal, and B. G. van Bloemen Waanders, "Inexact objective function evaluations in a trust-region algorithm for PDE-constrained optimization under uncertainty," *SIAM J. Scientific Computing*, vol. 36, 2014.
29. R. H. Byrd, J. Nocedal, and R. B. Schnabel, "Representations of quasi-Newton matrices and their use in limited memory methods," *Mathematical Programming*, vol. 63, no. 1, pp. 129–156, 1994.
30. A. R. Conn, G. Gould, and P. L. Toint, *LANCELOT: a Fortran package for large-scale nonlinear optimization (Release A)*, vol. 17. Springer Science & Business Media, 2013.
31. J. Fowkes and L. Roberts, "PyCUTEst, https://jfowkes.github.io/pycutest," 2018.
32. J. Hückelheim, "PREDUCER, https://github.com/jhueckelheim/preducer," 2019.
33. J. Revels, M. Lubin, and T. Papamarkou, "Forward-mode automatic differentiation in Julia," *preprint arXiv:1607.07892*, 2016.
34. A. Shapiro, C. K. Potvin, and J. Gao, "Use of a vertical vorticity equation in variational dual-Doppler wind analysis," *Journal of Atmospheric and Oceanic Technology*, vol. 26, no. 10, pp. 2089–2106, 2009.
35. C. K. Potvin, A. Shapiro, and M. Xue, "Impact of a vertical vorticity constraint in variational dual-doppler wind analysis: Tests with real and simulated supercell data," *Journal of Atmospheric and Oceanic Technology*, vol. 29, no. 1, pp. 32–49, 2012.
36. J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang, "JAX: composable transformations of Python+NumPy programs, http://github.com/google/jax," 2018.
37. E. D. Dolan and J. J. Moré, "Benchmarking optimization software with performance profiles," *Mathematical Programming*, vol. 91, pp. 201–213, 2002.
38. J. J. Moré and S. M. Wild, "Benchmarking derivative-free optimization algorithms," *SIAM J. on Optimization*, vol. 20, no. 1, pp. 172–191, 2009.
39. M. Fagan, J. Schlachter, K. Yoshii, S. Leyffer, K. Palem, M. Snir, S. M. Wild, and C. Enz, "Overcoming the power wall by exploiting inexactness and emerging COTS architectural features: Trading precision for improving application quality," in *2016 29th IEEE International System-on-Chip Conference (SOCC)*, pp. 241–246, 2016.
40. S. Galal and M. Horowitz, "Energy-efficient floating-point unit design," *IEEE Transactions on Computers*, vol. 60, no. 7, pp. 913–922, 2011.