LMBOPT – a limited memory method for bound-constrained optimization

Morteza Kimiaei

Fakultät für Mathematik, Universität Wien
Oskar-Morgenstern-Platz 1, A-1090 Wien, Austria
email: kimiaeim83@univie.ac.at
WWW: http://www.mat.univie.ac.at/~kimiaei

Arnold Neumaier

Fakultät für Mathematik, Universität Wien
Oskar-Morgenstern-Platz 1, A-1090 Wien, Austria
email: Arnold.Neumaier@univie.ac.at
WWW: http://www.mat.univie.ac.at/~neum

Behzad Azmi

Johann Radon Institute for Computational and Applied Mathematics (RICAM)
Austrian Academy of Sciences
Altenbergerstraße 69, A-4040 Linz, Austria
email: behzad.azmi@ricam.oeaw.ac.at

August 30, 2019

Abstract. This paper describes the theory and implementation of LMBOPT, a first order algorithm for bound constrained optimization problems with continuously differentiable objective function. LMBOPT is based on the generic algorithm recently proposed by Neumaier & Azmi, which uses a gradient-free line search along a bent search path. LMBOPT includes many practical enhancements such as a new limited memory quasi-Newton direction and a robust bent line search. The numerical results on unconstrained and bound constrained problems from CUTEst [32] show that LMBOPT is very robust and efficient.

Contents

1 Introduction 3
   1.1 Past work .................................................. 3
   1.2 Mathematically background and notation .................. 5
   1.3 Algorithms and data structures .......................... 7
1 Introduction

In this paper we describe a new active set method for solving the bound constrained optimization problem
\[
\min f(x) \\
\text{s.t. } x \in \mathbb{R}^n, \quad x \leq x \leq x,
\]
where $x = [x, x]$ is a bounded or unbounded box in $\mathbb{R}^n$ describing the bounds on the variables and the objective function $f : x \to \mathbb{R}$ is continuously differentiable with gradient $g(x) := \partial f(x)/\partial x \in \mathbb{R}^n$.

Often variables of an optimization problems can only be considered meaningful within a particular interval [29]. Independent of this, problems with naturally given bounds appear in a wide range of applications including optimal design problem [4], contact and friction in rigid body mechanics [46], the obstacle problem [50], journal bearing lubrication and flow through a porous medium [44]. Some approaches [1] reduce the solution of variational inequalities and complementarity problems to bound constrained problems. The bound constrained optimization problem also arises as an important subproblem in algorithms for solving general constrained optimization problems based on augmented Lagrangians and penalty methods [15, 26, 36, 35, 47]. These facts led to a lot of research dealing with the development of efficient numerical algorithms for solving bound constrained optimization problems, especially when the number of variables is large.

1.1 Past work

A bound constrained optimization problem (BOPT) consists of minimizing a continuously differentiable objective function subject to a feasible region defined by simple bounds on the variables. In the past few decades, many algorithms have been developed for solving such problems. Active set methods are among the most effective methods for solving BOPT problems. They consist of two main stages that alternate until a solution is found. In the first stage one identifies a good approximation for the set of optimal active bound constraints, defining a face likely to contain a stationary point of the problem. A second stage then explores this face of the feasible region by approximately solving an unconstrained subproblem.

A classical reference on active set methods for bound constrained problems with convex quadratic objective function (QBOPT) is the projected conjugate gradient method of POLYAK [52], which dropped and added only one constraint in each iteration. That is, at each step of this active set method, the dimension of the subspace of active variables is changed only by one. This fact implies that if there are $n_1$ constraints active at the starting point $x_0$ and $n_2$ constraints active on the solution of QBOPT, we need at least
$|n_2 - n_1|$ iterations to reach the solution of QBOPT. This may be serious drawback in the case of large scale problems. Dembo & Tulowitzky [22] introduces in 1983 methods for QBOPT that are able to add and drop many constrains at each iteration. Their basic idea was further developed by Yang & Tolle [57] into an algorithm guaranteed to identify in finitely many iterations the face containing a local solution of the QBOPT, even when the solution of the problem is degenerate. For further research on the QBOPT we refer the reader to [24, 25, 49, 50].

For BOPT with a general nonlinear objective function, Bertsekas [3] proposed an active set algorithm that uses a gradient projection method to find optimal active variables. He showed that this method is able to find very quickly the face containing a local solution. Further research on convergence and properties of projected gradient methods can be found in [3, 13, 27]. The idea of using gradient projections for identifying optimal active constraints was followed up by many researchers. Many of them [11, 14, 16] combined Newton type methods with gradient projection method in order to accelerate the convergence. For example, L-BFGS-B, developed by Byrd, Lu, Zhu & Nocedal [11], performs the gradient projection method by computing the Cauchy point to determine the active variables. After the set of active variables is determined, the algorithm performs line searches along the search directions obtained by a limited memory BFGS method [12] to explore the subspace of nonactive variables. In fact, the use of limited memory BFGS matrices and the line search strategy are the main properties that distinguish this method from others, especially from the trust region type method proposed by Conn, Gould and Toint [14, 16].

A non-monotone line search was first introduced for Newton methods by Grippo, Lampariello & Lucidi (GLL) in [33], in order to improve the ability to follow a curved valley with steep walls. Later several papers [18, 21, 28, 34, 55, 58] on non-monotone line search methods pointed out that in many cases these methods are more efficient than monotone line search methods. Other papers [4, 8, 19, 20, 30, 45, 54] indicate that gradient projection approaches based on a Barzilai-Borwein step size [2] have impressive performance in a wide range of applications. Some recent works [5, 6, 7, 8, 9, 53] on Barzilai-Borwein gradient projection methods (BBGP) have modified them by incorporating them with the GLL non-monotone line search: For instance, Raydan [53] developed the BBGP method for solving unconstrained optimization problems, Dai & Fletcher [19, 20] proposed BBGP methods for large-scale bound constrained quadratic programming. Birgin, Martínez & Raydan [8, 9] developed the idea of Raydan [53] to an effective algorithm (SPG) for the minimization of differentiable functions subject to closed convex set. Later they used the SPG algorithm in the active set framework [5, 7] for dealing with bound constrained problems. In both of these methods, the task of SGP is to search through different faces of the feasible region.

To deal with the objective function within faces, [5] used the second-order trust region algorithm of Zhang & Xu [59], and [7] designed a new algorithm whose line search iteration is performed by means of backtracking and extrapolation. More recently, Hager & Zhang [40] developed an active set algorithm called ASA CG for large scale bound constrained problems. ASA CG consists of two main steps within a framework for branching between these two steps: a non-monotone gradient projection step which is based on their research on
cyclic Barzilai-Borwein method [21], and an unconstrained step that utilizes their developed
conjugate gradient algorithms [37, 38, 39, 41]. ASA_CG version 3.0 has been updated by
calling CG_descent version 6.0 which uses the variable HardConstraint to evaluate the
function or gradient at a point that violates the bound constraints, so that it could improve
performance by giving the code additional flexibility in the starting step size routine.

A considerable amount of literature has been published on line search algorithms, most
of which satisfy the Wolfe conditions (WOLFE [56]) or Goldstein conditions (GOLDSTEIN
[31]). A problem of line search algorithms satisfying the Wolfe conditions is the need to
calculate a gradient at every trial point. On the other hand, line search algorithms based on
the Goldstein conditions are gradient-free, but they have very poor behaviour in strongly
nonconvex regions. NEUMAIER & AZMI [51] introduced an efficient gradient free curved
line search CLS (Algorithm 3.3 in [51]) using a new active set method BOPT (Algorithm
9.1 in [51] = Algorithm 1.1).

1.2 Mathematically background and notation

We define some notation that will be used frequently throughout the paper.

In the pseudo-code for all algorithms, a Matlab like notation is used.

- ∼ (or not) denotes logical negation.
- ◦ and // denote componentwise multiplication and componentwise division, respectively.
- $A \backslash b$ denotes the solution $x$ of the system of linear equations $Ax = b$.
- The notation == is comparison operator for equality.
- $A_{:, k}$ denotes the $k$th column of a matrix $A$.
- length$(v)$ denotes the length of the vector $v$.
- ones$(n, 1)$ denotes a $n \times 1$ vector whose entries are 1.
- zeros$(n, 1)$ denotes a $n \times 1$ vector whose entries are 0.
- isnan$(A)$ returns an array of the same size as $A$ containing logical 1 (true) where the
elements of $A$ are NaNs and logical 0 (false) where they are not.

The reduced gradient at $x$ is $g_{\text{red}}(x)$ the vector defined with components

$$
(g_{\text{red}}(x))_i := \begin{cases} 
0 & \text{if } x_i = \underline{x}_i = \bar{x}_i, \\
\min(0, g_i) & \text{if } x_i = \underline{x}_i < \bar{x}_i, \\
\max(0, g_i) & \text{if } x_i = \bar{x}_i > \underline{x}_i, \\
g_i & \text{otherwise},
\end{cases}
$$

where $g_i := g_i(x)$ is the $i$th component of gradient vector at $x$. 5
The bound $\overline{x}_i$ or $\underline{x}_i$ (and the index $i$) is called **active** if $x_i = \overline{x}_i$ or $x_i = \underline{x}_i$, respectively. The set of **free indices** of $x$ is defined by

$$I_-(x) := \{i \mid \underline{x}_i < x_i < \overline{x}_i\},$$

and the set of **free or freeable indices** of $x$ is presented by

$$I_+(x) := I_-(x) \cup \{i \mid (g_{\text{red}})_i \neq 0\}$$

$$= I_-(x) \cup \{i \mid \underline{x}_i = x_i < \overline{x}_i, g_i < 0 \text{ or } \underline{x}_i < x_i = \overline{x}_i, g_i > 0\}$$

where

$$g := g(x), \quad g_{\text{red}} := g_{\text{red}}(x).$$

Given a descent direction $p$ with $g_{\text{red}}^T p < 0$, the each line search along a **bent search path**

$$x(\alpha) := \pi[x + \alpha p],$$

is obtained by projecting the ray $x + \alpha p$ ($\alpha \geq 0$) into the feasible set, using the projection $\pi[x]$ with components

$$\pi[x]_i := \sup(\underline{x}_i, \inf(x_i, \overline{x}_i)) = \begin{cases} 
\underline{x}_i & \text{if } x_i \leq \underline{x}_i, \\
\overline{x}_i & \text{if } x_i \geq \overline{x}_i, \\
x_i & \text{otherwise.}
\end{cases}$$

According to Neumaier & Azmi [51], the convergence of **BOPT** is guaranteed when the following conditions hold for some positive constant $\delta > 0$, any index set $I = I_\pm(x)$, the gradient $g = g(x)$, and the search directions $p$

$$p_i = 0 \quad \text{for } i \notin I,$$

$$\frac{g_I^T p_I}{\|g_I\|\|p_I\|} \leq -\delta < 0,$$

$$g_ip_i \leq 0 \quad \text{for all } i \quad \text{if } I = I_+(x) \neq I_-(x),$$

$$\|g_I\|^2 \geq \rho\|g_{\text{red}}\|^2,$$

where $p_I$ stands for the restriction of $p$ to the index set $I$. The examples in [51] described the unfavorable zigzagging behaviour depending on which variables enter into the working set $I$. By definition of the reduced gradient, (10) always holds for the choice of $I = I_+(x)$. But the choice of $I = I_-(x)$ might violate (10); in this case the working set is updated by $I_+(x)$.

**1.1 Algorithm. (BOPT, bound constrained optimization)**

**Purpose**: minimize smooth $f(x)$ subject to $x \in x = [\underline{x}, \overline{x}]$

**Input**: $x^0 \in \mathbb{R}^n$ (starting point)

**Parameters**: $\beta \in ]0, 1[^2$, $q > 1$ (line search parameters)

$0 < \delta < 1$ (reduced angle parameters)

$0 < \rho < 1/n$ (factor safeguarding (10))

and parameters specifying a pair of monotone dual norms
In this paper, our goal is to present and test the limited memory method for bound-constrained optimization (LMBOPT). It uses a gradient-free line search along a bent search path. Since it conforms to the assumptions of [51], it finds all strongly active variables and fixes them after finitely many iterations. Novelties compared to the literature include a new quadratic limited-memory model for progressing in a subspace and safeguards for the line search in finite precision arithmetic.

Numerical results for small and large unconstrained and bound constrained CUTEst problems [32] show that compared to other state of the art, LMBOPT ranks highest according to several criteria.

The paper is organized as follows. In Section 1.3 we give a list of all algorithms defined in present paper whose in-out dependence are compiled as a data structure. We use some notations for implementation of the robust version of bent line search algorithm [51] in Section 2. In Section 3, we describe how to implement the subspace step. The master algorithm is introduced in Section 4.6, and some results are given in Section 5.

### 1.3 Algorithms and data structures

The LMBOPT solver solves a bound constrained optimization problem with continuously differentiable objective function, using routines for evaluating the function and the gradient. It uses beyond the theory in [51], a new limited memory quasi Newton method and the robustified bent line search method. It is followed as follows:

<table>
<thead>
<tr>
<th>Step</th>
<th>dependencies</th>
</tr>
</thead>
<tbody>
<tr>
<td>LMBOPT</td>
<td>Preprocessor, Determiner, Updater, Postprocessor</td>
</tr>
<tr>
<td>Preprocessor</td>
<td>Initializer, Improver, Problem object</td>
</tr>
<tr>
<td>Determiner</td>
<td>Reducer, Worker selector, Successor, Unsuccessor</td>
</tr>
<tr>
<td>Updater</td>
<td>Worker, Info, Subspace</td>
</tr>
<tr>
<td>Successor</td>
<td>Subspace selector, Director, Problem object</td>
</tr>
<tr>
<td>Problem object</td>
<td>Generator, Adjuster</td>
</tr>
</tbody>
</table>
Table 1: Mathematical dependency graph of \textit{LMBOPT}

<table>
<thead>
<tr>
<th>Director</th>
<th>Local solvers, Conjugator</th>
</tr>
</thead>
</table>
| Conjugator   | Robustifier I, Gamma, Regularizer  
Conjugate gradient direction |
| Unsuccessor  | Enforcer, Bender, Nullifier |
| Nullifier    | Neighbourhood, Problem object |
| Bender       | Robustifier I, Bent line search, Robustifier II  
Problem object |

The top levels. \textit{LMBOPT} calls a Preprocessor to initialize all necessary information, then alternates calls to a Determiner and an Updater. Once the norm of reduced gradient in the current best point is below a given threshold, it ends up. Finally, it calls a Postprocessor to prepare the output.

Preprocessor uses an Initializer initializing the subspace and other necessary information, then calls an Improver to improve the starting point, and calls a Problem object to compute and adjust the function value and the gradient vector.

Determiner includes a Reducer computing the reduced gradient, a Worker selector changing or keeping the free index set $I_-(x)$, a Successor containing the successful iterations and an Unsuccessor containing the unsuccessful iterations.

Updater calls a Worker generating the working set (the free index set), an Info updating all necessary information such as the best point, and a Subspace updating the subspace and quasi Newton.

The lower levels. Successor first calls a Subspace selector to determine the type of subspace and then uses a Director to compute the direction. Afterwards, it uses a Conjugator producing the conjugate gradient method.

Director calls local solvers to compute the search direction such as a new limited memory quasi Newton and then uses a Conjugator generating the conjugate gradient direction.

Problem object calls possibly many times a Generator to compute the function value in each iteration and only once in each iteration to compute the gradient vector. Afterwards, it calls an Adjuster to adjust the gradient vector.

Conjugator contains a Robustifier I finding a good starting step size, a Gamma calculating $\gamma$, a Regularizer doing a regularization for numerical stability, and a conjugate gradient direction.

UnSuccessor tries to enforce the angle condition by an Enforcer, then calls a robust bent line search method to update the best point, and uses a Nullifier avoiding too many null steps.
Nullifier calls a Neighbourhood to generate a point around the current (previous) best point and then a Problem object to compute and adjust the function value and gradient vector.

Bender calls a Robustifier I to find a good step size and performs a bent line search along a regularized direction. Afterwards, it calls a Robustifier II to obtain the robust step size and then computes and adjusts the function value and the gradient vector.

<table>
<thead>
<tr>
<th>Subalgorithm</th>
<th>Subalgorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Initializer</strong></td>
<td>initInfo</td>
</tr>
<tr>
<td><strong>Improver</strong></td>
<td>projStartPoint</td>
</tr>
<tr>
<td><strong>Determiner</strong></td>
<td>getSuccess</td>
</tr>
<tr>
<td><strong>Working selector</strong></td>
<td>findFreePos</td>
</tr>
<tr>
<td><strong>Worker</strong></td>
<td>findFreeNeg</td>
</tr>
<tr>
<td><strong>Info</strong></td>
<td>updateInfo</td>
</tr>
<tr>
<td><strong>Subspace</strong></td>
<td>updateSubspace</td>
</tr>
<tr>
<td><strong>Subspace selector</strong></td>
<td>typeSubspace</td>
</tr>
<tr>
<td><strong>Local solvers</strong></td>
<td>scaleDir, quasiNewtonDir, AvoidZigzagDir</td>
</tr>
<tr>
<td><strong>Generator</strong></td>
<td>fun, dfun</td>
</tr>
<tr>
<td><strong>Adjuster</strong></td>
<td>adjustGrad</td>
</tr>
<tr>
<td><strong>Reducer</strong></td>
<td>redGrad</td>
</tr>
<tr>
<td><strong>Robustifier I</strong></td>
<td>goodStep</td>
</tr>
<tr>
<td><strong>Gamma</strong></td>
<td>getGam</td>
</tr>
<tr>
<td><strong>Regularizer</strong></td>
<td>regDenom</td>
</tr>
<tr>
<td><strong>Conjugate gradient direc</strong></td>
<td>ConjGradDir</td>
</tr>
<tr>
<td><strong>Enforcer</strong></td>
<td>enforceAngle</td>
</tr>
<tr>
<td><strong>Nullifier</strong></td>
<td>nullStep</td>
</tr>
<tr>
<td><strong>bent line search</strong></td>
<td>BLS</td>
</tr>
<tr>
<td><strong>Robustifier II</strong></td>
<td>robustStep</td>
</tr>
</tbody>
</table>

Table 2: The lowest level

The subalgorithms of LMBOPT are listed in Table 3. They depend on one or more data structures point, step, tune, par and info according to the input/output list indicated. These data structures themselves are briefly described in Table 4.

<table>
<thead>
<tr>
<th>Algorithm 2.1</th>
<th>function [step] = goodStep(point, step, tune);</th>
</tr>
</thead>
<tbody>
<tr>
<td>goodStep</td>
<td>Try to find the starting good step size</td>
</tr>
<tr>
<td>Algorithm 2.2</td>
<td>function [point, step] = robustStep(point, step, tune);</td>
</tr>
</tbody>
</table>

9
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>robustStep</strong></td>
<td>Try to find a point with smallest robust change</td>
</tr>
<tr>
<td>Algorithm 2.3</td>
<td><code>function [point, step, info] = BLS(fun, point, step, tune, info);</code></td>
</tr>
<tr>
<td><strong>BLS</strong></td>
<td>Find a step size $\alpha$ satisfying a sufficient descent condition</td>
</tr>
<tr>
<td>Algorithm 2.4</td>
<td><code>function [point, step, par, info] = ... nullStep(fun, point, step, par, tune, info);</code></td>
</tr>
<tr>
<td><strong>nullStep</strong></td>
<td>Try to prevent producing the null steps</td>
</tr>
<tr>
<td>Algorithm 3.1</td>
<td><code>function [point] = adjustGrad(point, tune);</code></td>
</tr>
<tr>
<td><strong>adjustGrad</strong></td>
<td>Adjust the gradient vector</td>
</tr>
<tr>
<td>Algorithm 3.2</td>
<td><code>function [point] = redGrad(point);</code></td>
</tr>
<tr>
<td><strong>redGrad</strong></td>
<td>Compute the reduced gradient</td>
</tr>
<tr>
<td>Algorithm 3.3</td>
<td><code>function [point, par] = findFreePos(point, par, tune);</code></td>
</tr>
<tr>
<td><strong>findFreePos</strong></td>
<td>Update the working set</td>
</tr>
<tr>
<td>Algorithm 3.4</td>
<td><code>function [point, par, info] = findFreeNeg(point, par, tune, info);</code></td>
</tr>
<tr>
<td><strong>findFreeNeg</strong></td>
<td>Find the free index set</td>
</tr>
<tr>
<td>Algorithm 3.5</td>
<td><code>function [point] = updateSubspace(point, step, par, tune);</code></td>
</tr>
<tr>
<td><strong>updateSubspace</strong></td>
<td>Update the subspace information</td>
</tr>
<tr>
<td>Algorithm 3.7</td>
<td><code>function [step] = enforceAngle(point, step, par, tune);</code></td>
</tr>
<tr>
<td><strong>enforceAngle</strong></td>
<td>Enforce the angle condition</td>
</tr>
<tr>
<td>Algorithm 3.8</td>
<td><code>function [point, step] = quasiNewtonDir(point, step);</code></td>
</tr>
<tr>
<td><strong>quasiNewtonDir</strong></td>
<td>Compute quasi Newton direction</td>
</tr>
<tr>
<td>Algorithm 3.9</td>
<td><code>function [step, par] = scaleDir(point, step, par);</code></td>
</tr>
<tr>
<td><strong>scaleDir</strong></td>
<td>Choose components of sensible sign and scale</td>
</tr>
<tr>
<td>Algorithm 3.10</td>
<td><code>function [step] = AvoidZigzagDir(point, step, tune, info);</code></td>
</tr>
<tr>
<td><strong>AvoidZigzagDir</strong></td>
<td>Modify the direction to avoid zigzagging</td>
</tr>
<tr>
<td>Algorithm 3.11</td>
<td><code>function [point, step, par] = searchDir(point, step, par, tune, info);</code></td>
</tr>
<tr>
<td><strong>searchDir</strong></td>
<td>Construct starting trial search direction</td>
</tr>
<tr>
<td>Algorithm 3.12</td>
<td><code>function [point, step, par, info] = ... getGam(fun, point, step, tune, par, info);</code></td>
</tr>
<tr>
<td><strong>getGam</strong></td>
<td>Compute $\gamma$</td>
</tr>
<tr>
<td>Algorithm 3.13</td>
<td><code>function [par] = regDenom(point, step, par, tune);</code></td>
</tr>
<tr>
<td><strong>regDenom</strong></td>
<td>Construct regularize denominator</td>
</tr>
<tr>
<td>Algorithm 3.14</td>
<td><code>function [point, step, par, info] = ... ConjGradDir(fun, point, step, par, tune, info);</code></td>
</tr>
<tr>
<td><strong>ConjGradDir</strong></td>
<td>Construct the conjugate gradient direction</td>
</tr>
<tr>
<td>Algorithm 4.1</td>
<td><code>function [point] = projStartPoint(point, tune);</code></td>
</tr>
</tbody>
</table>

10
### Table 3: List of algorithms defined in present paper. The main algorithm *LMBOPT* solves a bound constrained problem; the others are called within *LMBOPT*.

<table>
<thead>
<tr>
<th>Table 3: List of algorithms defined in present paper. The main algorithm <em>LMBOPT</em> solves a bound constrained problem; the others are called within <em>LMBOPT</em>.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>projStartPoint</strong></td>
</tr>
</tbody>
</table>
| Algorithm 4.2 | **function** [point, step, par, info] = ···

getSuccess(fun, point, step, par, tune, info); |
| getSuccess | Determine whether subspace iteration is successful or not |
| Algorithm 4.3 | **function** [point] = initInfo(point, tune); |
| initInfo | Initialize best point and factor for adjusting acceptable increase in $f$ |
| Algorithm 4.4 | **function** [point, par] = updateInfo(point, par, tune, info); |
| updateInfo | Update best point and factor for adjusting acceptable increase in $f$ |
| Algorithm 4.7 | **function** [point] = LMBOPT(point, step, tune, par); |
| LMBOPT | Minimize smooth $f(x)$ subject to $x \in x = [\underline{x}, \overline{x}]$ |

---

**fun** and **dfun** (structure with information about function handle)

<table>
<thead>
<tr>
<th>Table 3: List of algorithms defined in present paper. The main algorithm <em>LMBOPT</em> solves a bound constrained problem; the others are called within <em>LMBOPT</em>.</th>
</tr>
</thead>
</table>
| **point** (structure with information about function handle)

$x, f, g$ (old point, its function value and gradient vector)

$x_{new}, f_{new}, g_{new}$ (newest point, its function value and gradient vector)

$x_{best}, f_{best}$ (best point and its function value)

$x_{init}, f_{init}$ (starting point and its function value)

$\underline{x}, \overline{x}$ (lower and upper bound)

$y$ (the difference of current gradient with its old one; $g_{new} - g$)

$I$ (working set), $I_+$ (the set of free or freeable indices), $I_-$ (the set of new free indices)

$m$ (subspace dimension), $mf$ (memory for $Df$), $ch$ (counter for $m$)

$m_0$ (the length of subspace), $Df$ (list of $mf$ acceptable increase in $f$)

$S$ (a list of $m$ previous search directions), $Y$ (a list of $m$ vectors $y_1, \ldots, y_m$)

$H$ (Hessian matrix), $q$ (extrapolation factor)

$df$ (acceptable increase in $f$), $\Delta f$ (factor for adjusting $df$) |
| **step** (structure with information about the step management)

$p_{init}$ (starting search direction in each iteration), $p$ (Krylov search direction), $gp$ ($g^T p$)

$\alpha_{good}$ (the starting step-size generated by *goodStep*), $s$ (search direction; $x_{new} - x$)

$\mathcal{A}$ (list of some step-sizes generated by *BLS*) |
| **tune** (structure with fixed parameters for tuning the performance)

$\varepsilon$ (accuracy for reduced gradient), $m$ (subspace dimension), $mf$ (memory for $Df$)

$\Delta_x$ (tiny factor for interior move), $\Delta_u$ (factor for adjusting $\overline{x}$)

$\Delta_g$ (factor for adjusting gradient), $\Delta_{angle}$ (regularization angle)

$\Delta_w$ (for guaranteeing $w > 0$), $\Delta_r$ (factor for finding almost flat step)

$\Delta_{pg}$ (tiny factor for regularizing $g^T p$ in *ConjGradDir*)

$\Delta_{reg}$ (tiny factor for regularizing $g^T p$ in *BLS*) |
\( \Delta \alpha \) (tiny factor for starting step), \( \Delta \beta \) (tiny factor for breakpoint)
\( \Delta H \) (tiny regularization factor for subspace Hessian)
\( \Delta m \) (tiny factor for regularizing \( \Delta f \) if not monotone)
\( \Delta po \) (gradient tolerance for skipping update)

**typeH** (choose update formula for Hessian (0 or 1))

**gfac** (parameter for scaling direction), **\theta** (parameter for adjusting the direction)

\( \beta > 0 \) (threshold for determining efficiency), **del** (parameter for null step)

**exact** (enforce exact line search on quadratics), **nnulmax** (iteration limit in null step)

\( \beta CG \) (threshold for efficiency of CG), **lmax** (iteration limit in efficient line search)

**nlf** (number of local steps before freeing is allowed)

**rfac** (restart after **rfac**\( n_l \) local steps), **facf** (relative accuracy of \( f \) in first step)

**nsmin** (how many stucks before taking special action?)

**nwait** (number of local steps before CG is started), **mdf** (parameters for updating \( df \))

**bis** (bisection: 0: geometric mean, 1: cubic, 2: geometric mean and cubic))

\( \zeta_{\min} \) and \( \zeta_{\max} \) (Safeguarded parameters for \( \zeta \) in *ConjGradDir*)

**mbis** (parameter for bisection), **nstuckmax** (iteration limit in number of stucks)

**par** (structure with parameters modified during the search)

**estuck** (a robust increase is counted as **success** if stuck enough)

**freeing** (parameter for finding appropriate free variables)

**flags** (null step ?), **cosine** (descent direction ?),

**monotone** (parameter for improvement on function values)

**CG** (parameter for determining the type of subspace)

**success** (successful/unsuccessful subspace iterations, 0 or 1)

**fixed** (parameter for changing activity), **nlocal** (number of local steps)

**nstuck** (number of stuck iterations), **null** (number of null steps)

**quad** (determine whether \( f \) is close to quadratic or not)

**hist** (list of at most \( m \) subspace basis)

**perm** (permute subspace basis so that oldest column is first)

**firstAngle** (calling *enforceAngle*: 1: first call, 0: second call)

**info** (structure with information about the info management)

**nf** (number of function evaluations), **ng** (number of gradient evaluations)

**nsub** (number of successful iterations), **nfmax** (maximal number of function evaluations)

**ngmax** (number of gradient evaluations), **nf2gmax** (**nfmax** + 2**ngmax**)

**eff** (efficiency status for **BLS**), **nstuck** (number of stuck iterations)

---

**Table 4**: Global data structures for the algorithms of the present paper
2 A robust bent line search

A bent line search along the lines proposed by Neumaier & Azmi [51] is used to project the ray obtained by a search direction into the bound constraints and to impose a sufficient descent condition.

2.1 The starting step size

If the step size is too small, rounding errors will often prevent in practice that the function value is strictly decreasing. Due to cancellation of leading digits, the Goldstein quotient can become very inaccurate, which may lead to a wrong bracket and then to a failure of the line search. The danger is particularly acute when the search direction is almost orthogonal to the gradient. Hence, before doing each line search method, we need to produce a starting step-size by a method we call `goodStep`. It works as follows:

- The minimum of the lower and upper breakpoints is computed in finite precision arithmetic, whose the corresponding bounds is guaranteed to be active, and updated due to roundoff error.
- The minimal step size is found by a heuristic process and then the target step size is chosen.
- The role of boolean variable `exact` is to enforce at the second trial step an exact line search on quadratics.
- When the good step size $\alpha_{\text{good}}$ equals with the minimum of the breakpoints, adverse finite-precision effects are avoided.
- If the number of stuck iterations reached its limit, the trial step is increased by the factor $2 \cdot n_{\text{stuck}}$ to avoid remaining stuck.

2.1 Algorithm. (goodStep)

<table>
<thead>
<tr>
<th>Purpose: Try to find the starting good step size</th>
</tr>
</thead>
<tbody>
<tr>
<td>function [step]= goodStep(point, step, tune);</td>
</tr>
<tr>
<td>ind = {i</td>
</tr>
<tr>
<td>if (ind $\neq \emptyset$), $\alpha_{\text{break}} = \min{(\pi_i - x_i)/p_i</td>
</tr>
<tr>
<td>ind = {i</td>
</tr>
<tr>
<td>if (ind $\neq \emptyset$), $\alpha_{\text{break}} = \min{(\pi_i - x_i)/p_i</td>
</tr>
<tr>
<td>$\alpha_{\text{break}} = \min(\alpha_{\text{break}}, \alpha_{\text{break}});$ $\alpha_{\text{break}} = \alpha_{\text{break}}(1 + \Delta_b);$</td>
</tr>
<tr>
<td>% define minimal step size</td>
</tr>
<tr>
<td>ind = {i</td>
</tr>
<tr>
<td>if ($x = 0$ &amp; ind $\neq \emptyset$), $\alpha_{\text{min}} = \Delta_{\alpha}</td>
</tr>
<tr>
<td>else,</td>
</tr>
<tr>
<td>if (ind $\neq \emptyset$), $\alpha_{\text{min}} = \Delta_{\alpha} \max{</td>
</tr>
<tr>
<td>else, $\alpha_{\text{min}} = 1;$ $\alpha_{\text{good}} = 1; \text{ return; }$ % zero direction</td>
</tr>
<tr>
<td>end;</td>
</tr>
<tr>
<td>end;</td>
</tr>
</tbody>
</table>

Continued on next page
\[ \alpha_{\text{target}} = \max(\alpha_{\min}, \frac{\text{df}}{|g|p}); \]

if exact, \( \alpha_{\text{target}} = \min(\alpha_{\text{target}}, \alpha_{\text{break}}) \)

end; if \((q\alpha_{\text{target}} \leq \alpha_{\text{break}}), \alpha_{\text{good}} = \alpha_{\text{target}}; \) else, \( \alpha_{\text{good}} = \max(\alpha_{\min}, \alpha_{\text{break}}) \);

end; if \((n_{\text{stuck}} \geq n_{\text{min}}), \alpha_{\text{good}} := 2(n_{\text{stuck}})\alpha_{\text{good}}; \) end;

### 2.2 A robustified step size (\textit{robustStep})

If the line search fails to give an improvement on the function values, we find a point with small significant change by performing the following algorithm:

#### 2.2 Algorithm. Robusted step size (\textit{robustStep})

<table>
<thead>
<tr>
<th>Purpose:</th>
<th>Try to find a point with smallest robust change</th>
</tr>
</thead>
<tbody>
<tr>
<td>function</td>
<td>\textit{point, step} = \textit{robustStep}(point, step, tune);</td>
</tr>
</tbody>
</table>
| \( \text{dF}_b \) | = \text{min}(\text{dF}); \( i_{\text{rob}} = \{i \mid \text{dF}_i = \text{dF}_b \} \)
| if \( (\text{dF}_b < 0) \) |, \( \alpha_{\text{new}} = A_{i_{\text{rob}}}; \text{f}_{\text{new}} = f + \text{dF}_{i_{\text{rob}}}; \) \text{return}; end; |
| \% treat failed line search (no improvement); find point with smallest robust change | \text{ind} = \{i \mid \text{dF}_i > 0 \& \text{dF}_i < +\infty \}; \text{dF}_b = \text{min}(\text{dF}); i_{\text{new}} = \{i \in \text{ind} \mid \text{dF}_i = \text{dF}_b \}; i_{\text{rob}} = \text{ind}_{i_{\text{new}}}; |
| \% quality robust with robust change | if \( (\text{dF}_b \leq \text{df}) \), \( \alpha_{\text{new}} = A_{i_{\text{rob}}}; \text{f}_{\text{new}} = f + \text{dF}_{i_{\text{rob}}}; \) \text{return}; end; |
| \( \text{idF} \) | = \{i \mid \text{dF}_i \leq 0 \}; |
| if \( (i_{\text{new}} == \emptyset \) or \( \text{idF} \neq \emptyset ) \), \% function almost flat; take step with largest \( \text{dF} \) | \text{ind} = \{i \mid \text{dF}_i < +\infty \}; \text{dF}_b = \text{min}(\text{dF}); i_{\text{new}} = \{i \in \text{ind} \mid \text{dF}_i = \text{dF}_b \}; i_{\text{rob}} = \text{ind}_{i_{\text{new}}}; |
| \% function is flat; take largest step | if \( (\text{dF}_b \leq 0) \), \( \alpha_{\text{new}} = \max(A); i_{\text{new}} = \{i \in \text{ind} \mid A_i = \alpha_{\text{new}} \}; i_{\text{rob}} = \text{ind}_{i_{\text{new}}}; \) end; |
| else \% take largest almost flat step | if \( (\text{dF}_b > \Delta_r \text{df}) \) \text{ind} = \{i \mid \text{dF}_i \leq \text{df} \}; \alpha_{\text{new}} = \max(A); i_{\text{new}} = \{i \in \text{ind} \mid A_i = \alpha_{\text{new}} \}; i_{\text{rob}} = \text{ind}_{i_{\text{new}}}; \) end; |
| \( \alpha_{\text{new}} = A_{i_{\text{rob}}}; \text{f}_{\text{new}} = f + \text{dF}_{i_{\text{rob}}}; \) |

\textit{robustStep} first needs to check whether there exists an improvement on the function value or not; if there is no improvement then it tries to find a point with smallest robust change. There would be found a point with robust change if the minimum of \( \text{dF} \) is smaller than or equal \( \text{df} \). Otherwise, if the function is almost flat or flat; then a step with largest \( \text{df} \) is chosen. Otherwise, a point with nonrobust change might be chosen provided that the minimum of \( \text{df} \leq \Delta_r \text{df} \).
2.3 The bent line search (BLS)

The bent line search BLS is a variant of the curved line search of Neumaier & Azmi [51] with enhancements for numerical stability.

- At first, the acceptable increase \( \Delta f \) in \( f \) is updated.
- A regularized directional derivative is used.
- \textit{goodStep} is recalled to find the starting good step-size \( \alpha_{\text{good}} \), the target step size \( \alpha_{\text{target}} \) and the minimum step size \( \alpha_{\min} \).
- Change to find a step size \( \alpha > 0 \) satisfying the \textbf{sufficient descent condition}

\[
\mu(\alpha)|\mu(\alpha) - 1| \geq \beta \tag{11}
\]

with fixed \( \beta > 0 \), where

\[
\mu(\alpha) := \frac{f(x(\alpha)) - f(x)}{\alpha^2 g(x)^T p} \quad \text{for} \quad \alpha > 0 \tag{12}
\]

is called the \textbf{Goldstein quotient} (Goldstein [31]). (11) enforces that \( \mu(\alpha) \) is neither too close to one nor sufficiently positive. It prevents the step sizes which are too long or too small, leading to convergence.

- Once the sufficient descent condition holds, it ends, giving an efficient line search.

- In the first iteration if \( \mu < 1 \), the secant step for the Goldstein quotient is used. Otherwise an extrapolation is done by the factor \( q > 1 \). In the next iteration, if the Goldstein quotient doesn’t satisfy, then the function is far from the quadratic and bounded. In such a case, either an interpolate is performed if the lower bound for step-size is zero or an extrapolation is done by the factor \( q > 1 \) until once a bracket \([a, b]\) is found. Then, either the geometric mean or the cubic bisection or the geometric mean alternated with the \textit{mbis} cubic bisection is used.

- A limit on the number of iterations is used.

- At the end, \textit{robustStep} is used to robust the step size if the line search is not efficient.

Moreover, two arrays \( \mathcal{A} \) and \( \mathcal{dF} \) use to restore step sizes and gains, respectively. The variable \( \text{eff} \) indicates what is the status of step, belonging to \{1, 2, 3, 4\}.

2.3 Algorithm. Bent line search (BLS)

| Purpose: Find a step size \( \alpha \) with \( \mu(\alpha)|\mu(\alpha) - 1| \geq \beta \) |
|---|
| \textbf{function} \([\text{point, step, info}] = \text{BLS}(\text{fun, point, step, tune, info});\) |
| if \((\text{ng} == 1), \Delta f = \Delta f;\) |
| else |
| \begin{align*}
| & \text{if}\ (\text{mod(ng,mdf)} == 0), \Delta f = \Delta f(|f| + 1); \text{ else, } \Delta f = \max(\Delta f); \text{ end; } \\
| & \text{end; } \\
| & \text{gp} = \min(g_j^T p_1, -\Delta p_2(|g_j|)|p_1|)) \% \text{ regularized directional derivative} \\
| & \text{Continued on next page}|
| 15 |
goodStep; % get $\alpha_{\text{good}}$

first = 1; descent = 0; rob = 0; eff = NaN; i = 0; $\alpha = 0; \bar{\alpha} = \infty$

$\alpha = \alpha_{\text{good}}$; $dF_1 = 0; A_1 = 0$

while 1,

\[ x_{\text{new}} = \max \{ x, \min \{ \bar{\alpha}, x_{\text{init}} + \alpha p \} \}; \]
\[ f_{\text{new}} = \text{fun}(\text{fun}, x_{\text{new}}); i = i + 1; \]
if (isnan($f_{\text{new}}$)), $f_{\text{new}} = +\infty$; end;

$\Delta F_{i+1} = f_{\text{new}} - f_{\text{init}}; A_{i+1} = \alpha; \mu = (f_{\text{init}} - \Delta F_{i+1})/(\alpha p)$;

if ($\mu|\mu - 1| \geq \beta$ or eff $\equiv 1$)

\[ \text{eff} = 1; \text{% line search efficient} \]
if $\sim$ exact, break; end;

elseif ($i > 1 \& \&$ descent $\& \&$ rob $> 0$)

\[ \text{eff} = 2; \text{% robust nonmonotone step accepted} \]
break;
elseif ($i \geq l_{\text{max}}$) % limit on function values reached

if descent, eff $\equiv 3$; % descent else, eff $\equiv 4$; % no descent end;
break;
end;
% update bracket
if descent
% update bracket for descent
if ($\mu \geq \frac{1}{2}$), $\alpha = \alpha$
else % linear decrease or more
\[ \bar{\alpha} = \alpha; \]
end;
elseif ($dF_{i+1} < 0$) % first descent step

descent $\equiv 1$
% create bracket for descent
\[ \text{ind} = \{ i \mid dF_i < 0 \}; \bar{\alpha} = \max(\mathcal{A}_{\text{ind}}); \text{rob} = -1; \text{% lower part} \]
\[ \text{ind} = \{ i \mid dF_i \geq 0 \& A_i > \bar{\alpha} \}; \]
if (ind $\equiv 0$), $\bar{\alpha} = +\infty$; else, $\bar{\alpha} = \min(\mathcal{A}_{\text{ind}})$; end;
else % no descent; update robust bracket
if ($dF_{i+1} \leq \Delta f$), $\alpha = \alpha; \text{rob} = dF_{i+1}$; else $\bar{\alpha} = \alpha_{\text{new}}$; end;
end;
if first, first $\equiv 0$; % first step
if ($\mu < 1$),
\[ \alpha = \frac{1}{2}\alpha/(1 - \mu); \text{% secant step for Goldstein quotient} \]
if ($\alpha \equiv 0$), $\alpha = \alpha_{\text{min}}$; end;
else, $\alpha = \max\{\alpha_{\text{min}}, q\alpha\}$; % extrapolation

Continued on next page

16
end:
  exact = 0;
else
  if (α == ∞), α = αq; % extrapolation
  elseif (α == 0), α = \(\frac{1}{2}\alpha/(1-\mu)\); % contraction
  else
    switch bis
      case 0 % geometric mean bisection
        α0 = max{α, αmin}; α = \(\sqrt{α0/\tau}\);
      case 1 % cubic bisection
        α0 = max{α, αmin}; α = \(\alpha(\tau/α0)^{1/3}\);
      case 2 % geometric mean and cubic bisection
        gc = mod(nf, mbis); α0 = max{α, αmin};
        if gc, α = \(\alpha\sqrt{\tau/α0}\), else, α = \(\alpha(\tau/α0)^{1/3}\); end;
    end;
  end;
  α = min(α, αmax);
end;
nf = nf + i;
robustStep; % robust step size

2.4 Avoiding too many null steps (nullStep)

If at least \texttt{nnulmax} null steps were found, \texttt{nullStep} algorithm tries to get rid of this weakness, depending on the output parameter \texttt{eff} in \texttt{BLS}. If the maximal number of function evaluations for \texttt{BLS} was exceeded, \texttt{eff}=4, a point around the old best point is generated instead of the point obtained by \texttt{BLS}. Otherwise a point around the current best point generated by \texttt{BLS} is constructed.

2.4 Algorithm. (nullStep)

<table>
<thead>
<tr>
<th>Purpose: Try to prevent producing the null steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textbf{function} [point, step, par, info] = nullStep(point, step, tune, par, info);</td>
</tr>
<tr>
<td>flags = (|s| == 0);</td>
</tr>
<tr>
<td>if (nnul &gt; 2 &amp; flags)</td>
</tr>
<tr>
<td>if (eff == 4), (\hat{x} = \max(x, \min(x_{\text{best}}(1-\text{del}), \tau)));</td>
</tr>
<tr>
<td>else, (\hat{x} = \max(x, \min(x(1-\text{del}), \tau)));</td>
</tr>
<tr>
<td>end;</td>
</tr>
<tr>
<td>ind = {i</td>
</tr>
</tbody>
</table>

Continued on next page
if flags, nnull = nnull + 1; else, nnull = 0; end;
f = fun(x); nf = nf + 1;
if isnan(f), f = +∞; end; % adjust f
end;

3 Working set and search directions

In this section, we give a description of the search direction used at each iteration. First we ignore the bound constraints and assume that the problem is unconstrained.

The starting trial search direction $p_{\text{init}}$ can be computed by an arbitrary local method. Then the search direction $p_{\text{init}}$ will be improved to be a direction in an adequate subspace by approximating the solution $p$ of the problem

$$\min \{ f(x + p) | p \in \text{Span}(S, p_{\text{init}}) \}. \quad (13)$$

3.1 The reduced gradient

Before computing the reduced gradient, we adjust the components of the gradient that are $\infty$ or NaN.

3.1 Algorithm. (adjustGrad)

<table>
<thead>
<tr>
<th>Purpose: Adjust the gradient vector $g$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>function</strong> [point] = adjustGrad(point, tune);</td>
</tr>
<tr>
<td>ind = {i</td>
</tr>
<tr>
<td>if ind \not= \emptyset, % NaN in gradient</td>
</tr>
<tr>
<td>ind1 = {i</td>
</tr>
<tr>
<td>ind2 = (ind &amp; ind1); $g_{\text{ind2}} = \Delta y \ast \text{ones}(\text{length}(\text{ind2}), 1)$;</td>
</tr>
<tr>
<td>ind3 = (ind &amp; \sim ind1); $g_{\text{ind3}} = -\Delta y \ast \text{ones}(\text{length}(\text{ind3}), 1)$;</td>
</tr>
<tr>
<td>end;</td>
</tr>
<tr>
<td>ind = {i</td>
</tr>
<tr>
<td>if ind \not= \emptyset, $g_{\text{ind}} = \Delta y \ast \text{ones}(\text{lind}, 1)$; end;</td>
</tr>
<tr>
<td>ind = {i</td>
</tr>
<tr>
<td>if ind \not= \emptyset, $g_{\text{ind}} = -\Delta y \ast \text{ones}(\text{lind}, 1)$; end;</td>
</tr>
</tbody>
</table>

The following algorithm shows how to compute the reduced gradient:

3.2 Algorithm. (redGrad)
3.2 The working set

In order to determine the working set $I$, we use the algorithms of findFreePos and findFreeNeg. At the first iteration, findFreePos finds $I_+(x)$, considered as the working set. Then findFreeNeg finds the free index set $I_-$, determines freeing, and updates nlocal. If the number of new free index set is smaller than that of the old free index set, i.e., fixed = 1, then the free index set must be changed; hence nlocal = 0. Otherwise, nlocal will be restarted to avoid cycling or updated whenever iterations are unsuccessful. At the end, freeing is determined, while it holds if at least one of the following holds:

- There is no improvement on the function value.
- The number of the new free index set is greater than the old one.
- The maximal number of local steps before freeing is exceeded.

3.3 Algorithm. (findFreeNeg)

```matlab
Purpose: Find the free index set $I_-$

function [point, par] = findFreeNeg(point, par, tune);

% find free indices
I_- = {i | $x_i > \bar{x}$ & $x_i < \underline{x}$};
I_- = length(I_-); fixed = (nI_- < nI);
if fixed, nlocal = 0; % free index set changed
elseif (nstuck > 0) % avoid cycling
    if (nlocal > nwait + m), nlocal = nwait; else, nlocal = nlocal + 1; end;
elseif (~ quad) % restart
    if (nlocal ≥ nwait), nlocal = nwait; else, nlocal = nlocal + 1; end;
elseif (~ fixed), nlocal = nlocal + 1; % local
end;
freeing = (~ monotone or nI_- > nI or nlocal ≥ nlf);
```

If $I = I_+(x) \neq I_-(x)$, the iteration is called a freeing iteration. It is enforced in four different cases:

- **Corner**: All components of current point are active. In this case, $I_-(x)$ is empty.
- **Monotone** (monotone = 1): The current point improved the function value and the norm of
gradient restricted to $I_-(x)$ is below $\varepsilon$.

- **Nonmonotone** ($\text{monotone} = 0$): The current point did not improve the function value and the norm of gradient restricted to $I_-(x)$ is below $\varepsilon$.

- **Local**: the current point is an ordinary one and the norm of gradient restricted to $I_-(x)$ is not below $\varepsilon$.

In all cases, the working set $I$ is update by $I_+(x)$.

`findFreePos` tries to update the working set $I$ such that the condition (10) holds. In the first iteration, it finds the free indices set $I_+(x)$, which is used as the working set since $\text{freeing} = 0$. In the other iterations, $\text{freeing}$ determined by `findFreeNeg` in the last iteration is updated by `findFreePos`. If it holds, the free index set $I_+(x)$ is found and considered as the working set. Otherwise $I_-(x)$ generated by `findFreeNeg` in the previous iteration is kept as the working set.

### 3.4 Algorithm. (`findFreePos`)

<table>
<thead>
<tr>
<th>Purpose: Find the free index set and update working set</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>function [point, par, info] = findFreePos(point, par, tune, info);</code></td>
</tr>
<tr>
<td>% find free indices $I_+$ if freeing holds</td>
</tr>
<tr>
<td>$\rho = (1/\max(1, ng - 1))$; $\text{freeing} = (\text{freeing} \text{ or } |g_{\text{new}}|^2 &lt; \rho |g_{\text{red}}|^2)$;</td>
</tr>
<tr>
<td>if freeing, % freeing step: corner, monotone, nonmonotone and local</td>
</tr>
<tr>
<td>$I_+ = {i \mid (x_i &gt; x_i^0 &amp; x_i &lt; x_i^r) \text{ or } (g_{\text{red}})<em>i \neq 0}; n</em>{I_+} = \text{length}(I_+);$</td>
</tr>
<tr>
<td>if (ng == 1 or $n_{I_+} &gt; n_I$), $I_+ = I_+; \text{nlocal} = 0; \text{end};$</td>
</tr>
<tr>
<td>end; % update working set</td>
</tr>
<tr>
<td>$I = I_-$; $n_I = \text{length}(I); \omega = |g_I|^2;$</td>
</tr>
</tbody>
</table>

### 3.3 Subspace information

Throughout our implementation, we define the matrix $S$ as a $n \times m$ matrix whose columns are (in the actual implementation a permutation of) the previous $m$ search directions,

$$S := \{s^1, ..., s^m\} = \{x^1 - x^0, ..., x^m - x^{m-1}\},$$

and the matrix $Y \in \mathbb{R}^{n \times m}$ the corresponding gradient differences

$$Y := \{y^1, ..., y^0\} = \{g^1 - g^0, ..., g^m - g^{m-1}\}.$$  (15)

One of column of both $S$ and $Y$ is updated whenever a new pair of $s$ and $y$ satisfies the Powell condition

$$|g^T y| \geq \Delta_{po} g^T g.$$  (16)

This condition is necessary to prove the convergence of `LMBOPT`; for more details see Theorems 5.1 and 7.2 in [51].
If the objective function is quadratic with (symmetric) Hessian \( B \) and gradient \( c \) and no rounding errors are made, the matrices \( S, Y \in \mathbb{R}^{n \times m} \) satisfy the quasi-Newton condition

\[ BS = Y. \tag{17} \]

Since \( B \) is symmetric,

\[ H := S^T Y = S^T B S \tag{18} \]

must be symmetric. If we calculate \( y = B p \) at the direction \( p \neq 0 \), we have the consistency relations

\[ h := S^T B p = Y^T p = S^T y, \]
\[ 0 < \gamma := p^T B p = y^T p = \frac{f(x + \alpha p) - f - \alpha g^T p}{\alpha^2 / 2}, \tag{19} \]

for all \( \alpha \in \mathbb{R} \). If the columns of \( S \) (and hence those of \( Y \)) are linearly independent then \( m \leq n \), and \( H \) is positive definite. Then the minimum of \( f(x + S z) \) with respect to \( z \in \mathbb{R}^m \) is attained at

\[ z_{\text{new}} := -H^{-1} c, \tag{20} \]

where \( c := S^T g \), and the associated point and gradient are

\[ x_{\text{new}} = x + S z_{\text{new}}, \quad g_{\text{new}} := g(x_{\text{new}}) = g + Y z_{\text{new}}, \]

and we have

\[ S^T g(x_{\text{new}}) = 0. \tag{21} \]

We may now cheaply form the augmented matrices

\[ S_{\text{new}} := (S \ s), \quad Y_{\text{new}} = G S_{\text{new}} = (Y \ y), \quad H_{\text{new}} = S_{\text{new}}^T G S_{\text{new}} = \begin{pmatrix} H & h \\ h^T & \gamma \end{pmatrix}, \]

and the augmented vectors

\[ c_{\text{new}} := S_{\text{new}}^T g_{\text{new}} = \begin{pmatrix} 0 \\ s^T g_{\text{new}} \end{pmatrix}, \]
\[ z_{\text{new}} := -H_{\text{new}}^{-1} c_{\text{new}}. \tag{22} \]

If the objective function is not quadratic, then \( H := S^T Y \) need not be symmetric since \( B \) is not symmetric. However, the update procedure \( \text{updateSubspace} \) always produces a symmetric \( H \) as long as there is no null step and either the Powell condition (16) holds or the number of local steps is greater than that of before \( \text{CG} \) is started.

But if the allowed memory for \( S \) and \( Y \) is used we replace the the oldest columns of \( S \) and \( Y \) by the new vectors of \( s \) and \( y \), respectively.

3.5 Algorithm. (\textit{updateSubspace})

<table>
<thead>
<tr>
<th>Purpose: Update the subspace information</th>
</tr>
</thead>
<tbody>
<tr>
<td>function [point] = updateSubspace(point, step, par, tune);</td>
</tr>
</tbody>
</table>

Continued on next page
3.4 The quasi-Newton direction

We construct a Hessian approximation of the form

\[ B = D + WXW^T, \]

for some symmetric matrix \( W \in \mathbb{R}^{n \times m} \) and some matrix \( X \in \mathbb{R}^{n \times m} \). Thus, temporarily, the additional assumption is made that \( B \) deviates from a diagonal matrix \( D \) by a matrix of rank at most \( m \). Under these assumptions, we reconstruct the Hessian uniquely from the data \( S \) and \( Y = GS = BS \), in a manifestly symmetric form that can be used (just like the LBFGS-B formula) as a surrogate Hessian even when this structural assumption is not satisfied.

3.6 Theorem. Let \( D \in \mathbb{R}^{n \times n} \) be diagonal, \( \Sigma \in \mathbb{R}^{n \times m} \) and \( U \in \mathbb{R}^{n \times m} \). Then (17) and (23) imply

\[ B = D + U\Sigma^{-1}U^T, \]

where \( U := Y - DS \) and \( \Sigma := U^TS \) is symmetric. The solution of \( Bp = -g \) is given in terms of the symmetric matrix

\[ M := U^TD^{-1}Y = \Sigma^{-1}, \]

by

\[ p = D^{-1}(Uz - g), \]

where is the solution of \( Mz = U^TD^{-1}g \).

Proof. The matrices \( U := Y - DS \) and \( \Sigma := U^TS \) are computable from \( S \) and \( Y \), and we have

\[ U = Y - DS = BS - DS = (B - D)S = WXW^TS, \]

and since \( B \) is symmetric, \( \Sigma = S^T(B - D)S \) is symmetric, too. Assuming that the \( m \times m \) matrix \( Z := XW^TS \) is invertible, we find \( W = UZ^{-1} \), hence \( Z = XZ^{-T}U^TS = XZ^{-T}\Sigma \). This product relation and the invertibility of \( Z \) imply that \( \Sigma \) is invertible, too, and we conclude that \( X = Z\Sigma^{-1}Z^T \), hence

\[ B = D + UZ^{-1}XZ^{-T}U^T = D + U\Sigma^{-1}U^T. \]

22
To apply it to the bound constrained case, we note that the first order optimality condition predicts the point \( x + p \), where the nonactive part \( p_I \) of \( p \) solves the equation

\[
B_{II}p_I = -g_I.
\]

Noting that

\[
B_{II} = D_{II} + U_I\Sigma^{-1}U_I^T,
\]

we find \( D_{II} + U_I\Sigma^{-1}U_I^T \), hence

\[
p_I = D_{II}^{-1}(U_I z - g_I),
\]

where \( z := -\Sigma^{-1}U_I^T p_I \). Now \( -\Sigma z = U_I^T p_I = U_I^T D_{II}^{-1}(U_I z - g_I) \), hence \( z \) solves the linear system

\[
(\Sigma + U_I^T D_{II}^{-1} U_I) z = U_I^T D_{II}^{-1} g_I.
\]

Given the symmetric matrix (18), we introduce the symmetric \( m \times m \) matrix

\[
M := \Sigma + U_I^T D_{II}^{-1} U_I = U_I^T S + U_I^T D_{II}^{-1} D_{II}^{-1} U_I = U_I^T D_{II}^{-1} (D_{II} S_I + U_I).
\]

and find \( z = M^{-1}U_I^T D_{II}^{-1} g_I \), hence

\[
p_I = D_{II}^{-1}(U_I z - g_I). \tag{24}
\]

**Enforcing the angle condition.** Given \( z \), we could compute the nonactive part of \( p \) from (24); however, this does not always lead to a descent direction. We therefore compute

\[
h := U_I z,
\]

and choose

\[
p_I = D_{II}^{-1}(h - t g_I) \tag{25}
\]

with a suitable factor \( t \in [0, 1] \).

Due to rounding error, a computed descent direction \( p \) may not satisfy the angle condition

\[
\frac{g^T p}{\sqrt{g^T g \cdot p^T p}} \leq -\Delta_{\text{angle}}. \tag{26}
\]

We add a multiple of the gradient to enforce the angle condition for the modified direction

\[
p_{\text{new}} = p - t g \tag{27}
\]

with a suitable factor \( t \geq 0 \); the case \( t = 0 \) corresponds to the case where \( p \) already satisfies the bounded angle condition. The choice of \( t \) depends on the three numbers

\[
\sigma_1 := g^T g > 0, \quad \sigma_2 := p^T p > 0, \quad \sigma := g^T p,
\]
these are related by the Cauchy–Schwarz inequality
\[ \sigma_{\text{new}} := \frac{\sigma}{\sqrt{\sigma_1 \sigma_2}} \in [-1, 1]. \]

We want to choose \( t \) such that the angle condition (26) holds with \( p_{\text{new}} \) in the place of \( p \)
\[ \frac{g^T p_{\text{new}}}{\sqrt{g^T g \cdot p_{\text{new}}^T p_{\text{new}}}} \leq -\Delta_{\text{angle}}. \quad (28) \]
holds. In terms of the \( \sigma_i \), this reads
\[ \frac{\sigma - t \sigma_1}{\sqrt{\sigma_1 (\sigma_2 - 2t \sigma + t^2 \sigma_1)}} \leq -\Delta_{\text{angle}}. \]

If \( \sigma_{\text{new}} \leq -\Delta_{\text{angle}} \), this holds for \( t = 0 \), and we make this choice. Otherwise we enforce equality, using Proposition 5.2 in [51]. This modifications of the direction \( p \) is done by \textit{enforceAngle}:

\section*{3.7 Algorithm. (enforceAngle)}

<table>
<thead>
<tr>
<th>Purpose:</th>
<th>Enforce the angle condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>function</td>
<td>( [\text{step}] = \text{enforceAngle}(\text{point}, \text{step}, \text{par}, \text{tune}); )</td>
</tr>
<tr>
<td>( \sigma = g_T^T p_I; )</td>
<td>% move away from maximizer or saddle point</td>
</tr>
<tr>
<td>if (( \sigma &gt; 0 ))</td>
<td>act = ( { i \in I \mid g_i p_i &gt; 0 }; ) ( (p_I)<em>{\text{act}} = -(p_I)</em>{\text{act}}; ) ( \sigma = -\sigma; ) end;</td>
</tr>
<tr>
<td>( \sigma_1 = g_I^T g_I; ) ( \sigma_2 = p_{\text{new}}^T p_I; ) ( \sigma_3 = \sigma_1 \sigma_2; ) ( \sigma_{\text{new}} = \sigma/\sqrt{\sigma_3}; )</td>
<td>if (( \sigma_{\text{new}} \leq -\Delta_{\text{angle}} ))</td>
</tr>
<tr>
<td>else</td>
<td>( w = (\sigma_3 \max(\Delta w, 1 - \sigma_{\text{new}}^2))/(1 - \Delta_{\text{angle}}^2); ) ( t = (\sigma + \Delta_{\text{angle}} \sqrt{w})/\sigma_1; )</td>
</tr>
<tr>
<td>( \text{if} \ (w &gt; 0 \ &amp; \ t \neq \pm \infty), \ p_I = p_I - ty_I; \ \text{else}, \ p_I = -g_I; ) end;</td>
<td>end;</td>
</tr>
</tbody>
</table>

The following algorithm computes (25) and calls \textit{enforceAngle} to enforce the angle condition:

\section*{3.8 Algorithm. (quasiNewtonDir)}

<table>
<thead>
<tr>
<th>Purpose:</th>
<th>Compute quasi Newton direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>function</td>
<td>( [\text{point}, \text{step}] = \text{quasiNewtonDir}(\text{point}, \text{step}); )</td>
</tr>
<tr>
<td>( YY = Y_I; \circ Y_I; ) ( SS = S_I; \circ S_I; ) ( d = \sqrt{(\sum_{i=1}^m YY_i)/(\sum_{i=1}^m SS_i)}; )</td>
<td></td>
</tr>
<tr>
<td>( \text{Ok} = { i \mid \text{isnan}(d_i) \ \text{or} \ d_i == 0 \ \text{or} \ d_i == \pm \infty }; ) ( d_{\text{Ok}} = 1; ) ( d = d(\cdot); )</td>
<td></td>
</tr>
<tr>
<td>( U = Y_I - d \circ S_I; ) ( M = (Y_I^T(Y_I//d)) - H; ) ( z = M \backslash (U^T(g_I//d)); ) ( p_I = (Uz - g_I)//d; ) \text{enforceAngle};</td>
<td></td>
</tr>
</tbody>
</table>
3.5 Conjugate gradient step

The conjugate gradient method chooses the direction $s$ in the subspace generated by typeSubspace, enforcing the conjugacy relation (37); thus making $H$ diagonal. Both conditions together determine $s$ up to a scaling factor: $s$ must be a multiple of

$$p := Sq + p_{\text{init}}$$

for some $q \in \mathbb{R}^m$, in which $p_{\text{init}}$ is a descent direction, computed by searchDir.

Then (37) requires

$$Hq = r := -Y^T p_{\text{init}},$$

and we must have

$$q = H^{-1} r.$$  

Afterwards, if there exists the subspace, $m_0 > 0$, the conjugate gradient direction is constructed by

$$p = -\zeta p_{\text{init}} + S(z_{\text{new}} + \zeta r),$$

for which

$$\zeta := \frac{g^T p_{\text{init}} + q^T z_{\text{new}}}{\gamma - qr}.$$  

Otherwise, both (31) and (32) can be reduced and reformulated by

$$p = -\zeta p_{\text{init}}, \quad \zeta := \frac{g^T p_{\text{init}}}{\gamma}.$$  

In (32), if the denominator of $\zeta$, $\gamma - qr$, is near zero, then $\zeta$ cannot be computed; hence we use a regularized computation. To do so, let us regularize $\gamma$ by

$$\gamma_{\text{new}} = \frac{|f(x + \alpha p_{\text{init}}) - f| + \alpha |g^T p_{\text{init}}|}{\alpha^2/2},$$

and then compute

$$\gamma - qr = \begin{cases} \gamma - qr + \Delta_H (\gamma_{\text{new}}/2 + |g^T r|) & \text{if } \gamma - qr \geq 0, \\ \gamma - qr - \Delta_H (\gamma_{\text{new}}/2 + |g^T r|) & \text{if } \gamma - qr < 0, \end{cases}$$

so that

$$\zeta_{\text{new}} := \frac{g^T p_{\text{init}} + q^T z_{\text{new}}}{\gamma - qr}, \quad p_{\text{new}} = -\zeta p_{\text{init}} + S(z_{\text{new}} + \zeta_{\text{new}} r).$$

The implementation of conjugate gradient direction. The value of $\gamma$ depends on the search direction. Here the searchDir algorithm is used for computing $\gamma$ including scaleDir, quasiNewtonDir and AvoidZigzagDir. It works as follows:

- If $ng = 1$, the starting search direction makes use of the gradient signs only, and has nonzero entries in some components that can vary. Each starting search direction is computed by scaleDir, which is as follows:

3.9 Algorithm. (scaleDir)
Purpose: Choose components of sensible sign and scale

function [step, par] = scaleDir(point, step, par);

for i = 1 : n,
    if (x_i == 0),
        sc = min(1, x_i - x_i); % width defines a scale
        if (g_i < 0), p_i = sc; else, p_i = -sc; end;
    else
        sc = |x_i|; % x_i defines a scale
        if (x_i == x_i), p_i = sc;
        elseif (x_i == f_i), p_i = -sc;
        elseif (g_i < 0), p_i = sc;
        else, p_i = -sc;
    end;
end;

• If nlocal ≠ nwait, a modified direction is used to avoid zigzagging by AvoidZigzagDir. It is easily obtained that such a direction will be a descent direction and then a line search along with the direction p_init can be performed. Zigzagging is the main source of inefficiency of simple methods such as steepest descent. Any search direction p must satisfy gT p < 0. In order to avoid zigzagging we choose the search direction p as the vector with a fixed value gT p = −γ < 0 closest (with respect to the 2-norm) to the previous search direction. By Theorem 7.1 in [51],

\[ p = p_{old} - \hat{\lambda} g, \] (34)

where

\[ \hat{\lambda} = \frac{\tau + g^T p_{old}}{g^T g}. \] (35)

Rescaling p_{old} by a factor β > 0, the improved direction is expressed by

\[ p := \beta p_{old} - \lambda g, \] (36)

in which

\[ \lambda := \frac{\tau + \beta g^T p_{old}}{g^T g}. \]

Since gT p = −τ, the direction will be a descent direction. This direction is computed by AvoidZigzagDir, using a heuristic choice of

\[ \beta := 1/(a \ell + b)^\theta, \]

with tuning parameters satisfying 0 < θ < 1, a, b > 0.

3.10 Algorithm. (AvoidZigzagDir)
Purpose: Modify the direction to avoid zigzagging

\[
\begin{align*}
\text{function } & \text{ AvoidZigzagDir}(\text{point, step, tune, info}); \\
\gamma &= \max(g_y, 1); \quad \bar{\beta} = 1/(1 + nf + 3ng)^{\theta}; \quad \bar{\lambda} = (\gamma + \bar{\beta} gp)/\omega; \quad p_I = \bar{\beta} p - \bar{\lambda} g_I;
\end{align*}
\]

• If \( ng > 1 \) and \( n_{local} = n_{wait} \), \( quasiNewtonDir \) is used in subspace. Finally, by changing the sign of \( g \), we may enforce \( g_T p_{init} \leq 0 \). Even though \( g \neq 0 \), cancellation may lead to a tiny \( g_T p_{init} \) (and even of the wrong sign). Given the tiny parameter \( \Delta_{pg} \), to overcome this weakness, subtract \( \Delta_{pg} |g_T| p_{init} \) can be a bound on the rounding error to have the theoretically correct sign.

We now compute \( p_{init} \) by the following algorithm:

### 3.11 Algorithm. (searchDir)

Purpose: Construct the search direction \( p_{init} \)

**function** \( \text{searchDir}(\text{point, step, par, tune, info}); \)

```
if (ng == 1), scaleDir; CG = 1; % scaling direction
elseif (nlocal == nwait) % quasi-Newton direction
    quasiNewtonDir;
else % try to avoid zigzagging
    AvoidZigzagDir;
end;

gp = g_T p_I;
if (gp >= 0), J = \{ i \in I | p_i g_i > 0 \}; \quad (p_I)_j = -(p_I)_j; \quad gp = g_T p_I; \quad \text{end};
ok = (gp <= \Delta_{pg} |g_I| p_I);\nif ok, p_I = -g_I; \quad gp = p_I^T g_I; \quad \text{end};
p_{init} = \text{zeros}(n, 1); \quad p_{init} = p_I;
```

Using \( p_{init} \) computed by \text{searchDir}, \( \gamma \) is generated by the following algorithm:

### 3.12 Algorithm. (getGam)

Purpose: Compute \( \gamma \) (19)

**function** \( \text{getGam}(\text{point, step, par, info}); \)

```
df = \Delta_f; \quad \text{% goodStep needs to it for getting } \alpha_{\text{target}}
goodStep; \quad \text{% get } \alpha_{\text{good}}

x_{new} = \max(x, \min(x + \alpha_{\text{good}} p_{init}, \overline{\tau})); \quad f_{new} = \text{fun}(x_{new}); \quad nf = nf + 1;
```

Continued on next page
if isnan(fnew), fnew = +∞; end; % adjust f
d2f = |fnew - f - αgood * gp| + eps; γ = 2 * d2f/α²good;

The mentioned regularized computation is implemented by the following algorithm:

3.13 Algorithm. (regDenom)

<table>
<thead>
<tr>
<th>Purpose: Construct regularize denominator</th>
</tr>
</thead>
<tbody>
<tr>
<td>function [par] = regDenom(point, step, par, tune);</td>
</tr>
<tr>
<td>e2f =</td>
</tr>
<tr>
<td>if (denom ≥ 0), denom = denom + dcor; else, denom = denom - dcor; end;</td>
</tr>
</tbody>
</table>

Finally, the implementation of ConjGradDir for computing p in (31) is given next. The subprogram ConjGradDir

• computes γ by calling getGam and then the krylov direction,
• computes the subspace direction if there exists the subspace, m₀ > 0,
• constructs the regularize denominator of (33) by calling regDenom,
• projects the new point into the box x.

3.14 Algorithm. (ConjGradDir)

<table>
<thead>
<tr>
<th>Purpose: Construct the conjugate gradient direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>function [point, step, par, info] = ConjGradDir(fun, point, step, par, tune, info);</td>
</tr>
<tr>
<td>getGam; % compute γ</td>
</tr>
<tr>
<td>% construct r, q and z_new</td>
</tr>
<tr>
<td>if (m₀ &gt; 0) % subspace step possible</td>
</tr>
<tr>
<td>c = Σ g ∘ S_i; q = Σ p_{init} ∘ Y_i; rhs = [-c, q]; Hoh = H_{hist,hist};</td>
</tr>
<tr>
<td>sol = Hoh\rhs; nsub = {i</td>
</tr>
<tr>
<td>if nsub ≠ ∅ % no subspace step possible</td>
</tr>
<tr>
<td>ζ = gp/γ;</td>
</tr>
<tr>
<td>if isnan(ζ), ζ = ζ_{max}; end;</td>
</tr>
<tr>
<td>ζ = min(ζ_{max}, max(ζ_{min})); cosine = -gp*ζ; p = -p_{init}ζ;</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>z_{new} = sol.1; r = sol.2;</td>
</tr>
</tbody>
</table>

Continued on next page
\texttt{regDenom; \% construct regularize denominator}
\texttt{ζ = (gp + q^T z_{new})/denom;}
\texttt{if isnan(ζ), ζ = ζ_{max}; end;}
\texttt{ζ = min(ζ_{max}, max(ζ, ζ_{min})); z_{new} = z_{new} + ζ r;}
\texttt{cosine = −gp * ζ + c^T z_{new}; p = −p_{init} ζ + S_{hist} z_{new};}
\texttt{end;}
\texttt{else \% no subspace step possible}
\texttt{\% here doing instead an exact line search saves function values}
\texttt{ζ = gp / γ;}
\texttt{if isnan(ζ), ζ = ζ_{max}; end;}
\texttt{ζ = min(ζ_{max}, max(ζ, ζ_{min})); cosine = −gp * ζ; p = −p_{init} ζ;}
\texttt{end;}
\texttt{x_{new} = max(x, min(x + p, x)); p = x_{new} − x;}

4 Starting point and master algorithm

4.1 The starting point (\textit{projStartPoint})

In order that the gradient contains significant information about all components, the starting point should be chosen not too special. This is especially important in the bound constrained case, where the signs of gradient components determine which variables may be freed. For example, consider minimizing the quadratic function
\[ f(x) := (x_1 - 1)^2 + \sum_{i=2}^{n} (x_i - x_{i-1})^2 \]
started from \( x_0 = 0 \). If a diagonal preconditioner is used, it is easy to see by induction that, for any method that chooses its search directions as linear combinations of the previously computed preconditioned gradients, the \( i \)th iteration point has zero in all coordinates \( k > i \) and its gradient has zero in all coordinates \( k > i + 1 \). Since the solution is the all-one vector, this implies that at least \( n \) iterations are needed to reduce the maximal error in components of \( x \) to below one. Situations like this are likely to occur when both the Hessian and the starting point are sparse.

The following algorithm moves a user-given starting point \( x \) slightly into the relative interior of the feasible domain. \( Δ_ x \) is a number in \([0, \frac{1}{2} [\); choosing it instead as 0 just projects the starting point into the feasible box.

4.1 Algorithm. (\textit{projStartPoint})

\begin{verbatim}
Purpose: Improve the starting point
function [point] = projStartPoint(point, tune);
Continued on next page
\end{verbatim}
\[ \text{if } \Delta x = 0, \quad x = \max(\overline{x}, \min(x, \underline{x})); \]
\[ \text{else } \]
\[ \text{ind} = \{i \mid x_i \leq x_i, \ i = 1, \cdots, n\}; \quad x_{\text{ind}} = x_{\text{ind}} + \Delta x \min(\Delta u |x_{\text{ind}}|, |x_{\text{ind}} - x_{\text{ind}}|); \]
\[ \text{ind} = \{i \mid x_i \geq x_i, \ i = 1, \cdots, n\}; \quad x_{\text{ind}} = x_{\text{ind}} - \Delta x \min(\Delta u |x_{\text{ind}}|, |x_{\text{ind}} - x_{\text{ind}}|); \]
\[ \text{end}; \]
\[ \text{ind} = \{i \mid x_i = \pm \infty\}; \quad x_{\text{ind}} = \max(x_{\text{ind}}, \min(0, x_{\text{ind}})); \]

### 4.2 Successful iteration (\textit{getSuccess})

The goal of \textit{getSuccess} is to test whether the sufficient descent condition holds or not. The Goldstein quotient is computed provided that all of the following hold:

- The direction is descent, but not zero; in this case the direction was generated by \textit{ConjGradDir}.
- Either the direction is not the ordinary subspace step or the number of stuck iterations reaches.

After computing the Goldstein quotient, iteration will be successful if either line search is efficient, meaning the sufficient descent condition holds, or there exists an improvement on the function value by at least \( \Delta f \).

### 4.2 Algorithm. (\textit{getSuccess})

**Purpose:** Determine whether iteration is successful or not

```matlab
function \([\text{point, step, par, info}] = \text{getSuccess}(\text{fun, point, step, par, tune, info});\]
quad = 0; estuck = 0; Ip = \{i \mid p_i \neq 0\};
defQuad = (\text{cosine} < 0 \& \ & Ip \neq \emptyset \& (CG > 0 \text{ or } nstuck \geq nsmin));
if defQuad
    \% data are consistent with a definite quadratic function
    fnew = \text{fun}(xnew); nf = nf + 1;
    if isnan(fnew), fnew = +\infty; \text{end}; \% adjust f
    gp = g^Tp; \mu = (fnew - f)/gp;
    if (fnew == f \text{ or } \mu \mid \mu - 1 \geq \beta CG), quad = CG; \text{end}; \% check s.d. condition
    estuck = (nstuck \geq nsmin \& fnew \leq f + \Delta f);
end;
success = (quad or estuck);
```

### 4.3 Initializing information (\textit{initInfo})

\textit{initInfo} initializes the best point and its function value.
4.3 Algorithm. \textit{(initInfo)}

<table>
<thead>
<tr>
<th><strong>Purpose</strong>: Initialize best point and $\Delta_f$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>function</strong> [point] = initInfo(point, tune);</td>
</tr>
<tr>
<td>if ($f \neq \pm \infty &amp; f \neq 0$), $\Delta_f = \text{facf} \ast</td>
</tr>
<tr>
<td>$Df_{mf} = -\infty$; $Df_{mf} = \Delta_f$;</td>
</tr>
<tr>
<td>$f_{best} = f$;</td>
</tr>
</tbody>
</table>

4.4 Updating information \textit{(updateInfo)}

The goal of \textit{updateInfo} is first to update the best point and its function value. Second, it determines whether there exists an improvement on the function value or not; in such cases $\Delta_f$ and the list of its \textit{mf} previous values are updated. It is tried that the amount of $\Delta_f$ is updated by using its \textit{mf} previous values, preventing very tiny value; especially when the function value is very small.

4.4 Algorithm. \textit{(updateInfo)}

<table>
<thead>
<tr>
<th><strong>Purpose</strong>: Update best point and $\Delta_f$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>function</strong> [point, par] = updateInfo(point, par, tune, info);</td>
</tr>
<tr>
<td>% update $f_{best}$</td>
</tr>
<tr>
<td>if ($f_{new} &lt; f_{best}$), nstuck = 0; $f_{best} = f_{new}$; $x_{best} = x_{new}$; else, nstuck = nstuck + 1; <strong>end</strong>;</td>
</tr>
<tr>
<td>dec = ($f_{new} &lt; f$);</td>
</tr>
<tr>
<td>if dec % improvement</td>
</tr>
<tr>
<td>monotone = 1; $\Delta_f = f - f_{new}$; nm = mod(ng,\textit{mf});</td>
</tr>
<tr>
<td>if (nm == 0), $Df_{mf} = \Delta_f$; else, $Df_{nm} = \Delta_f$; <strong>end</strong></td>
</tr>
<tr>
<td>elseif ($f_{new} == f$), monotone = 0; % stalled</td>
</tr>
<tr>
<td>else % no descent</td>
</tr>
<tr>
<td>monotone = 0; $\Delta_f = \max(2\Delta_f, \Delta_m(</td>
</tr>
<tr>
<td>if (nm == 0), $Df_{mf} = \Delta_f$; else, $Df_{nm} = \Delta_f$; <strong>end</strong>;</td>
</tr>
<tr>
<td><strong>end</strong>;</td>
</tr>
<tr>
<td>$f = f_{new}$; % update $f$</td>
</tr>
</tbody>
</table>

4.5 The type of a subspace step \textit{(typeSubspace)}

We need to determine what to be the subspace. \textit{typeSubspace} uses three variables $m_0$ (length of subspace), \textit{hist} (list of subspace basis) and \textit{CG} (type of subspace) to determine the subspace. The
**conjugacy relation** is defined by

\[ h = Y^T s = 0. \] (37)

It works as follow:

- If \( n_{local} < n_{wait} \), the **ordinary subspace step** is used since the full subspace direction may be contaminated by nonactive components and so lead to premature freeing if used directly.

- If \( n_{local} = n_{wait} \), the **quasi-Newton step** generated by \( quasiNewtonDir \) is used if (25) holds, the subspace basis is permuted so that the oldest columns are shifted with newest ones.

- If \( n_{local} < n_{wait} + \hat{m} \), the conjugacy relation (37) is preserved by restricting the subspace.

- Otherwise, the full subspace step is preserved the conjugacy.

### 4.5 Algorithm. (typeSubspace)

**Purpose:** Determine the type of subspace

```plaintext
function [point, par] = typeSubspace(point, tune, par);

\( \hat{m} = \min(m, nh) \);

if (nlocal < nwait) % ordinary subspace step
  m0 = \min(ng - 1, \hat{m}); hist = [1 : m0]; CG = 0;
elseif (nlocal == nwait) % restart: steepest descent direction
  m0 = 0; hist = Ø; perm = [ch + 1 : \hat{m}, 1 : ch]; ch = 0;
  S = S_perm; Y = Y_perm; H = H_perm_perm; CG = 1;
elseif (nlocal < nwait + \hat{m}) % preserve conjugacy by restricting the subspace
  m0 = nlocal - nwait; hist = [1 : m0]; CG = 2;
else % full subspace step preserves conjugacy
  m0 = \hat{m}; hist = [1 : \hat{m}]; CG = 3;
end;
```

In **typeSubspace** whenever \( n_{local} = n_{wait} \), there is no subspace since \( m_0 = 0 \). In this case, a premature replacement of \( s(y) \) with the first column of subspace matrix \( S(Y) \) is made before \( ch \) exceeds \( m \).

### 4.6 The master algorithm

We now recall the main ingredients for **LMOPT**, the new **limited memory bound constrained optimization** method. It first calls the algorithm **projStartPoint** described in Subsection 4.1 to improve the starting point. Then the function value and gradient vector for such a point are computed and adjusted; the same is done later in every such calculation. In the main loop, **LMOPT** first computes the reduced gradient by **redGrad** in per iteration and then the working
set is determined and updated by \textit{findFreePos}. As long as the reduced gradient is not below a minimum threshold, it generates the starting direction \(p_{\text{init}}\) by \textit{searchDir} and then constructs the subspace conjugate gradient direction \(p\) by \textit{ConjGradDir} in the hope of achieving an successful iteration. Such an iteration is determined by \textit{getSuccess} and then the best point is updated. Otherwise it performs a gradient-free line search \textit{BLS} along a regularized direction (\textit{enforceAngle}) since the function is not near the quadratic case. Then if the null steps are repeated at least \(\text{nnullmax}\) in a sequence, the point leading to such steps is replaced by \textit{nullStep} with a point around the previous best point if \textit{BLS} is not efficient; otherwise with the current point generated by \textit{BLS}. This is repeated until no null step is found. Afterwards, the gradient is computed and adjusted by \textit{adjustGrad}. In addition, the new free index set is found by \textit{findFreeNeg}. At the end of every iteration, the subspace is updated provided that (i) there is no null step, (ii) either the Powell condition holds or the number of local steps exceeds its threshold.

For the convergence analysis of Algorithm 4.7 we refer to Section 11 of [51]. It can be seen that the conditions (7)–(9) on the search direction and the bent line search are essential for the convergence.

4.6 Theorem. Let \(f\) be continuously differentiable, with Lipschitz continuous gradient \(g\). Let \(x^\ell\) denote the value of \(x\) in Algorithm 1.1 after its \(\ell\)th update. Then one of the following three cases holds:

(i) The iteration stops after finitely many steps at a stationary point.

(ii) We have \[
\lim_{\ell \to \infty} f(x^\ell) = \hat{f} \in \mathbb{R}, \quad \inf_{\ell \geq 0} \|g_{\text{red}}(x^\ell)\|_* = 0.
\]
Some limit point \(\hat{x}\) of the \(x^\ell\) satisfies \(f(\hat{x}) = \hat{f} \leq f(x^0)\) and \(g_{\text{red}}(\hat{x}) = 0\).

(iii) \(\sup_{\ell \geq 0} \|x^\ell\| = \infty\).

4.7 Algorithm. (LMOPT, limited memory bound constrained optimization)
5 Numerical results

In this section we compare our new solver with other state-of-the-art solvers on a large public benchmark. More detailed tables are available in the file results*.pdf of the online package LMBOPT of our Matlab implementation, publicly available at the address given in Section 5.2.

5.1 Test problems used

LMBOPT is compared with many other codes from the literature (see Subsection 5.3) on all 1088 unconstrained and bound constrained problems from the CUTEst [32] collection of test problems for optimization with up to 100001 variables, in case of variable dimension problems for all allowed dimensions in this range.

nf, ng and msec denote the number of function evaluations, the number of gradient evaluations, and the time in milliseconds, respectively; nf2g = nf + 2ng. There will be two runs. In the first and second runs, to avoid guessing the solution of toy problems with a simple solution (such as all zero or all one), we shifted the arguments, for all \( i = 1, \ldots, n \), by

\[
x_i = (-1)^{i-1} \frac{m}{m + i},
\]

(38)
Figure 1: The number of problems with variables in a given range solved by at least one solver: 989 problems with dimensions 1 up 100001
where \( m = 2 \). In the third run, the standard starting point is used for unsolved test problems in the first run. We limited the budget available for each solver by allowing at most

\[
\begin{cases}
20n + 10000 & \text{in the first and second runs}, \\
50n + 200000 & \text{in the third run}
\end{cases}
\]

function evaluations plus two times gradient evaluations for a problem with \( n \) variables and at most

\[
\begin{cases}
180 & \text{in the first run}, \\
1800 & \text{in the second run}, \\
3600 & \text{in the third run}
\end{cases}
\]

seconds of run time. A problem is considered solved if

\[ \|g_k\| \leq 10^{-6}. \]

### 5.2 Default parameters for LMBOPT

LMBOPT was implemented in Matlab; the source code is obtainable from

http://www.mat.univie.ac.at/~neum/software/LMBOPT.

For our tests we used in **tune** the following parameters:

\[
\begin{array}{l}
m = 12; \text{mf} = 2; \Delta_x = 10^{-12}; \Delta_u = 1000; \Delta_g = 100, \Delta_{\text{angle}} = 10^{-12}; \Delta_w = \varepsilon_M; \\
\Delta_r = 20; \Delta_{pg} = \varepsilon_M; \Delta_{\text{reg}} = 10^{-12}; \Delta_\alpha = 5\varepsilon_M; \Delta_b = 10\varepsilon_M; \Delta_H = \varepsilon_M; \Delta_m = 10^{-13}; \\
\Delta_{po} = \varepsilon_M, \text{mdf} = 20; \text{typeH} = 0; \theta = 0.85; \beta = 0.02; \text{del} = 10^{-10}; \text{exact} = 0; \\
\text{bis} = 1; \text{nwait} = 1; \beta_{CG} = 0.001; \text{lmax} = 3; \text{nlf} = 2; \text{rfac} = 2.5; \text{facf} = 10^{-8}; \\
\text{nsmin} = 1; \text{nstuckmax} = +\infty; q = 25; \text{nnulmax} = 5; \zeta_{\min} = -10^{10}; \zeta_{\max} = -\zeta_{\min};
\end{array}
\]

They are based on limited tuning by hand. How to find optimal tuning parameters [43] would be interesting and very important since the quality of LMBOPT depends on it.

### 5.3 Codes compared

We compare LMBOPT with the following solvers for unconstrained and bound constrained optimization. For some of the solvers we chose options different from the default to make them more competitive.

**Bound constrained solvers:**

- **ASACG** (asa), obtained from

  http://users.clas.ufl.edu/hager/papers/CG/Archive/ASA(CG)-3.0.tar.gz, 

  is an active set algorithm for solving a bound constrained optimization problem by HAGER & ZHANG [40]. The default parameters have been used. Only **memory** = 12 and other parameters have been chosen as default.
• **LBFGSB** (lbf), obtained from
  
  \[ \text{http://users.iems.northwestern.edu/~nocedal/Software/Lbfgsb.3.0.tar.gz} \]
  
  is a limited-memory quasi-Newton code for bound-constrained optimization by Zhu et al. [11, 48, 60]. Only \( m = 12 \) and other parameters have been chosen as default.

• **ASABCP** (asb), obtained from
  
  \[ \text{https://sites.google.com/a/dis.uniroma1.it/asa-bcp/download} \]
  
  is a two-stage active-set algorithm for bound-constrained optimization by Cristofari et al. [17]. The default parameters have been used.

• **SPG** (spg), obtained from
  
  \[ \text{https://www.ime.usp.br/~egbirgin/tango/codes.php} \]
  
  is a spectral projected gradient algorithm for solving a bound constrained optimization problem by Birgin et al. [8, 9]. The default parameters have been used.

**Unconstrained solvers:**

• **CGdescent** (cdg), obtained from
  
  \[ \text{http://users.clas.ufl.edu/hager/papers/CG/Archive/CG_DESCENT-C-6.8.tar.gz} \]
  
  is a conjugate gradient algorithm for solving an unconstrained minimization problem by Hager & Zhang [38, 39, 41, 42]. Only memory = 12 and other parameters have been chosen as default.

• **LMBFG-MT** (ll1), obtained from
  
  \[ \text{http://gratton.perso.enseeiht.fr/LBFGS/index.html} \]
  
  is a limited memory line-search algorithm \( L-BFGS \) based on the MORE-THUENTE line search by Burdakov et al. [10]. Only \( m = 12 \) and other parameters have been chosen as default.

• **LMBFG-MTBT** (ll2), obtained from
  
  \[ \text{http://gratton.perso.enseeiht.fr/LBFGS/index.html} \]
  
  is a limited memory line-search algorithm \( L-BFGS \) based on the MORE-THUENTE line search and the starting step is obtained using backtrack by Burdakov et al. [10]. Only \( m = 12 \) and other parameters have been chosen as default.

• **LMBFGS-TR** (ll3), obtained from
  
  \[ \text{http://gratton.perso.enseeiht.fr/LBFGS/index.html} \]
  
  is a limited memory line-search algorithm \( L-BFGS \) that takes a trial step along the quasi-Newton direction inside the trust region by Burdakov et al. [10]. Only \( m = 12 \) and the other parameters have been chosen as default.

• **LMBFGS-BWX-MS** (lt1), obtained from
  
  \[ \text{http://gratton.perso.enseeiht.fr/LBFGS/index.html} \]
  
  is a limited memory trust-region algorithm BWX-MS by Burdakov et al. [10]. It applies the Moré & Sorensen approach for solving the TR subproblem defined in the Euclidean norm. Only \( m = 12 \) and the other parameters have been chosen as default.
• **LMBFG-DDOGL** ($lt2$) is a limited memory trust-region algorithm *D-DOGL* by *Burdakov* et al. [10]. Only $m = 12$ and the other parameters have been chosen as default.

• **LMBFG-EIG-curve-inf** ($lt4$) is a limited memory trust-region algorithm *EIG($\infty$, 2)* by *Burdakov* et al. [10]. Only $m = 12$ and the other parameters have been chosen as default.

• **LMBFG-EIG-inf-2** ($lt5$), obtained from http://gratton.perso.enseeiht.fr/LBFGS/index.html, is a limited memory trust-region algorithm *EIG*($\infty$, 2) based on the eigenvalue-based norm, with the exact solution to the TR subproblem in closed form by *Burdakov* et al. [10]. Only $m = 12$ and other parameters have been chosen as default.

• **LMBFG-EIG-MS** ($lt6$) is a limited memory trust-region algorithm *EIG-MS* by *Burdakov* et al. [10]. Only $m = 12$ and other parameters have been chosen as default.

• **LMBFG-EIG-MS-2-2** ($lt7$), obtained from http://gratton.perso.enseeiht.fr/LBFGS/index.html, is a limited memory trust-region algorithm *EIG−MS*($2$, 2) based on the eigenvalue-based norm, with the *Moré & Sorensen* approach for solving a low-dimensional TR subproblem by *Burdakov* et al. [10]. Only $m = 12$ and other parameters have been chosen as default.

Unconstrained solvers were turned into bound-constrained solvers by pretending that the reduced gradient at the point $\pi[x]$ is the requested gradient at $x$. Therefore no theoretical analysis is available, the results show that this is a simple and surprisingly effective strategy.

### 5.4 The results for stringent resources

#### 5.4.1 Unconstrained and bound constrained optimization problems

We tasted all 15 solvers for problems in dimension 1 up to 100001. The problems unsolved by all solvers are given in Table 11.

Performance plots [23] for four cost measures $nf$ (number of function evaluations needed to reach the target), $ng$ (number of gradient evaluations needed to reach the target), $nf2g$ ($nf+2ng$) and $msec$ (time used in milliseconds) are shown in Figure 2.

For a more refined statistics, we use our test environment (Kimiae & Neumaier [43]) for comparing optimization routines on the CUTEst test problem collection by Gould et al. [32]. For a given collection $S$ of solvers, the strength of a solver $so \in S$ – relative to an ideal solver that matches on each problem the best solver – is measured, for any given cost measure $c_s$ by the number, $q_{so}$ defined by

$$q_{so} := \begin{cases} 
\frac{(\min_{s \in S} c_s)}{c_{so}}, & \text{if } so \text{ solved the problem}, \\
0, & \text{otherwise},
\end{cases}$$

38
Table 11: The problems unsolved by all solvers

<table>
<thead>
<tr>
<th>Problem</th>
<th>Solver</th>
<th>Solver</th>
<th>Solver</th>
<th>Solver</th>
</tr>
</thead>
<tbody>
<tr>
<td>BROWNBS</td>
<td>PALMER7A</td>
<td>PALMER5E</td>
<td>PALMER5B</td>
<td></td>
</tr>
<tr>
<td>OSCIGRAD:10</td>
<td>OSCIPATH:10</td>
<td>STRATEC</td>
<td>SBRYBND:10</td>
<td></td>
</tr>
<tr>
<td>SCOSINE:10</td>
<td>SCURLY10:10</td>
<td>SCOND1LS</td>
<td>OSCIGRAD:15</td>
<td></td>
</tr>
<tr>
<td>OSCIGRAD:25</td>
<td>ANTWERP</td>
<td>NONMSQRT:49</td>
<td>HS110:50</td>
<td></td>
</tr>
<tr>
<td>SBRYBND:50</td>
<td>RAYBENDS</td>
<td>RAYBENDL:66</td>
<td>RAYBENDS:66</td>
<td></td>
</tr>
<tr>
<td>HYDC20LS</td>
<td>FLETCHBV:100</td>
<td>HS110:100</td>
<td>NONMSQRT:100</td>
<td></td>
</tr>
<tr>
<td>OSCIGRAD:100</td>
<td>SBRYBND:100</td>
<td>SCOSINE:100</td>
<td>SCURLY10:100</td>
<td></td>
</tr>
<tr>
<td>SCOND1LS:102</td>
<td>RAYBENDL:130</td>
<td>RAYBENDS:130</td>
<td>QR3DLS</td>
<td></td>
</tr>
<tr>
<td>GRIDGENA:170</td>
<td>DRCAV1LQ</td>
<td>HS110:200</td>
<td>SPMRTLS:499</td>
<td></td>
</tr>
<tr>
<td>PENALTY2:500</td>
<td>SBRYBND:500</td>
<td>SCOND1LS:502</td>
<td>MSQRTALS:529</td>
<td></td>
</tr>
<tr>
<td>MSQRTBLS:529</td>
<td>NONMSQRT:529</td>
<td>GRIDGENA</td>
<td>QR3DLS:610</td>
<td></td>
</tr>
<tr>
<td>LINVERSE:999</td>
<td>CURLY20</td>
<td>CHENHARK</td>
<td>FLETCHBV:1000</td>
<td></td>
</tr>
<tr>
<td>PENALTY2:1000</td>
<td>SBRYBND</td>
<td>SCOSINE</td>
<td>SCURLY10</td>
<td></td>
</tr>
<tr>
<td>SSCOSINE</td>
<td>SPMRTLS:1000</td>
<td>SCOND1LS:1002</td>
<td>MSQRTALS:1024</td>
<td></td>
</tr>
<tr>
<td>MSQRTBLS:1024</td>
<td>NONMSQRT:1024</td>
<td>RAYBENDL:1026</td>
<td>RAYBENDS:1026</td>
<td></td>
</tr>
<tr>
<td>DRCAV1LQ:1225</td>
<td>DRCAV2LQ:1225</td>
<td>DRCAV3LQ:1225</td>
<td>GRIDGENA:1226</td>
<td></td>
</tr>
<tr>
<td>RAYBENDL:2050</td>
<td>GRIDGENA:2114</td>
<td>EIGENALS:2550</td>
<td>GRIDGENA:3242</td>
<td></td>
</tr>
<tr>
<td>DRCAV3LQ:4489</td>
<td>GRIDGENA:4610</td>
<td>MSQRTALS:4900</td>
<td>SQRRTBLS:4900</td>
<td></td>
</tr>
<tr>
<td>SPMRTLS:4999</td>
<td>FLETCHBV:5000</td>
<td>FLETCHBV:5000</td>
<td>SBRYBND:5000</td>
<td></td>
</tr>
<tr>
<td>SSCOSINE:5000</td>
<td>SPARSINE:5000</td>
<td>SCOND1LS:5002</td>
<td>SCOND1LS:5002</td>
<td></td>
</tr>
<tr>
<td>BRATU1D:5003</td>
<td>GRIDGENA:6218</td>
<td>CURLY10:1000</td>
<td>CURLY20:1000</td>
<td></td>
</tr>
<tr>
<td>CURLY30:10000</td>
<td>FLETCHBV:10000</td>
<td>FLETCHBV:10000</td>
<td>NONCVXUN:10000</td>
<td></td>
</tr>
<tr>
<td>SCOSINE:10000</td>
<td>SCURLY10:10000</td>
<td>SPARSINE:10000</td>
<td>SPMRTLS:10000</td>
<td></td>
</tr>
<tr>
<td>SSCOSINE:100000</td>
<td>DRCAV3LQ:10816</td>
<td>ODNAMUR</td>
<td>GRIDGENA:12482</td>
<td></td>
</tr>
<tr>
<td>SSCOSINE:100000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

39
Figure 2: (a)-(e): Performance plots for $\text{ng}/(\text{best ng})$, $\text{nf}/(\text{best nf})$, $\text{nf2g}/(\text{best nf2g})$ and $\text{msec}/(\text{best msec})$, respectively. $\rho$ designates the percentage of problems solved within a factor $\tau$ of the best solver. Problem solved by no solver are ignored.
called the **efficiency** of the solver so with respect to this cost measure. In the tables, efficiencies are given in percent. Larger efficiencies in the table imply a better average behaviour; a zero efficiency indicates failure. All values are rounded (towards zero) to integers. Mean efficiencies are taken over the 991 problems tried by all solvers and solved by at least one of them, from a total of 1088 problems. In the following tables, #100 and !100 count the number of times we have nf2g efficiency 100% or unique nf2g efficiency 100%. $T_{\text{mean}}$ is defined by

$$T_{\text{mean}} := \frac{\sum \text{solved}}{\# \text{solved}}.$$ 

Failure reasons were reported in the anomaly columns:

- $n$ indicates that nf2g $\geq 20n + 10000$ was reached.
- $t$ indicates that sec $\geq 300$ was reached.
- $f$ indicates that the algorithm failed for other reasons.

In the times, the (for some problems significant) setup time for CUTEst is not included. Although running times are reported, the comparison of times is not very reliable for several reasons:

(i) The times were obtained under different conditions (solver source code Fortran, C and Matlab).
(ii) In unsuccessful runs, the actual running time depends a lot on when and why the solver was stopped.
(iii) Function and gradient evaluation includes times for computing various statistics and the interface to CUTEst; cf. Figure 3. In [43], getfg have been introduced to compute the function value and gradient of function handle fun at $x$, collect statistics and enforce stopping tests. In CUTEst, both function value and gradient are computed by cutest_obj without returning any information about statistics.

![Figure 3](image-url)

**Figure 3:** Comparison of $q_{\text{cutest}} := \frac{t_f(\text{cutest})}{t_f(\text{cutest})}$, $q_{\text{getfg}} := \frac{t_f(\text{getfg})}{t_f(\text{getfg})}$ and $q_{\text{over}} := \frac{t_{f2g}(\text{getfg})}{t_{f2g}(\text{cutest})}$ versus dimensions, respectively, where $t_f$ and $t_g$ are considered the time to compute $f$ and $g$ by cutest or getfg and $t_{f2g} := t_f + 2t_g$.

As can be seen from Table 13, LMBOPT is stood out as the most robust solver for unconstrained and bound constrained optimization problems; it is the best in terms of number of solved problems and gradient evaluations. Other best solvers in that the number of solved problems and nf2g are ASACG and LMBFG-EIG-MS, respectively. LBFGSB is the best in terms of number of function evaluations #100 and !100, but is not comparable in that the number of solved problems with other algorithms.
Table 13: The summary results for all problems

<table>
<thead>
<tr>
<th>solver</th>
<th>dim≤[1,100001] solved</th>
<th># of anomalies</th>
<th>mean efficiency in % for cost measure</th>
</tr>
</thead>
<tbody>
<tr>
<td>LMBOPT</td>
<td>lmb 948</td>
<td>nf2g 45</td>
<td>58 69 42 11</td>
</tr>
<tr>
<td>ASACG</td>
<td>asa 935</td>
<td>ng 45</td>
<td>58 59 51 63</td>
</tr>
<tr>
<td>LMBFG-EIG-MS</td>
<td>li6 924</td>
<td>ng 29</td>
<td>60 57 60 34</td>
</tr>
<tr>
<td>LMBFG-EIG-curve-inf</td>
<td>li4 918</td>
<td>ng 29</td>
<td>60 57 60 34</td>
</tr>
<tr>
<td>ASABCP</td>
<td>asb 900</td>
<td>ng 24</td>
<td>41 36 44 46</td>
</tr>
<tr>
<td>LMBFG-DDOGL</td>
<td>lt2 896</td>
<td>ng 29</td>
<td>60 56 59 34</td>
</tr>
<tr>
<td>CGdescent</td>
<td>cg 895</td>
<td>ng 22</td>
<td>54 56 47 55</td>
</tr>
<tr>
<td>LMBFG-EIG-MS-2-2</td>
<td>lt7 895</td>
<td>ng 22</td>
<td>50 45 57 34</td>
</tr>
<tr>
<td>LMBFG-BWX-MS</td>
<td>lt1 888</td>
<td>ng 17</td>
<td>51 45 58 32</td>
</tr>
<tr>
<td>SPG</td>
<td>spg 840</td>
<td>ng 14</td>
<td>34 34 31 9</td>
</tr>
<tr>
<td>LBFGSB</td>
<td>lb 803</td>
<td>ng 14</td>
<td>57 51 51 32</td>
</tr>
<tr>
<td>LMBFG-EIG-inf-2</td>
<td>lt5 753</td>
<td>ng 14</td>
<td>50 47 49 28</td>
</tr>
<tr>
<td>LMBFGS-TR</td>
<td>lt3 733</td>
<td>ng 14</td>
<td>48 44 48 36</td>
</tr>
<tr>
<td>LMBFG-MTBT</td>
<td>lt2 669</td>
<td>ng 14</td>
<td>45 41 46 26</td>
</tr>
<tr>
<td>LMBFG-MT</td>
<td>lt1 657</td>
<td>ng 14</td>
<td>45 39 48 32</td>
</tr>
</tbody>
</table>

984 of 1088 problems solved. $\text{sec} \leq 1800$

<table>
<thead>
<tr>
<th>solver</th>
<th>dim≤[1,100001] solved</th>
<th># of anomalies</th>
<th>mean efficiency in % for cost measure</th>
</tr>
</thead>
<tbody>
<tr>
<td>LMBOPT</td>
<td>lmb 953</td>
<td>nf2g 51</td>
<td>67 75 52 17</td>
</tr>
<tr>
<td>ASACG</td>
<td>asa 936</td>
<td>nf2g 25</td>
<td>67 65 63 82</td>
</tr>
<tr>
<td>LMBFG-EIG-MS</td>
<td>li6 932</td>
<td>nf2g 25</td>
<td>71 64 75 48</td>
</tr>
</tbody>
</table>
Table 14: The summary results for unconstrained and bound constrained problems

<table>
<thead>
<tr>
<th>solver</th>
<th>solved</th>
<th>#100</th>
<th>!100</th>
<th>T\text{mean}</th>
<th>#n</th>
<th>#t</th>
<th>#f</th>
<th>nf2g</th>
<th>ng</th>
<th>nf</th>
<th>msec</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASACG</td>
<td>asa</td>
<td>533</td>
<td>132</td>
<td>121</td>
<td>1331</td>
<td>53</td>
<td>16</td>
<td>13</td>
<td>67</td>
<td>66</td>
<td>61</td>
</tr>
<tr>
<td>LMOPT</td>
<td>lmb</td>
<td>531</td>
<td>162</td>
<td>160</td>
<td>3962</td>
<td>50</td>
<td>34</td>
<td>0</td>
<td>68</td>
<td>75</td>
<td>53</td>
</tr>
<tr>
<td>LMBFG-EIG-MS</td>
<td>lt6</td>
<td>522</td>
<td>271</td>
<td>258</td>
<td>3055</td>
<td>68</td>
<td>21</td>
<td>4</td>
<td>69</td>
<td>59</td>
<td>73</td>
</tr>
</tbody>
</table>

552 of 615 problems without bounds solved

425 of 473 problems with bounds solved

5.4.2 Classified by constraints

Summarise of separate results for unconstrained and bound constrained problems are given in Table 14. For both unconstrained and bound constrained problems, LMOPT is most robust algorithm in terms of number of solved problems and gradient evaluations. It has same efficiency in terms of nf2g with ASACG, however, LMBFG-EIG-MS is the best in terms of nf and nf2g.

5.4.3 Classified by dimension

Results for the three best solvers for all problems classified by dimension are given in Table 15. Table 15 shows that

- LMOPT is the best in terms of ng for n < 50001 and it is the best in terms of the number of solved problems for n ≤ 100 and n ∈ [1001,3000].

- LMBFG-EIG-MS is the best in terms of nf and nf2g in most dimension ranges.

- For very large scale problems, n ∈ [50001,100001], ASACG is the best in terms of the number of solved problems, nf, ng, nf2g, !100 and #100. Moreover, ASACG is the best in terms of msec in all dimension ranges.
Table 15: The summary results classified by dimension for all problems

<table>
<thead>
<tr>
<th>dim ∈ [1, 5] problems solved</th>
<th>stopping test: $|g|_\infty \leq 1e-06$, sec $\leq 1800$, $nf + 2 \ast ng \leq 20 \ast n + 10000$</th>
<th>mean efficiency in % for cost measure</th>
</tr>
</thead>
<tbody>
<tr>
<td># of anomalies</td>
<td># of anomalies</td>
<td># of anomalies</td>
</tr>
<tr>
<td>solver</td>
<td>solved</td>
<td>#100</td>
</tr>
<tr>
<td>LMBOPT</td>
<td>lmb</td>
<td>115</td>
</tr>
<tr>
<td>ASACG</td>
<td>asa</td>
<td>110</td>
</tr>
<tr>
<td>LMBFG-EIG-MS</td>
<td>lt6</td>
<td>106</td>
</tr>
<tr>
<td>n ∈ [6, 10], 112 of 121 problems solved</td>
<td># of anomalies</td>
<td>for cost measure</td>
</tr>
<tr>
<td>LMBOPT</td>
<td>lmb</td>
<td>112</td>
</tr>
<tr>
<td>ASACG</td>
<td>asa</td>
<td>107</td>
</tr>
<tr>
<td>LMBFG-EIG-MS</td>
<td>lt6</td>
<td>94</td>
</tr>
<tr>
<td>n ∈ [11, 30], 75 of 80 problems solved</td>
<td># of anomalies</td>
<td>for cost measure</td>
</tr>
<tr>
<td>LMBOPT</td>
<td>lmb</td>
<td>74</td>
</tr>
<tr>
<td>ASACG</td>
<td>asa</td>
<td>68</td>
</tr>
<tr>
<td>n ∈ [31, 100], 194 of 209 problems solved</td>
<td># of anomalies</td>
<td>for cost measure</td>
</tr>
<tr>
<td>LMBFG-EIG-MS</td>
<td>lt6</td>
<td>184</td>
</tr>
<tr>
<td>n ∈ [101, 300], 51 of 58 problems solved</td>
<td># of anomalies</td>
<td>for cost measure</td>
</tr>
<tr>
<td>LMBFG-EIG-MS</td>
<td>lt6</td>
<td>50</td>
</tr>
<tr>
<td>ASACG</td>
<td>asa</td>
<td>49</td>
</tr>
<tr>
<td>LMBOPT</td>
<td>lmb</td>
<td>48</td>
</tr>
<tr>
<td>n ∈ [301, 1000], 141 of 163 problems solved</td>
<td># of anomalies</td>
<td>for cost measure</td>
</tr>
<tr>
<td>LMBFG-EIG-MS</td>
<td>lt6</td>
<td>137</td>
</tr>
<tr>
<td>ASACG</td>
<td>asa</td>
<td>136</td>
</tr>
<tr>
<td>LMBOPT</td>
<td>lmb</td>
<td>135</td>
</tr>
<tr>
<td>n ∈ [1001, 3000], 81 of 94 problems solved</td>
<td># of anomalies</td>
<td>for cost measure</td>
</tr>
<tr>
<td>LMBFG-EIG-MS</td>
<td>lt6</td>
<td>79</td>
</tr>
<tr>
<td>ASACG</td>
<td>asa</td>
<td>168</td>
</tr>
<tr>
<td>LMBOPT</td>
<td>lmb</td>
<td>168</td>
</tr>
<tr>
<td>n ∈ [3001, 10000], 173 of 201 problems solved</td>
<td># of anomalies</td>
<td>for cost measure</td>
</tr>
<tr>
<td>ASACG</td>
<td>asa</td>
<td>168</td>
</tr>
<tr>
<td>LMBFG-EIG-MS</td>
<td>lt6</td>
<td>168</td>
</tr>
<tr>
<td>n ∈ [50001, 100001], 6 of 7 problems solved</td>
<td># of anomalies</td>
<td>for cost measure</td>
</tr>
<tr>
<td>ASACG</td>
<td>asa</td>
<td>168</td>
</tr>
<tr>
<td>LMBFG-EIG-MS</td>
<td>lt6</td>
<td>168</td>
</tr>
<tr>
<td>LMBOPT</td>
<td>lmb</td>
<td>168</td>
</tr>
</tbody>
</table>
### Table 16: The summary results for hard problems

<table>
<thead>
<tr>
<th>Stopping test</th>
<th>$|g|_\infty \leq 1e-06$, $\text{sec} \leq 3600$, $\text{nf} + 2 \times \text{ng} \leq 50 \times n + 200000$</th>
</tr>
</thead>
<tbody>
<tr>
<td>dim $\in [1,100001]$</td>
<td>mean efficiency in %</td>
</tr>
<tr>
<td>solver</td>
<td>solved</td>
</tr>
<tr>
<td>LMBFG-EIG-MS</td>
<td>lt6</td>
</tr>
<tr>
<td>LMOPT</td>
<td>lmb</td>
</tr>
<tr>
<td>ASACG</td>
<td>asa</td>
</tr>
</tbody>
</table>

62 of 93 problems solved

<table>
<thead>
<tr>
<th>31 of 56 problems without bounds solved</th>
<th>mean efficiency in %</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASACG</td>
<td>asa</td>
</tr>
<tr>
<td>LMBFG-EIG-MS</td>
<td>lt6</td>
</tr>
<tr>
<td>LMOPT</td>
<td>lmb</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>31 of 37 problems with bounds solved</th>
<th>mean efficiency in %</th>
</tr>
</thead>
<tbody>
<tr>
<td>LMBFG-EIG-MS</td>
<td>lt6</td>
</tr>
<tr>
<td>LMOPT</td>
<td>lmb</td>
</tr>
<tr>
<td>ASACG</td>
<td>asa</td>
</tr>
</tbody>
</table>

### Table 17: The hard problems unsolved by all solvers

<table>
<thead>
<tr>
<th>OSCIPATH:10</th>
<th>SCOND1LS</th>
<th>ANTWERP</th>
<th>HYDC20LS</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLETCHBV:100</td>
<td>NONMSQRT:100</td>
<td>SBRYBND:100</td>
<td>SCOSINE:100</td>
</tr>
<tr>
<td>SCURLY10:100</td>
<td>PENALTY2:500</td>
<td>SCOND1LS:502</td>
<td>NONMSQRT:529</td>
</tr>
<tr>
<td>FLETCHBV:1000</td>
<td>PENALTY2:1000</td>
<td>SCOSINE</td>
<td>SCURLY10</td>
</tr>
<tr>
<td>SCOND1LS:1002</td>
<td>NONMSQRT:1024</td>
<td>FLETCHBV3:5000</td>
<td></td>
</tr>
<tr>
<td>FLETCHBV:50000</td>
<td>SBRYBND:5000</td>
<td>SCOSINE:5000</td>
<td>SCOND1LS:5002</td>
</tr>
<tr>
<td>BRATU1D:5003</td>
<td>FLETCHBV:10000</td>
<td>FLETCHBV:10000</td>
<td>NONCVXUN:10000</td>
</tr>
</tbody>
</table>

### 5.5 Results for hard problems

All solvers have been run again on the hard problems defined as, the 100 test problems unsolved in the first run. In this case, the standard starting point has been used instead of (38) and both $\text{nfmax}$ and $\text{secmax}$ have been increased. 31 test problems were not solved by all solvers for dimensions 1 up 100001, given in Table 17.

From Table 16, we conclude

- **LMOPT** is the second best solver in terms of the number of solved problems.
- **LMBFG-EIG-MS** and **LMOPT** are the best solvers in terms of ng.
- **LMBFG-EIG-MS** is the best solver in terms of nf2g and nf.
Acknowledgement The first author acknowledges the financial support of the Doctoral Program “Vienna Graduate School on Computational Optimization” funded by Austrian Science Foundation under Project No W1260-N35.

References


