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

In this paper we describe a new active set method for solving the bound constrained optimization problem

$$\begin{array}{ll} \min & f(x) \\ \text{s.t.} & x \in \mathbb{R}^n, \quad \underline{x} \le x \le \overline{x}, \end{array} \tag{1}$$

where $\mathbf{x} = [\underline{x}, \overline{x}]$ is a bounded or unbounded box in $\mathbb{R}^n$ describing the bounds on the variables and the **objective function** $f : \mathbf{x} \to \mathbb{R}$ 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 drops 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 Mathematicaly 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.

• $\sim$ (or `not`) denotes logical negation.

• $\circ$ 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 kth 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

$$(g_{\text{red}}(x))_i := \begin{cases} 0 & \text{if } x_i = \underline{x}_i = \overline{x}_i, \\ \min(0, g_i) & \text{if } x_i = \underline{x}_i < \overline{x}_i, \\ \max(0, g_i) & \text{if } x_i = \overline{x}_i > \underline{x}_i, \\ g_i & \text{otherwise,} \end{cases} \tag{2}$$

where $g_i := g_i(x)$ is the ith component of gradient vector at x.

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

$$I_-(x) := \{i \mid \underline{x}_i < x_i < \overline{x}_i\}, \tag{3}$$

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

$$\begin{aligned} I_+(x) &:= I_-(x) \cup \{i \mid (g_{\text{red}})_i \neq 0\} \\ &= I_-(x) \cup \{i \mid \underline{x}_i = x_i < \overline{x}_i, g_i < 0 \text{ or } \underline{x}_i < x_i = \overline{x}_i, g_i > 0\} \end{aligned} \tag{4}$$

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**

$$x(\alpha) := \pi[x + \alpha p], \tag{5}$$

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

$$\pi[x]_i := \sup(\underline{x}_i, \inf(x_i, \overline{x}_i)) = \begin{cases} \underline{x}_i & \text{if } x_i \leq \underline{x}_i, \\ \overline{x}_i & \text{if } x_i \geq \overline{x}_i, \\ x_i & \text{otherwise.} \end{cases} \tag{6}$$

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

$$p_i = 0 \quad \text{for } i \notin I, \tag{7}$$

$$\frac{g_I^T p_I}{\|g_I\| \|p_I\|} \leq -\delta < 0, \tag{8}$$

$$g_i p_i \leq 0 \quad \text{for all } i \quad \text{if } I = I_+(x) \neq I_-(x), \tag{9}$$

$$\|g_I\|^2 \geq \rho \|g_{\text{red}}\|^2, \tag{10}$$

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 $x \in \mathbf{x} = [\underline{x}, \overline{x}]$
Input: $x^0 \in \mathbb{R}^n$ (starting point)
Parameters: $\beta \in]0, \frac{1}{4}[$, $q > 1$ (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

```
$x = x^0$;  $I = I_+(x)$;  freeing=0;
while $g_{\mathrm{red}}(x) \neq 0$,
    Update $x$ by performing the line search
        CLS along a bent search path (5)
        with $q$ satisfying (7), (8), and (9);
    update $I = I_-(x)$;
    freeing=((10) fails);
    if freeing, update $I = I_+(x)$; end;
end;
```

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 then 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:

Step	dependencies
LMBOPT	*Preprocessor*, *Determiner*, *Updater*, *Postprocessor*
Preprocessor	*Initializer*, *Improver*, *Problem object*
Determiner	*Reducer*, *Worker selector*, *Successor*, *Unsuccessor*
Updater	*Worker*, *Info*, *Subspace*
Successor	*Subspace selector*, *Director*, *Problem object*
Problem object	*Generator*, *Adjuster*

Director	*Local solvers*, *Conjugator*
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*

Table 1: Mathematical dependcy graph of *LMBOPT*

The top levels. *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 γ, 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.

Initializer	*initInfo*
Improver	*projStartPoint*
Determiner	*getSuccess*
Working selector	*findFreePos*
Worker	*findFreeNeg*
Info	*updateInfo*
Subspace	*updateSubspace*
Subspace selector	*typeSubspace*
Local solvers	*scaleDir*, *quasiNewtonDir*, *AvoidZigzagDir*
Generator	*fun*, *dfun*
Adjuster	*adjustGrad*
Reducer	*redGrad*
Robustifier I	*goodStep*
Gamma	*getGam*
Regularizer	*regDenom*
Conjugate gradient diection	*ConjGradDir*
Enforcer	*enforceAngle*
Nullifier	*nullStep*
bent line search	*BLS*
Robustifier II	*robustStep*

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.

Algorithm 2.1	**function** [`step`] = **goodStep**(`point`, `step`, `tune`);
`goodStep`	Try to find the starting good step size
Algorithm 2.2	**function** [`point`, `step`] = **robustStep**(`point`, `step`, `tune`);

robustStep	Try to find a point with smallest robust change
Algorithm 2.3	**function** [`point`, `step`, `info`] = **BLS**(*fun*, `point`, `step`, `tune`, `info`);
BLS	Find a step size α satisfying a sufficient descent condition
Algorithm 2.4	**function** [`point`, `step`, `par`, `info`] = $\cdots$ **nullStep**(*fun*, `point`, `step`, `par`, `tune`, `info`);
nullStep	Try to prevent producing the null steps
Algorithm 3.1	**function** [`point`] = **adjustGrad**(`point`, `tune`);
adjustGrad	Adjust the gradient vector
Algorithm 3.2	**function** [`point`] = **redGrad**(`point`);
redGrad	Compute the reduced gradient
Algorithm 3.3	**function** [`point`, `par`] = **findFreePos**(`point`, `par`, `tune`);
findFreeNeg	Update the working set
Algorithm 3.4	**function** [`point`, `par`, `info`] = **findFreeNeg**(`point`, `par`, `tune`, `info`);
findFreePos	Find the free index set
Algorithm 3.5	**function** [`point`] = **updateSubspace**(`point`, `step`, `par`, `tune`);
updateSubspace	Update the subspace information
Algorithm 3.7	**function** [`step`] = **enforceAngle**(`point`, `step`, `par`, `tune`);
enforceAngle	Enforce the angle condition
Algorithm 3.8	**function** [`point`, `step`] = **quasiNewtonDir**(`point`, `step`);
quasiNewtonDir	Compute quasi Newton direction
Algorithm 4.5	**function** [`par`] = **typeSubspace**(`tune`, `par`);
typeSubspace	Determine the type of subspace
Algorithm 3.9	**function** [`step`, `par`] = **scaleDir**(`point`, `step`, `par`);
scaleDir	Choose components of sensible sign and scale
Algorithm 3.10	**function** [`step`] = **AvoidZigzagDir**(`point`, `step`, `tune`, `info`);
AvoidZigzagDir	Modify the direction to avoid zigzagging
Algorithm 3.11	**function** [`point`, `step`, `par`] = **searchDir**(`point`, `step`, `par`, `tune`, `info`);
searchDir	Construct starting trial search direction
Algorithm 3.12	**function** [`point`, `step`, `par`, `info`] = $\cdots$ **getGam**(*fun*, `point`, `step`, `tune`, `par`, `info`);
getGam	Compute γ
Algorithm 3.13	**function** [`par`] = **regDenom**(`point`, `step`, `par`, `tune`);
regDenom	Construct regularize denominator
Algorithm 3.14	**function** [`point`, `step`, `par`, `info`]= $\cdots$ **ConjGradDir**(*fun*, `point`, `step`, `par`, `tune`, `info`);
ConjGradDir	Construct the conjugate gradient direction
Algorithm 4.1	**function** [`point`] = **projStartPoint**(`point`, `tune`);

projStartPoint	Improve the starting point
Algorithm 4.2	**function** [`point`, `step`, `par`, `info`] = $\cdots$ **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 $x \in \mathbf{x} = [\underline{x}, \overline{x}]$

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

fun and *dfun* (structure with information about function handle)
`point` (structure with information about points and function values)
x, f, g (old point, its function value and gradient vector) $x_{\rm new}$, $f_{\rm new}$, $g_{\rm new}$ (newest point, its function value and gradient vector) $x_{\rm best}$, $f_{\rm best}$ (best point and its function value) $x_{\rm init}$, $f_{\rm init}$ (starting point and its function value) $\underline{x}$, $\overline{x}$ (lower and upper bound) y (the difference of current gradient with its old one; $g_{\rm new} - g$) 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 $y_1, \cdots, y_m$) H (Hessian matrix), q (extrapolation factor) `df` (acceptable increase in f), Δ_f (factor for adjusting `df`)
`step` (structure with information about the step management)
$p_{\rm init}$ (starting search direction in each iteration), p (Krylov search direction), `gp` (g^Tp) $\alpha_{\rm good}$ (the starting step-size generated by *goodStep*), s (search direction; $x_{\rm new} - x$) $\mathcal{A}$ (list of some step-sizes generated by *BLS*)
`tune` (structure with fixed parameters for tuning the performance)
ε (accuracy for reduced gradient), m (subspace dimension), `mf` (memory for `Df`) Δ_x (tiny factor for interior move), Δ_u (factor for adjusting $\overline{x}$) Δ_g (factor for adjusting gradient), $\Delta_{\rm angle}$ (regularization angle) Δ_w (for guaranteeing $w > 0$), Δ_r (factor for finding almost flat step) Δ_{pg} (tiny factor for regularizing g^Tp in *ConjGradDir*) Δ_{reg} (tiny factor for regularizing g^Tp in *BLS*)

Δ_α (tiny factor for starting step), Δ_b (tiny factor for breakpoint)
Δ_H (tiny regularization factor for subspace Hessian)
Δ_m (tiny factor for regularizing Δ_f if not monotone)
Δ_{po} (gradient tolerance for skipping update)
`typeH` (choose update formula for Hessian (0 or 1))
`gfac` (parameter for scaling direction), θ (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)
$\beta_{\texttt{CG}}$ (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`$*n_I$ 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 ζ 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), `nnull` (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 c ancellation 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 α_{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 $2 * \texttt{nstuck}$ to avoid remaining stuck.

2.1 Algorithm. (**goodStep**)

```
Purpose: Try to find the starting good step size
function [step]= goodStep(point, step, tune);
ind = $\{i \mid p_i < 0 \ \& \ x_i > \underline{x}_i\}$; % find the index of first breakpoint
if (ind $\neq \emptyset$), $\underline{\alpha}_{\text{break}} = \min\{(\underline{x}_i - x_i)/p_i \mid i \in \texttt{ind}\}$; else, $\underline{\alpha}_{\text{break}} = +\infty$; end;
ind = $\{i \mid p_i > 0 \ \& \ x_i < \overline{x}_i\}$; % find the index of second breakpoint
if (ind $\neq \emptyset$), $\overline{\alpha}_{\text{break}} = \min\{(\overline{x}_i - x_i)/p_i \mid i \in \texttt{ind}\}$; else, $\overline{\alpha}_{\text{break}} = +\infty$; end;
$\alpha_{\text{break}} = \min(\underline{\alpha}_{\text{break}}, \overline{\alpha}_{\text{break}})$; $\alpha_{\text{break}} = \alpha_{\text{break}}(1 + \Delta_b)$;
% define minimal step size
ind = $\{i \mid p_i \neq 0\}$;
if ($x == 0$ & ind $\neq \emptyset$), $\alpha_{\min} = \Delta_\alpha |f/\texttt{gp}|$;
else,
    if (ind $\neq \emptyset$), $\alpha_{\min} = \Delta_\alpha \max\Big(|f/\texttt{gp}|, \min\Big\{|x_i/p_i| \mid i \in \texttt{ind}\Big\}\Big)$;
    else, $\alpha_{\min} = 1$; $\alpha_{\text{good}} = 1$; return; % zero direction
    end;
end;
```

Continued on next page

```
$\alpha_{\text{target}} = \max(\alpha_{\min}, \texttt{df}/|\texttt{gp}|)$;
**if** exact, $\alpha_{\text{target}} = \min(\alpha_{\text{target}}, \alpha_{\text{break}})$; **end**;
**if** $(q\alpha_{\text{target}} \le \alpha_{\text{break}})$, $\alpha_{\text{good}} = \alpha_{\text{target}}$; **else**, $\alpha_{\text{good}} = \max(\alpha_{\min}, \alpha_{\text{break}})$; **end**;
**if** $(\texttt{nstuck} \ge \texttt{nsmin})$, $\alpha_{\text{good}} := 2(\texttt{nstuck})\alpha_{\text{good}}$; **end**;
```

2.2 A robustified step size (*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 (**robustStep**)

Purpose: Try to find a point with smallest robust change

function [point, step] = **robustStep**(point, step, tune);

```
$\texttt{dF}_b = \min(\texttt{dF})$; $\texttt{irob} = \{i \mid \texttt{dF}_i = \texttt{dF}_b\}$
if $(\texttt{dF}_b < 0)$, $\alpha_{\text{new}} = \mathcal{A}_{\texttt{irob}}$; $f_{\text{new}} = f + \texttt{dF}_{\texttt{irob}}$; return; end;
% treat failed line search (no improvement); find point with smallest robust change
$\texttt{ind} = \{i \mid \texttt{dF}_i > 0 \;\&\; \texttt{dF}_i < +\infty\}$; $\texttt{dF}_b = \min_{i\in\texttt{ind}}(\texttt{dF})$;
$i_{\text{new}} = \{i \in \texttt{ind} \mid \texttt{dF}_i = \texttt{dF}_b\}$; $\texttt{irob} = \texttt{ind}_{i_{\text{new}}}$;
% quality robust with robust change
if $(\texttt{dF}_b \le \texttt{df})$, $\alpha_{\text{new}} = \mathcal{A}_{\texttt{irob}}$; $f_{\text{new}} = f + \texttt{dF}_{\texttt{irob}}$; return; end;
$\texttt{idF} = \{i \mid \texttt{dF}_i \le 0\}$;
if $(i_{\text{new}} == \emptyset$ or $\texttt{idF} \ne \emptyset)$, % function almost flat; take step with largest dF
    $\texttt{ind} = \{i \mid \texttt{dF}_i < +\infty\}$; $\texttt{dF}_b = \min_{i\in\texttt{ind}}(\texttt{dF})$; $i_{\text{new}} = \{i \in \texttt{ind} \mid \texttt{dF}_i = \texttt{dF}_b\}$; $\texttt{irob} = \texttt{ind}_{i_{\text{new}}}$;
    % function is flat; take largest step
    if $(\texttt{dF}_b \le 0)$, $\alpha_{\text{new}} = \max_{i\in\texttt{ind}}(\mathcal{A})$; $i_{\text{new}} = \{i \in \texttt{ind} \mid \mathcal{A}_i = \alpha_{\text{new}}\}$; $\texttt{irob} = \texttt{ind}_{i_{\text{new}}}$; end;
else % take largest almost flat step
    if $(\texttt{dF}_b > \Delta_r\texttt{df})$
        $\texttt{ind} = \{i \mid \texttt{dF}_i \le \texttt{df}\}$; $\alpha_{\text{new}} = \max_{i\in\texttt{ind}}(\mathcal{A})$; $i_{\text{new}} = \{i \in \texttt{ind} \mid \mathcal{A}_i = \alpha_{\text{new}}\}$; $\texttt{irob} = \texttt{ind}_{i_{\text{new}}}$;
    end;
end;
$\alpha_{\text{new}} = \mathcal{A}_{\texttt{irob}}$; $f_{\text{new}} = f + \texttt{dF}_{\texttt{irob}}$;
```

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 dF is smaller than or equal df. Otherwise, if the function is almost flat or flat; then a step with largest dF is chosen. Otherwise, a point with nonrobust change might be chosen provided that the minimum of $\texttt{dF} \le \Delta_r\texttt{df}$.

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 `df` in f is updated.

• A regularized directional derivative is used.

• *goodStep* is recalled to find the starting good step-size α_{good}, the target step size α_{target} and the minimum step size $\alpha_{\min}$.

• Change to find a step size $\alpha > 0$ satisfying the **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 **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 $[\underline{\alpha}, \overline{\alpha}]$ is found. Then, either the geometric mean or the cubic bisection or the geometric mean alternated with the `mbis` cubic bisection is used.

• A limit on the number of iterations is used.

• At the end, *robustStep* is used to robust the step size if the line search is not efficient.

Moreover, two arrays $\mathcal{A}$ and `dF` use to restore step sizes and gains, respectively. The variable `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 α with $\mu(\alpha)|\mu(\alpha) - 1| \geq \beta$

function [point, step, info] = **BLS**(*fun*, point, step, tune, info);

```
if (ng == 1), df = $\Delta_f$;
else
    if (mod(ng, mdf) == 0), df = $\Delta_f(|f| + 1)$; else, df = max(Df); end;
end;
gp = $\min(g_I^T p_I, -\Delta_{pg}(|g_I|^T |p_I|))$; % regularized directional derivative
```

Continued on next page

```
$goodStep$; % get $\alpha_{\text{good}}$
first $= 1$; descent $= 0$; rob $= 0$; eff $=$ NaN; $i = 0$; $\underline{\alpha} = 0$; $\overline{\alpha} = \infty$;
$\alpha = \alpha_{\text{good}}$; $\mathtt{dF}_1 = 0$; $\mathcal{A}_1 = 0$;
while 1,
    $x_{\text{new}} = \max\{\underline{x}, \min\{\overline{x}, x_{\text{init}} + \alpha p\}\}$; $f_{\text{new}} = fun(fun, x_{\text{new}})$; $i = i + 1$;
    if (isnan($f_{\text{new}}$)), $f_{\text{new}} = +\infty$; end;
    $\mathtt{dF}_{i+1} = f_{\text{new}} - f_{\text{init}}$; $\mathcal{A}_{i+1} = \alpha$; $\mu = (f_{\text{init}} - \mathtt{dF}_{i+1})/(\alpha \mathtt{gp})$;
    if ($\mu|\mu - 1| \geq \beta$ or eff $== 1$)
        eff $=$ 1; % line search efficient
        if $\sim$ exact, break; end;
    elseif ($i > 1$ & $\sim$ descent & rob $> 0$)
        eff $= 2$; % robust nonmonotone step accepted
        break;
    elseif ($i \geq$ lmax) % limit on function values reached
        if descent, eff $=$ 3; % descent else, eff $=$ 4; % no descent end;
        break;
    end;
    % update bracket
    if descent
        % update bracket for descent
        if ($\mu \geq \frac{1}{2}$), $\underline{\alpha} = \alpha$;
        else % linear decrease or more
            if ($\alpha == \alpha_{\max}$), break; end;
            $\overline{\alpha} = \alpha$;
        end;
    elseif ($\mathtt{dF}_{i+1} < 0$) % first descent step
        descent $=$ 1;
        % create bracket for descent
        ind $= \{i \mid \mathtt{dF} < 0\}$; $\underline{\alpha} = \max(\mathcal{A}_{\mathtt{ind}})$; rob $= -1$; % lower part
        ind $= \{i \mid \mathtt{dF}_i \geq 0$ & $\mathcal{A}_i > \underline{\alpha}\}$;
        if (ind $== \emptyset$), $\overline{\alpha} = +\infty$; else, $\overline{\alpha} = \min(\mathcal{A}_{\mathtt{ind}})$; end;
    else % no descent; update robust bracket
        if ($\mathtt{dF}_{i+1} \leq$ df), $\underline{\alpha} = \alpha$; rob $= \mathtt{dF}_{i+1}$; else $\overline{\alpha} = \alpha_{\text{new}}$; end;
    end;
if first, first $= 0$; % first step
    if ($\mu < 1$),
        $\alpha = \frac{1}{2}\alpha/(1 - \mu)$; % secant step for Goldstein quotient
        if ($\alpha == 0$), $\alpha = \alpha_{\min}$; end;
    else, $\alpha = \max\{\alpha_{\min}, q\alpha\}$; % extrapolation
```
Continued on next page

```
    end;
    exact = 0;
else
    if ($\overline{\alpha} == \infty$), $\alpha = \alpha q$; % extrapolation
    elseif ($\underline{\alpha} == 0$), $\alpha = \frac{1}{2}\alpha/(1-\mu)$; % contraction
    else
        switch bis
        case 0 % geometric mean bisection
            $\underline{\alpha}_0 = \max\{\underline{\alpha}, \alpha_{\min}\}$; $\alpha = \sqrt{\underline{\alpha}_0\,\overline{\alpha}}$;
        case 1 % cubic bisection
            $\underline{\alpha}_0 = \max\{\underline{\alpha}, \alpha_{\min}\}$; $\alpha = \underline{\alpha}(\overline{\alpha}/\alpha_0)^{1/3}$;
        case 2 % geometric mean and cubic bisection
            gc = mod(nf, mbis); $\underline{\alpha}_0 = \max\{\underline{\alpha}, \alpha_{\min}\}$;
            if gc, $\alpha = \underline{\alpha}\sqrt{\overline{\alpha}/\alpha_0}$; else, $\alpha = \underline{\alpha}(\overline{\alpha}/\alpha_0)^{1/3}$; end;
        end;
    end;
    $\alpha = \min(\alpha, \alpha_{\max})$;
end;
nf = nf + $i$;
robustStep; % robust step size
```

2.4 Avoiding too many null steps (*nullStep*)

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

2.4 Algorithm. (**nullStep**)

Purpose: Try to prevent producing the null steps

function [point, step, par, info] = **nullStep**(point, step, tune, par, info);

```
flags = ($\|s\| == 0$);
if (nnull > 2 & flags)
    if (eff == 4), $\widehat{x} = \max(\underline{x}, \min(x_{\text{best}}(1-\texttt{del}), \overline{x}))$;
    else, $\widehat{x} = \max(\underline{x}, \min(x(1-\texttt{del}), \overline{x}))$;
    end;
    ind = $\{i \mid \widehat{x}_i = 0\}$; $\widehat{x}_{\texttt{ind}} = \texttt{del}$; $s = \widehat{x} - x$; $x = \widehat{x}$; flags = ($\|s\| == 0$);
```

Continued on next page

```
    if flags, nnull = nnull + 1; else, nnull = 0; end;
    $f = fun(x)$; nf = nf + 1;
    if isnan($f$), $f = +\infty$; 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_{init} can be computed by an arbitrary local method. Then the search direction p_{init} will be improved to be a direction in an adequate subspace by approximating the solution p of the problem

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

3.1 The reduced gradient

Before computing the reduced gradient, we adjust the components of the gradient that are ∞ or NaN.

3.1 Algorithm. (**adjustGrad**)

```
Purpose: Adjust the gradient vector $g$
function [point] = adjustGrad(point, tune);
ind = $\{i \mid \text{isnan}(g_i)\}$;
if ind $\neq \emptyset$, % NaN in gradient
    ind1 = $\{i \mid x_i - \underline{x}_i > \overline{x}_i - x_i\}$;
    ind2 = (ind & ind1); $g_{\texttt{ind2}} = \Delta_g *$ ones(length(ind2), 1);
    ind3 = (ind & $\sim$ ind1); $g_{\texttt{ind3}} = -\Delta_g *$ ones(length(ind3), 1);
end;
ind = $\{i \mid g_i = +\infty\}$; lind = length(ind); % $+\infty$ in gradient
if ind $\neq \emptyset$, $g_{\texttt{ind}} = \Delta_g *$ ones(lind, 1); end;
ind = $\{i \mid g_i = -\infty\}$; lind = length(ind); % $-\infty$ in gradient
if ind $\neq \emptyset$, $g_{\texttt{ind}} = -\Delta_g *$ ones(lind, 1); end;
```

The following algorithm shows how to compute the reduced gradient:

3.2 Algorithm. (**redGrad**)

```
Purpose: Compute the reduced gradient $g_{\text{red}}$
function [point] = redGrad(point);
$g_{\text{red}} = g$; $I = \{i \mid x_i \le \underline{x}_i\}$; $(g_{\text{red}})_I = \min(0, (g_{\text{red}})_I)$;
$I = \{i \mid x_i \ge \overline{x}_i\}$; $(g_{\text{red}})_I = \max(0, (g_{\text{red}})_I)$;
```

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**)

```
Purpose: Find the free index set $I_-$
function [point, par] = findFreeNeg(point, par, tune);
% find free indices $I_-$
$I_- = \{i \mid x_i > \underline{x}$ & $x_i < \overline{x}\}$; $n_{I_-}$ = length($I_-$); fixed = $(n_{I_-} < n_I)$;
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 $n_{I_-} > n_I$ 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 ε.

• **Nonmonotone** (`monotone` $= 0$): The current point did not improve the function value and the norm of gradient restricted to $I_-(x)$ is below ε.

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

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 `freeing` $= 0$. In the other iterations, `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**)

Purpose: Find the free index set and update working set

function [`point`, `par`, `info`] = **findFreePos**(`point`, `par`, `tune`, `info`);

```
% find free indices $I_+$ if freeing holds
$\rho = (1/\max(1, \texttt{ng} - 1))$; freeing = (freeing or $\|g_{\text{new}}\|^2 < \rho\|g_{\text{red}}\|^2$);
if freeing, % freeing step: corner, monotone, nonmonotone and local
    $I_+ = \{i \mid (x_i > \underline{x}_i \ \& \ x_i < \overline{x}_i)$ or $(g_{\text{red}})_i \neq 0\}$; $n_{I_+}$ = length($I_+$);
    if (ng == 1 or $n_{I_+} > n_I$), $I_- = I_+$; nlocal = 0; end;
end;
% update working set
$I = I_-$; $n_I$ = length($I$); $\omega = \|g_I\|^2$;
```

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,

$$S := \{s^1, ..., s^m\} = \{x^1 - x^0, ..., x^m - x^{m-1}\}, \tag{14}$$

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

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

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

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

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,

$$H := S^T Y = S^T BS \tag{18}$$

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

$$\begin{array}{rcl} h & := & S^T Bp = Y^T p = S^T y, \\ 0 < \gamma & := & p^T Bp = y^T p = \dfrac{f(x+\alpha p) - f - \alpha g^T p}{\alpha^2/2}, \end{array} \tag{19}$$

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 $f(x+Sz)$ 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

$$x_{\text{new}} = x + Sz_{\text{new}}, \quad g_{\text{new}} := g(x_{\text{new}}) = g + Yz_{\text{new}},$$

and we have

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

We may now cheaply form the augmented matrices

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

and the augmented vectors

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

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 *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 `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. (**updateSubspace**)

Purpose: Update the subspace information
function [`point`] = **updateSubspace**(`point`, `step`, `par`, `tune`);

Continued on next page

```
flagnull = (nnull == 0);
if flagnull,
    gy = $g^Ty$; powell = |gy|/$\omega$; flagpowell = (powell ≥ $\Delta_{po}$); flaglocal = (nlocal ≤ nwait);
    subOk = (flaglocal & (∼ flaglocal | powell ≥ $\Delta_{po}$));
    if subOk,
        if (ch < $m$), ch = ch + 1 else, ch = 1; end;
        $S_{:\mathtt{ch}} = s$; $Y_{:\mathtt{ch}} = y$;
        if typeH, $H_{\mathtt{ch}:} = s^TY$; else, $H_{\mathtt{ch}:} = y^TS$; end;
        $H_{:\mathtt{ch}} = H_{\mathtt{ch}:}^T$; nh = nh + 1;
    end;
end;
```

3.4 The quasi-Newton direction

We construct a Hessian approximation of the form

$$B = D + WXW^T, \tag{23}$$

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 Σ is invertible, too, and we conclude that $X = Z\Sigma^{-1}Z^T$, hence

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

$\square$

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 $D_{II}p_I + U_{I:}\Sigma^{-1}U_{I:}^T p_I = -g_I$, 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

$$\begin{aligned} M &:= \Sigma + U_{I:}^T D_{II}^{-1} U_{I:} = U^T S + U_{I:}^T D_{II}^{-1} U_{I:} = U_{I:}^T D_{II}^{-1}(D_{II}S_{I:} + U_{I:}) \\ &= U_{I:}^T D_{II}^{-1} Y_{I:} = (Y_{I:} - D_{II}S_{I:})^T D_{II}^{-1} Y_{I:} = Y_{I:}^T D_{II}^{-1} Y_{I:} - S_{I:}^T Y_{I:} \\ &= Y_{I:}^T D_{II}^{-1} Y_{I:} - H \end{aligned}$$

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

$$p_I = D_{II}^{-1}(h - tg_I) \tag{25}$$

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

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

$$\frac{g^T p}{\sqrt{g^T g \cdot p^T p}} \le -\Delta_{\text{angle}}. \tag{26}$$

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

$$p_{\text{new}} = p - tg \tag{27}$$

with a suitable factor $t \ge 0$; 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_{new} in the place of p

$$\frac{g^T p_{\text{new}}}{\sqrt{g^T g \cdot p_{\text{new}}^T p_{\text{new}}}} \le -\Delta_{\text{angle}}. \tag{28}$$

holds. In terms of the σ_i, this reads

$$\frac{\sigma - t\sigma_1}{\sqrt{\sigma_1(\sigma_2 - 2t\sigma + t^2\sigma_1)}} \le -\Delta_{\text{angle}}.$$

If $\sigma_{\text{new}} \le -\Delta_{\text{angle}}$, 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 *enforceAngle*:

3.7 Algorithm. (enforceAngle)

Purpose: Enforce the angle condition

function [step] = **enforceAngle**(point, step, par, tune);

```
$\sigma = g_I^T p_I$;
% move away from maximizer or saddle point
if ($\sigma > 0$), act = $\{i \in I \mid g_i p_i > 0\}$; $(p_I)_{\texttt{act}} = -(p_I)_{\texttt{act}}$; $\sigma = -\sigma$; end;
$\sigma_1 = g_I^T g_I$; $\sigma_2 = p_I^T p_I$; $\sigma_3 = \sigma_1\sigma_2$; $\sigma_{\text{new}} = \sigma/\sqrt{\sigma_3}$;
if ($\sigma_{\text{new}} \le -\Delta_{\text{angle}}$)
else
    $w = (\sigma_3 \max(\Delta_w, 1-\sigma_{\text{new}}^2))/(1-\Delta_{\text{angle}}^2)$; $t = (\sigma + \Delta_{\text{angle}}\sqrt{w})/\sigma_1$;
    if ($w > 0$ & $t \ne \pm\infty$), $p_I = p_I - t g_I$; else, $p_I = -g_I$; end;
end;
```

The following algorithm computes (25) and calls *enforceAngle* to enforce the angle condition:

3.8 Algorithm. (quasiNewtonDir)

Purpose: Compute quasi Newton direction

function [point, step] = **quasiNewtonDir**(point, step);

```
YY = $Y_{I:} \circ Y_{I:}$; SS = $S_{I:} \circ S_{I:}$; $d = \sqrt{(\sum_{i=1}^m \texttt{YY}_{:i})//(\sum_{i=1}^m \texttt{SS}_{:i})}$;
Ok = $\{i \mid \text{isnan}(d_i) \text{ or } d_i == 0 \text{ or } d_i == \pm\infty\}$; $d_{\texttt{Ok}} = 1$; $d = d(:)$;
$U = Y_{I:} - d \circ S_{I:}$; $M = (Y_{I:}^T(Y_{I:}//d)) - H$; $z = M\backslash(U^T(g_I//d))$;
$p_I = (Uz - g_I)//d$; enforceAngle;
```

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

$$p := Sq + p_{\text{init}} \tag{29}$$

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

Then (37) requires

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

and we must have

$$q = H^{-1}r. \tag{30}$$

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

$$p = -\zeta p_{\text{init}} + S(z_{\text{new}} + \zeta r), \tag{31}$$

for which

$$\zeta := \frac{g^T p_{\text{init}} + q^T z_{\text{new}}}{\gamma - qr}. \tag{32}$$

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 ζ, $\gamma - qr$, is near zero, then ζ cannot be computed; hence we use a regularized computation. To do so, let us regularize γ by

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

and then compute

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

so that

$$\zeta_{\text{new}} := \frac{g^T p_{\text{init}} + q^T z_{\text{new}}}{\gamma - qr}, \quad p_{\text{new}} = -\zeta p_{\text{init}} + S(z_{\text{new}} + \zeta_{\text{new}} r). \tag{33}$$

The implementation of conjugate gradient direction. The value of γ depends on the search direction. Here the *searchDir* algorithm is used for computing γ 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 = $\min(1, \overline{x}_i - \underline{x}_i)$; % 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 ($x_i == \underline{x}_i$), $p_i$ = sc;
        elseif ($x_i == \overline{x}_i$), $p_i$ = −sc;
        elseif ($g_i < 0$), $p_i$ = sc;
        else, $p_i$ = −sc;
    end;
end;
```

• If nlocal $\neq$ 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 $g^Tp < 0$. In order to avoid zigzagging we choose the search direction p as the vector with a fixed value $g^Tp = -\overline{\gamma} < 0$ closest (with respect to the 2-norm) to the previous search direction. By Theorem 7.1 in [51],

$$p = p_{\text{old}} - \widehat{\lambda}g, \tag{34}$$

where

$$\widehat{\lambda} = \frac{\overline{\gamma} + g^Tp_{\text{old}}}{g^Tg}. \tag{35}$$

Rescaling p_{old} by a factor $\overline{\beta} > 0$, the improved direction is expressed by

$$p := \overline{\beta}p_{\text{old}} - \overline{\lambda}g, \tag{36}$$

in which

$$\overline{\lambda} := \frac{\overline{\gamma} + \overline{\beta}g^Tp_{\text{old}}}{g^Tg}.$$

Since $g^Tp = -\overline{\gamma}$, the direction will be a descent direction. This direction is computed by *AvoidZigzagDir*, using a heuristic choice of

$$\overline{\beta} := 1/(a\ell + b)^{\theta},$$

with tuning parameters satisfying $0 < \theta < 1$, $a, b > 0$.

3.10 Algorithm. (**AvoidZigzagDir**)

```
Purpose: Modify the direction to avoid zigzagging
function [step] = AvoidZigzagDir(point, step, tune, info);
$\overline{\gamma} = \max(\mathtt{gy}, 1)$; $\overline{\beta} = 1/(1 + \mathtt{nf} + 3\mathtt{ng})^{\theta}$; $\overline{\lambda} = (\overline{\gamma} + \overline{\beta}\mathtt{gp})/\omega$; $p_I = \overline{\beta}p - \overline{\lambda}g_I$;
```

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

We now compute $p_{\rm init}$ by the following algorithm:

3.11 Algorithm. (**searchDir**)

```
Purpose: Construct the search direction $p_{\rm init}$
function [point, step, par] = searchDir(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 = $g_I^T p_I$;
if (gp $\ge 0$), $J = \{i \in I \mid p_i g_i > 0\}$; $(p_I)_J = -(p_I)_J$; gp = $g_I^T p_I$; end;
ok = (gp $\le \Delta_{pg}|g_I|^T|p_I|$);
if ok, $p_I = -g_I$; gp = $p_I^T g_I$; end;
$p_{\rm init}$ = zeros($n$, 1); $p_{\rm init} = p_I$;
```

Using $p_{\rm init}$ computed by *searchDir*, γ is generated by the following algorithm:

3.12 Algorithm. (**getGam**)

```
Purpose: Compute $\gamma$ (19)
function [point, step, par, info] = getGam(point, step, par, tune, info);
df = $\Delta_f$; % goodStep needs to it for getting $\alpha_{\rm target}$
goodStep; % get $\alpha_{\rm good}$
$x_{\rm new} = \max(\underline{x}, \min(x + \alpha_{\rm good} p_{\rm init}, \overline{x}))$; $f_{\rm new} = fun(x_{\rm new})$; nf = nf + 1;
```

Continued on next page

```
if isnan($f_{\text{new}}$), $f_{\text{new}} = +\infty$; end; % adjust $f$
d2f $= |f_{\text{new}} - f - \alpha_{\text{good}} *$ gp$|$ + eps; $\gamma = 2 *$ d2f$/\alpha_{\text{good}}^2$;
```

The mentioned regularized computation is implemented by the following algorithm:

3.13 Algorithm. (**regDenom**)

```
Purpose: Construct regularize denominator
function [par] = regDenom(point, step, par, tune);
e2f $= |f_{\text{new}} - f| + \alpha_{\text{good}}(|g|^T|p_{\text{init}}|)$; denom $= \gamma - q^T r$; dcor $= \Delta_H($e2f$/\alpha_{\text{good}}^2 + |q|^T|r|)$;
if (denom $\ge 0$), denom = denom + dcor; else, denom = denom − dcor; end;
```

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, $m_0 > 0$,

• constructs the regularize denominator of (33) by calling *regDenom*,

• projects the new point into the box $\mathbf{x}$.

3.14 Algorithm. (**ConjGradDir**)

```
Purpose: Construct the conjugate gradient direction
function [point, step, par, info] = ConjGradDir(fun, point, step, par, tune, info);
getGam; % compute $\gamma$
% construct $r$, $q$ and $z_{\text{new}}$
if ($m_0 > 0$) % subspace step possible
    $c = \sum_{i\in\text{hist}} g \circ S_{:i}$; $q = \sum_{i\in\text{hist}} p_{\text{init}} \circ Y_{:i}$; rhs $= [-c, q]$; Hoh $= H_{\text{hist},\text{hist}}$;
    sol = Hoh\rhs; nsub $= \{i \mid$ isnan(sol) or sol $== \pm\infty\}$;
    if nsub $\ne \emptyset$ % no subspace step possible
        $\zeta =$ gp$/\gamma$;
        if isnan($\zeta$), $\zeta = \zeta_{\max}$; end;
        $\zeta = \min(\zeta_{\max}, \max(\zeta, \zeta_{\min}))$; cosine $= -$gp $* \zeta$; $p = -p_{\text{init}}\zeta$;
    else
        $z_{\text{new}} =$ sol$_{:1}$; $r =$ sol$_{:2}$;
```

Continued on next page

```
        $regDenom$; % construct regularize denominator
        $\zeta = (\mathtt{gp} + q^T z_{\text{new}})/\mathtt{denom}$;
        if isnan($\zeta$), $\zeta = \zeta_{\max}$; end;
        $\zeta = \min(\zeta_{\max}, \max(\zeta, \zeta_{\min}))$; $z_{\text{new}} = z_{\text{new}} + \zeta r$;
        $\mathtt{cosine} = -\mathtt{gp} * \zeta + c^T z_{\text{new}}$; $p = -p_{\text{init}}\zeta + S_{:\mathtt{hist}} z_{\text{new}}$;
    end;
else % no subspace step possible
    % here doing instead an exact line search saves function values
    $\zeta = \mathtt{gp}/\gamma$;
    if isnan($\zeta$), $\zeta = \zeta_{\max}$; end;
    $\zeta = \min(\zeta_{\max}, \max(\zeta, \zeta_{\min}))$; $\mathtt{cosine} = -\mathtt{gp} * \zeta$; $p = -p_{\text{init}}\zeta$;
end;
$x_{\text{new}} = \max(\underline{x}, \min(x + p, \overline{x}))$; $p = x_{\text{new}} - x$;
```

4 Starting point and master algorithm

4.1 The starting point (*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 ith 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. (**projStartPoint**)

Purpose: Improve the starting point
function [`point`] = **projStartPoint**(`point`, `tune`);
Continued on next page

```
if $\Delta_x == 0$, $x = \max(\overline{x}, \min(x, \underline{x}))$;
else
    ind $= \{i \mid x_i \leq \overline{x}_i,\ i = 1, \cdots, n\}$; $x_{\texttt{ind}} = \overline{x}_{\texttt{ind}} + \Delta_x \min(\Delta_u|\overline{x}_{\texttt{ind}}|, \underline{x}_{\texttt{ind}} - \overline{x}_{\texttt{ind}})$;
    ind $= \{i \mid x_i \geq \underline{x}_i,\ i = 1, \cdots, n\}$; $x_{\texttt{ind}} = \underline{x}_{\texttt{ind}} - \Delta_x \min(\Delta_u|\underline{x}_{\texttt{ind}}|, \underline{x}_{\texttt{ind}} - \overline{x}_{\texttt{ind}})$;
end;
ind $= \{i \mid x_i = \pm\infty\}$; $x_{\texttt{ind}} = \max(\underline{x}_{\texttt{ind}}, \min(0, \overline{x}_{\texttt{ind}}))$;
```

4.2 Successful iteration (*getSuccess*)

The goal of *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 *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 Δ_f.

4.2 Algorithm. (**getSuccess**)

```
Purpose: Determine whether iteration is successful or not
function [point, step, par, info] = getSuccess(fun, point, step, par, tune, info);
quad = 0; estuck = 0; Ip = $\{i \mid p_i \neq 0\}$;
defQuad=(cosine $< 0$ & Ip $\neq \emptyset$ & (CG $> 0$ or nstuck $\geq$ nsmin));
if defQuad
    % data are consistent with a definite quadratic function
    $f_{\rm new} = fun(x_{\rm new})$; nf = nf + 1;
    if isnan($f_{\rm new}$), $f_{\rm new} = +\infty$; end; % adjust $f$
    gp $= g^T p$; $\mu = (f_{\rm new} - f)/$gp;
    if ($f_{\rm new} == f$ or $\mu|\mu - 1| \geq \beta_{\texttt{CG}}$), quad = CG; end; % check s.d. condition
    estuck = (nstuck $\geq$ nsmin & $f_{\rm new} \leq f + \Delta_f$);
end;
success = (quad or estuck);
```

4.3 Initializing information (*initInfo*)

initInfo initializes the best point and its function value.

4.3 Algorithm. (**initInfo**)

```
Purpose: Initialize best point and $\Delta_f$
function [point] = initInfo(point, tune);
if ($f \neq \pm\infty$ & $f \neq 0$), $\Delta_f$ = facf $* |f|$; else, $\Delta_f = 1$; end;
$\mathtt{Df}_{1:\mathtt{mf}-1} = -\infty$; $\mathtt{Df}_{\mathtt{mf}} = \Delta_f$;
$f_{\text{best}} = f$;
```

4.4 Updating information (*updateInfo*)

The goal of *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 Δ_f and the list of its `mf` previous values are updated. It is tried that the amount of Δ_f is updated by using its `mf` previous values, preventing very tiny value; especially when the function value is very small.

4.4 Algorithm. (**updateInfo**)

```
Purpose: Update best point and $\Delta_f$
function [point, par] = updateInfo(point, par, tune, info);
% update $f_{\text{best}}$
if ($f_{\text{new}} < f_{\text{best}}$), nstuck = 0; $f_{\text{best}} = f_{\text{new}}$; $x_{\text{best}} = x_{\text{new}}$; else, nstuck = nstuck + 1; end;
dec = ($f_{\text{new}} < f$);
if dec % improvement
    monotone = 1; $\Delta_f = f - f_{\text{new}}$; nm = mod(ng, mf);
    if (nm == 0), $\mathtt{Df}_{\mathtt{mf}} = \Delta_f$; else, $\mathtt{Df}_{\mathtt{nm}} = \Delta_f$; end
elseif ($f_{\text{new}} == f$), monotone = 0; % stalled
else % no descent
    monotone = 0; $\Delta_f = \max(2\Delta_f, \Delta_m(|f| + |f_{\text{new}}|))$; nm = mod(ng, mf);
    if (nm == 0), $\mathtt{Df}_{\mathtt{mf}} = \Delta_f$; else, $\mathtt{Df}_{\mathtt{nm}} = \Delta_f$; end;
end;
$f = f_{\text{new}}$; % update $f$
```

4.5 The type of a subspace step (*typeSubspace*)

We need to determine what to be the subspace. *typeSubspace* uses three variables m_0 (length of subspace), `hist` (list of subspace basis) and `CG` (type of subspace) to determine the subspace. The

conjugacy relation is defined by

$$h = Y^T s = 0. \qquad (37)$$

It works as follow:

• If `nlocal` $<$ `nwait`, 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 `nlocal` $=$ `nwait`, 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 `nlocal` $<$ `nwait` $+ \widehat{m}$, 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

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

```
$\widehat{m} = \min(m, \mathtt{nh})$;
if (nlocal < nwait) % ordinary subspace step
    $m_0 = \min(\mathtt{ng} - 1, \widehat{m})$; hist = $[1 : m_0]$; CG = 0;
elseif (nlocal == nwait) % restart: steepest descent direction
    $m_0 = 0$; hist = $\emptyset$; perm = $[\mathtt{ch} + 1 : \widehat{m}, 1 : \mathtt{ch}]$; ch = 0;
    $S = S_{:\mathtt{perm}}$; $Y = Y_{:\mathtt{perm}}$; $H = H_{\mathtt{perm},\mathtt{perm}}$; CG = 1;
elseif (nlocal < nwait + $\widehat{m}$) % preserve conjugacy by restricting the subspace
    $m_0$ = nlocal − nwait; hist = $[1 : m_0]$; CG = 2;
else % full subspace step preserves conjugacy
    $m_0 = \widehat{m}$; hist = $[1 : \widehat{m}]$; CG = 3;
end;
```

In *typeSubspace* whenever `nlocal` $=$ `nwait`, 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 **LMBOPT**, 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, *LMBOPT* first computes the reduced gradient by *redGrad* in per iteration and then the working set is determined and updated by *findFreePos*. As long as the reduced gradient is not below a minimum threshold, it generates the starting direction p_{init} by *searchDir* and then constructs the subspace conjugate gradient direction p by *ConjGradDir* in the hope of achieving an successful iteration. Such an iteration is determined by *getSuccess* and then the best point is updated. Otherwise it performs a gradient-free line search *BLS* along a regularized direction (*enforceAngle*) since the function is not near the quadratic case. Then if the null steps are repeated at least `nnullmax` in a sequence, the point leading to such steps is replaced by *nullStep* with a point around the previous best point if *BLS* is not efficient; otherwise with the current point generated by *BLS*. This is repeated until no null step is found. Afterwards, the gradient is computed and adjusted by *adjustGrad*. In addition, the new free index set is found by *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^ℓ denote the value of x in Algorithm 1.1 after its ℓth update. Then one of the following three cases holds:*

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

(ii) We have

$$\lim_{\ell\to\infty} f(x^\ell) = \widehat{f} \in \mathbb{R}, \qquad \inf_{\ell\ge 0} \|g_{\text{red}}(x^\ell)\|_* = 0.$$

Some limit point $\widehat{x}$ of the x^ℓ satisfies $f(\widehat{x}) = \widehat{f} \le f(x^0)$ and $g_{\text{red}}(\widehat{x}) = 0$.

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

4.7 Algorithm. (**LMBOPT**, **limited memory bound constrained optimization**)

```
Purpose: Minimize smooth $f(x)$ subject to $x \in \mathbf{x} = [\underline{x}, \overline{x}]$
function [$x$, $f$, info] = LMBOPT($fun$, $dfun$, $x$, $\underline{x}$, $\overline{x}$, tune, info);
% get point
$m = \max(1, \min(m, n))$; $S$ = zeros($n, m$); $Y$ = zeros($n, m$);
$H$ = zeros($m$); ch = 0; nh = 0; $q = 0.5/\beta$;
% get par
monotone = 0; nlocal = $-1$; fixed = 0; nstuck = 0;
success = 0; nnull = 0; freeing = 1;
projStartPoint; % improve the starting point
% compute starting function value and its gradient
$[f, g] = fun(x)$; nf = 1; ng = 1;
if isnan($f$), $f = +\infty$; end; % adjust $f$
adjustGrad; initInfo;
while 1,
```

Continued on next page

```
    $redGrad$; $findFreePos$;
    % check stopping tests (gradient accuracy, work limit, stuck)
    if ($\|g_{\text{red}}\|_\infty < \varepsilon$ or nstuck > nstuckmax), break; end;
    if (nlocal > max(nwait, rfac * $n_I$)), nlocal = nwait; end; % test for local restart
    $typeSubspace$; $searchDir$; $ConjGradDir$; $getSuccess$;
    if success, $x = x_{\text{new}}$; % iteration is successful; update best point
    else % perform a line search along a regularized direction
        $enforceAngle$; $BLS$;
        $x_0 = x$; $s = x$; $x = \max(\underline{x}, \min(x + \alpha_{\text{new}} p, \overline{x}))$; $s = x - s$; % update best point
        $nullStep$;
        if flags, nnull = nnull + 1;
                if (nnull > nnullmax), break; end
        end
    end;
    if (nnull ≤ nnullmax) % significant step; get new gradient
        $g_0 = g$; $g = dfun(x)$; ng = ng + 1; $adjustGrad$; $y = g - g_0$;
    end;
    $updateInfo$; $findFreeNeg$; $updateSubspace$;
end;
```

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` + 2`ng`. 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 $i = 1, \dots, n$, by

$$x_i = (-1)^{i-1} \frac{m}{m+i}, \tag{38}$$

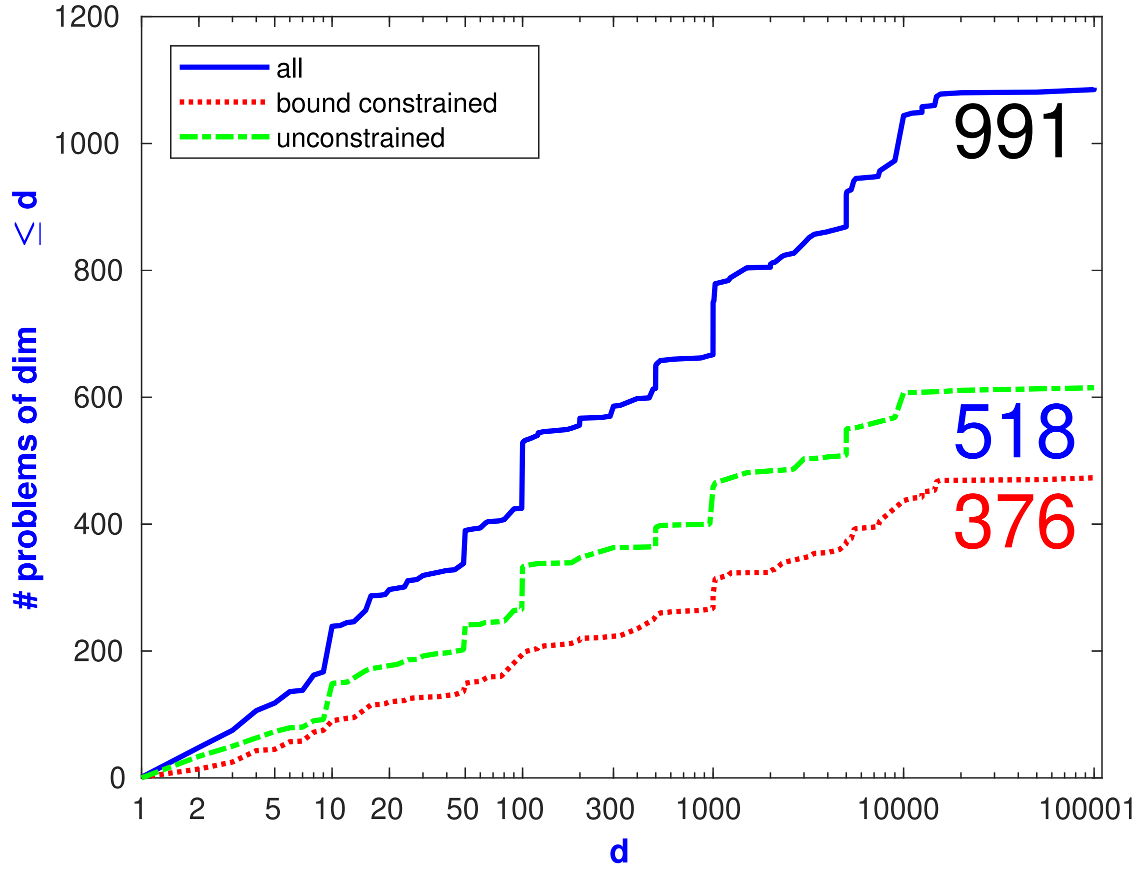

Figure 1: The number of problems with variables in a given range solved by at least one solver: 989 problems with dimensions 1 up 100001

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:

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

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

  `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

  `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

  `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

  `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

  `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

  `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

  `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.

- *LMBFG-BWX-MS* (*lt1*), obtained from

  `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 $\texttt{EIG}(\infty, 2)$ 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 $\texttt{EIG}(\infty, 2)$ 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* (*ll7*), obtained from

  `http://gratton.perso.enseeiht.fr/LBFGS/index.html,`

  is a limited memory trust-region algorithm $\texttt{EIG} - \texttt{MS}(2, 2)$ 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 (KIMIAEI & 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

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

Table 11: The problems unsolved by all solvers

BROWNBS	PALMER7A	PALMER5E	PALMER5B
OSCIGRAD:10	OSCIPATH:10	STRATEC	SBRYBND:10
SCOSINE:10	SCURLY10:10	SCOND1LS	OSCIGRAD:15
OSCIGRAD:25	ANTWERP	NONMSQRT:49	HS110:50
SBRYBND:50	RAYBENDS	RAYBENDL:66	RAYBENDS:66
HYDC20LS	FLETCHBV:100	HS110:100	NONMSQRT:100
OSCIGRAD:100	SBRYBND:100	SCOSINE:100	SCURLY10:100
SCOND1LS:102	RAYBENDL:130	RAYBENDS:130	QR3DLS
GRIDGENA:170	DRCAV1LQ	HS110:200	SPMSRTLS:499
PENALTY2:500	SBRYBND:500	SCOND1LS:502	MSQRTALS:529
MSQRTBLS:529	NONMSQRT:529	GRIDGENA	QR3DLS:610
LINVERSE:999	CURLY20	CHENHARK	FLETCHBV:1000
PENALTY2:1000	SBRYBND	SCOSINE	SCURLY10
SSCOSINE	SPMSRTLS:1000	SCOND1LS:1002	MSQRTALS:1024
MSQRTBLS:1024	NONMSQRT:1024	RAYBENDL:1026	RAYBENDS:1026
DRCAV1LQ:1225	DRCAV2LQ:1225	DRCAV3LQ:1225	GRIDGENA:1226
RAYBENDL:2050	GRIDGENA:2114	EIGENALS:2550	GRIDGENA:3242
DRCAV3LQ:4489	GRIDGENA:4610	MSQRTALS:4900	MSQRTBLS:4900
SPMSRTLS:4999	FLETCBV3:5000	FLETCHBV:5000	SBRYBND:5000
SCOSINE:5000	SPARSINE:5000	SSCOSINE:5000	SCOND1LS:5002
BRATU1D:5003	GRIDGENA:6218	CURLY10:10000	CURLY20:10000
CURLY30:10000	FLETCBV3:10000	FLETCHBV:10000	NONCVXUN:10000
SCOSINE:10000	SCURLY10:10000	SPARSINE:10000	SPMSRTLS:10000
SSCOSINE:10000	DRCAV3LQ:10816	ODNAMUR	GRIDGENA:12482
SSCOSINE:100000			

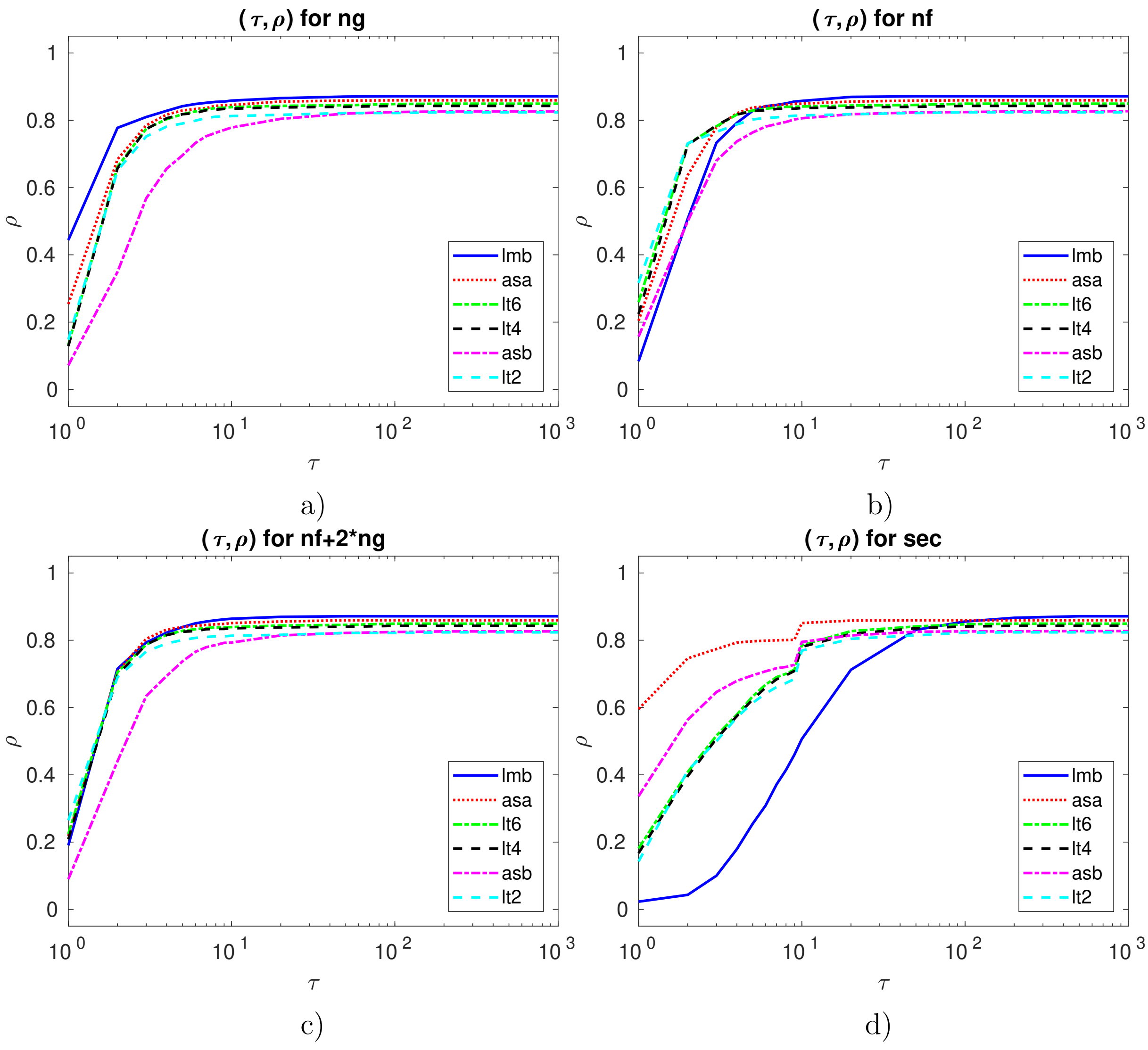

Figure 2: (a)-(e): Performance plots for `ng`/(`best ng`), `nf`/(`best nf`), `nf2g`/(`best nf2g`) and `msec`/(`best msec`), respectively. ρ designates the percentage of problems solved within a factor τ of the best solver. Problem solved by no solver are ignored.

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_{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` ≥ 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.

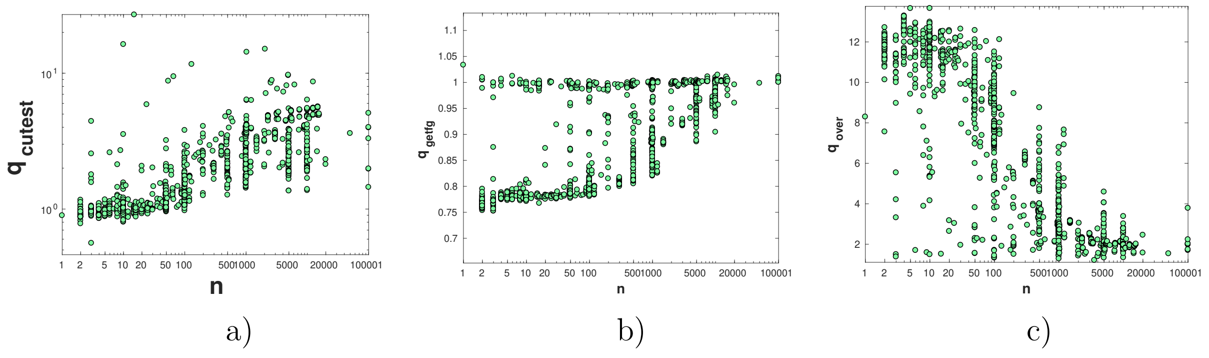

Figure 3: Comparison of $q_{\texttt{cutest}} := \frac{t_g(\texttt{cutest})}{t_f(\texttt{cutest})}$, $q_{\texttt{getfg}} := \frac{t_g(\texttt{getfg})}{t_f(\texttt{getfg})}$ and $q_{\text{over}} := \frac{t_{f2g}(\texttt{getfg})}{t_{f2g}(\texttt{cutest})}$ versus dimensions, respectively, where t_f and 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

stopping test:		$\|g\|_\infty \leq$ `1e-06,`		`sec` $\leq$ `300,`			`nf` $+ 2 *$ `ng` $\leq 20 *$ `n` $+$ `10000`					
991 of 1088 problems solved										mean efficiency in %		
dim∈[1,100001]						# of anomalies				for cost measure		
solver		solved	#100	!100	T_{mean}	#n	#t	#f	nf2g	ng	nf	msec
LMBOPT	*lmb*	948	170	143	4544	92	48	0	58	69	42	11
ASACG	*asa*	935	155	26	1416	98	21	34	58	59	51	63
LMBFG-EIG-MS	*lt6*	924	108	48	2970	119	26	19	60	57	60	34
LMBFG-EIG-curve-inf	*lt4*	918	94	33	3330	118	25	27	60	56	59	34
ASABCP	*asb*	900	75	52	2404	142	25	21	41	36	44	46
LMBFG-DDOGL	*lt2*	896	113	52	2937	61	21	110	60	56	59	33
CGdescent	*cgd*	895	135	14	2559	77	17	99	54	56	47	55
LMBFG-EIG-MS-2-2	*lt7*	895	38	0	3390	112	21	60	50	45	57	34
LMBFG-BWX-MS	*lt1*	888	39	1	2694	56	21	123	51	45	58	32
SPG	*spg*	840	103	69	5901	182	58	8	34	34	31	9
LBFGSB	*lbf*	803	238	192	713	0	0	285	57	51	61	32
LMBFG-EIG-inf-2	*lt5*	753	85	25	3275	76	26	233	50	47	49	28
LMBFGS-TR	*ll3*	733	101	43	2904	242	92	21	48	44	48	36
LMBFG-MTBT	*ll2*	669	75	22	2257	55	14	350	45	41	46	26
LMBFG-MT	*ll1*	657	101	49	2677	57	14	360	45	39	48	32
984 of 1088 problems solved, `sec` $\leq$ 1800										mean efficiency in %		
LMBOPT	*lmb*	953	257	227	6969	115	20	0	67	75	52	17
ASACG	*asa*	936	286	243	2135	116	1	35	67	65	63	82
LMBFG-EIG-MS	*lt6*	932	508	469	6079	134	2	20	71	64	75	48

Table 14: The summary results for unconstrained and bound constrained problems

stopping test:	$\|g\|_\infty \leq$ `1e-06`,			`sec` $\leq$ `300`,				$\mathtt{nf} + 2 * \mathtt{ng} \leq 20 * \mathtt{n} + 10000$				
552 of 615 problems without bounds solved									mean efficiency in %			
dim∈[1,100001]						# of anomalies			for cost measure			
solver		solved	#100	!100	T_{mean}	#n	#t	#f	nf2g	ng	nf	msec
ASACG	*asa*	533	132	121	1331	53	16	13	67	66	61	82
LMBOPT	*lmb*	531	162	160	3962	50	34	0	68	75	53	16
LMBFG-EIG-MS	*lt6*	522	271	258	3055	68	21	4	69	59	73	47
425 of 473 problems with bounds solved									mean efficiency in %			
LMBOPT	*lmb*	417	106	78	5283	42	14	0	66	74	51	20
ASACG	*asa*	402	148	116	1530	43	5	21	65	61	64	79
LMBFG-EIG-MS	*lt6*	402	225	199	2859	51	5	15	71	64	73	48

stopping test:	$\|g\|_\infty \leq$ `1e-06`,			`sec` $\leq$ `1800`,				$\mathtt{nf} + 2 * \mathtt{ng} \leq 20 * \mathtt{n} + 10000$				
552 of 615 problems without bounds solved									mean efficiency in %			
ASACG	*asa*	533	137	126	1391	67	1	14	68	67	62	82
LMBOPT	*lmb*	533	155	153	5677	67	15	0	68	75	53	15
LMBFG-EIG-MS	*lt6*	522	273	260	3721	87	2	4	69	59	74	47
432 of 473 problems with bounds solved									mean efficiency in %			
LMBOPT	*lmb*	420	102	74	8608	48	5	0	66	74	50	20
LMBFG-EIG-MS	*lt6*	410	235	209	9082	47	0	16	73	66	76	48
ASACG	*asa*	403	149	117	3119	49	0	21	66	61	64	81

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, *LMBOPT* 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

• *LMBOPT* is the best in terms of `ng` for $n < 50001$ and it is the best in terms of the number of solved problems for $n \leq 100$ and $n \in [1001, 3000]$.

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

• For very large scale problems, $n \in [50001, 100001]$, *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

stopping test:	$\|g\|_\infty \leq$ `1e-06`,		`sec` $\leq$ `1800`,				`nf` $+ 2*$ `ng` $\leq 20*$ `n` $+$ `10000`					
116 of 118 problems solved										mean efficiency in %		
dim∈[1,5]						# of anomalies				for cost measure		
solver		solved	#100	!100	T_{mean}	#n	#t	#f	nf2g	ng	nf	msec
LMBOPT	*lmb*	115	29	22	199	3	0	0	73	84	58	18
ASACG	*asa*	110	40	32	25	6	0	2	76	77	69	85
LMBFG-EIG-MS	*lt6*	106	61	53	36	12	0	0	77	69	82	58
n ∈[6,10], 112 of 121 problems solved						# of anomalies				for cost measure		
LMBOPT	*lmb*	112	34	30	321	9	0	0	69	74	58	17
ASACG	*asa*	107	47	42	37	6	0	8	74	70	71	83
LMBFG-EIG-MS	*lt6*	94	40	35	110	26	0	1	58	53	62	46
n ∈[11,30], 75 of 80 problems solved						# of anomalies				for cost measure		
LMBOPT	*lmb*	74	27	22	291	6	0	0	75	82	59	16
LMBFG-EIG-MS	*lt6*	74	28	24	494	5	0	1	70	60	75	69
ASACG	*asa*	68	29	20	17	8	0	4	70	67	65	78
n ∈[31,100], 194 of 209 problems solved						# of anomalies				for cost measure		
LMBOPT	*lmb*	188	49	46	449	21	0	0	67	76	53	16
ASACG	*asa*	184	60	55	117	19	0	6	68	68	63	82
LMBFG-EIG-MS	*lt6*	184	93	88	133	23	0	2	71	65	75	59
n ∈[101,300], 51 of 58 problems solved						# of anomalies				for cost measure		
LMBFG-EIG-MS	*lt6*	50	24	22	264	5	0	3	72	64	73	56
ASACG	*asa*	49	21	19	333	6	0	3	72	72	65	78
LMBOPT	*lmb*	48	9	8	1055	10	0	0	64	72	47	15
n ∈[301,1000], 141 of 163 problems solved						# of anomalies				for cost measure		
LMBFG-EIG-MS	*lt6*	137	71	67	2004	21	0	5	68	62	72	25
ASACG	*asa*	136	42	38	422	24	0	3	65	63	65	83
LMBOPT	*lmb*	135	35	32	2549	28	0	0	62	71	47	12
n ∈[1001,3000], 81 of 94 problems solved						# of anomalies				for cost measure		
LMBOPT	*lmb*	79	7	7	11536	15	0	0	63	74	44	12
LMBFG-EIG-MS	*lt6*	79	67	65	5708	13	0	2	79	72	80	31
ASACG	*asa*	76	9	7	1458	15	0	3	58	59	51	79
n ∈[3001,10000], 173 of 201 problems solved						# of anomalies				for cost measure		
ASACG	*asa*	168	31	25	4790	28	0	5	60	57	59	82
LMBFG-EIG-MS	*lt6*	168	104	97	13170	28	0	5	72	64	76	44
LMBOPT	*lmb*	166	49	44	21178	23	12	0	64	72	49	22
n ∈[10001,50000], 35 of 37 problems solved						# of anomalies				for cost measure		
LMBFG-EIG-MS	*lt6*	35	18	18	60050	1	0	1	86	70	91	65
ASACG	*asa*	32	3	3	10902	4	0	1	57	52	54	81
LMBOPT	*lmb*	32	14	14	31617	0	5	0	79	84	57	37
n ∈[50001,100001], 6 of 7 problems solved						# of anomalies				for cost measure		
ASACG	*asa*	6	4	2	105117	0	1	0	67	64	72	79
LMBFG-EIG-MS	*lt6*	5	2	0	107508	0	2	0	49	44	54	39
LMBOPT	*lmb*	4	4	2	160765	0	3	0	57	57	53	44

Table 16: The summary results for hard problems

stopping test: $\|g\|_\infty \le$ `1e-06`, `sec` $\le$ `3600`, `nf` $+ 2 *$ `ng` $\le$ `50` $*$ `n` $+$ `200000`												
62 of 93 problems solved									mean efficiency in %			
dim∈[1,100001]						# of anomalies			for cost measure			
solver		solved	#100	!100	T_{mean}	#n	#t	#f	nf2g	ng	nf	msec
LMBFG-EIG-MS	*lt6*	49	31	31	152667	28	6	10	47	42	48	33
LMBOPT	*lmb*	46	15	14	179457	35	12	0	38	42	30	17
ASACG	*asa*	45	17	16	120625	32	0	16	40	40	36	47
31 of 56 problems without bounds solved									mean efficiency in %			
ASACG	*asa*	25	7	7	182304	23	0	8	37	35	34	43
LMBFG-EIG-MS	*lt6*	23	15	15	265649	21	6	6	37	34	38	21
LMBOPT	*lmb*	22	9	9	208808	23	11	0	31	34	25	16
31 of 37 problems with bounds solved									mean efficiency in %			
LMBFG-EIG-MS	*lt6*	23	17	17	1148	2	0	3	73	67	75	61
LMBOPT	*lmb*	22	3	3	1641	5	1	0	55	64	42	21
ASACG	*asa*	20	8	8	262	5	0	3	57	60	49	69

Table 17: The hard problems unsolved by all solvers

OSCIPATH:10	SCOND1LS	ANTWERP	HYDC20LS
FLETCHBV:100	NONMSQRT:100	SBRYBND:100	SCOSINE:100
SCURLY10:100	PENALTY2:500	SCOND1LS:502	NONMSQRT:529
FLETCHBV:1000	PENALTY2:1000	SCOSINE	SCURLY10
SSCOSINE	SCOND1LS:1002	NONMSQRT:1024	FLETCBV3:5000
FLETCHBV:5000	SBRYBND:5000	SCOSINE:5000	SCOND1LS:5002
BRATU1D:5003	FLETCBV3:10000	FLETCHBV:10000	NONCVXUN:10000
SCOSINE:10000	SCURLY10:10000	SSCOSINE:100000	

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 `nfmax` and `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

- *LMBOPT* is the second best solver in terms of the number of solved problems.
- *LMBOPT* and *LMBFG-EIG-MS* 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

[1] R. Andreani, A. Friedlander, and J. M. Martínez. On the solution of finite-dimensional variational inequalities using smooth optimization with simple bounds. *J. Optim. Theory Appl.* **94** (1997), 635–657. [3]

[2] Jonathan Barzilai and Jonathan M. Borwein. Two-point step size gradient methods. *IMA J. Numer. Anal.* **8** (198), 141–148. [4]

[3] D.P. Bertsekas. Projected Newton methods for optimization problems with simple constraints. *SIAM J. Control Opim.* **20** (1982), 221–246. [4]

[4] E. G. Birgin, I. Chambouleyron, and J. M. Martínez. Estimation of the optical constants and thickness of thin films using unconstrained optimization. *J. Comput. Phys.* **151** (1999), 862–880. [3, 4]

[5] E.G. Birgin and J.M. Martínez. A box-constrained optimization algorithm with negative curvature directions and spectral projected gradients. In *Topics in Numerical Analysis* (Goetz Alefeld and Xiaojun Chen, eds.), Vol. 15 of *Computing Supplementa*, pp. 49–60. Springer Vienna (2001). [4]

[6] Ernesto G. Birgin, José Mario, and Martínez Marcos Raydan. Inexact spectral projected gradient methods on convex sets. *IMA J. Numer. Anal.* **23** (2003), 539–559. [4]

[7] Ernesto G. Birgin and José Mario Martínez. Large-scale active-set box-constrained optimization method with spectral projected gradients. *Comput. Optim. Appl.* **23** (2002), 101–125. [4]

[8] Ernesto G. Birgin, José Mario Martínez, and Marcos Raydan. Nonmonotone spectral projected gradient methods on convex sets. *SIAM J. Optim.* **10** (1999), 1196–1211. [4, 37]

[9] Ernesto G. Birgin, José Mario Martínez, and Marcos Raydan. Algorithm 813: Spg-software for convex-constrained optimization. *ACM Trans. Math. Softw.* **27** (2001), 340–349. [4, 37]

[10] Oleg Burdakov, Lujin Gong, Spartak Zikrin, and Ya-xiang Yuan. On efficiently combining limited-memory and trust-region techniques. *Math. Program. Comput.* **9** (2017), 101–134. [37, 38]

[11] R.H. Byrd, P. Lu, J. Nocedal, and C. Zhu. A limited memory algorithm for bound constrained optimization. *SIAM J. Sci. Comput.* **16** (1995), 1190. [4, 37]

[12] Richard H Byrd, Jorge Nocedal, and Robert B Schnabel. Representations of quasi-newton matrices and their use in limited memory methods. *Math. Program.* **63** (1994), 129–156. [4]

[13] Paul Calamai and Jorge Moré. Projected gradient methods for linearly constrained problems. *Math. Program.* **39** (1987), 93–116. [4]

[14] Andrew R Conn, Nicholas I M Gould, and Philippe L Toint. Testing a class of methods for solving minimization problems with simple bounds on the variables. *Math. Comp.* **50** (1988), 399–430. [4]

[15] Andrew R. Conn, Nicholas I. M. Gould, and Philippe L. Toint. A globally convergent augmented lagrangian algorithm for optimization with general constraints and simple bounds. *SIAM J. Numer. Anal.* **28** (1991), 545–572. [3]

[16] A.R. Conn, N.I.M. Gould, and Ph.L. Toint. Global convergence of a class of trust region algorithms for optimization with simple bounds. *SIAM J. Numer. Anal.* **25** (1988), 433. [4]

[17] A. Cristofari, M. De Santis, S. Lucidi, and F. Rinaldi. A two-stage active-set algorithm for bound-constrained optimization. *J. Optim. Theory Appl.* **172** (2017), 369–401. [37]

[18] Y. H. Dai. On the nonmonotone line search. *J. Optim. Theory Appl.* **112** (2002), 315–330. [4]

[19] Yu-Hong Dai and Roger Fletcher. Projected Barzilai-Borwein methods for large-scale box-constrained quadratic programming. *Numer. Math.* **100** (2005), 21–47. [4]

[20] Yu-Hong Dai and Roger Fletcher. New algorithms for singly linearly constrained quadratic programs subject to lower and upper bounds. *Math. Program.* **106** (2006), 403–421. [4]

[21] Yu-Hong Dai, William W. Hager, Klaus Schittkowski, and Hongchao Zhang. The cyclic Barzilai–Borwein method for unconstrained optimization. *IMA J. Numer. Anal.* **26** (2006), 604–627. [4, 5]

[22] R. S. Dembo and U. Tulowitzki. On the minimization of quadratic functions subject to box constraints. Technical report, School of Organization and Management, Yale University, New Haven, CT (1983). [4]

[23] Elizabeth D Dolan and Jorge J Moré. Benchmarking optimization software with performance profiles. *Math. Program.* **91** (2001), 13. [38]

[24] Zdenek Dostál. Box constrained quadratic programming with proportioning and projections. *SIAM J. Optim* **7** (1997), 871–887. [4]

[25] Zdenek Dostál. A proportioning based algorithm with rate of convergence for bound constrained quadratic programming. *Numer. Algorithms* **34** (2003), 293–302. [4]

[26] Zdenek Dostál, Ana Friedlander, and Sandra A Santos. Solution of coercive and semicoercive contact problems by feti domain decomposition. *Contemp. Math.* **218** (1998), 82–93. [3]

[27] J. C. Dunn. On the convergence of projected gradient processes to singular critical points. *J. Optim. Theory Appl.* **55** (1987), 203–216. [4]

[28] Roger Fletcher. On the Barzilai-Borwein method. *Optimization and Control with Applications* (2005), 235–256. [4]

[29] P. E. Gill, W. Murray, and M. H. Wright. *Practical Optimization.* Academic Press, London (1981). [3]

[30] W. Glunt, T. L. Hayden, and M. Raydan. Molecular conformations from distance matrices. *J. Comput. Chem.* **14** (1993), 114–120. [4]

[31] A. Goldstein and J. Price. An effective algorithm for minimization. *Numer. Math.* **10** (1967), 184–189. [5, 15]

[32] N.I.M. Gould, D. Orban, and Ph.L. Toint. CUTEst: a constrained and unconstrained testing environment with safe threads for mathematical optimization. *Comput. Optim. Appl.* **60** (2015), 545–557. [1, 7, 34, 38]

[33] L. Grippo, F. Lampariello, and S. Lucidi. A nonmonotone line search technique for newton's method. *SIAM J. Numer. Anal.* **23** (1986), 707–716. [4]

[34] L. Grippo and M. Sciandrone. Nonmonotone globalization techniques for the Barzilai-Borwein gradient method. *Comput. Optim. Appl.* **23** (2002), 143–169. [4]

[35] W. W. Hager. Dual techniques for constrained optimization. *J. Optim. Theory Appl.* **55** (1987), 37–71. [3]

[36] W. W. Hager. Analysis and implementation of a dual algorithm for constrained optimization. *J. Optim. Theory Appl.* **79** (1993), 427–462. [3]

[37] W. W. Hager and H. Zhang. CG_DESCENT user's guide. Technical report, Department of Mathematics, University of Florida, Gainesville, FL (2004). [5]

[38] William W. Hager and Hongchao Zhang. A new conjugate gradient method with guaranteed descent and an efficient line search. *SIAM J. Optim.* **16** (2005), 170–192. [5, 37]

[39] William W. Hager and Hongchao Zhang. Algorithm 851: CG_DESCENT, a conjugate gradient method with guaranteed descent. *ACM Trans. Math. Softw.* **32** (2006), 113–137. [5, 37]

[40] W.W. Hager and H. Zhang. A new active set algorithm for box constrained optimization. *SIAM J. Optim.* **17** (2006), 526–557. [4, 36]

[41] W.W. Hager and H. Zhang. A survey of nonlinear conjugate gradient methods. *Pac. J. Optim.* **2** (2006), 35–58. [5, 37]

[42] W.W. Hager and H. Zhang. The limited memory conjugate gradient method. *SIAM J. Optim.* **23** (2013), 2150–2168. [37]

[43] M. Kimiaei and A. Neumaier. Testing and tuning optimization algorithm. Preprint, Vienna University, Fakultät für Mathematik, Universität Wien, Oskar-Morgenstern-Platz 1, A-1090 Wien, Austria (2019). [36, 38, 41]

[44] Y. Lin and C. W. Cryer. An alternating direction implicit algorithm for the solution of linear complementarity problems arising from free boundary problems. *Appl. Math. Optimization* **13** (1987), 1–17. [3]

[45] W. Liu and Y. H. Dai. Minimization algorithms based on supervisor and searcher cooperation. *J. Optim. Theory Appl.* **111** (2001), 359–379. [4]

[46] P Lötstedt. Solving the minimal least squares problem subject to bounds on the variables. *BIT* **24** (1984), 206–224. [3]

[47] José Mario Martínez. BOX-QUACAN and the implementation of augmented lagrangian algorithms for minimization with inequality constraints. *Comput. Appl. Math.* **19** (2000), 31–36. [3]

[48] J. L. Morales and J. Nocedal. Remark on ”Algorithm 778: L-BFGS-B: Fortran subroutines for large-scale bound constrained optimization”. *ACM Trans. Math. Softw.* **38** (2011), 167–185. [37]

[49] J J Moré and G Toraldo. Algorithms for bound constrained quadratic programming problems. *Numer. Math.* **55** (1989), 377–400. [4]

[50] J J Moré and G Toraldo. On the solution of large quadratic programming problems with bound constraints. *SIAM J. Optim.* **1** (1991), 93–113. [3, 4]

[51] A. Neumaier and B. Azmi. Line search and convergence in bound-constrained optimization. Technical report, University of Vienna (2019). [5, 6, 7, 13, 15, 20, 24, 26, 33]

[52] B. T. Polyak. The conjugate gradient method in extremal problems. *USSR Comput. Math. Math. Phys.* **9** (1969), 94–112. [3]

[53] Marcos Raydan. The Barzilai and Borwein gradient method for the large scale unconstrained minimization problem. *SIAM J. Optim.* **7** (1997), 26–33. [4]

[54] Thomas Serafini, Gaetano Zanghirati, and Luca Zanni. Gradient projection methods for quadratic programs and applications in training support vector machines. *Optim. Methods Softw.* **20** (2005), 353–378. [4]

[55] Philippe L. Toint. An assessment of nonmonotone linesearch techniques for unconstrained optimization. *SIAM J. Sci. Comput.* **17** (1996), 725–739. [4]

[56] P. Wolfe. Convergence conditions for ascent methods. *SIAM Rev.* **11** (1969), 226–235. [5]

[57] Eugene K. Yang and Jon W. Tolle. A class of methods for solving large, convex quadratic programs subject to box constraints. *Math. Program.* **51**, 223–228. [4]

[58] Hongchao Zhang and William W. Hager. A nonmonotone line search technique and its application to unconstrained optimization. *SIAM J. Optim.* **14** (2004), 1043–1056. [4]

[59] Jianzhong Zhang and Chengxian Xu. A class of indefinite dogleg path methods for unconstrained minimization. *SIAM J. Optim.* **9** (1999), 646–667. [4]

[60] C. Zhu, R.H. Byrd, P. Liu, and J. Nocedal. L-BFGS-B: Algorithm 778: L-BFGS-B, fortran routines for large scale bound constrained optimization. *ACM Trans. Math. Softw.* **23** (1997), 550–560. [37]