LMBOPT – supplementary Material

1 Algorithms and data structures

LMBOPT solves a bound constrained optimization problem with a continuously differentiable objective function, using routines for evaluating the function and the gradient. It uses beyond the theory in [12], a new limited memory quasi Newton method and the robust curved line search method. It is followed as follows:

Step	dependencies
LMBOPT	**Preprocessor**, **Determiner**, **Updater**, **Postprocessor**
Preprocessor	**Initializer**, **ImproverPoint**, **ProblemObject**
Determiner	**ReducerGrad**, **WorkerSelector**, **Successor**, **Unsuccessor**
Updater	**Worker**, **UpdaterInfo**, **Subspace**
Successor	**SubspaceSelector**, **Director**, **ProblemObject**
ProblemObject	**GeneratorFun**, **AdjusterGrad**
Director	**LocalSolvers**, **Conjugator**, **GeneratorCauchy**
Conjugator	**RobustifierI**, **GeneratorGamma**, **Regularizer** **GeneratorDirection**
Unsuccessor	**Enforcer**, **GeneratorCurve**, **Nullifier**
Nullifier	**Neighbourhood**, **ProblemObject**
GeneratorCurve	**RobustifierI**, **CurveSearch**, **RobustifierII** **ProblemObject**

Table 1: Mathematical structure of **LMBOPT**

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

Preprocessor uses **Initializer** initializing the subspace and other necessary information, then calls **ImproverPoint** improving the starting point, and calls **ProblemObject** computing and adjusting the function value and the gradient vector.

Determiner includes **ReducerGrad** computing the reduced gradient, **WorkerSelector** changing or keeping the free index set $I_-(x)$, **Successor** containing the successful iterations and **Unsuccessor** containing the unsuccessful iterations.

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

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

Director calls LocalSolvers to compute the search direction such as a new limited memory quasi Newton and then uses **Conjugator** generating the conjugate gradient direction. Whenever the activity is fixed, **GeneratorCauchy** is used to compute a scaled Cauchy point.

ProblemObject calls possibly many times **GeneratorFun** to compute the function value in each iteration and only once in each iteration to compute the gradient vector. Afterwards, it calls **AdjusterGrad** to adjust the gradient vector.

Conjugator contains **RobustifierI** finding a good starting step size, **GeneratorGamma** calculating γ – one entry of the Hessian approximation, **Regularizer** doing a regularization for numerical stability, and a conjugate gradient direction.

UnSuccessor tries to enforce the angle condition by **Enforcer**, then calls a robust bent line search method to update the best point, and uses **Nullifier** avoiding too many null steps.

Nullifier calls **Neighbourhood** to generate a point around the current (previous) best point and then **ProblemObject** to compute and adjust the function value and gradient vector.

GeneratorCurve calls **RobustifierI** to find a good step size and performs a bent line search along a regularized direction. Afterwards, it calls **RobustifierII** to obtain the robust step size and then computes and adjusts the function value and the gradient vector.

Initializer	**initInfo**
ImproverPoint	**projStartPoint**
Determiner	**getSuccess**
WorkingSelector	**findFreePos**
Worker	**findFreeNeg**
UpdaterInfo	**updateInfo**
Subspace	**updateSubspace**
SubspaceSelector	**typeSubspace**
LocalSolvers	**scaleDir**, **quasiNewtonDir**, **AvoidZigzagDir**
GeneratorFun	**fun**, **dfun**
AdjusterGrad	**adjustGrad**
ReducerGrad	**redGrad**
RobustifierI	**goodStep**
GeneratorGamma	**getGam**
Regularizer	**regDenom**
GeneratorDirection	**ConjGradDir**

GeneratorCauchy	**scaleCauchy**
Enforcer	**enforceAngle**
Nullifier	**nullStep**
CurveSearch	**CLS**
RobustifierII	**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`, `info`, and `st` according to the input/output list indicated. These data structures themselves are briefly described in Table 4.

function [`step`] = **goodStep**(`point`, `step`, `tune`);
Try to find the starting good step size
function [`point`, `step`] = **robustStep**(`point`, `step`, `tune`);
Try to find a point with smallest robust change
function [`point`, `step`, `info`] = **CLS**(**fun**, `point`, `step`, `par`, `tune`, `info`);
Find a step size α satisfying a sufficient descent condition
function [`point`, `step`, `par`, `info`] = **nullStep**(**fun**, `point`, `step`, `par`, `tune`, `info`);
Try to prevent producing the null steps
function [`point`] = **adjustGrad**(`point`, `tune`);
Adjust the gradient vector
function [`point`] = **redGrad**(`point`);
Compute the reduced gradient
function [`point`, `par`, `info`] = **findFreePos**(`point`, `par`, `info`);
Update the working set
function [`point`, `par`, `info`] = **findFreeNeg**(`point`, `par`, `tune`, `info`);
Find the free index set
function [`point`] = **updateSubspace**(`point`, `step`, `par`, `tune`);
Update the subspace information
function [`step`] = **enforceAngle**(`point`, `step`, `par`, `tune`, `info`);
Enforce the angle condition
function [`point`, `step`, `par`] = **quasiNewtonDir**(`point`, `step`, `par`, `tune`);
Compute quasi Newton direction
function [`point`, `par`] = **typeSubspace**(`tune`, `par`, `tune`, `info`);
Determine the type of subspace
function [`step`, `par`] = **scaleDir**(`point`, `step`, `par`);
Choose components of sensible sign and scale
function [`step`] = **AvoidZigzagDir**(`point`, `step`, `par`, `tune`);
Modify the direction to avoid zigzagging
function [`point`, `step`, `par`] = **searchDir**(`point`, `step`, `par`, `tune`, `info`);

Construct starting trial search direction
function [`point`, `step`, `par`, `info`] = **getGam**(**fun**, `point`, `step`, `tune`, `par`, `info`);
Compute γ
function [`par`] = **regDenom**(`point`, `step`, `par`, `tune`);
Construct regularize denominator
function [`point`, `step`, `par`, `info`]= **ConjGradDir**(**fun**, `point`, `step`, `par`, `tune`, `info`);
Construct the conjugate gradient direction
function [`point`, `step`, `par`]= **scaleCauchy**(`point`, `step`, `par`, `tune`, `info`);
Construct a scaled Cauchy point
function [`point`] = **projStartPoint**(`point`, `tune`);
Improve the starting point
function [`point`, `step`, `par`, `info`] = **getSuccess**(**fun**, `point`, `step`, `par`, `tune`, `info`);
Determine whether subspace iteration is successful or not
function [`point`] = **initInfo**(`point`, `tune`);
Initialize best point and factor for adjusting acceptable increase in f
function [`point`, `par`] = **updateInfo**(`point`, `par`, `tune`, `info`);
Update best point and factor for adjusting acceptable increase in f
function [x, f, `info`] = **LMBOPT**(**fun**, x, $\underline{x}$, $\overline{x}$, `tune`, `st`);
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_{new}, f_{new}, g_{new} (newest point, its function value and gradient vector) x_{best}, f_{best} (best point and its function value) x_{init}, f_{init} (starting point and its function value) $\underline{x}$, $\overline{x}$ (lower and upper bound) y (the difference of current gradient with its old one; $g_{\text{new}} - g$) 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_{init} (starting search direction in each iteration), p (Krylov search direction), `gp` ($g^T p$) α_{good} (the starting step-size generated by **goodStep**), s (search direction; $x_{\text{new}} - x$)
`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), Δ_{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 **CLS**)
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`)
$\zeta_{\min}$ and $\zeta_{\max}$ (Safeguarded parameters for ζ in **ConjGradDir**)
`nstuckmax` (iteration limit in number of stuck)
Δ_D (parameter for controlling entries of scaling diagonal matrix D)
`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))
`sub` (point generated by the subspace changes the activity?)
`probcase` (problem is bound constrained?)
`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 **CLS**), `nstuck` (number of stuck iterations)
`st` (= initial `info`) (structure with stop and print criteria)
`nf` (number of function evaluations), `ng` (number of gradient evaluations)
`nfmax` (maximal number of function evaluations)
`ngmax` (number of gradient evaluations), `nf2gmax` (`nfmax` + 2`ngmax`)
`prt` (print level: -1: nothing, 0: little, ≥ 1: more and more)

Table 4: Global data structures for the algorithms of the present paper

2 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 [8]. 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 Byrd et al. [4]. 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. [5]. 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. [1, 2]. 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 [6, 7, 9, 10]. Only `memory` = 12 and other parameters have been chosen as default.
- **LMBFG**, obtained from
  `http://gratton.perso.enseeiht.fr/LBFGS/index.html`,
  is a limited memory quasi Newton package by Burdakov et al. [3]:

  (a) **LMBFG-MT** (**ll1**) is a limited memory line-search algorithm **L-BFGS** based on the More-Thuente line search. Only $m = 12$ and other parameters have been chosen as default.

(b) **LMBFG-MTBT** (**ll2**) 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. [3]. Only $m = 12$ and other parameters have been chosen as default.

(c) **LMBFGS-TR** (**ll3**), is a limited memory line-search algorithm **L-BFGS** that takes a trial step along the quasi-Newton direction inside the trust region. Only $m = 12$ and the other parameters have been chosen as default.

(d) **LMBFG-BWX-MS** (**lt1**) is a limited memory trust-region algorithm BWX-MS. 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.

(e) **LMBFG-DDOGL** (**lt2**) is a limited memory trust-region algorithm **D-DOGL**. Only $m = 12$ and the other parameters have been chosen as default.

(f) **LMBFG-EIG-curve-inf** (**lt4**) is a limited memory trust-region algorithm $\texttt{EIG}(\infty, 2)$. Only $m = 12$ and the other parameters have been chosen as default.

(g) **LMBFG-EIG-inf-2** (**lt5**) 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. Only $m = 12$ and other parameters have been chosen as default.

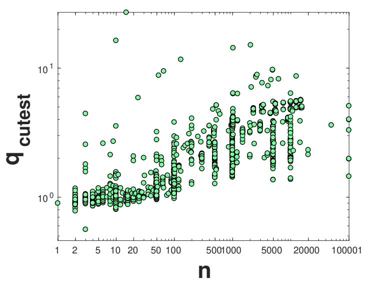
(h) **LMBFG-EIG-MS** (**lt6**) is a limited memory trust-region algorithm **EIG-MS**. Only $m = 12$ and other parameters have been chosen as default.

(i) **LMBFG-EIG-MS-2-2** (**ll7**) 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. Only $m = 12$ and other parameters have been chosen as default.

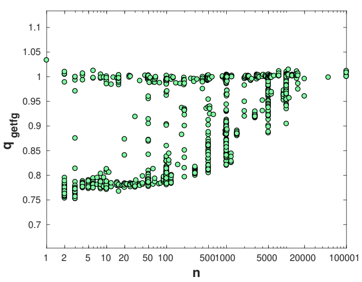
3 Why `nf2g` is reasonable for the performance profile?

In [11], `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.

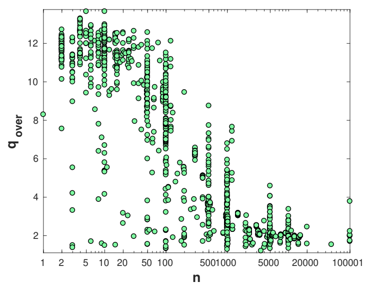
Subfigures (a) and (b) of Figure 1 show that the time for computing the gradient by `cutest_obj` and `getfg` are more than that of the function value, respectively. Hence, `nf2g` is a reasonable cost measure for the performance profile. In addition, Subfigure (c) of Figure 1 shows that `getfg` is expansive than `cutest_obj` due to **collect statistics and enforce stopping tests**.



a)



b)



c)

Figure 1: 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 $q_{\texttt{over}} := \frac{t_{f2g}(\texttt{getfg})}{t_{f2g}(\texttt{cutest})}$ versus dimensions, respectively, where t_f and t_g are considered the time to compute f and g by `cutest` or `getfg` and $t_{f2g} := t_f + 2t_g$.

4 Problems unsolved by all solvers

A list of problems unsolved by all solvers is given in Table 5.

Table 5: Problems unsolved by all solvers

BROWNBS	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	SSCOSINE: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
LINVERSE:1999	RAYBENDL:2050	RAYBENDS: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	SCOSINE:10000	SCURLY10:10000	SPARSINE:10000
SPMSRTLS:10000	SSCOSINE:10000	DRCAV3LQ:10816	ODNAMUR
GRIDGENA:12482	SSCOSINE:100000		

5 Test problem selection

It is seen from Figure 2 that the number of unconstrainrd, bound constrained, and unconstrained and bound constrained optimization problems – solved at least by one of solvers – are 517, 375, and 990 respectively.

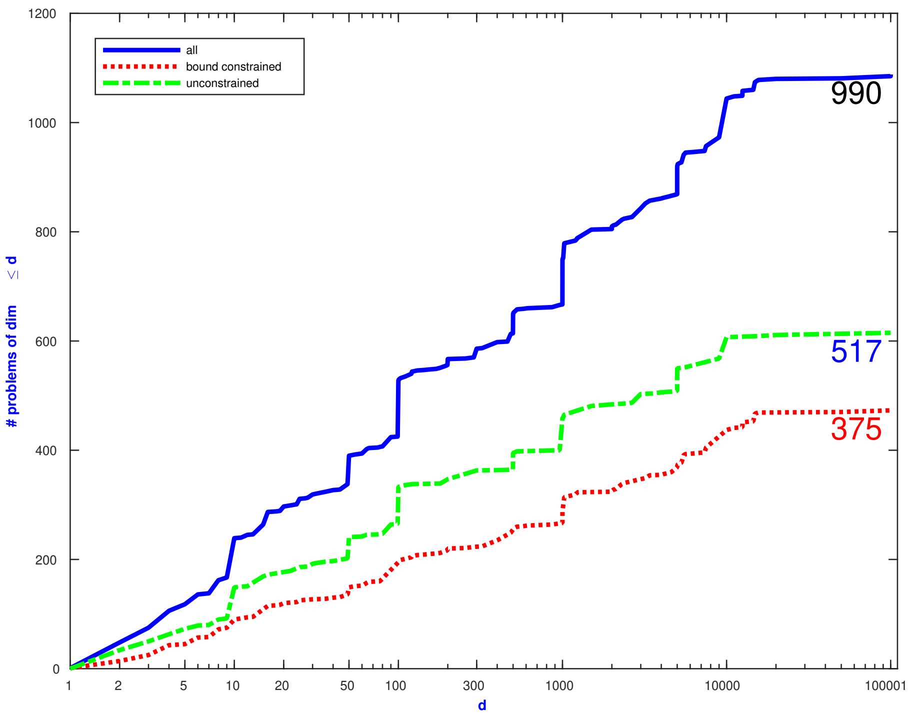


Figure 2: The number of problems with variables in a given range solved by at least one solver: 990 problems with dimensions 1 up to 100001

References

[1] E. G. Birgin, J. M. Martínez, and M. Raydan. Nonmonotone spectral projected gradient methods on convex sets. *SIAM J. Optim.* **10** (1999), 1196–1211.

[2] E. G. Birgin, J. M. Martínez, and M. Raydan. Algorithm 813: Spg-software for convex-constrained optimization. *ACM Trans. Math. Softw.* **27** (2001), 340–349.

[3] O. Burdakov, L. Gong, S. Zikrin, and Y. Yuan. On efficiently combining limited-memory and trust-region techniques. *Math. Program. Comput.* **9** (2017), 101–134.

[4] 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.

[5] 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.

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

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

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

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

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

[11] 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).

[12] A. Neumaier and B. Azmi. Line search and convergence in bound-constrained optimization. Technical report, University of Vienna (2019).