pySLEQP: A Sequential Linear Quadratic Programming Method Implemented in Python

Felix Lenders, Christian Kirches, and Hans Georg Bock

Abstract We present a prototype implementation of a Sequential Linear Equality-Constrained Qudratic Programming (SLEQP) method for solving the nonlinear programming problem. Similar to SQP active set methods, SLEQP methods are iterative Newton-type methods. In every iteration, a trust region constrained linear programming problem is solved to estimate the active set. Subsequently, a trust region equality constrained quadratic programming problem is solved to obtain a step that promotes locally superlinear convergence. This class of methods has several appealing properties for future research in large-scale nonlinear programming. Implementations of SLEQP methods accessible for research, however, are scarcely found. To this end, we present *pySLEQP*, an implementation of an SLEQP method in Python. The performance and robustness of the method and our implementation are assessed using the CUTEst and CUTEr benchmark collections of nonlinear programming problems. *pySLEQP* is found to show robust behavior and reasonable performance.

Felix Lenders
Interdisciplinary Center for Scientific Computing (IWR), Heidelberg University. Im Neuenheimer Feld 368, 69120 Heidelberg, GERMANY. e-mail: felix.lenders@iwr.uni-heidelberg.de

Christian Kirches
Institut für Mathematische Optimierung, Technische Universität Carolo-Wilhelmina zu Braunschweig. Am Fallersleber Tore 1, 38100 Braunschweig, GERMANY e-mail: c.kirches@tu-bs.de *and* Interdisciplinary Center for Scientific Computing (IWR), Heidelberg University. Im Neuenheimer Feld 368, 69120 Heidelberg, GERMANY. e-mail: christian.kirches@iwr.uni-heidelberg.de

Hans Georg Bock
Interdisciplinary Center for Scientific Computing (IWR), Heidelberg University. Im Neuenheimer Feld 368, 69120 Heidelberg, GERMANY. e-mail: bock@iwr.uni-heidelberg.de

1 Introduction

In this article, we take interest in computing local minima of the nonlinear programming problem

$$\begin{cases} \min\limits_{x\in\mathbb{R}^n} f(x) & \text{s.t.} \quad c_i(x) = 0, \quad i \in \mathscr{E} \\ & \qquad\ \ c_i(x) \leq 0, \quad i \in \mathscr{I} \end{cases}, \tag{NLP}$$

where $f : \mathbb{R}^n \to \mathbb{R}$ and $c : \mathbb{R}^n \to \mathbb{R}^{|\mathscr{I}\cup\mathscr{E}|}$ are $\mathscr{C}^2$ functions, and $\mathscr{I}$ and $\mathscr{E}$ denote disjoint finite index sets, using sequential linear equality-constrained quadratic programming (SLEQP) methods, a class of Newton-type active-set methods first proposed by [10, 11].

Nonlinear programming has applications in many areas of mathematical optimization, e.g. as an important class of subproblems in mixed-integer nonlinear optimization [2] or in direct methods for optimal control [3, 19]. Beyond this, many applied research domains make use of nonlinear optimization techniques to study real-world problems in, e.g., process and control engineering.

Different from the more widely known class of sequential quadratic programming (SQP) methods, SLEQP methods successively solve linear models of an exact penalty function to estimate the active set, and promote locally superlinear convergence by solving a quadratic model on that active set. Variants of this approach have been described in, e.g., [4, 5, 7]. With *Knitro* [6], a commercial, closed-source solver exists. SLEQP methods have particular appeal for future large-scale nonlinear programming efforts, as a) linear programming technology is highly mature, e.g. [16], and still significantly more powerful than QP technology; b) the linear program utilized for active set estimation can be replaced by any oracle that provides an active set guess, e.g. [21]; c) KKT systems can be solved iteratively [14]; d) the costly-to-compute Hessian of the Lagrangian is never explicitly required.

Contributions. Open implementations of SLEQP methods that are accessible to research are scarcely found. Addressing this gap, we present a prototypical Python implementation, named *pySLEQP*, of an SLEQP method for solving the nonlinear programming problem. We assess its performance on the established benchmark libraries CUTEst and CUTEr and find it to be most robust in the sense that it solves the largest fraction of problems. Performance is found to be acceptable for an interpreted language. The source code of *pySLEQP* and the data of the numerical studies presented §4 are available at [20].

Structure. The remainder of this article is laid out as follows. In §2 we introduce nonlinear programming terminology as required, and give a concise description of the family of sequential linear equality-constrained quadratic programming methods. *pySLEQP*, a new prototype implementation of an SLEQP method in the interpreted language Python is presented in §3. The performance of this implementation is evaluated in §4, using the well-established CUTEst and CUTEr benchmark collections of instances of nonlinear programming problems. The article concludes with a brief summary and an outlook on future research topics in §5.

2 Sequential Linear Equality Constrained Quadratic Programming

We start by recalling a few basic concepts from nonlinear programming before presenting this class of SLEQP methods in greater detail. We denote the Lagrange function of (NLP) by $L : \mathbb{R}^n \times \mathbb{R}^{|\mathscr{I} \cup \mathscr{E}|} \to \mathbb{R}$ with $L(x, \lambda) := f(x) + \langle \lambda, c(x) \rangle$. Assuming a suitable constraint qualification, e.g. MFCQ [23], existence of a KKT tuple is a necessary condition for a local minimizer of (NLP).

Theorem 1. *Let x^* be a local minimizer for* (NLP) *and let MFCQ hold, i.e., the vectors $(\nabla c_i(x^*))_{i \in \mathscr{E}}$ are l.i. and there is $d \in \mathbb{R}^n$ such that $\langle \nabla c_i(x^*), d \rangle = 0$ for all $i \in \mathscr{E}$ and $\langle \nabla c_i(x^*), d \rangle < 0$ for all $i \in \mathscr{I}$ with $c_i(x^*) = 0$. Then there is $\lambda^* \in \mathbb{R}^{|\mathscr{I} \cup \mathscr{E}|}$ satisfying*

$$\nabla_x L(x^*, \lambda^*) = 0, \quad \lambda^*_{\mathscr{I}} \geq 0, \quad \langle \lambda^*_{\mathscr{I}}, c_{\mathscr{I}}(x^*) \rangle = 0. \tag{1}$$

Next, a sufficient condition for a KKT tuple (x^*, λ^*) to be a local minimum of (NLP) is given by the following theorem.

Theorem 2. *Let (x^*, λ^*) satisfy Thm. 1 and let $\langle d, \nabla_{xx} L(x^*, \lambda^*) d \rangle \geq 0$ for all vectors $d \in \mathbb{R}^n$ such that $\langle \nabla c_i(x^*), d \rangle = 0$ for all $i \in \mathscr{E} \cup \mathscr{I}$ with $c_i(x^*) = 0$. Then (x^*, λ^*) is a local minimum of* (NLP).

Proofs of both theorems can be found in, e.g., [25, Ch. 12]. SLEQP methods are active set methods that use an estimate of the working set $\mathscr{W}$. A step is computed by minimizing a suitable quadratic model of the problem taking into account only those constraints predicted by the working set $\mathscr{W}$, which is estimated by minimizing a linear model of an exact penalty function. A summary of the computational steps required for one iteration of an SLEQP method is presented in Algorithm 1. In the following, we elaborate on the four major components.

Estimation of the Active Set

We use an ℓ_1-penalty function approach that minimizes the exact penalty function

$$\phi_\nu(x) := f(x) + \nu \sum_{i \in \mathscr{E}} |c_i(x)| + \nu \sum_{i \in \mathscr{I}} [c_i(x)]^+, \tag{2}$$

for (NLP) due to [17]; $[\,\cdot\,]^+$ denotes clamping to the nonnegative. For a linearization point $\bar{x} \in \mathbb{R}^n$, we obtain an active set estimate by minimizing the linearization

$$\ell_\nu(\bar{x}, d) := \langle \nabla f(\bar{x}), d \rangle + \nu \sum_{i \in \mathscr{E}} |c_i(\bar{x}) + \langle \nabla c_i(\bar{x}), d \rangle| + \nu \sum_{i \in \mathscr{I}} [c_i(\bar{x}) + \langle \nabla c_i(\bar{x}), d \rangle]^+$$

of (2) subject to an ℓ_∞ trust region of size Δ_{LP}. This problem reads

$$\min_{d \in \mathbb{R}^n} \ell_\nu(\bar{x}, d) \text{ s.t. } \|d\|_\infty \leq \Delta_{\mathrm{LP}}, \tag{LP($\nu, \Delta_{\mathrm{LP}}$)}$$

and a true linear reformulation, easily obtained by appropriate introduction of slack variables, can be solved using highly mature simplex technology, e.g. [16].

Given a point x, a working set $\mathscr{W}(x)$ at x is a maximal subset of the active set $\{i \in \mathscr{E} \cup \mathscr{I} \,|\, c_i(x) = 0\}$ such that $(\nabla c_i(x))_{i\in\mathscr{W}}$ is linearly independent. Besides the step d_{LP}, the solution of (LP$(\nu, \Delta_{\mathrm{LP}})$) provides an estimation of $\mathscr{W} = \mathscr{W}(\bar{x} + d_{\mathrm{LP}})$ using simplex basis information, where active trust region bounds are omitted.

Cauchy Step and Estimation of Lagrange Multipliers

We form a quadratic model of the penalty function (2),

$$q(\bar{x}, d) := \ell(\bar{x}, d) + \tfrac{1}{2}\langle d, \nabla_{xx} L(\bar{x}, \lambda_{\mathrm{LS}}) d\rangle, \tag{3}$$

and define the Cauchy step d_{C} to be the minimizer of $q(\bar{x}, \cdot)$ along the segment $d \in [0, d_{\mathrm{LP}}]$. Convergence of SLEQP can be shown, e.g. [5], if steps are taken that make progress at least as good as d_{C}. In practice, we compute an approximation to d_{C} via a backtracking line search of Armijo type [25] along $[0, d_{\mathrm{LP}}]$.

The Lagrange multipliers λ_{LS} required in (3) are obtained by computing a minimum residual solution of the KKT stationarity condition

$$\begin{bmatrix} I & (\nabla c_i(x))_{i\in\mathscr{W}}^{\mathsf{T}} \\ (\nabla c_i(x))_{i\in\mathscr{W}} & 0 \end{bmatrix} \begin{pmatrix} * \\ -\hat{\lambda}_{\mathrm{LS}} \end{pmatrix} = \begin{pmatrix} -\nabla f(x) \\ 0 \end{pmatrix}. \tag{4}$$

The component * denotes the residual vector of the least squares estimation, and is not needed. The obtained multipliers are projected onto the feasible set of multipliers, $(\lambda_{\mathrm{LS}})_{\mathscr{I}} := [(\hat{\lambda}_{\mathrm{LS}})_{\mathscr{I}}]^+$, $(\lambda_{\mathrm{LS}})_{\mathscr{E}} := (\hat{\lambda}_{\mathrm{LS}})_{\mathscr{E}}$, $(\lambda_{\mathrm{LS}})_i = 0$ for $i \notin \mathscr{W}$. While we could have chosen the multipliers λ_{LP} (or λ_{EQP} below) provided by the solution of (LP$(\nu, \Delta_{\mathrm{LP}})$) or (EQP$(\nu, \Delta_{\mathrm{EQP}})$), the choice (4) provides us with multipliers satisfying KKT stationarity best. This is possible at negligible cost as the matrix in the linear system to compute the multiplier has to be factorized in the EQP step as well and the additional cost of computing $\hat{\lambda}_{\mathrm{LS}}$ is only one solve.

Computation of a Newton-Type Step

We denote by $\mathscr{V} := \{i \in \mathscr{I} \setminus \mathscr{W} \,|\, c_i(\bar{x}) > 0\} \cup \{i \in \mathscr{E} \setminus \mathscr{W} \,|\, c_i(\bar{x}) \neq 0\}$ the set of violated inequalities not covered by the working set and associate with it the penalty function $m_\nu(x) := f(x) + \nu \sum_{i\in\mathscr{V}} |c_i(x)|$. Since $c_i(\bar{x}) \neq 0$ for $i \in \mathscr{V}$ this is a $\mathscr{C}^2$ function in a neighbourhood of $\bar{x}$ by continuity of c. m_ν penalizes violated constraints that will not be enforced as equality in the EQP. A Newton-type step d_{EQP} is computed by minimizing a quadratic model $q_\nu(\bar{x}, d)$ of the penalty function m_ν around $\bar{x}$,

$$q_\nu(\bar{x}, d) = \langle \nabla_x m_\nu(\bar{x}), d\rangle + \tfrac{1}{2}\langle d, \nabla_{xx} m_\nu(\bar{x}) d\rangle, \tag{5}$$

subject to linearized constraints estimated by $\mathscr{W}$ and an ℓ_2 trust region of size Δ_{EQP},

$$\begin{cases} \min\limits_{d\in\mathbb{R}^n} \; q_\nu(\bar{x},d) \quad \text{s.t.} \quad c_i(\bar{x}) + \langle \nabla c_i(\bar{x}), d\rangle = 0, \quad i \in \mathscr{W}, \\ \qquad\qquad\qquad\qquad \|d\|_2 \leq \Delta_{\text{EQP}}. \end{cases} \tag{EQP(ν, Δ_{EQP})}$$

The EQP can be solved directly, by projecting the trust region onto the null space of the linear constraints. We have preferred an iterative solution and make use of the projected conjugate gradient method GLTR that respects the trust region constraint or finds a suitable point on the trust region boundary, see [14]. For large-scale problems or expensive constraint functions c as arising in optimization with differential equations, this is appealing: the Hessian is only accessed by evaluations of the linear mapping $d \mapsto \nabla_{xx} L(\bar{x}, \lambda) d$.

Algorithm 1 Sequential Linear Equality Constrained Quadratic Programming

Require: Initialization $k \leftarrow 0$, x^0, Δ^0_{LP}, Δ^0_{EQP}, ν^0, ρ_{acc}
1: **while** termination criterion not satisfied **do**
2: $\quad \nu^k \leftarrow$ Adjust Penalty Parameter
3: $\quad d^k_{\text{LP}}, \mathscr{W}^k \leftarrow$ Solution of Active-Set-LP with trust region Δ^k_{LP}
4: $\quad \lambda^k \leftarrow$ Estimate Multiplier using Minimum Norm Estimation
5: $\quad d^k_{\text{C}} \leftarrow$ Compute Cauchy-Step
6: $\quad d^k_{\text{EQP}} \leftarrow$ Solution of EQP with trust region Δ^k_{EQP}, Working Set $\mathscr{W}^k$
7: $\quad x^k_{\text{try}} \leftarrow x^k + d^k_{\text{C}} + d^k_{\text{EQP}}$ trial step
8: $\quad \rho^k \leftarrow$ Ratio Actual vs. Predicted Reduction
9: $\quad$ **if** $\rho^k \geq \rho_{\text{acc}}$ **then**
10: $\quad\quad x^{k+1} \leftarrow x^k_{\text{try}}$
11: $\quad$ **else**
12: $\quad\quad d_{\text{SOC}} \leftarrow$ compute second order correction via (6)
13: $\quad\quad x^k_{\text{try}} \leftarrow x^k_{\text{try}} + d^k_{\text{SOC}}$ second order correction
14: $\quad\quad \rho^k \leftarrow$ Ratio Actual vs. Predicted Reduction
15: $\quad\quad$ **if** $\rho^k \geq \rho_{\text{acc}}$ **then** $x^{k+1} \leftarrow x^k_{\text{try}}$ **else** $x^{k+1} \leftarrow x^k$
16: $\quad$ **end if**
17: $\quad \Delta^{k+1}_{\text{LP}}, \Delta^{k+1}_{\text{EQP}} \leftarrow$ Adjust Trust Region Radii
18: $\quad k \leftarrow k+1$
19: **end while**

Miscellaneous Topics

If the attempted trial step fails (lines 7-9 in Algorithm 1), we employ a second order correction step d_{SOC} that is obtained as minimum norm solution of the constraints in the working set $\mathscr{W}$ at the trial point $x = x_{\text{try}}$:

$$\begin{bmatrix} I & (\nabla c_i(x))^{\intercal}_{i\in\mathscr{W}} \\ (\nabla c_i(x))_{i\in\mathscr{W}} & 0 \end{bmatrix} \begin{pmatrix} d_{\text{SOC}} \\ * \end{pmatrix} = \begin{pmatrix} 0 \\ (c_i(x_{\text{SOC}})_{i\in\mathscr{W}} \end{pmatrix}. \tag{6}$$

Again, the component * of the solution vector is not needed. We use MA57 [8] to factorize the matrix needed to determine the least-squares estimate of the multiplier, in the second order correction, and in the projected conjugate gradient method.

Details of the heuristics for choosing the penalties ν^k in line 2 and the radii Δ^k_{LP} and Δ^k_{EQP} in line 14 have been implemented as described in [5].

3 Prototype Implementation in Python

The algorithm *pySLEQP* has been implemented using the Python scripting language. Similar to Matlab, Python is an interpreted language that provides fast methods to work on numerical data with the NumPy and SciPy packages [18]. Via the Cython package [1], C, C++, and Fortran code can be used directly from Python. Thus, rapid prototyping is possible while time critical components of the algorithm can be implemented in a compiled language.

We use the dual simplex method of GuRoBi 6.0 [16] to solve ($\mathrm{LP}(\nu, \Delta_{\mathrm{LP}})$) and GLTR [14] to solve the ($\mathrm{EQP}(\nu, \Delta_{\mathrm{EQP}})$). The implementation has been realized as a Python module `pysleqp` that provides a class `SLEQP` holding one instance of a NLP. We allow slightly more general formulations for the NLP, while the class of problems remains the same:

$$\begin{cases} \min\limits_{x\in\mathbb{R}^n} f(x) & \text{s.t.} \quad x_i^{\mathrm{l}} \leq \quad x_i \quad \leq x_i^{\mathrm{u}}, \quad i = 1,\dots,n, \\ & \qquad\quad c_i^{\mathrm{l}} \leq c_i(x) \leq c_i^{\mathrm{u}}, \quad i = 1,\dots,m. \end{cases}$$

To instantiate such a class, the user must provide the following data:

- `n`, `m`: Numbers of variables and constraints;
- `firstorder`: Python callback function with input argument x and output arguments ($f(x)$, $\nabla f(x)$, $c(x)$, $J_c(x)$). Here, $x, \nabla f(x), c(x)$ are NumPy arrays, $f(x)$ is a Python double scalar, and $J_c(x)$ is a SciPy sparse matrix;
- `hessprod`: Python callback function with input arguments (x, σ, λ, d) and output argument $h = \sigma\langle\nabla_{xx} f(x), d\rangle + \sum_{i=1}^{m} \lambda_i \langle\nabla_{xx} c_i(x), d\rangle$. Here x, λ, d, h are NumPy arrays, and σ is a python scalar that currently will always be 1.0;
- `hessprodre`: Python callback function with the same signature as `hessprod`. This function will be called if (x, λ, d) coincide with the arguments of the most recent call to compute a Hessian-vector product, and only the direction d differs;
- `x`: NumPy array with initial solution guess;
- `xl`, `xu`: NumPy array with variable bounds $x^{\mathrm{l}} \leq x \leq x^{\mathrm{u}}$;
- `cl`, `cu`: NumPy array with constraint function bounds $c^{\mathrm{l}} \leq c(x) \leq c^{\mathrm{u}}$.

After instantiation, the method `optimize` starts the optimization loop. After termination, the class variable `terminate` contains a python dictionary with all relevant solution information and detailed timings. The class variable `x` contains the solution point in case of a successful termination, or the last point that has been considered in the algorithm in case of non-convergence.

4 Performance of *pySLEQP* on a Benchmark Collection

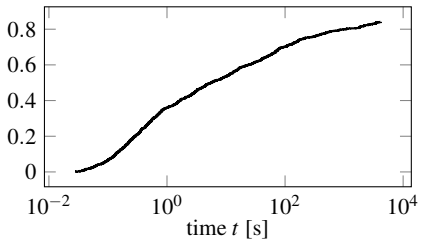
In this section, we use CUTEr and its successor CUTEst [15] as benchmark collections to assess the performance of the implementation. CUTEst is an up-to-date selection of 1149 nonlinear programming problem instances arising from various

fields of optimization and including instance with up to 250,000 variables and constraints.

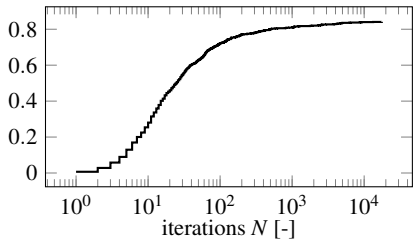
Performance on CUTEst

We have omitted 40 instances from CUTEst for which evaluations fail due to, e.g., starting points for which functions are not well-defined, and the remaining benchmark set then consists of 1109 instances. Computations were run on an Intel(R) Core(TM) i7 920 at 2.67 GHz and 6 GB RAM running Ubuntu Linux 14.04 LTS, using one core per solver. *pySLEQP* solved 85% of the benchmark set within a wall time limit of one hour per instance, i.e $\rho(3600) = 0.85$. Fig. 1a shows the ratio $\rho(t)$ of instances that could be solved within a wall time limit of t. Fig. 1b shows the ratio $\rho(N)$ of instances that could be solved within an iteration limit of N. Fig. 1c shows the ratio $\rho(N)$ of instances that could be solved within a limit of N on the number of Hessian-vector products.

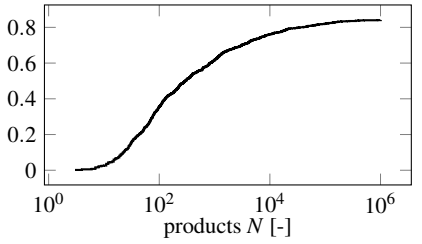
Table 1d gives a breakdown of the relative computational cost of the steps of Algorithm 1, excluding function evaluations. As can be seen, solving $(\mathrm{LP}(v, \Delta_{\mathrm{LP}}))$ to obtain an active set estimate and solving $(\mathrm{EQP}(v, \Delta_{\mathrm{EQP}}))$ to obtain a Newton-type



(a) Performance in terms of CPU time t.



(b) Performance in terms of # N of iterations.



(c) Performance in terms of # N of matrix-vector products with the Hessian $\nabla_{xx}L(x,\lambda)$.

Lines	of Algorithm 1	▼ mean %	var. %
3	Active Set Determination	55.5	3.4
6	EQP Setup	12.6	1.8
6	Line search d_{EQP}	11.4	0.7
6	EQP Solution	11.3	4.7
4, 6, 12	Factorization pCG	2.9	0.1
2, 1	Penalty, Term. Test	2.6	0
5	Line search d_{C}	2.0	0.1
12	Second Order Corr.	1.7	0.1

(d) Distribution of CPU time spent inside *pySLEQP*, excluding function evaluations.

Fig. 1: Ratio ρ of problems of the CUTEst benchmark collection solved by the SLEQP implementation *pySLEQP* within (a): at most t seconds, (b): at most N iterations, (c): at most N products with of the Hessian of the Lagrangian $\nabla_{xx}L(x,\lambda)$.

step dominate the computational effort. Significant amounts of interpreted Python code are executed during EQP setup, trust region ratio computation, penalty function evaluation, and in the termination test. Here, one may expect speed-ups after reimplementation in a compiled language.

Comparison on CUTEr

Not all the solvers we are using to compare our implementation directly support CUTEst. Hence, we have chosen to use the AMPL translation of CUTEr for that purpose. The advantage of using the modeling system AMPL [12] is given by the fact that it provides a unified interface to different solvers. The AMPL translation consists of a 924 instance subset of CUTEr that may slightly differ from its CUTEst counterpart in supplied start points, parameter values, and choices for variable sized problems. Like done for CUTEst, we omit all instances for which evaluations fail. We also omit instances that could be solved by any solver in less than 0.11 seconds which constitutes a test set including 183 problems. For such tiny instances, the overhead time required for starting the Python interpreter (0.11 seconds) dominates the actual solution time. Again, we imposed a wall time limit of one hour on the solution time per instance.

To compare our prototypical SLEQP implementation with the established active set solvers filterSQP [9], SNOPT [13], MINOS [24], and the active-set solver of Knitro [6], we compute an extended performance profile according to [22]: For $\mathscr{P}$ the set of problems and $\mathscr{S}$ the set of solvers, let $t_{s,p}$ denote the CPU time solver $s \in \mathscr{S}$ needs to solve instance $p \in \mathscr{P}$, and let $r_{s,p}$ denote the ratio solver $s \in \mathscr{S}$

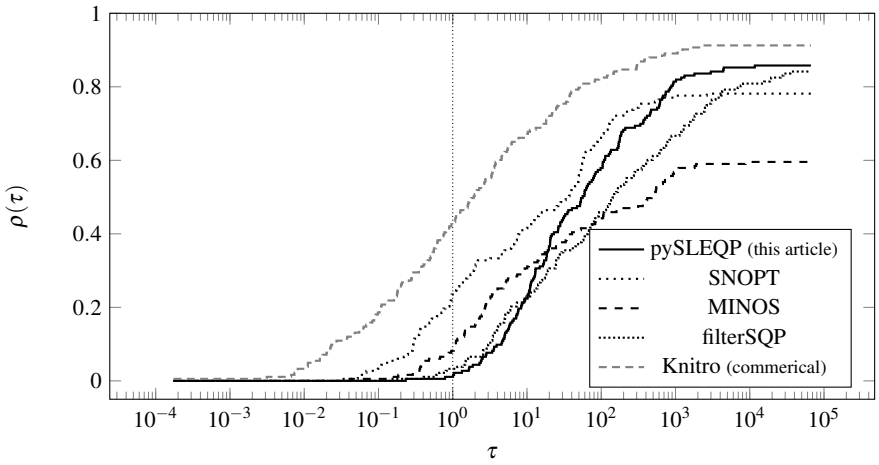


Fig. 2: Extended performance profile comparing the SLEQP implementation *pySLEQP* with state-of-the-art active set nonlinear programming solvers on CUTEr.

needs for a certain instance $p \in \mathscr{P}$ in comparison to the fastest solver. Then $\rho_s(\tau)$ denotes the fraction of problems solver s solves in at least τ times the CPU time of the fastest solver, where

$$r_{s,p} := t_{s,p} / \min\{t_{i,p} \mid i \in \mathscr{S},\ i \neq s\}, \qquad \rho_s(\tau) := |\{p \in \mathscr{P} \mid r_{s,p} \leq \tau\}| / |\mathscr{P}|. \quad (7)$$

The result for the above subset of the CUTEr benchmark collection is shown in Fig. 2. It can be seen that together with Knitro and filterSQP our implementation *pySLEQP* is among the most robust of the five solvers, in the sense that they solve the largest fraction of problems within the wall time limit. *pySLEQP* is implemented in the interpreted language Python that incurs some speed limitations, and is hence not the overall fastest solver. Still, it achieves a performance that is competitive with other solvers that have been implemented in the compiled languages Fortran/C++.

5 Summary and Conclusions

In this article, we have presented the prototypical Python implementation *pySLEQP* of an SLEQP method for solving the nonlinear programming problem. Our contribution fills a gap in the landscape of academic research codes for nonlinear programming. On the well-established CUTEr benchmark collection, *pySLEQP* has been show to deliver competitive performance and to be more robust than three popular NLP solvers examined. Hence, the implementation provides a reliable foundation for investigating future developments in high-performance nonlinear programming that will allow to treat challenging real-world problems. Future developments will have to address the trust region search and the EQP subproblem. First, problem (LP(v, Δ_{LP})) really is a parametric problem in Δ_{LP}. As already noted in [5], its solution using a parametric simplex method may allow a more elaborate choice of the radius Δ_{LP}. Second, when solving the EQP, the method requires only matrix-vector products with the Hessian $\nabla_{xx} L(x, \lambda)$. This advantage of SLEQP methods over SQP methods is particularly promising in conjunction with ODE/DAE constrained NLPs arising in optimal control, e.g. [3, 19], wherein evaluating full second derivatives may be prohibitively expensive while computing a few directional derivatives may be feasible. Finally, both the EQP and the Lagrange multiplier estimate open up the possibility of preconditioning. A conversion of the Python implementation of *pySLEQP* to a compiled language such as Fortran, C, or C++ promises to bring speedups that may help to shift the *pySLEQP* curve of Fig. 2 further to the left, increasing competitiveness with SNOPT also for smaller instances.

Acknowledgements F. L. and C. K. were supported by DFG Graduate School 220 funded by the German Excellence Initiative. Financial support by the German Federal Ministry of Education and Research, grant n$^{\mathrm{o}}$ 05M2013-GOSSIP, by the European Union within the 7$^{\mathrm{th}}$ Framework Programme under Grant Agreement n$^{\mathrm{o}}$ 611909, and by German Research Foundation within DFG project n$^{\mathrm{o}}$ BO364/19-1 is gratefully acknowledged. F. L. gratefully acknowledges funding by the German National Academic Foundation.

References

1. Behnel, S., Bradshaw, R., Citro, D., Dalcin, L., Seljebotn, D., Smith, K.: Cython: The Best of Both Worlds. Computing in Science Engineering **13**(2), 31–39 (2011).
2. Belotti, P., Kirches, C., Leyffer, S., Linderoth, J., Luedtke, J., Mahajan, A.: Mixed-Integer Nonlinear Optimization. In: Acta Numerica, vol. 22, pp. 1–131. Cambridge Univ. Press (2013)
3. Bock, H.G., Plitt, K.J.: A Multiple Shooting algorithm for direct solution of optimal control problems, In: Proceedings of the 9th IFAC World Congress, 242–247 (1984). Pergamon Press
4. Byrd, R., Gould, N., Nocedal, J., Waltz, R.: An algorithm for nonlinear optimization using linear programming and equality constrained subproblems. Math. Prog. **100**(1), 27–48 (2003)
5. Byrd, R., Gould, N., Nocedal, J., Waltz, R.: On the Convergence of Successive Linear-Quadratic Programming Algorithms. SIAM Journal on Optimization **16**(2), 471–489 (2005)
6. Byrd, R., Nocedal, J., Waltz, R.: Knitro: An integrated package for nonlinear optimization. In: G. Pillo, M. Roma (eds.) Large-Scale Nonlinear Optimization, *Nonconvex Optimization and Its Applications*, vol. 83, pp. 35–59. Springer US (2006)
7. Byrd, R., Waltz, R.: An active-set algorithm for nonlinear programming using parametric linear programming. Optimization Methods and Software **26**(1), 47–66 (2011)
8. Duff, I.: MA57 — a code for the solution of sparse symmetric definite and indefinite systems. ACM Transactions on Mathematical Software **30**(2), 118–144 (2004)
9. Fletcher, R., Leyffer, S.: Nonlinear programming without a penalty function. Mathematical Programming **91**(2), 239–269 (2002)
10. Fletcher, R., de la Maza, E.S.: Nonlinear programming and nonsmooth optimization by successive linear programming. Mathematical Programming **43**(1–3), 235–256 (1989)
11. Chin, C.M., Fletcher, R.: On the global convergence of an SLPfilter algorithm that takes EQP steps. Mathematical Programming **96**(1), 161–177 (2003)
12. Fourer, R., Gay, D., Kernighan, B.: A Modeling Language for Mathematical Programming. Management Science **36**, 519–554 (1990)
13. Gill, P., Murray, W., Saunders, M.: SNOPT: An SQP algorithm for large-scale constrained optimization. SIAM Journal of Optimization **12**, 979–1006 (2002)
14. Gould, N., Lucidi, S., Roma, M., Toint, P.: Solving the Trust-Region Subproblem using the Lanczos Method. SIAM Journal on Optimization **9**(2), 504–525 (1999)
15. Gould, N., Orban, D., Toint, P.: CUTEst: a constrained and unconstrained testing environment with safe threads. Tech. Rep. RAL-TR-2013-005 (2013)
16. GuRoBi Optimization, Inc.: GuRoBi Optimizer Version 6.0 Reference Manual (2014)
17. Han, S.: A globally convergent method for nonlinear programming. JOTA **22**, 297–310 (1977)
18. Jones, E., Oliphant, T., Peterson, P., et al.: SciPy: Open source scientific tools for Python (2001–2015). URL http://www.scipy.org/
19. Kirches, C.: Fast Numerical Methods for Mixed-Integer Nonlinear Model-Predictive Control. In: H. Bock, W. Hackbusch, M. Luskin, R. Rannacher (eds.) Advances in Numerical Mathematics. Springer Vieweg, Wiesbaden (2011). ISBN 978-3-8348-1572-9
20. Lenders, F., Kirches, C.: pySLEQP Source Code. http://www.iwr.uni-heidelberg.de/groups/optimus/software/
21. Leyffer, S., Munson, T.: A Globally Convergent Filter Method for MPECs. Preprint ANL/MCS-P1457-0907, Mathematics and Computer Science Division, Argonne National Laboratory, 9700 South Cass Avenue, Argonne, IL 60439, U.S.A. (2007)
22. Mahajan, A., Leyffer, S., Kirches, C.: Solving mixed-integer nonlinear programs by QP diving. Technical Report ANL/MCS-P2071-0312, Mathematics and Computer Science Division, Argonne National Laboratory, 9700 South Cass Avenue, Argonne, IL 60439, U.S.A. (2011)
23. Mangasarian, O., Fromovitz, S.: Fritz John necessary optimality conditions in the presence of equality and inequality constraints. J. Math. Anal. Appl. **17**, 37–47 (1967)
24. Murtagh, B., Saunders, M.: MINOS 5.51 User's Guide. Tech. Rep. SOL 83-20R (2003)
25. Nocedal, J., Wright, S.: Numerical Optimization, second edn. Springer Verlag, Berlin Heidelberg New York (2006). ISBN 0-387-30303-0 (hardcover)