Les Cahiers du GERAD ISSN: 0711–2440

NLP.py: An object-oriented environment for large-scale optimization

S. Arreckx, D. Orban,
N. van Omme

G–2016–42

June 2016

Cette version est mise à votre disposition conformément à la politique de libre accès aux publications des organismes subventionnaires canadiens et québécois.

Avant de citer ce rapport, veuillez visiter notre site Web (https://www.gerad.ca/fr/papers/G-2016-42) afin de mettre à jour vos données de référence, s'il a été publié dans une revue scientifique.

This version is available to you under the open access policy of Canadian and Quebec funding agencies.

Before citing this report, please visit our website (https://www.gerad.ca/en/papers/G-2016-42) to update your reference data, if it has been published in a scientific journal.

Les textes publiés dans la série des rapports de recherche *Les Cahiers du GERAD* n'engagent que la responsabilité de leurs auteurs.

La publication de ces rapports de recherche est rendue possible grâce au soutien de HEC Montréal, Polytechnique Montréal, Université McGill, Université du Québec à Montréal, ainsi que du Fonds de recherche du Québec – Nature et technologies.

Dépôt légal – Bibliothèque et Archives nationales du Québec, 2016
– Bibliothèque et Archives Canada, 2016

The authors are exclusively responsible for the content of their research papers published in the series *Les Cahiers du GERAD*.

The publication of these research reports is made possible thanks to the support of HEC Montréal, Polytechnique Montréal, McGill University, Université du Québec à Montréal, as well as the Fonds de recherche du Québec – Nature et technologies.

Legal deposit – Bibliothèque et Archives nationales du Québec, 2016
– Library and Archives Canada, 2016

GERAD HEC Montréal
3000, chemin de la Côte-Sainte-Catherine
Montréal (Québec) Canada H3T 2A7

Tél. : 514 340-6053
Téléc. : 514 340-5665
info@gerad.ca
www.gerad.ca

NLP.py: An object-oriented environment for large-scale optimization

Sylvain Arreckx
Dominique Orban
Nikolaj van Omme

GERAD & Department of Mathematics and Industrial Engineering, Polytechnique Montréal, Montréal (Québec) Canada

sylvain.arreckx@gerad.ca
dominique.orban@gerad.ca
nikolaj.van-omme@polymtl.ca

June 2016

Les Cahiers du GERAD
G–2016–42

Copyright © 2016 GERAD

Abstract: NLP.py is a programming environment to model continuous optimization problems and to design computational methods in the high-level and powerful Python language with performance-critical parts implemented in Cython, a low-level superset of Python that compiles to machine code. With the aim of designing numerical methods, NLP.py is accompanied by an extensive set of building blocks to solve the linear algebra and subproblems typically encountered in the solution of large-scale convex and nonconvex problems, including direct and iterative method for linear systems, linesearch strategies, trust-region subproblems, and bound-constrained subproblems. NLP.py supports several sparse matrix packages, including our own novel CySparse library. NLP.py features turnkey algorithms for problems with specific structure along with tools to assess performance. The extensible nature of NLP.py combines with the might and ubiquity of Python to make it a powerful development and analysis environment for optimization researchers and practitioners.

Keywords: Cython, linear algebra, linear and nonlinear optimization, object-oriented programming, Python, scientific computing, sparse matrix

Résumé : NLP.py constitue un écosystème de programmation simplifiant le développement d'algorithmes d'optimisation dans un langage de haut-niveau aussi puissant que Python. Il facilite également la modélisation de problèmes d'optimisation continue. Les tâches demandant plus de ressources sont, quant à elles, implémentées en Cython, un surensemble de bas niveau de Python. Dans le but de concevoir des méthodes numériques, NLP.py donne accès à un ensemble de blocs permettant de résoudre les systèmes linéaires et sous-problèmes généralement rencontrés dans la résolution de problèmes d'optimisation convexe et non convexe à grande échelle. Parmi ces blocs se trouvent un ensemble de méthodes directes et itératives pour la résolution de systèmes linéaires. En outre, plusieurs recherches linéaires y sont implémentées de même que des méthodes résolvant des sous-problèmes avec contraintes de borne ou de région de confiance. NLP.py supporte plusieurs librairies de matrice creuse, y compris notre toute nouvelle librairie CySparse. Finalement, plusieurs algorithmes de pointe pour des problèmes avec une structure particulière y ont été implémentés ainsi que des outils pour évaluer leur performance. La nature extensible de NLP.py combinée à la puissance et l'ubiquité de Python en font un environnement de développement et d'analyse puissant pour les chercheurs et les utilisateurs de l'optimisation.

Mots clés : Cython, algèbre linéaire, optimisation linéaire et non linéaire, programmation orientée-objet, Python, calcul scientifique, matrice creuse

1 Introduction

We describe an open source Python programming environment for optimization that combines the advantages of scripting and compiled languages. NLP.py, which stands for *nonlinear programming in Python*, is written with the design of large-scale novel optimization methods in mind but may also be used as a set of optimization solvers. Rather than trying to offer as wide a selection as possible of methods, a conscious decision in the design of NLP.py is to implement a few efficiency-driven cutting-edge methods that represent recent research, and to provide access to effective commonly-used building blocks. With those building blocks, users are able to assemble existing methods for testing, or to prototype novel methods for research. As a consequence, the researcher is able to concentrate on the logic of the algorithms rather than on the intricacies of core features such as a Wolfe linesearch or the efficient solution of a symmetric indefinite linear system. One of the main design goals is to let users experiment with variants of computational methods by swapping an implementation detail for another, e.g., changing a linesearch scheme for another, or a quasi-Newton approximation for another. The Python language itself reinforces this aspect, being non-intrusive, ubiquitous, and easy to learn.

Core NLP.py focuses on optimization and is accompanied by satellite packages that users can install as needed and focusing mainly on linear algebra operations, including sparse matrix libraries, symmetric indefinite factorizations, sparse QR factorization, and Krylov methods.

NLP.py has already been used successfully to implement numerical algorithms for linear programming and unconstrained, bound-constrained, and nonlinearly-constrained optimization. Those algorithms have been applied to solve, among others, problems from the CUTE/AMPL collection (Gould, Orban, and Toint, 2003b; Vanderbei, 2009), the COPS collection (Dolan, Moré, and Munson, 2004), the McMPEC collection of problems with complementarity constraints (Leyffer, 2004; Coulibaly and Orban, 2012) and structural optimization problems formulated as programs with vanishing constraints (Curatolo, 2008). More recently, it has been used to develop a factorization-free augmented Lagrangian implementation for structural design problems that arise in aircraft wing design optimization (Arreckx, Lambe, Martins, and Orban, 2015).

It is our contention that NLP.py offers the flexibility, power and ease of use for teaching optimization, studying numerical methods, researching new algorithms and exploring new ideas, all the while retaining efficiency and promoting expandability and code reuse at all levels. NLP.py may be obtained from github.com/PythonOptimizers/NLP.py.git.

Related work

Object-oriented languages such as C++ have gained considerable momentum in the scientific computing community in the past two decades, partly because of the flexibility that they offer and partly because of their popularity in businesses and research institutions. Object-oriented languages offer a degree of abstraction that lets programmers devise natural, powerful and expandable toolkits. Among the advantages offered by object-oriented languages, abstraction, inheritance and polymorphism allow the programmer to specialize a given method of a given object via subclassing and overriding. This is a useful feature in the context of optimization algorithms because it allows to subclass, say, an algorithm, and specialize it.

C++ is often the language of choice for large-scale object-oriented solvers and libraries. Meza, Oliva, Hough, and Williams (2007) propose the OPT++ object-oriented C++ toolkit for optimization that distinguishes between an algorithm-independent class hierarchy for problems and a class hierarchy for numerical methods that is based on common algorithmic traits. OPT++ supports parallel computing capabilities and simulation-based optimization, which make it a promising library for large-scale real-world problems. OOQP (Gertz and Wright, 2003) is a C++ package for convex quadratic programming that allows the solution of problems with specialized structure via subclassing and polymorphism. The TAO Toolkit for Advanced Optimization of Benson, McInnes, Moré, Munson, and Sarich (2016) is an object-oriented framework for optimization that is largely based on the parallel linear algebra capabilities of the PETSc library (Balay, Buschelman, Gropp, Kaushik, Knepley, McInnes, Smith, and Zhang, 2009). OOPS (Gondzio and Sarkissian, 2003; Gondzio and Grothey, 2007) is a parallel interior-point solver for large-scale linear, quadratic and nonlinear programming that exploits the nested block structure often present in large-scale problems.

However, low-level languages such as Fortran or C++ have a rather steep learning curve and long write-compile-link-debug cycles. High-level scripting languages such as Python let the programmer do away with memory allocation concerns, and are typically far easier to learn and use than low-level compiled languages, which often outweighs the performance hit normally associated with them. Python is a full-fledged object-oriented programming language with a standard library that is nearly as extensive as that of C and C++. There has been much activity recently devoted to developing collections of tools for both researchers, modelers and practitioners in the Python programming language. We review those that, in our view, are prominent in the current landscape.

Pyomo (Hart, Laird, Watson, and Woodruff, 2012) is a Python library for modeling optimization problems and is part of the Coopr (Hart, 2009) optimization repository. The syntax of Pyomo is strongly inspired by that of the AMPL modeling language and it currently supports linear, nonlinear and stochastic programs. In Pyomo, derivatives are computed via the ASL but it does not rely on availability of an AMPL engine. Rather, a Pyomo model is converted to a so-called *nl* file, representing computational graphs, which is then fed to the ASL (Gay, 2005).

PuLP (Stuart, OSullivan, and Dunning, 2011) is a modeling tool for linear programming that is able to export problems to MPS or LP format and subsequently call a linear programming solver such as GLPK (Makhorin, 2006), COIN, CPLEX or X-PRESS.

CVXOPT (Dahl and Vandenberghe, 2009) is a complete environment for modeling and solving convex problems, whether differentiable or not, by way of an interior-point method. CVXOPT uses its own interface to the BLAS for fast array and dense matrix operations and its own implementation of a sparse matrix library.

PyOpt (Perez, Jansen, and Martins, 2012) is a Python framework for formulating and solving optimization problems. Its goal is somewhat different from NLP.py as it provides an optimization framework with access to a variety of existing optimization algorithms accessible through a common interface. The focus is on formulating and solving nonlinear constrained optimization problems rather than developing new optimization algorithms.

Finally, Audet, Dang, and Orban (2010) propose the OPAL Python environment for modeling and solving non-smooth optimization problems with particular emphasis on algorithmic parameter optimization. OPAL interfaces a mesh-adaptive direct search method and provides modeling facilities to describe parameter-optimization problems and to assess performance by way of tailored combinations of atomic performance measures.

Python's notoriety is also apparent in major linear algebra libraries such as PETSc and TRILINOS, which offer Python bindings (Balay et al., 2009; Sala, Spotz, and Heroux, 2008).

The rest of the paper is organized as follows: Section 2 describes the design philosophy of NLP.py, Section 3 summarizes the modeling facilities offered, Section 4 describes our in-house Cython sparse matrix library, Section 5 reviews the optimization and linear algebra building blocks available, Section 6 gives a brief overview of a few of the complete solvers written in NLP.py, and Section 7 shows examples illustrating the modularity of NLP.py. Finally, some concluding remarks are provided in Section 8.

2 Overall design and structure

NLP.py may be viewed as a collection of tools written in Python and interfaces to core libraries written with the Cython (Behnel, Bradshaw, Citro, Dalcin, Seljebotn, and Smith, 2011) extension of Python. Cython is a superset of Python that facilitates interfacing with libraries that expose a C API and allows users to type variables. Because Cython code is compiled, it results in increased performance, yet remains easier to maintain and update than C or C++. Thanks to the strong connection between Python and Cython, core libraries appear transparently to the user as regular Python objects.

A solid sparse linear algebra library is essential if one is to solve large-scale sparse problems efficiently. The library of choice in NLP.py is CySparse (Arreckx, Orban, and van Omme, 2016b), but several other libraries are also supported to varying degrees. CySparse is described in Section 4.

The components of NLP.py revolve around the main tasks with which one is confronted in both modeling and algorithmic design. We now briefly review those tasks and describe them in more depth in the next sections.

Supplying derivatives The first important decision about a model is whether derivatives are available and in the affirmative, whether they will be implemented by hand, supplied by an automatic differentiation engine, or approximated using, say, finite differences or a quasi-Newton scheme.

Matrix storage schemes If it is anticipated that a solver will require Jacobians and/or Hessians, a user must decide how they should be stored—e.g., as dense arrays, sparse matrices in a specific storage format, or implicitly as linear operators. The choice of storage scheme is closely related to what types of operations a solver will need to perform with matrices, e.g., construct block saddle-point systems, and what linear algebra kernels will be used during the iterations.

Modeling Models are represented as Python objects that derive from one or two base classes. The object-oriented design of NLP.py dictates that a model results by inheritance from a model class defining the provenance of derivatives with another model class specifying a matrix storage scheme. For instance, multiple model classes obtain their derivatives from ADOL-C—one per matrix storage scheme—and all derive from a base class named `AdolcModel`. Similarly, multiple model classes store matrices according to one of the formats available in the CySparse library—one per derivative provider, and all derive from a base class named `CySparseModel`. For instance, a `CySparseAdolcModel` class could be defined as inheriting from both `AdolcModel` and `CySparseModel`.

Passing models to solvers The choice of a solver depends on multiple factors, including the matrix storage scheme, but more importantly, the structure of the problem. In NLP.py, several solvers are available and correspond to several types of problems. There are solvers for unconstrained optimization, linear and convex quadratic optimization, equality-constrained convex or nonconvex optimization, problems with complementarity constraints, and general problems featuring a mixture of equality constraints, inequality constraints and bounds. A main driver that may be called from the command line is supplied with each solver for problems written in the AMPL modeling language. The PyKrylov companion package supplies several iterative solvers for linear least-squares problems, including regularized problems and problems with a trust-region constraint.

Designing solvers Optimization researchers must often make implementation choices that strike a balance between performance, adherence to theoretical requirements and ease of implementation. An example that comes to mind is a linesearch-based method in which the search must ensure satisfaction of the strong Wolfe conditions. Implementing such a linesearch is far more involved than implementing a simple Armijo backtracking search. The latter may not satisfy the assumptions necessary for convergence, but may still perform well in practice. Similarly, a trust-region method that relies on models using exact second derivatives may perform well, but what if a quasi-Newton approximation is used instead? What if we wish to terminate the iterations early if a condition depending on external factors is satisfied? What if we wish to experiment with a different merit function? It seems important for researchers to be able to experiment easily and swap a linesearch procedure for another, swap a model with exact second derivatives for a quasi-Newton model, and so forth.

The next sections elaborate on the above aspects.

3 Modeling

In NLP.py, a general optimization problem is formulated as

$$\underset{x\in\mathbb{R}^n}{\text{minimize}} \;\; f(x) \quad \text{subject to} \;\; c^{\mathrm{L}} \le c(x) \le c^{\mathrm{U}}, \;\; \ell \le x \le u, \tag{1}$$

where $f : \mathbb{R}^n \to \mathbb{R}$ is the objective function, $c : \mathbb{R}^n \to \mathbb{R}^m$ is the (vector-valued) constraint function, and $c^{\mathrm{L}} \in (\mathbb{R} \cup \{-\infty\})^m$, $c^{\mathrm{U}} \in (\mathbb{R} \cup \{+\infty\})^m$, $\ell \in (\mathbb{R} \cup \{-\infty\})^n$, and $u \in (\mathbb{R} \cup \{+\infty\})^n$. Bounds on the variables appear separately from other types of constraints because numerical methods often treat them differently. The j-th general constraint is an equality constraint if $c_j^{\mathrm{L}} = c_j^{\mathrm{U}}$. Similarly, if $\ell_i = u_i$, the i-th variable is fixed and may technically be eliminated from the problem.

General optimization problems are represented using a subclass of the `NLPModel` abstract class. The main idea is that `NLPModel` acts as a placeholder for attributes and methods common to all optimization problems, and that a subclass specifies the structure of a problem of interest—e.g., an unconstrained problem, a bound-constrained problems, a quadratic problem, and so forth. An instance of the subclass gives life to the model by filling in the blanks. It does so by giving values to attributes such as the number of variables and of general constraints, arrays representing c^{L}, c^{U}, ℓ and u, an initial guess, and initial Lagrange multiplier estimates. The instance has methods for querying the model, including evaluating the objective function, the gradient of the objective, the (sparse) Jacobian matrix of the constraint functions, and the (sparse) Hessian matrix of the Lagrangian.

NLP.py predefines the `UnconstrainedNLPModel` and `BoundConstrainedNLPModel` subclasses of `NLPModel` that may be used as a shortcut to model unconstrained and bound-constrained problems. Listing 1 shows one way to implement the generalized Rosenbrock function for an arbitrary value of n and how to create an instance with $n = 10$.

```
1  from nlp.model.nlpmodel import UnconstrainedNLPModel
2  from numpy import sum
3  from numpy.random import random
4
5  class Rosenbrock(UnconstrainedNLPModel):
6  def obj(self, x):
7  return sum((1-x[:-1])**2 + 100*(x[1:]-x[:-1]**2)**2)
8
9  prob = Rosenbrock(10, name="Generalized Rosenbrock")
10 prob.obj(random(10))  # evaluate objective at a random point
```

Listing 1: Subclassing `UnconstrainedNLPModel` and creating an instance.

While Listing 1 seems intuitive we immediately hit severe limitation of this approach: we must hard code the derivatives of the objective and constraints functions if a solver is to make use of them. One might use finite-difference approximations but the order of convergence of descent methods often suffers from such approximations. A much more viable approach is to use automatic differentiation (AD) (Griewank, 2000). This is especially interesting in the large-scale case as the cost of evaluating derivatives via AD is typically a moderate multiple of the cost of evaluating the function itself. A number of generic automatic-differentiation packages are readily available—see `www.autodiff.org`. Some AD packages already have mature Python bindings and there also exist pure Python AD packages. NLP.py offers interfaces to ALGOPy (Walter, 2011b), PyADOLC (Walter, 2011c) and PyCppAD (Walter, 2011a).

Modeling with, e.g., PyADOLC is as simple as these authors could hope. Models in which derivatives should be computed by ADOL-C should be instances of a subclass of `AdolcModel`. Defining a class for unconstrained problems whose derivatives should be computed by ADOL-C (Walther, Kowarz, and Griewank, 2005) consists in inheriting from both `UnconstrainedNLPModel` and `AdolcModel`. The latter is itself a subclass of `NLPModel` that ensures that only functions need be supplied and subsequent derivatives will be evaluated behind the scenes by ADOL-C. Listing 2 illustrates the process without repeating operations already performed in Listing 1.

```
1  from nlp.model.adolcmodel import AdolcModel
2
3  class UnconstrainedAdolcModel(UnconstrainedNLPModel, AdolcModel):
4  pass  # do nothing; the base classes do all the work
5
6  class AdolcRosenbrock(UnconstrainedAdolcModel):
7  def obj(self, x):
8  return np.sum((1-x[:-1])**2 + 100*(x[1:]-x[:-1]**2)**2)
9
10 prob = AdolcRosenbrock(10, name="Generalized Rosenbrock")
11 prob.obj(random(10))  # evaluate objective at a random point
12 prob.grad(random(10))  # evaluate gradient at a random point
13 prob.hess(random(10))  # evaluate Hessian at a random point
```

Listing 2: Inheriting from both `UnconstrainedNLPModel` and `AdolcModel`.

Listing 2 illustrates how multiple inheritance is used in NLP.py to mix base classes together so as to obtain models with the desired features. The same effect could be achieved by defining the class `AdolcRosenbrock` as inheriting from the `Rosenbrock` class of Listing 1 and from `AdolcModel`. The resulting `AdolcRosenbrock` model has dense second derivatives, but we obtain sparse Hessians by inheriting from `SparseAdolcModel` instead of `AdolcModel`. We obtain sparse Hessians in SciPy format by inheriting from `SciPyAdolcModel`, etc. The same principles may be used to define unconstrained quasi-Newton models, quadratic models in which the Hessian is stored in CySparse format, bound-constrained models in which derivatives are computed by CppAD, *etc*.

Another convenient approach implemented in NLP.py is to use the AMPL modeling language (Fourer, Gay, and Kernighan, 2002) and leverage its mature automatic-differentiation features as most researchers and users of optimization are already familiar with it. Modeling in AMPL is simple and intuitive, and several standard benchmark collections are modeled in AMPL, e.g., (Dolan et al., 2004; Vanderbei, 2009). In NLP.py, AMPL models are used in the same way as `NLPModel`s and are generated from the *model* and *data* files constituting the AMPL model, or from the *nl* file decoded by the AMPL engine. Essentially, the *nl* file contains the expression tree of all linear and nonlinear expressions in the model and allows for automatic differentiation via the ASL (Gay, 1997).

In a similar vein, a subclass of `NLPModel` that represents a model decoded from a CUTEst problem (Gould, Orban, and Toint, 2015b) is in development.

Lastly, PDE-constrained problems have been in the spotlight in recent years as challenging and highly structured. In our experience, PDE libraries have a steep learning curve and the optimization methods that they feature, if any, are few and often written to solve specific problems. The great divide between optimization libraries and PDE libraries makes it difficult for optimization research to benefit from testing on a large base of PDE-constrained problems and for PDE libraries to benefit from the latest advances in optimization. In the Python language, there is however a good match. FEniCS (Logg, Mardal, Wells, et al., 2012) is an extensive finite-element library with a full-featured Python interface named DOLFIN (Logg and Wells, 2010). DOLFIN allows the user to define domains, meshes, function spaces, finite-element families to approximate unknowns, and to model functionals and sets of PDEs in weak form with extraordinary ease. For given domain, mesh, function space and finite-element family, a functional is automatically discretized and it is possible to evaluate it as well as its derivatives with respect to the unknown function. Similarly, a set of PDEs in weak form with accompanying boundary conditions is automatically discretized and its derivatives can also be obtained. That makes it possible to devise a generic modeling interface for PDE-constrained problems. In NLP.py, the latter has materialized as a subclass of `NLPModel` named `PDENLPModel`. In order to be instantiated, a `PDENLPModel` must be associated with a domain, a mesh, a function space, boundary conditions, as well as an optional initial guess, bounds on the unknown function, and possibly constraint left and right-hand sides. Before creating an instance, a user must subclass `PDENLPModel` in order to specify an objective functional and the constraints. PDE-constrained problems, and variational calculus problems in general, are a recent addition to NLP.py and will be the subject of a follow-up report. In the present paper,

we show a simple example that illustrates the anatomy of such problems. Consider the distributed Poisson control problem with Dirichlet boundary conditions

$$\begin{aligned} \underset{u,f}{\text{minimize}} \quad & \tfrac{1}{2}\int_\Omega |u(x)-u_0(x)|^2 \, \mathrm{d}x + \tfrac{1}{2}\beta \int_\Omega |f(x)|^2 \, \mathrm{d}x \\ \text{subject to} \quad & -\nabla\cdot(\nabla u(x)) = f(x) \quad x \in \Omega, \\ & u(x) = u_0(x) \quad x \in \partial\Omega, \end{aligned}$$

where $\beta > 0$ is a regularization parameter and u_0 is given. The implementation of the distributed control problem in NLP.py is given in Listing 3. The specification of the boundary conditions was left out for conciseness. In DOLFIN notation, `dot(f,f)*dx` represents the integral over the entire domain of $|f|^2$. Though not all details are shown, if `model` is an instance of the `DistributedControl` class, it is an `NLPModel` like any other, and one may call `model.obj()`, `model.grad()` and `model.hess()` as with any other model. Behind the scenes, DOLFIN is computing derivatives for us and the infrastructure set in `PDENLPModel` arranges so the model appears to the user as any other model.

```
1 class DistributedControl(PDENLPModel):
2 def register_objective_functional(self):
3 u, f = split(self.u)    # (u,f) lives in mixed-FEM space
4 du = u - self.target    # self.target is u0
5 return 0.5 * (dot(du,du)*dx + self.beta * dot(f,f)*dx)
6
7 def register_constraint_form(self):
8 u, f = split(self.u)
9 v, w = split(TestFunction(self.function_space))
10 return dot(grad(v),grad(u))*dx - v*f*dx
```

Listing 3: Implementation of a simple boundary control problem.

The above examples show that NLP.py tries to attain the level of abstraction that is necessary in order to be able to write optimization solvers that are agnostic to the provenance of models, provided that they adhere to the (very general) interface described by `NLPModel`.

Any instance of any subclass of `NLPModel` may be passed directly to optimization algorithms in NLP.py or may be composed first with another class so as to apply problem transformations. An example situation where this is useful is when an algorithm expects to receive a problem with only equality constraints and bounds. The `SlackFramework` class does just this by inheriting from `NLPModel` and specializing the model by adding slack variables. This means that a `SlackFramework` object behaves exactly as an `NLPModel` object but the methods to evaluate the constraints, the bounds and the constraints Jacobian reflect the new form of the problem:

$$\underset{x\in\mathbb{R}^n, s\in\mathbb{R}^m}{\text{minimize}} \; f(x) \quad \text{subject to} \; c(x) - s = 0, \; c^{\mathrm{L}} \le s \le c^{\mathrm{U}}, \; \ell \le x \le u. \tag{2}$$

A more elaborate example is the `AugmentedLagrangian` class, which derives from `NLPModel` and represents the bound-constrained proximal augmented Lagrangian problem whose (parametrized) objective function is

$$f(x) - y_k^T(c(x)-s) + \tfrac{1}{2}\rho_k(\|x-x_k\|^2 + \|s-s_k\|^2) + \tfrac{1}{2}\delta_k\|c(x)-s\|_2^2, \tag{3}$$

where $\rho_k \ge 0$ is a proximal parameter, $\delta_k > 0$ is a penalty parameter, $y_k \in \mathbb{R}^m$ represents a current approximation to the vector of Lagrange multipliers associated to the equality constraints of (2), and whose only constraints are the bounds in (2).

4 CySparse: A fast sparse matrix library

CySparse (Arreckx et al., 2016b) is a sparse matrix library written in Cython that can be used in Python and Cython projects. CySparse was written as a successor of PySparse (Geus, Orban, and Wheeler, 2009) with the main goal of eliminating C libraries and the intricacies of interfacing them, and using instead libraries

written in Python and Cython. CySparse's typed matrix indices and elements are fully compatible with NumPy (Oliphant, 2006) internal types.

The three sparse matrix data structures currently supported are modeled after PySparse: LL (Linked List), CSC (Compressed Sparse Column) and CSR (Compressed Sparse Row). SciPy (Jones, Oliphant, Peterson, et al., 2001) offers corresponding structures. We compare them in section 4.1.

A specialized implementation of each data structure exists for each index and element type. In addition, each data structure provides the option to store explicit zeros. Only the lower triangle of symmetric matrices is stored. With two index types and seven element types, there are 84 combinations. In order to specialize code for each combination, we created our own code generation tool based on the Jinja2 (Ronacher, 2008) templating engine and packaged it as the cygenja (Arreckx, Orban, and van Omme, 2016a) standalone library, which may be reused in any Cython project. NumPy and SciPy use similar tools to generate their own typed versions of templated code. When Cython's *fused types* mechanism has matured, the templating engine can technically be removed from CySparse, which should reduce its footprint.

One of the strengths of CySparse is the frequent use of *lazy evaluation*. Certain operations such as matrix addition and multiplication, and multiplication by a scalar are delayed until an end result is required. The matrix product AB of two sparse matrices is not computed until the user explicitly requests a result, such as the product itself, the application of AB to a vector, the (i, j)-th element of AB, etc.

LL matrices have a *view* attached to them—a proxy data structure that corresponds to selected parts of it. Views are pointers and consume little memory. Therefore, extracting submatrices from a given matrix is a cheap operation. Similarly, associated matrices, such as the transposed, conjugate and transpose conjugate, are proxies and can be used almost anywhere a real matrix can.

4.1 Benchmarks

We provide a few benchmarks to showcase the performance of CySparse. We compare it with PySparse (Geus et al., 2009) and SciPy and we only discuss multiplication of a sparse matrix with a dense NumPy vector. We randomly generate square sparse matrices of size $n \times n$ with nnz nonzero elements. We generate dense NumPy vectors with NumPy's `arange(0, n, dtype=np.float64)`.

For each benchmark, we run the four scenarii described in Table 1 100 times for each operation. For each scenario, operations are ranked by runtime. The most efficient implementation gets a value of 1.0. The values of the other operations are relative, i.e. an operation with value k takes k times as long to execute as the most efficient operation.

Table 1: Four benchmark scenarii.

Scenario	n	nnz
1	10^4	10^3
2	10^5	10^4
3	10^6	10^5
4	10^6	$5 \cdot 10^3$

CySparse, PySparse and SciPy, were compiled and tested with the same flags on the same machine and using `int32` indices and `float64` elements.

Table 2 reports benchmarks on the basic `matvec` operation, i.e. the multiplication Av where A is a $n \times n$ sparse matrix and v is a NumPy vector of length n.

CySparse lets the user simply type `A*v` to compute the product. PySparse does not support the notation `A*v` and only offers `A.matvec(v,y)` where `y` is a preallocated vector to store the result. Because both SciPy and CySparse allocate such a vector transparently, we take into account the time required by PySparse to allocate `y`.[1]

[1] Using `y = numpy.empty(n, dtype=numpy.float64)`.

Table 2: Sparse matrix with dense contiguous vector multiplication.

y = A*v with A a LL sparse matrix							
Scenario 1		Scenario 2		Scenario 3		Scenario 4	
PySparse	1.000	PySparse	1.000	CySparse 2	1.000	PySparse	1.000
CySparse 2	1.158	CySparse 2	1.023	CySparse	1.023	CySparse	1.043
CySparse	1.220	CySparse	1.032	PySparse	1.045	CySparse 2	1.064
SciPy 2	86.395	SciPy	101.536	SciPy 2	55.690	SciPy	133.495
SciPy	87.267	SciPy 2	102.131	SciPy	56.398	SciPy 2	135.512
y = A*v with A a CSR sparse matrix							
Scenario 1		Scenario 2		Scenario 3		Scenario 4	
PySparse	1.000	CySparse 2	1.000	CySparse	1.000	PySparse	1.000
CySparse 2	1.178	CySparse	1.022	PySparse	1.000	CySparse	1.018
CySparse	1.212	PySparse	1.037	CySparse 2	1.009	CySparse 2	1.061
SciPy 2	1.333	SciPy 2	1.287	SciPy 2	1.114	SciPy	1.419
SciPy	1.373	SciPy	1.308	SciPy	1.135	SciPy 2	1.421
y = A*v with A a CSC sparse matrix							
Scenario 1		Scenario 2		Scenario 3		Scenario 4	
SciPy 2	1.000	SciPy	1.000	SciPy 2	1.000	SciPy	1.000
SciPy	1.102	SciPy 2	1.004	SciPy	1.002	SciPy 2	1.000
CySparse 2	1.107	CySparse	1.084	CySparse	1.059	CySparse	1.138
CySparse	1.146	CySparse 2	1.120	CySparse 2	1.060	CySparse 2	1.148

CySparse and CySparse 2 in Table 2 represent `y = A*v` and `y = A.matvec(v)` respectively while SciPy and SciPy 2 represent `y = A*v` and `y = A._mul_vector(v)`, respectively. The "2" variants are equivalent to `A*v` minus a small convenience layer that permits the shorthand notation.

Table 2 reveals that using LL and CSR formats, CySparse and PySparse are on par while SciPy is slightly faster than CySparse when using the CSC format.

The second benchmark in Table 3 investigates the case where the dense NumPy vector is not contiguous in memory. We can see that our specialized implementation pays off.

Table 3: CSC sparse matrix with dense non contiguous vector multiplication.

Scenario 1		Scenario 2		Scenario 3		Scenario 4	
CySparse 2	1.000	CySparse 2	1.000	CySparse 2	1.000	CySparse	1.000
CySparse	1.011	CySparse	1.098	CySparse	1.006	CySparse 2	1.007
SciPy 2	1.354	SciPy	2.273	SciPy 2	1.733	SciPy 2	3.193
SciPy	1.394	SciPy 2	2.286	SciPy	1.742	SciPy	3.216

The third and last benchmark in Table 4 compares the multiplication of two sparse matrices with a dense NumPy vector. CySparse computes $A \cdot B \cdot v$ as $A \cdot (B \cdot v)$ and clearly this is faster than first computing $A \cdot B$ and then $(A \cdot B) \cdot v$. Even when we force SciPy to compute $A \cdot (B \cdot v)$, CySparse remains slightly faster.

5 Building blocks for optimization

5.1 Globalization strategies

One of the most important ingredients in nonlinear optimization is the globalization strategy, i.e., the mechanism by which locally-convergent methods, such as Newton's method, can converge from a remote initial guess. Such mechanisms essentially come in two flavors: the linesearch and the trust region. We examine them in turn and how they are implemented in NLP.py.

Table 4: Sparse matrix with dense contiguous vector multiplication where the sparse matrix is obtained as the product of two sparse matrices.

`y = A*B*v` with `A` a CSR sparse matrix, B a CSC sparse matrix and `v` a dense NumPy vector							
Scenario 1		Scenario 2		Scenario 3		Scenario 4	
CySparse	1.000	CySparse	1.000	CySparse	1.000	CySparse	1.000
SciPy	5.386	SciPy	3.921	SciPy	3.850	SciPy	5.207
`y = A*(B*v)` with `A` a CSR sparse matrix, B a CSC sparse matrix and `v` a dense NumPy vector							
Scenario 1		Scenario 2		Scenario 3		Scenario 4	
CySparse	1.000	CySparse	1.000	CySparse	1.000	CySparse	1.000
SciPy	1.019	SciPy	1.016	SciPy	1.011	SciPy	1.049

In constrained optimization, progress is often measured by way of a *merit function* which is typically a weighted combination of the objective and constraint functions. Examples include ℓ_p-norm exact merit functions, the augmented Lagrangian function, the logarithmic barrier function or the objective function itself in unconstrained optimization. We refer the interested reader to, e.g., (Fletcher, 1987) for further information.

Let ψ denote a merit function. In linesearch methods, a search direction d is identified from the current iterate x and a steplength $t > 0$ is sought so as to produce a *sufficient* decrease in the value of ψ. The notion of sufficient decrease varies with the context and the smoothness of the merit function. When ψ is continuously differentiable, typical conditions on t are the *Armijo condition*

$$\psi(x + td) \le \psi(x) + \alpha t \nabla\psi(x)^T d, \tag{4}$$

where $0 < \alpha < 1$ is a parameter, or the *strong Wolfe conditions*, which additionally require that

$$|\nabla\psi(x + td)^T d| \le \sigma |\nabla\psi(x)^T d|, \tag{5}$$

where $\alpha < \sigma < 1$.

In NLP.py, a linesearch is represented by an abstract `LineSearch` class and is initialized by first restricting a model to a line, which is done by creating an instance of the `C1LineModel` class. If `model` represents a problem with objective f and constraints c, if `x` is the current iterate and `d` is a nonzero vector, then `C1LineModel(model, x, d)` represents a model with objective $\phi(t) := f(x + td)$ and constraints $\gamma(t) := c(x + td)$ as well as bounds on t computed from bounds in `model`, `x` and `d`. Strictly speaking, the restricted model may be infeasible unless x is feasible for the original model.

Two types of linesearch are currently available in NLP.py: The Armijo linesearch and the strong Wolfe linesearch as implemented by Moré and Thuente (1994). In addition, a modified Armijo linesearch that initially increases the step in hopes to satisfy the Wolfe conditions is used in our Python implementation of the limited-memory BFGS method described in Section 6.

The second type of globalization mechanism available in NLP.py is the trust region. In a trust-region method for unconstrained optimization, a model $m(x + d)$ of the objective f about the current iterate x is approximately minimized within a compact *trust region* containing x. Typically, this region is a Euclidean ball or a box centered at x and the model is a quadratic agreeing with f at x at least up to first order. A typical trust-region problem is thus to

$$\underset{d \in \mathbb{R}^n}{\text{minimize}} \; m(x + d) \quad \text{subject to} \; \|d\| \le \Delta. \tag{6}$$

In constrained optimization, the model is often a simplified version of a merit function. We refer the reader to the book of Conn, Gould, and Toint (2000) for more information. In the simplest case, building a quadratic model of f is done by creating an instance `QPModel(g, H)` where `g` is a vector specifying the linear term

and H is a symmetric matrix or linear operator specifying the quadratic term, but note that it is possible for the quadratic model to have constraints. The abstract TrustRegionFramework class encapsulates the details pertaining to the definition of a trust region and to its management, including the computation of the ratio of achieved to predicted reduction. For flexibility, the definition of the model is left to the user. A TrustRegionFramework is initialized with an initial trust-region radius and constants related to its management and to step acceptance.

5.2 Numerical linear algebra

In this section, we briefly describe several companion packages that combine with NLP.py and provide linear algebra tools of critical importance to optimization.

Ordering and scaling The HSL.py satellite package (Arreckx, Orban, and van Omme, 2016c) provides flexible access to ordering and scaling methods from the Harwell Subroutine Library (2007), including the profile and wavefront reducing orderings of MC60, the row-permutation package MC21, which aims to produce a matrix with as many nonzeros as possible on the main diagonal, and the scaling package MC29.

Factorization of symmetric matrices NLP.py provides access to several libraries for the solution of sparse symmetric linear systems. The Harwell Subroutine Library (2007) subroutines MA27 of Duff and Reid (1982) and MA57 of Duff (2004) implement multifrontal sparse symmetric indefinite factorizations and are flexible enough to allow the factorization of definite, symmetric indefinite, and symmetric quasi-definite (Vanderbei, 1995) matrices. HSL.py interfaces both MA27 and MA57 via the common generic class Sils.[2] Two specialized classes MA27Solver and MA57Solver provide seamless access to the factorization, solution and iterative refinement routines, and to statistics on the factorization, including the inertia of the matrix, and in particular its number of negative eigenvalues—a critical information when factorizing saddle-point matrices (Gould, 1985).

MUMPS (Amestoy, Duff, and L'Excellent, 1998) is a multifrontal factorization for both symmetric and unsymmetric matrices that targets distributed memory computers, features out-of-core factorization, and accomodates sparse right-hand sides. MUMPS.py provides a Python interface to MUMPS that is compatible with CySparse matrices. Because MUMPS.py and HSL.py share the same solver interfaces, MA27, MA57 and MUMPS may be simply swapped when used in NLP.py.

Factorization of non-symmetric matrices NLP.py can be used in conjunction with the satellite packages qr_mumps.py and SuiteSparse.py to factorize rectangular matrices, which is crucial to least-squares problems, as well as square unsymmetric matrices.

qr_mumps.py is a set of interfaces to qr_mumps (Buttari, 2013), a multicore QR factorization well suited to the solution of large and sparse least-squares and least-norm problems. In the same vein, SuiteSparse.py provides access to the multifrontal sparse LU factorization implemented in UMFPACK (Davis, 2004) and the supernodal sparse Cholesky factorization implemented in CHOLMOD.

The Python classes exposing those packages accept a sparse matrix in coordinate format or a CySparse matrix. Therefore, any CySparse matrix originating from NLP.py may be passed to those classes and factorization methods can be easily interchanged.

5.2.1 Iterative methods

PyKrylov (Orban, 2009) is a pure Python implementation of a number of Krylov subspace methods for symmetric and non-symmetric linear systems together with a library of *linear operators*, i.e., abstract objects that derive from a base LinearOperator class and encapsulate the action of a linear operator on a vector. They obey the laws of mathematical linear operators, i.e., $(A+B)x = Ax + Bx$ and $(\alpha A)x = \alpha(Ax)$. Linear operators may be used to wrap matrices and linear functions, and may be chained by way of multiplication.

[2] Symmetric Indefinite Linear System

For example, if `A` and `B` are two linear operators, `C=A*B` is another linear operator but its construction is cheap; only its action `C*x` is computed when requested, and it will be computed as `A*(B*x)`. Linear operators may be transposed, conjugated, added together, restricted or composed into block operators, making them ideal tools to work with in the context of iterative methods for linear systems.

PyKrylov provides access to limited-memory quasi-Newton operators in standard or compact form (Byrd, Nocedal, and Schnabel, 1994), including the limited-memory BFGS, DFP and SR1 approximations.

Because the most expensive operations in Krylov methods are operator-vector products, dot products and *axpy*s, the performance hit incurred by PyKrylov for being implemented in a high-level language is low. Assuming the user implements efficient operator-vector products, by relying on a fast sparse matrix package or otherwise, NumPy handles vector operations efficiently via an optimized BLAS. A considerable advantage of pure Python implementations of Krylov methods is that adding new methods is simple via subclassing. PyKrylov currently features the conjugate gradient algorithm (Hestenes and Stiefel, 1952), the bi-conjugate gradient stabilized algorithm (van der Vorst, 1992), the symmetric LQ method (Paige and Saunders, 1975), the transpose-free quasi-minimum residual algorithm (Freund, 1993), the conjugate gradient squared algorithm (Sonneveld, 1989), MINRES (Paige and Saunders, 1975), LSMR (Fong and Saunders, 2011), LSQR (Paige and Saunders, 1982b) and CRAIG (Saunders, 1995). PyKrylov is distributed separately from the main NLP.py source code as it is useful in other contexts.

Other iterative methods are part of NLP.py itself because of their relevance to optimization. Those include the truncated conjugate-gradient algorithm of Steihaug (1983), and the projected conjugate gradient algorithm of Gould, Hribar, and Nocedal (2001).

Preconditioning A good preconditioner is often essential in the iterative solution of linear systems or trust-region subproblems. The diversity of applications of optimization makes it difficult to supply useful specialized preconditioners. For this reason, a few generic preconditioners are available in NLP.py. There are currently three types of preconditioners available: diagonal, band and based on the limited-memory LDL^T factorization.

A diagonal preconditioner is simple and only consists in extracting the diagonal from an explicit matrix. If A is a square matrix, the diagonal preconditioner is D^{-1} where $d_{ii} = \max\{|a_{ii}|, 1\}$. Band preconditioners require one of the factorizations for symmetric definite matrices covered in Section 5.2. More efficient factorizations exist for banded systems, such as that of Gill, Murray, and Wright (1981) and of Schnabel and Eskow (1999). Incomplete factorizations are popular preconditioners in a variety of fields such as multigrid methods for partial-differential equations. They typically consist in computing a factorization of a matrix, dropping elements in the factors that fall below a specified threshold in absolute value or that exceed the amount of memory the user is prepared to expend. This has the effect of promoting sparse factors, at the expense of preconditioner quality. A generalization of the incomplete Cholesky factorization of Lin and Moré (1999) to symmetric quasi-definite matrices is proposed by Orban (2014) and implemented in the LLDL.py package (Arreckx, Orban, and van Omme, 2016d).

6 Solvers

NLP.py provides complete solvers for various areas of optimization. Each solver can be run programmatically or as a stand-alone executable from the command line to solve problems in AMPL format. The intent is that solver classes be subsequently called from other applications in which optimization problems must be solved. In this section, we briefly describe the solvers currently available and the problem classes to which they apply, and highlight implementation specifics.

Unconstrained optimization Unconstrained optimization problems have the general form

$$\underset{x \in \mathbb{R}^n}{\text{minimize}} \;\; f(x), \tag{7}$$

where $f : \mathbb{R}^n \to \mathbb{R}$ is generally assumed to be once or twice continuously differentiable. An important distinction between numerical methods for (7) is based on the availability of second-order derivatives. The numerical methods for unconstrained optimization currently implemented in NLP.py reflect this distinction.

TRUNK, is a factorization-free non-monotone trust-region method. At each iteration, the second-order model

$$m_k(x_k + s) = f(x_k) + \nabla f(x_k)^T s + \tfrac{1}{2} s^T \nabla^2 f(x_k) s \approx f(x_k + s)$$

is approximately minimized inside a Euclidean trust region by way of the truncated conjugate gradient algorithm of Steihaug (1983). Instead of shrinking the trust-region radius on unsuccessful iterations, a backtracking linesearch is performed along the step in the spirit of Nocedal and Yuan (1998). The truncated conjugate gradient may be preconditioned with a user-supplied preconditioner such as a band preconditioner or a limited-memory Cholesky factorization—see Section 5.2.1. The convergence of TRUNK is covered in (Conn et al., 2000).

The second method implements the limited-memory BFGS algorithm of Liu and Nocedal (1989). An approximation of the Hessian matrix is implicitly maintained as a limited-memory BFGS matrix H_k such that $H_k^{-1} \approx \nabla^2 f(x_k)$. Global convergence is promoted by way of a strong Wolfe linesearch, but in practice we have found that a modified Armijo linesearch is nearly as effective. The management of H_k is confined to a linear operator defined in PyKrylov so that products of the form $H_k^{-1} d$ take the abstract form `H*d`. Internally, the product is computed by the two-loop recursion formula (Nocedal, 1980) or using the compact storage of limited-memory matrices (Byrd et al., 1994). The main class defining the framework for the limited-memory BFGS method has as one of its members an `InverseLBFGSOperator` object whose role is to manage the circular stacks and compute matrix-vector products with H_k^{-1}. This modular design allows `InverseLBFGS` objects to be used in other contexts, such as for preconditioning.

Convex quadratic programming Interior-point methods are a well-established class of methods that have proved to be especially efficient on linear and convex quadratic programs, i.e., problems of the form

$$\underset{x \in \mathbb{R}^n}{\text{minimize}} \;\; g^T x + \tfrac{1}{2} x^T H x \quad \text{subject to} \;\; Ax = b, \;\; x \geq 0, \tag{8}$$

where $g \in \mathbb{R}^n$, $H = H^T \in \mathbb{R}^{n \times n}$ is positive semi-definite, $A \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^m$. When $H = 0$, (8) is a linear program. Note that a linear or convex quadratic program modeled for solution in NLP.py need not be in standard form, although (8) is written in this way for simplicity. Two of the best-known interior-point methods for (8) are the long-step method (Kojima, Megiddo, and Mizuno, 1993) and the predictor-corrector method of Mehrotra (1992) in augmented form (Fourer and Mehrotra, 1993). To account for situations where A is (numerically) rank deficient, we implement the primal-dual regularized variant of Friedlander and Orban (2012). At each iteration, a quasi-definite linear system with coefficient of the form

$$\begin{bmatrix} -(X^{-1} Z + \rho I) & A^T \\ A & \delta I \end{bmatrix} \tag{9}$$

is solved, for iteration-dependent values ρ and δ decreasing to zero. Factorizing the latter matrix is typically substantially cheaper and faster than factorizing the indefinite augmented matrix with $\rho = \delta = 0$. Our implementation features row and column equilibration of A prior to solution which in our experience increases robustness and the accuracy of the linear system solution.

In NLP.py, the main abstract class implementing the primal-dual-regularized interior-point solver is named `InteriorPointSolver`. A user may elect to use the scaling implemented in the Harwell Subroutine Library (2007) subroutine MC29 instead of the row and column equilibration. In our implementation, this is realized by subclassing `InteriorPointSolver` and overriding the `scale()` and `unscale()` methods.

Dehghani and Orban (2016) subclass `InteriorPointSolver` to implement a corresponding method for linear least-squares problems with linear constraints without forming the normal equations operator.

Bound-constrained optimization Bound-constrained problems have the form

$$\underset{x\in\mathbb{R}^n}{\text{minimize}}\ f(x) \quad \text{subject to}\ \ \ell \le x \le u, \tag{10}$$

and occur naturally in numerous practical applications such as image reconstruction and the discretization of optimal control problems with obstacles, but they also occur as subproblems in methods for general nonlinear problems, such as augmented-Lagrangian methods. In (10), the function f is not assumed to be convex but its second derivatives are assumed to exist and be continuous.

We elected to implement TRON (Lin and Moré, 1998), an active-set method that iteratively determines a current working set by way of a projected gradient method, and explores faces of the feasible set using a Newton trust-region method. Our implementation is factorization-free in the sense that only Hessian-vector products are required, which allows us to use quasi-Newton approximations when second-order derivatives are not available or costly to obtain. When used with limited-memory BFGS approximations, TRON becomes a trust-region variant of L-BFGS-B (Byrd, Lu, Nocedal, and Zhu, 1995).

Equality-constrained optimization NLP.py features an implementation of the funnel method of Gould and Toint (2008) for the general equality-constrained problem

$$\underset{x\in\mathbb{R}^n}{\text{minimize}}\ f(x) \quad \text{subject to}\ \ c(x) = 0. \tag{11}$$

Funnel avoids the use of penalty parameters or filters and make use of two trust-region mechanisms to promote global convergence. Steps are computed as a combination of normal and tangential components. An example of opportunity for specialization occurs in the normal step computation, which requires the solution of an inconsistent underdetermined linear least-squares problem subject to a trust-region constraint. Such a problem can be solved using one of the least-squares solvers available in PyKrylov, and various least-squares solvers can be used via subclassing.

General constrained optimization In general constrained optimization, we seek to solve (1) in its most general form, i.e., the objective function is nonconvex as are the constraint functions. For simplicity of notation and because the numerical method described in this section treats bound constraints and general inequality constraints alike, we rewrite the problem as

$$\underset{x\in\mathbb{R}^n}{\text{minimize}}\ f(x) \quad \text{subject to}\ \ c_{\mathcal{E}}(x) = 0,\ \ c_{\mathcal{I}}(x) \ge 0, \tag{12}$$

where $\mathcal{E}$ and $\mathcal{I}$ are finite index sets with $n_{\mathcal{E}}$ and $n_{\mathcal{I}}$ elements, respectively. Note that (1) can always be cast as (12).

The first method we elected to implement as part of NLP.py for the solution of general constrained optimization problems consists in a mixed interior/exterior penalty method. The formulation (12) is transformed by using an ℓ_1-penalty function to penalize infeasibility. The nonsmooth penalty terms are subsequently rewritten as smooth terms by adding *elastic variables* s to the problem. The penalized problem takes the form

$$\underset{x\in\mathbb{R}^n,\ s\in\mathbb{R}^{n_{\mathcal{C}}}}{\text{minimize}}\ \phi(x,s;\nu) \quad \text{subject to}\ \ c_i(x) + s_i \ge 0,\ \ s_i \ge 0, \quad \text{for all}\ i \in \mathcal{C}, \tag{13}$$

where

$$\phi(x,s;\nu) = f(x) + \nu \sum_{i\in\mathcal{E}} (c_i(x) + 2s_i) + \nu \sum_{i\in\mathcal{I}} s_i.$$

In the above formulation, $\mathcal{C} = \mathcal{E} \cup \mathcal{I}$, $n_{\mathcal{C}} = n_{\mathcal{E}} + n_{\mathcal{I}}$, and $\nu > 0$ is a penalty parameter that is updated at each iteration. The strong relationship between (12) and (13) makes this approach attractive. Moreover, the elastic variables have a regularizing effect in the sense that (13) always satisfies a constraint qualification even if (12) does not. For this reason, Coulibaly and Orban (2012) apply the same idea to problems with complementarity constraints. We refer the interested reader to (Gould, Orban, and Toint, 2015a) for more details. The `ElasticFramework` abstract class performs the transformation above behind the scenes using a derived class of `NLPModel`.

The second method for (12) that is implemented is an augmented Lagrangian method. Firstly slack variables are introduced so the problem has constraints of the form $c(x) = 0$ and $\ell \le x \le u$. The k-th outer iteration of the augmented-Lagrangian algorithm consists in approximately solving the bound constrained subproblem

$$\underset{x \in \mathbb{R}^n}{\text{minimize}} \;\; f(x) + c(x)^T y_k + \tfrac{1}{2}\delta_k \|c(x)\|_2^2 \quad \text{subject to} \;\; \ell \le x \le u, \tag{14}$$

where y_k is the current vector of Lagrange multipliers estimates and $\delta_k > 0$. Subproblem (14) is solved using TRON with a limited-memory structured quasi-Newton Hessian approximation, i.e., one of the form $B_k + \delta_k J(x_k)^T J(x_k)$ where $B_k \approx \nabla_{xx} L(x_k, y_k)$, and where $L(x, y) = f(x) - c(x)^T y$ is the Lagrangian of the problem and $J(x)$ is the Jacobian of c at x. This implementation of the augmented Lagrangian resembles that in LANCELOT (Gould, Orban, and Toint, 2003a) with the exception of the solution of (14). Arreckx et al. (2015) use this augmented Lagrangian in the context of high-fidelity structural-design optimization.

7 Applications

In this final section, we illustrate a few concepts from the previous sections via two examples.

Quasi-Newton TRON The first example is a customization of the TRON solver described briefly in Section 6. TRON, in the original version of Lin and Moré (1998), minimizes a sequence of quadratic models using the conjugate gradient method with an incomplete Cholesky factorization preconditioner. For this reason, it relies on exact second derivatives stored as an explicit matrix. In NLP.py, the default `TRON` class only assumes that Hessian-vector products are available and thus, applies the conjugate gradient method without preconditioner. Listing 4 shows how to define a subclass `QNTRON` of the `TRON` base class to handle models that employ a quasi-Newton Hessian approximation. In NLP.py, all solver classes possess a callback method named `post_iteration()` that is called, as the name indicates, at the end of every iteration. It allows users to perform additional tasks such as updating a limited-memory quasi-Newton approximation. After a new iterate is computed, the oldest vector pair in the set of pairs $\{s_i, y_i\}$, for $i = 1, ..., m$ is replaced by the new pair $\{\texttt{dvars}, \texttt{dgrad}\}$ computed from the most recent step. The rest of Listing 4 illustrates the creation of a model using the compact representation of a symmetric rank one approximation of the Hessian and the instantiation of a `QNTRON` solver that uses a truncated conjugate gradient to solve trust-region subproblems. The solver is started by calling its `solve()` method.

```
1 from pykrylov.linop import CompactLSR1Operator as LSR1
2 from nlp.optimize.tron import TRON
3 from nlp.optimize.pcg import TruncatedCG
4 from nlp.model.amplmodel import QNAmplModel as Model
5
6 class QNTRON(TRON):
7 """A variant of TRON with L-SR1 quasi-Newton Hessian."""
8
9 def __init__(self, *args, **kwargs):
10 super(QNTRON, self).__init__(*args, **kwargs)
11 self.save_g = True
12
13 def post_iteration(self, **kwargs):
14 # Update quasi-Newton approximation.
15 if self.step_accepted:
16 self.model.H.store(self.dvars, self.dgrad)
17
18 model = Model(problem, H=LSR1, npairs=5, scaling=True)
19
20 tron = QNTRON(model, TruncatedCG)
21 tron.solve()
```

Listing 4: Subclassing `TRON` for quasi-Newton approximations.

The performance profiles of Dolan and Moré (2002) are employed to compare the impact of using quasi-Newton approximations with TRON from the points of view of efficiency and robustness. Efficiency and

robustness rates are readable respectively on the left and right vertical axes of the profile. Figure 1 compares the basic version of TRON, which uses exact second derivatives, with two limited-memory symmetric rank one versions, using 5 and 10 pairs in history on 124 unconstrained problems and 61 bound-constrained problems from the AMPL/CUTEr collection (Vanderbei, 2009). The plots indicate the performance loss that may be expected when using limited-memory SR1 approximations to the second derivatives. They show however that the compact representation and two-loop recursion implementations perform very similarly, and that there does not seem to be a definite advantage to using 10 pairs instead of 5.

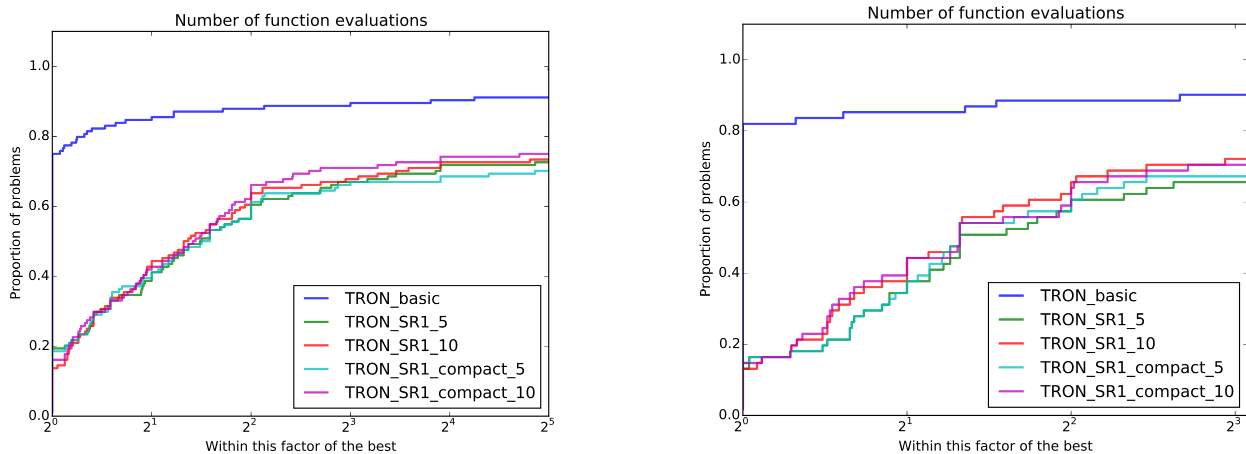

Figure 1: Performance profiles comparing the TRON with exact second derivatives and with symmetric rank one approximations using 5 or 10 pairs. Left: 124 unconstrained problems. Right: 61 bound-constrained problems.

Lagrange multipliers estimates in Funnel In the funnel method of Gould and Toint (2008), a sequence of normal and tangential steps are performed either to reduce constraint violation or the objective function while retaining the improvement in constraint violation. In preparation for a tangential step, new local Lagrange multiplier estimates y_k are computed by solving the least-squares problem

$$\min_y \tfrac{1}{2}\|g_k^N + J_k^T y\|^2,$$

where g_k^N is the gradient of a quadratic model of the objective function and J_k is the Jacobian of the constraints at x_k. In practice, one can compute such an approximation by applying a Krylov method such as LSQR (Paige and Saunders, 1982a,b), which is available in PyKrylov, starting from $y = 0$. Listing 5 illustrates how to implement this strategy by defining a subclass `LSQRFunnel` of the `Funnel` base class. Our subclass must implement the `multipliers_estimate()` method to provide Lagrange multipliers estimates.

```
1 from pykrylov.lls.lsqr import LSQRFramework
2 from nlp.optimize.funnel import Funnel
3
4 class LSQRFunnel(Funnel):
5
6 def multipliers_estimate(self, A, b):
7 LSQR = LSQRFramework(A)
8 LSQR.solve(b, show=False)
9 return (LSQR.x, LSQR.xnorm)
```

Listing 5: Subclassing `Funnel` for estimation of the Lagrange multipliers using LSQR.

If the user would now like to compute multipliers using a multi-core sparse QR factorization, the procedure is the same, and consists in subclassing `Funnel` and overloading the `multipliers_estimate()` method. That is illustrated in Listing 6.

```
from qr_mumps.solver import QRMUMPSSolver
from nlp.optimize.funnel import Funnel

class QRMUMPSFunnel(Funnel):

def multipliers_estimate(self, A, b):
qr = QRMUMPSSolver(A)
qr.factorize()
x = qr.solve(b)
return (x, numpy.linalg.norm(x))
```

Listing 6: Subclassing `Funnel` for estimation of the Lagrange multipliers using qr_mumps.

8 Conclusion

The design of the NLP.py programming environment for optimization makes extensive use of object-oriented features, such as abstract classes and multiple inheritance. The environment provides basic building blocks for large-scale computational optimization. The Python language is mature but in constant evolution and scientific extensions to the language have grown for the past 15 years from basic interfaces to extensive state-of-the-art libraries.

Admittedly, we had to make choices as to which complete solvers to include in NLP.py. Numerous other solvers are promising candidates and will certainly be included in future versions, as will other sets of tools. At the present time, our hope is that researchers find NLP.py to be a valuable development platform for novel optimization methods and that the list of tools and solvers expands. NLP.py may be obtained from github.com/PythonOptimizers/NLP.py.git.

References

P. R. Amestoy, I. S. Duff, and J.-Y. L'Excellent. Multifrontal parallel distributed symmetric and unsymmetric solvers. Computer Methods in Applied Mechanics and Engineering, 184:501–520, 1998.

S. Arreckx, A. Lambe, J. R. R. A. Martins, and D. Orban. A matrix-free augmented Lagrangian algorithm with application to large-scale structural design optimization. Optimization and Engineering, pages 1–26, October 2015.

S. Arreckx, D. Orban, and N. van Omme. Cygenja: A source code generator using Jinja2 templates. github.com/PythonOptimizers/cygenja, March 2016a.

S. Arreckx, D. Orban, and N. van Omme. CySparse: A Python/Cython library for sparse matrices. github.com/PythonOptimizers/cysparse, March 2016b.

S. Arreckx, D. Orban, and N. van Omme. HSL.py: A Python/Cython interface to the Harwell Subroutine Library. github.com/PythonOptimizers/HSL.py, March 2016c.

S. Arreckx, D. Orban, and N. van Omme. LLDL.py: A limited-memory ldl^t factorization in Python. github.com/PythonOptimizers/LLDL.py, March 2016d.

C. Audet, C.-K. Dang, and D. Orban. Algorithmic parameter optimization of the DFO method with the OPAL framework. In J. Cavazos K. Naono, K. Teranishi and R. Suda, editors, Software Automatic Tuning: From Concepts to State-of-the-Art Results, Springer, New-York, NY, 255–274, 2010.

S. Balay, K. Buschelman, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang. PETSc Web page, 2009. www.mcs.anl.gov/petsc.

S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. S. Seljebotn, and K. Smith. Cython: The best of both worlds. Computing in Science Engineering, 13(2):31–39, March 2011.

S. Benson, L. C. McInnes, J. J. Moré, T. Munson, and J. Sarich. TAO user manual (revision 3.7). Technical Report ANL/MCS-TM-322, Argonne National Laboratory, Argonne, Illinois, USA, 2016. www.mcs.anl.gov/tao.

A. Buttari. Fine-grained multithreading for the multifrontal QR factorization of sparse matrices. SIAM Journal on Scientific Computing, 35(4):C323–C345, January 2013.

R. H. Byrd, J. Nocedal, and R. B. Schnabel. Representations of quasi-Newton matrices and their use in limited memory methods. Mathematical Programming, 63(1):129–156, 1994.

R. H. Byrd, P. Lu, J. Nocedal, and C. Zhu. A limited memory algorithm for bound constrained optimization. SIAM Journal on Scientific Computing, 16(5):1190–1208, September 1995.

A. R. Conn, N. I. M. Gould, and Ph. L. Toint. Trust-Region Methods. SIAM, Philadelphia, USA, 2000.

Z. Coulibaly and D. Orban. An ℓ_1 elastic interior-point method for mathematical programs with complementarity constraints. SIAM Journal on Optimization, 22(1):187–211, 2012.

P.-R. Curatolo. Méthodes de pénalisation pour l'optimisation de structures. Ms thesis, École Polytechnique de Montréal, Montréal, Québec, Canada, 2008.

J. Dahl and L. Vandenberghe. CVXOPT: Python software for convex optimization. abel.ee.ucla.edu/cvxopt, July 2009.

T. A. Davis. Algorithm 832: UMFPACK, an unsymmetric-pattern multifrontal method. ACM Transactions on Mathematical Software, 30(2):196–199, June 2004.

M. Dehghani and D. Orban. A regularized interior-point method for constrained linear least squares. Cahier du GERAD G-2016-xx, GERAD, Montréal, QC, Canada, 2016. In preparation.

E. Dolan and J. J. Moré. Benchmarking optimization software with performance profiles. Mathematical Programming, Series B, 91(2):201–213, January 2002.

E. D. Dolan, J. J. Moré, and T. S. Munson. Benchmarking optimization software with COPS 3.0. Technical Report ANL/MCS-273, Argonne National Laboratory, Argonne, Illinois, USA, 2004.

I. S. Duff. MA57—a code for the solution of sparse symmetric definite and indefinite systems. ACM Transactions on Mathematical Software, 30(2):118–144, June 2004.

I. S. Duff and J. K. Reid. MA27—a set of Fortran subroutines for solving sparse symmetric sets of linear equations. Report AERE R10533, HMSO, London, UK, 1982.

R. Fletcher. Practical Methods of Optimization. J. Wiley and Sons, Chichester, England, second edition, May 1987.

D. C.-L. Fong and M. A. Saunders. LSMR: An iterative algorithm for sparse least-squares problems. SIAM Journal on Scientific Computing, 33(5):2950–2971, January 2011.

R. Fourer and S. Mehrotra. Solving symmetric indefinite systems in an interior-point method for linear programming. Mathematical Programming, 62(1-3):15–39, February 1993.

R. Fourer, D. M. Gay, and B. W. Kernighan. AMPL: A Modeling Language for Mathematical Programming. Duxbury Press / Brooks/Cole Publishing Company, second edition, 2002.

R. W. Freund. A transpose-free quasi-minimal residual algorithm for non-hermitian linear systems. SIAM Journal on Scientific Computing, 14(2):470–482, March 1993.

M. P. Friedlander and D. Orban. A primal–dual regularized interior-point method for convex quadratic programs. Mathematical Programming Computation, 4(1):71–107, 2012.

D. M. Gay. Hooking your solver to AMPL. Technical Report 97-4-06, Lucent Technologies Bell Labs Innovations, Murray Hill, NJ, 1997. www.ampl.com/REFS/HOOKING.

D. M. Gay. Writing `.nl` files. Technical Report SAND2005-7907P, Sandia National Laboratories, Albuquerque, NM, 2005.

E. M. Gertz and S. J. Wright. Object-oriented software for quadratic programming. ACM Transactions on Mathematical Software, 29(1):58–81, March 2003.

R. Geus, D. Orban, and D. Wheeler. PySparse: a fast sparse matrix library in Python. pysparse.sf.net, July 2009.

Ph. E. Gill, W. Murray, and M. H. Wright. Practical Optimization. Academic Press, London, 1981.

J. Gondzio and A. Grothey. Parallel interior-point solver for structured quadratic programs: Application to financial planning problems. Annals of Operations Research, 152(1):319–339, July 2007.

J. Gondzio and R. Sarkissian. Parallel interior-point solver for structured linear programs. Mathematical Programming, 96(3):561–584, 2003.

N. I. M. Gould. On practical conditions for the existence and uniqueness of solutions to the general equality quadratic programming problem. Mathematical Programming, 32(1):90–99, May 1985.

N. I. M. Gould and Ph. L. Toint. Nonlinear programming without a penalty function or a filter. Mathematical Programming, 122(1):155–196, 2008.

N. I. M. Gould, M. E. Hribar, and J. Nocedal. On the solution of equality constrained quadratic programming problems arising in optimization. SIAM Journal on Scientific Computing, 23(4):1376–1395, January 2001.

N. I. M. Gould, D. Orban, and Ph. L. Toint. GALAHAD—a library of thread-safe Fortran 90 packages for large-scale nonlinear optimization. ACM Transactions on Mathematical Software, 29(4):353–372, December 2003a.

N. I. M. Gould, D. Orban, and Ph. L. Toint. CUTEr and SifDec, a Constrained and Unconstrained Testing Environment, revisited. ACM Transactions on Mathematical Software, 29(4):373–394, December 2003b.

N. I. M. Gould, D. Orban, and Ph. L. Toint. An interior-point ℓ_1-penalty method for nonlinear optimization. In M. Al-Baali, L. Grandinetti, and A. Purnama, editors, Recent Developments in Numerical Analysis and Optimization, volume 134 of Proceedings in Mathematics and Statistics, pages 117–150, Switzerland, 2015a. Springer. special issue of NAOIII, Muscat, Oman, 2014.

N. I. M. Gould, D. Orban, and Ph. L. Toint. CUTEst: a Constrained and Unconstrained Testing Environment with safe threads for Mathematical Optimization. Computational Optimization and Applications, 60:545–557, 2015b.

A. Griewank. Evaluating derivatives: Principles and techniques of algorithmic differentiation. Number FR19 in Frontiers in Applied Mathematics. SIAM, Philadelphia, USA, 2000.

W. E. Hart. Coopr: A common optimization repository. Technical Report, Sandia National Laboratory, Albuquerque, NM, 2009. software.sandia.gov/trac/coopr.

W. E. Hart, C. Laird, J.-P. Watson, and D. L. Woodruff. Pyomo – Optimization Modeling in Python. Springer US, 2012.

Harwell Subroutine Library. A collection of Fortran codes for large-scale scientific computation. AERE Harwell Laboratory, Harwell, Oxfordshire, England, 2007. URL www.numerical.rl.ac.uk/hsl.

M. R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. Journal of research of the National Bureau of Standards, 49:409–436, 1952.

E. Jones, T. Oliphant, P. Peterson, et al. SciPy: Open source scientific tools for Python, 2001. URL www.scipy.org.

M. Kojima, N. Megiddo, and S. Mizuno. A primal-dual infeasible-interior-point algorithm for linear programming. Mathematical Programming, 61(1-3):263–280, August 1993.

S. Leyffer. MacMPEC: AMPL collection of MPECs. www.mcs.anl.gov/∼leyffer/MacMPEC, 2004.

C.-J. Lin and J. J. Moré. Newton's method for large bound-constrained optimization problems. SIAM Journal on Optimization, 9:1100–1127, 1998.

C.-J. Lin and J. J. Moré. Incomplete Cholesky factorizations with limited memory. SIAM Journal on Scientific Computing, 21(1):24–45, 1999.

D. Liu and J. Nocedal. On the limited memory BFGS method for large scale optimization. Mathematical Programming, 45(1-3):503–528, August 1989.

A. Logg and G. N. Wells. DOLFIN: Automated finite element computing. ACM Transactions on Mathematical Software, 37(2):1–28, April 2010.

A. Logg, K.-A. Mardal, G. N. Wells, et al. Automated Solution of Differential Equations by the Finite Element Method. Springer, 2012. ISBN 978-3-642-23098-1.

A. Makhorin. GNU Linear Programming Kit version 4.11. Department for Applied Informatics, Moscow Aviation Institute, Moscow, Russia, 2006. www.gnu.org/software/glpk/glpk.html.

S. Mehrotra. On the implementation of a primal-dual interior-point method. SIAM Journal on Optimization, 2(4): 575–610, November 1992.

J. C. Meza, R. A. Oliva, P. D. Hough, and P. J. Williams. OPT++: An object-oriented toolkit for nonlinear optimization. ACM Transactions on Mathematical Software, 33(2):12, June 2007.

J. J. Moré and D. J. Thuente. Line search algorithms with guaranteed sufficient decrease. ACM Transactions on Mathematical Software, 20(3):286–307, September 1994.

J. Nocedal. Updating quasi-newton matrices with limited storage. Mathematics of Computations, 35:773–782, 1980.

J. Nocedal and Y. Yuan. Combining trust region and line search techniques. In Y. Yuan, editor, Advances in Nonlinear Programming, Kluwer Academic Publishers, Dordrecht, The Netherlands, 153–176, 1998.

T. E. Oliphant. Guide to NumPy. Provo, UT, 2006. URL www.tramy.us.

D. Orban. PyKrylov: Krylov subspace methods in pure Python. github.com/dpo/pykrylov, July 2009.

D. Orban. Limited-memory ldl^t factorization of symmetric quasi-definite matrices with application to constrained optimization. Numerical Algorithms, 70(1):9–41, 2014.

C. C. Paige and M. A. Saunders. Solution of sparse indefinite systems of linear equations. SIAM Journal on Numerical Analysis, 12(4):617–629, 1975.

C. C. Paige and M. A. Saunders. Algorithm 583; LSQR: Sparse linear equations and least-squares problems. ACM Transactions on Mathematical Software, 8(2):195–209, June 1982a.

C. C. Paige and M. A. Saunders. LSQR: An algorithm for sparse linear equations and sparse least squares. ACM Transactions on Mathematical Software, 8(1):43–71, March 1982b.

R. E. Perez, P. W. Jansen, and J. R. R. A. Martins. pyOpt: A Python-based object-oriented framework for nonlinear constrained optimization. Structures and Multidisciplinary Optimization, 45(1):101–118, 2012.

A. Ronacher. Jinja2: A full featured template engine for Python. jinja.pocoo.org/, 2008.

M. Sala, W. F. Spotz, and M. A. Heroux. PyTrilinos: High-performance distributed-memory solvers for Python. ACM Transactions on Mathematical Software, 34(2):1–33, March 2008.

M. A. Saunders. Solution of sparse rectangular systems using LSQR and CRAIG. BIT Numerical Mathematics, 35(4): 588–604, December 1995.

R. B. Schnabel and E. Eskow. A revised modified Cholesky factorization algorithm. SIAM Journal on Optimization, 9 (4):1135–1148, January 1999.

P. Sonneveld. CGS, a fast Lanczos-type solver for nonsymmetric linear systems. SIAM Journal on Scientific and Statistical Computing, 10(1):36–52, January 1989.

T. Steihaug. The conjugate gradient method and trust regions in large scale optimization. SIAM Journal on Numerical Analysis, 20(3):626–637, June 1983.

M. Stuart, M. OSullivan, and I. Dunning. PuLP: a linear programming toolkit for Python. www.optimization-online.org/DB_FILE/2011/09/3178.pdf, 2011.

H. A. van der Vorst. Bi-CGSTAB: A fast and smoothly converging variant of bi-CG for the solution of nonsymmetric linear systems. SIAM Journal on Scientific and Statistical Computing, 13(2):631–644, March 1992.

R. J. Vanderbei. Symmetric quasi-definite matrices. SIAM Journal on Optimization, 5(1):100–113, February 1995.

R. J. Vanderbei. Nonlinear optimization models. www.orfe.princeton.edu/∼rvdb/ampl/nlmodels, July 2009.

S. Walter. PyCPPAD. github.com/b45ch1/pycppad, March 2011a.

S. Walter. ALGOPy. github.com/b45ch1/algopy, March 2011b.

S. Walter. PyADOLC. github.com/b45ch1/pyadolc, March 2011c.

A. Walther, A. Kowarz, and A. Griewank. Documentation of ADOL-C, 2005. URL projects.coin-or.org/ADOL-C. Updated version of Griewank et al., 1996.