Structural and Multidisciplinary Optimization manuscript No.
(will be inserted by the editor)

pyOpt: A Python-Based Object-Oriented Framework for Nonlinear Constrained Optimization

Ruben E. Perez · Peter W. Jansen · Joaquim R.R.A. Martins

Received: date / Accepted: date

Abstract We present pyOpt, an object-oriented framework for formulating and solving nonlinear constrained optimization problems in an efficient, reusable and portable manner. The framework uses object-oriented concepts, such as class inheritance and operator overloading, to maintain a distinct separation between the problem formulation and the optimization approach used to solve the problem. This creates a common interface in a flexible environment where both practitioners and developers alike can solve their optimization problems or develop and benchmark their own optimization algorithms. The framework is developed in the Python programming language, which allows for easy integration of optimization software that is programmed in Fortran, C, C++, and other languages. A variety of optimization algorithms are integrated in pyOpt and are accessible through the common interface. We solve a number of problems of increasing complexity to demonstrate how a given problem is formulated using this framework, and how the framework can be used to benchmark the various optimization algorithms.

Ruben E. Perez
Department of Mechanical and Aerospace Engineering, Royal Military College of Canada, Kingston, ON, Canada, K7K 7B4
Tel.: +1-613-541-6000 ext. 6168
Fax: +1-613-542-8612
E-mail: Ruben.Perez@rmc.ca

Peter W. Jansen
Department of Mechanical and Aerospace Engineering, Royal Military College of Canada, Kingston, ON, Canada, K7K 7B4
E-mail: Peter.Jansen@rmc.ca

Joaquim R.R.A. Martins
Department of Aerospace Engineering, University of Michigan, Ann Arbor, MI 48109, USA
Tel.: +1-734-615-9652
E-mail: jrram@umich.edu

Keywords Optimization algorithms · Constrained optimization · Object-oriented programming · Aerostructural optimization

1 Introduction

Various high quality numerical optimization packages are available to solve design optimization problems (Moré and Wright 1993). These packages are written in different programming languages and each one of them has a unique way of formulating the optimization problem to be solved. For a given optimization problem, practitioners tend to spend substantial time and effort in learning and implementing code-specific interfaces for their applications. If the optimization package is developed in a low-level language, the interfacing task becomes particularly challenging, as complex syntax, enforced syntax typing, special consideration for memory management, and compiling are required to use such packages. Optimization algorithm developers are confronted with similar problems when they want to test, compare, and benchmark new algorithms by solving multiple problems with different optimizers.

There have been efforts towards helping both practitioners and developers with the above difficulties. A number of approaches have been used. One approach has been to development of algebraic modeling languages (AMLs) such as AMPL (Fourer et al 2003), GAMS (Rosenthal 2008), and AIMMS (Bisschop and Roelofs 2008), which use a common set of mathematical constructs. Solvers make use of the common constructs to translate the optimization problem to their specific algorithmic representations. While a large set of problems can be modeled with this approach, it is

hard to integrate existing application models already programmed in other languages.

Another approach has been to develop a framework — using a standard programming language — that enables the use of different optimizers by executing them via a system call interface that uses a set of standardized files for optimization problem definition input and solution output. An example of this approach is the DAKOTA toolkit (Eldred et al 2007). Since a standard programming language is used in lieu of an algebraic modeling language, this type of framework development adds the flexibility of handling existing application models programmed in the same language.

More recently, object-oriented programming has been incorporated into such frameworks, allowing them not only to take advantage of code reusability but also to enable them to use programming constructs that are similar in nature to the mathematical constructs used by ALMs. While existing object-oriented optimization frameworks enable the solution of large and complex problems with existing application models, until recently they have mainly been programmed in low-level languages such as C/C++. An example of such a framework is OPT++ (Meza 1994; Meza et al 2007).

The problem of interfacing application models and optimization algorithms programmed in different languages still remains in such frameworks. An alternative to address such a problem is to use a high-level programming language, such as Matlab or Python. Such languages not only provide a flexible environment to model optimization problems, but also enable easier interfacing of application models and optimizers written in different low-level languages. This approach, enhanced with object-oriented constructs, leverages the ease of use and integration provided by a high-level language with the efficiency of numerical computations of compiled languages.

The advantage provided by high-level languages can be observed in a plethora of recent projects. For example, pyIPOPT (Xu 2009), CVXOPT (Dahl and Vandenberghe 2008), SciPy.optimize (Jones et al 2001), and TANGO (Tan 2007) provide direct Python interfaces to compiled-language optimizers. Similarly, the Pyomo (Hart 2009) project provides algebraic modeling capabilities within Python, while NLpy (Friedlander and Orban 2008) connects Python to the AMPL algebraic modeling language. Other projects, such as YALMIP (Lofberg 2004) and TOMLAB (Holmström et al 2010) in Matlab, or puLP (Mitchell 2009) and OpenOpt (Kroshko 2010) in Python, also offer system call interfacing frameworks to different optimizers. In some projects, the optimization algorithms themselves are implemented in Python as opposed to a compiled language Jacobs et al (2004); Friedlander and Orban (2008).

The goal of the effort described herein is to develop an object-oriented framework programmed in Python that facilitates the formulation of optimization problems, the integration of application models developed in different programming languages, and the solution and benchmarking of multiple optimizers. This facilitates the tasks for both practitioners and developers alike.

The problems to be solved can be written as a general constrained nonlinear optimization problem, i.e.,

$$\begin{aligned} \min_{x} \quad & f(x) \\ \text{subject to} \quad & g_j(x) = 0, && j = 1, \ldots, m_e, \\ & g_j(x) \leq 0, && j = m_e + 1, \ldots, m, \\ & x_{i_L} \leq x_i \leq x_{i_U}, && i = 1, \ldots, n, \end{aligned} \tag{1}$$

where $x \in \Re^n$, $f : \Re^n \rightarrow \Re^1$, and $g : \Re^n \rightarrow \Re^m$. It is assumed that the objective function $f(x)$ is a nonlinear function, and that the equality and inequality constraints can be either linear or nonlinear functions of the design variables x.

The main characteristics of the pyOpt framework are described below.

Problem-Optimizer Independence: Object-oriented constructs allow for true separation between the optimization problem formulation and its solution by different optimizers. This enables a large degree of flexibility for problem formulation and solution, allowing the use of advanced optimization features, such as nested optimization and automated solution refinement with ease and efficiency.

Flexible Optimizer Integration : pyOpt already provides an interface to a number of optimizers and enables the integration of additional optimizers both open-source and commercial alike. Furthermore, the interface allows for easy integration of gradient-based, gradient-free, and population-based optimization algorithms that solve the general constrained nonlinear optimization problem (1).

Multi-Platform Compatibility: The framework base classes and all implemented optimizers can be used and tested in different operating system environments, including Windows, Linux, and OS X, and different computing architectures, including parallel systems.

Parallelization Capability: Using the message passing interface (MPI) standard, the framework can solve optimization problems where the function evaluations from the model applications run in parallel environments. For gradient-based optimizers, it can

also evaluate gradients in parallel, and for gradient-free optimizers it can distribute function evaluations.

History and Warm-Restart Capability: The user has the option to store the solution history during the optimization process. A partial history can also be used to warm-restart the optimization.

This article is organized as follows. The next section outlines the software implementation philosophy and the programming language selection. Section 3 describes the class design in pyOpt and lists the optimization algorithms integrated into the framework. Section 4 demonstrates the solution of three optimization problems using pyOpt with multiple optimization algorithms. In the last section we present our conclusions.

2 Software Design

The design of pyOpt is driven by the need to provide an easy-to-use optimization framework not only for practitioners, but also for developers. Different programming languages were considered for the development of pyOpt and Python (Beazley 2006) was selected. Python is a free, high-level programming language that supports object-oriented programming and has a large following in the scientific computing community (Oliphant 2007; Langtangen 2008). Python fullfils all of our code design development requirements according to the principles described below.

2.1 Clarity and Usability

For optimization practitioners, the framework should be usable by someone who has only basic knowledge of optimization. An intuitive interface should be provided in which the optimization problem formulation resembles its mathematical formulation. For developers, the framework should provide intuitive object-oriented class structures where new algorithms can be integrated and tested by a wide range of developers with diverse programming backgrounds. Python provides a clear and readable syntax with intuitive object orientation and a large number of data types and structures. It is highly stable and run in interactive mode, making it easy to learn and debug. The language supports user-defined raising and catching of exceptions, resulting in cleaner error handling. Moreover, automatic garbage collection is performed, which frees the programmer from the burden of memory management.

2.2 Extensibility

The framework and its programming language should provide a solid foundation for additional extensions or modifications to the framework architecture, to its classes and modeling routines, to the optimization algorithm interfaces, and to the user application models. Python provides a simple model for loading Python code developed by users. Additionally, it includes support for shared libraries and dynamic loading, so new capabilities can be dynamically integrated into Python applications.

2.3 Portability

An important requirement for the framework is that it must work on several computer architectures. Not only should it be easily ported across computer platforms, but it should also allow easy integration of the user models and optimizers across computer platforms. Python is available on a large number of computer architectures and operating systems, so portability is typically not a limitation for Python-based applications.

2.4 Integration Flexibility

The framework should also provide the flexibility to support both tight coupling integration (where a model or optimizer is directly linked into the framework) and loose coupling integration (where a model or optimizer is externally executed through system calls). Furthermore, application models and solvers developed in heterogeneous programming languages should be easily integrated into the framework. Python excels at interfacing with high- and low-level languages. It was designed to interface directly with C, making the integration of C codes straightforward. Integration of Fortran and C++ codes can be done using freely available tools, such as `f2py` (Peterson et al 2001) and `SWIG` (Blezek 1998), respectively, which automate the integration process, while enabling access to the original code functionality from Python.

2.5 Standard Libraries

The framework should have access to a large set of libraries and tools to support additional capabilities, such as specialized numerical libraries, data integration tools, and plotting routines. Python includes a large set of standard libraries, facilitating the programming of a wide array of tasks. Furthermore, a large number

of open-source libraries are available, such as the scientific computing library `SciPy`, the numerical computing library `NumPy`, and the plotting library `matplotlib`. These libraries further extend the capabilities available to both optimization practitioners and developers alike.

2.6 Documentation

The programming language used for the framework development should be well documented, and should also provide tools to properly document the code and generate API documentation. An extensive set of articles, books, online documentation, newsgroups, and special interest groups are available for Python and its extended set of libraries. Furthermore, a large number of tools, such as `pydoc`, are available to generate API documentation automatically, and to make it available in a variety of formats, including web pages.

2.7 Flexible Licensing

To facilitate the use and distribution of pyOpt, both the programming language and the framework should have open-source licenses. Python is freely available and its open-source license enables the modification and distribution of Python-based applications with almost no restrictions.

3 Implementation

Abstract classes have been used throughout pyOpt to facilitate reuse and extensibility. Figure 1 illustrates the relationship between the main classes in the form of a unified modeling language (UML) class diagram. The class structure in pyOpt was developed based on the premise that the definition of an optimization problem should be independent of the optimizer. An optimization problem is defined by the `Optimization` abstract class, which contains class instances representing the design variables, constraints, and the objective function. Similarly, optimizers are defined by the `Optimizer` abstract class, which provides the methods necessary to interact with and solve an optimization problem instance. Each solution, as provided by a given optimizer instance, is stored as a `Solution` class instance. The details for each class are discussed below, where all class diagrams presented follow the standard UML representation (Arlow and Neustadt 2002).

3.1 Optimization Problem Class

Any nonlinear optimization problem (1) can be represented by the `Optimization` class. The attributes of this class are the name of the optimization problem (`name`), the pointer to the objective function (`objfun`), and the dictionaries that contain the instances of each optimization variable (`variables`), constraint (`constraints`), and objective (`objectives`). Each design variable, constraint, and objective is defined with its own instance to provide greater flexibility for problem reformulation. The class provides methods that help set, delete and list one or more variables, constraints and objectives instances. For example, `addCon` instantiates a constraint and appends it to the optimization problem `constraints` set. The class also provides methods to add, delete, or list any optimization problem solution that is stored in the dictionary of solution instances (`solutions`).

The design variables are represented by the `Variable` class. Attributes of the class include a variable name (`name`), a variable type identifier (`type`), its current value (`value`), as well as its upper and lower bounds (`upper` and `lower`). Three different variable types can be used: continuous, integer, and discrete. If a variable type is continuous or discrete, the user-specified upper and lower bounds are used directly. If a variable is defined as discrete, the user provides a set of choices (`choices`) and the lower and upper bounds are automatically set to represent the choice indices.

Similarly, the `Constraint` class allows the definition of equality and inequality constraints. The class attributes include the constraint name (`name`), a type identifier (`type`), and its value (`value`).

Finally, the `Objective` class is used to encapsulate the objective function value.

3.1.1 Optimization Solution Class

For a given `Optimization` instance, the `Solution` class stores information related to the optimum found by a given optimizer. The class inherits from the `Optimization` class, and hence it shares all attributes and methods from `Optimization`. This allows the user to perform automatic refinement of a solution with ease, where the solution of one optimizer is used directly as the initial point of another optimizer. Additional attributes of the class include details from the solver and its solution, such as the optimizer settings used, the computational time, and the number of evaluations required to solve the problem.

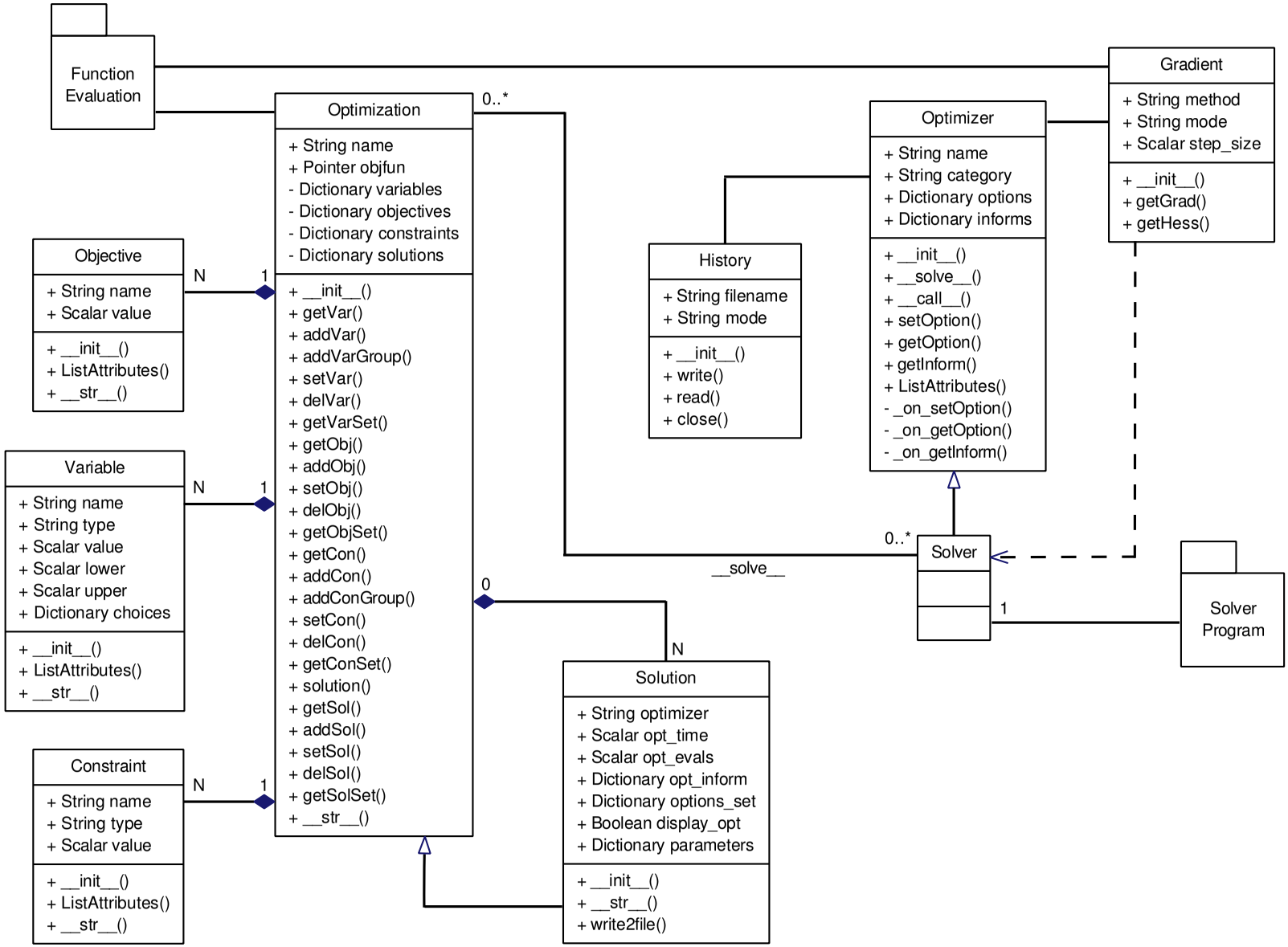


Fig. 1 pyOpt class relationships UML diagram

3.2 Optimization Solver Class

All optimization problem solvers inherit from the `Optimizer` abstract class. The class attributes include the solver name (`name`), an optimizer type identifier (`category`), and dictionaries that contain the solver setup parameters (`options`) and message output settings (`informs`). The class provides methods to check and change default solver parameters (`getOption`, `setOption`), as well as a method that runs the solver for a given optimization problem (`solve`). As long as an optimization package is wrapped with Python, this class provides a common interface to interact with and solve an optimization problem as defined by the `Optimization` class. When the solver is instantiated, it inherits the `Optimizer` attributes and methods and is initialized with solver-specific options and messages. By making use of object-oriented polymorphism, the class performs all the solver-specific tasks required to obtain a solution. For example, each solver requires different array workspaces to be defined. Depending on the solver that is used, sensitivities can be calculated using the `Gradient` class. Once a solution has been obtained, it can be stored as a solution instance that is contained in the `Optimization` class, maintaining the separation between the problem being solved and the optimizer used to solve it. The history of the solver optimization can also be stored in the `History` class. A partially stored history can also be used to enable a "warm-restart" of the optimization.

3.3 Optimization Solvers

A number of constrained optimization solvers are currently integrated into the framework. All these optimizers are designed to solve the general nonlinear optimization problem (1). They include traditional gradient-based optimizers, as well as gradient-free optimizers. A brief description of each optimizer currently implemented is presented below.

3.3.1 SNOPT

This is a sparse nonlinear optimizer written in Fortran that is particularly useful for solving large-scale constrained problems with smooth objective functions and constraints (Gill et al 2002). The algorithm consists of a sequential quadratic programming (SQP) algorithm that uses a smooth augmented Lagrangian merit function, while making explicit provision for infeasibility in the original problem and in the quadratic programming subproblems. The Hessian of the Lagrangian is approximated using a limited-memory quasi-Newton method.

3.3.2 NLPQL

This is another sequential quadratic programming (SQP) method that is also written in Fortran and solves problems with smooth continuously differentiable objective function and constraints (Schittkowski 1986). The algorithm uses a quadratic approximation of the Lagrangian function and a linearization of the constraints. A quadratic subproblem is formulated and solved to generate a search direction. The line search can be performed with respect to two alternative merit functions, and the Hessian approximation is updated by a modified BFGS formula.

3.3.3 SLSQP

This optimizer is a sequential least squares programming algorithm (Kraft 1988). It is written in Fortran and uses the Han–Powell quasi-Newton method with a BFGS update of the B-matrix and an L1-test function in the step-length algorithm. The optimizer uses a slightly modified version of Lawson and Hanson's NNLS nonlinear least-squares solver (Lawson and Hanson 1974).

3.3.4 FSQP

This code, which is available in either C or Fortran, implements an SQP approach that is modified to generate feasible iterates (Lawrence and Tits 1996). In addition to handling general single objective constrained nonlinear optimization problems, the code is also capable of handling multiple competing linear and nonlinear objective functions (minimax), linear and nonlinear inequality constraints, as well as linear and nonlinear equality constraints (Zhou and Tits 1996).

3.3.5 CONMIN

This optimizer implements the method of feasible directions in Fortran (Vanderplaats 1973). CONMIN solves the nonlinear programming problem by moving from one feasible point to an improved one by choosing at each iteration a feasible direction and step size that improves the objective function.

3.3.6 MMA/GCMMA

This is a Fortran implementation of the method of moving asymptotes (MMA). MMA uses a special type of convex approximation (Svanberg 1987). For each step of the iterative process, a strictly convex approximating subproblem is generated and solved. The generation of these subproblems is controlled by the so-called moving

asymptotes, which both stabilize and speed up the convergence of the general process. A variant of the original algorithm (GCMMA) has also been integrated into the framework. The variant extends the original MMA functionality and guarantees convergence to some local minimum from any feasible starting point (Svanberg 1995).

3.3.7 KSOPT

This Fortran code reformulates the constrained problem into an unconstrained one using a composite Kreisselmeier–Steinhauser objective function (Kreisselmeier and Steinhauser 1979) to create an envelope of the objective function and set of constraints (Wrenn 1989). The envelope function is then optimized using a sequential unconstrained minimization technique (SUMT) (Fiacco and McCormick 1968). At each iteration, the unconstrained optimization problem is solved using the Davidon–Fletcher–Powell (DFP) algorithm.

3.3.8 COBYLA

This is an implementation of Powell's nonlinear derivative-free constrained optimization that uses a linear approximation approach (Powell 1994). The algorithm is written in Fortran and is a sequential trust-region algorithm that employs linear approximations to the objective and constraint functions, where the approximations are formed by linear interpolation at $n+1$ points in the space of the variables and tries to maintain a regular-shaped simplex over iterations.

3.3.9 SOLVOPT

This optimizer, which is available in either C or Fortran, uses a modified version of Shor's r-algorithm (Shor 1985) with space dilation to find a local minimum of nonlinear and non-smooth problems (Kuntsevich and Kappel 1997). The algorithm handles constraints using an exact penalization method (Kiwiel 1985).

3.3.10 ALPSO

This is a parallel augmented Lagrange multiplier particle swarm optimizer developed in Python (Perez and Behdinan 2007). It solves nonlinear non-smooth constrained problems using an augmented Lagrange multiplier approach to handle constraints. This algoritm has been used in challenging constrained optimization applications with multiple local optima, including the aerostructural optimization of aircraft with non-planar lifting surfaces (Jansen et al 2010). Other versions of particle swarm algorithms have also been used to optimize aircraft structures (Venter and Sobieszczanski-Sobieski 2004).

3.3.11 NSGA2

This optimizer is a non-dominating sorting genetic algorithm developed in C++ that solves non-convex and non-smooth single and multiobjective optimization problems (Deb et al 2002). The algorithm attempts to perform global optimization, while enforcing constraints using a tournament selection-based strategy.

3.3.12 ALHSO

This Python code is an extension of the harmony search optimizer (Geem et al 2001; Lee and Geem 2005) that handles constraints. It follows an approach similar to the augmented Lagrange multiplier approach used in ALPSO to handle constraints.

3.3.13 MIDACO

This optimizer implements an extended ant colony optimization to solve non-convex nonlinear programming problems(Schlüter et al 2009). The algorithm is written in Fortran and handles constraints using an oracle penalty method (Schlüter and Gerdts 2009).

3.4 Optimization Gradient Class

Some of the solvers described above use gradient information. The `Gradient` class provides a unified interface for the gradient calculation. pyOpt provides an implementation of the finite-difference method (default setting) and the complex-step method (Martins et al 2003). The complex-step method is implemented automatically by pyOpt for any code in Python; for other programming languages, the user must implement the method. pyOpt also allows users to define their own sensitivity calculation, such as a semi-analytic adjoint method or automatically differentiated code. A parallel gradient calculation option can be used for optimization problems with large numbers of design variables. The calculation of the gradients for the various design variables is distributed over different processors using the message passing interface (MPI), and the Jacobian is then assembled and sent to the optimizer.

3.5 Optimization History Class

When any of the `Optimizer` instances are called to solve an optimization problem, an option to store the solution history can be used. This initializes an instance of the `History` class, which has two purposes. The first is to store all the data associated with each call to the objective function. The data consists of the values for the design variables, objective, constraints, and if applicable, the gradients. The `History` class opens two files when initialized: a binary file with the the actual data and an ASCII file that stores the cues to that data. The data is flushed immediately to the files at each write call.

The second purpose of the `History` class is to allow the warm-restart of a previously interrupted optimization, even when the actual optimization package does not support it. The approach works for any deterministic optimization algorithm and relies on the fact that any deterministic algorithm will follow exactly the same path when starting from the same point. If the history file exists for previous a optimization that finished prematurely for some reason (due to a time limit, or a convergence tolerance that was set to high), pyOpt can restart the optimization using that history file to provide the optimizer with the objective and constraint values for all the points in the path that was previously followed. Instead of recomputing the function values at these points, pyOpt provides the previously computed values until the end of the history. After the end of the history has been reached, the optimization continues with the new part of the path.

To allow quick access, the cue file is read in at the initialization of the class. The position and number-of-values cues are then used to read in the required values, and only those values, from the binary file. The optimizer can be called with options for both storing a history and reading in a previous history. In this case two instances of the history class are initialized, one in write mode and one in read mode. If the same name is used in both history files, the history instance in read mode is only maintained until all its history data has been read and used by the optimizer.

4 Examples

We illustrate the capabilities of pyOpt by solving four different optimization problems. The first three problems involve explicit analytic formulations, while the last problem is an engineering design example involving more complex numerical simulation.

4.1 Problem 1

This first example demonstrates the flexibility enabled by maintaining independence between the optimization problem and the optimization solver. The problem is taken from the set of nonlinear programming examples by Hock and Schittkowski (Hock and Schittkowski 1981) and it is defined as,

$$\begin{aligned} \min_{x_1,x_2,x_3} \quad & -x_1x_2x_3 \\ \text{subject to} \quad & x_1 + 2x_2 + 2x_3 - 72 \leq 0 \\ & -x_1 - 2x_2 - 2x_3 \leq 0 \\ & 0 \leq x_1 \leq 42 \\ & 0 \leq x_2 \leq 42 \\ & 0 \leq x_3 \leq 42. \end{aligned} \tag{2}$$

The optimum of this problem is at $(x_1^*, x_2^*, x_3^*) = (24, 12, 12)$, with an objective function value of $f^* = -3456$, and constraint values $g(x^*) = (0, -72)$.

Figure 2 lists the source code for this example using two optimizers. There are four major sections in the code:

1. Import of the modules containing the optimization problem and the solver classes
2. Definition of the objective function
3. Instantiation of the optimization problem object and definition of the problem design variables, objective, objective function, and constraints
4. Instantiation of the optimization solver objects, setting of the solver options, and solution of the problem by the solvers

Each optimizer is instantiated with a set of default options that work in most cases. This way, we maximize the likelihood of success for less experienced optimization practitioners. On the other hand, users that have experience with a given optimizer can easily modify the default set of options. In Figure 2 for example, the first optimizer makes use of its default derivative estimation (finite differences), while the second optimizer makes use of the complex-step method (set by the "`CS`" input flag). The outputs of the solution by the two sample solvers are shown in Figure 3. Since all solvers share the same `Optimizer` abstract class, the output format of the results is standardized to facilitate interpretation and comparison.

Since complete independence between the optimization problem and the solvers is maintained, it is easy to solve the same optimization problem instance with different optimizers. For example, Table 1 and Figure 4 show a comparison of the different optimization approaches currently implemented into the framework for

```python
# Import Framework Module
from pyOpt_optimization import Optimization

# Import Framework Optimizers
from pySNOPT import SNOPT
from pyNLPQL import NLPQL

# Define Optimization Problem Objective Function
def objfunc(x):
    f = -x[0]*x[1]*x[2]
    g = [0.0]*2
    g[0] = x[0] + 2.*x[1] + 2.*x[2] - 72.0
    g[1] = -x[0] - 2.*x[1] - 2.*x[2]
    fail = 0
    return f,g,fail

# Instantiate Optimization Problem
opt_prob = Optimization('Hock and Schittkowski NLP Problem',
                        objfunc)

# Define Problem Design Variables (3 Continuous Variables)
opt_prob.addVarGroup('x',3,'continuous',lower=0.0,
                     upper=42.0, value=10.0)

# Define Problem Objective Function Variable
opt_prob.addObj('f')

# Define Problem Constraint Variables
# (2 Inequality Constraints)
opt_prob.addConGroup('g',2,'inequality')

# Display Defined Optimization Problem
print opt_prob

# Instantiate Optimizers
snopt = SNOPT()
nlpql = NLPQL()

# First Optimizer - Set Option and Solve Problem
# (Finite Differences)
snopt.setOption('Major feasibility tolerance',1.0e-6)
snopt(opt_prob)
print opt_prob.solution(0)

# Second Optimizer - Set Option and Solve Problem
# Complex-Step)
nlpql.setOption('Accuracy',1.0e-6)
nlpql(opt_prob,'CS')
print opt_prob.solution(1)
```

Fig. 2 Python source code that implements Problem 1

```
SNOPT Solution to Hock and Schittkowski NLP Problem
======================================================================
        Objective Function: objfunc

    Solution:
----------------------------------------------------------------------
    Total Time: 0.0310
    Total Function Evaluations: 13
    Lambda: [-144.0000003 0.0]
    Sensitivities: FD

    Objectives:
        Name        Value
        f       -3456.000000
    Variables (c - continuous, i - integer, d - discrete):
        Name    Type        Value       Lower Bound  Upper Bound
        x1       c       23.999996       0.00e+000    4.20e+001
        x2       c       12.000001       0.00e+000    4.20e+001
        x3       c       12.000001       0.00e+000    4.20e+001
    Constraints (e - equality, i - inequality):
        Name    Type                    Bound
        g1       i      -1.00e+021 <=   0.000000 <= 0.00e+000
        g2       i      -1.00e+021 <= -72.000000 <= 0.00e+000
----------------------------------------------------------------------

NLPQL Solution to Hock and Schittkowski NLP Problem
======================================================================
        Objective Function: objfunc

    Solution:
----------------------------------------------------------------------
    Total Time: 0.0160
    Total Function Evaluations: 8
    Lambda: [-144.00 0.0]
    Sensitivities: CS

    Objectives:
        Name        Value
        f       -3456.000000
    Variables (c - continuous, i - integer, d - discrete):
        Name    Type        Value       Lower Bound  Upper Bound
        x1       c       24.000003       0.00e+000    4.20e+001
        x2       c       12.000000       0.00e+000    4.20e+001
        x3       c       11.999999       0.00e+000    4.20e+001
    Constraints (e - equality, i - inequality):
        Name    Type                    Bound
        g1       i      -1.00e+021 <=   0.000000 <= 0.00e+000
        g2       i      -1.00e+021 <= -72.000000 <= 0.00e+000
----------------------------------------------------------------------
```

Fig. 3 Solution output of optimization Problem 1 for two different optimizers: SNOPT and NLPQL

Problem 1. All optimizers started from the same initial point specified in the code (Figure 2). The convergence criteria and settings for each optimizer vary, so we tried to achieve the same level of convergence tolerance. Gradient-based methods calculated sensitivities using the default finite-difference approach with a step size of 10^{-6}. Since an analytic solution can be obtained for this example, the optimum solutions returned by the different optimizers were compared to the exact one. In Table 1, the comparison is made using an l_2-norm of the difference between the optimal design variables, the objective function and constraint values, and their corresponding exact values, i.e.,

$$\begin{aligned} \bar{\varepsilon}_x &= \|x^* - x_{\text{exact}}\|_2 \\ \bar{\varepsilon}_f &= \|f(x^*) - f(x_{\text{exact}})\|_2 \\ \bar{\varepsilon}_g &= \|g(x^*) - g(x_{\text{exact}})\|_2 \end{aligned} \tag{3}$$

Similarly, the convergence histories for the various optimizers are shown in Figure 4. The comparison metric used here is the relative error of the objective function values, i.e.,

$$\varepsilon_f = \left\| \frac{f(x^*) - f(x_{\text{exact}})}{f(x_{\text{exact}})} \right\|_2 \tag{4}$$

In Table 1 and Figure 4, we can see that for this convex problem, the four SQP optimizers (SNOPT, NLPQL, SLSQP, and FSQP) provide the most accurate solutions for the specified convergence tolerance. The FSQP optimizer requires the largest number of evaluations of all SQP approaches, since it generates a feasible point at each iteration before solving the SQP subproblem. Similarly to the SQP optimizers, the modified method of feasible directions (CONMIN) and both versions of the method of moving asymptotes (MMA and GCMMA) provide accurate solutions with a low number of function evaluations. The other local gradient-based optimizer (KSOPT) solves the problem using a SUMT approach with a composite objective function, leading to more function evaluations and reduced accuracy in the solution. The gradient-free locally constrained optimizers (COBYLA, and SOLVOPT) require a larger number of function evaluations relative to the gradient-based optimizers. As expected, the gradient-free constrained global optimization approaches take a significantly larger number of function evaluations. However it is worth noting that ALPSO finds the optimum solution with a good degree of accuracy, while requiring a significantly fewer evaluations than other gradient-free algorithms.

Table 1 Comparison of solutions for Problem 1

Solver	fevals	ε_f	$\bar{\varepsilon}_f$	$\bar{\varepsilon}_x$	$\bar{\varepsilon}_g$
SNOPT	26	2.8948e-14	6.1845e-11	3.9170e-6	4.4409e-16
NLPQL	19	2.8948e-14	2.0171e-7	1.3159e-4	8.5650e-10
SLSQP	32	2.8948e-14	7.1917e-6	7.2722e-8	7.0629e-8
FSQP	112	2.2204e-16	1.3642e-12	1.8364e-8	4.4409e-16
CONMIN	29	2.2204e-16	3.5016e-11	6.4512e-8	3.4165e-13
MMA	13	2.2204e-16	4.0164e-8	1.7463e-6	3.9456e-10
GCMMA	14	2.2204e-16	1.3816e-6	3.0437e-4	1.7208e-8
KSOPT	2648	6.6359e-4	9.4230e-1	2.8081e-3	9.2547e-3
COBYLA	112	2.2204e-16	6.3664e-12	7.5885e-7	4.4409e-16
SOLVOPT	201	2.7777e-7	4.8339e-5	1.3612e-3	3.4287e-7
ALPSO	4600	2.3148e-7	7.9805e-4	3.9254e-3	6.7994e-6
NSGA2	150000	1.0602e-3	3.6636	5.5472e-1	2.4889e-6
ALHSO	39384	9.6354e-4	3.3253	8.5523e-1	8.9124e-5
MIDACO	15000	1.6297e-2	5.6322e+1	3.8072	1.4091e-4

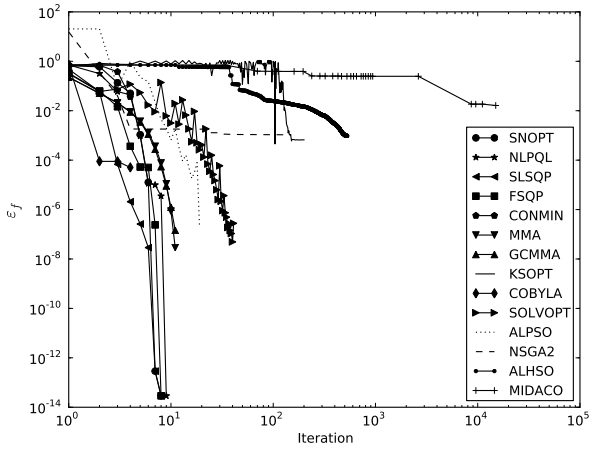


Fig. 4 Comparison of convergence histories for Problem 1

4.2 Problem 2

In this example, we show how optimizer instances can be reused multiple times to solve different optimization problems. To illustrate this capability, we analyze the effect of changing the number of design variables for the various gradient-based optimizers currently available in pyOpt. The optimization problem is the classic generalized Rosenbrock's function (Rosenbrock 1960; De Jong 1975; Schittkowski 1987) modified by the authors to include an n-dimensional constraint, i.e.,

$$\begin{aligned} \min_{x_i} \quad & \sum_{i=1}^{n-1} \left(100({x_i}^2 - x_{i+1})^2 + (1 - x_i)^2\right) \\ \text{subject to} \quad & \sum_{i=1}^{n-1} \Big(0.1 - (x_i - 1)^3 - (x_{i+1} - 1)\Big) \leq 0 \\ & -5.12 \leq x_i \leq 5.12, \quad i = 1, \ldots, n \end{aligned} \tag{5}$$

where the constraint is active at the optimum.

Similarly to the first example, all optimizers started from the same initial point, $(x_1, \ldots, x_n) = (4, \ldots, 4)$, and use their default options. Sensitivities were computed using the complex-step derivative approximation with a step size of 10^{-20}. Figures 5 and 6 show a comparison of the relative error in the optimum objective function (4) and in the optimum design variable values for the different optimizers versus the number of design variables in the problem. Comparisons of all gradient-based and gradient-free optimizers are made with respect to the solution obtained by a reference optimizer (SNOPT). Similarly, Figure 7 shows a comparison of the total number of objective function evaluations requested by each optimizer for increasing dimensionality of the problem described by Equation (5). For gradient-based optimizers, analytic gradients are provided with each function evaluation. All optimizers find the problem optimum while maintaining a good degree of accuracy. The variation in accuracy between the different results is due to the difference in the convergence criteria used by each optimizer. All SQP based optimizers, NLPQL, SLSQP and FSQP, provide good solution accuracy when compared to SNOPT, while maintaining the same degree of relative error at higher dimensionality. This comes at the expense of larger number of function evaluations as the dimensionality increases. The sequential linear approach used by the SLSQP optimizer starts to suffer from degraded convergence accuracy beyond a dimensionality of 100. The CONMIN optimizer seems to be less accurate when compared to SNOPT, with a significant loss of accuracy for more than 50 design variables. For both moving asymptote based algorithms (MMA and GCMMA), a consistent level of accuracy per required number of evaluations can be observed regardless of the problem dimensionality. In the case of KSOPT, accuracy of the solution seems

to improve slightly with increased problem dimensionality. However, the number of required function evaluations increases significantly. For lower problem dimensionality, both gradient-free optimizers (COBYLA, SOLVOPT) maintain a constant number of evaluations and accuracy level until the problem dimensionality reaches 50, above which significant accuracy degradation and increase in number of evaluations can be observed. When compared to other accurate optimizers, SNOPT shows a less pronounced increase in number function evaluations versus dimensionality.

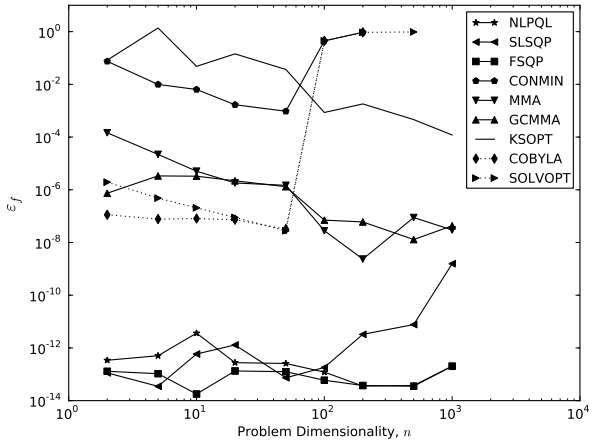


Fig. 5 Objective function accuracy versus dimensionality of Problem 2

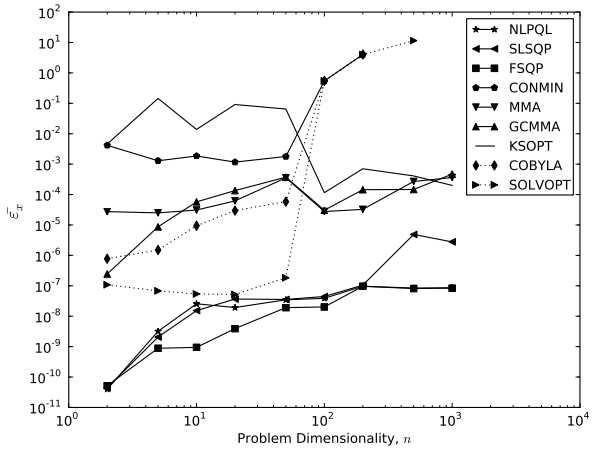


Fig. 6 Design variable accuracy versus dimensionality of Problem 2

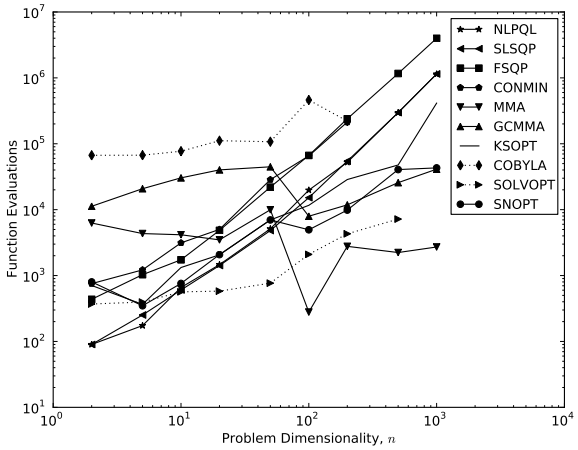
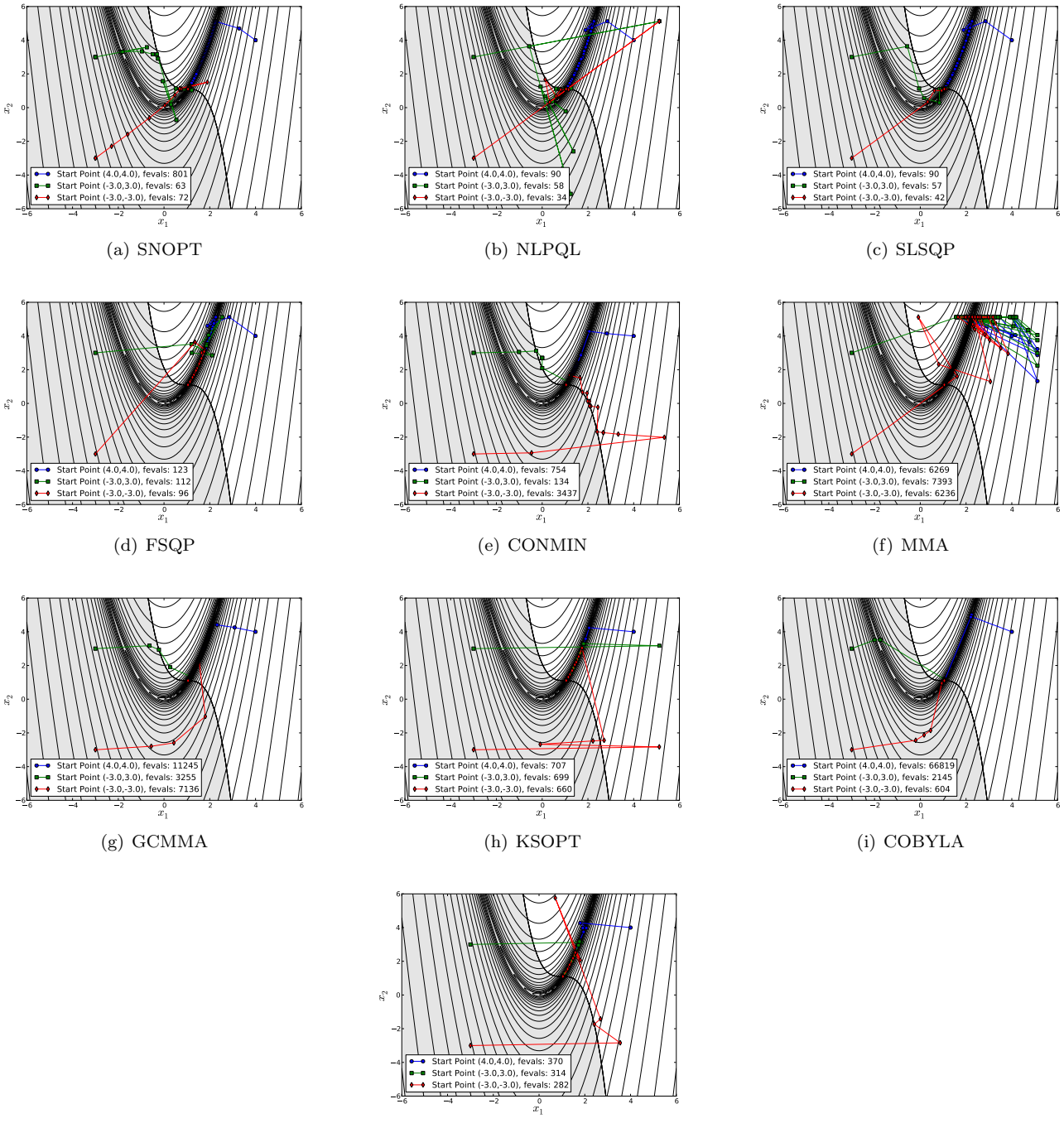


Fig. 7 Number of function evaluations versus dimensionality of Problem 2

4.2.1 Optimizers Paths

The behavior of different optimizers when solving Problem (5) in two dimensions is shown in Figure 8. The optimization paths of each optimizer for three different starting points are shown along with the total number of function evaluations.

All SQP approaches (SNOPT, NLPQL, SLSQP, and FSQP) follow similar paths. When starting rom the feasible starting point, they follow the narrow valley of the Rosenbrock function toward the optimum. Wen starting from the infeasible starting points, both SNOPT and NLPQL show backtracking, with NLPQL moving to the upper bound of the design variables in its first step. The FSQP algorithm moves into the feasible design space in its first iteration since it enforces feasibility at each iteration. From all three starting points, MMA quickly moves towards the design variable upper bounds, and then slowly follows the function valley. For all three starting points, the initial steps of GCMMA and CONMIN follow a path that seems influenced by the steepest descent direction at the starting point. For the infeasible starting points, CONMIN rapidly reaches the feasible region and then moves near the edge of the constraint until it reaches the function valley, where it slowly descends towards the optimum. When starting from the feasible starting point, KSOPT, COBYLA, and SOLVOPT follow the narrow valley toward the optimum. COBYLA requires the largest number of function evaluations. When starting from the infeasible starting points, KSOPT and SOLVOPT do some backtracking, while COBYLA follows a similar descent direction toward the optimum as CONMIN and GCMMA.



(a) SNOPT (b) NLPQL (c) SLSQP

(d) FSQP (e) CONMIN (f) MMA

(g) GCMMA (h) KSOPT (i) COBYLA

(j) SOLVOPT

Fig. 8 Optimizer paths for Problem 2 with $n = 2$

4.3 Problem 3

In this example, we show how the object-oriented design of pyOpt can be used to enhance the capabilities available to the optimization practitioner. Many optimization problems have multiple local minima and non-convex design spaces. Consider for example, the following modification of the unconstrained problem originally defined by Bersini et al. (Bersini et al 1996), to

which we added a constraint,

$$\min_{x_1,x_2} -\sum_{k=1}^{5} c_k \exp\left(-\frac{1}{\pi}\sum_{i=1}^{2}(x_i - a_{ki})^2\right)\cos\left(\pi\sum_{i=1}^{2}(x_i - a_{ki})^2\right)$$
$$\text{s.t.}\begin{cases} 20.04895 - \left((x_1+2)^2 + (x_1+1)^2\right) \le 0 \\ -2 \le x_1 \le 10 \\ -2 \le x_2 \le 10, \end{cases} \tag{6}$$

where,

$$a = \begin{bmatrix} 3 & 5 & 2 & 1 & 7 \\ 5 & 2 & 1 & 4 & 9 \end{bmatrix} \tag{7}$$
$$c = \begin{bmatrix} 1 & 2 & 5 & 2 & 3 \end{bmatrix}$$

Figure 9 shows the unconstrained design space for this problem. A large number of local optima exists and the global optimum is at $(x_1^*, x_2^*) = (2.003, 1.006)$, where the function value is $f^* = -5.1621$ and the constraint is active.

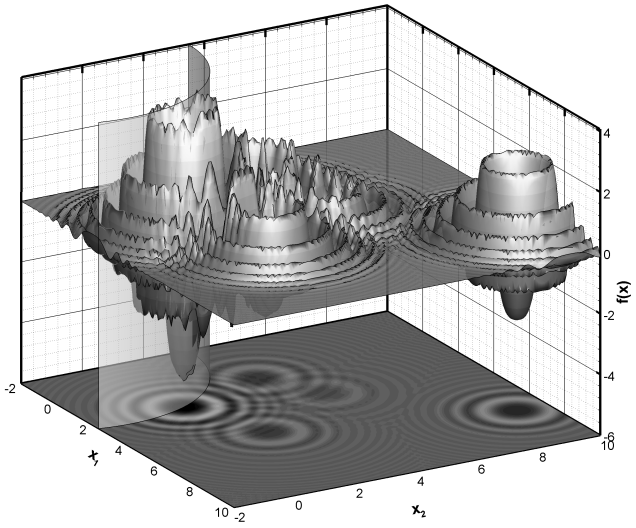


Fig. 9 Contour plot for optimization Problem 3, showing the inequality constraint.

The utility of gradient-based optimizers for this type of problem is severely limited. The strategy we used to solve this type of problem is to find the probable global minimum using constrained global optimizers such as ALPSO and NSGA2, and then refine the solution by using a gradient-based optimizer. With most optimization software, this refinement requires the preparation of two optimization inputs for the two optimizers, and extensive scripting. In a typical scenario for the second optimizer that provides a solution refinement, the optimum values obtained by the first optimizer have to be manually processed and used to produce a new input file. With pyOpt, the optimization solution is inherited directly from the optimization problem class, sharing all its attributes and properties. Inheritance allows us to directly pass the solution of one optimizer and make it the initial point of another. Figure 10 shows the source code that implements this strategy for the solution of the problem stated in Equation (6). Following the class inheritance, the refinement results are stored as a solution of the first optimizer. With this nesting of solutions, multiple levels of refinement are possible.

```
# Import Framework Module
from pyOpt_optimization import Optimization

# Import Framework Optimizers
from pyALPSO import ALPSO
from pySNOPT import SNOPT

# Define Optimization Problem Objective Function
def objfunc(x):
    a = [3,5,2,1,7]
    b = [5,2,1,4,9]
    c = [1,2,5,2,3]
    f = 0.0
    for i in xrange(5):
        f += -(c[i]*exp(-(1/pi)*((x[0]-a[i])**2 + \
              (x[1]-b[i])**2))*cos(pi*((x[0]-a[i])**2 + \
              (x[1]-b[i])**2)))
    #end
    g = [0.0]*1
    g[0] = 20.04895 - (x[0]+2.0)**2 - (x[1]+1.0)**2
    fail = 0
    return f,g,fail

# Instanciate Optimization Problem
opt_prob = Optimization('Langermann Multimodal Problem',
                        objfunc)

# Define Problem Design Variables (3 Continuous Variables)
opt_prob.addVarGroup('x',2,'continuous',lower=-2.0,
                     upper=10.0)

# Define Problem Objective Function Variable
opt_prob.addObj('f')

# Define Problem Constraint Variables
# (1 Inequality Constraint)
opt_prob.addCon('g','inequality')

# Display Defined Optimization Problem
print opt_prob

# Instanciate Optimizers
alpso = ALPSO()
snopt = SNOPT()

# Solve Problem with Constrained Global Optimizer
alpso(opt_prob)
print opt_prob.solution(0)

# Refine Solution Using Local Gradient-Based Optimizer
snopt(opt_prob.solution(0))
print opt_prob.solution(0).solution(0)
```

Fig. 10 Python source code that implements the automatic refinement for Problem 3

Table 2 and Figure 11 show a comparison of two optimization refinement cases used to solve Problem 3. The first case, whose source code is listed in Figure 10, uses the ALPSO as the gradient-free global optimizer, while the second case uses NSGA2. Both cases are automatically refined in the second stage using SNOPT. Both ALPSO and NSGA2 use the same population size (40). To maintain uniformity in the comparison, the total number of generations allowed for the NSGA2 is specified so that the number of function evaluations match those required by the ALPSO algorithm to find

a solution. As seen in the results, both gradient-free global optimizers are able to successfully find the region containing the global minimum. The ALPSO optimizer provides a solution with lower absolute error than NSGA2. SNOPT is able to further refine the solution obtained and achieve the required accuracy in both cases. Note that the solution refinement is not limited to the strategy we used. Different refinement strategies are possible, combining any of the optimizers in pyOpt.

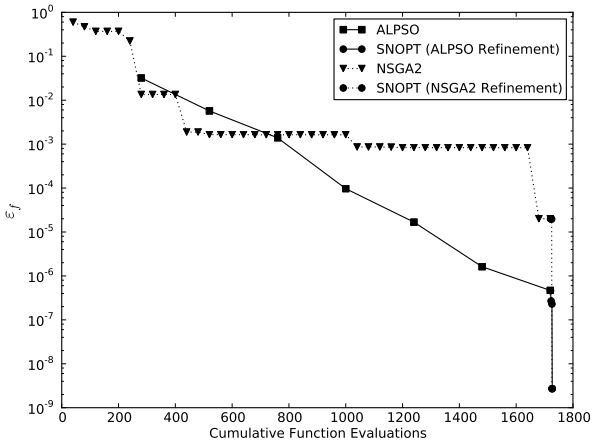


Fig. 11 Convergence histories for the automatic refinement solution of Problem 3

4.4 Problem 4

In this final example, the capability to parallelize the gradient computation is shown in the aerostructural design of the wing for a medium-range commercial aircraft (Jansen et al 2010). Medium-fidelity aerodynamic and structural analyses are used (Chittick and Martins 2008; Poon and Martins 2007). Aerodynamic forces and moments are computed by a vortex-lattice panel method with added estimates for viscous drag and transonic compressibility effects (Jansen et al 2010). The panel method forms and solves the following system:

$$A\Gamma - b = 0, \tag{8}$$

where A is the aerodynamic influence coefficient matrix, Γ is the vector of panel circulations, and b is the boundary condition vector given by the flight conditions and the wing twist distribution. Structural deflections, stresses, and weight are computed using an equivalent beam finite-element model of the aircraft wingbox. The structural analysis is governed by the following equation:

$$Ku - f = 0, \tag{9}$$

where K is the stiffness matrix of the structure, u is the nodal displacement vector, and f is the vector of external nodal forces.

These two systems of equations are coupled through b, which depends on u, and through f, which is a function of Γ. The simultaneous solution of the linear systems (8) and (9), which defines the state of the aerostructural system, is obtained by a block Gauss–Seidel iteration. A more detailed description of these models is provided in Jansen et al. (Jansen et al 2010).

The goal of the optimization is to minimize the takeoff weight, W_{TO}, while ensuring that the aircraft meets a specified design range and making sure that the structural stresses at the maneuver condition does not exceed the yield stress. The aerostructural optimization problem as shown in Figure 12 is formulated as follows:

$$\min_{\alpha,\alpha_{\mathrm{mvr}},\gamma,t_j} W_{\mathrm{TO}}$$
$$\text{s.t.} \begin{cases} L = W_i \\ \dfrac{C_{L_{\mathrm{mvr}}}}{C_{L_{\mathrm{cruise}}}} = 2.5 \\ 1.5\sigma_j|_{C_{L_{mvr}}} \leq \sigma_{\mathrm{yield}} \\ -15 \leq \alpha \leq 15 \\ -15 \leq \gamma \leq 15 \\ 0.06 \leq t_j \leq R_j \end{cases} \tag{10}$$

where α and α_{mvr} are the aircraft angles of attack at the cruise and maneuver conditions, respectively, and γ is the tip twist angle. The wall thicknesses of the beam elements are represented by t_j, where $j = 1, \ldots, n_{\mathrm{elems}}$. These thicknesses are prevented from exceeding the radius of the beam, R_j, and have a lower bound corresponding to the minimum gauge thickness. The aircraft weight depends on the structural weight of the wing and the weight of the fuel required to meet the specified range. The lift is constrained to match the weight of the aircraft at cruise, W_i. The stresses of the beam elements are calculated at the structurally critical 2.5g pull-up maneuver condition and are constrained to be equal to or lower than the yield stress of the material with a 1.5 safety factor.

The maximum number of processors that can be used in the parallel gradient computation depends on the number of design variables in the optimization problem. This is due to the processor management approach, which statically allocates the gradient computation of each individual design variable to the available processors. The aerostructural optimization problem is solved using SNOPT, with gradients computed in parallel by a SiCortex SC072-PDS computer. The wing is discretized using 29 elements resulting in 32 design variables, and hence a maximum of 32 processors are used. The op-

Table 2 Automatic refinement solutions for optimization Problem 3

Solver	fevals	ε_f	$\bar{\varepsilon}_f$	$\bar{\varepsilon}_x$	$\bar{\varepsilon}_g$
Case 1					
ALPSO	1720	4.6749e-7	2.4132e-6	3.2809e-4	1.8079e-3
SNOPT Refinement	8	2.8014e-12	1.4461e-11	1.9884e-8	4.4409e-16
Case 2					
NSGA2	1720	1.9971e-5	1.0309e-4	2.3613e-3	2.1151e-2
SNOPT Refinement	8	7.5309e-13	3.8876e-12	1.2599e-9	4.4409e-16

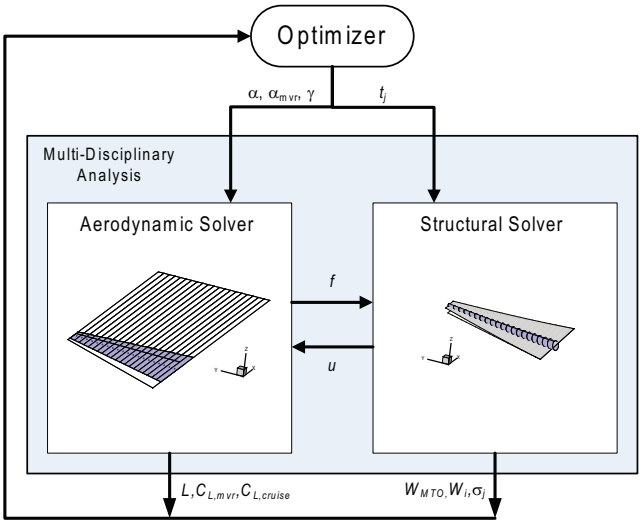
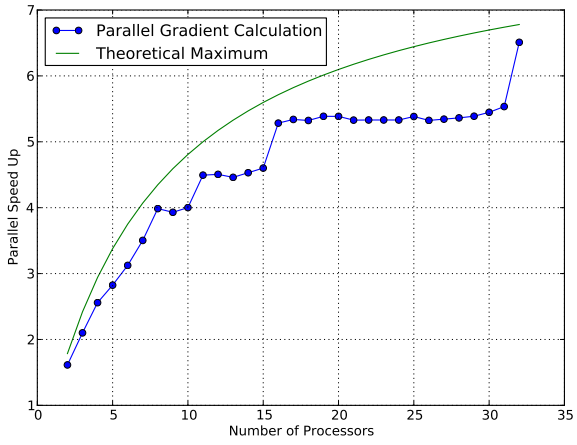
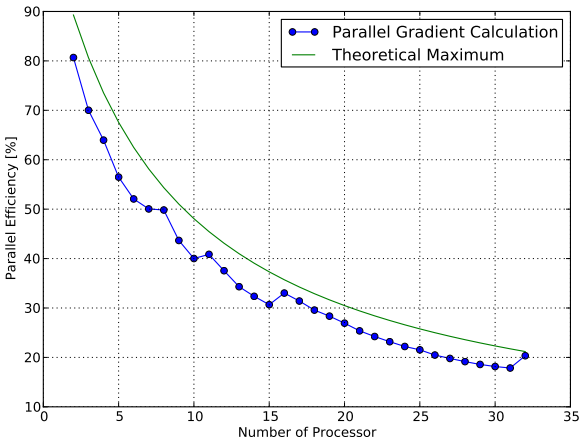


Fig. 12 Aerostructural design optimization problem



(a) Parallel Speed-Up



(b) Parallel Efficiency

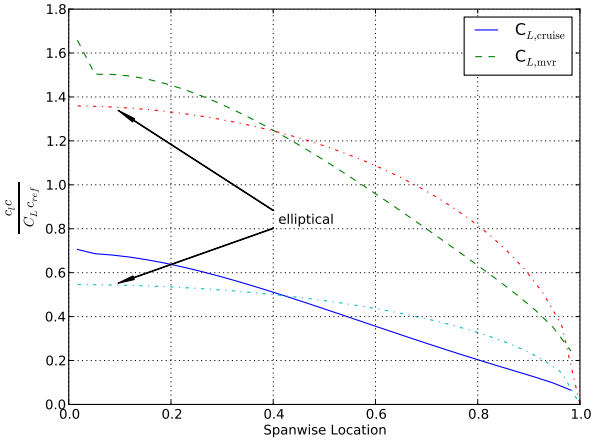
Fig. 13 Parallel gradient calculations performance for Problem 4

timization parallel gradient implementation speed-up and efficiencies are shown in Figures 13(a) and 13(b), respectively.

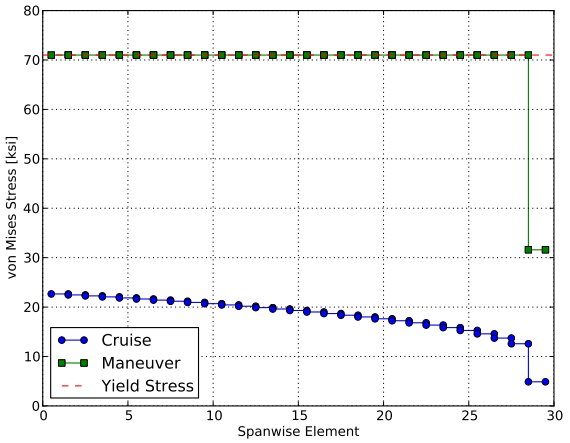
The ideal speed-up and efficiency of a parallel computation are given by Amdahl's law. The portion of the algorithm that can be parallelized, which corresponds to the gradient computation, was estimated to be 89% of the total execution time. Due to the static processor allocation, the efficiency of the parallel gradient calculation is significantly reduced when the number of design variables is not an integer multiple of the number of processors used. This results in the steps exhibited by the speed-up graph. For 8, 12, 16, and 32 processors the speed-up and efficiency is close to the theoretical maximum, while between those numbers, the increase in processors provides little improvement. This trend becomes even more pronounced as the number of processors increases. Beyond 16 processors, the performance deteriorates with increasing number of processors up to 32 processors, where performance closely matches the theoretical maximum again.

The optimal lift and stress distributions obtained for the aerostructural optimization problem are shown in Figures 14(a) and 14(b), respectively. The optimal lift distribution shows the shift of the lift towards the root, which alleviates the bending moment in the wing and enables a reduction in structural weight. This shift in loading is slightly more pronounced in the structurally critical maneuver condition. At the cruise condition, a more elliptical distribution is favored in order to im-

prove the aerodynamic performance. The elements of the beam are fully stressed at the maneuver condition to minimize the structural weight.



(a) Lift Distribution



(b) Stress Distribution

Fig. 14 Results for Problem 4

5 Summary

In this article, we presented pyOpt: a flexible object-oriented framework developed in Python to solve nonlinear constrained optimization problems using a number of existing optimization packages. The framework design makes extensive use of object-oriented features, such as abstract classes, common interfaces, and logical hierarchical structures. This enables consistent and intuitive constructs that help in the solution of optimization problems and the use of different optimization algorithms. Since the optimization problem is kept separate from the optimization algorithms, a large degree of flexibility is provided to both users and developers alike. Users and developers can currently take advantage of the solvers already integrated into the framework, as well as history storage, warm-restart, and parallel gradient computation capabilities. The four example problems demonstrate the simplicity and power of pyOpt in formulating, solving, and refining optimization problems, as well as in using and comparing optimization solvers. While pyOpt is still in development, it will be made available as an open source project that provides a simple but powerful platform for optimization.

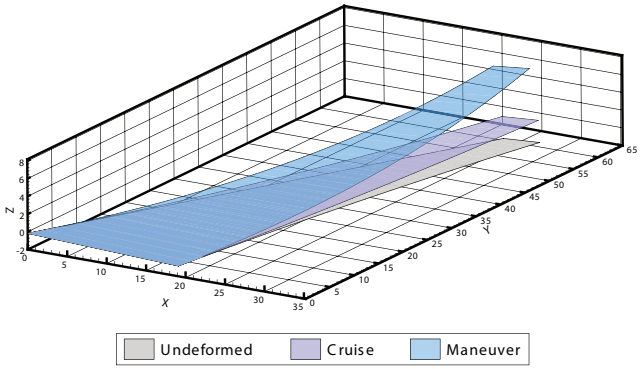


Fig. 15 Optimized wing geometry for Problem 4, showing jig shape and deformed shapes for both the cruise and maneuver conditions, with dimensions in feet.

References

(2007) TANGO Project: Trustable Algorithms for Nonlinear General Optimization. URL http://www.ime.usp.br/ egbirgin/-tango/

Arlow J, Neustadt I (2002) UML and the Unified Process: Practical Object-Oriented Analysis and Design. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA

Beazley DM (2006) Python Essential Reference, 3rd edn. Sams Publishing, Indianapolis, IN

Bersini H, Dorigo M, Langerman S, Geront G, Gambardella L (1996) Results of the first international contest on evolutionary optimisation (1st ICEO). In: IEEE International Conference on Evolutionary Computation, pp 611–615

Bisschop J, Roelofs M (2008) AIMMS — User's guide. Tech. rep., Paragon Decision Technology, Haarlem, The Netherlands

Blezek D (1998) Rapid prototyping with SWIG. C/C++ Users Journal 16(11):61–65

Chittick IR, Martins JRRA (2008) Aero-structural optimization using adjoint coupled post-optimality sensitivities. Structural and Multidisciplinary Optimization 36(1):59–77

Dahl J, Vandenberghe L (2008) CVXOPT: Python Software for Convex Optimization, Documentation. URL http://abel.ee.ucla.edu/cvxopt/

De Jong KA (1975) An analysis of the behavior of a class of genetic adaptive systems. PhD thesis, University of Michigan

Deb K, Pratap A, Agarwal S, Meyarivan T (2002) A fast and elitist multiobjective genetic algorithm: NSGA-II. Evolution-

ary Computation, IEEE Transactions on 6(2):181–197, DOI 10.1109/4235.996017

Eldred MS, Brown SL, Adams BM, Dunlavy DM, Gay DM, Swiler LP, Giunta AA, Hart WE, Watson JP, Eddy JP, Griffin JD, Hough PD, Kolda TG, Martinez-Canales ML, Williams PJ (2007) DAKOTA, a multilevel parallel object-oriented framework for design optimization, parameter estimation, uncertainty quantification, and sensitivity analysis: Version 4.0 users manual. Technical Report SAND 2006-6337, Sandia National Laboratories

Fiacco AV, McCormick GP (1968) Nonlinear Programming: Sequential Unconstrained Minimization Techniques. John Wiley, New York

Fourer R, Gay DM, Kernighan BW (2003) AMPL: A Modelling Language for Mathematical Programming, 2nd edn. Brooks/Cole–Thomson Learning, Pacific Grove, CA

Friedlander M, Orban D (2008) NLpy: Nonlinear Programming in Python. URL http://nlpy.sourceforge.net/index.html

Geem ZW, Kim JH, Loganathan GV (2001) A new heuristic optimization algorithm: Harmony search. Simulation 76:60–68, DOI 10.1177/003754970107600201

Gill PE, Murray W, Saunders MA (2002) SNOPT: An SQP algorithm for large-scale constrained optimization. SIAM Journal on Optimization 12:979–1006, DOI 10.1137/S0036144504446096

Hart W (2009) Operations Research and Cyber-Infrastructure, Springer, chap Python Optimization Modeling Objects (Pyomo), pp 3–19

Hock W, Schittkowski K (1981) Test Examples for Nonlinear Programming Codes, Lecture Notes in Economics and Mathematical Systems, vol 187. Springer

Holmström K, Göran A, Edvall M (2010) User's Guide for TOMLAB 7. TOMLAB Optimization AB, URL http://www.tomlab.biz

Jacobs J, Etman L, van Keulen F, Rooda J (2004) Framework for sequential approximate optimization. Structural and Multidisciplinary Optimization 27:384—400, DOI 10.1007/s00158-004-0398-8

Jansen P, Perez R, Martins J (2010) Aerostructural optimization of nonplanar lifting surfaces. AIAA Journal of Aircraft 47(5):1490–1503, DOI 10.2514/1.44727

Jones E, Oliphant T, Peterson P, et al (2001) SciPy: Open Source Scientific Tools for Python. URL http://www.scipy.org

Kiwiel KC (1985) Methods of descent for nondifferentiable optimization. In: Lecture Notes in Mathematics, vol 1133, Springer-Verlag, Berlin

Kraft D (1988) A software package for sequential quadratic programming. Tech. Rep. DFVLR-FB 88-28, DLR German Aerospace Center — Institute for Flight Mechanics, K oln, Germany

Kreisselmeier G, Steinhauser R (1979) Systematic control design by optimizing a vector performance index. In: IFAC Symposium on Computer-Aided Design of Control Systems, International Federation of Active Controls, Zurich, Switzerland

Kroshko DK (2010) OpenOpt. URL http://openopt.org

Kuntsevich A, Kappel F (1997) SolvOpt manual: The solver for local nonlinear optimization problems. Tech. rep., Institute for Mathematics, Karl-Franzens University of Graz, Graz, Autria

Langtangen HP (2008) Python Scripting for Computational Science, Texts in Computational Science and Engineering, vol 3, 3rd edn. Springer

Lawrence CT, Tits AL (1996) Nonlinear equality constraints in feasible sequential quadratic programming. Optimization Methods and Software 6:265–282

Lawson CL, Hanson RJ (1974) Solving Least Square Problems. Prentice-Hall, Englewood Cliffs, N.J.

Lee KS, Geem ZW (2005) A new meta-heuristic algorithm for continuous engineering optimization: Harmony search theory and practice. Computer Methods in Applied Mechanics and Engineering 194:3902–3933, DOI 10.1016/j.cma.2004.09.007

Lofberg J (2004) YALMIP: A toolbox for modeling and optimization in MATLAB. In: CACSD Conference, Taipei, Taiwan

Martins JRRA, Sturdza P, Alonso JJ (2003) The complex-step derivative approximation. ACM Transactions on Mathematical Software 29(3):245–262, DOI http://doi.acm.org/10.1145/838250.838251

Meza JC (1994) OPT++: An object oriented class library for nonlinear optimization. Technical Report SAND 1994-8225, Sandia National Laboratories

Meza JC, Oliva RA, Hough PD, Williams PJ (2007) OPT++: An object oriented toolkit for nonlinear optimization. ACM Transactions on Mathematical Software 33(2):12:1–12:27, DOI 10.1145/1236463.1236467

Mitchell S (2009) puLP: An LP modeler in Python, Documentation. URL https://www.coin-or.org/PuLP/

Moré JJ, Wright SJ (1993) Optimization Software Guide. SIAM Publications

Oliphant TE (2007) Python for scientific computing. Computing in Science and Engineering 9(3):10–20

Perez R, Behdinan K (2007) Swarm Intelligence: Focus on Ant and Particle Swarm Optimization, 1st edn, International Journal of Advanced Robotic Systems, chap Particle Swarm Optimization in Structural Design, pp 373–394. ISBN 978-3-902613-09-7

Peterson P, Martins JJRA, Alonso JJ (2001) Fortran to Python interface generator with an application to aerospace engineering. In: 9th International Python Conference, Long Beach, California

Poon NMK, Martins JRRA (2007) An adaptive approach to constraint aggregation using adjoint sensitivity analysis. Structural and Multidisciplinary Optimization 30(1):61–73

Powell MJD (1994) Advances in Optimization and Numerical Analysis, Kluwer Academic, Dordrecht, chap A direct search optimization method that models the objective and constraint functions by linear interpolation, pp 51–67

Rosenbrock HH (1960) An automatic method for finding the greatest or least value of a function. Computer Journal 3:175–184

Rosenthal RE (2008) GAMS — A user's guide. Tech. rep., GAMS Development Corporation, Washington, DC, USA

Schittkowski K (1986) NLPQL: A Fortran subroutine for solving constrained nonlinear programming problems. Annals of Operations Research 5(2):485–500

Schittkowski K (1987) More test problems for nonlinear programming codes. Lecture Notes in Economics and Mathematical Systems 282

Schlüter M, Gerdts M (2009) The oracle penalty method. Journal of Global Optimization 47(2):293–325, DOI 10.1007/s10898-009-9477-0

Schlüter M, Egea J, Banga J (2009) Extended ant colony optimization for non-convex mixed integer nonlinear programming. Computers and Operations Research 36(7):2217–2229

Shor N (1985) Minimization methods for non-differentiable functions. In: Springer Series in Computational Mathematics, vol 3, Springer-Verlag, Berlin

Svanberg K (1987) The method of moving asymptotes — A new method for structural optimization. International Journal for Numerical Methods in Engineering 24(2):359–373, DOI 10.1002/nme.1620240207

Svanberg K (1995) A globally convergent version of MMA without linesearch. In: First World Congress of Structural and Multidisciplinary Optimization, Goslar, Germany

Vanderplaats GN (1973) CONMIN — A Fortran program for constrained function minimization. Technical Memorandum TM X-62282, NASA Ames Research Center, Moffett Field, California

Venter G, Sobieszczanski-Sobieski J (2004) Multidisciplinary optimization of a transport aircraft wing using particle swarm optimization. Structural and Multidisciplinary Optimization 26:121–131, URL http://dx.doi.org/10.1007/s00158-003-0318-3, 10.1007/s00158-003-0318-3

Wrenn G (1989) An indirect method for numerical optimization using the Kreisselmeier–Steinhauser function. Contractor Report NASA CR-4220, NASA Langley Research Center, Hampton, VA

Xu E (2009) pyIPOpt: An IPOPT connector to Python. URL http://code.google.com/p/pyipopt/

Zhou JL, Tits AL (1996) An SQP algorithm for finely discretized continuous minimax problems and other minimax problems with many objective functions. SIAM Journal on Optimization 6(2):461–487