Python Optimization Modeling Objects (Pyomo)

William E. Hart

Abstract We describe Pyomo, an open-source tool for modeling optimization applications in Python. Pyomo can be used to define abstract problems, create concrete problem instances, and solve these instances with standard solvers. Pyomo provides a capability that is commonly associated with algebraic modeling languages like AMPL and GAMS. Pyomo leverages the capabilities of the Coopr software, which integrates Python packages for defining optimizers, modeling optimization applications, and managing computational experiments.

Key words: Python, Modeling language, Optimization, Open Source Software

1 Introduction

Although high quality optimization solvers are commonly available, the effective integration of these tools with an application model is often a challenge for many users. Optimization solvers are typically written in low-level languages like Fortran or C/C++ because these languages offer the performance needed to solve large numerical problems. However, direct development of applications in these languages is quite challenging. Low-level languages like these can be difficult to program; they have complex syntax, enforce static typing, and require a compiler for development.

There are several ways that optimization technologies can be more effectively integrated with application models. For restricted problem domains, optimizers can be directly interfaced with application modeling tools. For example, modern spreadsheets like Excel integrate optimizers that can be applied to linear programming and simple nonlinear programming problems in a natural way. Similarly, engineering design frameworks like the Dakota toolkit (Eldred et al, 2006) can apply optimizers

William E. Hart
Sandia National Laboratories, Discrete Math and Complex Systems Department, PO Box 5800, Albuquerque, NM 87185 e-mail: `wehart@sandia.gov`

to nonlinear programming problems by executing separate application codes via a system call interface that use standardized file I/O.

Algebraic Modeling Languages (AMLs) are alternative approach that allows applications to be interfaced with optimizers that can exploit problem structure. AMLs are high-level programming languages for describing and solving mathematical problems, particularly optimization-related problems (Kallrath, 2004). AMLs like AIMMS (AIMMS, 2008), AMPL (AMPL, 2008; Fourer et al, 2003) and GAMS (GAMS, 2008) have programming languages with an intuitive mathematical syntax that supports concepts like sparse sets, indices, and algebraic expressions. AMLs provide a mechanism for defining variables and generating constraints with a concise mathematical representation, which is essential for large-scale, real-world problems that involve thousands of constraints and variables.

A related strategy is to use a standard programming language in conjunction with a software library that uses object-oriented design to support similar mathematical concepts. Although these modeling libraries sacrifice some of the intuitive mathematical syntax of an AML, they allow the user to leverage the greater flexibility of standard programming languages. For example, modeling tools like FlopC++ (FLOPC++, 2008), OPL (OPL, 2008) and OptimJ (OptimJ, 2008) enable the solution of large, complex problems with application models defined within a standard programming language.

The Python Optimization Modeling Objects (Pyomo) package described in this paper represents a fourth strategy, where a high level programming language is used to formulate a problem that can be solved by optimizers written in low-level languages. This two-language approach leverages the flexibility of the high-level language for formulating optimization problems and the efficiency of the low-level language for numerical computations. This approach is increasingly common in scientific computing tools, and the Matlab TOMLAB Optimization Environment (TOMLAB, 2008) is probably the most mature optimization software using this approach.

Pyomo supports the definition and solution of optimization applications using the Python scripting language. Python is a powerful dynamic programming language that has a very clear, readable syntax and intuitive object orientation. Pyomo was strongly influenced by the design of AMPL. It includes Python classes that can concisely represent mixed-integer linear programming (MILP) models. Pyomo is interated into Coopr, a COmmon Optimization Python Repository. The Coopr Opt package supports the execution of models developed with Pyomo using standard MILP solvers.

Section 2 describes the motivation and design philosophy behind Pyomo, including why Python was chosen for the design of Pyomo. Section 3 describes Pyomo and contrasts Pyomo with AMPL. Section 4 reviews other Python optimization packages that have been developed, and discusses the high-level design decisions that distinguish Coopr. Section 5 describes the Coopr Opt package and contrasts its capabilities with other Python optimization tools. Finally, Section 6 describes future Coopr developments that are planned.

2 Pyomo Motivation and Design Philosophy

The design of Pyomo is motivated by a variety of factors that have impacted applications at Sandia National Laboratories. Sandia's discrete mathematics group has successfully used AMPL to model and solve large-scale integer programs for many years. This application experience has highlighted the value of AMLs for real-world applications, which are now an integral part of operations research solutions at Sandia.

Pyomo was developed to provide an alternative platform for developing math programming models that facilitates the application and deployment of optimization capabilities. Consequently, Pyomo is not intended to perform modeling *better* than existing tools. Instead, it supports a different modeling approach for which the software is designed for flexibility, extensibility, portability, and maintainability.

2.1 Design Goals and Requirements

2.1.1 Open Source

A key goal of Pyomo is to provide an open-source math programming modeling capability. Although open-source optimization solvers are widely available in packages like COIN-OR, surprisingly few open-source tools have been developed to model optimization applications. An open-source capability for Pyomo is motivated by several factors:

- **Transparency and Reliability**: When managed well, open-source projects facilitate transparency in the software design and implementation. Since any developer can study and modify the software, bugs and performance limitations can be identified and resolved by a wide range of developers with diverse software experience. Consequently, there is growing evidence that managing software as open-source can improve its reliability.
- **Customizable Capability**: A key limitation of commercial modeling tools is the ability to customize the modeling or optimization process. An open-source project allows a diverse range of developers to prototype new capabilities. These extensions can customize the software for specific applications, and they can motivate capabilites that are integrated into future software releases.
- **Flexible Licensing**: A variety of significant operations research applications at Sandia National Laboratories have required the use of a modeling tool with a non-commercial license. Open-source license facilitate the free distribution of Pyomo within other open-source projects.

Of course, the use of an open-source model is not a panacea. Ensuring high reliability of the software requires careful software management and a commited developer community. However, flexible licensing appears to be a distinct feature of open-source software. The Coopr software, which contains Pyomo, is licensed under the BSD.

2.1.2 Flexible Modeling Language

Another goal of Pyomo is to directly use a modern programming language to support the definition of math programming models. In this manner, Pyomo is similar to tools like FlopC++ and OptimJ, which support modeling in C++ and Java respectively. The use of an existing programming language has several advantages:

- **Extensibility and Robustness**: A well-used modern programming language provides a robust foundation for developing and applying models, because the language has been well-tested in a wide variety of contexts. Further, extensions typically do not require changes to the language but instead involve additional classes and modeling routines that can be used in the modeling process. Thus, support of the modeling language is not a long-term factor when managing the software.
- **Documentation**: Modern programming languages are typically well-documented, and there is often a large on-line community to provide feedback to new users.
- **Standard Libraries**: Languages like Java and Python have a rich set of libraries for tackling just about every programming task. For example, standard libraries can support capabilities like data integration (e.g. working with spreadsheets), thereby avoiding the need to directly support this in a modeling tool.

An additional aspect of general-purpose programming languages is that they can support modern language features, like classes and first-class functions, that can be critical when defining complex models.

Pyomo is implemented in Python, a powerful dynamic programming language that has a very clear, readable syntax and intuitive object orientation. When compared with AMLs like AMPL, Pyomo has a more verbose and complex syntax. Thus, a key issue with this approach concerns the target user community and their level of comfort with standard programming concepts. Our examples in this paper compare and contrast AMPL and Pyomo models, which illustrate this trade-off.

2.1.3 Portability

A requirement of Pyomo's design is that it work on a diverse range of compute platforms. In particular, working well on both MS Windows and Linux platforms is a key requirement for many Sandia applications. The main impact of this requirement has been to limit the choice of programming languages. For example, the .Net languages were not considered for the design of Pyomo due to portability considerations.

2.1.4 Solver Integration

Modeling tools can be roughly categorized into two classes based on how they integrate with optimization solvers: *tightly coupled* modeling tools directly link in optimization solver libraries (including dynamic linking), and *loosely coupled* modeling tools apply external optimization executables (e.g. through system calls). Of course, these options are not exclusive, and a goal of Pyomo is to support both types of solver interfaces.

This design goal has led to a distinction in Pyomo between model formulation and optimization execution. Pyomo uses a high level programming language to formulate a problem that can be solved by optimizers written in low-level languages. This two-language approach leverages the flexibility of the high-level language for formulating optimization problems and the efficiency of the low-level language for numerical computations.

2.1.5 Abstract Models

A requirement of Pyomo's design is that it support the definition of abstract models in a manner similar to the AMPL. AMPL separates the declaration of a model from the data that generates a model instance. This is supports an extremely flexible modeling capability, which has been leveraged extensively in applications at Sandia.

To mimic this capability, Pyomo uses a symbolic representation of data, variables, constraints, etc. Model instances are then generated from external data sets using construction routines that are provided by the user when defining sets, parameters, etc. Further, Pyomo is designed to use data sets in the AMPL format to facilitate translation of models between AMPL and Pyomo.

2.2 Why Python?

Pyomo has been developed in Python for a variety of reasons. First, Python meets the criteria outlined in the previous section:

- **Open Source License:** Python is freely available, and its liberal open source license lets you modify and distribute a Python-based application with few restrictions.
- **Features:** Python has a rich set of datatypes, support for object oriented programming, namespaces, exceptions, and dynamic loading.
- **Support and Stability:** Python is highly stable, and it is well supported through newsgroups and special interest groups.
- **Documentation:** Users can learn about Python from extensive online documentation, and a number of excellent books that are commonly available.
- **Standard Library:** Python includes a large number of useful modules.

- **Extendability and Customization:** Python has a simple model for loading Python code developed by a user. Additionally, compiled code packages that optimize computational kernels can be easily used. Python includes support for shared libraries and dynamic loading, so new capabilities can be dynamically integrated into Python applications.
- **Portability:** Python is available on a wide range of compute platforms, so portability is typically not a limitation for Python-based applications.

Another factor, not to be overlooked, is the increasing acceptance of Python in the scientific community (Oliphant, 2007). Large Python projects like SciPy (Jones et al, 2001–) and SAGE (Stein, 2008) strongly leverage a diverse set of Python packages.

Finally, we note that several other popular programming languages were also considered for Pyomo. However, in most cases Python appears to have distinct advantages:

- **.Net:** As mentioned earlier, the .Net languages are not portable to Linux platforms, and thus they were not suitable for Pyomo.
- **Ruby:** At the moment, Python and Ruby appear to be the two most widely recommended scripting languages that are portable to Linux platforms, and comparisons suggest that their core functionality is similar. Our preference for Python is largely based on the fact that it has a nice syntax that does not require users to type weird symbols (e.g. \$, %, @). Thus, we expect this will be a more natural language for expressing math programming models.
- **Java:** Java has a lot of the same strengths as Python, and it is arguably as good a choice for Pyomo. However, two aspects of Python recommended it for Pyomo instead of Java. First, Python has a powerful interactive interpreter that allows realtime code development and encourages experimentation with Python software. Thus, users can work interactively with Pyomo models to become familiar with these objects and to diagnose bugs. Second, it is widely acknowledged that Python's dynamic typing and compact, concise syntax makes software development quick and easy. Although some very interesting optimization modeling tools have been developed in languages like C++ and Java, there is anecdotal evidence that users will not be as productive in these languages as they will when using tools developed in languages like Python (PythonVSJava, 2008).
- **C++:** Models formulated with the FlopC++ package are similar to models developed with Pyomo. They are be specified in a declarative style using classes to represent model components (e.g. sets, variables and constraints). However, C++ requires explicit compilation to execute code, and it does not support an interactive interpreter. Thus, we believe that Python will provide a more flexible language for users.

3 Pyomo Overview

Pyomo can be used to define abstract problems, create concrete problem instances, and solve these instances with standard solvers. Pyomo can generate problem instances and apply optimization solvers with a fully expressive programming language. Python's clean syntax allows Pyomo to express mathematical concepts with a reasonably intuitive syntax. Further, Pyomo can be used within an interactive Python shell, thereby allowing a user to interactively interrogate Pyomo-based models. Thus, Pyomo has many of the advantages of both AML interfaces and modeling libraries.

3.1 A Simple Example

In this section we illustrate Pyomo's syntax and capabilities by demonstrating how a simple AMPL example can be replicated with Pyomo Python code. Consider the AMPL model, `prod.mod`:

```
set P;

param a {j in P};
param b;
param c {j in P};
param u {j in P};

var X {j in P};

maximize Total_Profit: sum {j in P} c[j] * X[j];

subject to Time: sum {j in P} (1/a[j]) * X[j] <= b;

subject to Limit {j in P}: 0 <= X[j] <= u[j];
```

To translate this into Pyomo, the user must first import the Pyomo module and create a Pyomo **Model** object:

```
#
# Import Pyomo
#
from coopr.pyomo import *

#
# Create model
#
```

```
model = Model()
```

This import assumes that Pyomo is available on the users's Python path (see Python documentation for PYTHONPATH for further details). Next, we create the sets and parameters that correspond to the data used in the AMPL model. This can be done very intuitively using the **Set** and **Param** classes.

```
model.P = Set()

model.a = Param(model.P)
model.b = Param()
model.c = Param(model.P)
model.u = Param(model.P)
```

Note that parameter *b* is a scalar, while parameters *a*, *c* and *u* are arrays indexed by the set *P*.

Next, we define the decision variables in this model.

```
model.X = Var(model.P)
```

Decision variables and model parameters are used to define the objectives and constraints in the model. Parameters define constants and the variables are the values that are optimized. Parameter values are typically defined by a data file that is processed by Pyomo.

Objectives and constraints are explicitly defined expressions in Pyomo. The **Objective** and **Constraint** classes require a **rule** option that specifies how these expressions are constructed. This is a function that takes one or more arguments: the first arguments are indices into a set that defines the set of objectives or constraints that are being defined, and the last argument is the model that is used to define the expression.

```
def Objective_rule(model):
    ans = 0
    for j in model.P:
      ans = ans + model.c[j] * model.X[j]
    return ans
model.Total_Profit = Objective(rule=Objective_rule,
                               sense=maximize)

def Time_rule(model):
    ans = 0
    for j in model.P:
      ans = ans + (1.0/model.a[j]) * model.X[j]
    return ans < model.b
model.Time = Constraint(rule=Time_rule)
```

```
def Limit_rule(j, model):
    return (0, model.X[j], model.u[j])
model.Limit = Constraint(model.P, rule=Limit_rule)
```

The rules used to construct these objects use standard Python functions. The **Time_rule** function includes the use of < and > operators on the expression, which define upper and lower bounds on the constraints. The **Limit_rule** function illustrates another convention that is supported by Pyomo; a rule can return a tuple that defines the lower bound, body and upper bound for a constraint. The value 'None' can be returned for one of the limit values if a bound is not enforced.

Once an abstract model has been created, it can be printed as follows:

```
model.pprint()
```

This summarize the information in the Pyomo model, but it does not print out explicit expressions. This is due to the fact that an abstract model needs to be instanted with data to generate the model objectives and constraints:

```
instance = model.create("prod.dat")
instance.pprint()
```

Once a model instance has been constructed, an optimizer can be applied to it to find an optimal solution. For example, the PICO integer programming solver can be used within Pyomo as follows:

```
opt = solvers.SolverFactory("pico")
opt.keepFiles=True
results = opt.solve(instance)
```

This creates an optimizer object for the PICO executable, and it indicates that temporary files should be kept. The Pyomo model instance is optimized, and the optimizer returns an object that contains the solutions generated during optimization.

3.2 *Pyomo Commandline Script*

Appendix 7 provides a complete Python script for the model described in the previous section. Although this Python script can be executed directly, Coopr includes a `pyomo` script that can construct this model, apply an optimizer and summarize the results. For example, the following command line executes Pyomo using a data file in a format consistent with AMPL:

```
  pyomo prod.py prod.dat
```

The `pyomo` script has a variety of command line options to provide information about the optimization process. Options can control how debugging information is printed, including logging information generated by the optimizer and a summary of the model generated by Pyomo. Further, Pyomo can be configured to keep all intermediate files used during optimization, which can support debugging of the model construction process.

4 Related Python Optimization Tools

A variety of related optimization packages have been developed in Python that are designed to support the formulation and solution of specific classes of structure optimization applications:

- **CVXOPT:** A Python package for convex optimization (CVXOPT, 2008).
- **PuLP:** A Python package that can be used to describe linear programming and mixed-integer linear programming optimization problems (PuLP, 2008).
- **POAMS:** A Python modeling tool for linear and mixed-integer linear programs that defines Python objects for abstract sets, constraints, objectives, decision variables, and solver interfaces.
- **OpenOpt:** A relatively new numerical optimization framework that is closely coupled with the SciPy scientific Python package (OpenOpt, 2008).
- **NLPy:** A Python optimization framework that leverages AMPL to create problem instances, which can then be processed in Python (NLPy, 2008).
- **Pyiopt:** A Python interface to the COIN-OR Ipopt solver (Pyipopt, 2008).

Pyomo is closely related to the modeling capabilities of PuLP and POAMS. Pyomo defines Python objects that can be used to express models, and like POAMS, Pyomo supports a clear distinction between abstract models and problem instances. The main distinguishing feature of Pyomo is support for an instance construction process that is automated by object properties. This is akin to the capabilities of AML's like AMPL and GAMS, and it provides a standardized technique for constructing model instances. Pyomo models can be initialized with a generic data object, which can be initialized with a variety of data sources (including AMPL *.dat files).

Like NLPy and OpenOpt, the goal of Coopr Opt is to support a diverse set of optimization methods and applications. Coopr Opt includes a facility for transforming problem formats, which allows optimizers to solve problems without the user worrying about solver-specific implementation details. Further, Coopr Opt supports mechanisms for reporting detailed information about optimization solutions, in a manner akin to the OSrL data format supported by the COIN-OR OS project (Fourer et al, 2008).

In the remainder of this section we use the following example to illustrate the differences between PuLP, POAMS and Pyomo:

$$\begin{aligned} \text{minimize } & -4x_1 - 5x_2 \\ \text{subject to } & 2x_1 + x_2 \leq 3 \\ & x_1 + 2x_2 \leq 3 \\ & x_1, x_2 \geq 0 \end{aligned} \tag{1}$$

4.1 PuLP

PuLP relies on overloading operators and commonly used mathematical functions to define expression objects that define objectives and constraints. A problem object is defined, and the objective and constraints are added using the += operator. Further, problem variables can be defined over index sets to enable compact specification of constraints and objectives.

The following PuLP example minimizes the LP (1):

```
from pulp import *
x1 = LpVariable("x1",0)
x2 = LpVariable("x2",0)
prob = LpProblem("Example", LpMinimize)
prob += -4*x1 - 5*x2
prob += 2*x1 + x2 <= 3
prob += x1 + 2*x2 <= 3
prob.solve()
```

4.2 POAMS

POAMS is a Python modeling tool for linear and mixed-integer linear programs that defines Python objects for abstract sets, constraints, objectives, decision variables, and solver interfaces. These objects can be used to compose an abstract model definition, which is then used to construct a concrete problem instance from a given data set. This separation of the problem instance from the data facilitates the definition of abstract models that can be populated from a diverse range of data sources.

POAMS models are managed by classes derived from the POAMS LP object. The following POAMS example minimizes the LP (1) by deriving a class, instantiating it, and then running the model:

```
from poams import *

class Example(LP):

    index = Set(1,2)
    x = Var(index)
    obj = Objective()
    c1 = Constraint()
    c2 = Constraint()

    def model(self):
        self.obj.min(-4*self.x[1] - 5*self.x[2])
        self.c1.load( 2*self.x[1] +   self.x[2] <= 3.0)
        self.c2.load(   self.x[1] + 2*self.x[2] <= 3.0)

prob = Example().model()
prob.solve()
```

4.3 Pyomo

The following Pyomo example minimizes LP (1) by instantiating an abstract model, populating the model with symbols, generating an instance, and then applying the PICO MIP optimizer:

```
from coopr.pyomo import *

model = Model()

model.index = Set(initialize=[1,2])
model.x = Var(model.index)

def obj_rule(model):
    return -4*model.x[1]-5*model.x[2]
model.obj = Objective(rule=obj_rule)

def c1_rule(model):
    ans = 2*model.x[1] + model.x[2]
    return ans < 3.0
model.c1 = Constraint(rule=c1_rule)

def c2_rule(model):
    ans = model.x[1] + 2*model.x[2]
    return ans < 3.0
model.c2 = Constraint(rule=c2_rule)


instance = model.create()
opt = solvers.SolverFactory("pico")
```

```
results = opt.solve(instance)
```

5 The Coopr Opt Package

The goal of the Coopr Opt package is to support the execution of optimizers in a generic manner. Although Pyomo uses this package, Coopr Opt is designed to support a wide range of optimizers. However, Coopr Opt is not as mature as the OpenOpt package; it currently only supports interfaces to a limited number of optimizers aside from the LP and MILP solvers used by Pyomo.

Coopr Opt is supports a simple strategy for setting up and executing an optimizer, which is illustrated by the following script:

```
opt = SolverFactory( name )
opt.reset()
results = opt.solve( problem )
results.write()
```

This script illustrates several design principles that Coopr follows:

- **Dynamic Registration of Optimizers**: Optimizers are registered via a plugin mechanism that provides an extensible architecture for developers of third-party optimizers. This plugin mechanism includes the specification of parameters that can be initialized from a configuration file.
- **Separation of Problems and Solvers**: Coopr Opt treats problems and solvers as separate entities. This promotes the development of tools like Pyomo that support flexible definition of optimization applications, and it enables automatic transformation of problem instances.
- **Problem Transformation**: A key challenge for optimization packages is the need to support a diverse set of problem formats. This is an issue even for LP and MILP solver packages, where MPS is the least common denominator for users. Coopr Opt supports an automatic problem transformation mechanism that enables the application of optimizers to problems with a wide range of formats.
- **Generic Representation of Optimizer Results**: Coopr Opt borrows and extends the representation used by the COIN-OR OS project to support a general representation of optimizer results. The *results* object returned by a Coopr optimizer includes information about the problem, the solver execution, and one or more solutions generated during optimization.

If the problem in Appendix 7 is being solved, this script would print the following information that is contained in the `results` object:

```
==========================================================
---   Solver Results                                   ---
==========================================================
----------------------------------------------------------
----   Problem Information                          ----
----------------------------------------------------------
  name: None
  num_constraints: 5
  num_nonzeros: 6
  num_objectives: 1
  num_variables: 2
  sense: maximize
  upper_bound: 192000
----------------------------------------------------------
----   Solver Information                           ----
----------------------------------------------------------
  error_rc: 0
  nbounded: None
  ncreated: None
  status: ok
  systime: None
  usrtime: None
----------------------------------------------------------
----   Solution 0
----------------------------------------------------------
  gap: 0.0
  status: optimal
  value: 192000
  Primal Variables
        X_bands_        6000
        X_coils_        1400
  Dual Variables
        c_u_Limit_1     4
        c_u_Time_0      4200
----------------------------------------------------------
```

It is worth noting that Coopr Opt currently does not support direct library interfaces to optimizers, which is a feature that is strongly supported by Python. However, this is not a design limitation, but instead has been a matter of development priorities. Efforts are planned with the POAMS and PuLP developers to adapt the direct solver interfaces used in these packages for use within Coopr.

Although Coopr Opt development has focused on developing interfaces to LP and MILP solvers, we have recently begun developing interfaces to general-purpose nonlinear programming methods. One of the goals of this effort is to develop application interfaces that are consistent with the interfaces supported by Acro's COLIN optimization library (ACRO, 2008). COLIN has recently been extended to support a system call interface that uses standardized file I/O. An XML format has been developed that can be more rigorously checked than the file format used by the

Dakota toolkit (Eldred et al, 2006), and this format can be readily extended to new application results. Coopr Opt supports applications defined using this system call interface, which will simplify the integration of COLIN optimizers into Coopr Opt.

6 Discussion

Coopr is being actively developed to support real-world applications at Sandia National Laboratories. This experience has validated our assessment that Python is an effective language for supporting the solution of optimization applications. Although it is clear that custom languages can support a much more mathematically intuitive syntax, Python's clean syntax and programming model make it a natural choice for optimization tools like Coopr.

Coopr will be publicly released as an open source project in 2008. Future development will focus on several key design issues:

- Interoperable with commonly available optimization solvers, and the relationship of Coopr and OpenOpt.
- Exploiting synergy with POAMS and PuLP. Developers of Coopr, POAMS and PuLP are assessing this intersection to identify where synergistic efforts can be leveraged. For example, the direct solver interface used by POAMS and PuLP can be adapted for use in Pyomo.
- Extending Pyomo to support the definition of general nonlinear models. Conceptually, this is straightforward, but the model generation and expression mechanisms need to be re-designed to support capabilities like automatic differentiation.

Acknowledgements We thank the ICS reviewers for their critical feedback. We also thank Jon Berry, Robert Carr and Cindy Phillips for their critical feedback on the design of Pyomo, and David Gay for developing the Coopr interface to AMPL NL and SOL files. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under Contract DE-AC04-94-AL85000.

References

ACRO (2008) ACRO optimization framework. `http://software.sandia.gov/acro`
AIMMS (2008) AIMMS home page. `http://www.aimms.com`
AMPL (2008) AMPL home page. `http://www.ampl.com/`
CVXOPT (2008) CVXOPT home page. `http://abel.ee.ucla.edu/cvxopt`

Eldred MS, Brown SL, 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 (2006) DAKOTA, a multilevel parallel object-oriented framework for design optimization, parameter estimation, uncertainty quantification, and sensitivity analysis: Version 4.0 users manual. Tech. Rep. SAND2006-6337, Sandia National Laboratories

FLOPC++ (2008) FLOPC++ home page. `https://projects.coin-or.org/FlopC++`

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

Fourer R, Ma J, Martin K (2008) Optimization services: A framework for distributed optimization. Mathematical Programming (submitted)

GAMS (2008) GAMS home page. `http://www.gams.com`

Jones E, Oliphant T, Peterson P, et al (2001–) SciPy: Open source scientific tools for Python. URL `http://www.scipy.org/`

Kallrath J (2004) Modeling Languages in Mathematical Optimization. Kluwer Academic Publishers

NLPy (2008) NLPy home page. `http://nlpy.sourceforge.net/`

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

OpenOpt (2008) OpenOpt home page. `http://scipy.org/scipy/scikits/wiki/OpenOpt`

OPL (2008) OPL home page. `http://www.ilog.com/products/oplstudio`

OptimJ (2008) Ateji home page. `http://www.ateji.com`

PuLP (2008) PuLP: A python linear programming modeler. `http://130.216.209.237/engsci392/pulp/FrontPage`

Pyipopt (2008) Pyipopt home page. `http://code.google.com/p/pyipopt/`

PythonVSJava (2008) Python & java: A side-by-side comparison. `http://www.ferg.org/projects/python_java_side-by-side.html`

Stein W (2008) Sage: Open Source Mathematical Software (Version 2.10.2). The Sage Group, `http://www.sagemath.org`

TOMLAB (2008) TOMLAB optimization environment. `http://www.tomopt.com/tomlab`

7 A Complete Pyomo Example

```
#
# Imports
#
from coopr.pyomo import *

#
# Setup the model
#
model = Model()

model.P = Set()

model.a = Param(model.P)
model.b = Param()
model.c = Param(model.P)
model.u = Param(model.P)

model.X = Var(model.P)

def Objective_rule(model):
    ans = 0
    for j in model.P:
      ans = ans + model.c[j] * model.X[j]
    return ans
model.Total_Profit = Objective(rule=Objective_rule,
                               sense=maximize)

def Time_rule(model):
    ans = 0
    for j in model.P:
      ans = ans + (1.0/model.a[j]) * model.X[j]
    return ans < model.b
model.Time = Constraint(rule=Time_rule)

def Limit_rule(j, model):
    return (0, model.X[j], model.u[j])
model.Limit = Constraint(model.P, rule=Limit_rule)
```