

Coopr: A COmmon Optimization Python Repository

William E. Hart*

Abstract

We describe Coopr, a COmmon Optimization Python Repository. Coopr integrates Python packages for defining optimizers, modeling optimization applications, and managing computational experiments. A major driver for Coopr development is the Pyomo package, that 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.

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 [6] can apply optimizers 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 [12]. AMLs like AIMMS [2], AMPL [3, 8] and GAMS [10] have programming languages with an intuitive mathematical syntax that supports concepts

*Sandia National Laboratories, Discrete Math and Complex Systems Department, PO Box 5800, Albuquerque, NM 87185; 505-844-2217; wehart@sandia.gov

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++ [7], OPL [15] and OptimJ [16] enable the solution of large, complex problems with application models defined within a standard programming language.

The Coopr Python 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 [20] is probably the most mature optimization software using this approach.

Coopr is a COmmon Optimization Python Repository that 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. Coopr was strongly influenced by the design of AMPL. Coopr's Pyomo package supports Python classes that can concisely represent mixed-integer linear programming (MILP) models, and these models can be exported and solved by standard MILP solvers. Coopr also provides a generic system call interface that is similar to the interface used by Dakota.

Section 2 describes why Python was chosen for the design of Coopr and the impact that this has had on the use of Coopr. Section 3 reviews other Python optimization packages that have been developed, and discusses the high-level design decisions that distinguish Coopr. The next sections describe two Coopr packages for optimization: Pyomo and Opt. Section 4 describes Pyomo and contrasts Pyomo with AMPL. 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 Why Python?

Python has been gaining significant acceptance in the scientific community. Python is widely used in bioinformatics [17], and it has proven a flexible language for scripting high-performance scientific software [4]. The SciPy [11] library includes a wide variety of mathematical tools used for scientific computing, and the SAGE program [19] integrates these and other math analysis tools into an application that is akin to Matlab and Mathematica.

Oliphant [13] provides a concise summary of the reasons why Python is an effective

language for scientific computing:

- **Features:** Python has a rich set of datatypes, support for object oriented programming, namespaces, exceptions, dynamic loading, and a large number of useful modules.
- **Syntax:** Python has a nice syntax that does not require users to type weird symbols (e.g. \$, %, @). Further, Python's language naturally supports both procedural and object-oriented software design.
- **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.
- **Interpreter:** Python has a powerful interactive interpreter that allows realtime code development and encourages experimentation with Python software.
- **Documentation:** Users can learn about Python from extensive online documentation, and a number of excellent books that are commonly available.
- **Support and Stability:** Python is highly stable, and it is well supported through newsgroups and special interest groups.
- **Portability:** Python is available on a wide range of compute platforms, so portability is typically not a limitation for Python-based applications.
- **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.

Another factor, not to be overlooked, is the increasing acceptance of Python in the scientific community. Large Python projects like SciPy and SAGE strongly leverage a diverse set of Python packages.

When considering languages for Coopr, the stability and open source license of Python were critical features. Key motivating applications at Sandia National Laboratories required an open source solution, which precludes the use of proprietary languages and tools like AMPL and Matlab. The simplicity of Python's language was also critical, since we needed to have an intuitive syntax for Coopr's Pyomo modeling package. Finally, the features of Python's standard libraries has been critical to the effective deployment of Coopr applications. For example, Python's `win32com` module made it easy to read and write data in Excel spreadsheets.

It is widely acknowledged that Python's dynamic typing makes software development quick and easy. However, this shifts the burden to developers to implement software tests that validate the correctness of a Python package. Fortunately, Python packages can be easily installed to support unit tests and record code coverage when they are run (see the `nose` and `coverage` packages). Unlike similar tools used for low-level languages, the execution of these testing tools does not require the recompilation or configuration of the Python software.

3 Python Optimization Tools

A variety of optimization packages have been developed in Python. We briefly describe these packages here and illustrate their syntax for the following simple linear program:

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

Note that the following packages are designed to support the formulation and solution of specific classes of structure optimization applications. Several authors have also developed general-purpose optimizers in Python, such as genetic algorithms and swarm optimization.

3.1 CVXOPT

CVXOPT is a software package for convex optimization [5]. Its main purpose is to make the development of software for convex optimization applications straightforward by building on Python’s extensive standard library and on the strengths of Python as a high-level programming language. CVXOPT includes interfaces to BLAS and LAPACK, and it leverages the Python Numpy package to efficiently manipulate arrays and matrices in Python. CVXOPT interfaces with a variety of packages (GLPK, MOSEK and DSDP) to solve a variety of convex problems, such as linear programs, second-order cone programs and semi-definite programs.

CVXOPT overloads a variety of arithmetic operators and mathematical functions. These can be used to define expressions for constraints and objectives of a problem. CVXOPT leverages the fact that matrices and arrays are used in the specification of these expressions to support a very concise syntax.

The following CVXOPT example minimizes the LP (1):

```
>>> from cvxopt.base import matrix
>>> from cvxopt import solvers
>>> c = matrix([-4., -5.])
>>> G = matrix([[2., 1., -1., 0.], [1., 2., 0., -1.]])
>>> h = matrix([3., 3., 0., 0.])
>>> sol = solvers.lp(c, G, h)
```

3.2 PuLP

PuLP is an open source software written in Python that can be used to describe linear programming and mixed-integer linear programming optimization problems [18]. PuLP can call a variety of external LP solvers (GLPK, CPLEX, XPRESS etc) to solve a model. Like CVXOPT, 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()
```

3.3 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()
```

3.4 OpenOpt and SciPy

OpenOpt is a relatively new numerical optimization framework that is closely coupled with the SciPy scientific Python package [14]. The primary goal of OpenOpt is to provide a common syntax for many different optimization solvers. OpenOpt includes interfaces to a diverse set of optimizers for problems like linear programs, nonlinear least squares and global optimization problems. OpenOpt includes interfaces to external optimizers from the SciPy package, CVXOPT, native Python optimizers, and interfaces to standalone packages.

The following OpenOpt example minimizes the LP (1):

```
from numpy import *
from scikits.openopt import LP
f = array([-4.0, -5.0 ])
A = mat('2 1; 1 2')
b = [3, 3]
lb = [0, 0]
ub = [inf, inf]
p = LP(f, A=A, b=b, lb=lb, ub=ub)
r = p.solve('cvxopt_lp')
```

3.5 Coopr

We conclude this section by comparing and contrasting Coopr with these packages. The Coopr Pyomo package 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).

The Coopr Opt package is closely related to the optimization objects defined by PuLP. Like OpenOpt, the goal of this package 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 [9].

4 The Coopr Pyomo Package

The Python Optimization Modeling Objects (Pyomo) package is a Coopr Python package that 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.

4.1 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 summarizes 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 instantiated 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.

4.2 Pyomo Commandline Script

Appendix A 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.

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 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 A 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
systemtime:	None
usertime:	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 program-

ming 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 [1]. 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 [6], and this format can be readily extended to new application results. Coop Opt supports applications defined using this system call interface, which will simplify the integration of COLIN optimizers into Coop 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 Coop Opt.

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 Coop Opt and OpenOpt.
- Exploiting synergy with POAMS and PuLP. Developers of Coop Opt, 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 Jon Berry, Robert Carr and Cindy Phillips for their critical feedback on the design of Pyomo. We also thank David Gay for developing the Coop Opt 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

- [1] *ACRO optimization framework*. <http://software.sandia.gov/acro>.

- [2] *AIMMS home page*. <http://www.aimms.com>.
- [3] *AMPL home page*. <http://www.ampl.com/>.
- [4] D. BEAZLEY AND P. LOMDAHL, *Building flexible large-scale scientific computing applications with scripting languages*, in Proc. 8th SIAM Conference on Parallel Processing for Scientific Computing, 1997.
- [5] *CVXOPT home page*. <http://abel.ee.ucla.edu/cvxopt>.
- [6] M. S. ELDRED, S. L. BROWN, D. M. DUNLAVY, D. M. GAY, L. P. SWILER, A. A. GIUNTA, W. E. HART, J.-P. WATSON, J. P. EDDY, J. D. GRIFFIN, P. D. HOUGH, T. G. KOLDA, M. L. MARTINEZ-CANALES, AND P. J. WILLIAMS, *DAKOTA, a multilevel parallel object-oriented framework for design optimization, parameter estimation, uncertainty quantification, and sensitivity analysis: Version 4.0 users manual*, Tech. Report SAND2006-6337, Sandia National Laboratories, 2006.
- [7] *FLOPC++ home page*. <https://projects.coin-or.org/FlopC++>.
- [8] R. FOURER, D. M. GAY, AND B. W. KERNIGHAN, *AMPL: A Modeling Language for Mathematical Programming, 2nd Ed.*, Brooks/Cole–Thomson Learning, Pacific Grove, CA, 2003.
- [9] R. FOURER, J. MA, AND K. MARTIN, *Optimization services: A framework for distributed optimization*, Mathematical Programming. (submitted).
- [10] *GAMS home page*. <http://www.gams.com>.
- [11] E. JONES, T. OLIPHANT, P. PETERSON, ET AL., *SciPy: Open source scientific tools for Python*, 2001–.
- [12] J. KALLRATH, *Modeling Languages in Mathematical Optimization*, Kluwer Academic Publishers, 2004.
- [13] T. E. OLIPHANT, *Python for scientific computing*, Computing in Science and Engineering, (2007), pp. 10–20.
- [14] *OpenOpt home page*. <http://scipy.org/scipy/scikits/wiki/OpenOpt>.
- [15] *OPL home page*. <http://www.ilog.com/products/oplstudio>.
- [16] *Ateji home page*. <http://www.ateji.com>.
- [17] J. PAINTER AND E. A. MERRITT, *mmLib Python toolkit for manipulating annotated structural models of biological macromolecules*, J. Applied Crystallography, 37 (2004), pp. 174–178.

- [18] *PuLP: A python linear programming modeler*. <http://130.216.209.237/engsci392/pulp/FrontPage>.
- [19] W. STEIN, *Sage: Open Source Mathematical Software (Version 2.10.2)*, The Sage Group, 2008. <http://www.sagemath.org>.
- [20] *TOMLAB optimization environment*. <http://www.tomopt.com/tomlab>.

A A Complete Python 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)
```