Open Source Tools for Optimization in Python

Ted Ralphs

Sage Days Workshop
IMA, Minneapolis, MN, 21 August 2017
1. Introduction

2. COIN-OR

3. Modeling Software

4. Python-based Modeling Tools
   - PuLP/DipPy
   - CyLP
   - yaposib
   - Pyomo
Caveats

I have no idea about the background of the audience.

The talk may be either too basic or too advanced.

Why am I here?

I’m not a Sage developer or user (yet!).

I’m hoping this will be a chance to get more involved in Sage development.

Please ask lots of questions so as to guide me in what to dive into!
Mathematical optimization provides a formal language for describing and analyzing optimization problems. Elements of the model:

- Decision variables
- Constraints
- Objective Function
- Parameters and Data

The general form of a mathematical optimization problem is:

\[
\text{min or max } \quad f(x) \quad \text{(1)}
\]

\[
s.t. \quad g_i(x) \left\{ \begin{array}{l}
\leq \\
= \\
\geq \\
\end{array} \right\} b_i \quad \text{(2)}
\]

\[
x \in X \quad \text{(3)}
\]

where \(X \subseteq \mathbb{R}^n\) might be a discrete set.
The type of a mathematical optimization problem is determined primarily by:
- The form of the objective and the constraints.
- The form of the set $X$.

The most important factors in whether a mathematical optimization problem is “tractable” are the convexity of the objective function and the feasible region.

Mathematical optimization problems are generally classified according to the following dichotomies.
- Linear/nonlinear
- Convex/nonconvex
- Discrete/continuous
- Stochastic/deterministic

See the NEOS guide for a more detailed breakdown.
Quick Intro to Modeling and Analysis

Modeling and Analysis Process

- Develop a conceptual model of the system to be optimized.
- Formulate a corresponding mathematical optimization problem.
- Develop an “abstract” model using a modeling system.
- Populate the model with data to obtain an instance.
- “Solve” the resulting instance using appropriate software.
- Analyze the results.

These steps generally involve several different pieces of software working in concert.

- For optimization problems, the modeling is often done with an algebraic modeling system.
- Data can be obtained from a wide range of sources, including spreadsheets.
- Solution of the model is usually relegated to specialized software, depending on the type of model.
Example: Optimal Bond Portfolio

- A bond portfolio manager has $100K to allocate to two different bonds.

<table>
<thead>
<tr>
<th>Bond</th>
<th>Yield</th>
<th>Maturity</th>
<th>Rating</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>4</td>
<td>3</td>
<td>A (2)</td>
</tr>
<tr>
<td>B</td>
<td>3</td>
<td>4</td>
<td>Aaa (1)</td>
</tr>
</tbody>
</table>

- The goal is to maximize total return subject to the following limits.
  - The average rating must be at most 1.5 (lower is better).
  - The average maturity must be at most 3.6 years.

- Any cash not invested will be kept in a non-interest bearing account and is assumed to have an implicit rating of 0 (no risk).
Let $x_1$ be the amount of Bond A to be purchased and $x_2$ the amount of Bond B to be purchased.

Then the problem of determining the optimal portfolio can be formulated as follows.

\[
\begin{align*}
\text{max} & \quad 4x_1 + 3x_2 \\
\text{s.t.} & \quad x_1 + x_2 \leq 100 \\
& \quad 2x_1 + x_2 \leq 150 \\
& \quad 3x_1 + 4x_2 \leq 360 \\
& \quad x_1, x_2 \geq 0
\end{align*}
\]
Abstracting the Model

- The previous model is not very satisfactory from a practical perspective, since the basic parameters might change after the model is specified.
- An *abstract algebraic model* is a model that doesn’t have values for the input data.
- Components of an abstract algebraic model are

**Data**
- **Sets**: Sets that index variables and constraints.
- **Parameters**: Specific values of numerical inputs.

**Model**
- **Variables**: Values in the model that need to be decided upon.
- **Objective Function**: A function of the variable values to be maximized or minimized.
- **Constraints**: Functions of the variable values that must lie within given bounds.
Let $B$ be a set of bonds available for purchase and let $F$ be a set of salient features of these bonds to be limited and/or optimized.

Let $x_b$ be the amount of bond $b$ to be purchased.

Let $c_b$ be the value of the feature to be optimized for bond $b$ and $a_{fb}$ be the value of feature $f$ for bond $b$.

Finally, let $L_f$ be the limit on the average value of feature $f$ (assume are all are upper limits).

Abstract Bond Portfolio Model

\[
\begin{align*}
\max & \quad \sum_{b \in B} c_b x_b \\
\text{s.t.} & \quad \sum_{f \in F} \sum_{b \in B} a_{fb} x_b \leq L_b \\
& \quad x_b \geq 0 \quad \forall b \in B
\end{align*}
\]
Outline

1 Introduction

2 COIN-OR

3 Modeling Software

4 Python-based Modeling Tools
   - PuLP/DipPy
   - CyLP
   - yaposib
   - Pyomo
The COIN-OR Foundation

- A non-profit foundation promoting the development and use of interoperable, open-source software for operations research.
- A consortium of researchers in both industry and academia dedicated to improving the state of computational research in OR.
- A venue for developing and maintaining standards.
- A forum for discussion and interaction between practitioners and researchers.

The COIN-OR Repository

- A collection of interoperable software tools for building optimization codes, as well as a few stand alone packages.
- A venue for peer review of OR software tools.
- A development platform for open source projects, including a wide range of project management tools.

See www.coin-or.org for more information.
We currently have 50+ projects and more are being added all the time.

COIN-OR has solvers for most common optimization problem classes.

- Linear programming
- Nonlinear programming
- Mixed integer linear programming
- Mixed integer nonlinear programming (convex and nonconvex)
- Stochastic linear programming
- Semidefinite programming
- Graph problems
- Combinatorial problems (VRP, TSP, SPP, etc.)
- Graphics and visualization

COIN-OR has various utilities for reading/building/manipulating/preprocessing optimization models and getting them into solvers.

COIN-OR has overarching frameworks that support implementation of broad algorithm classes.

- Parallel search
- Branch and cut (and price)
- Decomposition-based algorithms
The COIN-OR Optimization Suite

- **COIN-OR** distributes a free and open source suite of software that can handle all the classes of problems we’ll discuss.
  - Clp (LP)
  - Cbc (MILP)
  - Ipopt (NLP)
  - **SYMPHONY** (MILP, BMILP)
  - DIP (MILP)
  - Bonmin (Convex MINLP)
  - Couenne (Non-convex MINLP)
  - Optimization Services (Interface)

- COIN also develops **standards and interfaces** that allow software components to interoperate.

- Check out the Web site for the project at [http://www.coin-or.org](http://www.coin-or.org)
Modular Structure of the Suite

- One of the hallmarks of good open source tools is *modularity*.
- The suite is made up of building blocks with well-defined interfaces that allow construction of higher level tools.
- There have been 75 authors over time and most have never coordinated directly with each other!
- This is the open source model of development.
The CoinUtils project contains a wide range of low-level utilities used in almost every project in suite.

- Factorization
- File parsing
- Sparse matrix and array storage
- Presolve
- Memory management
- Model building
- Parameter parsing
- Timing
- Basic data structures
Basic Building Blocks: Open Solver Interface

Uniform API for a variety of solvers:

- CBC
- CLP
- CPLEX
- DyLP
- FortMP
- XPRESS-MP
- GLPK
- Mosek
- OSL
- Soplex
- SYMPHONY
- Volume Algorithm

- Read input from MPS or CPLEX LP files or construct instances using COIN-OR data structures.
- Manipulate instances and output to MPS or LP file.
- Set solver parameters.
- Calls LP solver for LP or MIP LP relaxation.
- Manages interaction with dynamic cut and column generators.
- Calls MIP solver.
- Returns solution and status information.
Building Blocks: Cut Generator Library

- A collection of cutting-plane generators and management utilities.
- Interacts with OSI to inspect problem instance and solution information and get violated cuts.
- Cuts include:
  - Combinatorial cuts: AllDifferent, Clique, KnapsackCover, OddHole
  - Flow cover cuts
  - Lift-and-project cuts
  - Mixed integer rounding cuts
  - General strengthening: DuplicateRows, Preprocessing, Probing, SimpleRounding
Getting the COIN-OR Optimization Suite

- Source builds out of the box on Windows, Linux, OSX using the Gnu autotools (or with Visual Studio project files on Windows).
- Source is available from
  
  https://www.coin-or.org
  https://github.com/coin-or

- Packages are available to install on many Linux distros.
- Homebrew recipes are available for many projects on OSX (we are working on this).
- Binaries are automatically built and deployed here:
  
  https://bintray.com/coin-or/download

- For many more details, see Lecture 1 of this tutorial (slightly out of date now):
  
  http://coral.ie.lehigh.edu/ ted/teaching/coin-or
Outline

1. Introduction
2. COIN-OR
3. Modeling Software
4. Python-based Modeling Tools
   - PuLP/DipPy
   - CyLP
   - yaposib
   - Pyomo
Most existing modeling software can be used with COIN solvers.

- **Commercial Systems**
  - GAMS
  - MPL
  - AMPL
  - AIMMS

- **Python-based Open Source Modeling Languages and Interfaces**
  - Pyomo
  - PuLP/Dippy
  - CyLP (provides API-level interface)
  - yaposib
Other Front Ends (mostly open source)

- FLOPC++ (algebraic modeling in C++)
- CMPL
- MathProg.jl (modeling language built in Julia)
- GMPL (open-source AMPL clone)
- ZMPL (stand-alone parser)
- SolverStudio (spreadsheet plug-in: www.OpenSolver.org)
- Open Office spreadsheet
- R (RSymphony Plug-in)
- Matlab (OPTI)
- Mathematica
- \textit{Sage}!
Although not required, it’s useful to know something about how modeling languages interface with solvers. In many cases, modeling languages interface with solvers by writing out an intermediate file that the solver then reads in. It is also possible to generate these intermediate files directly from a custom-developed code.

**Common file formats**

- **MPS format**: The original standard developed by IBM in the days of Fortran, not easily human-readable and only supports (integer) linear modeling.
- **LP format**: Developed by CPLEX as a human-readable alternative to MPS.
- **.nl format**: AMPL’s intermediate format that also supports non-linear modeling.
- **OSIL**: an open, XML-based format used by the Optimization Services framework of COIN-OR.

Several projects use **Python C Extensions** to get the data into the solver through memory.
Outline

1 Introduction

2 COIN-OR

3 Modeling Software

4 Python-based Modeling Tools
   - PuLP/DipPy
   - CyLP
   - yaposib
   - Pyomo
Where to Get the Examples

- The remainder of the talk will review a wide range of examples.
- These and many other examples of modeling with Python-based modeling languages can be found at the below URLs.

- [https://github.com/tkralphs/FinancialModels](https://github.com/tkralphs/FinancialModels)
- [http://projects.coin-or.org/browser/Dip/trunk/Dip/src/dippy/examples](http://projects.coin-or.org/browser/Dip/trunk/Dip/src/dippy/examples)
- [https://github.com/Pyomo/PyomoGallery/wiki](https://github.com/Pyomo/PyomoGallery/wiki)
- [https://github.com/coin-or/pulp/tree/master/examples](https://github.com/coin-or/pulp/tree/master/examples)
- [https://pythonhosted.org/PuLP/CaseStudies](https://pythonhosted.org/PuLP/CaseStudies)
1 Introduction

2 COIN-OR

3 Modeling Software

4 Python-based Modeling Tools
   - PuLP/DipPy
   - CyLP
   - yaposib
   - Pyomo
PuLP is a modeling language in COIN-OR that provides data types for Python that support algebraic modeling.

PuLP only supports development of linear models.

Main classes
- LpProblem
- LpVariable

Variables can be declared individually or as “dictionaries” (indexed sets).

We do not need an explicit notion of a parameter or set here because Python provides data structures we can use.

In PuLP, models are technically “concrete,” since the model is always created with knowledge of the data.

However, it is still possible to maintain a separation between model and data.

Developer: Stuart Mitchell

pip install pulp
from pulp import LpProblem, LpVariable, lpSum, LpMaximize, value

prob = LpProblem("Dedication Model", LpMaximize)

X1 = LpVariable("X1", 0, None)
X2 = LpVariable("X2", 0, None)

prob += 4*X1 + 3*X2
prob += X1 + X2 <= 100
prob += 2*X1 + X2 <= 150
prob += 3*X1 + 4*X2 <= 360

prob.solve()

print 'Optimal total cost is: ', value(prob.objective)

print "X1 :", X1.varValue
print "X2 :", X2.varValue
```python
from pulp import LpProblem, LpVariable, lpSum, LpMaximize, value
from bonds import bonds, max_rating, max_maturity, max_cash

prob = LpProblem("Bond Selection Model", LpMaximize)

buy = LpVariable.dicts('bonds', bonds.keys(), 0, None)

prob += lpSum(bonds[b]['yield'] * buy[b] for b in bonds)

prob += lpSum(buy[b] for b in bonds) <= max_cash, "cash"

prob += (lpSum(bonds[b]['rating'] * buy[b] for b in bonds)
         <= max_cash*max_rating, "ratings")

prob += (lpSum(bonds[b]['maturity'] * buy[b] for b in bonds)
         <= max_cash*max_maturity, "maturities")
```
bonds = {'A' : {'yield' : 4,
              'rating' : 2,
              'maturity' : 3},
       'B' : {'yield' : 3,
              'rating' : 1,
              'maturity' : 4},

max_cash = 100
max_rating = 1.5
max_maturity = 3.6
Notes About the Model

- We can use Python’s native `import` mechanism to get the data.
- Note, however, that the data is read and stored *before* the model.
- This means that we don’t need to declare sets and parameters.

Constraints

- Naming of constraints is optional and only necessary for certain kinds of post-solution analysis.
- Constraints are added to the model using an intuitive syntax.
- Objectives are nothing more than expressions without a right hand side.

Indexing

- Indexing in Python is done using the native dictionary data structure.
- Note the extensive use of comprehensions, which have a syntax very similar to quantifiers in a mathematical model.
Notes About the Data Import

- We are storing the data about the bonds in a “dictionary of dictionaries.”
- With this data structure, we don’t need to separately construct the list of bonds.
- We can access the list of bonds as `bonds.keys()`.
- Note, however, that we still end up hard-coding the list of features and we must repeat this list of features for every bond.
- We can avoid this using some advanced Python programming techniques, but how to do this with SolverStudio later.
prob.solve()

epsilon = .001

print 'Optimal purchases:'
for i in bonds:
    if buy[i].varValue > epsilon:
        print 'Bond', i, ':', buy[i].varValue
More Complexity: Facility Location Problem

- We have $n$ locations and $m$ customers to be served from those locations.
- There is a fixed cost $c_j$ and a capacity $W_j$ associated with facility $j$.
- There is a cost $d_{ij}$ and demand $w_{ij}$ for serving customer $i$ from facility $j$.
- We have two sets of binary variables.
  - $y_j$ is 1 if facility $j$ is opened, 0 otherwise.
  - $x_{ij}$ is 1 if customer $i$ is served by facility $j$, 0 otherwise.

### Capacitated Facility Location Problem

$$\begin{align*}
\text{min} & \quad \sum_{j=1}^{n} c_j y_j + \sum_{i=1}^{m} \sum_{j=1}^{n} d_{ij} x_{ij} \\
\text{s.t.} & \quad \sum_{j=1}^{n} x_{ij} = 1 \quad \forall i \\
& \quad \sum_{i=1}^{m} w_{ij} x_{ij} \leq W_j \quad \forall j \\
& \quad x_{ij} \leq y_j \quad \forall i, j \\
& \quad x_{ij}, y_j \in \{0, 1\} \quad \forall i, j
\end{align*}$$
PuLP Model: Facility Location Example

from products import REQUIREMENT, PRODUCTS
from facilities import FIXED_CHARGE, LOCATIONS, CAPACITY

prob = LpProblem("Facility_Location")

ASSIGNMENTS = [(i, j) for i in LOCATIONS for j in PRODUCTS]
assign_vars = LpVariable.dicts("x", ASSIGNMENTS, 0, 1, LpBinary)
use_vars = LpVariable.dicts("y", LOCATIONS, 0, 1, LpBinary)

prob += lpSum(use_vars[i] * FIXED_COST[i] for i in LOCATIONS)

for j in PRODUCTS:
    prob += lpSum(assign_vars[(i, j)] for i in LOCATIONS) == 1

for i in LOCATIONS:
    prob += lpSum(assign_vars[(i, j)] * REQUIREMENT[j] for j in PRODUCTS) <= CAPACITY * use_vars[i]

prob.solve()

for i in LOCATIONS:
    if use_vars[i].varValue > 0:
        print "Location ", i, " is assigned: ",
        print [j for j in PRODUCTS if assign_vars[(i, j)].varValue > 0]
PuLP Data: Facility Location Example

# The requirements for the products
REQUIREMENT = {
    1 : 7,
    2 : 5,
    3 : 3,
    4 : 2,
    5 : 2,
}

# Set of all products
PRODUCTS = REQUIREMENT.keys()
PRODUCTS.sort()

# Costs of the facilities
FIXED_COST = {
    1 : 10,
    2 : 20,
    3 : 16,
    4 : 1,
    5 : 2,
}

# Set of facilities
LOCATIONS = FIXED_COST.keys()
LOCATIONS.sort()

# The capacity of the facilities
CAPACITY = 8
DIP (w/ M. Galati and J. Wang)

A software framework and stand-alone solver for implementation and use of a variety of decomposition-based algorithms.

- Decomposition-based algorithms have traditionally been difficult to implement and compare.
- Abstracts the common, generic elements of these methods.
  - Key: API is in terms of the compact formulation.
  - The framework takes care of reformulation and implementation.
  - DIP is now a fully generic decomposition-based parallel MILP solver.

DipPy (w/ M. O’Sullivan)

- PuLP plus a Python C extension.
- User can express decompositions in a “natural” way.
- Allows access to multiple decomposition methods and callbacks.

← Joke!
from products import REQUIREMENT, PRODUCTS
from facilities import FIXED_CHARGE, LOCATIONS, CAPACITY

prob = dippy.DipProblem("Facility_Location")

ASSIGNMENTS = [(i, j) for i in LOCATIONS for j in PRODUCTS]
assign_vars = LpVariable.dicts("x", ASSIGNMENTS, 0, 1, LpBinary)
use_vars = LpVariable.dicts("y", LOCATIONS, 0, 1, LpBinary)

prob += lpSum(use_vars[i] * FIXED_COST[i] for i in LOCATIONS)

for j in PRODUCTS:
    prob += lpSum(assign_vars[(i, j)] for i in LOCATIONS) == 1

for i in LOCATIONS:
    prob.relaxation[i] += lpSum(assign_vars[(i, j)] * REQUIREMENT[j] for j in PRODUCTS) <= CAPACITY * use_vars[i]

dippy.Solve(prob, doPriceCut:1)

for i in LOCATIONS:
    if use_vars[i].varValue > 0:
        print "Location ", i, " is assigned: ",
        print [j for j in PRODUCTS if assign_vars[(i, j)].varValue > 0]
def solve_subproblem(prob, index, redCosts, convexDual):
    ...
    return knapsack01(obj, weights, CAPACITY)
def knapsack01(obj, weights, capacity):
    ...
    return solution
def first_fit(prob):
    ...
    return bvs
prob.init_vars = first_fit
def choose_branch(prob, sol):
    ...
    return ([], down_branch_ub, up_branch_lb, [])
def generate_cuts(prob, sol):
    ...
    return new_cuts
def heuristics(prob, xhat, cost):
    ...
    return sols
dippy.Solve(prob, {'doPriceCut': '1'})
Introduction

COIN-OR

Modeling Software

Python-based Modeling Tools

- PuLP/DipPy
- CyLP
- yaposib
- Pyomo
CyLP: Low-level Modeling and API for Cbc/Clp/Cgl

CyLP provides a low-level modeling language for accessing details of the algorithms and low-level parts of the API.

The included modeling language is “close to the metal”, works directly with numerical data with access to low-level data structures.

- **Clp**
  - Pivot-level control of algorithm in Clp.
  - Access to fine-grained results of solve.

- **Cbc**
  - Python classes for customization

- **Cgl**
  - Python class for building cut generators wrapped around Cgl.

**Developers**: Mehdi Towhidi and Dominique Orban
lp = CyClpSimplex()
x = lp.addVariable('x', numVars)
lp += x_u >= x >= 0

lp += A * x <= b if cons_sense == '<=' else A * x >= b

lp.objective = -c * x if obj_sense == 'Max' else c * x
lp.primal(startFinishOptions = 1)
numCons = len(b)
print 'Current solution is', lp.primalVariableSolution['x']
print 'Current tableaux is', lp.tableaux
for row in range(lp.nConstraints):
    print 'Variables basic in row', row, 'is', lp.basicVariables[row],
    print 'and has value' lp.rhs[row]
Introduction

2 COIN-OR

3 Modeling Software

4 Python-based Modeling Tools
   - PuLP/DipPy
   - CyLP
   - yaposib
   - Pyomo
Provides Python bindings to any solver with an OSI interface

```python
solver = yaposib.available_solvers()[0]

for filename in sys.argv[1:]:
    problem = yaposib.Problem(solver)
    print("Will now solve %s" % filename)
    err = problem.readMps(filename)
    if not err:
        problem.solve()
        if problem.status == 'optimal':
            print("Optimal value: %f" % problem.obj.value)
            for var in problem.cols:
                print("\t%s = %f" % (var.name, var.solution))
        else:
            print("No optimal solution could be found.")
```
Introduction

COIN-OR

Modeling Software

Python-based Modeling Tools
- PuLP/DipPy
- CyLP
- yaposib
- Pyomo
Pyomo

- An algebraic modeling language in Python similar to PuLP.
- Can import data from many sources, including AMPL-style data files.
- More powerful, includes support for nonlinear modeling.
- Allows development of both concrete models (like PuLP) and abstract models (like other AMLs).
- Also include PySP for stochastic Programming.

Primary classes

- `ConcreteModel`, `AbstractModel`
- `Set`, `Parameter`
- `Var`, `Constraint`

**Developers**: Bill Hart, John Siirola, Jean-Paul Watson, David Woodruff, and others...

- More on Wednesday 9:00 AM!
Thank You!

- There seems to be great potential in incorporating more of what COIN-OR offers into Sage.
- I’d be happy to hear from anyone who agrees about potential future projects.

Questions?