Introduction to Optimization using MOSEK and Python. MOSEK technical report: TR-2010-1.

Joachim Dahl*

Revised on May 18th, 2012.

Abstract

Python is an extremely popular open source programming language. One reason is that Python is like the Swiss army knife of programming; indeed it includes tools for almost any programming task, e.g., database connectivity and XML processing. Moreover, Python is interactive (interpreted) which is very convenient for experimentation, rapid prototyping and scripting. All these features make Python a good choice for building optimization applications and that is the reason why most optimization vendors nowadays include a Python interface to their optimization engines. In this short correspondence we introduce one such interface, i.e., the Python interface to MOSEK.

1 Introduction

One of the obstacles faced by optimization practitioners is the sometimes difficult transition from a mathematical formulation of an optimization problem to a vendor specific software formulation. Several software packages exist that provide a more unified and convenient interface to different optimization solvers, e.g., AMPL, GAMS, MPL and AIMMS, or MATLAB packages such as Yalmip and CVX.

Such interfaces have proved to be an efficient aid to users of optimization software, but they are not easily interfaced to the large number of external libraries typically needed when embedding an optimization algorithm into a larger application. Over the last decade the Python programming language has become a very popular language partly because of its very large number of available interfaces to existing programs and libraries, e.g., all major database formats, networking protocols, excellent visualization toolkits, office packages, etc. For scientists and engineers Python is also rapidly gaining popularity with the development of numerical and scientific packages such as:

- Numpy and SciPy which allows Python to be used interactively to manipulate and visualize data much as in, e.g., MATLAB.
- CVXOPT, CVXMOD, OpenOpt, PyMathProg, Pyomo for optimization.
- SymPy for symbolic algebra.
- NetworkX for graph computations.

The SAGE project developed in Python provides a free and viable alternative to, e.g., Magma, Maple and Mathematica and includes different scientific Python packages in a common framework, and demonstrates that developing large software projects using Python is very feasible.

In the following sections we give an introduction to the MOSEK Python interface and give examples of Python optimization applications using MOSEK. The interested reader can find more information about the MOSEK solvers and interfaces at `www.mosek.com`, and additional information about the Python programming language can be found at `www.python.org`.

*MOSEK ApS, Fruebjergvej 3, Box 16, 2100 Copenhagen, Denmark. Email: support@mosek.com

2 Examples

We next give a few simple examples of optimization problems solved using MOSEK. A typical MOSEK program contains the following 5 steps.

1. Create an environment.
2. Create an optimization task.
3. Load a problem into the task object.
4. Optimization.
5. Extracting the solution.

2.1 A simple linear programming problem

The first example solves a simple linear programming problem

$$\begin{array}{lrcrcl} \text{minimize} & 1x_0 & + & 1x_1 & & \\ \text{subject to} & 1x_0 & - & 2x_1 & \leq & 0, \\ & 1x_0 & + & 1x_1 & \geq & 2, \end{array} \tag{1}$$

having the bounds

$$\begin{array}{rcrcl} 0 & \leq & x_0 & \leq & \infty, \\ 0 & \leq & x_1 & \leq & 10. \end{array} \tag{2}$$

The program for solving the linear programming problem is given below:

```
import sys, mosek
from mosek.array import array, zeros, ones

inf = 0.0

# Make a MOSEK environment
env = mosek.Env ()
task = env.Task(0,0)
# Attach a printer to the task
task.set_Stream (mosek.streamtype.log, lambda x: sys.stdout.write(x))

task.append(mosek.accmode.con,2)  # number of constraints
task.append(mosek.accmode.var,2)  # number of variables
task.putclist([0, 1], [1.0, 1.0]) # setup objective

# setup bounds on variables and constraints
task.putbound(mosek.accmode.var, 0, mosek.boundkey.lo, 0.0, +inf)
task.putbound(mosek.accmode.var, 1, mosek.boundkey.ra, 0.0, 10)
task.putbound(mosek.accmode.con, 0, mosek.boundkey.up, -inf, 0.0)
task.putbound(mosek.accmode.con, 1, mosek.boundkey.lo, 2.0, +inf)

# input coefficient matrix column by column
task.putavec(mosek.accmode.var, 0, array([0,1]), array([1.0,1.0]))
task.putavec(mosek.accmode.var, 1, array([0,1]), array([-2.0,1.0]))

# Optimize the task
task.putobjsense(mosek.objsense.minimize)
task.optimize()
xx = zeros(2, float)
task.getsolutionslice(mosek.soltype.bas, mosek.solitem.xx, 0, 2, xx)
print "solution: %s" % xx
```

Although the program is neither extremely short or pretty, it is certainly shorter and simpler than corresponding C code, which is often used for creating and specifying optimization problems.

2.2 Data Envelopment Analysis

We next consider data envelopment analysis (DEA) which is a popular management tool to evaluate the efficiency of a number of producers or *decision making units*. In DEA we solve a problem of the form

$$\begin{array}{ll} \text{minimize} & \Theta \\ \text{subject to} & Y\lambda \geq Y_k \\ & \Theta X_k - X\lambda \geq 0 \\ & \lambda \geq 0 \end{array} \tag{3}$$

in the variables (Θ, λ) where k is the index of the decision making unit (DMU) to evaluate and X and Y denote the input and output of the DMUs, respectively. A complete Python example for solving a DEA instance is given below:

```
import sys, mosek as msk

def dea(data, ninputs, dmu):
    '''Solves the DEA linear programming problem

    minimize    theta
    subject to  Y*lambda >= Y[dmu]
                theta*X[dmu] - X*lambda >= 0
                lambda >= 0

    X is stored in the first 'ninputs' positions of 'data', Y is stored
    in the remaining positions, and 'dmu' denotes the DMU we evaluate.
    '''
    inf = 0.0  # numeric value doesn't matter
    vars = [x for x in data ]
    mx, my, n = ninputs, len(data[dmu]) - ninputs, len(data) + 1
    asub, acof, colcnt = [], [], (n+1)*[0]
    nz, idx = 0, 0
    for v in vars:
        acof += data[v]
        asub += range(mx+my)
        nz += mx+my
        colcnt[idx+1] = nz
        idx += 1

    acof += [ -xi for xi in data[dmu][:ninputs] ]
    asub += range(mx)
    colcnt[idx+1] = colcnt[idx] + mx

    env = msk.Env()
    env.init ()
    task = env.Task(0,0)
    task.set_Stream (msk.streamtype.log, lambda x: sys.stdout.write(x))
    task.inputdata (mx+my,    # number of constraints
                    n,        # number of variables
                    (n-1)*[0.0] + [1.0], # linear objective coefficients
                    0.0,      # objective fixed value
                    colcnt[:-1], colcnt[1:],
                    asub, acof,
                    mx*[ msk.boundkey.up ] + my*[ msk.boundkey.lo ], # bkc
                    mx*[-inf] + data[dmu][ninputs:], # blc
                    mx*[0] + my*[inf],   # buc
                    (n-1)*[ msk.boundkey.lo ] + [ msk.boundkey.fr ], # bkx
                    (n-1)*[0] + [-inf],  # blx
                    (n-1)*[inf] + [inf]  # bux
                    )

    task.putobjsense(msk.objsense.minimize)
    task.optimize()
    x = msk.array.zeros(n, float)
    task.getsolutionslice(msk.soltype.bas, msk.solitem.xx, 0, n, x)

    theta, lam = x[-1], dict()
    for k in range(len(vars)): lam[vars[k]] = x[k]

    return theta, lam

data = {'DMU1': [ 5, 14, 9, 4, 16],
        'DMU2': [ 8, 15, 5, 7, 10],
        'DMU3': [ 7, 12, 4, 9, 13]}

for dmu in ['DMU1','DMU2','DMU3']:
    theta, lam = dea(data, 2, dmu)
    print "Efficiency for %s : %1.3e" %(dmu, theta)
```

We consider a scenario with 3 DMUs each taking two inputs and producing three outputs, according to the table

DMU	Input		Output		
1	5	14	9	4	16
2	8	15	5	7	10
3	7	12	4	9	14

and when we run the DEA example, we get the following output illustrating that both DMU 1 and 3 are efficient, but DMU 2 is not.

```
Efficiency for DMU1 : 1.000e+00
Efficiency for DMU2 : 7.733e-01
Efficiency for DMU3 : 1.000e+00
```

2.3 Basis pursuit

Often we seek a sparse solution to

$$Ax = b,$$

i.e., we wish to express b using the least number of columns in A. The *basis pursuit* algorithm finds an ℓ_1 approximation by solving the linear programming problem

$$\begin{array}{ll} \text{minimize} & \|x\|_1 \\ \text{subject to} & Ax = b. \end{array}$$

The ℓ_1 approximation shares the same problem as other *shrinkage* algorithms, namely that large coefficients are over-penalized, and one improvement is to solve a sequence of problems

$$\begin{array}{ll} \text{minimize} & \|\,\mathbf{diag}(w^k)x^k\|_1 \\ \text{subject to} & Ax^k = b, \end{array}$$

where the weights $w_i^k = \frac{1}{|x_i^{k-1}|+\epsilon}$ evens out the magnitude dependency of the different components. Solving such a sequence of problems therefore often results in a solution with fewer non-zero elements than the traditional ℓ_1 solution. The program below illustrates how to implement this concept in MOSEK. In the example, we have

$$A = \begin{bmatrix} 2 & 1 & 1 \\ 1 & 1 & 2 \end{bmatrix}, \qquad b = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \qquad \epsilon = 10^{-1},$$

and the example identifies the correct *minimum cardinality* solution

$$x^1 = \begin{bmatrix} 1/3 \\ 0 \\ 1/3 \end{bmatrix}, \qquad x^2 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}.$$

Of course, there is no guarantee that the sparsest solution will be obtained in general by the reweighted ℓ_1 heuristic.

```
import sys, mosek
from mosek.array import zeros

inf = 0.0

def rwl1(A, b, wi):
    '''
    minimize                    sum(t)
    subject to  -infty  <= x - diag(wi)*t <=    0
                   0    <= x + diag(wi)*t <=  infty
                   b    <=       A*x      <=    b
                   0    <=        t       <=  infty
    '''
    m, n = len(A), len(A[0])
    # Make a MOSEK environment
    env = mosek.Env ()
    task = env.Task(0,0)
    # Attach a printer to the task
    task.set_Stream (mosek.streamtype.log, lambda x: sys.stdout.write(x))

    task.append(mosek.accmode.var,2*n)  # number of variables
    task.append(mosek.accmode.con,2*n+m)  # number of constraints
    task.putclist(range(2*n), n*[0.0]+n*[1.0]) # setup objective

    # setup bounds on variables and constraints
    task.putboundslice(mosek.accmode.var,
                       0, n, n*[mosek.boundkey.fr], n*[-inf],  n*[inf])
    task.putboundslice(mosek.accmode.var,
                       n, 2*n, n*[mosek.boundkey.lo], n*[0.0], n*[inf])
    task.putboundslice(mosek.accmode.con,
                       0, n,     n*[mosek.boundkey.up], n*[-inf], n*[0.])
    task.putboundslice(mosek.accmode.con,
                       n, 2*n,   n*[mosek.boundkey.lo], n*[ 0. ], n*[inf])
    task.putboundslice(mosek.accmode.con,
                       2*n, 2*n+m, m*[mosek.boundkey.fx], b, b)

    # input A matrix row by row
    for i in range(n):
        task.putavec(mosek.accmode.con, i,   [i, i+n], [1., -wi[i] ])
        task.putavec(mosek.accmode.con, i+n, [i, i+n], [1.,  wi[i] ])

    for i in range(m):
        task.putavec(mosek.accmode.con, i+2*n, range(n), A[i])

    # Optimize the task
    task.putobjsense(mosek.objsense.minimize)

    task.optimize()
    task.solutionsummary(mosek.streamtype.log)
    x = zeros(n, float)
    task.getsolutionslice(mosek.soltype.bas, mosek.solitem.xx, 0, n, x)
    return x

A = [ [2., 1., 1.], [1., 1., 2.] ]
b = [1., 1.]

eps = 0.1
x1 = rwl1(A, b, [1.0, 1.0, 1.0])
x2 = rwl1(A, b, [1.0/(abs(xi)+eps) for xi in x1 ])
```

3 Interactive Python Inspection

Python is also a useful tool for interactively inspecting the data of an optimization problem, and for example, reoptimize a problem after changing parts of the problem. The following example demonstrates

how simple it is to read an existing problem into MOSEK from an MPS file and inspect the constraints.

```
import mosek
from mosek.array import array, zeros

env = mosek.Env()
task = env.Task(0, 0)
task.readdata('dea.mps')

# find the position of largest magnitude in the first constraint
sub = zeros( task.getnumvar() )
val = array( task.getnumvar()*[0.0] )

nzi = task.getavec(mosek.accmode.con, 0, sub, val)
val = [abs(vi) for vi in val][:nzi]
print val.index(max(val))
```

4 Conclusion

In this short paper we have demonstrated that MOSEK can easily be used from Python. In many cases this provides a viable and cost-effective alternative to, e.g., dedicated modeling languages such as AMPL, GAMS, etc., and at the same time offers the user the flexibility of a modern programming language. A good example of the flexibility of Python is the DEA example provided in §2.2, which has (easily) been integrated into a webserver application by a MOSEK customer.

One the other hand, Python is also very useful for developing prototypes, debugging larger applications using MOSEK, or for inspecting data, for example by loading MPS files from a Python shell.