Quadratic Programming with Python and CVXOPT

This guide assumes that you have already installed the NumPy and CVXOPT packages for your Python distribution. You can check if they are installed by importing them:

```
import numpy
import cvxopt
```

Let us first review the standard form of a QP (following CVXOPT notation):

$$\begin{aligned} \min_x \quad & \tfrac{1}{2} x^\top P x + q^\top x \\ \text{subject to} \quad & Gx \preceq h \\ & Ax = b \end{aligned}$$

Note that $x^\top$ denotes the transpose of x, and $Gx \preceq h$ means that the inequality is taken element-wise over the vectors Gx and h. The above objective function is convex if and only if P is positive-semidefinite, which is the realm we are concerned with.[1]

The CVXOPT QP framework expects a problem of the above form, defined by the parameters $\{P, q, G, h, A, b\}$; P and q are required, the others are optional. Alternate QP formulations must be manipulated to conform to the above form; for example, if the inequality constraint was expressed as $Gx \succeq h$, then it can be rewritten $-Gx \preceq -h$. Also, to specify lower and upper bounds on x, an identity matrix can form part of G, since $x \preceq u$ is equivalent to $Ix \preceq u$. Note that x itself is not provided to the solver, since it is an internal variable being optimized over. In particular, this means that the solver has no explicit knowledge of x itself; everything is implicity defined by the supplied parameters. It is essential that the same variable order is maintained for the relevant parameters (e.g., q_i, h_i, b_i should correspond to variable x_i).

[1] Non-convexity implies the existence of local optima, making it difficult to find global optima.

Let us consider a simple example:

$$\begin{aligned} \min_{x,y} \quad & \tfrac{1}{2}x^2 + 3x + 4y \\ \text{subject to} \quad & x, y \geq 0 \\ & x + 3y \geq 15 \\ & 2x + 5y \leq 100 \\ & 3x + 4y \leq 80 \end{aligned}$$

First, we rewrite the above in the given standard form:

$$\begin{aligned} \min_{x,y} \quad & \tfrac{1}{2} \begin{bmatrix} x \\ y \end{bmatrix}^\top \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 3 \\ 4 \end{bmatrix}^\top \begin{bmatrix} x \\ y \end{bmatrix} \\ & \begin{bmatrix} -1 & 0 \\ 0 & -1 \\ -1 & -3 \\ 2 & 5 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \preceq \begin{bmatrix} 0 \\ 0 \\ -15 \\ 100 \\ 80 \end{bmatrix} \end{aligned}$$

By inspection, the variable is defined by $\begin{bmatrix} x \\ y \end{bmatrix}$, and the parameters P, q, G, h are given. Note how we collapsed all inequality constraints into a single G matrix of the standard form. Since there are no equality constraints, we do not need to provide the empty A, b. Note that even though y^2 did not appear in the original objective, we had to include it with zero coefficients in P because the solver parameters must be defined using the full set of variables. Even if certain variables only appear in constraints, they will still need to be expressed with zero coefficients in the objective parameters, and *vice versa*.

Let us first define the above parameters in Python. CVXOPT supplies its own matrix object; all arguments given to its solvers must be in this matrix type. There are two ways to do this. The first is to define the matrix directly with (potentially nested) lists:

```
from cvxopt import matrix
P = matrix([[1.0,0.0],[0.0,0.0]])
q = matrix([3.0,4.0])
G = matrix([[-1.0,0.0,-1.0,2.0,3.0],[0.0,-1.0,-3.0,5.0,4.0]])
h = matrix([0.0,0.0,-15.0,100.0,80.0])
```

Observe how we defined the lists to contain real numbers (doubles) instead of integers, and that G was defined by its columns instead of its rows. The CXVOPT solver only accepts matrices containing doubles, and if a list containing only integers was supplied to the matrix constructor, it will create an integer matrix and eventual lead to a cryptic error.

An alternative way, perhaps more convenient if you are familiar with NumPy, is to first create the matrices in NumPy, then call the CVXOPT matrix constructor on them:

```
import numpy
from cvxopt import matrix
P = matrix(numpy.diag([1,0]), tc='d')
q = matrix(numpy.array([3,4]), tc='d')
G = matrix(numpy.array([[-1,0],[0,-1],[-1,-3],[2,5],[3,4]]), tc='d')
h = matrix(numpy.array([0,0,-15,100,80]), tc='d')
```

Here we created integer NumPy arrays and matrices because we used the `tc='d'` option to explicitly construct a matrix of doubles (this could work for the previous example as well).

The hard work is mostly over now! As you will often find, formulating the problem is usually the hard step. Invoking a solver is straightforward:

```
from cvxopt import solvers
sol = solvers.qp(P,q,G,h)
```

That's it! If you had A, b as well, you would call:

```
sol = solvers.qp(P,q,G,h,A,b)
```

You can even specify more options, such as the solver used and initial values to try. See the CVXOPT QP documentation in the references on the final page.

Many properties about the solution can be extracted from the `sol` variable (dictionary). In particular, the `'status'`, `'x'`, and `'primal objective'` are probably the most important. If `status` is `optimal`, then the latter two give the optimal solution and its objective value. You can access these fields as you would with a regular Python dictionary:

```
sol['x']                    # [7.13e-07, 5.00e+00]
sol['primal objective']     # 20.0000061731
```

We can verify[2] that $x^* = 0, y^* = 5$, with an optimal value 20.

[2]It is crucial to verify the solution! Don't just trust what the solver gives back to you!

The code is reproduced below for your convenience:

```
# Import the necessary packages
import numpy
from cvxopt import matrix
from cvxopt import solvers

# Define QP parameters (directly)
P = matrix([[1.0,0.0],[0.0,0.0]])
q = matrix([3.0,4.0])
G = matrix([[-1.0,0.0,-1.0,2.0,3.0],[0.0,-1.0,-3.0,5.0,4.0]])
h = matrix([0.0,0.0,-15.0,100.0,80.0])

# Define QP parameters (with NumPy)
P = matrix(numpy.diag([1,0]), tc='d')
q = matrix(numpy.array([3,4]), tc='d')
G = matrix(numpy.array([[-1,0],[0,-1],[-1,-3],[2,5],[3,4]]), tc='d')
h = matrix(numpy.array([0,0,-15,100,80]), tc='d')

# Construct the QP, invoke solver
sol = solvers.qp(P,q,G,h)

# Extract optimal value and solution
sol['x']                    # [7.13e-07, 5.00e+00]
sol['primal objective']     # 20.0000061731
```

References:

`http://abel.ee.ucla.edu/cvxopt/userguide/coneprog.html#quadratic-programming`
Section about QP in CVXOPT user's guide. The example here is not very illuminating; see the second reference below. The section on linear cone programs at the top of the page explains what the fields in the solution dictionary mean.

`http://abel.ee.ucla.edu/cvxopt/examples/tutorial/qp.html`
A simple QP example in CVXOPT.