Quadratic Programming with Python and OpenOpt

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

```python
import numpy
from openopt import QP
```

The following content expands upon the (minimal) documentation provided on the OpenOpt QP webpage [http://openopt.org/QP](http://openopt.org/QP) and the single piece of example code provided there.

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

$$
\begin{align*}
\min \quad & \frac{1}{2} x^\top H x + f^\top x \\
\text{subject to} \quad & l \leq x \leq u \\
& Ax \leq b \\
& A_{eq} x = b_{eq}
\end{align*}
$$

Note that $x^\top$ denotes the transpose of $x$, and the vector inequalities are taken element-wise. The above objective function is convex if and only if $H$ is positive-semidefinite, and is the realm we are concerned with. In particular, the solver introduced below can only handle this type of problem.\textsuperscript{1} If there are elements within $x$ that are unconstrained or only partially constrained, the respective elements in $l$ and $u$ will be $-\infty$ and $+\infty$.

The OpenOpt QP framework expects a problem of the above form, defined by the parameters $\{H, f, l, u, A, b, A_{eq}, b_{eq}\}$; $H$ and $f$ are required, the other are optional. Alternate QP formulations must be manipulated to conform to the above form; for example, if the inequality constraint was expressed as $Ax \geq b$, then it can be rewritten $-Ax \leq -b$. 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 implicitly defined by the supplied parameters. It is essential that the same variable order is maintained for the relevant parameters (e.g., $f_i, l_i, u_i$ should correspond to variable $x_i$).

\textsuperscript{1}Non-convexity implies the existence of local optima, making it difficult to find global optima. To solve for local optima in OpenOpt, use the framework for non-linear problems: [http://openopt.org/NLP](http://openopt.org/NLP)
Let us consider a simple example:

\[
\begin{align*}
\min_{x,y} & \quad \frac{1}{2} x^2 + 3x + 4y \\
\text{subject to} & \quad x, y \geq 0 \\
& \quad x + 3y \geq 15 \\
& \quad 2x + 5y \leq 100 \\
& \quad 3x + 4y \leq 80
\end{align*}
\]

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

\[
\begin{align*}
\min_{x,y} & \quad \frac{1}{2} \begin{bmatrix} x \\ y \end{bmatrix}^T \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 3 \\ 4 \end{bmatrix}^T \begin{bmatrix} x \\ y \end{bmatrix} \\
\text{subject to} & \quad \begin{bmatrix} 0 \\ 0 \end{bmatrix} \leq \begin{bmatrix} x \\ y \end{bmatrix} \bigg( \leq \begin{bmatrix} +\infty \\ +\infty \end{bmatrix} \bigg) \\
& \quad \begin{bmatrix} -1 & -3 \\ 2 & 5 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \leq \begin{bmatrix} -15 \\ 100 \\ 80 \end{bmatrix}
\end{align*}
\]

By inspection, the variable is defined by \( \begin{bmatrix} x \\ y \end{bmatrix} \), and the parameters \( H, f, l, A, b \) are given. \( u \) was also given above (in parentheses), but the +\( \infty \) upper bound (i.e., unconstrained) is assumed if \( u \) is not provided. Since there are no equality constraints, we do not need to provide the empty \( A_{eq}, b_{eq} \). Note that even though \( y^2 \) did not appear in the original objective, we had to include it with zero coefficients in \( H \) 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 \textit{vice versa}. Finally, note that \( H \) here is singular; we will return to this issue later.

Let us first define the above parameters in Python:

```python
import numpy

myH = numpy.diag([1,0])
myf = numpy.array([3,4])
myl = numpy.zeros(2)
myA = numpy.array([[-1,-3],[2,5],[3,4]])
myb = numpy.array([-15,100,80])
# myu = numpy.inf * numpy.ones(2)  # We won’t use this, just for reference
```
Next, we construct a QP instance within OpenOpt:

```python
from openopt import QP
p = QP(H=myH, f=myf, lb=myl, A=myA, b=myb)
```

Note that because all arguments apart from $H$ and $f$ are optional, you need to specify which parameters you are providing, hence the `A=myA` arguments. Here’s how to specify all of them:

```python
p = QP(H=myH, f=myf, lb=myl, ub=myu, A=myA, b=myb, Aeq=myAeq, beq=mybeq)
```

In our case, we only provided some of the arguments; the rest are assumed vacuous. Finally, note that because $H$ and $f$ are required, we could have alternatively written:

```python
p = QP(myH, myf, lb=myl, A=myA, b=myb)
```

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 if your problem is ‘well-behaved’:

```python
sol = p.solve('qlcp')
```

That’s it! All you need to do is call `solve` and specify the solver, and we chose the `qlcp` solver that is provided with OpenOpt. Other solvers are given on the website [http://openopt.org/QP](http://openopt.org/QP) of which `cvxopt_qp` is a good alternative, but requires further installation. Note that you can only call `p.solve` once; to solve again, construct another QP.

In fact, if you followed along and tried to solve our example problem, the above step should give you an error. It gives a `LinAlgError` with description ‘Singular matrix’. Looking at our example problem, the only matrix that could have given this error is $H$, which is indeed singular. The reason this comes up is because `qlcp` inverts $H$ when solving the problem. This is bad news in general, because matrix inversion is computationally expensive and usually destroys sparsity in matrices, making it inappropriate for large problems. Solvers like `cvxopt_qp` handle these problems more elegantly. For us, however, we can do with an old trick: perturb $H$ to make it non-singular. We add some small $\epsilon = 10^{-3}$ to the diagonal:

```python
myH = numpy.diag([1,0]) + (1e-3)*numpy.eye(2)
```

If we re-construct and solve our QP using this new `myH`, you will see the optimization run without error. To extract the optimal value and solution:

```python
opt_val = p.ff  # opt_val == array([20.0125])
opt_sol = p.xf  # opt_sol == array([-7.2359e-13, 5.0000e+00])
```

Adjusting for the perturbation, 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:

```python
# Import the necessary packages
import numpy
from openopt import QP

# Define QP parameters
# myH = numpy.diag([1,0])  # We will use the pertubed version below
myH = numpy.diag([1,0]) + (1e-3)*numpy.eye(2)
myf = numpy.array([3,4])
myl = numpy.zeros(2)
myA = numpy.array([[[-1,-3],[2,5],[3,4]]])
myb = numpy.array([-15,100,80])
# myu = numpy.inf * numpy.ones(2)  # We won’t use this, just for reference

# Construct the QP
p = QP(H=myH, f=myf, lb=myl, A=myA, b=myb)

# Invoke qlcp solver
sol = p.solve('qlcp')

# Extract optimal value and solution
opt_val = p.ff  # opt_val == array([20.0125])
opt_sol = p.xf  # opt_sol == array([-7.2359e-13, 5.0000e+00])
```