CS 6491 Project

Name: Zhengxiang ZHOU
StudentID: 55817209
Topic: 4) Small is Beautiful: Tiniest Optimization Solver

Introduction

Generally, the third-party libraries that can handle optimization problems contain tons of codes. Some of these codes are responsible for the functionalities related to the optimization process, but a large portion of those codes are to meet the requirements for the usability of the whole libraries. As a consequence, the dependencies in those codes are complex and it is a difficult task to find the codes dealing with a specific problem.

However, sometimes we might want to solve a small problem and the storage might be limited, like the IoT devices which are light. Their poor performances disallow them to run large programs. In this case, we need a small program which are short and designed for a particular problem. And in this report, I will design and build a small Python solver for the particular quadratic programming problems.

Problem Statement

Math:
The solver is built for equality quadratic programming, for problems like:

$$\text{EQP:}\min_x \frac{1}{2}\mathrm{x}^T\mathrm{Q}\mathrm{x} + \mathrm{c}^T\mathrm{x}$$
$$\text{s.t} \quad \mathrm{A}\mathrm{x} = \mathrm{b}$$
$$\mathrm{x} \geq 0$$

Setup
The language I choose is Python, which is convenient if the libraries can be imported. However, in fact we should build the pure solver without any dependency, since usually the imported codes mean a large number of trivial codes that are irrelevant to the original problem.

Challenges:
1. Since Python is not a language designed for scientific research, the pure language doesn't support matrix operations like multiplications, additions, and etc., not to mention to solve a linear system or get the inverse of a matrix. Thus the first challenge is to write the needed matrix operations.
2. Besides, the length of the codes for different operations differs based on the complexities of them. Therefore, to write a small program, the second challenge is that I have to find a simple method that use as few matrix operations as possible.

3. There might be more than one way to write a segment of code to achieve something in Python. For example, to add each element in a list by one, we can write like:

```
for i in range(len(A)):
    A[i]+=1
```

or in a shorter way:

```
A=list(map(lambda x:x+1,A))
```

The third challenge is that the codes should be converted to the forms that take up less space.

Design

The method I choose is for the equality quadratic programming, which is called *Range-space methods*. For the given problem, we can write the KKT matrix as:

$$\begin{bmatrix} Q & -A^T \\ A & 0 \end{bmatrix} \begin{bmatrix} x^* \\ \lambda^* \end{bmatrix} = \begin{bmatrix} -c \\ b \end{bmatrix}$$

Take a guess x_0, we can express x^* as $x^* = x_0 + p$, where p is the step. Then the KKT matrix becomes:

$$\begin{bmatrix} Q & -A^T \\ A & 0 \end{bmatrix} \begin{bmatrix} -p \\ \lambda^* \end{bmatrix} = \begin{bmatrix} c + Qx_0 \\ Ax_0 - b \end{bmatrix}$$

Now applies the range-space methods if Q is positive definite and easy to invert(diagonal or small matrix). We solve for λ^* by the second equation:

$$(AQ^{-1}A^T)\lambda^* = (AQ^{-1}c + b)$$

And then recover p by:

$$Qp = A^T\lambda^* - c - Qx_0$$

Thus get the optimal value $x^* = x_0 + p$.

Codes transformation:

Stage 1:

The codes are written with its correct functionalities, seen in Appendix 1. In this stage, the codes are still readable and easy to redevelop. In total, I need to write codes to deal with the matrix additions, multiplications, to inverse, and transpose a matrix, to get the determinant of a matrix, to solve a linear system, and to solve the QP problem by range-space methods. At this stage the number of characters is **2045**.

Stage 2:

In this stage, the meaningful names are removed from the codes. For example, the function names like “add” are transformed to a single character “a”. The redundant codes like

```
L=mat_mult(A,getMatrixInverse(Q))
L=mat_mult(L,transposeMatrix(A))
```

are transformed to one sentence.

```
L=m(m(A,I(Q)),T(A))
```

Besides, assignments for several independent variables are changed to one sentence like from “a=1;b=2” to “a,b=1,2”. Functions are called in nested way instead of write assignment sentences several times, and unnecessary temporary variables are removed. Several sentences are just in one line to avoid the indent of four spaces. After these transformations, the codes are shown in Appendix 2. The characters count is reduced to **1332.**

Stage 3:

More technical changes like using a variable to store the iterations of range(n) to avoid reusing the long keyword “range”.

Remove the indents for some sentences after a colon. Use “lambda” instead of “def” for small function definitions, like:

```
def T(m):return list(map(list,zip(*m)))
--->
T=lambda m:list(map(list,zip(*m)))
```

Replace the sum of list with the function “sum”, and combines the previous optimizations together. Change “==2” to “<3”, and rename functions “range” and “len”.

After all transformations, the characters count is **1021**, and the codes take up 1,039 bytes of the storage.

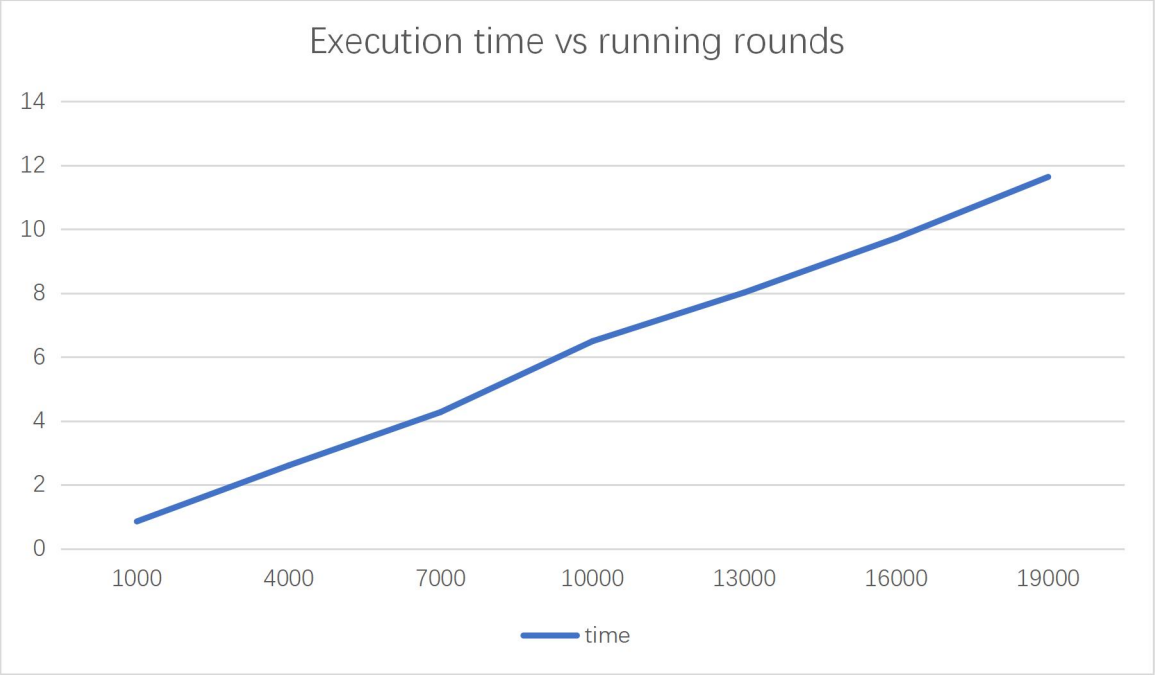
```
R,L=lambda X:range(X),lambda Y:len(Y);a,T,M=lambda A,B,C=1:[[A[i][j]+B[i][j]*C for j
 in R(L(A[0]))]for i in R(L(A))],lambda m:list(map(list,zip(*m))),lambda v,I,J:[r[:J]
+r[J+1:]for r in (v[:I]+v[I+1:])]
def m(a,b):n,k,m=L(a),L(b),L(b[0]);c=[[0 for i in R(m)]for j in R(n)];return[[sum([a
[i][l]*b[l][j]for l in R(k)])for j in R(m)]for i in R(n)]
D=lambda m:m[0][0]*m[1][1]-m[0][1]*m[1][0]if L(m)<3 else sum([((-1)**c)*m[0][c]*D(M(
m,0,c))for c in R(L(m))])
def I(m):
    d,_=D(m),R(L(m))
    if L(m)<3:return[[m[1][1]/d,-m[0][1]/d],[-m[1][0]/d,m[0][0]/d]]
    h=T([[(-1)**(r+c)*D(M(m,r,c))for c in _]for r in _])
    return[[h[r][c]/d for c in _]for r in _]
def s(A,B):
    _=list(R(L(A)))
    for f in _:
        t=1/A[f][f];B[f][0]*=t
        for j in _:A[f][j]*=t
        for i in _[0:f]+_[f+1:]:
            u=A[i][f];B[i][0]-=u*B[f][0]
            for j in _:A[i][j]-=u*A[f][j]
    return B
def qp(Q,A,b,c):x=[[0]for _ in R(L(Q))];return a(x,s(Q,a(m(T(A),s(m(m(A,I(Q)),T(A)),
a(m(m(A,I(Q)),c),b))),a(c,m(Q,x)),C=-1)))
```

Benchmark

For better evaluation of the stability and efficiency of the program, I conduct two experiments with the cvx quadratic programming.

E1.

The first is to evaluate whether the programs can maintain stable after thousands runs of the same code segments. The problem is 4-dimension quadratic programming and the they will be run from 1000 to 19000 times. Each slot of the data is taken the average for 5 experiments with the same number of rounds.

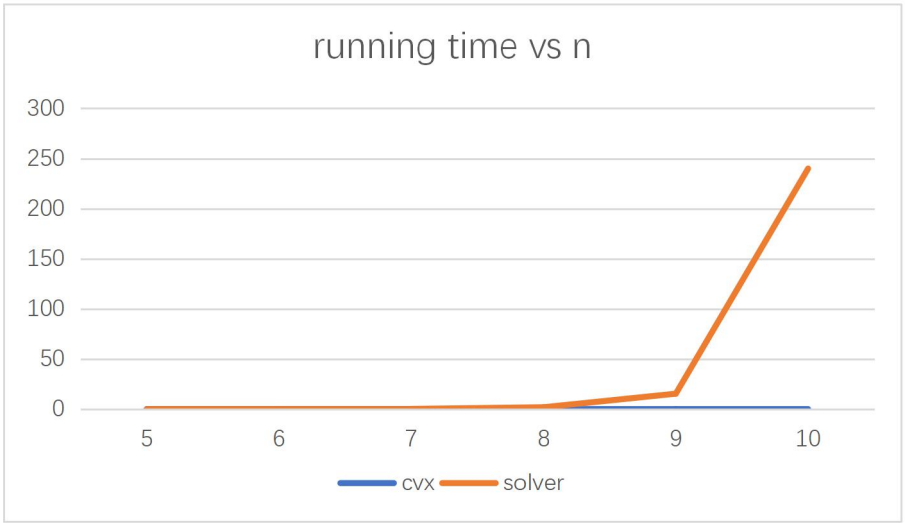


From the figure above, the curve is almost a line, which indicates that the codes are stable and the execution will not have extra burden on the computer.

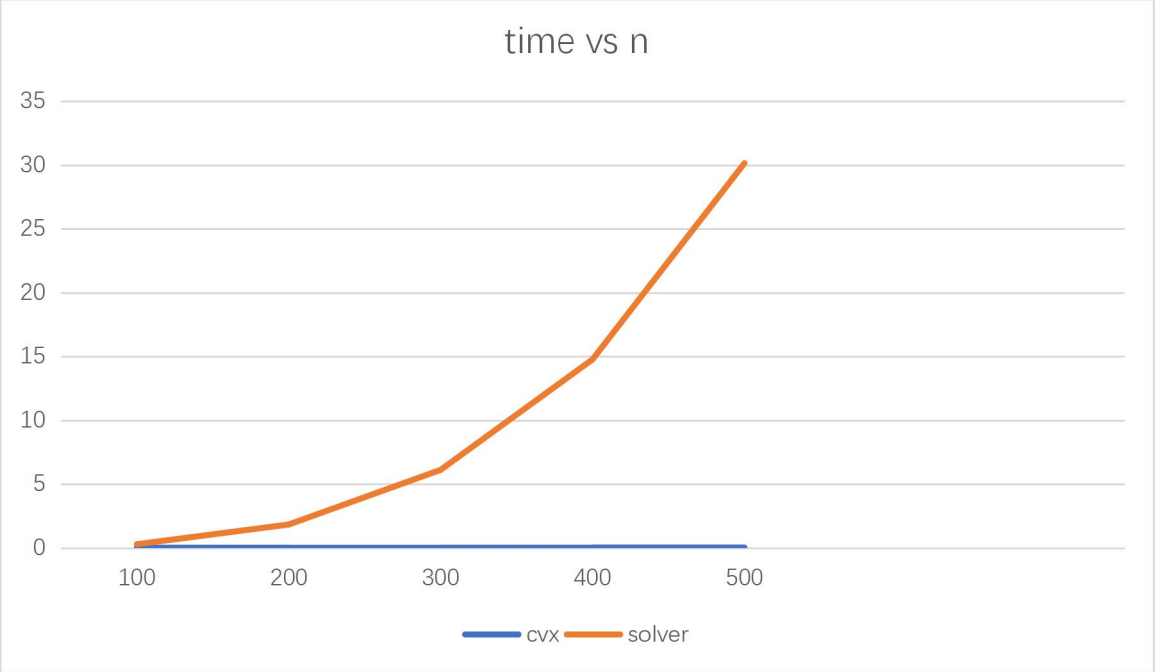
E2.

The second is to evaluate whether it can scale to large problems, like a thousand variables. This code will also be compared to the popular optimization library "cvx".

For common quadratic programming, where the Q can be any positive-definite matrix. The small solver cannot work when the number of variables is more than 9. It cannot finish in 4 mins when n=10, while cvx almost doesn't change the running time in this small scale.



This can be due to the time-consuming inverse operations. For a particular task where the Q matrix is diagonal, the solver can be further optimized. In this case the characters count can be reduced to **905**. (Appendix 3) The small solver can scale better for this weaker problem than that with the general Q matrix. With a fixed number of constraints which is 10, the figure shows:



Compared with the performance for a general Q matrix, the scale of the smaller solver grows from 10 to 500 variables, which can be somehow thought to be practical.

Discussion

To get a small python QP solver, it is not possible to choose a fast method. For most of the practical methods, there is a need of complex matrix operations like finding the null-space basis for matrix A, or to take derivatives of the inputs. Unlike Matlab, which supports the matrix operations on its own, these operations cannot be written using Python in just a few lines of codes. Therefore, the Python small solver must have some tradeoffs between the performance and the code length.

In fact, for the problem where Q is diagonal, there does exist more optimizations like the quadratic form $x^TQ^{-q}x$ can be calculated by $x[i][i]^2*Q[i][i]$, which reduce the complexity from $O(n^3)$ to $O(n)$. However, since the Gaussian elimination is $O(n^3)$, it can be trivial.

Conclusion

For a small Python quadratic programming solver, there must be a tradeoff between the code length and performance. The fast methods to solve quadratic programming problems all demand complex matrix operations, and the developer needs to write these operations himself in pure Python, which doesn't meet the requirement of a small solver. With some weakening of the original problem, I have built the program with 1021 characters that can solve all equality-constrained quadratic programming problems under 10

variables, and the program with 905 characters that can solve equality-constrained quadratic programming problems with a diagonal matrix Q under 500 variables. Nevertheless, with other assumptions about the problem, there can be other types of small Python solvers for QP. The future of small Python QP solver can be different solvers for problems under their corresponding assumptions respectively.

Appendix

1

```
def add(a,b,C=1):
    for i in range(len(a)):
        for j in range(len(a[0])):
            a[i][j]+=b[i][j]*C
    return a
def mat_mult(a,b):
    n=len(a);k=len(b);m=len(b[0])
    c=[[0 for i in range(m)]for j in range(n)]
    for i in range(n):
        for j in range(m):
            for l in range(k):
                c[i][j]+=a[i][l]*b[l][j]
    return c
def transposeMatrix(m):
    return list(map(list,zip(*m)))
def getMatrixMinor(m,i,j):
    return [row[:j]+row[j+1:] for row in (m[:i]+m[i+1:])]
def det(m):
    if len(m)==2:
        return m[0][0]*m[1][1]-m[0][1]*m[1][0]
    determinant=0
    for c in range(len(m)):
        determinant+=((-1)**c)*m[0][c]*det(getMatrixMinor(m,0,c))
    return determinant
def getMatrixInverse(m):
    d=det(m)
    if len(m)==2:
        return [[m[1][1]/d,-1*m[0][1]/d],[-1*m[1][0]/d,m[0][0]/d]]
    cofactors=[]
    for r in range(len(m)):
        cofactorRow=[]
        for c in range(len(m)):
            minor=getMatrixMinor(m,r,c)
```

```
            cofactorRow.append(((-1)**(r+c)) * det(minor))
        cofactors.append(cofactorRow)
    cofactors=transposeMatrix(cofactors)
    for r in range(len(cofactors)):
        for c in range(len(cofactors)):
            cofactors[r][c]=cofactors[r][c]/d
    return cofactors
def solve(AM ,BM):
    n=len(AM)
    indices=list(range(n))
    for fd in range(n):
        fdScaler=1.0 / AM[fd][fd]
        for j in range(n):
            AM[fd][j]*=fdScaler
        BM[fd][0]*=fdScaler
        for i in indices[0:fd]+indices[fd+1:]:
            crScaler=AM[i][fd]
            for j in range(n):
                AM[i][j]=AM[i][j]-crScaler*AM[fd][j]
            BM[i][0]=BM[i][0]-crScaler*BM[fd][0]
    return BM
def qp(Q,A,b,c,x=[[0],[0]]):
    L=mat_mult(A,getMatrixInverse(Q))
    L=mat_mult(L,transposeMatrix(A))
    R= mat_mult(A,getMatrixInverse(Q))
    R=mat_mult(R,c)
    R=add(R,b)
    lad=solve(L,R)
    nL=Q
    nR=mat_mult(transposeMatrix(A),lad)
    nR=add(nR,c,C=-1)
    nR=add(nR,mat_mult(Q,x),C=-1)
    p=solve(nL,nR)
    x=add(x,p)
    return x
```

2.

```
def a(A,B,C=1):
    return [[A[i][j]+B[i][j]*C for j in range(len(A[0]))]for i in range(len(A))]
def m(a,b):
    n,k,m=len(a),len(b),len(b[0])
    c=[[0 for i in range(m)]for j in range(n)]
    for i in range(n):
```

```
        for j in range(m):
            for l in range(k):
                c[i][j]+=a[i][l]*b[l][j]
    return c
def T(m):
    return list(map(list,zip(*m)))
def M(m,i,j):
    return [r[:j]+r[j+1:]for r in (m[:i]+m[i+1:])]
def D(m):
    if len(m)==2:
        return m[0][0]*m[1][1]-m[0][1]*m[1][0]
    g=0
    for c in range(len(m)):
        g+=((-1)**c)*m[0][c]*D(M(m,0,c))
    return g
def I(m):
    d,h,_=D(m),[],list(range(len(m)))
    if len(m)==2:
        return [[m[1][1]/d,-1*m[0][1]/d],[-1*m[1][0]/d,m[0][0]/d]]
    for r in _:
        o=[((-1)**(r+c))*D(M(m,r,c))for c in _]
        h.append(o)
    h=T(h)
    return [[h[r][c]/d for c in range(len(h))]for r in range(len(h))]
def s(A,B):
    n=len(A);_=list(range(n))
    for f in _:
        t=1./A[f][f];B[f][0]*=t
        for j in _:
            A[f][j]*=t
        for i in _[0:f]+_[f+1:]:
            u=A[i][f];B[i][0]=B[i][0]-u*B[f][0]
            for j in _:
                A[i][j]=A[i][j]-u*A[f][j]
    return B
def qp(Q,A,b,c):
    x=[[0]for _ in range(len(Q))];return a(x,s(Q,a(m(T(A),s(m(m(A,I(Q)),T(A)),a(m(m(
A,I(Q)),c),b))),a(c,m(Q,x)),C=-1)))
```

3

```
R,L=lambda X:range(X),lambda Y:len(Y);a,T,M=lambda A,B,C=1:[[A[i][j]+B[i][j]*C for j
 in R(L(A[0]))]for i in R(L(A))],lambda m:list(map(list,zip(*m))),lambda v,I,J:[r[:J]
+r[J+1:]for r in (v[:I]+v[I+1:])]
```

```
def m(a,b):n,k,m=L(a),L(b),L(b[0]);c=[[0 for i in R(m)]for j in R(n)];return[[sum([a
[i][l]*b[l][j]for l in R(k)])for j in R(m)]for i in R(n)]
D=lambda m:m[0][0]*m[1][1]-m[0][1]*m[1][0]if L(m)<3 else sum([((-1)**c)*m[0][c]*D(M(
m,0,c))for c in R(L(m))])
def I(m):return[[1/m[i][j] if m[i][j] else 0 for j in R(L(m))]for i in R(L(m))]
def s(A,B):
    _=list(R(L(A)))
    for f in _:
        t=1/A[f][f];B[f][0]*=t
        for j in _:A[f][j]*=t
        for i in _[0:f]+_[f+1:]:
            u=A[i][f];B[i][0]-=u*B[f][0]
            for j in _:A[i][j]-=u*A[f][j]
    return B
def qp(Q,A,b,c):
    x=[[0]for _ in R(L(Q))];return a(x,s(Q,a(m(T(A),s(m(m(A,I(Q)),T(A)),a(m(m(A,I(Q))
,c),b))),a(c,m(Q,x)),C=-1)))
```