

Python から lpsolve を使う方法

小林和博

2010 年 8 月 4 日

1 Python で Lp_solve を使う

Python はアルゴリズムを試験的に動かしてみるのによいので、MIP をサブルーチンにつかったアルゴリズムを動かすために Python から LP/MIP ソルバを使いたい。そのために lp_solve を Python から動かしてみる。Python でうまくいくことを確認できたら、そのアルゴリズムは C で書くのがよい。

2 基本的な使い方

変数、制約式を指定できれば LP/MIP は定義できる。そのためには、次のような命令を実行すればよい。

```
>>> lpsolve('add_constraint', lp, [0, 78.26, 0, 2.9], GE, 92.3)
```

GE は、左辺 \geq 右辺 LE は、左辺 \leq 右辺

```
>>> lpsolve('set_lowbo',lp,1,28.6) は、最初の変数 x1 について、 $x1 \geq 28.6$ 
```

```
>>>lpsolve('set_upbo',lp,4,48.98) は、4 番目の変数 x4 について  $x4 \leq 48.8$ 
```

```
>>> ret = lpsolve('set_row_name', lp, 3, 'LASTROW')
```

```
>>> ret = lpsolve('set_col_name', lp, 3, 'COLTHREE')
```

は行または列に名前をつける。

```
print lpsolve('get_mat', lp, 1, 2)
```

は、制約行列の 1 行 2 列の値を表示。

```
>>> lpsolve('solve', lp)
```

```
0L
```

```
>>> print lpsolve('get_objective', lp)
```

```
31.7827586207
```

```
>>> print lpsolve('get_variables', lp)[0]
```

```
[28.600000000000001, 0.0, 0.0, 31.827586206896552]
```

```
>>> print lpsolve('get_constraints', lp)[0]
```

```
[92.299999999999997, 6.863999999999999, 391.2928275862069]
```

は最適値の表示, 変数の表示, 制約の表示

3 行列

```
>>lpsolve('add_constraint',lp,[0.24,0,11.31,0],1,14.8);
```

a1 がリスト型の変数である場合

```
>> lpsolve('add_constraint',lp,a1,1,14.8)
```

と指定できる. 2次元の行列がつかわれるときは

```
>>lpsolve('add_constraint',lp,[[1,2,3],[4,5,6]])
```

でも可. ただし, [1,2,3] は最初の行で, [4,5,6] は2番目の行.

整数変数であることを指定するには, 変数ごとに指定する他に, ベクトルでまとめて指定する方法がある.

```
>>> lpsolve('set_int',lp,[must_be_int])..
```

とすれば, ベクトル [must_be_int] に含まれる添字の変数が整数変数となる.

無限大を指定するには, たとえば

```
ret = lpsolve('set_upbo', lp, [Infinite, Infinite, Infinite, 48.98])
```

-Infinite で, マイナス無限大を表現

Python は, リスト上での行列演算の機能は提供していない. 適当なパッケージで, リスト型の変数は行列型に変更されないといけない. たとえば NumPy を使うと, 行列やベクトルの演算が可能.

```
>>> lp = lpsolve('make_lp', 0, 4)
```

```
>>> lpsolve('set_verbosity', lp, IMPORTANT)
```

```
>>> ret = lpsolve('set_obj_fn', lp, [1, 3, 6.24, 0.1])
```

```
>>> ret = lpsolve('add_constraint', lp, [0, 78.26, 0, 2.9], GE, 92.3)
```

```
>>> ret = lpsolve('add_constraint', lp, [0.24, 0, 11.31, 0], LE, 14.8)
```

```
>>> ret = lpsolve('add_constraint', lp, [12.68, 0, 0.08, 0.9], GE, 4)
```

```
>>> ret = lpsolve('set_lowbo', lp, [28.6, 0, 0, 18])
```

```
>>> ret = lpsolve('set_upbo', lp, [Infinite, Infinite, Infinite, 48.98])
```

```
>>> ret = lpsolve('set_col_name', lp, ['COLONE', 'COLTWO', 'COLTHREE', 'COLFOUR'])
```

```
>>> ret = lpsolve('set_row_name', lp, ['THISROW', 'THATROW', 'LASTROW'])
```

```
>>> ret = lpsolve('write_lp', lp, 'a.lp')
```

```
>>> print lpsolve('get_mat', lp)[0]
```

```
[[0.0, 78.26, 0.0, 2.9], [0.24, 0.0, 11.31, 0.0], [12.68, 0.0, 0.08, 0.9]]
```

```
>>> A = array(lpsolve('get_mat', lp)[0])
```

```
>>> A
```

```
array([[ 0. , 78.26,  0. ,  2.9 ],
```

```

    [ 0.24,  0. , 11.31,  0. ],
    [ 12.68,  0. ,  0.08,  0.9 ]])
>>> X = array(lpsolve('get_variables', lp)[0])
>>> X
array([ 28.6      ,  0.      ,  0.      , 31.82758621])

```

numpy だと、行列 A と行列 X のかけ算では、以下で実行可能

```

>>> A
array([[ 0. , 78.26,  0. ,  2.9 ],
       [ 0.24,  0. , 11.31,  0. ],
       [ 12.68,  0. ,  0.08,  0.9 ]])
>>> X
array([ 28.6      ,  0.      ,  0.      , 31.82758621])

```

```

>>> matrix(A)*matrix(X).transpose()
matrix([[ 92.3      ],
        [  6.864    ],
        [391.29282759]])
>>>

```

とすれば、行列のかけ算になる

lp_solve は、Constant は大文字小文字の区別をしない。小文字で入力しても常に大文字に変換される

```

>>> ret = lpsolve('set_constr_type', lp, ['LE', 'EQ', 'GE']);

```

という風に、リストで制約の型を指定することも可

4 混合整数 (線形) 計画問題 (MILP) の解き方

```

[obj,x,duals]=lp_solve(f,a,b,e,vlb,vub,xint,scalemode,keep)

```

によって、MILP を解くことができる。lp_solve のマニュアルでは、以下の標準形が用いられる。

```

max v = f'*x
a*x <> b
vlb <= x <= vub
x(int) are integer

```

入力:

f: n 次元ベクトル, 目的関数の係数

a: $m \times n$ の行列,

b: m 次元ベクトル. $Ax=b$ の右辺

e: m 不等式の意味を定める m 次元ベクトル.

- $e(i) = -1 \rightarrow$ Less Than,
- $e(i) = 0 \rightarrow$ Equals
- $e(i) = 1 \rightarrow$ Greater Than

vlb:下限を表す n 次元ベクトル. 省略するか空の場合は, 0

vub:上限を表す n 次元ベクトル

xint: 整数変数のベクトル. 空集合, 省略可

scalemode:省略化. スケールのモードを定める

keep: 解いたあとに lp を保存しておくかどうかのフラグ. 省略したら, Lp は解いたあとに消される (deleted)

出力:

obj: 最適値

x: 最適解

duals: 双対問題の最適解

使用例

max: $-x_1 + 2 x_2$;

C1: $2x_1 + x_2 < 5$;

$-4 x_1 + 4x_2 < 5$;

int x_2, x_1 ;

を解くには, 以下のコードで OK

```
>>> from lp_solve import *
>>> [obj, x, duals] = lp_solve([-1, 2], [[2, 1], [-4, 4]], [5, 5], [-1, -1], None, None, [1, 2])
>>> print obj
3.0
>>> print x
[1.0, 2.0]
```

定数をつかってかくと,

```
>>> f=[-1,2]
>>> a=[[2,1],[-4,4]]
>>> b=[5,5]
>>> e=[-1,-1]
>>> xint=[1,2]
>>> [obj,x,duals]=lp_solve(f,a,b,e,None,None,xint)
```

```
>>> obj
```

問題を作るだけの `lp_maker()` も存在する。Handle を返す。

```
>>> from lp_maker import *
>>> lp = lp_maker([-1, 2], [[2, 1], [-4, 4]], [5, 5], [-1, -1], None, None, [1, 2])
>>> lp
```

問題を解いて答えを得るには、

```
>>> lpsolve('solve', lp)
0L
>>> lpsolve('get_objective', lp)
3.0
>>> lpsolve('get_variables', lp)[0]
[1.0, 2.0]
```

つかい終わったあとは、必ず delete する

```
>>> lpsolve('delete_lp', lp)
```

4.1 多品種フロー問題

多品種フロー問題を書いてみる。

$$\min \sum_{(i,j) \in A} \sum_{k \in K} c_{ij}^k x_{ij}^k \quad (1)$$

$$\text{subject to } \sum_{i \in N_n^+} x_{in}^k - \sum_{j \in N_n^-} x_{nj}^k = d_n^k \quad \forall n \in N, k \in K \quad (2)$$

$$\sum_{k \in K} x_{ij}^k \leq C_{ij} \quad (3)$$

$$0 \leq x_{ij}^k \leq d^k \quad \forall (i,j) \in A, k \in K \quad (4)$$

問題を定義するのに必要なデータは、 c_{ij}^k , d_n^k , C_{ij} , d^k である。いま、 $K = \{1, 2, 3\}$ とする。 $A = N \times N$ で、 $N = \{1, 2\}$ とする。完全グラフとすると、 C は $|N| \times |N|$ の行列、 c_{ij}^k は $|N| \times |N|$ の行列 $|K|$ 個にする。 d_n^k は、 $|N| \times |K|$ の行列にする。

乱数からデータを作成する場合は、

```
>>> A=numpy.random.randn(5,5)
>>> b=numpy.array(range(1,6))
```

ただし、こうして生成した行列 A の要素は負になりうるので、 $A+=10$ とすれば、 A の全ての要素に 10 が足されて正になる。ちなみに、

```
>>> from numpy import linalg
>>> linalg.solve(A,b)
```

とすると、線形方程式 $Ax=b$ の解 x が得られる。

たとえば、

```
>>> C=numpy.random.randn(len(N),len(N))+10
```

とすれば、 $|N| \times |N|$ の行列が得られる。対称にしたければ、 $C=C+C.transpose()$ を追加。C は、整数のほうがいいとすれば、

```
>>> C=numpy.ones( (len(N),len(N)),dtype=int)
```

```
>>> b=numpy.random.random( (len(N),len(N)) )
```

```
>>> b*=100 //100 は要素の適当な定数
```

```
>>> C+=b
```

```
>>> C=C+C.transpose()
```

とすれば、C の要素は整数になる。 $|K|$ 個の $|N| \times |N|$ 行列を覚えるひとつの方法は、

```
>>> c=range(3)
```

```
>>> for k in K:
```

```
    c[k] = numpy.random.rand(len(N),len(N))
```

あるいは、整数のほうがよければ、

```
>>> c=range(3)
```

```
>>> for k in K:
```

```
... c[k] = numpy.ones( (len(N),len(N)),dtype=int )
```

```
... b=numpy.random.random( (len(N),len(N)) )
```

```
... b*=5
```

```
... c[k]+=b
```

```
... c[k]=c[k]+c[k].transpose()
```

とすればよい。これで、 $k=1$ にたいする 2×2 の行列 $c^k = (c_{ij}^k)_{(i,j) \in A}$ は、 $c[k]$ でアクセスできる。

d_n^k をあらわす行列は、 n を最初のインデックス (行)、 k を 2 番目のインデックス (列) として、次のように生成する

```
>>> d=numpy.zeros( len(N)*len(K)).reshape(len(N),len(K))
```

これを、行列らしく扱うには、

```
>>> mat_d = numpy.matrix(d)
```

とする。こうすると、 $|N| \times |K|$ の行列らしく扱える。たとえば、(2, 1) 成分の表示は、

```
>>> mat_d[2,1]
```

```
0.0
```

でできるし、(1, 2) 成分に値を代入するには

```
>>> mat_d[1,2]
0.0
>>> mat_d[1,2]=10
>>> mat_d
matrix([[ 0.,  0.,  0.],
        [ 0.,  0., 10.]])
>>>
```

とすればよい. 1行目,1列目を表示するにはそれぞれ

```
>>> mat_d[0,:]
>>> mat_d[:,0]
```

とすればよい. MATLAB とおなじだ. d_n^k は $|N| \times |K|$ の行列として作ることにする. つまり, 行が n に, 列が k に対応する. いま, $K = \{1, 2, 3\}$, $N = \{1, 2, 3, 4\}$ とする. 各品種の始点終点を適当にきめて, d_n^k を以下のように設定する. 先にのべた, d の生成手順より, mat_d は, $|N| \times |K|$ の行列となる.

```
>>>mat_d[0,0]=-20 //品種 1 の始点は n=1
>>>mat_d[2,0]=20 //品種 1 の終点 n=3
>>>mat_d[1,1]=-10 //品種 2 の始点は n=2,
>>>mat_d[3,1]=10 //品種 2 の終点は n=4
>>>mat_d[0,2]=-15 //品種 3 の始点は n=1
>>>mat_d[2,2]=15 //品種 3 の終点は n=3
```

最初から行列を全部生成してから lp をつくるのではなくて, 段階的に制約と列を追加したい場合もある. このようなとき, ひとまず LP をつくるための手順は以下のとおり

```
>>> m=2; //行 (制約式の数)
>>> n=0; //列 (変数の数)
>>> lp=lpsolve('make_lp',2,0);
```

とすると, 空の制約式 2 本で, 変数の数が 0 の LP が定義される.

```
>>> coef=[1,2,3]
>>> lpsolve('add_column',lp,coef)
```

を実行すると, 変数が 1 つ追加される. ここで, 3 番目の引数として与えるベクトル coef は, $\text{coef}[0]$ が, 新しく加える変数の目的関数での係数, $\text{coef}[1]$ が 1 番目の制約式での新しく加える変数の係数, $\text{coef}[2]$ が 2 番目の制約式での新しく加える変数の係数をあらわす. この引数の次元が, $[\text{制約式の数} + 1]$ でなければ, エラーになる.

変数の数と制約式の数を意識しておいて, 変数を加えるときは $[\text{現在の制約式の数} + 1(\text{目的関数})]$ の次元のベクトルを指定し, 制約式を加えるときは変数の数の次元のベクトルを指定すれば, 随時制約式でも変数でも追加することができる. たとえば,

```
>>> lp=lpsolve('make_lp',2,0)
```

```

>>> lpsolve('write_lp',lp,'a.lp')
>>> lpsolve('add_column',lp,coef)
>>> lpsolve('add_column',lp,coef2)
>>> lpsolve('add_constraint',lp,[1,2],GE,5)
>>> lpsolve('write_lp',lp,'a.lp')

```

を実行するとファイル a.lp には、以下のように生成された問題が出力される

```

/* Objective function */
min: +C1 +0.1 C2;

/* Constraints */
+2 C1 +0.2 C2 <= 0;
+3 C1 +0.3 C2 <= 0;
+C1 +2 C2 >= 5;

```

これらの手続をつかって、多品種フロー問題を定義する。まず、変数 x_{ij}^k を生成する。

```

>>>tc={}
>>>lp=lpsolve('make_lp',1,0)
>>>coef=[0,0] // [目的関数での係数, 第一制約式での係数]
>>>for k in K:
...     for i in N:
...         for j in N:
...             lpsolve('add_column',lp,[numpy.matrix(c[k])[i,j],0])
...             // 目的関数での係数は c_{ij}^k, 第一制約式での係数は 0
...             tc.update((k,i,j):icnt)}
>>>icnt=1
>>>for k in K:
...     for i in N:
...         for j in N:
...             lpsolve('set_col_name',lp,icnt,'COL'+str(k)+'_'+str(i)+'_'+str(j))
...             icnt=icnt+1

```

これで、変数 x_{ij}^k が生成された。後で、 x_{ij}^k が何番目に追加された変数かを調べたいので、Python の辞書型変数 tc で対応を記録しておく。これにより、tc[(k,i,j)] で、変数 x_{ij}^k が何番目の変数かがわかる。途中、add_column の実行時に第三引数のベクトルを指定するところで、 $k \in K$ に対する (i,j) 成分は、以下のループで順にアクセスすることができることを利用した。

```

>>> for k in K:
...     for i in N:
...         for j in N:
...             numpy.matrix(c[k])[i,j]

```

この時点では、目的関数が定義されているのみである。ダミーの制約式が1つ含まれているが、これは `make_lp` の実行のために仮に入れたものである (制約式を入れずに `make_lp` を実行する方法が見つければこれは必要ない。)

つぎに、制約

$$\sum_{k \in K} x_{ij}^k \leq C_{ij}$$

を定義する。たとえば、 $(i, j) = (0, 1)$ についての上の制約式は、

$$x_{01}^1 + x_{01}^2 + x_{01}^3 \leq C_{01}$$

になる。この制約は、`add_constraints` 命令により追加するのだが、この命令の引数として、新たに追加する制約式における各変数の係数をあらわすベクトルが必要である。このベクトルの次元は、[変数の数]に等しくなければならない。係数といっても制約式の \sum にあらわれる x_{ij}^k に係数1をつければよい。これは、下の手順で実行できる。

```
>>> tcnt=lpsolve('get_Ncolumns',lp)
>>> for i in N:
...     for j in N:
...         atmp=numpy.zeros(tcnt)
...         for k in K:
...             atmp[tc[(k,i,j)]-1]=1
...         lpsolve('add_constraint',lp,list(atmp),LE,C[i,j])
```

つぎに、制約式

$$\sum_{i \in N_n^+} x_{in}^k - \sum_{j \in N_n^-} x_{nj}^k = d_n^k \quad \forall n \in N, k \in K \quad (5)$$

を追加する。これも同様に、以下の手順で追加できる。

```
>>> for k in K:
...     for n in N:
...         atmp=numpy.zeros(tcnt)
...         for j in N:
...             if n!=j:
...                 atmp[tc[(k,n,j)]-1]=-1
...                 atmp[tc[(k,j,n)]-1]=1
...         lpsolve('add_constraint',lp,list(atmp),EQ,mat_d[n,k])
```

品種 $k \in K$ の荷物の量は、 d_n^k から設定できる。具体的には、 $\max_{n \in N} d_n^k = d_k$ 。 d_n^k を、

```
>>>d=numpy.zeros(len(N)*len(K).reshape(len(N),len(K))
>>>mat_d=numpy.matrix(d)
>>>mat_d[0,0]=-10
>>>mat_d[1,0]=10
>>>mat_d[0,0]=20
```

```
>>>mat_d[1,0]==-20
>>>mat_d[0,0]==-15
>>>mat_d[1,0]=15
```

と生成していたとする。このとき、 d_k を Python の変数としては q で表すとすると、以下の手順で求まる

```
>>>q=numpy.zeros(len(K))
>>>t_max=-1
>>>for k in K:
...     t_max=-1
...     for i in N:
...         if mat_d[i,k]>t_max:
...             t_max=mat_d[i,k]
...     q[k]=t_max
```

これで、 $q[k]$ は d_k を表している。これを用いれば、変数の範囲の制約は、以下で定義することができる

```
>>> for k in K:
...     for i in N:
...         for j in N:
...             lpsolve('set_bounds',lp,tc[(k,i,j)],0,q[k])
```

ここまでで問題が生成できた。これを解くには

```
>>> lpsolve('solve',lp)
```

を実行すればよい。