模型-运筹学-规划论-线性规划【czy】

1. 模型名称

线性规划（Linear programming，简称LP）

2. 适用范围

线性约束下**线性**目标函数的**极值**问题的数学理论和方法

3. 形式

3.1一般形式

1. **决策变量**

   每一个问题都用一组决策变量$(x_1, x_2, ..., x_n)$表示某一方案,这组决策变量的值就代表一个具体方案。一般这些变量取值是非负且连续的。

2. **目标函数**

   一个要求达到的目标,它可用决策变量的线性函数(称为目标函数)来表示。按问题的不同,要求目标函数实现最大化或最小化。

3. **约束条件**

   存在一定的约束条件,这些约束条件可以用一组线性等式或线性不等式来表示。

满足以上三个条件的数学模型称为**线性规划的数学模型。**

其**一般形式**为:

$$\text{目标函数} max(min)z = c_1x_1 + c_2x_2 + ... + c_nx_n$$

$$\text{满足约束条件} \begin{cases} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n \le (=, \ge) b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n \le (=, \ge) b_2 \\ \cdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n \le (=, \ge) b_m \\ x_1, x_2, ..., x_n \ge 0 & \text{变量的非负约束条件} \end{cases}$$

3.1 标准形式

(M_1)

$$\max z = c_1x_1 + c_2x_2 + \cdots + c_nx_n$$
$$\text{s.t.} \begin{cases} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2 \\ \cdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n = b_m \\ x_1, x_2, ..., x_n \ge 0 \end{cases}$$

$(M_1^{'})$

$$\max z = \sum_{j=1}^{n} c_jx_j$$
$$\text{s.t.} \begin{cases} \sum_{j=1}^{n} a_{ij}x_j = b_i \\ x_j \ge 0, j = 1, 2, ..., n \end{cases}$$

标准形式中规定$b_i > 0; i \in (1, 2, \cdots, m)$。否则等式两端乘以"-1"。

3.2 矩阵形式

$(M_1^{''})$

$$\max z = CX$$
$$\text{s.t.} \begin{cases} \sum_{j=1}^{n} P_j x_j = b \\ x_j \geq 0, j = 1, 2, \ldots, n \end{cases}$$

其中：$C = (c_1, c_2, \ldots, c_n)$，$C$———价值向量

$$X = \begin{bmatrix} x_1 \\ x_2 \\ \cdots \\ x_n \end{bmatrix} P_j = \begin{bmatrix} a_{1j} \\ a_{2j} \\ \cdots \\ a_{mj} \end{bmatrix} b = \begin{bmatrix} b_1 \\ b_2 \\ \cdots \\ b_m \end{bmatrix}$$

X———决策变量向量，b———资源向量

向量P_j对应的决策变量为x_j

用矩阵描述时为：

$$max\ z = CX$$
$$AX = b$$
$$X \geq 0$$

其中

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ \cdots & \cdots & & \cdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} = (P_1, P_2, \cdots, P_n)；0 = \begin{bmatrix} 0 \\ 0 \\ \cdots \\ 0 \end{bmatrix}$$

A———约束条件的$m \times n$维**系数矩阵**,一般$m < n$

3.3 转化为标准形式

1. 目标函数是求最小值 $minZ = CX$，令 $S = -Z$ 则 $maxS = -CX$
2. 约束不等式为小于等于不等式，可以在左端加入非负松弛变量(Slack Variable)，转变为等式，比如：

$$x_1 + 2x_2 \leq 9 \Rightarrow \begin{cases} x_1 + 2x_2 + x_3 = 9 \\ x_3 \geq 0 \end{cases}$$

3. 约束不等式为大于等于不等式时，可以在左端减去一个非负松弛变量(Slack Variable)，也称剩余变量(Surplus Variable)，变为等式
4. 若约束条件右边的某一常数 $b_i < 0$，这时只要在 b_i 相对应的约束方程两边乘上 -1
5. 若存在取值无约束的变量，可转变为两个非负变量的差，比如：

$$-\infty \leq x_k \leq +\infty \Rightarrow \begin{cases} x_k = x_m - x_n \\ x_m, x_n \geq 0 \end{cases}$$

4. 概念和参数说明

4.1 线性规划问题的解的概念

4.1.1 可行解

满足$(M_1^{'})$中(s.t.)两式的解$X=(x_1,x_2,...,x_n)^T$,称为线性规划问题的可行解,其中使目标函数达到最大值的可行解称为**最优解。**

4.1.2 基、基向量和基解

A为约束条件的$m\times n$维系数矩阵,其秩为m。B是矩阵A中$m\times m$的矩阵，如果B是由A中m个线性独立的列向量组成，则称B是线性规划问题的一个基。

可设

$$B=\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ \cdots & \cdots & & \cdots \\ a_{m1} & a_{m2} & \cdots & a_{mm} \end{bmatrix}=(P_1,P_2,\cdots,P_m)$$

称$P_j(j=1,2,\cdots,m)$为基向量，与基向量P_j相应的变量$x_j(j=1,2,...,m)$为基变量，其他的决策变量称为非基变量。

由约束条件组可得方程组：

$$\sum_{j=1}^{m}P_jx_j=b-\sum_{j=m+1}^{n}P_jx_j$$

B是这个方程组的一个基，设X_B是对应于这个基的基变量$X_B=(x_1,x_2,,,x_m)^T$

若另非基变量$x_{m+1},x_{m+2},...,x_n=0$,这时变量的个数等于线性方程的个数,

利用高斯消去法可求出一个解

$$X=(x_1,x_2...,x_m,0,...,0)^T$$

该解的非零分量的数目不大于方程个数m,称X为**基解。**(有一个基，就可以求出一个基解)

约束方程组(1-5)具有基解的数目最多是C_n^m个

4.1.3 基可行解

满足**非负条件**的基解

4.1.4 可行基

对应于基可行解的基。

4.2 单纯形表

将方程组

$$(1-23)\begin{cases} x_1 \qquad + a_{1,m+1}x_{m+1}+\cdots+a_{1n}x_n=b_1 \\ \quad x_2 \quad + a_{2,m+1}x_{m+1}+\cdots+a_{2n}x_n=b_2 \\ \qquad\qquad \cdots \\ \qquad x_m+a_{m,m+1}x_{m+1}+\cdots+a_{mn}x_n=b_m \\ \\ -z+c_1x_1+c_2x_2+...+c_nx_n=0 \end{cases}$$

写成增广矩阵

$$
\left[\begin{array}{cccccccc|c}
-z & x_1 & x_2 & \cdots & x_m & x_{m+1} & \cdots & x_n & b \\
0 & 1 & 0 & \cdots & 0 & a_{1,m+1} & \cdots & a_{1n} & b_1 \\
0 & 0 & 1 & \cdots & 0 & a_{2.m+1} & \cdots & a_{2n} & b_2 \\
\cdots \\
0 & 0 & 0 & \cdots & 1 & a_{m,m+1} & \cdots & a_{mn} & b_m \\
1 & c_1 & c_2 & \cdots & c_m & c_{m+1} & \cdots & c_n & 0
\end{array}\right]
$$

将z看作不参与基变换的基变量,它与$x_1, x_2, \dots, x_m$的系数构成一个基,这时可采用行初等变换将$c_1, c_2, \dots, c_m$变换为零,使其对应的系数矩阵为单位矩阵。得到

$$
\left[\begin{array}{cccccccc|c}
-z & x_1 & x_2 & \cdots & x_m & x_{m+1} & \cdots & x_n & b \\
0 & 1 & 0 & \cdots & 0 & a_{1,m+1} & \cdots & a_{1n} & b_1 \\
0 & 0 & 1 & \cdots & 0 & a_{2,m+1} & \cdots & a_{2n} & b_2 \\
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
0 & 0 & 0 & \cdots & 1 & a_{m,m+1} & \cdots & a_{mn} & b_m \\
1 & 0 & 0 & \cdots & 0 & c_{m+1}-\sum_{i=1}^{m} c_i a_{i,m+1} & \cdots & c_n-\sum_{i=1}^{m} c_i a_{i,n} & -\sum_{i=1}^{m} c_i b_i
\end{array}\right]
$$

可根据上述增广矩阵设计计算表，见表1-2。

```
\begin{tabular}{c|c|c|c|c|c|c|c|c|c}
\hline
\multicolumn{3}{c|}{$c_{j} \rightarrow$} & $c_1$ & $\ldots$ & $c_{m}$ &
$c_{m+1}$ & $\ldots$ & $c_{n}$ &  \multirow{2}{c|}{$\theta_{i}$ }\\
\hline
$C_{B}$ & $X_{B}$ & $b$ & $x_{1}$ & $\cdots$ & $x_{m}$ & $x_{m+1}$ & $\cdots$ &
$x_{n}$  \\
\hline$c_{1}$ & $x_{1}$ & $b_1$ & 1 & $\cdots$ & 0 & $a_{1, m+1}$ & $\cdots$ &
$a_{1,n}$ & $\theta_{1}$ \\
$c_{2}$ & $x_{2}$ & $b_2$ & 0 & $\cdots$ & 0 & $a_{2, m+1}$ &$\cdots$ &
 $a_{2,n}$ & $\theta_{2}$ \\
$\ldots$ & $\cdots$ & $\cdots$ & $\cdots$ & $\cdots$ & $\cdots$ & $\cdots$ &
$\cdots$ & $\cdots$ &$\cdots$&  \\
$c_{m}$ & $x_{m}$ & $b_{m}$ & 0 & $\cdots$ & 1 & $a_{m, m+1}$ & $\cdots$ &
$a_{m,n}$& $\theta_m$\\
\hline
\multicolumn{2}{c|}{$-z$ }& $-\sum_{i=1}^{m} c_{i} b_{i}$ & 0 & $\cdots$ & 0 &
$c_{m+1}-\sum_{i=1}^{m} c_{i} a_{i, m+1}$ & $\cdots$ & $c_{n}-\sum_{i=1}^{m}
c_{i} a_{i,n}$ \\
\hline
\end{tabular}
```

可根据上述增广矩阵设计计算表,见表 1-2。

表 1-2

		$c_j \rightarrow$	c_1	$\cdots$	c_m	c_{m+1}	$\cdots$	c_n	θ_i
C_B	X_B	b	x_1	$\cdots$	x_m	x_{m+1}	$\cdots$	x_n	
c_1	x_1	b_1	1	$\cdots$	0	$a_{1,m+1}$	$\cdots$	a_{1n}	θ_1
c_2	x_2	b_2	0		0	$a_{2,m+1}$		a_{2n}	θ_2
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
c_m	x_m	b_m	0	$\cdots$	1	$a_{m,m+1}$	$\cdots$	a_{mn}	θ_m
$-z$		$-\sum_{i=1}^{m} c_i b_i$	0	$\cdots$	0	$c_{m+1} - \sum_{i=1}^{m} c_i a_{i,m+1}$	$\cdots$	$c_n - \sum_{i=1}^{m} c_i a_{in}$	

X_B列中填入基变量,这里是$x_1, x_2, ..., x_m$;

C_B列中填入基变量的价值系数,这里是$c_1, c_2, ..., c_m$;它们是与基变量相对应的;

b列中填入约束方程组右端的常数;

c_j行中填入基变量的价值系数$c_1, c_2, ..., c_n$;

θ_i列的数字是在确定换入变量后,按θ规则计算后填入;

最后一行称为检验数行,对应各非基变量x_j的检验数是

$$c_j - \sum_{i=1}^{m} c_i a_{ij}, j = 1, 2, ..., n$$

表1-2称为初始单纯形表,每迭代一步构造一个新单纯形表。

4.3 单纯形法的矩阵描述

设线性规划问题:max $z = CX$; $AX \leqslant b$; $X \geqslant 0$。给这线性规划问题的约束条件加入松弛变量 $X_s = (x_{s1}, x_{s2}, \cdots, x_{sm})^T$ 以后,得到标准型:

$$\max z = CX + OX_s;\ AX + IX_s = b;\ X, X_s \geqslant 0$$

这里的 I 是 $m \times m$ 单位矩阵。若以 X_s 为基变量,这时可标记成 X_B。其对应的单位矩阵就是基矩阵 B,这时将系数矩阵(A, I)分为(B, N)两块。N 是非基变量的系数矩阵,相应的决策变量被分为 $X = \begin{bmatrix} X_B \\ X_N \end{bmatrix}$,同时目标函数的系数 C 分为 C_B, C_N 分别对应于基变量和非基变量,并记作 $C = (C_B, C_N)$。

经过迭代运算后,在基矩阵中可能还存在松弛变量或全无松弛变量。为了阐述方便起见,设

$$X_B = \begin{bmatrix} X_{B1} \\ X_{S1} \end{bmatrix};\ X_N = \begin{bmatrix} X_{N1} \\ X_{S2} \end{bmatrix};\ X_S = \begin{bmatrix} X_{S1} \\ X_{S2} \end{bmatrix};\ A = \begin{bmatrix} B \\ N \end{bmatrix};\ N = \begin{bmatrix} N_1 \\ S_2 \end{bmatrix}$$

目标函数

$$max\ z = C_B X_B + C_N X_N = C_B X_B + C_{N1} X_{N1} + C_{S2} X_{S2}$$

约束条件

$$BX_B + NX_N = BX_B + N_1 X_{N1} + S_2 X_{S2} = b$$

非负条件

$$X_B, X_N \geq 0$$

$X_B = B^{-1}b - B^{-1}N_1X_{N1} - B^{-1}S_2X_{S2}$ (2-4)

整理得(S_2单位矩阵)：(2-5)

$$z = C_BB^{-1}b + (C_{N1} - C_BB^{-1}N_1)X_{N1} + (C_{S2} - C_BB^{-1}I)X_{S2}$$

令非基变量$X_N = 0$,得到一个基可行解

$$X^{(1)} = \begin{bmatrix} B^{-1}b \\ 0 \end{bmatrix}$$

这时目标函数$z = C_BB^{-1}b$

(1)非基变量的系数$(C_{N1} - C_BB^{-1}N_1)$就是检验数，因为$C_{S2} = 0$,I为单位矩阵，所以X_{S2}的系数为$-C_BB^{-1}$,

X_B在(2-5)中的系数是0，实质是$C_B - C_BB^{-1}B = 0$

(2)用矩阵表达时，θ规则的表达式

$$\theta = min[\frac{(B^{-1}b)_i}{(B^{-1}P_j)_i}|(B^{-1}P_j)_i > 0]$$

这里的$(B^{-1}b)_i$表示$(B^{-1}b)$中的第i个元素,$(B^{-1}P_j)_i$表示向量$(B^{-1}P_j)$中的第i个元素。

(3)单纯形表与矩阵表示的关系

由 (3-5)、 (3-6)式知

$$X_B + B^{-1}NX_N = B^{-1}b$$

$$-z + (C_N - C_BB^{-1}N)X_N = -C_BB^{-1}b$$

上两式用矩阵表示为

$$\begin{bmatrix} 0 & 1 & B^{-1}N \\ 1 & 0 & C_N - C_BB^{-1}N \end{bmatrix}\begin{bmatrix} -z \\ X_B \\ X_N \end{bmatrix} = \begin{bmatrix} B^{-1}b \\ -C_BB^{-1}b \end{bmatrix}$$

	基变量	非基变量		RHS 等式右边	
	X_S	X_B	X_N		$(AX = BX_B + NX_N)$
矩阵系数	I	B	N	b	$\rightarrow IX_S + AX = b$
检验数	0	C_B	C_N	$-z = 0$	$\rightarrow C_B X_B + C_N X_N = z$

	基变量	非基变量 X_N		RHS 等式右边	
	X_B	X_{N_1}	X_{S_2}		
系数矩阵	$B^{-1}B = I$	$B^{-1}N_1$	B^{-1}	$B^{-1}b$	$\rightarrow (2\text{-}4)$
检验数	$C_B - C_B B^{-1} B = 0$	$C_{N_1} - C_B B^{-1} N_1$	$-C_B B^{-1}$	$-z = -C_B B^{-1} b$	$\rightarrow (2\text{-}5)$

5. 求解方法

4.1 图解法(Graphical Solution)

4.1.1 步骤

1. 确定约束区域
2. 画出目标函数等值线，平移目标函数等值线求最优解

4.2 单纯形方法(Simplex Method)

4.2.1 基本思路

> 单纯形法是从一个初始解开始, 不断地改进现有的解, 直到所要求的目标函数值被满足为止。即是进行反复的迭代, 直到得到需要的最优解。
>
> (单纯形法就是通过设置不同的基向量，经过矩阵的线性变换，求得基可行解（可行域顶点），并判断该解是否最优，否则继续设置另一组基向量，重复执行以上步骤，直到找到最优解。)

4.2.2 步骤

---

1. **找出初始可行基,确定初始基可行解,建立初始单纯形表。**

4.2.2.1 初始基可行解的确定

(1)若线性规划问题

$$max\ z = \sum_{j=1}^{n} c_j x_j$$

$$\sum_{j=1}^{n} P_j x_j = b$$

$$x_j \geq 0, j = 1, 2, \ldots, n$$

从$P_j(j = 1, 2, \ldots, n)$中一般能直接观察到一个初始可行基

$$B=(P_1,P_2,\ldots,P_m)=\begin{bmatrix}1&0&\cdots&0\\0&1&\cdots&0\\\cdots&\cdots&\cdots&\cdots\\0&0&\cdots&1\end{bmatrix}$$

(2)对所有约束条件都是≤的不等式，可以利用化为标准型的方法,在每个约束条件的左端加上一个松弛变量。经过整理,重新对x_j及$a_{ij}(i=1,2,\ldots,m;j=1,2,\ldots,n)$进行编号,则可得下列方程组

$$(1-23)\begin{cases}x_1 \qquad +a_{1,m+1}x_{m+1}+\cdots+a_{1n}x_n=b_1\\ \quad x_2 \quad +a_{2,m+1}x_{m+1}+\cdots+a_{2n}x_n=b_2\\ \qquad\qquad \cdots\\ \qquad x_m+a_{m,m+1}x_{m+1}+\cdots+a_{mn}x_n=b_m\\ \qquad\qquad x_1,x_2,\ldots,x_n\geq 0\end{cases}$$

显然得到一个$m\times m$单位矩阵

$$B=(P_1,P_2,\ldots,P_m)=\begin{bmatrix}1&0&\cdots&0\\0&1&\cdots&0\\\cdots&\cdots&\cdots&\cdots\\0&0&\cdots&1\end{bmatrix}$$

令$x_{m+1},x_{m+2},\ldots,x_n=0$,可得

$$x_i=b_i\ \ (i=1,2,\ldots,m)$$

因$b_i\geq 0$,所以得到一个初始基可行解

$$X=(x_1,x_2,\ldots,x_m,0,\ldots,0)^T=(b_1,b_2,\ldots,b_m,0,\ldots,0)^T$$

(3)对所有约束条件是≥形式的不等式及等式约束情况,若不存在单位矩阵时,就采用人造基方法。即对不等式约束减去一个非负的剩余变量后,再加上一个非负的人工变量;对于等式约束再加上一个非负的人工变量,总能得到一个单位矩阵。

---

2. **检验各非基变量x_j的检验数是**

$$\sigma_j=c_j-\sum_{i=1}^{m}c_ia_{ij},\quad \text{若}\,\sigma_j\leq 0,\quad j=m+1,\ldots,n$$

则已得到最优解,可停止计算。否则转入下一步。

4.2.2.2 最优性检验与解的判别

对线性规划问题的求解结果可能出现唯一最优解、无穷多最优解、无界解和无可行解四种情况,为此需要建立对解的判别准则。

一般情况下,经过迭代后(1-23)式变成

$$x_i=b_i^{'}-\sum_{j=m+1}^{n}a_{ij}^{'}x_j\ (i=1,2,\ldots,m)$$

把目标函数z用非基变量表示

$$z=\sum_{i=1}^{m}c_ib_i^{'}+\sum_{j=m+1}^{n}(c_j-\sum_{i=1}^{m}c_ia_{ij}^{'})x_j$$

令

$$z_0=\sum_{i=1}^{m}c_ib_i^{'},z_j=\sum_{i=1}^{m}c_ia_{ij}^{'},j=m+1,\ldots,n$$

于是

$$z=z_0+\sum_{j=m+1}^{n}(c_j-z_j)x_j$$

再令

$$\sigma_j=c_j-z_j\ (j=m+1,\ldots,n)$$

则

$$z=z_0+\sum_{j=m+1}^{n}\sigma_jx_j$$

1. 最优解的判别定理

   若$X^{(0)}=(b_1^{'},b_2^{'},\ldots,b_m^{'},0,\ldots,0)^T$为对应基B的一个基可行解，且对于所有$j=m+1,\ldots,n$，有$\sigma_j\le 0$,则$X^{(0)}$为最优解，称$\sigma_j$为检验数

2. 无穷多最优解判别定理

   若$X^{(0)}=(b_1^{'},b_2^{'},\ldots,b_m^{'},0,\ldots,0)^T$为一个基可行解，对于所有$j=m+1,\ldots,n$，有$\sigma_j\le 0$,且存在某个非基变量的检验数$\sigma_{m+k}=0$，则线性规划问题有无穷多可行解

3. 无界解判别定理

   若$X^{(0)}=(b_1^{'},b_2^{'},\ldots,b_m^{'},0,\ldots,0)^T$为一个基可行解，有一个$\sigma_{m+k}>0$,并且对于$i=1,2,\ldots,m$,有

$a_{1,m+k}\le 0$(对应的系数列)，那么该线性规划问题具有无界解(或称无最优解)。

以上讨论都是**针对标准型**,即**求目标函数极大化**时的情况。当求目标函数极小化时,一种情况如前所述,将其化为标准型。如果不化为标准型,只需在上述1,2点中把$\sigma_j\le 0$改为$\sigma_j\ge 0$,第3点中将$\sigma_{m+k}>0$改写为$\sigma_{m+k}<0$即可。

---

3. **在$\sigma_j>0,j=m+1,\ldots,n$中,若有某个$\sigma_k$对应$x_k$的系数列向量$P_k\le 0$,则此问题是无界,停止计算。否则,转入下一步。**

---

4. **根据$\max(\sigma_j>0)=\sigma_k$,确定$x_k$为换入变量,按θ规则计算**

$$\theta=min(\frac{b_i}{a_{i,k}}|a_{i,k}>0)=\frac{b_l}{a_{lk}},\quad i=1,2,\ldots,m$$

   可确定x_l为换出变量,转入下一步。

4.2.2.3 基变换

(从一个基可行解到另一个基可行解的变换,就是进行一次基变换。从几何意义上讲,就是从可行域的一个顶点转向另一个顶点)

若初始基可行解$X^{(0)}$不是最优解及不能判别无界时,需要找一个新的基可行解。具体做法是从原可行解基中换一个列向量(当然要保证线性独立),得到一个新的可行基,这称为基变换。为了换基,先要确定换入变量,再确定换出变量,让它们相应的系数列向量进行对换,就得到一个新的基可行解。

1. 换入变量的确定

   当某些$\sigma_j > 0$时,x_j增加则目标函数值还可以增大,这时要将某个非基变量x_j换到基变量中去(称为换入变量)。若有两个以上的σ_j>0,为了使目标函数值增加得快,一般选$\sigma_j > 0$中的大者,即$\max_j(\sigma_j > 0) = \sigma_k$则对应的xk为换入变量。

2. 换出变量的确定

   设$P_1, P_2, ..., P_m$是一组线性独立的向量组，对应基可行解$X^{(0)}$，带入约束方程组有

$$\sum_{i=1}^{m} x_i^{(0)} P_i = b$$

   其他非基变量的列向量可以用$P_1, P_2, ..., P_m$线性表示，若确定P_{m+t}为换入变量，必然有一组不全为0的数使得

$$P_{m+t} = \sum_{i=1}^{m} \beta_{i,m+t} P_i$$

   在上式两边同时乘以正数θ,再加到上上式中，得

$$\sum_{i=1}^{m} (x_i^{(0)} - \theta\beta_{i,m+t}) P_i + \theta P_{m+t} = b$$

   θ取$\frac{x_i^{(0)}}{\beta_{i,m+t}}$中大于零的且最小的数。设$x_l$为换出变量($l$在1~m之间)。将这个$\theta = \frac{x_i^{(0)}}{\beta_{i,m+t}}$代入X中，得到新的基可行解

$$X^{(1)} = [x_1^{(0)} - \theta\beta_{1,m+t}, ..., 0, ..., x_m^{(0)} - \theta\beta_{m,m+t}, 0, ..., \frac{x_l^{(0)}}{\beta_{l,m+t}}, ..., 0]$$

---

5. **以a_{lk}为主元素进行迭代(即用高斯消去法或称为旋转运算),把x_k所对应的列向量P_k转变为P_l**

$$P_k = \begin{bmatrix} a_{1k} \\ a_{2k} \\ \cdots \\ a_{lk} \\ \cdots \\ a_{mk} \end{bmatrix} P_l = \begin{bmatrix} 0 \\ \cdots \\ 1 \\ 0 \\ \cdots \\ 0 \end{bmatrix}$$

   将X_B列中的x_l换为x_k,得到新的单纯形表。重复(2)~(5),直到终止。

4.2.2.4 迭代(旋转运算)

上述讨论的基可行解的转换方法是用**向量方程**来描述,在实际计算时不太方便,因此采用**系数矩阵法**。

约束方程组经过处理很容易得到

$$(1-23)\begin{cases} x_1 \quad\quad + a_{1,m+1}x_{m+1} + \cdots + a_{1n}x_n = b_1 \\ \quad x_2 \quad + a_{2,m+1}x_{m+1} + \cdots + a_{2n}x_n = b_2 \\ \quad\quad\quad \cdots \\ \quad\quad x_m + a_{m,m+1}x_{m+1} + \cdots + a_{mn}x_n = b_m \\ \quad\quad x_1, x_2, ..., x_n \geq 0 \end{cases}$$

其中$x_1, x_2, ..., x_m$为基变量，令其他非基变量等于零，可得到一个基可行解。若它不是最优解,则要另找一个使目标函数值增大的基可行解。这时从非基变量中确定x_k为换入变量。这时θ为

$$\theta = min(\frac{b_i}{a_{i,k}} | a_{i,k} > 0), i = 1, 2, ..., m$$

按θ规则确定x_l为换出变量，x_k, x_l的系数列向量分别为

$$P_k=\begin{bmatrix}a_{1k}\\a_{2k}\\\cdots\\a_{lk}\\\cdots\\a_{mk}\end{bmatrix} P_l=\begin{bmatrix}0\\\cdots\\1\\0\\\cdots\\0\end{bmatrix}$$

为了使x_k与x_l进行对换,须把P_k变为单位向量,这可以通过(1-23)式系数矩阵的增广矩阵进行初等变换来实现。

$$\begin{array}{ccc} x_1\cdots x_l\cdots x_m & x_{m+1}\cdots x_k\cdots x_n & b \\ \end{array}$$

$$\left[\begin{array}{cc|c} 1\cdots\cdots\cdots 0 & a_{1,m+1}\cdots a_{1,k}\cdots a_{1n} & b_1 \\ \cdots & \cdots & \cdots \\ 0\cdots 1\cdots 0 & a_{l,m+1}\cdots a_{l,k}\cdots a_{ln} & b_l \\ \cdots & \cdots & \cdots \\ 0\cdots\cdots\cdots 1 & a_{m,m+1}\cdots a_{m,k}\cdots a_{mn} & b_m \end{array}\right]$$

步骤

(1)将增广矩阵（1-34）第l行除以a_{lk},得到

$$[0,\ldots,0,\frac{1}{a_{lk}},0,\ldots,0,\frac{a_{l,m+1}}{a_{l,k}},\ldots,1,\ldots,\frac{a_{ln}}{a_{lk}}|\frac{b_l}{a_{lk}}]\ (1-35)$$

(2)将(1-34)中x_k列中各元素，除a_{lk}变换为1以外，其他都变换为0。

将(1-35)式乘以$a_{ik}(i\neq l)$后，从(1-34)式子的第i行减去，得到新的第i行

(3)经过初等变换后的新增广矩阵是(1-36)

$$\begin{array}{ccc} x_1\cdots x_l\cdots x_m & x_{m+1}\cdots x_k\cdots x_n & b \\ \end{array}$$

$$\left[\begin{array}{cc|c} 1\cdots-\frac{a_{1k}}{a_{lk}}\cdots 0 & a'_{1,m+1}\cdots 0\cdots a'_{1n} & b'_{1n} \\ \cdots & \cdots & \cdots \\ 0\cdots+\frac{1}{a_{1k}}\cdots 0 & a'_{l,m+1}\cdots 1\cdots a'_{ln} & b'_l \\ \cdots & \cdots & \cdots \\ 0\cdots-\frac{a_{mk}}{a_{lk}}\cdots 1 & a'_{m,m+1}\cdots 0\cdots a'_{mn} & b'_m \end{array}\right]$$

(4)非基变量为0，得到一个基可行解

$$X^{(1)}=(b_1^{'},\ldots,0,\ldots,b_m^{'},0,\ldots,b_k^{'},0,\ldots,0)$$

4.2.3 实例

某工厂在计划期内要安排生产Ⅰ、Ⅱ两种产品,已知生产单位产品所需的设备台时及A、B两种原材料的消耗,如表1-1所示。

例 1 某工厂在计划期内要安排生产Ⅰ、Ⅱ两种产品,已知生产单位产品所需的设备台时及 A、B 两种原材料的消耗,如表 1-1 所示。

表 1-1

	Ⅰ	Ⅱ	
设　备	1	2	8 台时
原材料 A	4	0	16 kg
原材料 B	0	4	12 kg

可以用数学模型表示

目标函数 $$max\ z = 2x_1 + 3x_2$$

满足约束条件：
$$\begin{cases} x_1 + 2x_2 \le 8 \\ 4x_1 \le 16 \\ 4x_2 \le 12 \\ x_1, x_2 \ge 0 \end{cases}$$

化为标准形式

$$max\ z = 2x_1 + 3x_2 + 0x_3 + 0x_4 + 0x_5$$
$$\begin{cases} x_1 + 2x_2 + x_3 = 8 \\ 4x_1 + x_4 = 16 \\ 4x_2 + x_5 = 12 \\ x_1, x_2, x_3, x_4, x_5 \ge 0 \end{cases}$$

约束方程的系数矩阵

$$A = (P_1, P_2, P_3, P_4, P_5) = \begin{bmatrix} 1 & 2 & 1 & 0 & 0 \\ 4 & 0 & 0 & 1 & 0 \\ 0 & 4 & 0 & 0 & 1 \end{bmatrix}$$

1. 很容易看出初始可行基

$$B = (P_3, P_4, P_5) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

基变量为x_3, x_4, x_5。它们和z都可以用非基变量x_1, x_2表示

$$\begin{cases} x_3 = 8 - x_1 - 2x_2 \\ x_4 = 16 - 4x_1 \\ x_5 = 12 - 4x_2 \end{cases}$$

令非基变量为0，得到初始基可行解

$$X^{(0)} = (0, 0, 8, 16, 12)\$z = \$$$

初始单纯行表为

表 1-3

$c_j \rightarrow$			2	3	0	0	0	θ
C_B	X_B	b	x_1	x_2	x_3	x_4	x_5	
0	x_3	8	1	2	1	0	0	4
0	x_4	16	4	0	0	1	0	-
0	x_5	12	0	[4]	0	0	1	3
$-z$		0	2	3	0	0	0	

2. 和3. 因检验数都大于0，且P_1, P_2有正分量存在，转入下一步

3. $max(\sigma_1, \sigma_2) = max(2, 3) = 3$,所以对应$x_2$为换入变量，计算$\theta$

$$\theta = min(\frac{b_i}{a_{i,2}} | a_{i,2} > 0) = min(8/2, -, 12/4) = 3$$

则它对应的x_5为换出变量。x_2所在列和x_5所在行的交叉处[4]称为主元素或枢元素(pivot element)。

4. 以[4]为主元素进行旋转运算,即初等行变换,使P2变换为(0,0,1),在x_B列中将x_2替换x_5,于是得到新表1-4。

表 1-4

$c_j \to$			2	3	0	0	0	θ_i
C_B	X_B	b	x_1	x_2	x_3	x_4	x_5	
0	x_3	2	[1]	0	1	0	-1/2	2
0	x_4	16	4	0	0	1	0	4
3	x_2	3	0	1	0	0	1/4	-
$-z$		9	2	0	0	0	-3/4	

得到新的基可行解为$x^{(1)}=(0,3,2,16,0)^T$，目标函数的取值为$z=9$。

4. (1-4)中x_1的$\sigma=2>0$,所以x_1为换入变量，重复运算，得到表1-5。

表 1-5

$c_j \to$			2	3	0	0	0	θ_i
C_B	X_B	b	x_1	x_2	x_3	x_4	x_5	
2	x_1	2	1	0	1	0	-1/2	-
0	x_4	8	0	0	-4	1	[2]	4
3	x_2	3	0	1	0	0	1/4	12
$-z$		-13	0	0	-2	0	1/4	

5. (1-5)中x_5,$\sigma=1/4>0$,所以x_5为换入变量，重复运算，得到表1-6。

续表

$c_j \to$			2	3	0	0	0	θ_i
C_B	X_B	b	x_1	x_2	x_3	x_4	x_5	
2	x_1	4	1	0	0	1/4	0	
0	x_5	4	0	0	-2	1/2	1	
3	x_2	2	0	1	1/2	-1/8	0	
$-z$		-14	0	0	-3/2	-1/8	0	

发现最后一行所有检验数都已经为负或0。表示目标函数值已不可能再增大，于是得到最优解

$$X^*=X^{(3)}=(4,2,0,0,4)^T$$

目标函数值

$$z^*=14$$

4.2.4 代码实现

4.2.4.1 python实现

```python
# 线性规划-单纯形算法
import numpy as np
np.set_printoptions(precision=4, suppress=True, threshold=np.inf)
# 线性规划转化为松弛形式
def get_loose_matrix(matrix):
    row, col = matrix.shape
    loose_matrix = np.zeros((row, row + col))
    for i, _ in enumerate(loose_matrix):
        loose_matrix[i, 0: col] = matrix[i]
        loose_matrix[i, col + i] = 1.0  # 对角线
    return loose_matrix
# 松弛形式的系数矩阵A、约束矩阵B和目标函数矩阵C组合为一个矩阵
```

```python
def join_matrix(a, b, c):
    row, col = a.shape
    s = np.zeros((row + 1, col + 1))
    s[1:, 1:] = a  # 右下角是松弛系数矩阵A
    s[1:, 0] = b  # 左下角是约束条件值矩阵B
    s[0, 1: len(c) + 1] = c  # 右上角是目标函数矩阵C
    return s
# 旋转矩阵–替换替出替入变量的角色位置
def pivot_matrix(matrix, k, j):
    # 单独处理替出变量所在行，需要除以替入变量的系数matrix[k][j]
    matrix[k] = matrix[k] / matrix[k][j]
    # 循环除了替出变量所在行之外的所有行
    for i, _ in enumerate(matrix):
        if i != k:
            matrix[i] = matrix[i] - matrix[k] * matrix[i][j]
# 根据旋转后的矩阵，从基本变量数组中得到一组基解
def get_base_solution(matrix, base_ids):
    X = [0.0] * (matrix.shape[1])  # 解空间
    for i, _ in enumerate(base_ids):
        X[base_ids[i]] = matrix[i + 1][0]
    return X
# 构造辅助线性规划
def Laux(matrix, base_ids):
    l_matrix = np.copy(matrix)
    # 辅助矩阵的最后一列存放x0的系数，初始化为-1
    l_matrix = np.column_stack((l_matrix, [-1] * l_matrix.shape[0]))
    # 辅助线性函数的目标函数为z = x0
    l_matrix[0, :-1] = 0.0
    l_matrix[0, -1] = 1
    k = l_matrix[1:, 0].argmin() + 1  # 选择一个b最小的那一行的基本变量作为替出变量
    j = l_matrix.shape[1] - 1   # 选择x0作为替入变量
    # 第一次旋转矩阵，使得所有b为正数
    pivot_matrix(l_matrix, k=k, j=j)
    base_ids[k - 1] = j  # 维护基本变量索引数组
    # 用单纯形算法求解该辅助线性规划
    l_matrix = simplex(l_matrix, base_ids)
    # 如果求解后的辅助线性规划中x0仍然是基本变量，需要再次旋转消去x0
    if l_matrix.shape[1] - 1 in base_ids:
        j = np.where(l_matrix[0, 1:] != 0)[0][0] + 1   # 找到矩阵第一行(目标函数)系数
不为0的变量作为替入变量
        k = base_ids.index(l_matrix.shape[1] - 1) + 1  # 找到x0作为基本变量所在的那一
行，将x0作为替出变量
        pivot_matrix(l_matrix, k=k, j=j)   # 旋转矩阵消去基本变量x0
        base_ids[k - 1] = j  # 维护基本变量索引数组
    return l_matrix, base_ids
# 从辅助函数中恢复原问题的目标函数
def resotr_from_Laux(l_matrix, z, base_ids):
    z_ids = np.where(z != 0)[0] - 1  # 得到目标函数系数不为0的索引数组(即基本变量索引数
组)
    restore_matrix = np.copy(l_matrix[:, 0:-1])  # 去掉x0那一列
    restore_matrix[0] = z  # 初始化矩阵的第一行为原问题的目标函数向量
    for i, base_v in enumerate(base_ids):
        # 如果原问题的基本变量存在新基本变量数组中，说明需要替换消去
        if base_v in z_ids:
            restore_matrix[0] -= restore_matrix[0, base_v + 1] *
restore_matrix[i + 1]  # 消去原目标函数中的基本变量
    return restore_matrix
# 单纯形算法求解线性规划
```

```python
def simplex(matrix, base_ids):
    matrix = matrix.copy()
    # 如果目标系数向量里有负数，则旋转矩阵
    while matrix[0, 1:].min() < 0:
        # 在目标函数向量里，选取系数为负数的第一个变量索引，作为替入变量
        j = np.where(matrix[0, 1:] < 0)[0][0] + 1
        # 在约束集合里，选取对替入变量约束最紧的约束行，那一行的基本变量作为替出变量
        k = np.array([matrix[i][0] / matrix[i][j] if matrix[i][j] > 0 else
0x7fff for i in
                      range(1, matrix.shape[0])]).argmin() + 1
        # 说明原问题无界
        if matrix[k][j] <= 0:
            print('原问题无界')
            return None, None
        pivot_matrix(matrix, k, j)  # 旋转替换替入变量和替出变量
        base_ids[k - 1] = j - 1  # 维护当前基本变量索引数组
    return matrix
# 单纯形算法求解步骤入口
def solve(a, b, c, equal=None):
    loose_matrix = get_loose_matrix(a)  # 转化得到松弛矩阵
    if equal is not None:
        loose_matrix = np.insert(loose_matrix, 0, equal, axis=0)
    matrix = join_matrix(loose_matrix, b, c)  # 得到ABC的组合矩阵
    base_ids = list(range(len(c), len(b) + len(c)))  # 初始化基本变量的索引数组
    # 约束系数矩阵有负数约束，证明没有可行解，需要辅助线性函数
    if matrix[:, 0].min() < 0:
        print('构造求解辅助线性规划函数...')
        l_matrix, base_ids = Laux(matrix, base_ids)  # 构造辅助线性规划函数并旋转求解
之
        if l_matrix is not None:
            matrix = resotr_from_Laux(l_matrix, matrix[0], base_ids)  # 恢复原问题
的目标函数
        else:
            print('辅助线性函数的原问题没有可行解')
            return None, None, None
    ret_matrix = simplex(matrix, base_ids)  # 单纯形算法求解拥有基本可行解的线性规划
    X = get_base_solution(ret_matrix, base_ids)  # 得到当前最优基本可行解
    if ret_matrix is not None:
        return matrix, ret_matrix, X
    else:
        print('原线性规划问题无界')
        return None, None, None
if __name__ == '__main__':
    equal = None
    # 不带等式约束的线性规划测试
    # a = np.array([[4, -1], [2, 1], [-5, 2]])
    # b = [8, 10, 2]
    # c = [-1, -1]
    # a = np.array([[1, 1], [-1, -1]])
    # b = [2, -1]
    # c = [1, 2]
    # a = np.array([[1, -1], [-1.5, 1], [50, 20]])
    # b = [0, 0, 2000]
    # c = [-1, -1]
    # a = np.array([[1, 1, 1, 1], [4, 8, 2, 5], [4, 2, 5, 5], [6, 4, 8, 4]])
    # b = [480, 2400, 2000, 3000]
    # c = [-9, -6, -11, -8]
    # a = np.array([[1, -1], [-1, -1], [-1, 4]])
```

```python
    # b = [8, -3, 2]
    # c = [-1, -3]
    # a = np.array([[-21, 0, 0, 0], [0, -28, -4, 0], [-2, -1, -10, -11]])
    # b = [-500, -600, -250]
    # c = [1, 1, 1, 1]
    # a = np.array([[2, -1], [1, -5]])
    # b = [2, -4]
    # c = [-2, 1]
    # 带等式约束的线性规划测试
    # a = np.array([[1, -2]])
    # equal = [1, 1] + [0] * a.shape[0]
    # b = [7, 4]
    # c = [-2, 3]
    # a = np.array([[1, -2, 1], [4, -1, -2]])
    # equal = [-2, 0, 1] + [0] * a.shape[0]
    # b = [1, 11, -3]
    # c = [-3, 1, 1]
    a = np.array([[-2, 5, -1], [1, 3, 1]])
    equal = [1, 1, 1] + [0] * a.shape[0]
    b = [7, -10, 12]
    c = [-2, -3, 5]
    matrix, ret_matrix, X = solve(a, b, c, equal=equal)
    print('本次迭代的最优解为：{}'.format(np.round(X[0: len(c)], 4)))
    print('该线性规划的最优值是：{}'.format(np.round(-ret_matrix[0][0], 4)))
```

```python
'''
线性规划实战—连续投资问题
考虑下列投资项目：
项目A：在第1~4年每年年初可以投资，并于次年年末收回本利115%；
项目B：第3年年初可以投资，一直到第5年年末能收回本利125%，且规定最大投资额不超过4万元；
项目C：第2年年初可以投资，一直到第5年年末能收回本利140%，且规定最大投资额不超过3万元；
项目D：5年内每一年年初均可以购买公债，并于当年年末归还本金，并加获得利息6%。
如果你有10万元本金，求确定一个有效的投资方案，使得第5年年末你拥有的资金的本利总额最大？
'''
import numpy as np
from copy import deepcopy
np.set_printoptions(precision=4, suppress=True, threshold=np.inf)
# 线性规划转化为松弛形式
def get_loose_matrix(matrix):
    row, col = matrix.shape
    loose_matrix = np.zeros((row, row + col))
    for i, _ in enumerate(loose_matrix):
        loose_matrix[i, 0: col] = matrix[i]
        loose_matrix[i, col + i] = 1.0  # 对角线
    return loose_matrix
# 松弛形式的系数矩阵A、约束矩阵B和目标函数矩阵C组合为一个矩阵
def join_matrix(a, b, c):
    row, col = a.shape
    s = np.zeros((row + 1, col + 1))
    s[1:, 1:] = a  # 右下角是松弛系数矩阵A
    s[1:, 0] = b  # 左下角是约束条件值矩阵B
    s[0, 1: len(c) + 1] = c  # 右上角是目标函数矩阵C
    return s
# 旋转矩阵—替换替出替入变量的角色位置
def pivot_matrix(matrix, k, j):
    # 单独处理替出变量所在行，需要除以替入变量的系数matrix[k][j]
    matrix[k] = matrix[k] / matrix[k][j]
```

```
    # 循环除了替出变量所在行之外的所有行
    for i, _ in enumerate(matrix):
        if i != k:
            matrix[i] = matrix[i] - matrix[k] * matrix[i][j]
# 根据旋转后的矩阵，从基本变量数组中得到一组基解
def get_base_solution(matrix, base_ids):
    X = [0.0] * (matrix.shape[1])  # 解空间
    for i, _ in enumerate(base_ids):
        X[base_ids[i]] = matrix[i + 1][0]
    return X
# 构造辅助线性规划
def Laux(matrix, base_ids):
    l_matrix = np.copy(matrix)
    # 辅助矩阵的最后一列存放x0的系数，初始化为-1
    l_matrix = np.column_stack((l_matrix, [-1] * l_matrix.shape[0]))
    # 辅助线性函数的目标函数为z = x0
    l_matrix[0, :-1] = 0.0
    l_matrix[0, -1] = 1
    k = l_matrix[1:, 0].argmin() + 1  # 选择一个b最小的那一行的基本变量作为替出变量
    j = l_matrix.shape[1] - 1   # 选择x0作为替入变量
    # 第一次旋转矩阵，使得所有b为正数
    pivot_matrix(l_matrix, k=k, j=j)
    base_ids[k - 1] = j  # 维护基本变量索引数组
    # 用单纯形算法求解该辅助线性规划
    l_matrix = simplex(l_matrix, base_ids)
    # 如果求解后的辅助线性规划中x0仍然是基本变量，需要再次旋转消去x0
    if l_matrix.shape[1] - 1 in base_ids:
        j = np.where(l_matrix[0, 1:] != 0)[0][0] + 1   # 找到矩阵第一行(目标函数)系数
不为0的变量作为替入变量
        k = base_ids.index(l_matrix.shape[1] - 1) + 1  # 找到x0作为基本变量所在的那一
行，将x0作为替出变量
        pivot_matrix(l_matrix, k=k, j=j)   # 旋转矩阵消去基本变量x0
        base_ids[k - 1] = j  # 维护基本变量索引数组
    return l_matrix, base_ids
# 从辅助函数中恢复原问题的目标函数
def resotr_from_Laux(l_matrix, z, base_ids):
    z_ids = np.where(z != 0)[0] - 1  # 得到目标函数系数不为0的索引数组(即基本变量索引数
组)
    restore_matrix = np.copy(l_matrix[:, 0:-1])  # 去掉x0那一列
    restore_matrix[0] = z  # 初始化矩阵的第一行为原问题的目标函数向量
    for i, base_v in enumerate(base_ids):
        # 如果原问题的基本变量存在新基本变量数组中，说明需要替换消去
        if base_v in z_ids:
            restore_matrix[0] -= restore_matrix[0, base_v + 1] *
restore_matrix[i + 1]  # 消去原目标函数中的基本变量
    return restore_matrix
# 单纯形算法求解线性规划
def simplex(matrix, base_ids):
    matrix = matrix.copy()
    # 如果目标系数向量里有负数，则旋转矩阵
    while matrix[0, 1:].min() < 0:
        # 在目标函数向量里，选取系数为负数的第一个变量索引，作为替入变量
        j = np.where(matrix[0, 1:] < 0)[0][0] + 1
        # 在约束集合里，选取对替入变量约束最紧的约束行，那一行的基本变量作为替出变量
        k = np.array([matrix[i][0] / matrix[i][j] if matrix[i][j] > 0 else
0x7fff for i in
                      range(1, matrix.shape[0])]).argmin() + 1
```

```
        # print('替出变量为：{}，替入变量为：{}， 值为：{}'.format(k, j, matrix[k]
[j]))
        # 说明原问题无界
        if matrix[k][j] <= 0:
            print('原问题无界')
            return None, None
        pivot_matrix(matrix, k, j)  # 旋转替换替入变量和替出变量
        base_ids[k - 1] = j - 1  # 维护当前基本变量索引数组
    return matrix
# 单纯形算法求解步骤入口
def solve(a, b, c, equal=None):
    loose_matrix = get_loose_matrix(a)  # 转化得到松弛矩阵
    if equal is not None:
        for i, e in enumerate(equal):
            loose_matrix = np.insert(loose_matrix, i, e, axis=0)
    matrix = join_matrix(loose_matrix, b, c)  # 得到ABC的组合矩阵
    base_ids = list(range(len(c), len(b) + len(c)))  # 初始化基本变量的索引数组
    # 约束系数矩阵有负数约束，证明没有可行解，需要辅助线性函数
    if matrix[:, 0].min() < 0:
        print('构造求解辅助线性规划函数...')
        l_matrix, base_ids = Laux(matrix, base_ids)  # 构造辅助线性规划函数并旋转求解
之
        if l_matrix is not None:
            matrix = resotr_from_Laux(l_matrix, matrix[0], base_ids)  # 恢复原问题
的目标函数
        else:
            print('辅助线性函数的原问题没有可行解')
            return None, None, None
    ret_matrix = simplex(matrix, base_ids)  # 单纯形算法求解拥有基本可行解的线性规划
    X = get_base_solution(ret_matrix, base_ids)  # 得到当前最优基本可行解
    if ret_matrix is not None:
        return matrix, ret_matrix, X
    else:
        print('原线性规划问题无界')
        return None, None, None
if __name__ == '__main__':
    x = [0 for _ in range(20)]  # 定义决策变量
    e1, e2, e3, e4, e5 = deepcopy(x), deepcopy(x), deepcopy(x), deepcopy(x),
deepcopy(x)  # 等式约束
    e1[0], e1[15] = 1, 1
    e2[1], e2[11], e2[16], e2[15] = 1, 1, 1, -1.06
    e3[2], e3[7], e3[17], e3[0], e3[16] = 1, 1, 1, -1.15, -1.06
    e4[3], e4[18], e4[1], e4[17] = 1, 1, -1.15, -1.06
    e5[19], e5[2], e5[18] = 1, -1.15, -1.06
    a1, a2 = deepcopy(x), deepcopy(x)  # 不等式约束
    a1[7] = 1
    a2[11] = 1
    a = np.array([a1, a2])
    b = [10, 0, 0, 0, 0] + [4, 3]
    equal = []
    equal.append(e1 + [0] * a.shape[0])
    equal.append(e2 + [0] * a.shape[0])
    equal.append(e3 + [0] * a.shape[0])
    equal.append(e4 + [0] * a.shape[0])
    equal.append(e5 + [0] * a.shape[0])
    c = deepcopy(x)  # 目标函数
    c[3], c[7], c[11], c[19] = -1.15, -1.25, -1.4, -1.06
    # 单纯形法求解数学模型
```

```
    matrix, ret_matrix, X = solve(a, b, c, equal=equal)
    Y = np.round(np.array(X[0: len(c)]).reshape(4, -1).T, 4)  # 得到最优投资方案
    print(Y)
    for i, y in enumerate(Y):
        print('第{}年年初投资组合：项目A：{}万元，项目B：{}万元，项目C：{}万元，项目D：{}万元，该年总投入：{}万元'.format(
            i + 1, y[-4], y[-3], y[-2], y[-1], np.sum(y)))
    total_money = np.abs(np.round(-ret_matrix[0][0], 4))  # 总收入本息
    profit = np.round((total_money - b[0]) / b[0] * 100, 4)  # 总赢利
    print('第5年年末总收入本金+利息为：{}万元，总赢利：{}%'.format(total_money, profit))
```

```
# 线性规划实战—投资的收益和风险
import numpy as np
from copy import deepcopy
import matplotlib.pyplot as plt
np.set_printoptions(precision=4, suppress=True, threshold=np.inf)
# 线性规划转化为松弛形式
def get_loose_matrix(matrix):
    row, col = matrix.shape
    loose_matrix = np.zeros((row, row + col))
    for i, _ in enumerate(loose_matrix):
        loose_matrix[i, 0: col] = matrix[i]
        loose_matrix[i, col + i] = 1.0  # 对角线
    return loose_matrix
# 松弛形式的系数矩阵A、约束矩阵B和目标函数矩阵C组合为一个矩阵
def join_matrix(a, b, c):
    row, col = a.shape
    s = np.zeros((row + 1, col + 1))
    s[1:, 1:] = a  # 右下角是松弛系数矩阵A
    s[1:, 0] = b  # 左下角是约束条件值矩阵B
    s[0, 1: len(c) + 1] = c  # 右上角是目标函数矩阵C
    return s
# 旋转矩阵—替换替出替入变量的角色位置
def pivot_matrix(matrix, k, j):
    # 单独处理替出变量所在行，需要除以替入变量的系数matrix[k][j]
    matrix[k] = matrix[k] / matrix[k][j]
    # 循环除了替出变量所在行之外的所有行
    for i, _ in enumerate(matrix):
        if i != k:
            matrix[i] = matrix[i] - matrix[k] * matrix[i][j]
# 根据旋转后的矩阵，从基本变量数组中得到一组基解
def get_base_solution(matrix, base_ids):
    X = [0.0] * (matrix.shape[1])  # 解空间
    for i, _ in enumerate(base_ids):
        X[base_ids[i]] = matrix[i + 1][0]
    return X
# 构造辅助线性规划
def Laux(matrix, base_ids):
    l_matrix = np.copy(matrix)
    # 辅助矩阵的最后一列存放x0的系数，初始化为-1
    l_matrix = np.column_stack((l_matrix, [-1] * l_matrix.shape[0]))
    # 辅助线性函数的目标函数为z = x0
    l_matrix[0, :-1] = 0.0
    l_matrix[0, -1] = 1
    k = l_matrix[1:, 0].argmin() + 1  # 选择一个b最小的那一行的基本变量作为替出变量
    j = l_matrix.shape[1] - 1   # 选择x0作为替入变量
```

```
    # 第一次旋转矩阵，使得所有b为正数
    pivot_matrix(l_matrix, k=k, j=j)
    base_ids[k - 1] = j  # 维护基本变量索引数组
    # 用单纯形算法求解该辅助线性规划
    l_matrix = simplex(l_matrix, base_ids)
    # 如果求解后的辅助线性规划中x0仍然是基本变量，需要再次旋转消去x0
    if l_matrix.shape[1] - 1 in base_ids:
        j = np.where(l_matrix[0, 1:] != 0)[0][0] + 1   # 找到矩阵第一行(目标函数)系数不为0的变量作为替入变量
        k = base_ids.index(l_matrix.shape[1] - 1) + 1  # 找到x0作为基本变量所在的那一行，将x0作为替出变量
        pivot_matrix(l_matrix, k=k, j=j)   # 旋转矩阵消去基本变量x0
        base_ids[k - 1] = j  # 维护基本变量索引数组
    return l_matrix, base_ids
# 从辅助函数中恢复原问题的目标函数
def resotr_from_Laux(l_matrix, z, base_ids):
    z_ids = np.where(z != 0)[0] - 1  # 得到目标函数系数不为0的索引数组(即基本变量索引数组)
    restore_matrix = np.copy(l_matrix[:, 0:-1])  # 去掉x0那一列
    restore_matrix[0] = z  # 初始化矩阵的第一行为原问题的目标函数向量
    for i, base_v in enumerate(base_ids):
        # 如果原问题的基本变量存在新基本变量数组中，说明需要替换消去
        if base_v in z_ids:
            restore_matrix[0] -= restore_matrix[0, base_v + 1] * restore_matrix[i + 1]  # 消去原目标函数中的基本变量
    return restore_matrix
# 单纯形算法求解线性规划
def simplex(matrix, base_ids):
    matrix = matrix.copy()
    # 如果目标系数向量里有负数，则旋转矩阵
    while matrix[0, 1:].min() < 0:
        # 在目标函数向量里，选取系数为负数的第一个变量索引，作为替入变量
        j = np.where(matrix[0, 1:] < 0)[0][0] + 1
        # 在约束集合里，选取对替入变量约束最紧的约束行，那一行的基本变量作为替出变量
        k = np.array([matrix[i][0] / matrix[i][j] if matrix[i][j] > 0 else 0x7fff for i in
                      range(1, matrix.shape[0])]).argmin() + 1
        # 说明原问题无界
        if matrix[k][j] <= 0:
            print('原问题无界')
            return None, None
        pivot_matrix(matrix, k, j)  # 旋转替换替入变量和替出变量
        base_ids[k - 1] = j - 1  # 维护当前基本变量索引数组
    return matrix
# 单纯形算法求解步骤入口
def solve(a, b, c, equal=None):
    loose_matrix = get_loose_matrix(a)  # 转化得到松弛矩阵
    if equal is not None:
        loose_matrix = np.insert(loose_matrix, 0, equal, axis=0)
    matrix = join_matrix(loose_matrix, b, c)  # 得到ABC的组合矩阵
    base_ids = list(range(len(c), len(b) + len(c)))  # 初始化基本变量的索引数组
    # 约束系数矩阵有负数约束，证明没有可行解，需要辅助线性函数
    if matrix[:, 0].min() < 0:
        print('构造求解辅助线性规划函数...')
        l_matrix, base_ids = Laux(matrix, base_ids)  # 构造辅助线性规划函数并旋转求解之
        if l_matrix is not None:
```

```
            matrix = resotr_from_Laux(l_matrix, matrix[0], base_ids)  # 恢复原问题
的目标函数
        else:
            print('辅助线性函数的原问题没有可行解')
            return None, None, None
    ret_matrix = simplex(matrix, base_ids)  # 单纯形算法求解拥有基本可行解的线性规划
    X = get_base_solution(ret_matrix, base_ids)  # 得到当前最优基本可行解
    if ret_matrix is not None:
        return matrix, ret_matrix, X
    else:
        print('原线性规划问题无界')
        return None, None, None
# 画图显示结果
def draw_result(x, y):
    plt.plot(x, y, color='r', linestyle=':', marker='*', label='Profit / Risk')
    plt.legend()
    plt.xlabel('Risk')
    plt.ylabel('Profit')
    plt.plot([x[6], x[6]], [y[0], y[6]], color='g', linestyle='--')
    plt.scatter([x[6], ], [y[6], ], 50, color='red')
    plt.annotate('Risk={0}\nProfit={1}'.format(round(x[6], 4), round(y[6], 4)),
xy=(x[6], y[6]), xytext=(-20, 20),
                 textcoords='offset points', fontsize=12)
    plt.plot([x[25], x[25]], [y[0], y[25]], color='g', linestyle='--')
    plt.scatter([x[25], ], [y[25], ], 50, color='red')
    plt.annotate('Risk={0}\nProfit={1}'.format(round(x[25], 4), round(y[25],
4)), xy=(x[25], y[25]), xytext=(-20, -40),
                 textcoords='offset points', fontsize=12)
    plt.show()
if __name__ == '__main__':
    a = np.array([[0, 0.025, 0, 0, 0], [0, 0, 0.015, 0, 0],
                  [0, 0, 0, 0.055, 0], [0, 0, 0, 0, 0.026]])  # 不等式约束系数
    equal = [1, 1.01, 1.02, 1.045, 1.065] + [0] * a.shape[0]  # 等式约束系数
    c = [-0.05, -0.27, -0.19, -0.185, -0.185]  # 目标函数系数
    M = 1  # 总投资
    risk = 0.0 / M  # 风险率
    x_point, y_point = [], []  # 不同风险对应的收益点
    while risk < 0.05:
        b = [M] + [risk] * a.shape[0]  # M为等式约束，其余为不等式约束
        matrix, ret_matrix, X = solve(np.copy(a), deepcopy(b), deepcopy(c),
deepcopy(equal))  # 单纯形算法求解目标函数
        if ret_matrix is not None:
            print('等式约束检查', np.dot(equal, X[0:-1]))
            print('当风险率为{}时，本次迭代的最优解为：{}'.format(np.round(risk, 4),
np.round(X[0: len(c)], 4).tolist()))
            print('该线性规划的最优值是：{}'.format(np.round(ret_matrix[0][0], 4)))
            x_point.append(risk)
            y_point.append(ret_matrix[0][0])
        risk += 0.001
    draw_result(x_point, y_point)
```

4.3 对偶单纯形法

4.3.1 概念

> 对偶单纯形法根据对偶问题的对称性, 可以保持对偶问题基本解是可行解, 而原问题由不可行基本解开始, 通过迭代, 逐步达到基本可行解, 这样也使原问题与对偶问题都达到最优, 采用这种思想的优化方法称为对偶单纯形法。

4.3.2 适用范围

1. 对变量较少而约束很多的线性规划问题, 首先将其变为对偶问题,再用对偶单纯形法求解, 从而简化计算。(这是因为在单纯形法的计算中, 工作量的大小主要取决于约束条件的多少而不是变量的多少)
2. 此法用于**灵敏度分析**

   概念:

   灵敏度分析以上在讨论线性规划时, 均把 C_j, a_{ij}, b_i 看做常数，但实际上这些数据有的是测量值, 有的是经验值, 既非绝对准确, 也非稳定不变, 因此, 有必要分析数据发生波动时，对当前最优解和最优值有何影响, 这就是所谓的灵敏度分析。

   灵敏度分析通常有两类问题:

   一是当 C、A、b 中某部分数据发生给定的变化时，讨论最优解和最优值如何变化;

   二是分析 C、A、 b 中数据在多大范围内波动, 原最优基仍保持不变, 同时最优解和最优值如何变化。

4.3.2 步骤

4.3.3. 实例

4.3.4 代码实现 🐞

简述：**COPT的Python接口**，手动设置模型，手动设置求解时间，**对偶单纯形法**求解**LP**问题

代码：

[lp.py](lp.py)

```python
"""
The problem to solve:

    Maximize:
        1.2x + 1.8y + 2.1z

    Subject to:
        1.5x + 1.2y + 1.8z <= 2.6
        0.8x + 0.6y + 0.9z >= 1.2

    where:
        0.1 <= x <= 0.6
        0.2 <= y <= 1.5
        0.3 <= z <= 2.8
"""

# Import the Python interface library
from coptpy import *

# Create COPT environment
env = Envr()

# Create COPT model
model = env.createModel("lp_ex1")
```

```
# Add variables: x, y, z
x = model.addVar(lb=0.1, ub=0.6, name="x")
y = model.addVar(lb=0.2, ub=1.5, name="y")
z = model.addVar(lb=0.3, ub=2.8, name="z")

# Add constraints
model.addConstr(1.5*x + 1.2*y + 1.8*z <= 2.6)
model.addConstr(0.8*x + 0.6*y + 0.9*z >= 1.2)

# Set objective function
model.setObjective(1.2*x + 1.8*y + 2.1*z, sense=COPT.MAXIMIZE)

# Set parameter
model.setParam(COPT.Param.TimeLimit, 10.0)

# Solve the model
model.solve()

# Analyze solution
if model.status == COPT.OPTIMAL:
    # Get objective value
    print("Objective value: {}".format(model.objval))
    allvars = model.getVars()

    # Get valiable solution
    print("Variable solution:")
    for var in allvars:
        print(" x[{0}]: {1}".format(var.index, var.x))

    # Get variable basis status
    print("Variable basis status:")
    for var in allvars:
        print(" x[{0}]: {1}".format(var.index, var.basis))

# Write MPS model, solution, basis and parameters files
model.write("lp_ex1.mps")
model.write("lp_ex1.sol")
model.write("lp_ex1.bas")
model.write("lp_ex1.par")
```

结果:

对偶单纯形法

4.4 内点法(Interior Point Method)

4.4.1 概念

> 1984 年 Karmarkar 提出了一个比 Khanchian 法好的多项式时间算法的内点法, 称为 Karmarkar 法。由于该法引用了非线性规划中的牛顿投影, 因此又称 Karmarkar 投影法。Karmarkar 法的提出在线性规划领域具有极大的理论意义。与椭球法不同, 这个新算法不仅在最坏情况下在时间复杂度上优于单纯形法, 在大型实际问题中也有潜力与单纯形法竞争。这一方法的提出掀起了一股内点法的研究热潮。鉴于 Karmarkar 法的难读性, 一些研究学者对 Karmarkar 法进行了适度的修改, 使其简便易读。然而直接用该方法编程解题的测试表明, 与目前基于单纯形法的商用软件相比, 并没有明显的优势。

4.4.2 步骤

4.4.3 实例

4.4.4 代码实现

简述：**COPT的Python接口**，从**MPS模型文件**或**LP格式模型文件**中读取，手动设置求解时间，**内点法**求解**LP**问题，并将求解过程输入求解日志

代码：

lp_in.py

```python
'''
The problem is read from:
    /Applications/copt30/examples/python/lp_ex1.mps
'''

# Import the Python interface library
from coptpy import *

# Create COPT environment
env = Envr()

# Create COPT model
model = env.createModel("lp_ex1")

# Read MPS model
model.read("lp_ex1.mps")

# Set log file
m.setLogFile("lp_ex1.log")

# Set parameter
m.setParam(COPT.Param.TimeLimit, 10.0)

# Set solving methos to Barrier
m.setParam(COPT.Param.LpMethod, 2)

# Solve the model
model.solve()

# Analyze solution
if model.status == COPT.OPTIMAL:
    # Get objective value
    print("Objective value: {}".format(model.objval))
    allvars = model.getVars()

    # Get valiable solution
    print("Variable solution:")
    for var in allvars:
        print(" x[{0}]: {1}".format(var.index, var.x))

    # Get variable basis status
    print("Variable basis status:")
    for var in allvars:
        print(" x[{0}]: {1}".format(var.index, var.basis))

# Write MPS model, solution, basis and parameters files
```

```
model.write("lp_ex1.mps")
model.write("lp_ex1.sol")
model.write("lp_ex1.bas")
model.write("lp_ex1.par")
```

结果：

内点法1

内点法2

5. 补充资料

1. 数模官网 - 线性规划
2. qq小群 - 培训资料 - “15-焦玲玲-COPT求解器使用教程.pdf” - p56 ~ p71
3. 1216 - 每日推送
4. 单纯形法
5. 教材 运筹学----第三版 .pdf
6. GitHub----线性规划-单纯形算法
7. 矩阵单纯性法