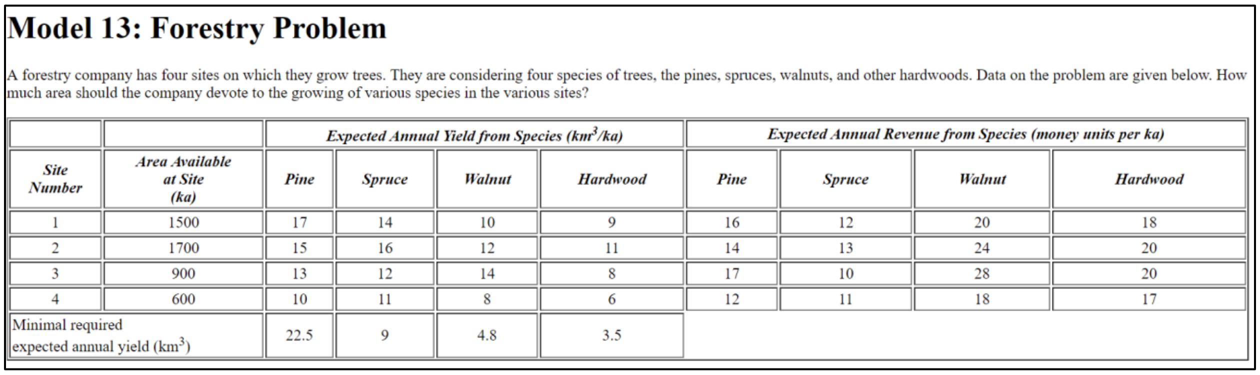
Linear Optimization Programming Project

Implementation of Simplex Method

Yung-Ching Chen

- **LP Problem: Model 13**



Model 13: Forestry Problem

A forestry company has four sites on which they grow trees. They are considering four species of trees, the pines, spruces, walnuts, and other hardwoods. Data on the problem are given below. How much area should the company devote to the growing of various species in the various sites?

		Expected Annual Yield from Species (km^3/ka)				*Expected Annual Revenue from Species (money units per ka)*			
Site Number	*Area Available at Site (ka)*	*Pine*	*Spruce*	*Walnut*	*Hardwood*	*Pine*	*Spruce*	*Walnut*	*Hardwood*
1	1500	17	14	10	9	16	12	20	18
2	1700	15	16	12	11	14	13	24	20
3	900	13	12	14	8	17	10	28	20
4	600	10	11	8	6	12	11	18	17
Minimal required expected annual yield (km^3)		22.5	9	4.8	3.5				

- **Programming Language: Python 3.6**
- **Commercial Solver: MATLAB**

I. Linear Programming Model

x_{ij}: number of ka grow in site i for type j of species; I = 1…4, j = 1…4 which represents pine, spruce, walnut, hardwood respectively

Maximize $16x_{11} + 12x_{12} + 20x_{13} + 18x_{14} + 14x_{21} + 13x_{22} + 24x_{23} + 20x_{24} + 17x_{31} + 10x_{32} + 28x_{33} + 20x_{34} + 12x_{41} + 11x_{42} + 18x_{43} + 17x_{44}$

s.t.

$$17x_{11} + 14x_{12} + 10x_{13} + 9x_{14} \leq 1500$$

$$15x_{21} + 16x_{22} + 12x_{23} + 11x_{24} \leq 1700$$

$$13x_{31} + 12x_{32} + 14x_{33} + 8x_{34} \leq 900$$

$$10x_{41} + 11x_{42} + 8x_{43} + 6x_{44} \leq 600$$

$$17x_{11} + 15x_{21} + 13x_{31} + 10x_{41} \geq 22.5$$

$$14x_{12} + 16x_{22} + 12x_{32} + 11x_{42} \geq 9$$

$$10x_{13} + 12x_{23} + 14x_{33} + 8x_{43} \geq 4.8$$

$$9x_{14} + 11x_{24} + 8x_{34} + 6x_{44} \geq 3.5$$

$$x_{ij} \geq 0$$

Then I transformed this LP problem into the ***standard form*** as following, which s_1 ~ s_8 are slack variables and a_1 ~ a_4 are artificial variables.

max $16x_{11} + 12x_{12} + 20x_{13} + 18x_{14} + 14x_{21} + 13x_{22} + 24x_{23} + 20x_{24} + 17x_{31} + 10x_{32} + 28x_{33} + 20x_{34} + 12x_{41} + 11x_{42} + 18x_{43} + 17x_{44}$

s.t.

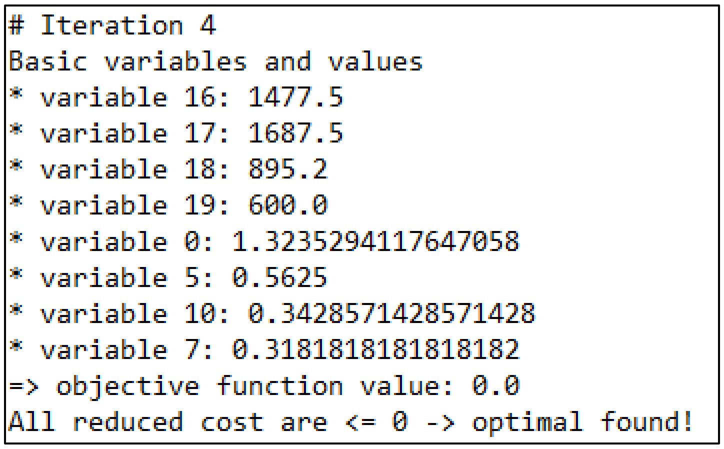
$$\begin{aligned}
&17x_{11} + 14x_{12} + 10x_{13} + 9x_{14} + s_1 &&= 1500\\
&15x_{21} + 16x_{22} + 12x_{23} + 11x_{24} + s_2 &&= 1700\\
&13x_{31} + 12x_{32} + 14x_{33} + 8x_{34} + s_3 &&= 900\\
&10x_{41} + 11x_{42} + 8x_{43} + 6x_{44} + s_4 &&= 600\\
&17x_{11} + 15x_{21} + 13x_{31} + 10x_{41} - s_5 + a_1 &&= 22.5\\
&14x_{12} + 16x_{22} + 12x_{32} + 11x_{42} - s_6 + a_2 &&= 9\\
&10x_{13} + 12x_{23} + 14x_{33} + 8x_{43} - s_7 + a_3 &&= 4.8\\
&9x_{14} + 11x_{24} + 8x_{34} + 6x_{44} - s_8 + a_4 &&= 3.5\\
&x_{ij} \geq 0
\end{aligned}$$

II. Python Code

1. Details of 6 requirements

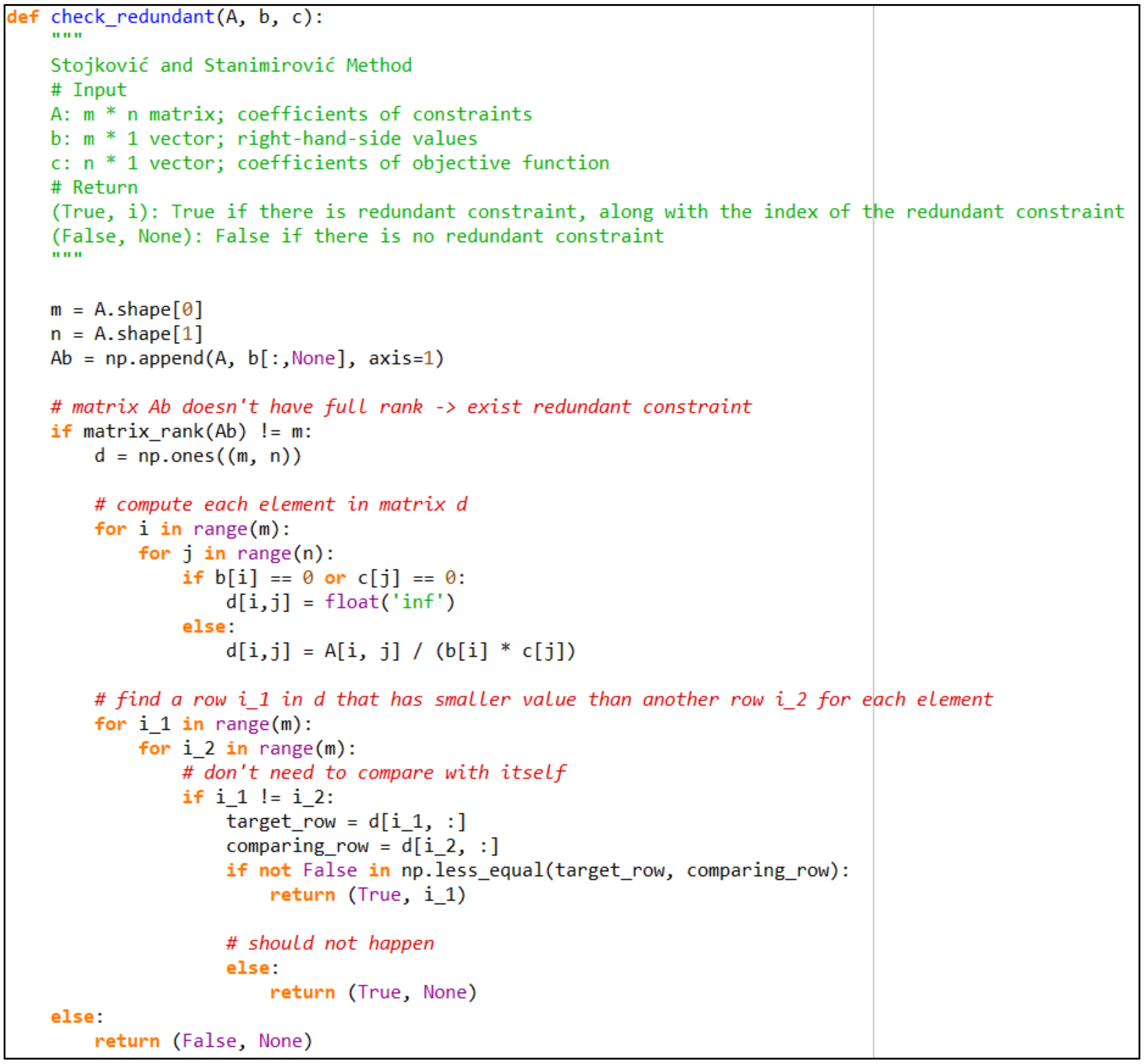
(1) LP need to be standard form
I transformed my LP problem into standard form manually. You can find that in part I. Linear Programming Model.

(2) Detect LP is feasible or not
I used **2-phase method** to detect the feasibility of the LP problem. If the objective function value of phase-one problem is equal to 0, then the original problem is feasible. From the final iteration result of the phase-one problem of model 13, since it has optimal objective function value that equals to 0, we can say that the original problem is feasible.


```
# Iteration 4
Basic variables and values
* variable 16: 1477.5
* variable 17: 1687.5
* variable 18: 895.2
* variable 19: 600.0
* variable 0: 1.3235294117647058
* variable 5: 0.5625
* variable 10: 0.3428571428571428
* variable 7: 0.3181818181818182
=> objective function value: 0.0
All reduced cost are <= 0 -> optimal found!
```

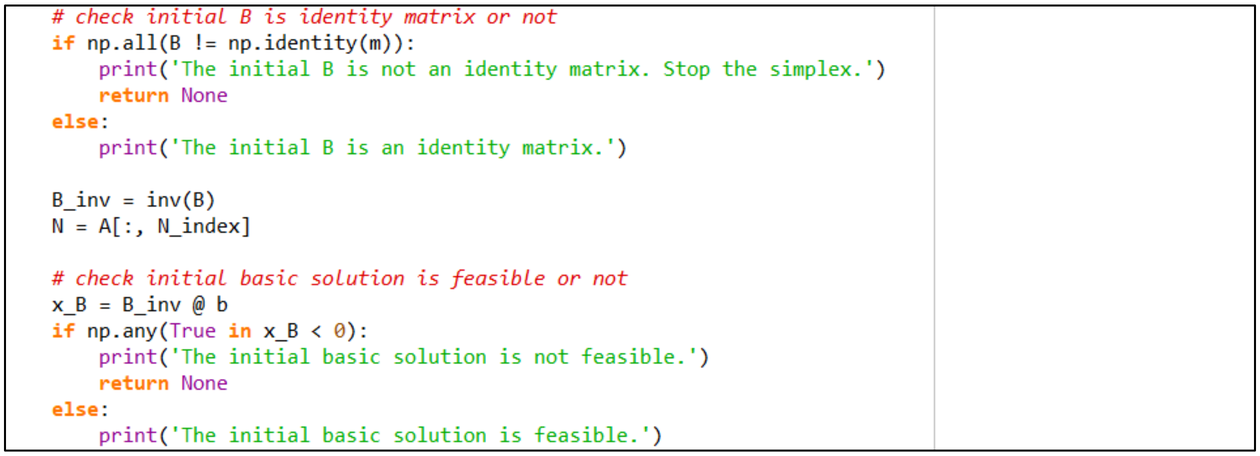
(3) Detect full rank and redundant constraint

In my code there is a function called ***check_redundant(A, b, c)***, which is used to check whether the matrix of constraints' coefficients has full rank or not. If it doesn't contain full rank, the function can also tell and return which constraint is redundant by using ***Stojković and Stanimirović method***.



```python
def check_redundant(A, b, c):
    """
    Stojković and Stanimirović Method
    # Input
    A: m * n matrix; coefficients of constraints
    b: m * 1 vector; right-hand-side values
    c: n * 1 vector; coefficients of objective function
    # Return
    (True, i): True if there is redundant constraint, along with the index of the redundant constraint
    (False, None): False if there is no redundant constraint
    """

    m = A.shape[0]
    n = A.shape[1]
    Ab = np.append(A, b[:,None], axis=1)

    # matrix Ab doesn't have full rank -> exist redundant constraint
    if matrix_rank(Ab) != m:
        d = np.ones((m, n))

        # compute each element in matrix d
        for i in range(m):
            for j in range(n):
                if b[i] == 0 or c[j] == 0:
                    d[i,j] = float('inf')
                else:
                    d[i,j] = A[i, j] / (b[i] * c[j])

        # find a row i_1 in d that has smaller value than another row i_2 for each element
        for i_1 in range(m):
            for i_2 in range(m):
                # don't need to compare with itself
                if i_1 != i_2:
                    target_row = d[i_1, :]
                    comparing_row = d[i_2, :]
                    if not False in np.less_equal(target_row, comparing_row):
                        return (True, i_1)

                    # should not happen
                    else:
                        return (True, None)
    else:
        return (False, None)
```

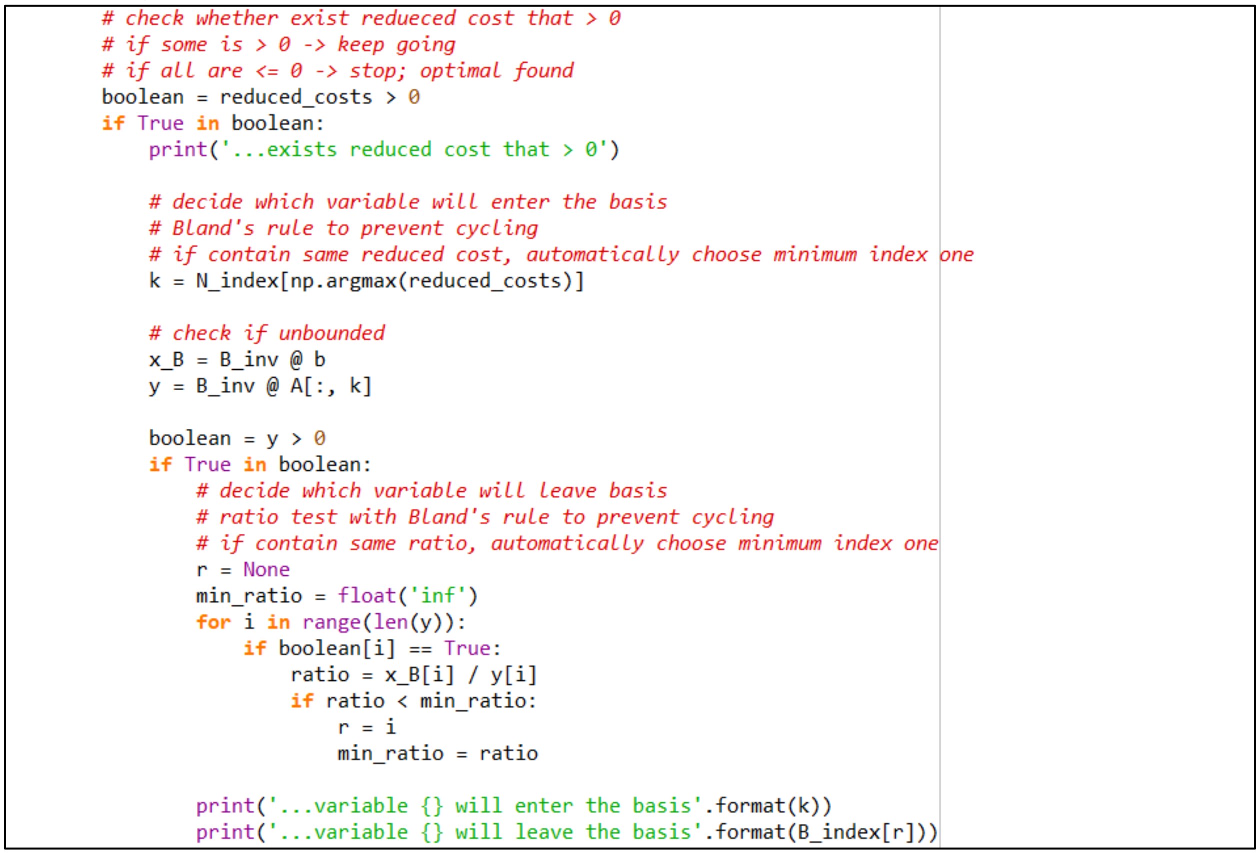
(4) Detect that simplex starts with an identity matrix and BFS



```
    # check initial B is identity matrix or not
    if np.all(B != np.identity(m)):
        print('The initial B is not an identity matrix. Stop the simplex.')
        return None
    else:
        print('The initial B is an identity matrix.')

    B_inv = inv(B)
    N = A[:, N_index]

    # check initial basic solution is feasible or not
    x_B = B_inv @ b
    if np.any(True in x_B < 0):
        print('The initial basic solution is not feasible.')
        return None
    else:
        print('The initial basic solution is feasible.')
```

(5) Prevent cycling

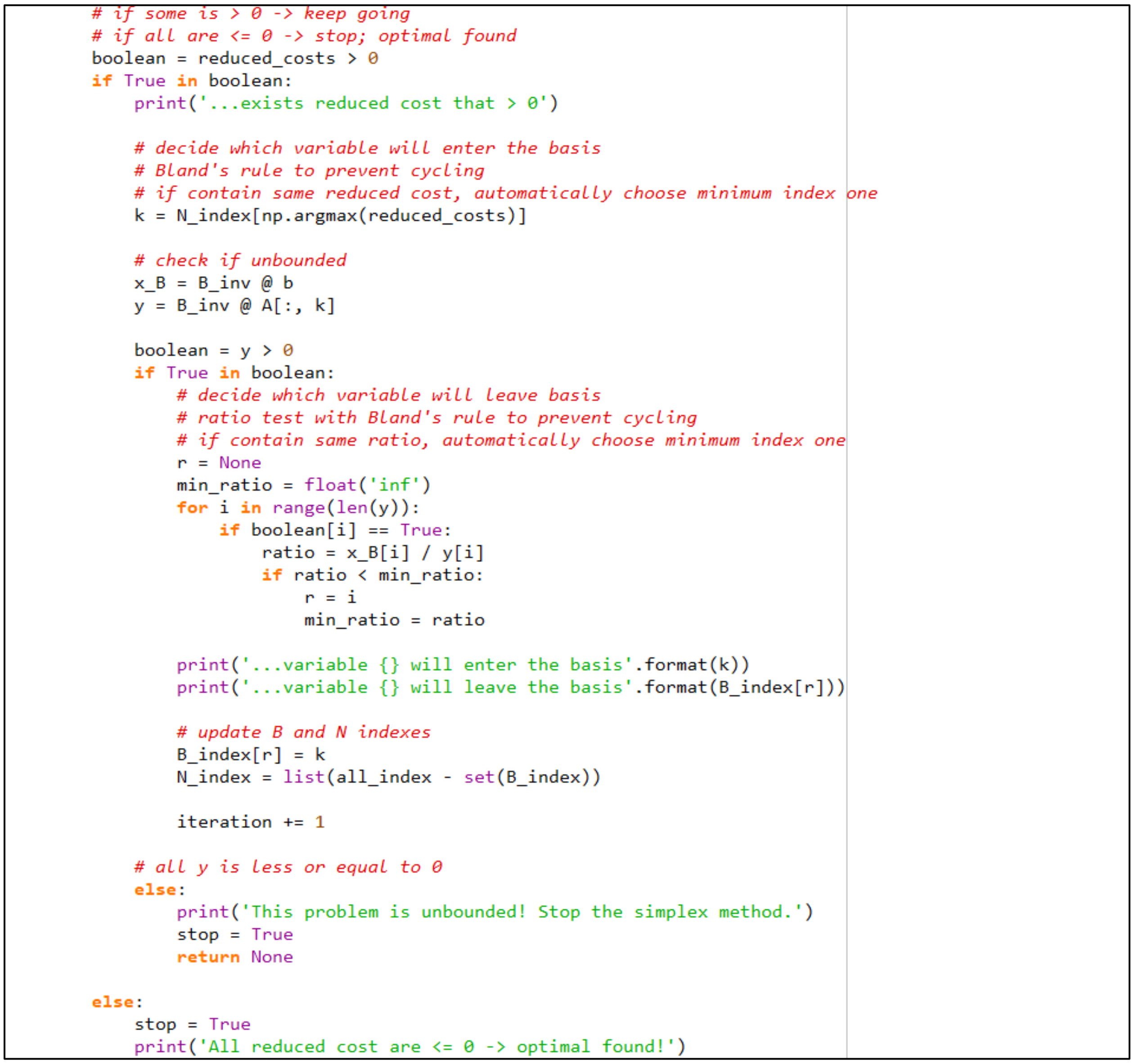
I used ***Bland's rule*** to prevent cycling. That is, if there are same maximum reduced costs, always choose the one with minimum index to enter the basis. Similarly, if there are same minimum ratio, always choose the one with minimum index to leave the basis.



```
        # check whether exist redueced cost that > 0
        # if some is > 0 -> keep going
        # if all are <= 0 -> stop; optimal found
        boolean = reduced_costs > 0
        if True in boolean:
            print('...exists reduced cost that > 0')

            # decide which variable will enter the basis
            # Bland's rule to prevent cycling
            # if contain same reduced cost, automatically choose minimum index one
            k = N_index[np.argmax(reduced_costs)]

            # check if unbounded
            x_B = B_inv @ b
            y = B_inv @ A[:, k]

            boolean = y > 0
            if True in boolean:
                # decide which variable will leave basis
                # ratio test with Bland's rule to prevent cycling
                # if contain same ratio, automatically choose minimum index one
                r = None
                min_ratio = float('inf')
                for i in range(len(y)):
                    if boolean[i] == True:
                        ratio = x_B[i] / y[i]
                        if ratio < min_ratio:
                            r = i
                            min_ratio = ratio

                print('...variable {} will enter the basis'.format(k))
                print('...variable {} will leave the basis'.format(B_index[r]))
```

(6) Detect termination

There are two possible termination scenarios. One is that if all the reduced cost are less or equal to 0, then the optimal solution is found. The other one is that if all y is less or equal to 0, then the LP is unbounded.



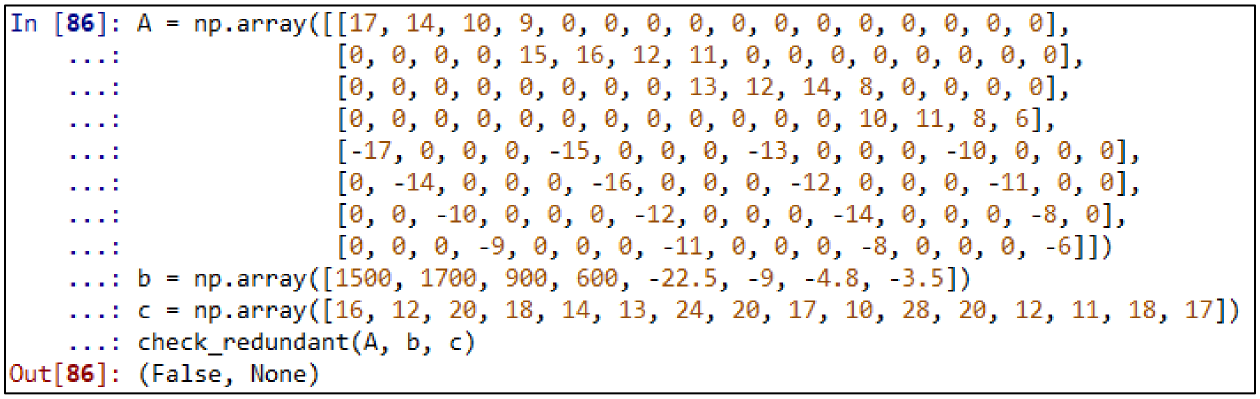
```python
        # if some is > 0 -> keep going
        # if all are <= 0 -> stop; optimal found
        boolean = reduced_costs > 0
        if True in boolean:
            print('...exists reduced cost that > 0')

            # decide which variable will enter the basis
            # Bland's rule to prevent cycling
            # if contain same reduced cost, automatically choose minimum index one
            k = N_index[np.argmax(reduced_costs)]

            # check if unbounded
            x_B = B_inv @ b
            y = B_inv @ A[:, k]

            boolean = y > 0
            if True in boolean:
                # decide which variable will leave basis
                # ratio test with Bland's rule to prevent cycling
                # if contain same ratio, automatically choose minimum index one
                r = None
                min_ratio = float('inf')
                for i in range(len(y)):
                    if boolean[i] == True:
                        ratio = x_B[i] / y[i]
                        if ratio < min_ratio:
                            r = i
                            min_ratio = ratio

                print('...variable {} will enter the basis'.format(k))
                print('...variable {} will leave the basis'.format(B_index[r]))

                # update B and N indexes
                B_index[r] = k
                N_index = list(all_index - set(B_index))

                iteration += 1

            # all y is less or equal to 0
            else:
                print('This problem is unbounded! Stop the simplex method.')
                stop = True
                return None

        else:
            stop = True
            print('All reduced cost are <= 0 -> optimal found!')
```

2. Complete Code

```python
import numpy as np
from numpy.linalg import inv
from numpy.linalg import matrix_rank


def check_redundant(A, b, c):
    """
    Stojković and Stanimirović Method
    # Input
    A: m * n matrix; coefficients of constraints
    b: m * 1 vector; right-hand-side values
    c: n * 1 vector; coefficients of objective function
    # Return
    (True, i): True if there is redundant constraint, along with the index of the redundant
constraint
    (False, None): False if there is no redundant constraint
    """

    m = A.shape[0]
    n = A.shape[1]
    Ab = np.append(A, b[:,None], axis=1)

    # matrix Ab doesn't have full rank -> exist redundant constraint
    if matrix_rank(Ab) != m:
        d = np.ones((m, n))

        # compute each element in matrix d
        for i in range(m):
            for j in range(n):
                if b[i] == 0 or c[j] == 0:
                    d[i,j] = float('inf')
                else:
                    d[i,j] = A[i, j] / (b[i] * c[j])

        # find a row i_1 in d that has smaller value than another row i_2 for each element
        for i_1 in range(m):
            for i_2 in range(m):
                # don't need to compare with itself
                if i_1 != i_2:
                    target_row = d[i_1, :]
                    comparing_row = d[i_2, :]
                    if not False in np.less_equal(target_row, comparing_row):
                        return (True, i_1)

                    # should not happen
                    else:
                        return (True, None)
    else:
        return (False, None)


def simplex_method(A, b, c, B_index, N_index):
    """
    # Input
    A: m * n matrix; coefficients of constraints
    b: m * 1 vector; right-hand-side values
    c: n * 1 vector; coefficients of objective function
    B_index: list of m elements; initial basic variable indexes
    N_index: list of (n-m) elements; initial non-basic variable indexes

    # Return
    B_index: list of m elements; final indexes of optimal variables
    x_B: m * 1 vector; the optimal basic variables values
    objFunValue: scalar; optimal objective function value
    final_table: m * n matrix; final table in tableau

    """

    all_index = set(B_index + N_index)

    # initial c_B
    c_B = np.array([c[i] for i in B_index])
    # initial B
    B = A[:, B_index]

    print('### Checking before simplex method')

    m = A.shape[0]
    n = A.shape[1]

    # check initial B is identity matrix or not
    if np.all(B != np.identity(m)):
        print('The initial B is not an identity matrix. Stop the simplex.')
        return None
    else:
        print('The initial B is an identity matrix.')

    B_inv = inv(B)
    N = A[:, N_index]

    # check initial basic solution is feasible or not
    x_B = B_inv @ b
    if np.any(True in x_B < 0):
        print('The initial basic solution is not feasible.')
        return None
    else:
        print('The initial basic solution is feasible.')

    # initial objective function value
    objFunValue = np.transpose(c_B) @ B_inv @ b


    # everything is ready -> start the simplex method
    print('\n### Start the simplex method')
    stop = False
    iteration = 0

    while stop == False:

        c_B = np.array([c[i] for i in B_index])
        c_N = np.array([c[i] for i in N_index])

        B = A[:, B_index]
        B_inv = inv(B)
        N = A[:, N_index]

        x_B = B_inv @ b
        objFunValue = np.transpose(c_B) @ B_inv @ b

        print('\n# Iteration {}'.format(iteration))
        print('Basic variables and values')
        for i in range(len(B_index)):
            print('* variable {}: {}'.format(B_index[i], x_B[i]))
        print('=> objective function value: {}'.format(objFunValue))

        # calcuate reduced costs for all nonbasic variables
        reduced_costs = np.transpose(c_B) @ B_inv @ N  - c_N

        # check whether exist redueced cost that > 0
        # if some is > 0 -> keep going
        # if all are <= 0 -> stop; optimal found
        boolean = reduced_costs > 0
        if True in boolean:
            print('...exists reduced cost that > 0')

            # decide which variable will enter the basis
            # Bland's rule to prevent cycling
            # if contain same reduced cost, automatically choose minimum index one
            k = N_index[np.argmax(reduced_costs)]

            # check if unbounded
            x_B = B_inv @ b
            y = B_inv @ A[:, k]

            boolean = y > 0
            if True in boolean:
                # decide which variable will leave basis
                # ratio test with Bland's rule to prevent cycling
                # if contain same ratio, automatically choose minimum index one
                r = None
                min_ratio = float('inf')
                for i in range(len(y)):
                    if boolean[i] == True:
                        ratio = x_B[i] / y[i]
                        if ratio < min_ratio:
                            r = i
                            min_ratio = ratio

                print('...variable {} will enter the basis'.format(k))
                print('...variable {} will leave the basis'.format(B_index[r]))

                # update B and N indexes
                B_index[r] = k
                N_index = list(all_index - set(B_index))

                iteration += 1

            # all y is less or equal to 0
            else:
                print('This problem is unbounded! Stop the simplex method.')
                stop = True
                return None

        else:
            stop = True
            print('All reduced cost are <= 0 -> optimal found!')

    final_table = np.ones((m, n))
    final_table[:, B_index] = np.identity(m)
    final_table[:, N_index] = B_inv @ A[:, N_index]

    return (B_index, x_B, objFunValue, final_table)


# =========================================================
# linear programming model 13
# https://sites.math.washington.edu/~burke/crs/407/models/m13.html

# check redundant constraints in canonical form
A = np.array([[17, 14, 10, 9, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
              [0, 0, 0, 0, 15, 16, 12, 11, 0, 0, 0, 0, 0, 0, 0, 0],
              [0, 0, 0, 0, 0, 0, 0, 0, 13, 12, 14, 8, 0, 0, 0, 0],
              [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 10, 11, 8, 6],
              [-17, 0, 0, 0, -15, 0, 0, 0, -13, 0, 0, 0, -10, 0, 0, 0],
              [0, -14, 0, 0, 0, -16, 0, 0, 0, -12, 0, 0, 0, -11, 0, 0],
              [0, 0, -10, 0, 0, 0, -12, 0, 0, 0, -14, 0, 0, 0, -8, 0],
              [0, 0, 0, -9, 0, 0, 0, -11, 0, 0, 0, -8, 0, 0, 0, -6]])
b = np.array([1500, 1700, 900, 600, -22.5, -9, -4.8, -3.5])
c = np.array([16, 12, 20, 18, 14, 13, 24, 20, 17, 10, 28, 20, 12, 11, 18, 17])
check_redundant(A, b, c)


# phase (I)
# check feasibility
# if objective function value == 0 -> LP problem is feasible
A = np.array([[17, 14, 10, 9, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 
0, 0, 0],
              [0, 0, 0, 0, 15, 16, 12, 11,0, 0, 0, 0, 0, 0, 0, 0,  0, 1, 0, 0, 0, 0, 0, 0, 
0, 0, 0, 0],
              [0, 0, 0, 0, 0, 0, 0, 0, 13, 12, 14, 8, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 
0, 0, 0],
              [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 10, 11, 8, 6, 0, 0, 0, 1, 0, 0, 0, 0, 0, 
0, 0, 0],
              [17, 0, 0, 0, 15, 0, 0, 0, 13, 0, 0, 0, 10, 0, 0, 0, 0, 0, 0, 0, -1, 0, 0, 0, 
1, 0, 0, 0],
              [0, 14, 0, 0, 0, 16, 0, 0, 0, 12, 0, 0, 0, 11, 0, 0, 0, 0, 0, 0, 0, -1, 0, 0, 
0, 1, 0, 0],
              [0, 0, 10, 0, 0, 0, 12, 0, 0, 0, 14, 0, 0, 0, 8, 0, 0, 0, 0, 0, 0, 0, -1, 0, 
0 ,0, 1, 0],
              [0, 0, 0, 9, 0, 0, 0, 11, 0, 0,0, 8, 0, 0, 0, 6, 0, 0, 0, 0, 0, 0, 0, -1, 
0 ,0 ,0, 1]])
b = np.array([1500, 1700, 900, 600, 22.5, 9, 4.8, 3.5])
B_index = [16, 17, 18, 19, 24, 25, 26, 27]
N_index = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 20, 21, 22, 23]
c_1 = np.array([0]*24 + [1]*4)
B_index, x_B, objFunValue, final_table = simplex_method(A, b, c_1, B_index, N_index)


if objFunValue == 0:
    print('Objective function value is equal to 0 -> The original problem is feasible.')
else:
    print('Objective function value is not equal to 0 -> The original problem is not 
feasible.')


# phase (II)
A = final_table[:, 0:24]
b = x_B
c_2 = np.array([-16, -12, -20, -18, -14, -13, -24, -20, -17, -10, -28, -20, -12, -11, -18, -
17] + [0]*8)
N_index = list(set(range(24)) - set(B_index))
B_index_2, x_B_2, objFunValue_2, final_table_2 = simplex_method(A, b, c_2, B_index, N_index)
```

III. Results of My Own Code

1. Check whether the model has redundant constraints or not. If the function return ***False***, it means that the matrix has full rank and this LP problem doesn't have redundant constraint.



```
In [86]: A = np.array([[17, 14, 10, 9, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    ...:               [0, 0, 0, 0, 15, 16, 12, 11, 0, 0, 0, 0, 0, 0, 0, 0],
    ...:               [0, 0, 0, 0, 0, 0, 0, 0, 13, 12, 14, 8, 0, 0, 0, 0],
    ...:               [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 10, 11, 8, 6],
    ...:               [-17, 0, 0, 0, -15, 0, 0, 0, -13, 0, 0, 0, -10, 0, 0, 0],
    ...:               [0, -14, 0, 0, 0, -16, 0, 0, 0, -12, 0, 0, 0, -11, 0, 0],
    ...:               [0, 0, -10, 0, 0, 0, -12, 0, 0, 0, -14, 0, 0, 0, -8, 0],
    ...:               [0, 0, 0, -9, 0, 0, 0, -11, 0, 0, 0, -8, 0, 0, 0, -6]])
    ...: b = np.array([1500, 1700, 900, 600, -22.5, -9, -4.8, -3.5])
    ...: c = np.array([16, 12, 20, 18, 14, 13, 24, 20, 17, 10, 28, 20, 12, 11, 18, 17])
    ...: check_redundant(A, b, c)
Out[86]: (False, None)
```

2. Phase (I)

```
### Checking before simplex method
The initial B is an identity matrix.
The initial basic solution is feasible.

### Start the simplex method

# Iteration 0
Basic variables and values
* variable 16: 1500.0
* variable 17: 1700.0
* variable 18: 900.0
* variable 19: 600.0
* variable 24: 22.5
* variable 25: 9.0
* variable 26: 4.8
* variable 27: 3.5
=> objective function value: 39.8
...exists reduced cost that > 0
...variable 0 will enter the basis
...variable 24 will leave the basis

# Iteration 1
Basic variables and values
* variable 16: 1477.5
* variable 17: 1700.0
* variable 18: 900.0
* variable 19: 600.0
* variable 0: 1.3235294117647058
* variable 25: 9.0
* variable 26: 4.8
* variable 27: 3.5
=> objective function value: 17.3
...exists reduced cost that > 0
...variable 5 will enter the basis
...variable 25 will leave the basis
```

```
# Iteration 2
Basic variables and values
* variable 16: 1477.5
* variable 17: 1691.0
* variable 18: 900.0
* variable 19: 600.0
* variable 0: 1.3235294117647058
* variable 5: 0.5625
* variable 26: 4.8
* variable 27: 3.5
=> objective function value: 8.3
...exists reduced cost that > 0
...variable 10 will enter the basis
...variable 26 will leave the basis

# Iteration 3
Basic variables and values
* variable 16: 1477.5
* variable 17: 1691.0
* variable 18: 895.2
* variable 19: 600.0
* variable 0: 1.3235294117647058
* variable 5: 0.5625
* variable 10: 0.3428571428571428
* variable 27: 3.5
=> objective function value: 3.5
...exists reduced cost that > 0
...variable 7 will enter the basis
...variable 27 will leave the basis

# Iteration 4
Basic variables and values
* variable 16: 1477.5
* variable 17: 1687.5
* variable 18: 895.2
* variable 19: 600.0
* variable 0: 1.3235294117647058
* variable 5: 0.5625
* variable 10: 0.3428571428571428
* variable 7: 0.3181818181818182
=> objective function value: 0.0
All reduced cost are <= 0 -> optimal found!
```

3. Phase (II)

```
### Checking before simplex method
The initial B is an identity matrix.
The initial basic solution is feasible.

### Start the simplex method

# Iteration 0
Basic variables and values
* variable 16: 1477.5
* variable 17: 1687.5
* variable 18: 895.2
* variable 19: 600.0
* variable 0: 1.3235294117647058
* variable 5: 0.5625
* variable 10: 0.3428571428571428
* variable 7: 0.3181818181818182
=> objective function value: -44.45260695187166
...exists reduced cost that > 0
...variable 15 will enter the basis
...variable 7 will leave the basis

# Iteration 1
Basic variables and values
* variable 16: 1477.5
* variable 17: 1691.0
* variable 18: 895.2
* variable 19: 596.5
* variable 0: 1.3235294117647058
* variable 5: 0.5625
* variable 10: 0.3428571428571428
* variable 15: 0.5833333333333334
=> objective function value: -48.005637254901956
...exists reduced cost that > 0
...variable 8 will enter the basis
...variable 0 will leave the basis

# Iteration 2
Basic variables and values
* variable 16: 1500.0
* variable 17: 1691.0
* variable 18: 872.7
* variable 19: 596.5
* variable 8: 1.7307692307692308
* variable 5: 0.5625
* variable 10: 0.3428571428571428
* variable 15: 0.5833333333333334
=> objective function value: -56.252243589743586
...exists reduced cost that > 0
...variable 23 will enter the basis
...variable 19 will leave the basis
```

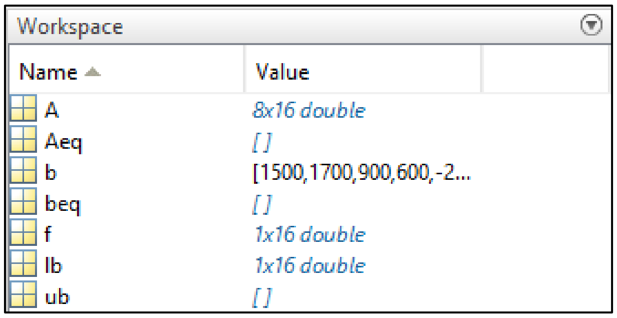
```
# Iteration 3
Basic variables and values
* variable 16: 1500.0
* variable 17: 1691.0
* variable 18: 872.7
* variable 23: 596.4999999999999
* variable 8: 1.7307692307692308
* variable 5: 0.5625
* variable 10: 0.3428571428571428
* variable 15: 100.0
=> objective function value: -1746.3355769230766
...exists reduced cost that > 0
...variable 7 will enter the basis
...variable 17 will leave the basis

# Iteration 4
Basic variables and values
* variable 16: 1500.0
* variable 7: 153.72727272727272
* variable 18: 872.7
* variable 23: 2287.5
* variable 8: 1.7307692307692308
* variable 5: 0.5625
* variable 10: 0.3428571428571428
* variable 15: 100.0
=> objective function value: -4820.881031468532
...exists reduced cost that > 0
...variable 1 will enter the basis
...variable 5 will leave the basis

# Iteration 5
Basic variables and values
* variable 16: 1491.0
* variable 7: 154.54545454545453
* variable 18: 872.7
* variable 23: 2296.5
* variable 8: 1.7307692307692308
* variable 1: 0.6428571428571428
* variable 10: 0.3428571428571428
* variable 15: 100.0
=> objective function value: -4837.646453546454
...exists reduced cost that > 0
...variable 11 will enter the basis
...variable 18 will leave the basis

# Iteration 6
Basic variables and values
* variable 16: 1491.0
* variable 7: 154.54545454545453
* variable 11: 109.0875
* variable 23: 3169.2
* variable 8: 1.7307692307692308
* variable 1: 0.6428571428571428
* variable 10: 0.3428571428571428
* variable 15: 100.0
=> objective function value: -7019.396453546455
...exists reduced cost that > 0
...variable 0 will enter the basis
...variable 8 will leave the basis

# Iteration 7
Basic variables and values
* variable 16: 1468.5
* variable 7: 154.54545454545453
* variable 11: 111.9
* variable 23: 3191.7
* variable 0: 1.3235294117647058
* variable 1: 0.6428571428571428
* variable 10: 0.3428571428571428
* variable 15: 100.0
=> objective function value: -7067.399847211613
...exists reduced cost that > 0
...variable 2 will enter the basis
...variable 10 will leave the basis

# Iteration 8
Basic variables and values
* variable 16: 1463.7
* variable 7: 154.54545454545453
* variable 11: 112.5
* variable 23: 3196.5
* variable 0: 1.3235294117647058
* variable 1: 0.6428571428571428
* variable 2: 0.48
* variable 15: 100.0
=> objective function value: -7079.399847211613
...exists reduced cost that > 0
...variable 3 will enter the basis
...variable 16 will leave the basis

# Iteration 9
Basic variables and values
* variable 3: 162.63333333333333
* variable 7: 154.54545454545453
* variable 11: 112.5
* variable 23: 4660.2
* variable 0: 1.3235294117647058
* variable 1: 0.6428571428571428
* variable 2: 0.48
* variable 15: 100.0
=> objective function value: -10006.799847211612
...exists reduced cost that > 0
...variable 4 will enter the basis
...variable 0 will leave the basis

# Iteration 10
Basic variables and values
* variable 3: 165.13333333333333
* variable 7: 152.5
* variable 11: 112.5
* variable 23: 4660.2
* variable 4: 1.5
* variable 1: 0.6428571428571428
* variable 2: 0.48
* variable 15: 100.0
=> objective function value: -10010.714285714286
...exists reduced cost that > 0
...variable 5 will enter the basis
...variable 1 will leave the basis

# Iteration 11
Basic variables and values
* variable 3: 166.13333333333333
* variable 7: 151.6818181818182
* variable 11: 112.5
* variable 23: 4660.2
* variable 4: 1.5
* variable 5: 0.5625
* variable 2: 0.48
* variable 15: 100.0
=> objective function value: -10011.948863636364
...exists reduced cost that > 0
...variable 6 will enter the basis
...variable 2 will leave the basis

# Iteration 12
Basic variables and values
* variable 3: 166.66666666666666
* variable 7: 151.24545454545458
* variable 11: 112.5
* variable 23: 4660.2
* variable 4: 1.5
* variable 5: 0.5625
* variable 6: 0.39999999999999997
* variable 15: 100.0
=> objective function value: -10012.82159090909
...exists reduced cost that > 0
```

```
...variable 22 will enter the basis
...variable 7 will leave the basis

# Iteration 13
Basic variables and values
* variable 3: 166.66666666666666
* variable 22: 1663.7
* variable 11: 112.49999999999999
* variable 23: 2996.5
* variable 4: 1.4999999999999999
* variable 5: 0.5625
* variable 6: 139.0416666666667
* variable 15: 100.0
=> objective function value: -10315.3125
...exists reduced cost that > 0
...variable 1 will enter the basis
...variable 5 will leave the basis

# Iteration 14
Basic variables and values
* variable 3: 165.66666666666666
* variable 22: 1672.7
* variable 11: 112.49999999999999
* variable 23: 2987.5
* variable 4: 1.4999999999999999
* variable 1: 0.6428571428571428
* variable 6: 139.7916666666667
* variable 15: 100.0
=> objective function value: -10315.714285714286
...exists reduced cost that > 0
...variable 0 will enter the basis
...variable 4 will leave the basis

# Iteration 15
Basic variables and values
* variable 3: 163.16666666666669
* variable 22: 1695.2
* variable 11: 112.49999999999999
* variable 23: 2965.0
* variable 0: 1.3235294117647058
* variable 1: 0.6428571428571238
* variable 6: 141.66666666666669
* variable 15: 100.0
=> objective function value: -10315.89075630252
...exists reduced cost that > 0
...variable 2 will enter the basis
...variable 3 will leave the basis

# Iteration 16
```

```
Basic variables and values
* variable 2: 146.85000000000002
* variable 22: 3163.7000000000007
* variable 11: 112.49999999999999
* variable 23: 1496.4999999999998
* variable 0: 1.3235294117647058
* variable 1: 0.6428571428571238
* variable 6: 141.66666666666669
* variable 15: 100.0
=> objective function value: -10315.89075630252
All reduced cost are <= 0 -> optimal found!
```

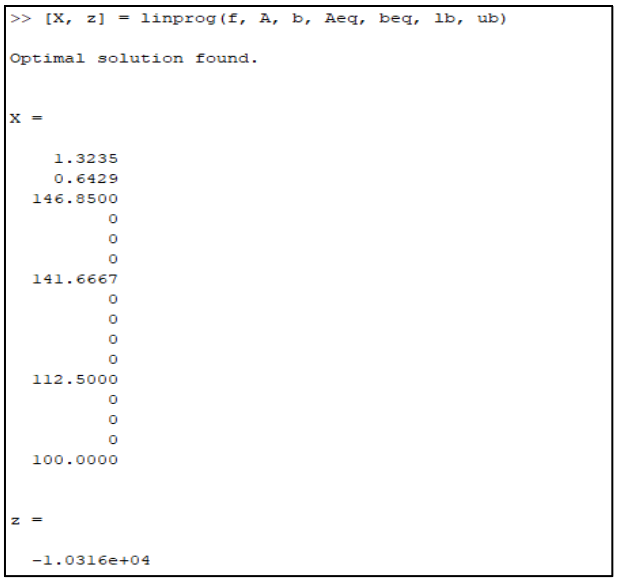
IV. Results of Commercial Solver (MATLAB)

Firstly, I manually typed the values into each matrix and vector. Be care that before typing, the LP need to be transformed into maximization problem and all the constraints need to be in less and equal form.



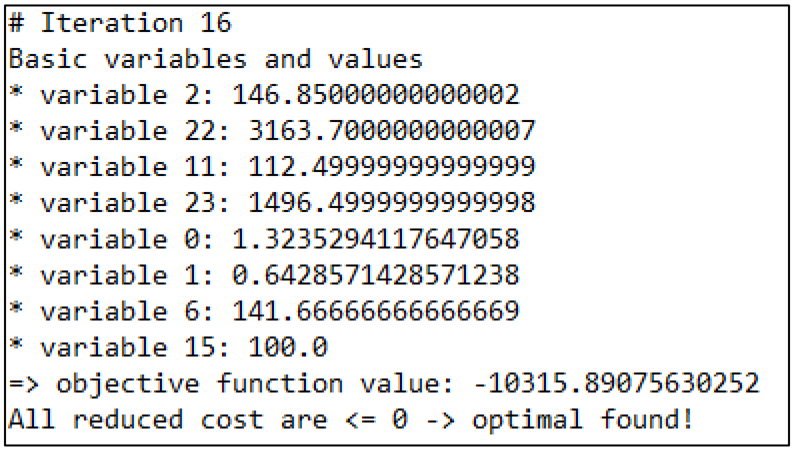
```
Workspace
Name        Value
A           8x16 double
Aeq         []
b           [1500,1700,900,600,-2...
beq         []
f           1x16 double
lb          1x16 double
ub          []
```

Then I ran the linear programming command and got the following result.



```
>> [X, z] = linprog(f, A, b, Aeq, beq, lb, ub)

Optimal solution found.


X =

    1.3235
    0.6429
  146.8500
         0
         0
         0
  141.6667
         0
         0
         0
         0
  112.5000
         0
         0
         0
  100.0000


z =

  -1.0316e+04
```

Comparing the above result with the below result of the final iteration from my own code, we can find that that basic variables and objective function value are the same. (As a reminder for you to compare the these two results, since the index starts from 0 in Python, variable 0 is actually the first variable in MATLAB, variable 1 is the second variable and so on.)



```
# Iteration 16
Basic variables and values
* variable 2: 146.85000000000002
* variable 22: 3163.7000000000007
* variable 11: 112.49999999999999
* variable 23: 1496.4999999999998
* variable 0: 1.3235294117647058
* variable 1: 0.6428571428571238
* variable 6: 141.66666666666669
* variable 15: 100.0
=> objective function value: -10315.89075630252
All reduced cost are <= 0 -> optimal found!
```