```python
############################################################################
# File Name: mySimplex.py                                                  #
# Author: Geonsik Yu, Purdue University, IE Dept                           #
#                                                                          #
# - Implement simplex method with the following requirements satisfied:    #
#                                                                          #
# Programming Requirements:                                                #
# (R1) Design a routine to do the conversion to the standard form LP.      #
# (R2) Design a routine to detect if the LP at hand is feasible or not.    #
#       (15 pt)                                                            #
# (R3) Design a routine to detect if the coefficient matrix associated with#
#     the standard form of the LP has full row rank; and if not, how to    #
#     remove the redundant constraints. (15 pt)                            #
# (R4) Design a routine to check if a basic feasible solution (BFS) with   #
#      the identity matrix as the corresponding basis is readily available. #
#      If yes, you can just go ahead to use the BFS to start your simplex   #
#      method;                                                             #
#      otherwise, use the methods we learned in class to start your simplex #
#      with an identity matrix as the initial basis. (15 pt)               #
# (R5) Your code needs to implement one rule to prevent the simplex method #
#      from cycling. (15 pt)                                               #
# (R6) Termination: your code needs to be able to handle both cases at     #
#      termination (6-1) a finite optimal solution or (6-2) unbounded(15 pt)#
############################################################################

from builtins import object
import numpy as np
from numpy import dot, zeros
from numpy.linalg import matrix_rank, norm
import sys

### Numpy printing parameters setting.
np.set_printoptions(precision = 4, linewidth = 200, threshold=sys.maxsize)
### Global threshold value for checking zero value with floating point errors.
THRESHOLD = 1e-8

class SimplexProblem(object):
    def __init__(self):
        ############################################################################
        # Constructor: This function constructs an LP problem object               #
        ############################################################################

        ## Attribute that will be initialized by setObjectiveDirection() function.
        self._objDirection = None     # Decides whether the given problem is maximization or minimization.
        self._objectiveValue = 0

        ## Attributes that will be initialized by setVariables() function.
        self._objCoeffs = None        # Coefficient values in the objective function for variables.
        self._varNames = None         # Names for variables.
        self._varLowerbounds = None   # Lowerbounds for variables.
        self._varCount = None         # Number of non-slack, non-artificial objCoeffs.
        self._reducedCosts = None     # Reduced costs

        # Attributes that will be initialized by addConstraint function.
        self._conNames = []         # List of names for all the constraints.
        self._AMatrix = []          # Coefficient Matrix "A"
        self._ineqDirs = []         #
        self._RHSs = []
        self._basisSet = []

        # Attributes for Two step method.
        self._twoStep = False
        self._2nd_objectiveValue = 0
        self._reducedSecondary = []   # Secondary reduced costs

    def debug(self):
        ############################################################################
        # This debug function prints out internal attributes.                      #
        ############################################################################
        print("# of original variables: ", self._varCount)
        print(self._varNames)
        print(self._reducedCosts)
        if self._twoStep == True:
            print(self._reducedSecondary)
        tmp = np.array(self._AMatrix)
        for idx in range(len(self._AMatrix)):
            print(tmp[idx], "||", self._RHSs[idx])
        print("Basis: ", self._basisSet)
        print("----------------------------------------")

    def setObjectiveDirection(self, Max = True):
        self._objDirection = Max
        ############################################################################
        # This function initializes the direction of optimization: If the problem  #
        # is on maximization, Max = True (Default), otherwise, Max = False.        #
        ############################################################################

    def setVariables(self, Names, ObjCoeffs, Lowerbounds=None):
        ############################################################################
        # This function initializes the decision variable set, their names, their  #
        # coefficients in the objective function, and their upper & lower bounds.  #
        ############################################################################
        self._objCoeffs = ObjCoeffs
        self._varNames = Names
        self._varLowerbounds = Lowerbounds

```

```
    def addConstraint(self, Name, rowVec, ineq_dir, RHS):
        ############################################################################
        # This function adds one constraint at a time. Each constraint requires to #
        # have coefficient vector with the same order with the variable vector, one#
        # inequality direction, and one right-hand side constant value.            #
        ############################################################################
        self._conNames.append(Name)
        self._AMatrix.append(rowVec)
        self._ineqDirs.append(ineq_dir)
        self._RHSs.append(RHS)

    def printCurrentStatus(self, Tableau):
        print("------------------------------------------------------------------")
        print("Current Basic Variables: ", ' '.join(self._basisSet) )
        print("Current Objective Value: ", repr(Tableau[0,-1]) )
        for idx in range(Tableau.shape[1]-1):
            value = 0.0
            if self._varNames[idx] in self._basisSet:
                value = Tableau[1+self._basisSet.index(self._varNames[idx]), -1]
            print(self._varNames[idx], " := ", repr(value))

    def setup(self):
        ############################################################################
        # Related to the requirement (R1).                                         #
        # (1) Transform the objective direction into "Minimization" and            #
        #     negate the signs of objective coefficients.                          #
        # (2) Transform the input constraint:                                      #
        #     2-1) [LHS <= RHS form]: pass.                                        #
        #     2-2) [LHS >= RHS form]: change to [-LHS <= -RHS].                    #
        #     2-3) [LHS = RHS form]: split into [LHS <= RHS form]                  #
        #                                   and [-LHS <= -RHS form].               #
        # (3) Check the resulting RHS's from (2).                                  #
        #     IF all the RHS's are positive:                                       #
        #          Go add (+1 coeff) slack variables for each constraints.         #
        #          Check the problem type as "COMMON".                             #
        #          Check the problem feasibility as "FEASIBLE"                     #
        #     Else:                                                                #
        #          Go add (+1 coeff) slack variables for constraints with positive #
        #             RHS's (name S_i),                                            #
        #             add (-1 coeff) slack variables for constraints with negative #
        #             RHS's (name S_i), and                                        #
        #             add (+1 coeff) artificial variables for constraints with     #
        #             negative RHS's (name A_i).                                   #
        #          Check the problem type as "TWOSTEP".                            #
        ############################################################################
        self._varCount = len(self._varNames)
        if self._objDirection == True:
            self._reducedCosts = self._objCoeffs
        else:
            self._reducedCosts = [-1*ele for ele in self._objCoeffs]

        for idx in range(len(self._conNames)):
            if self._ineqDirs[idx] == 'G':
                self._AMatrix[idx] = [-1*ele for ele in self._AMatrix[idx]]
                self._RHSs[idx] = -1*self._RHSs[idx]
            elif self._ineqDirs[idx] == 'E':
                self._AMatrix.append( [-1*ele for ele in self._AMatrix[idx]] )
                self._RHSs.append( -1*self._RHSs[idx] )

        s_ind, a_ind = 1, 1
        for idx in range(len(self._AMatrix)):
            self._varNames.append("S"+repr(s_ind))
            self._reducedCosts.append(0)
            for jdx in range(len(self._AMatrix)):
                if jdx == idx:
                    self._AMatrix[jdx].append(+1)
                else:
                    self._AMatrix[jdx].append(0)
            s_ind += 1

        for idx in range(len(self._RHSs)):
            if self._RHSs[idx] < 0:
                self._varNames.append("A"+repr(a_ind))
                self._reducedCosts.append(0)
                self._AMatrix[idx] = [-1*ele for ele in self._AMatrix[idx]]
                self._RHSs[idx] = -1*self._RHSs[idx]
                for jdx in range(len(self._AMatrix)):
                    if jdx == idx:
                        self._AMatrix[jdx].append(+1)
                    else:
                        self._AMatrix[jdx].append(0)
                self._basisSet.append("A"+repr(a_ind))
                a_ind += 1
            else:
                self._basisSet.append(self._varNames[self._varCount+idx])

        if a_ind < 1.0001: ## "COMMON" case.
            returnStr = "COMMON"
        else: ## "TWOSTEP" case.
            self._twoStep = True
            for name in self._varNames:
                if name[0] == "A":
                    self._reducedSecondary.append(-1)
                else:
                    self._reducedSecondary.append(0)
            returnStr = "TWOSTEP"
        return returnStr
```

```python

    def buildTableau(self):
        ##############################################################################
        # This function builds tableau for both "COMMON" and "TWOSTEP" cases        #
        # and reduce the constraints (non-zeroth rows of the tableau) using the     #
        # reduced row echelon form.                                                  #
        ##############################################################################
        Tableau = []
        if self._twoStep == True:
            Tableau.append( self._reducedSecondary )
        else:
            Tableau.append( self._reducedCosts )
        Tableau[-1].append( 0 )

        for idx in range(len(self._AMatrix)):
            Tableau.append( self._AMatrix[idx] )
            Tableau[-1].append( self._RHSs[idx] )

        Tableau = np.array(Tableau)
        #Tableau[1:,], tempIndices = self.reduceProblemMatrix(Tableau[1:,])
        tempConMat, tempIndices = self.reduceProblemMatrix(Tableau[1:,])
        tempConMat = np.vstack([Tableau[0,:], tempConMat])
        Tableau = tempConMat

        ## Reorder basis based on the indices from the reducedProblemMatrix
        tempList = [self._basisSet[i] for i in tempIndices]
        self._basisSet = tempList

        print("Initialize the given LP problem:")
        if self._twoStep:
            Tableau = self.initZeroth(Tableau)
            print("TWO STEP METHOD CALLED:")
        else:
            print("VANILLA SIMPLEX METHOD CALLED:")

        self.printCurrentStatus(Tableau)
        return Tableau

    def initZeroth(self, Tableau):
        ##############################################################################
        # This function initializes the 0th row of the tableau for basic columns.  #
        # It assumes that for the input tableau's basic columns, they are all one- #
        # hot vectors except for the zero-th element. This function simply finds   #
        # basic columns with -1 for the zero-th element, and add pivot rows to the #
        # zero-th row.                                                              #
        ##############################################################################
        colIndices = []
        for idx in range(Tableau.shape[1]):
            if (Tableau[0, idx] != 0 and self._varNames[idx] in self._basisSet):
                colIndices.append(idx)
        for idx in colIndices:
            rowIdx = np.argmax(Tableau[:, idx])
            ratio = -1*Tableau[0, idx]/Tableau[rowIdx, idx]
            Tableau[0, :] += ratio*Tableau[rowIdx, :]

        print("Initialize the zero-th row:")
        self.printCurrentStatus(Tableau)
        return Tableau

    def solve(self, Tableau):
        # Related to the requirement (R2).
        # Related to the requirement (R4).
        if self._twoStep == True:
            ## IF the problem type is "TWOSTEP":
            #     (a) Put a secondary objective function with (+1 coeff) for all artificial variables.
            #     (b) Run tableau simplex steps until it meets one of the [Tableau Simplex's Termination Conditions]
            #     (c) IF the optimal objective value from (b) is zero with all zero artificial values, then process to the 2nd step
            #         Otherwise, end procedure and say the original LP is not feasible.
            terminationFlag = 0
            print("Run the 1st step of Two Step Method")
            while(terminationFlag == 0):
                Tableau, terminationFlag = self.run_oneIteration(Tableau)
                self.printCurrentStatus(Tableau)
            if Tableau[0,-1] < THRESHOLD:
                ## When 1st phase of Two step method ends with feasibility guaranteed.
                ## (1) Setup 2nd phase tableau.
                removalIndices = [self._varNames.index(ele) for ele in self._varNames if ele[0]=="A"]
                Tableau = np.delete(Tableau, removalIndices, 1)
                for idx in range(Tableau.shape[1]-1):
                    Tableau[0,idx] = self._reducedCosts[idx]
                Tableau[0,-1] = 0
                ## (2) Restore 0 reduced costs for the basic columns.
                print("Run the Vanilla Simplex Method or the 2st step of Two Step Method")
                Tableau = self.initZeroth(Tableau)
                terminationFlag = 0
                while(terminationFlag == 0):
                    Tableau, terminationFlag = self.run_oneIteration(Tableau)
                    self.printCurrentStatus(Tableau)
            else:
                ## When 1st phase of Two step method ends otherwise.
                # [Tableau Simplex's Termination Conditions]
                # (T0) When 1st phase of Two step method ends otherwise.
                #    - "No feasible solution"
                print("- Terminate Procedure: No feasible solution for this problem.")
        else:
            ## IF the problem type is "COMMON":
            ## Run tableau simplex steps until it meets one of the [Tableau Simplex's Termination Conditions]
```

```python
            while(terminationFlag == 0):
                Tableau, terminationFlag = self.run_oneIteration(Tableau)
                self.printCurrentStatus(Tableau)
            self.printCurrentStatus(Tableau)
        return Tableau

    def run_oneIteration(self, Tableau):
        ##########################################################################
        # (1) Find a positive zeroth row element in the tableau with smallest index#
        #     value. Set the corresponding variable as "Entering Variable" in      #
        #     this iteration.                                                      #
        #     If there is no such element in the zeroth row, then terminate the    #
        #     iteration.                                                           #
        #     If there is no such element that (2) can find an element, then       #
        #     terminate the iteration.                                             #
        # (2) Find a non-negative element from the selected column with the minimun#
        #     ratio (rhs/ele). Set the corresponding variable as "Leaving Variable"#
        #     in this iteration.                                                   #
        #     If there exists a tie, then select the one with the smaller index.   #
        #     If there is no such element in the column, go to (1) and find another#
        #     column.                                                              #
        # (3) Multiply the pivot row by ratios (-Ele/pivotEle) and add it to the   #
        #     corresponding rows, respectively.                                    #
        #                                                                          #
        # Related to the requirement (R6)                                          #
        ##########################################################################
        terminationType = 0 # 0 - No termination
                            # 1 - Termination with optimal corner.
                            # 2 - Termination with unboundedness.
        pivotCol = None
        pivotRow = None
        failCol = []

        Once = False
        pivotCol = self.findCol(Tableau, failCol)

        if pivotCol == None:
            if Once == False:
                # [Tableau Simplex's Termination Conditions]
                # (T1) If all the reduced costs are negative.
                #    - "Optimal corner point found"
                #    - Return the optimal solution and the optimal objective value.
                print("- Terminate Procedure: Optimal Value = ", Tableau[0,-1])
                terminationType = 1
            else:
                # [Tableau Simplex's Termination Conditions]
                # (T2) If for all positive reduced cost columns, there are only non-positive elememts.
                #    - "Unboundedness found"
                #    - Return "-inf" for minimization problems and "+inf" for maximization problems.
                print("- Terminate Procedure: Unboundedness")
                terminationType = 2
        else:
            Once = True
            pivotRow = self.findRow(Tableau[:, pivotCol], Tableau[:, -1])
            if pivotRow == None:
                failCol.append(pivotCol)

            self._basisSet[pivotRow-1] = self._varNames[pivotCol]

            Tableau[pivotRow, :] = Tableau[pivotRow, :] / Tableau[pivotRow, pivotCol]

            for idx in range(Tableau.shape[0]):
                if idx != pivotRow:
                    ratio = -1*Tableau[idx, pivotCol]/Tableau[pivotRow, pivotCol];
                    Tableau[idx, :] += ratio * Tableau[pivotRow, :]
        return (Tableau, terminationType)

    def findCol(self, Tableau, failCol):
        ##########################################################################
        # This function returns the pivot column for simplex iteration.           #
        # Check the non-optimality condition from the last column because we always#
        # want to remove the artificial column first for the TWOSTEP level.       #
        # To achieve consistency in terms of the Blend's rule, we choose the      #
        # HIGHEST-numbered nonbasic column with a negative (reduced) cost.        #
        # Related to the requirement (R5): Blends' rule.                          #
        ##########################################################################
        pivotCol = None
        #for idx in range(Tableau.shape[1]-1):
        for idx in reversed(range(Tableau.shape[1]-1)):
            if (Tableau[0, idx] > THRESHOLD and idx not in failCol):
                pivotCol = idx
                break;
        return pivotCol

    def findRow(self, PivotCol, RHSs):
        ##########################################################################
        # This function returns the pivot row when pivot column is given in simplex#
        # We choose the row with the lowest ratio between the (transformed) right #
        # hand side and the coefficient in the pivot tableau where the coefficient #
        # is greater than zero.                                                   #
        # To achive consistency in terms of the blend's rule, when the minimum    #
        # ratio is shared by several rows, we choose the row with HIGHEST index.  #
        # Related to the requirement (R5): Blends' rule.                          #
        ##########################################################################
        pivotRow = None
        minRatio = float("Inf")
        #for idx in reversed(range(1, PivotCol.size)):
```

```python
        for idx in range(1, PivotCol.size):
            ## Selects larger index when it falls in tie situation.
            if PivotCol[idx] > 0:
                if RHSs[idx]/PivotCol[idx] <= minRatio:
                    minRatio = RHSs[idx]/PivotCol[idx]
                    pivotRow = idx
        return pivotRow


    def reduceProblemMatrix(self, ProblemMatrix):
        ##############################################################################
        # This function returns the problem matrix [A;b] after removing row vector #
        # redundancy.                                                              #
        # Related to the requirement (R3).                                         #
        ##############################################################################
        rref_results = self.rref(ProblemMatrix)

        rowFlags = [False]*rref_results[0].shape[0]
        # Check each rref matrix's row whether there is non-zero element and make a T/F flag array for it.
        for i in range(rref_results[0].shape[0]):
            for j in range(rref_results[0].shape[0]):
                if abs(rref_results[0][i][j]) < THRESHOLD:
                    rowFlags[i] = True
                    break;

        remainderIndices = []
        for idx in range(len(rowFlags)):
            if rowFlags[idx] == False:
                break;
            remainderIndices.append(rref_results[1][idx])
        return (ProblemMatrix[remainderIndices,:], remainderIndices)

    def rref(self, ProblemMatrix):
        ##############################################################################
        # This function returns (1) the reduced row echelon form (RREF) of the     #
        # problem matrix and (2) the list of original row indices from the input   #
        # matrix of each row of RREF. * ProblemMatrix = [A; b]                     #                         #
        # Reference Source:                                                        #
        # https://stackoverflow.com/questions/7664246/python-built-in-function-to- #
        #  do-matrix-reduction/7665269#7665269                                     #
        ##############################################################################
        Matrix = ProblemMatrix.copy()
        rows, cols = Matrix.shape
        r = 0
        pivots_pos = []
        row_exchanges = np.arange(rows)
        for c in range(cols):
            ## Find the pivot row:
            pivot = np.argmax (np.abs (Matrix[r:rows,c])) + r
            m = np.abs(Matrix[pivot, c])
            if m <= THRESHOLD:
                ## Skip column c, making sure the approximately zero terms are actually zero.
                Matrix[r:rows, c] = np.zeros(rows-r)
            else:
                ## keep track of bound variables
                pivots_pos.append((r,c))

                if pivot != r: ## Swap current row and pivot row
                    Matrix[[pivot, r], c:cols] = Matrix[[r, pivot], c:cols]
                    row_exchanges[[pivot,r]] = row_exchanges[[r,pivot]]

                ## Normalize pivot row
                Matrix[r, c:cols] = Matrix[r, c:cols] / Matrix[r, c];
                ## Eliminate the current column
                v = Matrix[r, c:cols]

                if r > 0: ## Above (before row r):
                    ridx_above = np.arange(r)
                    Matrix[ridx_above, c:cols] = Matrix[ridx_above, c:cols] - np.outer(v, Matrix[ridx_above, c]).T
                if r < rows-1: ## Below (after row r):
                    ridx_below = np.arange(r+1,rows)
                    Matrix[ridx_below, c:cols] = Matrix[ridx_below, c:cols] - np.outer(v, Matrix[ridx_below, c]).T
                r += 1
            if r == rows: ## Check if done
                break;
        return (Matrix, row_exchanges)
```