```python
"""L-Shaped Algorithm

A method for solving two-stage stochastic linear program with recourse

Implemented for:

Math 6367 Optimization (Prof. Dr. Ronald H.W. Hoppe)
HW01
Due 20 March 2019

Student:
Jonathan Schuba

Working with:
    python  3.6.6
    cvxpy   0.4.10
    numpy   1.15.1
"""

import cvxpy as cp
import numpy as np


class L_Shaped_Algorithm(object):
    """L-Shaped Algorithm for solving stochastic linear programs.

    The L-Shaped Algorithm is described in:

    Birge, J. R., & Louveaux, F. (2011). Introduction to stochastic
    programming. Springer Science & Business Media.

    Problems are in the form:

        minimize     c @ x + Q(x)
        subject to   A_eq @ x = b_eq
                     A_ineq @ x <= b_ineq
                     x >= 0

        where        Q(x) = E[Q(x, s(w))]
                     Q(x, s(w)) = min { q(w) @ y  |  W@y = h(w) - T(w)@x,  y>=0}
        note: here we use s to denote the random variable

    Parameters
    ----------
    c : array_like, required

    A_eq : array_like or matrix, or None

    b_eq : array_like, or None

    A_ineq : array_like or matrix, or None
```

```
b_ineq : array_like, or None

W : array_like or matrix, required

h_driver : function, required
    Should take two arguements (x, s) and return an array_like. See Notes.

T_driver : function, required
    Should take two arguements (x, s) and return an array_like. See Notes.

q : list of array_likes, required
    Entries of q, realizations, and probabilities should have aligned
    indicies.  Each member of these lists should correspond to a particular
    realization of the random variables

realizations : list of array_likes, required
    The value of the random variable(s) for each possible realization

probabilities : list of array_likes, required
    Probability for each random variable realization

max_iter : int, optional, default 100
    Maximum iterations before forced stopping

precision : float, optional, default 10e-6
    Precision to check for equality of floats.  Also, the precision to
    round off the final solution.  Full floating point precision is
    maintained during intermediate calculations.

verbose : bool, optional, default False
    Whether to print detailed information from intermediate steps. Basic
    information will always be printed.

debug : bool, optional, default False
    Whether to print advanced debug information


Attributes
----------
solution : numpy ndarray
    Final optimal solution to the stochastic linear program

value : numpy float
    Final optimal objective value at the solution

print_precision : int
    Number of digits after the decimal point. Created from precision.

K_ : int
    Number of possible realizations of the random variable(s)
```

```
nu : int
    Iteration counter

r : int
    Counter for feasibility cuts

s : int
    Counter for optimality cuts

D_list : list of ndarrays
    List of matrices for feasibility cuts

d_list : list of ndarrays
    List of vectors for feasibility cuts

E_list : list of ndarrays
    List of matrices for optimality cuts

e_list : list of ndarrays
    List of vectors for optimality cuts

x_nu_list : list of ndarrays
    List containing the solution values obtained in step 1 from each
    iterate

theta_nu_list : list of ndarrays
    List containing the value of theta obtained in step 1 from each
    iterate

objective_value_list : list of ndarrays
    List of the objective values from step 1 in each iterate



Notes
-----
In general, h(w) and T(w) may change depending on the value of the current
x iterate or the random variable value under consideration.  Therefore,
h(w) and T(w) are specified by driver functions which look like, for
example:

def T_driver(x, s):
    return np.array([[-1,0],
                     [0,-1],
                     [0,0],
                     [0,0],
                     [0,0],
                     [0,0]])

def h_driver(x, s):
```

```
    return np.array([0, 0, -0.8*s[0], -0.8*s[1], s[0], s[1]])

Examples
--------
#The following example solves Example 2 from Page 188 of Birge & Louveaux.

import numpy as np
from l_shaped_algorithm_cvx import L_Shaped_Algorithm

c = np.array([0])

A_ineq = [1]
b_ineq = [10]

W = np.array([1])

h = []
T = []

q = [[1],[1],[1]]
s = [1,2,4]
p = [1/3,1/3,1/3]

def T_driver(x, s):
    if x <= s:
        return np.array([1])
    else:
        return np.array([-1])

def h_driver(x, s):
    if x <= s:
        return np.array([s])
    else:
        return np.array([-s])

Solver = L_Shaped_Algorithm(c = c,
                            A_eq = None,
                            b_eq = None,
                            A_ineq = A_ineq,
                            b_ineq = b_ineq,
                            W = W,
                            h_driver = h_driver,
                            T_driver = T_driver,
                            q = q,
                            realizations = s,
                            probabilities = p,
                            max_iter = 100,
                            precision=10e-6,
                            verbose=False, debug=False)
x_opt = Solver.solve()
```

```python
print (Solver.value)
print (Solver.solution)

"""

def __init__(self, c, A_eq, b_eq, A_ineq, b_ineq, W, h_driver, T_driver, q,
             realizations, probabilities,
             max_iter = 100, precision=10e-6,
             verbose=False, debug=False):

    self.c = c
    self.A_eq = A_eq
    self.b_eq = b_eq
    self.A_ineq = A_ineq
    self.b_ineq = b_ineq
    self.W = W
    self.h_driver = h_driver
    self.T_driver = T_driver
    self.q = q
    self.realizations = realizations
    self.p = probabilities

    self.max_iter = max_iter
    self.precision = precision

    self.debug = debug
    self.verbose = verbose

    np.set_printoptions(suppress=True)
    np.set_printoptions(precision=abs(int(np.log10(self.precision))))
    self.print_precision = abs(int(np.log10(self.precision)))


    # K is the number of possible realizations of the random variable(s)
    self.K_ = len(q)

    # Check that lengths match
    if self.K_ != len(self.p):
        raise ValueError("q and p should be same length")


    #Initialize counters and lists to store computed quantities

    self.nu = 0          # iteration counter
    self.r = 0           # counter for feasibility cuts
    self.s = 0           # counter for optimality cuts

    #Matrices and vectors which will form constraints pertaining to
    #feasibility cuts, ie:
    #    D[i] @ x >= d[i]  where 1 <= i <= r
```

```
        self.D_list = []         # list of matrices for feasibility cuts
        self.d_list = []         # list of vectors for feasibility cuts


        #Matrices and vectors which will form constraints pertaining to
        #optimality cuts, ie:
        #   E[i] @ x >= e[i]  where 1 <= i <= s

        self.E_list = []         # list of matrices for optimality cuts
        self.e_list = []         # list of vectors for optimality cuts


        #Lists to hold the values obtained in each iteration
        self.x_nu_list = []
        self.theta_nu_list = []
        self.objective_value_list = []

        self.value = None
        self.solution = None


    def solve(self):
        """Solve the stochastic linear program as specified

        Returns
        -------
        solution : numpy ndarray
            The optimal solution, x, to the problem
        """

        for _ in range(self.max_iter):
            self.nu += 1 # iterate step counter
            print()
            print( "===========================================")
            print(f"=============== Iteration {self.nu} ===============")
            print( "===========================================")

            _ = self._step_1()

            cut_made = self._step_2()

            if cut_made == 1:
                # A feasibility cut was made
                # Go back to step 1
                continue
            else:
                print("No feasibility cuts needed")

            cut_made = self._step_3()
            if cut_made == 0:
                # optimal solution found
```

```
                self.value = np.round(self.objective_value_list[-1],
                                      self.print_precision)
                self.solution = np.round(self.x_nu_list[-1],
                                         self.print_precision)

                print()
                print("Optimal Solution Found")
                print()
                print("Objective Value  = ", self.value)
                print("Optimal Solution = ", self.solution)
                return self.solution

        # If no solution is found after max_iter steps, then return None
        print()
        print(f"Maximum iterations ({self.max_iter}) reached, and no ",
               "optimal solution found")
        print("Try increasing max_iter or decreasing precision")
        return None

    def dot(self, a, b):
        """Return the dot product of two vectors
        Uses the numpy @ operator.
        If the expression involves a cvxpy variable which is actually a scalar,
        the @ operator doesn't work, so return the product instead.
        """
        try:
            return a @ b
        except ValueError:
            return a * b


    def _step_1(self):
        """Solve the linear program with any constraints imposed by previous
        feasibility and optimality cuts.
        """

        print (f"----------------- Step 1 -----------------")

        n = len(self.c)

        x = cp.Variable(n)
        theta = cp.Variable(1)


        if self.s == 0:
            # There are no optimality cuts, so set theta to -inf
            objective = cp.Minimize(self.dot(self.c, x))
        else:
            objective = cp.Minimize(self.dot(self.c, x) + theta)
```

```
constraints = [x >= 0]

if self.A_eq is not None:
    # We must append the equality constraints on x
    constraints.append( self.dot(self.A_eq, x) == self.b_eq )
if self.A_ineq is not None:
    # We must append the inequality constraints on x
    constraints.append( self.dot(self.A_ineq, x) <= self.b_ineq )

for r in range(len(self.D_list)):
    # add constraints for each feasibility cut
    constraints.append( self.dot(self.D_list[r], x) >= self.d_list[r] )

for s in range(len(self.E_list)):
    # add constraints for each optimality cut
    constraints.append(
            self.dot(self.E_list[s], x) + theta >= self.e_list[s] )

prob = cp.Problem(objective, constraints)
result = prob.solve(verbose=self.verbose)

if result is None and self.nu == 1:
    self.objective_value_list.append(0)
    self.x_nu_list.append(np.zeros(self.c.shape))
    self.theta_nu_list.append(-np.inf)
    return 1

# CVX sometimes makes the variables into funny size matrices, so we
# need to make them n-by-1 vectors
x_solution = np.array([x.value])
x_solution = x_solution.reshape(x_solution.size)

if self.s == 0:
    theta_solution = -np.inf
else:
    theta_solution = theta.value


print ("objective value = ", np.round(result,
                                      self.print_precision))
print ("x_nu            = ", np.round(x_solution,
                                      self.print_precision))
print ("theta_nu        = ", np.round(theta_solution,
                                      self.print_precision))

self.objective_value_list.append(result)
self.x_nu_list.append(x_solution)
self.theta_nu_list.append(theta_solution)

return 1
```

```
def _step_2(self):
    """Solve LPs for each possible realization of the random variables, and
    make feasibility cuts as appropriate.
    """

    print ()
    print (f"----------------- Step 2 -----------------")

    n = self.W.shape[0]
    if len(self.W.shape) > 1:
        m = self.W.shape[1]
    else:
        m = 1

    for k in range(self.K_):
        vp = cp.Variable(n)
        vm = cp.Variable(n)
        y = cp.Variable(m)

        objective = cp.Minimize(cp.sum_entries(vp) + cp.sum_entries(vm))

        # We use the user-specified driver functions to get the correct
        # matrix T and h for this particular realization of the random
        # variables
        T = self.T_driver(self.x_nu_list[-1], self.realizations[k])
        h = self.h_driver(self.x_nu_list[-1], self.realizations[k])


        constraints = [
        self.dot(self.W, y) +vp-vm == h - self.dot(T, self.x_nu_list[-1]),
                           vp >= 0,
                           vm >= 0,
                           y >= 0]

        prob = cp.Problem(objective, constraints)
        result = prob.solve(verbose=self.verbose)

        if np.abs(result) > self.precision:
            # Then we need to add a feasibility cut
            self.r += 1

            # Get the dual variables
            sigma = -1 * constraints[0].dual_value
            sigma = np.array(sigma).reshape(sigma.size)

            print ("Feasibility cut identified")
            print ("objective      = ",
                   np.round(result, self.print_precision))
            print ("dual objective = ",
```

```
                np.round((h - T @ self.x_nu_list[-1]) @ sigma,
                         self.print_precision+1))
        print ("dual variables = ",
               sigma)

        D = sigma.T @ T
        d = sigma.T @ h
        print ("Dk = ", np.round(D, self.print_precision))
        print ("dk = ", np.round(d, self.print_precision))
        self.D_list.append(D)
        self.d_list.append(d)
        return 1 # cut was made

    # If we get through all realizations of the random variables, and no
    # infeasibilities were identified, then return 0
    return 0 # cut was not needed



def _step_3(self):
    """Solve LPs for each possible realization of the random variable, and
    make optimality cuts as appropriate.
    """

    #n = self.W.shape[0]
    if len(self.W.shape) > 1:
        m = self.W.shape[1]
    else:
        m = 1

    print ()
    print (f"----------------- Step 3 -----------------")

    # Setup the variables E and e
    E = np.zeros(len(self.x_nu_list[-1]))
    e = 0

    for k in range(self.K_):
        y = cp.Variable(m)

        # We use the user-specified driver functions to get the correct
        # matrix T and h for this particular realization of the random
        # variables
        T = self.T_driver(self.x_nu_list[-1], self.realizations[k])
        h = self.h_driver(self.x_nu_list[-1], self.realizations[k])

        # Define the objective function and constraints
        objective = cp.Minimize(self.dot(self.q[k], y[0:len(self.q[k])]))
        constraints = [
                self.dot(self.W, y) == h - self.dot(T, self.x_nu_list[-1]),
                y >= 0]
```

```
        prob = cp.Problem(objective, constraints)
        result = prob.solve(verbose=self.verbose)

        # Get the dual variables
        pi = -1 * np.array(constraints[0].dual_value)
        pi = np.array(pi).reshape(pi.size)

        if self.debug:
            print ("objective       = ", result)
            print ("dual objective = ", (h - self.dot(self.dot(T,
                                        self.x_nu_list[-1]), pi)))
            print ("dual variables = ", pi)

        E += self.p[k] * pi.T @ T
        e += self.p[k] * pi.T @ h

    w_nu = e - E @ self.x_nu_list[-1]

    if np.abs(self.theta_nu_list[-1] - w_nu) <= self.precision:
        # The solution is optimal
        return 0 # no cut needed, solution is optimal

    # Else append optimality cut
    if self.verbose:
        print ("w_nu = ", w_nu)
        print ("theta_nu = ", self.theta_nu_list[-1])
    print ("Optimality cut made")
    print ("E = ", np.round(E, self.print_precision))
    print ("e = ", np.round(e, self.print_precision))
    self.s += 1
    self.E_list.append(E)
    self.e_list.append(e)
    return 1 # a cut was made
```