ECI 249 HW #5

Kenneth Larrieu

a) Formulate this problem as an optimization problem, minimizing the net expected value cost of managing the floodplain costs and damage.

The optimal floodplain management options may be found by minimizing the net expected value cost for floodplain management. The next expected value cost is:

$$EVC = \sum_{i=1}^{3} c_{P_i} X_{P_i} + \sum_{s=1}^{5} p(s) \left(\sum_{j=1}^{3} c_{E_{js}} X_{E_{js}} + D_s \right)$$

where

X_{P_i} is the number of units used for permanent floodplain management option P_i

c_{P_i} is the annualized unit cost of the option

$p(s)$ is the probability of flow range s

$X_{E_{js}}$ is the number of units used for emergency floodplain management option E_{js} at flow range s

$c_{E_{js}}$ is the unit cost of that option, and

D_s is the damage incurred by flood s given the applied floodplain management options:

$$D_s = d_s - \sum_{i=1}^{3} r_{P_i} X_{P_i} + \sum_{j=1}^{3} r_{E_{js}} X_{E_{js}}$$

where d_s is the damage incurred at flood s in the absence of floodplain management actions

Thus, we want to find the vector X_{P_i} of permanent option quantities and the tensor $X_{E_{js}}$ of temporary option quantities at each discharge range which minimize EVC, subject to the imposed bounds on values of their elements.

The restriction of damage to non-negative values can be expressed by the following set of additional constraints on the system:

$$\sum_{i=1}^{3} r_{P_i} X_{P_i} + \sum_{j=1}^{3} r_{E_{js}} X_{E_{js}} \leq d_s, \quad \forall s$$

where r_i denotes the reduction in damage per unit X_i.

b) Minimize the expected value cost of flood damage and control costs by selecting which permanent and short-term flood control measures should be undertaken. Do this for both current conditions and with the upstream project.

The following information is given regarding the constants for the system:

Table 1: Event Probabilities and Damages with No Action

Peak Flow	Current Annual Probability	Annual Probability with Upstream Project	Event damage with No Action ($ millions)
< 5,000 cfs	0.80	0.90	0
5-6,000 cfs	0.11	0.05	2.1
6-8,000 cfs	0.06	0.03	3.0
8-10,000 cfs	0.02	0.01	4.2
10,000+ cfs	0.01	0.01	6.0

Table 2: Permanent Floodplain Management Option Characteristics

Characteristic	**Raising Structures**	**Warning system**	**Sacrificial First Stories**
Unit of implementation	yd^3 of fill	$ invested	Building sq. feet
Unit cost	$10	$1	$40
Limit of implementation	1,000,000	$200,000	200,000
Unit damage reduction per event:			
<5,000 cfs	0	0	0
5-6,000 cfs	$100	$2	$100
6-8,000 cfs	$70	$3	$60
8-10,000 cfs	$60	$4	$50
10,000 + cfs	$10	$7	$20

Table 3: Emergency Floodplain Management Option Characteristics

Characteristic	**Evacuation**	**Sandbagging of levees**	**Heightened levee monitoring**
Unit of implementation	0 or 1	ft.	$spent
Unit cost	$200,000	$20,000	$1
Limit of implementation	1	2	$20,000
Unit damage reduction per event:			
<5,000 cfs	0	0	0
5-6,000 cfs	$200,000	$1,000,000	$2
6-8,000 cfs	$300,000	$800,000	$1
8-10,000 cfs	$500,000	0	0
10,000 + cfs	$1,000,000	0	0

The linear programming problem is then implemented with the Python library `pulp`:

In [1]:

```python
import pulp
import pandas as pd
import numpy as np

# set LaTeX formatting
def mc(string):
    return '\multicolumn{1}{p{2.5cm}}{\centering %s}' % string.replace(', ', r' \\ ')
def _repr_latex_(self):
    lself = self
    lself = lself.rename(columns=dict(zip(self.columns.tolist(), [mc(name) for name in self.columns.
    return r'\begin{center}' + '\n%s\n' % lself.to_latex(index=True, escape=False) + r'\end{center}'
pd.DataFrame._repr_latex_ = _repr_latex_

def dot(l1, l2):
```

```python
    """Dot product between two lists"""
    return sum([e1 * e2 for e1, e2 in zip(l1, l2)])

class LP:
    def __init__(self, *args, **kwargs):
        self.q_names = ['<5,000 cfs', '5-6,000 cfs', '6-8,000 cfs', '8-10,000 cfs', '10,000+ cfs']
        self.q_probs = [0.8, 0.11, 0.06, 0.02, 0.01]
        if 'upstream' in args:
            self.q_probs = [0.9, 0.05, 0.03, 0.01, 0.01]
        self.damage0 = [0, 2.1e6, 3e6, 4.2e6, 6e6]

        self.perm_names = ['Raise Structures', 'Warning System', 'Sacrificial First Stories']
        self.perm_costs = [10, 1, 40]
        self.perm_lims = [1e6, 200e3, 200e3]
        self.perm_reds = [[0, 100, 70, 60, 10], [0, 2, 3, 4, 7], [0, 100, 60, 50, 20]]
        # test case (Jay's paper example)
        # self.perm_reds = [[0, 200, 90, 70, 10], [0, 3, 4, 6, 10], [0, 100, 60, 50, 20]]

        self.em_names = ['Evacuate', 'Sandbagging', 'Heightened Levee Monitoring']
        self.em_costs = [200e3, 20e3, 1]
        # test case (Jay's paper example)
        # self.em_costs = [100e3, 30e3, 1]
        self.em_lims = [1, 2, 20e3]
        self.em_reds = [[0, 200e3, 300e3, 500e3, 1e6], [0, 1e6, 800e3, 0, 0], [0, 2, 1, 0, 0]]

        if 'costs' in kwargs.keys():
            self.perm_costs = kwargs['costs'][:3]
            self.em_costs = kwargs['costs'][3:]

        # make set of linear coefficients for optimization
        # index 0-2: perm options
        self.cs = []
        for i in range(len(self.perm_names)):
            c = self.perm_costs[i] - dot(self.q_probs, self.perm_reds[i])
            self.cs.append(c)
        # index 3-17: em options, listed primarily by q, secondarily by option
        for q in range(len(self.q_names)):
            for j in range(len(self.em_names)):
                c = self.q_probs[q] * (self.em_costs[j] - self.em_reds[j][q])
                self.cs.append(c)


    def run_LP(self, print_out=True):
        # initialize decision variables
        # list of permanent variables
        perm_vars = [pulp.LpVariable(self.perm_names[i],
                                     lowBound=0, upBound=self.perm_lims[i], cat='Integer')
                     for i in range(len(self.perm_names))]
        # list of dicts of emergency variables, dict for each option w/ flow names as keys
        em_vars = [pulp.LpVariable.dict(self.em_names[j], self.q_names,
                                        lowBound=0, upBound=self.em_lims[j], cat='Integer')
```

```
                    for j in range(len(self.em_names))]

        # list of variables to dot with self.cs for objective function
        self.varlist = perm_vars
        for q in self.q_names:
            for j in range(len(self.em_names)):
                self.varlist.append(em_vars[j][q])

        # initialize model, set to minimize objective fn
        model = pulp.LpProblem('Optimizing Floodplain Management', pulp.LpMinimize)

        # define objective function
        model.objective += pulp.lpSum([self.cs[i] * self.varlist[i]
                                       for i in range(len(self.varlist))])
        model.objective += dot(self.q_probs, self.damage0)

        # initialize constraints
        for q in range(1, len(self.q_names)):
            model += pulp.lpSum([self.perm_reds[i][q] * perm_vars[i]
                                 for i in range(len(self.perm_names))]) + pulp.lpSum([self.em_reds[j

        model.solve()
        if print_out:
            print('\nModel status: %s' % pulp.LpStatus[model.status])
            print('\nObjective function value: %.2f\n' % pulp.value(model.objective))

        self.perm_df = pd.DataFrame({'Implemented Quantity':
                                     [perm_vars[i].value() for i in range(len(self.perm_names))]},
                                    index=self.perm_names).astype('Int64')
        self.perm_df.index.name = 'Permanent Option'
        self.em_df = pd.DataFrame(dict(zip(self.q_names,
                                           [[em_vars[j][q].value() for j in range(len(self.em_names)
                                            for q in self.q_names])), index=self.em_names).astype('I
        self.em_df = self.em_df[self.q_names]
        self.em_df.index.name = 'Emergency Option'

        return model


lp = LP()
model = lp.run_LP()
```

```
Model status: Optimal

Objective function value: 219300.00
```

In [2]:

```
lp.perm_df
```

Out [2]:

Permanent Option	Implemented Quantity
Raise Structures	1000
Warning System	0
Sacrificial First Stories	0

In [3]:

```
lp.em_df
```

Out [3]:

Emergency Option	<5,000 cfs	5-6,000 cfs	6-8,000 cfs	8-10,000 cfs	10,000+ cfs
Evacuate	0	0	1	1	1
Sandbagging	0	2	2	0	0
Heightened Levee Monitoring	0	0	0	0	0

Running the program again, this time with the upstream flood probability distribution:

In [4]:

```
lp2 = LP('upstream')
model2 = lp2.run_LP()
```

```
Model status: Optimal

Objective function value: 137200.00
```

In [5]:

```
lp2.perm_df
```

Out [5]:

Permanent Option	Implemented Quantity
Raise Structures	0
Warning System	0
Sacrificial First Stories	0

In [6]:

```
lp2.em_df
```

Out [6]:

Emergency Option	<5,000 cfs	5-6,000 cfs	6-8,000 cfs	8-10,000 cfs	10,000+ cfs
Evacuate	0	0	1	1	1
Sandbagging	0	2	2	0	0
Heightened Levee Monitoring	0	20000	0	0	0

c) What is the value of the upstream project, in terms of flood damage reduction?

In [7]:

```
pulp.value(model.objective) - pulp.value(model2.objective)
```

Out [7]:

```
82100.0
```

The reduction in expected cost for the upstream project is therefore $\boxed{\$82,100.}$

d) For each used and unused decision, what would be the range of unit costs for which these optimized decisions would not change? Present this as a table.

In [8]:

```
lower_bounds = []
upper_bounds = []
varlist = lp.perm_names + lp.em_names
original_costs = lp.perm_costs + lp.em_costs
low_check = [0] * len(original_costs)
hi_check = [1e9 * cost for cost in original_costs]

def step_size(original_val):
    '''set iteration step size for range accuracy within 1% of original unit cost'''
    order= round(np.log10(original_val), 0) - 2
    step = max(int(10**order), 1)
    return step

for i, var_name in enumerate(varlist):
    costs_l = original_costs[:]
    costs_u = original_costs[:]

    # get lower bound
    #check low limit first
    low_costs = original_costs[:]
    low_costs[i] = low_check[i]
    lp_low = LP(costs=low_costs)
    model_low = lp_low.run_LP(print_out=False)
    if (lp_low.perm_df.equals(lp.perm_df)) and (lp_low.em_df.equals(lp.em_df)):
        lower_bounds.append(0)
    else:
        step = step_size(original_costs[i])
        while True:
            # change value of variable
```

```
            costs_l[i] -= step
            # initialize new problem and solve
            lp_l = LP(costs=costs_l)
            model_l = lp_l.run_LP(print_out=False)
            # check if decisions change
            if not ((lp_l.perm_df.equals(lp.perm_df)) and (lp_l.em_df.equals(lp.em_df))):
                lower_bounds.append(int(costs_l[i]+step))
                break

    # get upper bound
    #check upper limit first
    hi_costs = original_costs[:]
    hi_costs[i] = hi_check[i]
    lp_hi = LP(costs=hi_costs)
    model_hi = lp_hi.run_LP(print_out=False)
    if (lp_hi.perm_df.equals(lp.perm_df)) and (lp_hi.em_df.equals(lp.em_df)):
        upper_bounds.append(None)
    else:
        step = step_size(original_costs[i])
        while True:
            # change value of variable
            costs_u[i] += step
            # initialize new problem and solve
            lp_u = LP(costs=costs_u)
            model_u = lp_u.run_LP(print_out=False)
            # check if decisions change
            if not ((lp_u.perm_df.equals(lp.perm_df)) and (lp_u.em_df.equals(lp.em_df))):
                upper_bounds.append(int(costs_u[i]-step))
                break


range_df = pd.DataFrame({'Default Unit Cost': original_costs,
                         'Lower Bound':lower_bounds,
                         'Upper Bound':upper_bounds},
                        index=varlist).astype('Int64')
range_df
```

Out [8]:

	Default Unit Cost	Lower Bound	Upper Bound
Raise Structures	10	6	11
Warning System	1	1	NaN
Sacrificial First Stories	40	10	NaN
Evacuate	200000	1000	299000
Sandbagging	20000	100	409000
Heightened Levee Monitoring	1	1	NaN

('NaN' indicates that there is no bound)