ECI 249 HW #4

Kenneth Larrieu

1. Develop a stochastic dynamic program to find the optimal release policy.

In [1]:

```
import numpy as np
import pandas as pd


class SDP:
    def __init__(self):
        self.stages = list(range(1, 21))
        self.sts = list(range(0, 80, 10))
        self.itm1s = list(range(10, 60, 10))
        self.rts = list(range(0, 60, 10))
        self.res_cap = 70
        self.r_target = 30
        self.trans_matrix = [[0.8, 0.1, 0.1, 0, 0],
                             [0.3, 0.3, 0.3, 0.1, 0],
                             [0.1, 0.3, 0.3, 0.2, 0.1],
                             [0.1, 0.3, 0.3, 0.2, 0.1],
                             [0.1, 0.2, 0.3, 0.3, 0.1]]
        self.decision_dict = {}
        self.obj_fn_dict = {}

    def pen(self, rt, st, it):
        penalty = 0
        # if we are able to release the release decision amount
        if rt <= st + it:
            # and release decision is less than target amount
            if rt < self.r_target:
                penalty += 10 * self.r_target *
                           (np.exp(2*(self.r_target-rt)/self.r_target) - 1)
        # if we cannot release the release decision amount
        else:
            penalty += 5000
            # in this case the actual amount released is st + it
            # if the actual amount released is less than target
            if st + it < self.r_target:
                penalty += 10 * self.r_target *
                           (np.exp(2*(self.r_target-(st + it))/self.r_target) - 1)

        return penalty

    def prob(self, it, itm1):
        # Markovian probability of I_t given I_{t-1}
        i1 = self.itm1s.index(itm1)
        i2 = self.itm1s.index(it)
        return self.trans_matrix[i1][i2]
```

```
    def obj_fn(self, stage, rt, st, itm1):
        # args: stage, state variables, and release decision
        # iterate over possible I_t values and get corresponding penalty
        # weight by probability of that I_t and sum
        ev_penalty = 0
        for it in self.itm1s:
            # the penalty today for the given it
            penalty_today = self.pen(rt, st, it)
            # the accumulated penalty for the given it
            if stage < self.stages[-1]:
                i1 = self.sts.index(np.clip(st-rt+it, 0, self.res_cap))
                i2 = self.itm1s.index(it)
                accum_penalty = self.obj_fn_dict[stage+1][i1][i2]
            else:
                accum_penalty = 0

            p = self.prob(it, itm1)

            ev_penalty += (penalty_today + accum_penalty) * p

        return ev_penalty

    def get_best_decision(self, stage):
        """
        At given stage, determine best decision at each state
        Save decision (release value) and objective function value to corresponding dicts
        """
        dec_array = []
        obj_fn_array = []
        # iterate over storage state var
        for st in self.sts:
            dec_row = []
            obj_fn_val_row = []
            # iterate over previous day inflow state var
            for itm1 in self.itm1s:
                # iterate over release decision var
                best_rt = None
                best_obj_fn_val = np.inf
                for rt in self.rts:
                    # evaluate objective function for each decision
                    obj_fn_val = self.obj_fn(stage, rt, st, itm1)
                    if obj_fn_val < best_obj_fn_val:
                        best_rt = rt
                        best_obj_fn_val = obj_fn_val
                # keep best decision and corresponding obj fn value
                dec_row.append(best_rt)
                obj_fn_val_row.append(best_obj_fn_val)

            dec_array.append(dec_row)
            obj_fn_array.append(obj_fn_val_row)
```

```
        self.decision_dict[stage] = dec_array
        self.obj_fn_dict[stage] = obj_fn_array

    def run_sdp(self):
        for stage in self.stages[::-1]:
            self.get_best_decision(stage)


sdp = SDP()
sdp.run_sdp()
```

2. Produce a table indicating the optimal release policy as a function of current storage and previous inflow.

In [2]:

```
df = pd.DataFrame(sdp.decision_dict[1], index=sdp.sts, columns = sdp.itm1s)
```

	Previous Day Inflows				
Storage	10	20	30	40	50
0	10	10	10	10	10
10	20	20	20	20	20
20	20	20	20	20	20
30	20	20	20	20	20
40	20	20	20	20	20
50	20	20	30	30	30
60	20	30	30	30	30
70	20	30	30	30	30

3. Very briefly identify some of the benefits and problems of solving such a problem by SDP.

- Advantages: DP is vastly superior to brute force enumeration for identifying the optimal policy because it restricts the parameter space based on the optimal policy for future stages. SDP is also a good strategy because it accounts for uncertainties at play e.g. the future inflows to the reservoir.
- Disadvantages: SDP uses discrete parameter values and stages. In addition, these discretized steps must be fairly coarse in order for SDP to be computationally tractable. As the number of decision/state variables increases, the problem becomes vastly more computationally expensive as well. Therefore, SDP can only practically be applied to vastly simplified versions of the system being modeled.