ECE276B-HW1-p3

Problem 3(b)

When T = 3, $A = \begin{bmatrix} 0.75 & -1 \\ 1 & 0.75 \end{bmatrix}$, $B = \begin{bmatrix} 1 & 0.5 \\ 0.5 & 0.5 \end{bmatrix}$, we can consider the problem based on (a).

The optimal value function

$$V_t^*(\mathbf{x}_t) = min\{l(\mathbf{x}_t, u_t) + V_{t+1}^*(A\mathbf{x}_{t+1}), l(\mathbf{x}_t, u_t) + V_{t+1}^*(B\mathbf{x}_{t+1})\}, t = 0, 1, ..., T-1$$

$$V_T^*(\mathbf{x}) = l(\mathbf{x}_T) = \frac{1}{2}\mathbf{x}_T^T\mathbf{x}_T, t = T$$

Thus, the dynamic programming actually can be realized by a recursive function, which is Optimal_Search(x, T, t) below.

The code is shown as following:

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        from mpl_toolkits.mplot3d import Axes3D
        from matplotlib import cm

        # Some useful functions
        def motion_model(xt, ut):
            """
                Inputs:
                    xt - current state
                    ut - control input.
                        ut = 0 <- policy 1
                        ut = 1 <- policy 2
                Output:
                    xt - next state
            """

            if ut == 0:
                A = np.array([[.75, -1],
                              [1, .75]])
                xt = A @ xt
            else:
                B = np.array([[1, .5],
                              [.5, .5]])
                xt = B @ xt
```

```
        return xt

    def ComputeCost(x):
        """
            Compute cost
        """
        x = x.reshape(2, 1)
        return 1/2 * (x.T @ x)
```

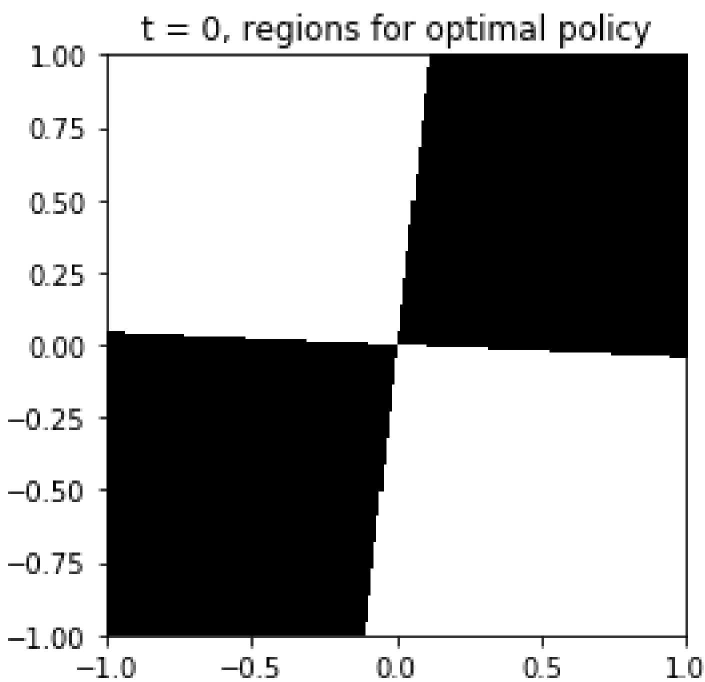
In [2]:
```
def Optimal_Search(x, T, t):
        """
            Inputs:
                x is the initial condition, it should be a 2-by-1 array.
                T is the horizon.
                 start at t = 2, T = 1;
                       at t = 1, T = 2;
                       at t = 0, T = 3.
            Outputs:
                policy is the policy for every stage.
                V_opt is the optimal cost-to-go values.
        """
        if t == T:
            return np.argmin(ComputeCost(x)), np.min(ComputeCost(x))
        else:
            V1 = ComputeCost(x) + Optimal_Search(motion_model(x, 0), T, t + 1)[1]
            V2 = ComputeCost(x) + Optimal_Search(motion_model(x, 1), T, t + 1)[1]
            policy = np.argmin([V1, V2])
            V_opt = np.min([V1, V2])
            return  policy, V_opt
```

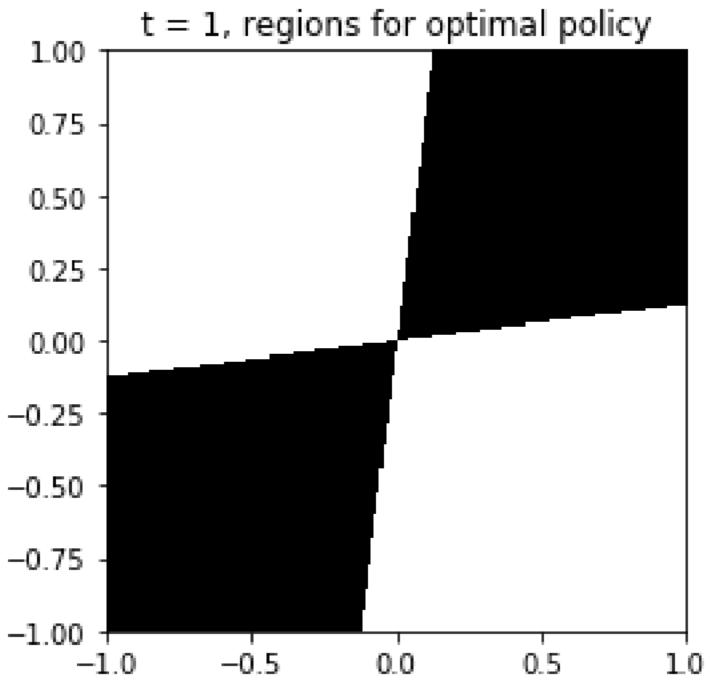
In [7]:
```
# main function
    # create a grid on [-1, 1] x [-1, 1]
    N = 200
    x_range = np.linspace(-1, 1, N)
    y_range = x_range.copy()
    X, Y = np.meshgrid(x_range, y_range)
    pts_list = np.vstack([X.ravel(), Y.ravel()])# return 2-by-N matrix
    pts = np.stack((X, Y), axis = 0) # stack X,Y as 2xNxN

    policy = np.zeros((3, N, N))
    V = np.zeros((3, N, N))
    for i in range(N):
        for j in range(N):
            x0 = pts[:, i, j]
            x0 = x0.reshape(2, 1)
            for t in range(3):
                policy[t, i, j], V[t, i, j] = Optimal_Search(x0, 3, t)
```

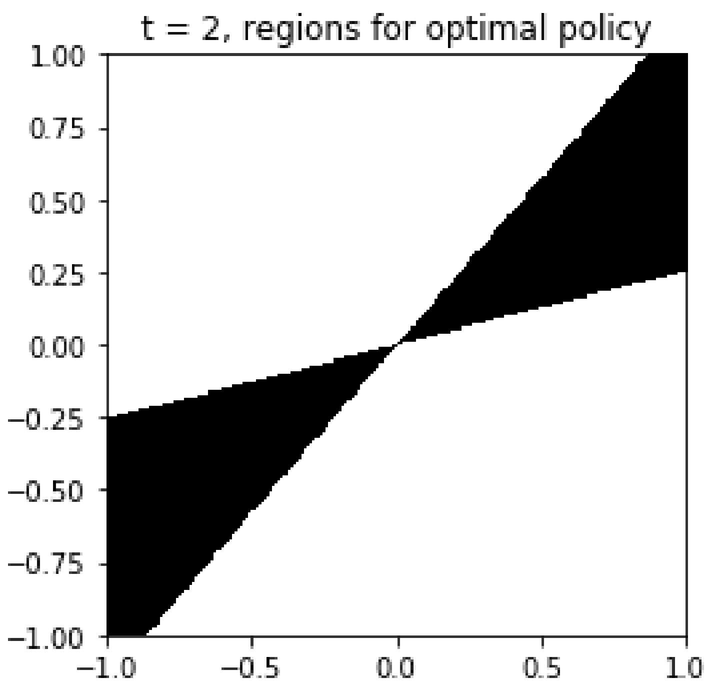
In [15]:
```
# display the results
```

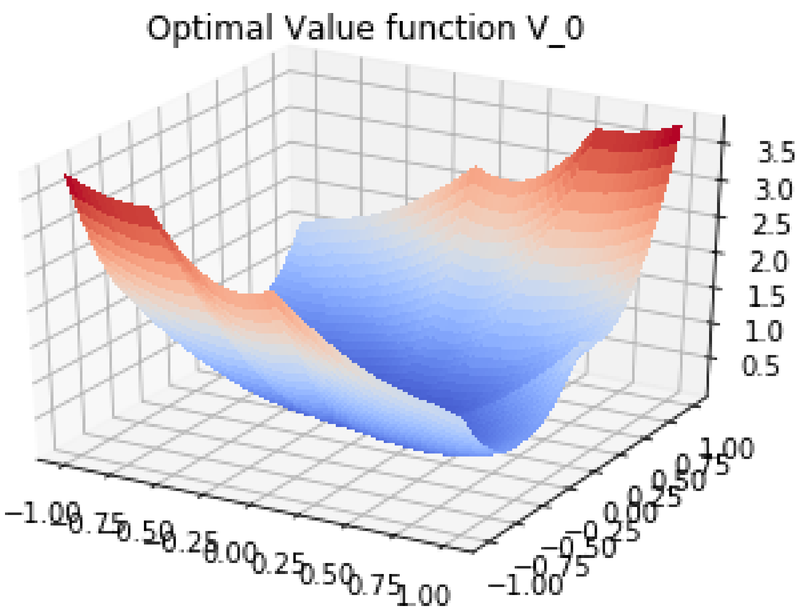
```
plt.figure()
plt.imshow(policy[0, :, :].T, cmap='gray', origin='lower', extent=[-1,1,-1,1])
plt.title("t = 0, regions for optimal policy")

plt.figure()
plt.imshow(policy[1, :, :].T, cmap='gray', origin='lower', extent=[-1,1,-1,1])
plt.title("t = 1, regions for optimal policy")

plt.figure()
plt.imshow(policy[2, :, :].T, cmap='gray', origin='lower', extent=[-1,1,-1,1])
plt.title("t = 2, regions for optimal policy")

Z = V[0, :, :].reshape(N, N)
fig = plt.figure()
ax = fig.gca(projection='3d')
surf = ax.plot_surface(X, Y, Z, cmap=cm.coolwarm,
                       linewidth=0, antialiased=False)
plt.title("Optimal Value function V_0")
```

Out[15]: Text(0.5,0.92,'Optimal Value function V_0')




t = 1, regions for optimal policy
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.0
−0.5
0.0
0.5
1.0


t = 2, regions for optimal policy
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.0
−0.5
0.0
0.5
1.0



For region figures, *black* region presents policy 1, where $u_t = 1$ and *white* region represents policy 2, where $u_t = 2$.

The boundary between policy 1 and policy 2 is actually the Intersection of 2 P.D. quadratic functions. For example, if t = 2, the boundary is to solve:

$$\mathbf{x}_2^T (A^T A) \mathbf{x}_2 = \mathbf{x}_2^T (B^T B) \mathbf{x}_2$$

which is

$$\mathbf{x}_2^T (A^T A - B^T B) \mathbf{x}_2 = \mathbf{x}_2^T D \mathbf{x}_2 = \mathbf{0}.$$

which also is

$$0.3125 x_1^2 - 1.5 x_1 x_2 + 1.0625 x_2^2 = 0.$$

Solve for x_2 we will have

$$x_2 = \frac{12 \pm \sqrt{59}}{17} x_1$$

This is the equations of 2 lines, which are consistent with our figures.

As t = 2 to t = 0, assume the action matrix (optimal) we take is $C_t = \{A, B\}$ (left-multiply A or B), we can always find the boundary is to solve

$$\mathbf{x}_t^T C_t^T A^T A C_t \mathbf{x}_t = \mathbf{x}_t^T C_t^T B^T B C_t \mathbf{x}_t$$

$$\mathbf{x}_t^T C_t^T (A^T A - B^T B) C_t \mathbf{x}_t = \mathbf{0},$$

where

$$A^T A - B^T B = \begin{bmatrix} 0.3125 & -0.75 \\ -0.75 & 1.0625 \end{bmatrix}.$$

Therefore, the boundary will change but still keeps linear.

As for optimal value, it is a P.D quadratic function as shown in figure.