# Homework 02

## AA203: Optimal and learning-based Control

Ricardo Luiz Moreira Paschoeto

rp304154@stanford.edu

**Problem 1:** Introduction to Q-learning

a) For learning rate = 0.2

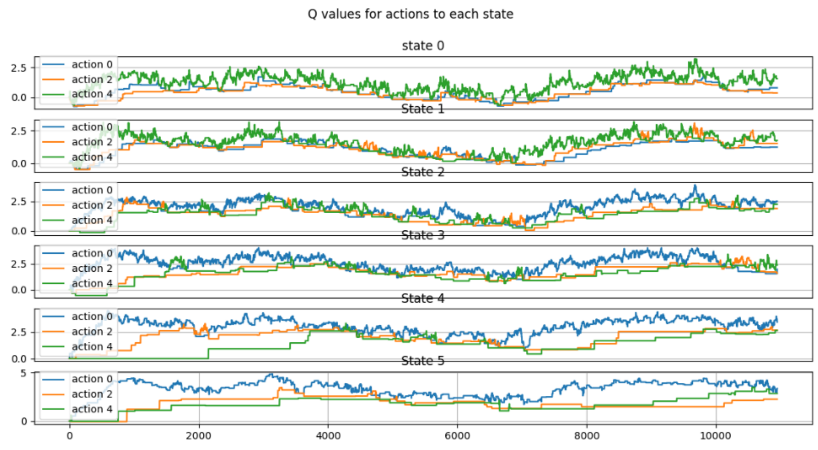

*Figure 1 - Q-values for state-actions*

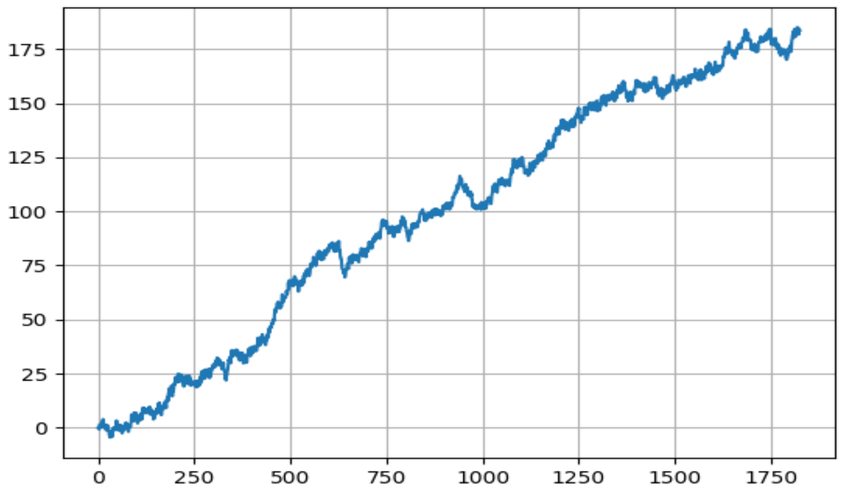

*Figure 2 - Aggregate profit over 5 years.*

Python code:

```python
def policy(state, Q):
    return np.argmax(Q[state])*2
```

```python
def e_greedy(state,Q):
    if np.random.random() < epsilon:
        a = random_policy()
        index = int(a/2)
        return random_policy(), index
    else:
        index = np.argmax(Q[state])
        a = index*2
        return a, index

def q_learning():
    q_values = np.zeros((len(sim.valid_states),len(sim.valid_actions)))
    for n in range(N):
        s = sim.reset()
        for t in range(len(data)):
            x0_hist.append(copy.deepcopy(q_values[0]))
            x1_hist.append(copy.deepcopy(q_values[1]))
            x2_hist.append(copy.deepcopy(q_values[2]))
            x3_hist.append(copy.deepcopy(q_values[3]))
            x4_hist.append(copy.deepcopy(q_values[4]))
            x5_hist.append(copy.deepcopy(q_values[5]))

            a, index = e_greedy(s,q_values)
            sp,r = sim.step(a)
            td = r + gamma*np.max(q_values[sp]) - q_values[s,index]
            q_values[s,index] += alpha*td
            s = sp

    return q_values

def simulation(Q):
    s = sim.reset()
    r_hist.append(0)
    for t in range(T):
        a = policy(s,Q)
        sp, r = sim.step(a)
        r_hist.append(r)
        s = sp
```

b) Result for *Q* from Iteration Values Iteration along 5 years

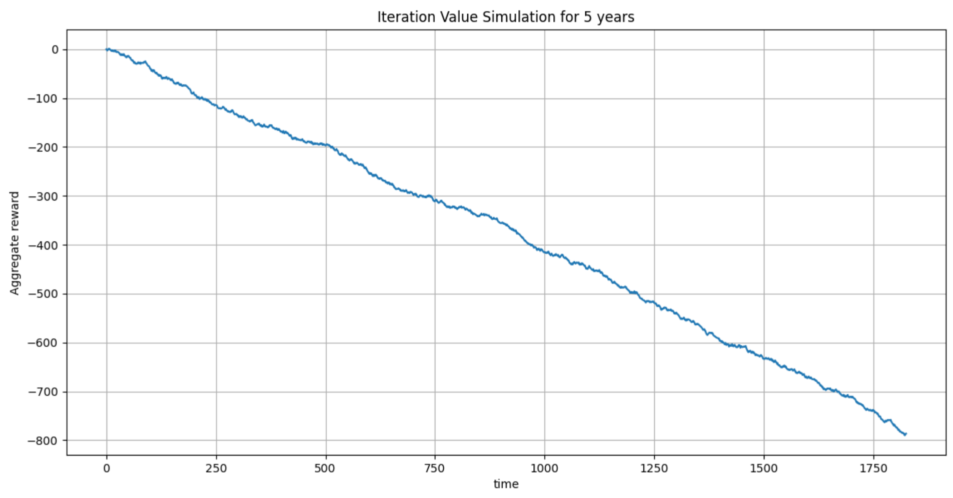

*Figure 3 - Simulation over 5 years*

Python code:

```
iteration = 0
def value_iteration(sim, epsilon=0.001):
    V = np.zeros((len(sim.valid_states)))
    q_values = np.zeros((len(sim.valid_states),len(sim.valid_actions)))

    def next_step(V, x):
        nv = 0
        policy = 0
        for a in sim.valid_actions:
            for d, prob in enumerate(sim.demand_probs):
                next_state = sim.transition(x,a, d)
                r = sim.get_reward(x, a, d)
                v = sim.demand_probs[next_state] * (r + gamma * V[next_st
ate])

                if nv < v:
                    nv = v
                    policy = a
        return nv, policy

    while True:
        delta = 0
        for x in sim.valid_states:
            prev_v = V[x]
            best_v, best_a = next_step(V, x)
            V[x] = best_v
            delta = max(delta, np.abs(prev_v - V[x]))
            q_values[x][int(best_a/2)] = V[x]
        print(delta)
        if delta < epsilon:
            break

    return q_values
```

c) Q-learning presented better results.

After read explanations about this behavior, where the Q-learning perform better than value iteration, my conclusion is how each algorithm works, Q-learning operate over a finite-horizon of five years while the Value Iteration operate over a infinite horizon. One of solutions to help us analyze this outperformance is doing the simulation with bigger values of gamma, forcing the VI policy cares about rewards in distant future. Q-learning already have this behavior, it cares about future reward.

**Problem 2:** Cart-pole swing-up

**a)** *A, B = jax.jacfwd(f, (0, 1))(s, u)*

**,b)**

$$cost = \frac{1}{2}(s_N - s^*)^T Q_N (s_N - s^*) + \frac{1}{2}\sum_{k=0}^{N-1}((s_k - s^*)^T Q (s_k - s^*) + u_k^T R u_k) \text{ (I)}$$

$$\Delta s_N = s_N - s_N^- \Rightarrow s_N = \Delta s_N + s_N^- \text{ (II)}$$

$$\Delta s_k = s_k - s_k^-, \Rightarrow s_k = \Delta s_k + s_k^- \text{ (III)}$$

$$\Delta u_k = u_k - u_k^-. \Rightarrow u_k = \Delta u_k + u_k^- \text{ (IV)}$$

Substituting (II),(III) and (IV) in (I)

$$cost = \frac{1}{2}(\Delta s_N + s_N^- - s^*)^T Q_N (\Delta s_N + s_N^- - s^*) + \frac{1}{2}\sum_{k=0}^{N-1}((\Delta s_k + s_k^- - s^*)^T Q (\Delta s_k + s_k^- - s^*) + (\Delta u_k + u_k^-)^T R (\Delta u_k + u_k^-))$$

Manipulating and considering *R, Q and* $Q_N$ are symmetric,

$$c(\Delta s_k, \Delta u_k) = c_k + c_N + \frac{1}{2}\Delta s_k^T Q_k \Delta s_k + \frac{1}{2}\Delta u_k^T R_k \Delta u_k + \frac{1}{2}\Delta s_N^T Q_N \Delta s_N + q_k^T \Delta s_k + q_N^T \Delta s_N + r_k^T \Delta u_k$$

where,

$$c_k = \frac{1}{2}(s_k^- - s^*)^T Q_k (s_k^- - s^*) + \frac{1}{2}u_k^{-T} R_k u_k^-,$$

$$c_N = \frac{1}{2}(s_N^- - s^*)^T Q_N (s_N^- - s^*),$$

And the terms,

$$q_N^T = (s_N^- - s^*)^T Q_N,$$

$$q_k^T = (s_k^- - s^*)^T Q,$$

$$r_k^T = u_k^{-T} R.$$

c) Python code:

```
    def costs():
        c = np.zeros((N,m)) # immediate state cost
        cs = np.zeros((N, n)) # dc / dx
        cu = np.zeros((N, m)) # dc / du
        cuu = np.zeros((N, m, m)) # d^2 c / du^2
        css = np.zeros((N, n, n)) # d^2 c / dx^2
        cus = np.zeros((N, m, n))  # d^2 c / du / dx == 0 Don't have cros
s terms (s,u)

        for k in range(N-1):
```

```python
            qkT = np.dot((s[k] - s_goal).T, Q) # (sk_bar - s_goal).T*Q
            rkT = u[k] @ R # uk.T*R
            c[k] = (0.5)*np.dot((s[k] - s_goal).T, np.dot(Q, (s[k] - s_go
al))) + (0.5)*np.dot(u[k].T, np.dot(R, u[k])) # (1/2)*(sk_bar - s_star).T
 * Qk * (sk_bar - s_star) + (1/2)*(uk_bar.T * Rk * uk_bar)
            cs[k] = qkT # qk.T
            cu[k] = rkT # rk.T
            cuu[k] = np.array(R) # just R
            css[k] = np.array(Q) # just Q

        qNT = (s[-1] - s_goal).T @ Qf # (sN_bar - s_star).T*Qf
        c[-1] = (0.5)*(s[-1] - s_goal).T @ Qf @ (s[-
1] - s_goal)  # (1/2)*(sN_bar - s_goal).T*QN*(sN_bar - s_goal)
        cs[-1] = qNT # qN.T ????
        css[-1] = Qf # final cost

        return np.array(c), np.array(cs), np.array(cu), np.array(cuu), np
.array(css), cus

    # iLQR loop
    for _ in range(max_iters):
        # Linearize the dynamics at each step `k` of `(s_bar, u_bar)`
        A, B = jax.vmap(linearize, in_axes=(None, 0, 0))(f, s[:-1], u)
        A, B = np.array(A), np.array(B)
        # WRITE YOUR CODE BELOW #########################################
######
        # Update the arrays `L`, `l`, `s`, and `u`.
        # for storing quadratized cost function

        # Foward pass:
        for k in range(N):
            s[k + 1] = f(s[k], u[k])

        # Compute cost
        c, cs, cu, cuu, css, cus = costs()

        # Backward pass:
        v, v_bold, V = c[-1].copy(), cs[-1].copy(), css[-1]
        l = np.zeros((N, m))
        L = np.zeros((N, m, n))
        for k in range(N - 1, -1, -1):
            Qk = c[k] + v
            Qs = cs[k] + (A[k].T @ v_bold)
            Qu = cu[k] + (B[k].T @ v_bold)
            Qss = css[k] + (A[k].T @ V @ A[k])
            Quu = cuu[k] + (B[k].T @ V @ B[k])
            Qus = cus[k] + (B[k].T @ V @ A[k])

            l[k] = -np.linalg.inv(Quu) @ Qu
            L[k] = -np.linalg.inv(Quu) @ Qus

            v = Qk - 0.5*(l[k].T @ Quu @ l[k])
            v_bold = Qs - (L[k].T @ Quu @ l[k])
            V = Qss - (L[k].T @ Quu @ L[k])

        for k in range(N - 1):
            u[k] = u_bar[k] + l[k] + np.dot(L[k], s[k] - s_bar[k])
            s[k + 1] = f(s[k], u[k])
        #################################################################
######
        print(np.max(np.abs(u - u_bar)))
        if np.max(np.abs(u - u_bar)) < eps:
            converged = True
            break
        else:
            u_bar[:] = u
            s_bar[:] = s
    if not converged:
        raise RuntimeError('iLQR did not converge!')
    return s_bar, u_bar, L, l
```

d) Python code:

```
        if simulate_continuous_time_dynamics:
            # WRITE YOUR CODE BELOW #####################################
######
            # Update `u[k]` using the final LQR policy `L`, `l` output by
            # `ilqr` above to track the planned trajectory when we simula
te the
            # continuous-time dynamics.
            u[k] = u_bar[k] + l[k] + np.dot(L[k], s[k] - s_bar[k])
            #############################################################
######
            s[k+1] = odeint(lambda s, t: f(s, u[k]), s[k], t[k:k+2])[1]
```

e) Plot Results:

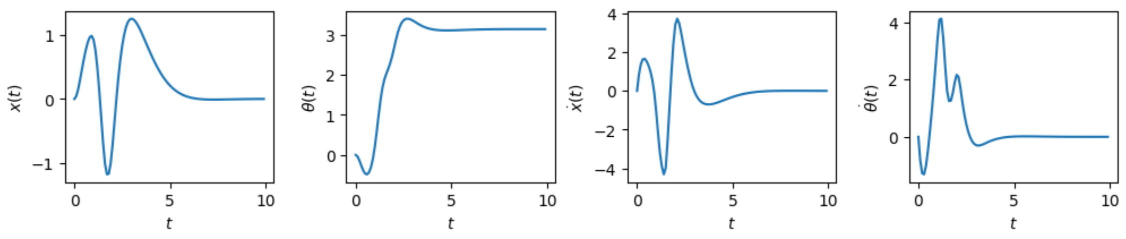

*Figure 4 - Results for cart-pole iLQR*

**Problem 3:** Cart-pole swing-up with limited actuation

a) the discrete dynamic is,

$$s_{k+1} = s_k + \Delta t f(s_k^-, u_k^-),$$

Linearing around $(s_k^-, u_k^-)$,

$$s_{k+1} = \left(s_k^- + \Delta t f(s_k^-, u_k^-)\right) + \left(I_n + \Delta t \frac{\partial f}{\partial s}(s_k^-, u_k^-)\right) + \left(0 + \Delta t \frac{\partial f}{\partial u}(s_k^-, u_k^-)\right),$$

Where,

$$c_k = s_k^- + \Delta t f(s_k^-, u_k^-),$$

$$A_k = \left(I_n + \Delta t \frac{\partial f}{\partial s}(s_k^-, u_k^-)\right),$$

$$B_k = \left(\Delta t \frac{\partial f}{\partial u}(s_k^-, u_k^-)\right).$$

$(DLOCP)_{k+1}$,

$$min_{(s_i,u_i)} \sum_{i=0}^{N-1} dt \left(\frac{1}{2}(s_N - s^*)^T Q_N (s_N - s^*) + \frac{1}{2}\sum_{k=0}^{N-1}((s_k - \bar{s})^T Q (s_k - \bar{s}) + u_k^{-T} R u_k^-)\right)$$

Subject to

$$s_{i+1} = c_i + A_i(s_i - \bar{s}_i) + B_i(u_i - \bar{u}_i), \mathrm{i} = 0, \dots, \mathrm{N} - 1$$

$$u_{lB} \le u_i \le u_{uB}, i = 0, \dots, N-1$$

$$||s_t - \bar{s}||_\infty \le \rho$$

$$||u_t - \bar{u}||_\infty \le \rho$$

$$s(0) = s_0, i = 0$$

$$s_N = s_f, i = N.$$

b) *A, B, c = jax.jacfwd(lambda x:f(x, u))(x), jax.jacfwd(lambda u:f(x, u))(u), f(x, u).*

c)Python code:

```
def scp_iteration(f,Q,R,Q_N,s_bar,u_bar,s_star,s0,N,dt,rho,uLB,uUB)
:

    ####################################################################
############
    # WRITE YOUR CODE HERE
    # implement one iteration of scp
    # HINT: See slides 34-38 of Recitation 1.

    n = Q.shape[0]
    m = R.shape[0]
    S = {}
    U = {}
    u = np.zeros((N, m))
    s = np.zeros((N+1, n))

    #print("s shape = " + str(s.shape))

    cost_terms = []
    constraints = []

    #A, B, c = linearize(f, s_bar, u_bar)

    for t in range(N):

        S[t] = cvx.Variable(n)
        U[t] = cvx.Variable(m)
        cost_terms.append(cvx.quad_form(S[t] - s_star, Q)) # State
cost
        cost_terms.append(cvx.quad_form(U[t], R)) # Control cost

        constraints.append(U[t] <= uUB)
        constraints.append(U[t] >= uLB)

        constraints.append(cvx.norm(U[t] - u_bar[t], "inf") <= rho)
 # Box constraint 1
        constraints.append(cvx.norm(S[t] - s_bar[t], "inf") <= rho)
 # Box constraint 2

        if t == 0: # S[0] == s0 # Initial condition
            constraints.append(S[t] == s0)

        if t > 0:
           A, B, c = linearize(f, s_bar[t - 1], u_bar[t - 1])
           constraints.append(A @ (S[t - 1] - s_bar[t - 1]) + B @ (
U[t - 1] - u_bar[t - 1]) + c == S[t])

    S[t + 1] = cvx.Variable(n)
    A, B, c = linearize(f, s_bar[t], u_bar[t])
    constraints.append(A @ (S[t] - s_bar[t]) + B @ (U[t] - u_bar[t]
) + c == S[t+1])
    cost_terms.append(cvx.quad_form(S[t + 1] - s_star, Q_N))

    obj = cvx.Minimize(cvx.sum(cost_terms))
    problem = cvx.Problem(obj, constraints)
    problem.solve()

    for k in range(len(U)):
       u[k, :] = U[k].value

    for t in range(len(S)):
        s[t, :] = S[t].value

    ###############################################################################
    return s,u
```

d)

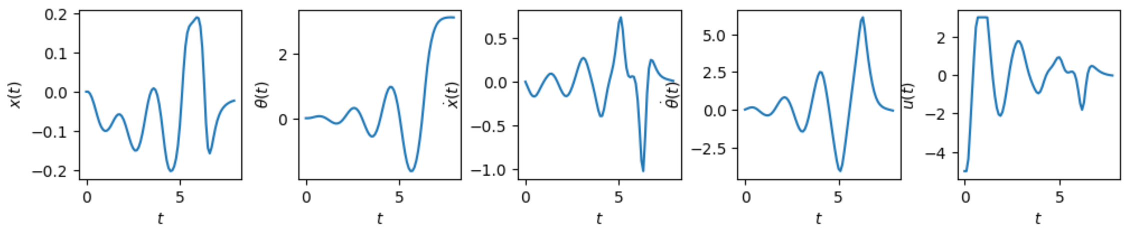

*Figure 5 - Results for cart-pole scp.*