[제어 및 시스템 공학 특론-RL 제어]

과목: 제어 및 시스템 공학 특론

담당교수: 안춘기 교수님

학과: 전자전기공학과

학번: 2024020813

이름: 서윤수

제출일: 2024.12.12

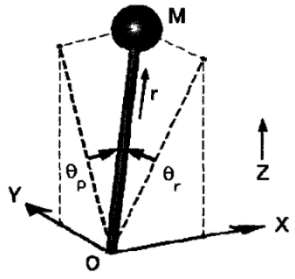
[목차]

1. Linear inverted pendulum 시스템 설계

(1) LIPM이란

Linear inverted pendulum model(LIPM)은 수직으로 서 있는 진자 형태의 시스템을 다루며, 이 시스템은 이상적인 조건에서 진자가 선형적으로 움직인다고 가정한다. 보통 질량 중심이 일정한 높이를 유지하면서 진자가 넘어지지 않도록 제어하는 문제를 해결하는데 사용되는데, 본 연구에서는 이 질량 중심을 휴머노이드의 base로 놓고 base가 수직 방향으로 일정하게 유지하며 보행할 때의 CoM trajectory를 생성하는 데에 관심이 있어 LIPM을 system으로 선택하였다.

(2) LIPM의 state space



[Figure 1]

LIPM 식을 처음 제안한 kajita 논문[1]에 따르면, [Figure 1]과 같은 단일 역진자의 운동방정식을 표현하면 아래와 같이 표현될 수 있다.

$$\ddot{y} = \frac{g}{z_c} y - \frac{1}{m z_c} u_r,$$
$$\ddot{x} = \frac{g}{z_c} x + \frac{1}{m z_c} u_p.$$

높이가 일정하기에 X와 Y축을 독립적으로 볼 수 있으며, 이를 X에 대해서 상태방정식을 discrete time으로 표현하면

$$\hat{x}_{k+1} = \begin{bmatrix} 1 & T & T^2/2 \\ 0 & 1 & T \\ 0 & 0 & 1 \end{bmatrix} \hat{x}_k + \begin{bmatrix} T^3/6 \\ T^2/2 \\ T \end{bmatrix} \dddot{x}_k$$

$$z_k = \begin{bmatrix} 1 & 0 & h_{CoM}/g \end{bmatrix} \hat{x}_k.$$

위와 같이 표현된다[2]. 이때, $\hat{x}$는 위치, 속도, 가속도로 이루어진 3×1 벡터이며, z는 Zero Moment Point(ZMP)를 뜻한다. ZMP 와 CoM 의(여기에서의 $\hat{x}$)위치가 support polygon 안에 위치하면 휴머노이드 보행에 있어 안정하다고 판단할 수 있다. 따라서, 해당 학습에서는 이 LIPM 시스템을 사용하여 initial 위치를 유지하는 서 있는 자세를 유지하는 것을 목표로 진행하였다.

---

[1] Kajita, Shuuji, et al. “The 3D linear inverted pendulum mode: A simiple modeling for a biped walking pattern generation”

[2] Wieber, P. B. “Trajectory free linear model predictive control for stable walking in the presence of strong perturbations”

2. On-policy 제어기

(1) 제어기 설명

[이론]

Reinforce learning에서의 목적은 optimal한 policy를 찾는 것이다. Bellman equation을 사용하여 policy evaluation과 policy improvement를 iterate하게 하면서 최적 policy를 찾을 수 있다. 이때, On-policy는 State-value function을 Bellman equation의 target으로 사용하게 된다. State-value function의 Bellman equation은 아래와 같다.

$$V^*(s) = \max_a \left\{ r(s,a,s') + \gamma \cdot V^*(s') \right\}$$

[Eq 1. - State-value function을 사용한 Bellman equation]

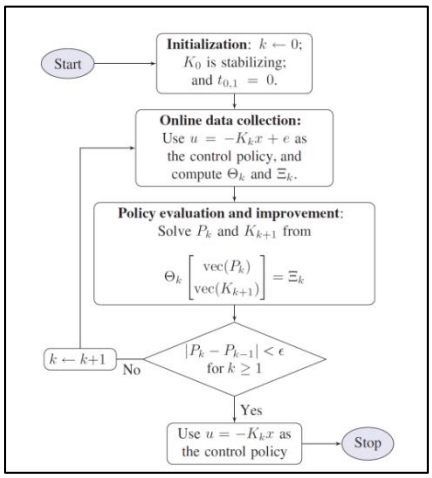
이때의 Optimal policy는 $\max_a \left\{ r(s,a,s') + \gamma \cdot V^*(s') \right\}$ 와 같이 표현되며, 다음 state에 대해 policy에 대한 항을 대입하면 아래와 같이 전개된다.

$$V^{\pi}(s) = r(s,a,s') + \gamma \cdot V^{\pi}(s') = r(s,a,F(s,\pi(s)) + \gamma \cdot V^{\pi}(F(s,\pi(s)))$$

따라서, Value Iteration의 과정을 다시한번 보자면, On-policy에서는 State-value function을 활용한 Bellman equation인 [eq.1]을 사용하여 (1) Value iteration을 진행한 뒤, Optimal policy equation을 활용하여 (2) Policy improvement를 진행한다. 이 과정 후, $|V_{i+1}(s) - V_i(s)| \leq \varepsilon$ 와 같이 다음 step의 state value function이 현재 값과 일정 수준으로 차이가 적어지는지 (3) Convergence check를 진행한다.

이때, On-policy 학습은 target을 state value function을 사용하고, 이는 behavior network(Old)와 같은 정책에서 데이터 수집과 정책 개선을 수행하기에 convergence에 유리하며 빠른 적응성에 장점을 가진다. 하지만, exploration이 작고 데이터 활용이 비효율적이라는 단점을 가진다.

[코드 - 알고리즘]



On-policy는 다음과 같은 Flowchart를 가진다. 이에 대한 과정을 더 자세히 살펴보면 다음과 같다.

1) Initialization

초기화를 하는 과정으로, 초기 제어 이득 k_0를 찾아 A - Bk_0가 Hurwitz한지 확인후 Hurwitz를 이루도록 설정한다. 이때, 처음으로 사용하는 k와 $t_{0,1}$는 0으로 시작한다.

2) Online data collection

시스템 제어 입력을 설정하는 과정으로 데이터 행렬의 각 행을 생성하여 랭크조건이 만족될 때까지 반복한다. 이때 사용되는 식은, $u = -K_k x + e$와 같다. 이때, u는 제어입력, k_k는 현재 정책에 의해 결정된 제어 이득, e는 탐색 노이즈로, 정책 학습을 위한 추가 신호이다.

3) Policy evaluation and improvement

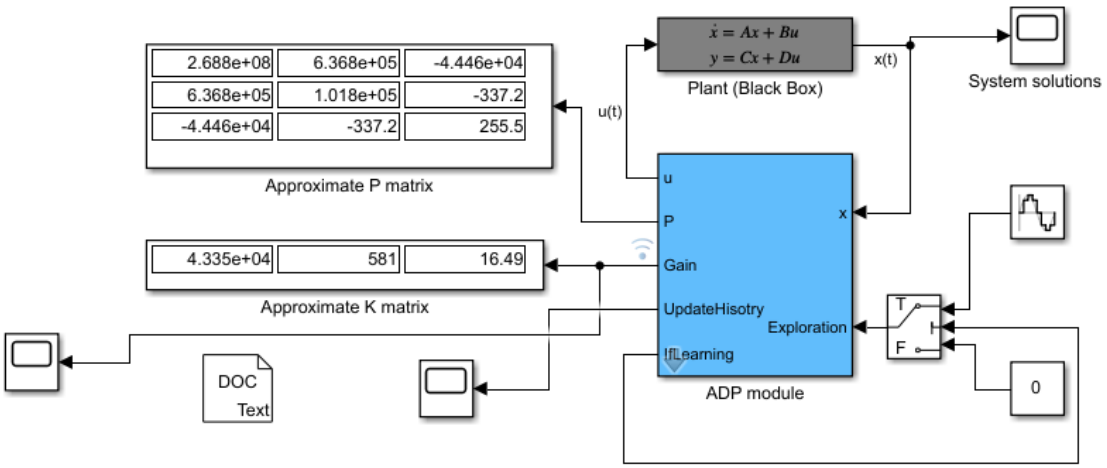
새로운 제어 이득 k_{k+1}과 P_k를 계산하여 현재 정책의 성능을 평가하고 개선된 정책을 계산한다. 이때 사용하는 식은 다음과 같다. $\frac{d}{dt}(x^T P_k x) = -\int_t^{t+\delta t} x^T Q_k x d\tau$

4) Stopping criterion

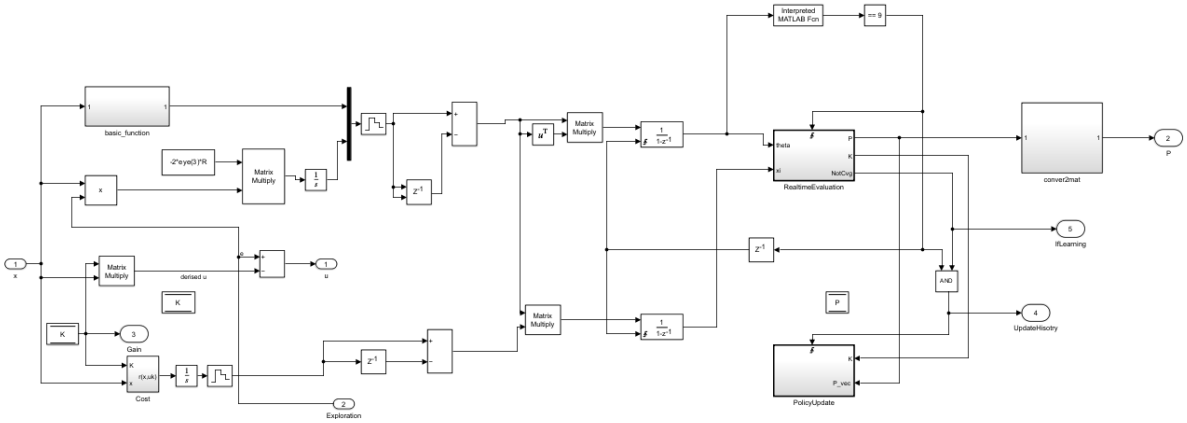
계산되어 나온 P_k가 이전의 값,P_{k-1},과 일정 이하로 차이가 난다면, 수렴했다고 생각할 수 있다. 따라서, $|P_k - P_{k-1}| < \epsilon$ 를 만족한다면 탐색 노이즈 e를 제거하고 제어 입력 u = $-K_k x$를 계산하여 최적 제어입력을 설정한다. 만약, 위 조건을 만족하지 않는다면 2)로 돌아가 반복한다.

[코드 - 시뮬링크]

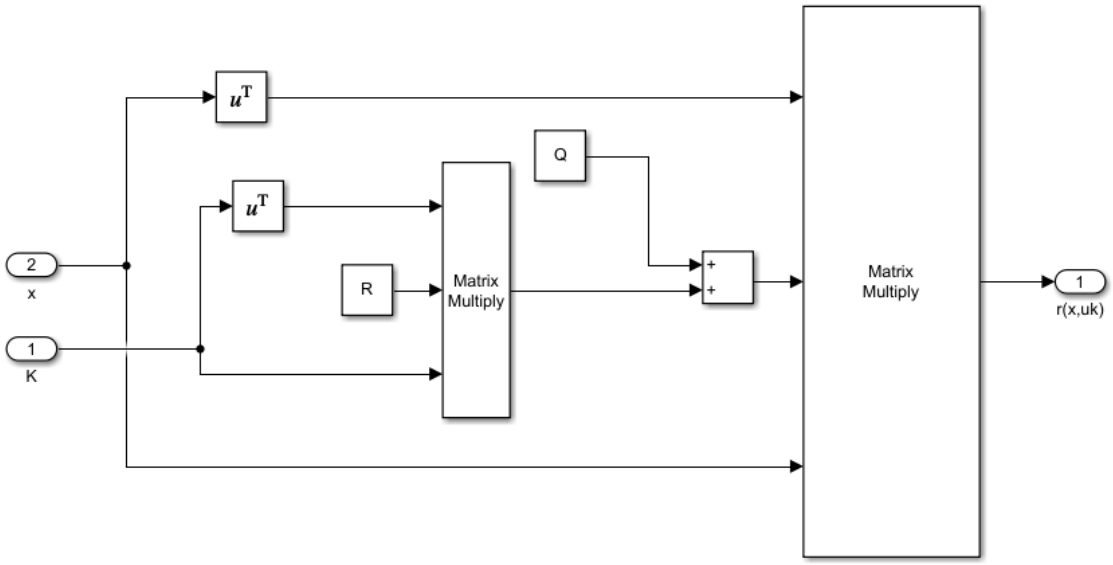
본 코드는 LinearADPSimullink_R2024b를 기반으로 수정하였다. 아래는 전체 시스템에 대한 블록 다이어그램이다.



보면, 사인 파동으로부터 노이즈를 생성하여 현재 상태 x(t)와 함께 ADP module 블록에 들어가고, output으로 나온 u가 plant에 다시 들어가 state를 내뱉으며 iterate하는 모습을 볼 수 있다. 그렇다면 ADP module안은 어떻게 생겼을까? ADP module은 아래와 같은 다이어그램으로 구성된다.



ADP module에서는 1)기본 입력 및 초기화로 input으로 받은 x에 대해 Matrix Multiply를 사용하여 행렬 곱을 수행한다. 이때, basic_function은 알고리즘 초기화를 진행한다. 그 다음, 2) 비용 함수 계산을 진행한다. X와 가중치 K를 사용하여 비용 함수를 계산하는데, 상세한 코드는 아래와 같다.



해당 비용함수 블록에서는 x와 k를 사용하여 제어입력을 계산하여 미리 정해준 가중치 함수들을 활용하여 비용함수 $r(x,u_k) = x^T Q x + u_k^T R u_k$ 를 계산한다. Cost를 계산하였으면 다음은 3)Policy update를 진행한다. Policy 블록에서는 P_vec인 정책 파라미터가 업데이트 되어 상태 및 보상 정보를 바탕으로 새로운 정책을 학습한다. 새로운 Policy를 업데이트했다면, 4)Realtime Evaluation을 통해 현재 정책을 사용하여 시스템 상태를 평가한다. 이때의 결과가 IfLearning과 UpdateHistory 블록으로 넘어가 학습 결과가 조건을 만족하는지의 여부와 학습 과정을 기록하게 되며 output을 내뱉게 된다.

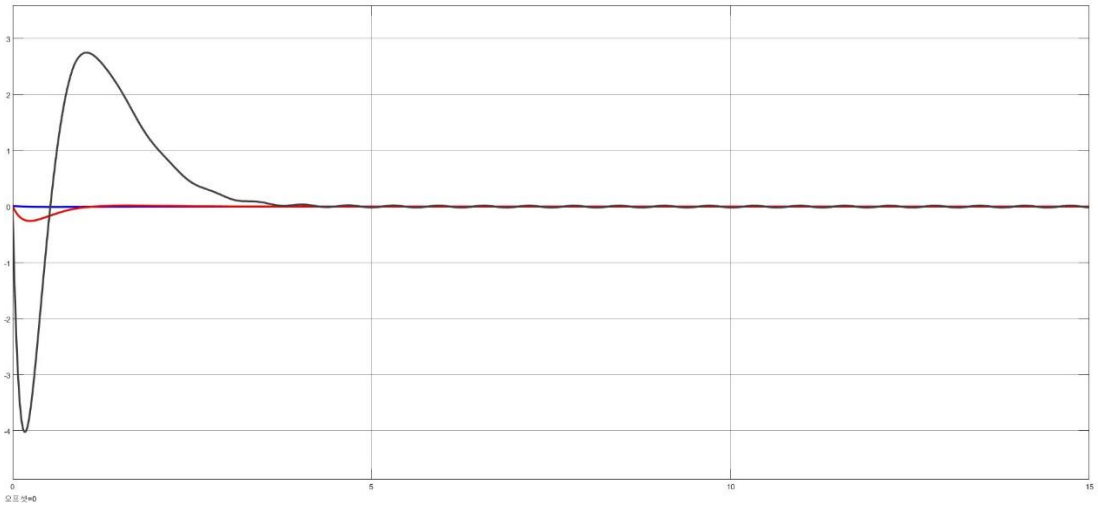
(2) 시뮬레이션 설정

학습을 하기 전, Initialize에서 필수적인 조건이 있었다. $A - Bk_0$가 Hurwitz한지의 여부를 따지는 거였는데, 본 학습에서 사용하려는 시스템은 아래와 같았다.

$$\hat{x}_{k+1} = \begin{bmatrix} 1 & T & T^2/2 \\ 0 & 1 & T \\ 0 & 0 & 1 \end{bmatrix} \hat{x}_k + \begin{bmatrix} T^3/6 \\ T^2/2 \\ T \end{bmatrix} \dddot{x}_k$$

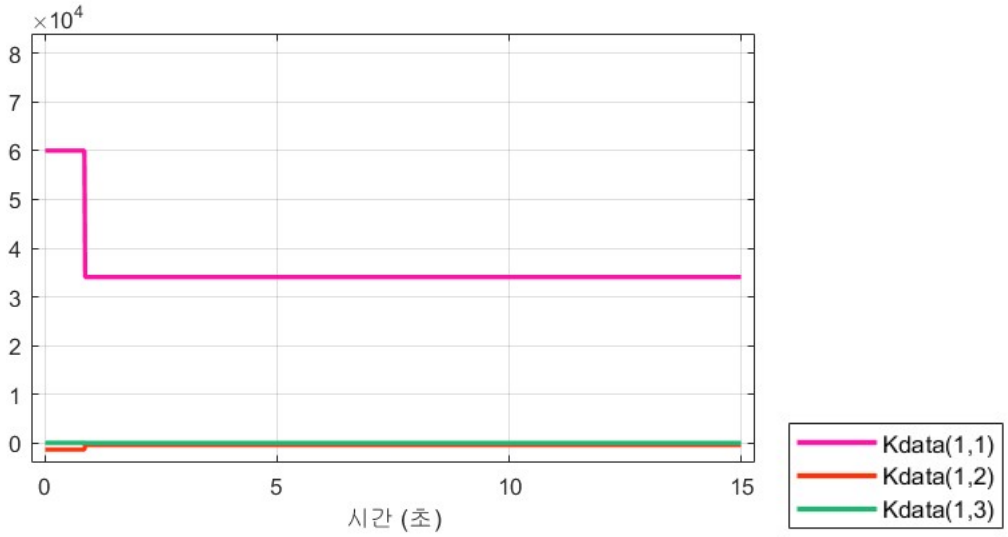
해당 A와 B를 가지고 k_0를 zero로 놓고 Hurwitz한지를 계산하였을 때, 안타깝게도 LIPM은 A가 stable하지 않아서 만족하지 않았다. 따라서, 임의로 closed-loop의 eigenvalue를 {-4, -3, -2}로 놓고(모두 음수로 만들기 위해서) k_0를 [60000 -1300 85]로 설정하고 학습을 진행하였다. 또한, 휴머노이드 보행에 있어 제일 중요한 건 CoM의 위치이므로 위치에 대한 가중치를 높인 Q,R을 사용하여 진행하였다. A와 B가 dT에 영향을 받기에 T는 0.1로 놓고 진행하였다.

(3) 학습 결과



[Figure 2 – 시간에 따른 X변화 그래프]

위 결과는 시간에 따른 X의 변화 그래프로, 초기 상태(0,0,0)에서 진동을 포함한 과도응답을 보이다가 시간이 지남에 따라 점차 안정되어 0으로 수렴하는 모습을 보여준다. 따라서, 목표 상태에 도달했기에 정책 학습이 성공적이었다고 판단할 수 있다.



[Figure 3 – 시간에 따른 K변화 그래프]

[Figure 3]은 시간에 따른 제어 매트릭스 K 의 변화로, 초기 state 인 [60000 -1300 85]에서 시작하여 수렴하는 모습을 보인다. 위 값을 보아 핑크색 궤적인 위치를 다루는 gain 값이 이 제어기의 주된 수정 대상이었음을 알 수 잇고, 이후에 변동이 거의 없음을 보아 안정적으로 제어되었다고 판단할 수 있다.

[Figure 2], [Figure 3]에서 모두 수렴 현상이 관찰되며, 시스템 상태 x 가 안정되고 제어 매트릭스 K 가 학습 완료 후 일정한 값을 유지하는 것을

확인하였다. 이는 시스템이 적절한 제어를 받고 있다는 것을 나타낸다. [Figure 2]에서는 진동이 초기 몇 초간 존재하나 비교적 짧은 시간내에 감쇠하고 0 에 수렴하는 것을 볼 수 있다.

3. Off-policy 제어기

(1) 제어기 설명

[이론]

Reinforce learning에서의 목적은 optimal한 policy를 찾는 것이다. Bellman equation을 사용하여 policy evaluation과 policy improvement를 iterate하게 하면서 최적 policy를 찾을 수 있다. 이때, Off-policy는 Action-value function을 Bellman equation의 target으로 사용하게 된다. Action-value function의 Bellman equation은 아래와 같다.

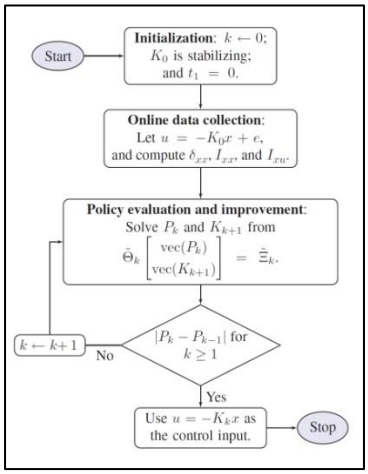
$$Q^*(s,a) = r(s,a,s') + \gamma \cdot \max_{a'}\{\, Q^*(s',a') \,\}$$

[Eq 2. - Action-value function을 사용한 Bellman equation]

이때의 Optimal policy는 $\max_{a'}\{ Q^*(s',a') \}$ 와 같이 표현된다. Off-policy도 On-policy와 마찬가지로 Value Iteration 과정을 거치는데, Off-policy에서는 Action-value function을 활용한 Bellman equation인 [eq.1]을 사용하여 (1) Value iteration을 진행한 뒤, Optimal policy equation을 활용하여 (2) Policy improvement를 진행한다. 이 과정 후, $|V_{i+1}(s) - V_i(s)| \leq \varepsilon$ 와 같이 다음 step의 Action value function이 현재 값과 일정 수준으로 차이가 적어지는지 (3) Convergence check를 진행한다.

Off-policy는 현재 정책과 독립적인 데이터를 기반으로 학습을 진행하기에 기존 정책 외의 데이터로 최적 정책을 탐색할 수 있다. 이는, 더 넓은 exploration area를 가진다는 뜻으로, 글로벌 최적 정책을 학습할 가능성이 높다. 그러나, 잘못된 데이터나 노이즈가 포함된 데이터를 재사용하면 성능에 악영향을 미치며 수렴속도가 On-policy보다 느리다는 단점이 있다.

[코드 - 알고리즘]



Off-policy는 다음과 같은 Flowchart를 가진다. 이에 대한 과정을 더 자세히 살펴보면 다음과 같다.

1) Initialization

초기화를 하는 과정으로, 초기 제어 이득 k_0를 찾아 A - Bk_0가 Hurwitz한지 확인후 Hurwitz를 이루도록 설정한다. 이때, 처음으로 사용하는 k는 0으로 시작한다.

2) Online data collection

시스템 제어 입력을 설정하는 과정으로 데이터 행렬의 각 행을 생성하여 랭크조건이 만족될 때까지 반복한다. 이때 사용되는 식은, $u_0 = -K_0x + e$ 와 같다. 이때, 데이터 행렬을 계산하여 k_{k+1}과 P_k을 할 수 있도록 한다.

3) Policy evaluation and improvement

새로운 제어 이득 k_{k+1}과 P_k를 계산하여 현재 정책의 성능을 평가하고 개선된 정책을 계산한다.

4) Off-policy iteration & Exploitation

계산되어 나온 P_k가 이전의 값,P_{k-1},과 일정 이하로 차이가 난다면, 수렴했다고 생각할 수 있다. 따라서, $|P_k - P_{k-1}| < \epsilon$ 를 만족한다면 탐색 노이즈 e를 제거하고 제어 입력 u = $-K_kx$를 계산하여 최적 제어입력을 설정한다. 만약, 위 조건을 만족하지 않는다면 2)로 돌아가 반복한다.

[코드 - matlab]

본 코드는 Robust Adaptive Dynamic Programming의 chapter 2의 Example 2.2 Off-policy learning for a turbocharged disel engine을 기반으로 LIPM 시스템에 적용하여 진행하였다. 다음은 전체 코드 중 중요한 부분들만 과정별로 설명이다.

1) 문제 설정 및 초기화

```
xn = 3; un = 1; % 상태 및 제어 입력 차원
Q = diag([100 0 0]); R = diag([1]); % 비용 함수 가중 행렬
Kinit = [8 0.05 0.0]*10000; % 초기 이득 행렬
x0 = [0.01; 0.01; 0.01]; % 초기 상태
T = 0.1; % 샘플링 시간
```

상태와 제어입력의 차원을 정의하는 코드로, Q와 R은 상태 가중치, 제어 입력 가중치이다. 초기 제어 이득 행렬 Kinit은 시스템이 stable할 경우에만 0으로 설정할 수 있기에, stable하게 만드는 Kinit을 계산하여 설정해주었다.

2) 데이터 수집

```
for i=1:N
    [t,X] = ode45(@(t,x)aug_sys(t,x,K,ifLearned,expl_noise_freq), ...
                  [(i-1)*T, i*T], X(end,:));
    Dxx = [Dxx; kron(X(end,1:xn),X(end,1:xn)) - kron(X(1,1:xn),X(1,1:xn))];
    Ixx = [Ixx; X(end,xn+1:xn+xn^2) - X(1,xn+1:xn+xn^2)];
    Ixu = [Ixu; X(end,xn+xn^2+1:end) - X(1,xn+xn^2+1:end)];
    x_save = [x_save; X]; t_save = [t_save; t];
end
```

이 루프는 시스템을 돌리고, 오프라인 학습에 필요한 데이터(Dxx,Ixx,Ixu)를 수집하는 코드이다. 이때, ode45는 주어진 시스템 동역학(aug_sys)을 기반으로 상태 궤적을 적분한다.

3) Off-Policy 학습

```
while norm(P-P_old) > 1e-8 & it < MaxIteration
    it = it + 1;
    QK = Q + K'*R*K; % 갱신된 비용 함수 가중 행렬
    Theta = [Dxx, -Ixx*kron(eye(xn), K') - Ixu];
    Xi = -Ixx*QK(:);
    pp = pinv(Theta)*Xi;
    P = reshape(pp(1:xn*xn), [xn, xn]); P = (P + P')/2;
    BPv = pp(end-(xn*un-1):end);
    K = inv(R) * reshape(BPv, un, xn) / 2;
```

```
    p_save = [p_save, norm(P-Popt)];
    k_save = [k_save, norm(K-Kopt)];
end
```

시스템 비용 행렬P와 제어 이득 K를 업데이트하는 학습 루프로, 2)에서 수집한 데이터를 바탕으로 주요 방정식을 최소자승법으로 풀어 P와 K를 추정한다. 이때, 비용 및 제어 이득의 수렴도를 추적하여 학습 성능을 평가한다.

4) 결과 시뮬레이션 및 plotting

```
figure(1)
subplot(211)
plot(0:length(p_save)-1, p_save, 'o', 0:length(p_save)-1, p_save, 'Linewidth', 2);
legend('||P_k-P^*||'); xlabel('Number of iterations');

subplot(212)
plot(0:length(k_save)-1, k_save, '^', 0:length(k_save)-1, k_save, 'Linewidth', 2);
legend('||K_k-K^*||'); xlabel('Number of iterations');

figure(2)
plot(t_final, x_final(:,1:3), 'Linewidth', 2);
legend('x_1', 'x_2', 'x_3'); xlabel('Time (sec)');
```

학습 결과의 수렴도를 그래프로 나타내는 코드이다.

(2) 시뮬레이션 설정

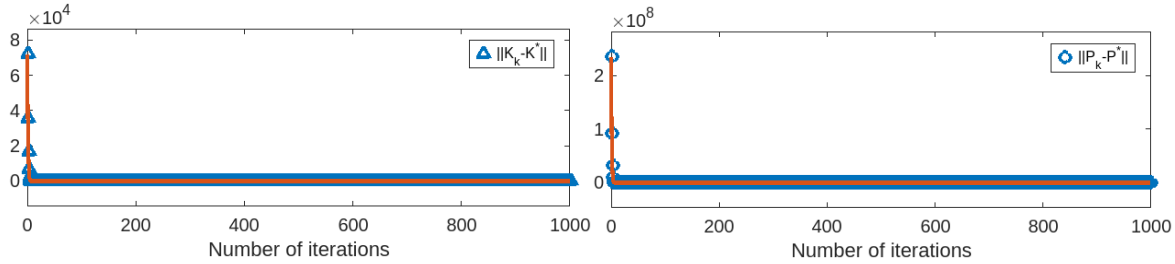
학습을 하기 전, Initialize에서 필수적인 조건이 있었다. $A-Bk_0$가 Hurwitz한지의 여부를 따지는 거였는데, 본 학습에서 사용하려는 시스템은 아래와 같았다.

$$\hat{x}_{k+1} = \begin{bmatrix} 1 & T & T^2/2 \\ 0 & 1 & T \\ 0 & 0 & 1 \end{bmatrix} \hat{x}_k + \begin{bmatrix} T^3/6 \\ T^2/2 \\ T \end{bmatrix} \dddot{x}_k$$

해당 A와 B를 가지고 k_0를 zero로 놓고 Hurwitz한지를 계산하였을 때, 안타깝게도 LIPM은 A가 stable하지 않아서 만족하지 않았다. 따라서, 임의로 closed-loop의 eigenvalue를 {-4, -3, -2}로 놓고(모두 음수로 만들기 위해서) k_0를 [60000 -1300 85]로 설정하고 학습을 진행하였다. 또한, 휴머노이드

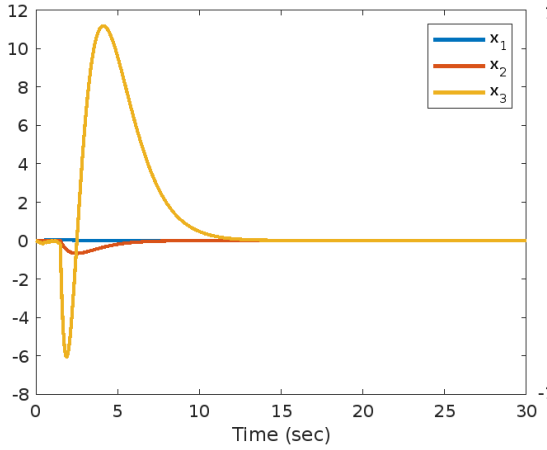
보행에 있어 제일 중요한 건 CoM의 위치이므로 위치에 대한 가중치를 높인 Q,R을 사용하여 진행하였다. A와 B가 dT에 영향을 받기에 T는 0.1로 놓고 진행하였다. N과 같은 경우, 보행의 CoM trajectory generator를 MPC로 풀 때 자주 사용하는 1.5sec을 기준으로 하였기에 15로 설정하였다.

(3) 학습 결과

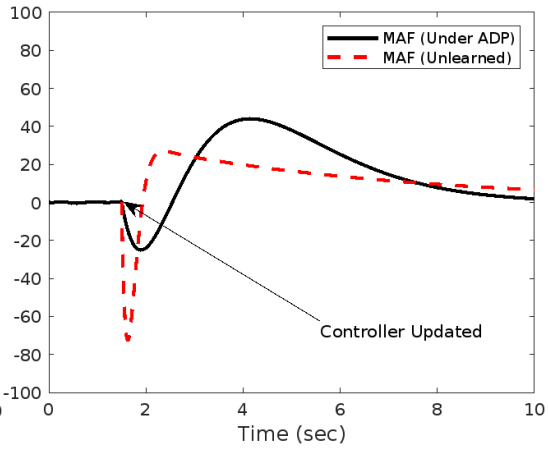


[Figure 4 – 제어 이득과 비용 행렬의 차이 비교]

X축은 학습 반복 횟수, Y축은 각각 제어 이득의 추정값과 최적값 사이의 차이, 비용 행렬의 추정값과 최적값의 차이를 나타내는 그래프이다. 두 그래프 모두 초기에는 큰 오차를 보이지만, 학습을 반복하며 빠르게 감소하는 모습을 확인할 수 있다. 5번 정도의 반복 이후, LQR을 통해 구한 최적 제어 이득과 비용 행렬값과 학습을 통한 추정값의 차이가 0으로 수렴한 것을 확인할 수 있다. 이는 학습 알고리즘이 효율적으로 작동하여 최적의 제어 이득에 도달했으며, 시스템 비용 함수에서 최적의 성능을 학습했음을 나타낸다.



[Figure 5 – 상태 변수 그래프]



[Figure 6 – 위치 비교 그래프]

두 그래프 모두 x축은 시간에 대한 항이고, [Figure 5]의 y축은 상태 변수(위치, 속도, 가속도)를, [Figure 6]의 y축은 위치에 대한 학습값과 학습하지 않은 값을 나타낸다. [Figure 5]를 확인하면, 초기에는 진동이 심하지만, 학습된 제어기를 적용한 후 모두 안정적으로 0에 수렴하는 모습을 확인할 수 있다. 특히, 가속도가 다른 상태 변수들에 비해 큰 진동을 보이나, 이는 dT가 0.1이기에 더 급격히 변화해야 하기 때문일 가능성이 크다. 가속도 또한 0으로 수렴하는 모습을 보아 시스

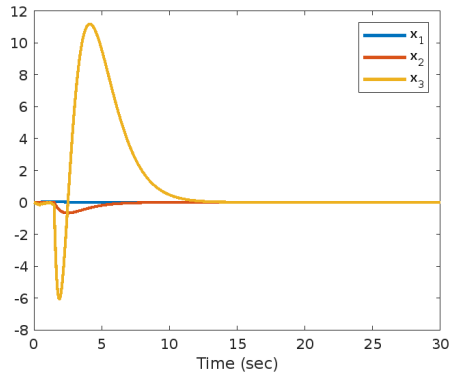
템의 안정성이 학습된 제어기 내에서 보장된다는 것을 확인할 수 있다. [Figure 6]은 학습된 제어기와 학습되지 않은 초기 제어기를 통해 학습된 제어기가 더욱 효율적이고 안정적이라는 것을 확인할 수 있다. 'Controller Updated'부분은 설정한 T=0.1, N=15에 따라 1.5가 지나면서 학습한 제어기과 초기값을 유지하는 제어기가 차이가 나기 시작하며, 학습된 제어기가 학습되지 않은 제어기에 비해 더 빠르게 응답하며 진동 없이 안정적인 궤적을 유지한다. 결과적으로, 학습된 제어기가 더욱 효율적이고 안정적인 시스템 제어가 가능하다는 것을 확인할 수 있다.

4. On-policy vs. Off-policy

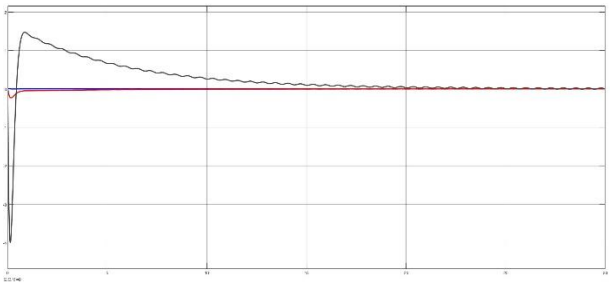
(1) 이론적 비교

On-policy는 현재 사용 중인 정책으로 학습하여, 동일한 정책에서 데이터 수집과 정책 개선을 수행한다. 그에 반해, Off-policy는 목표 정책과 행동 정책이 독립적으로 작동하며, 이로 인해 다양한 행동 정책 데이터를 기반으로 학습을 진행한다. 코드로 조금 더 자세히 보면, Off-policy는 On-policy와 달리 step2에서 data collection을 하는 과정에 있어서 u_0가 step3에 사용되는 K_k가 서로 다르다. 이렇기에 Off-policy는 더 큰 탐색 범위를 가질 수 있다. 또한, Off-policy가 샘플 재사용이 가능하므로 샘플 효율성이 높다. 하지만, 중요도 샘플링 등 추가적인 계산이 필요하여 연산비용이 많이 발생하고 데이터 품질에 민감하다는 단점이 있다.

(2) 시뮬레이션 비교



[Figure 7 – Off policy의 상태 변화 그래프]



[Figure 8 – On policy의 상태 변화 그래프]

다음은 각각 Off policy와 On policy의 상태 변화 그래프이다. 똑같은 30초에 대해서 비교해보았을 때, On policy가 Off policy보다 더 빠르게 수렴하는 모습을 확인할 수 있다. 이는, On policy가 실시간으로 데이터를 수집하고 제어이득을 업데이트하기에 더 빠르게 최적 값을 찾을 수 있기 때문이다. 하지만, On policy는 수렴 후 Off policy에 비해 진동이 계속 일어나는 모습을 관찰할 수 있었다. 이는 학습 중 발생하는 노이즈로 인해 생길 수 있다고 추정했다.

```
Approximate Cost Matrix

P =

   1.0e+07 *

    3.2000    0.0000    0.0027
    0.0000    0.0080    0.0000
    0.0027    0.0000    0.0000

Optimal Cost Matrix

Popt =

   1.0e+07 *

    3.2000    0.0000    0.0027
    0.0000    0.0080   -0.0000
    0.0027   -0.0000    0.0000
```

```
Approximate Gain Matrix

K =

   1.0e+03 *

    8.0000    0.4000    0.0267

Optimal Gain Matrix

Kopt =

   1.0e+03 *

    8.0000    0.4000    0.0267
```

[Figure 9 – Off policy에서의 P,K 추정값 및 LQR로 구한 P,K optimal값]

2.688e+08	6.367e+05	-4.446e+04
6.367e+05	1.018e+05	-337.2
-4.446e+04	-337.2	255.5

Approximate P matrix

4.336e+04	580.9	16.49

[Figure 10 – On policy에서의 P, K 추정값]

다음은 On-policy와 Off-policy의 P, K 추정값과 최적값을 비교해보았다. 두 학습법 모두 최적 비용 행렬과 최적 제어 이득값과 유사한 값을 얻을 수 있었다. Off-policy의 경우, LQR로 구한 최적값과 일치하였으며 이는 Max iteration을 100000으로 설정한 영향이 미쳤을 것이라고 생각한다. 반면, On-policy는 Off-policy보다 빠르게 수렴하였으나, P, K의 추정값에 있어서 최적값과 상당한 차이를 보였다. 이는 실시간 학습에서 발생하는 노이즈랑 에러 값 때문으로 보이며 이로인해 [Figure 8]과 같은 진동이 발생하였을 것이라고 추측된다.

5. Conclusion

Linear inverted pendulum model을 가지고 Off-policy와 On-policy 학습을 진행하였다. LIPM은 휴머노이드 보행에 있어서 가장 기본적으로 자주 사용하는 모델로 단순한 시스템에 비해 안정성이 보장되어 있지 않다는 단점을 가지고 있다. 하지만, 시스템이 stable하지 않아도 두 학습법 모두 제어 이득의 초기값을 잘 설정하여 $A - Bk_0$가 Hurwitz하도록 설정하였더니 모두 잘 수렴하는 것을 확인하였다.

두 학습을 진행하며, Off,On 모두 gain값에 대해 상당히 민감하다는 것을 알 수 있었다. 특히, On과 같은 경우, K의 initial값을 최적값이 아닌 다른 값을 넣을 때, Q에 대해 속도를 우선으로 주게 되는 등 gain을 다르게 하였을 때 발산하는 횟수가 많았고 이는 기존 데이터를 사용하지 않고 이전 값을 바로 사용하기에 생기는 문제라고 생각했다. Off-policy같은 경우, 초기 수렴 속도는 다른 gain값으로 비교하였을 때도 On에 비해 느렸었다. 하지만, Max iteration을 크게 주고 멀리 보았을 때 On에 비해 수렴하는 속도가 느리더라도 수렴하는 모습을 보였었다. 또한, Off의 경우 시스템 모델을 알고 있을 때, LQR과 같은 값이 나오는 값이 나왔기에 On보다 높은 정확성을 지닌다고 생각할 수 있을 것 같다.

한 자세를 유지하거나, Reference가 변화하지 않는다면 On policy가 수렴속도가 빠르기에 사용하는 것이 적합해 보인다. 하지만, 보행에서 사용함에 있어 두 ZMP의 reference가 계속하여 바뀔 텐데, 탐색 범위가 작은 On policy가 발산하지 않고 수렴할지에 대해서는 우려되는 부분이었다. 이번에 두 학습을 진행해보면서 각 task에 따라 다르게 학습 방법을 적용하는 것이 좋을 것이라고 판단할 수 있었다. 또한, deep network를 적용하여 학습하는 경우도 있다고 하던데 이와 ADP의 차이는 무엇이고 각 장단점 등이 궁금해져 앞으로도 지속적인 관심을 기울일 것 같다.