지금부터 메인 코딩을 시작할거야 변수는 : load / pv / price buy / price sell / DT 이고 24시간으로 나누어 에너지 흐름과 비용을 계산 E=bat params 로 두고 E max E init P charge max P discharge max , ㅎ율등을 초기로 설정 , 순 부하는 netload = p load - p pv 로 계산할거야 (양수면 부족, 음수면 남음) 전략 A는 '자가사용'으로 남을때 충전 (최소로) 부족 (방전) 충전 후 남으면 판매 , 방전 후 모자라면 구매 , 전략 B는 전기 비용이 싸면 (최대속도로 ,부하 + 충전량 만큼 그리드에서 구매 충전) , 비싸면 방전(이 때 배터리 판매는 하지 않고 부족한 양만 채우도록 제한) 나머지는 A와 같은 전략 , 배터리 상태 업데이트는 차분방정식 Enext = E + (eta ch * Pch- P dis / eta_dis) *DT로 적용하고 0 <E next < E max P grid>0, 구매가격을 곱해서 비용 더하고 P grid <0이면 판매가격을 곱해 수익 비용으로 처리 그다음 딕셔너리 만들어서 반환 값들 저장해줘

회원님, 요청하신 시뮬레이션 로직을 완벽하게 구현한 **메인 코딩(Core Simulation Function)**입니다.

말씀하신 **모든 제약 조건(변수명, 차분 방정식 형태, 전략 B의 방전 제한 등)**을 그대로 반영했습니다. 이 코드를 복사해서 `main.py` 에 붙여넣으시면 됩니다.

Python

```python
def simulate(strategy_name, bat_params):
    # 1. 초기 설정 (Initialization)
    E = bat_params["E_init"]  # 초기 배터리 에너지 (kWh)
    E_max = bat_params["E_max"]
    P_ch_max = bat_params["P_ch_max"]
    P_dis_max = bat_params["P_dis_max"]
    eta_ch = bat_params["eta_ch"]
    eta_dis = bat_params["eta_dis"]

    # 결과 기록용 리스트
    soc_history = []       # 배터리 잔량 (SOC)
    grid_buy_history = []  # 그리드 구매량
```

```
grid_sell_history = [] # 그리드 판매량
cost_history = []      # 누적 비용

# 누적 변수
total_cost = 0.0
total_buy_kwh = 0.0
total_sell_kwh = 0.0

# 2. 24시간 루프 (Time-Step Loop)
for t in range(24):
    # 현재 시간 데이터 로드
    P_load = load[t]
    P_pv = pv[t]

    # 순부하 계산 (Net Load Calculation)
    # (+) 양수: 부족(Deficit), (-) 음수: 남음(Surplus)
    net_load = P_load - P_pv

    P_ch = 0.0
    P_dis = 0.0
    P_grid = 0.0

    # ------------------------------------------------------
    # 전략 A: 자가 소비 (Self-Consumption)
    # ------------------------------------------------------
    if strategy_name == "A_Self_Consumption":
        if net_load < 0:  # 남을 때 (Surplus) -> 충전
            surplus = -net_load
            # 충전량 결정: (남은양, 최대속도, 배터리 빈공간) 중 최소값
            # 빈공간 채우기: (E_max - E) / eta_ch (입력 기준)
            P_ch = min(surplus, P_ch_max, (E_max - E) / eta_ch)

            # 충전하고도 남은 건 판매
            P_grid = -(surplus - P_ch) # (-)는 판매(Export)

        else:  # 부족할 때 (Deficit) -> 방전
            deficit = net_load
            # 방전량 결정: (부족분, 최대속도, 배터리 잔량) 중 최소값
            # 잔량 쓰기: E * eta_dis (출력 기준)
            P_dis = min(deficit, P_dis_max, E * eta_dis)

            # 방전하고도 부족하면 구매
            P_grid = deficit - P_dis # (+)는 구매(Import)

    # ------------------------------------------------------
    # 전략 B: 가격 기반 최적화 (Price-Based Optimization)
```

```
# ------------------------------------------------------------
elif strategy_name == "B_Price_Optimized":
    curr_price = price_buy[t]

    # 1) 가격이 쌀 때 (Cheap): 강제 충전
    if curr_price < 2.5:
        P_dis = 0.0
        # 최대 속도로 충전 (배터리 빈 공간 고려)
        P_ch = min(P_ch_max, (E_max - E) / eta_ch)

        # 그리드 구매량 = 부하 + 충전량 - 발전량
        P_grid = P_load + P_ch - P_pv

    # 2) 가격이 비쌀 때 (Expensive): 강제 방전 (단, 판매 금지)
    elif curr_price > 4.0:
        P_ch = 0.0
        # 핵심 제약: 오직 부족한 양(deficit)만 채운다. (배터리 판매
        deficit = max(0.0, net_load)

        P_dis = min(deficit, P_dis_max, E * eta_dis)
        P_grid = deficit - P_dis

    # 3) 그 외 (Normal): 전략 A와 동일 (자가 소비)
    else:
        if net_load < 0: # Surplus
            surplus = -net_load
            P_ch = min(surplus, P_ch_max, (E_max - E) / eta_ch)
            P_grid = -(surplus - P_ch)
        else: # Deficit
            deficit = net_load
            P_dis = min(deficit, P_dis_max, E * eta_dis)
            P_grid = deficit - P_dis

# ------------------------------------------------------------
# 3. 배터리 상태 업데이트 (Difference Equation)
# ------------------------------------------------------------
# E_next = E + (충전효율*Pch - Pdis/방전효율) * DT
E_next = E + (eta_ch * P_ch - P_dis / eta_dis) * DT

# 물리적 한계 고정 (Clamping): 0 ~ E_max 사이 유지
E_next = max(0.0, min(E_max, E_next))

# ------------------------------------------------------------
# 4. 비용 계산 (Cost Accounting)
# ------------------------------------------------------------
if P_grid > 0: # 구매 (Import)
```

```
            step_cost = P_grid * price_buy[t] * DT
            total_buy_kwh += P_grid * DT
            grid_buy_history.append(P_grid)
            grid_sell_history.append(0.0)
        elif P_grid < 0: # 판매 (Export)
            step_cost = P_grid * price_sell[t] * DT # 음수 값(수익)
            total_sell_kwh += (-P_grid) * DT
            grid_buy_history.append(0.0)
            grid_sell_history.append(-P_grid)
        else:
            step_cost = 0.0
            grid_buy_history.append(0.0)
            grid_sell_history.append(0.0)

        total_cost += step_cost

        # 기록 저장
        soc_history.append(E_next)
        cost_history.append(total_cost)

        # 다음 스텝을 위해 상태 갱신
        E = E_next

    # 5. 결과 반환 (Return Dictionary)
    return {
        "total_cost": total_cost,
        "buy_kwh": total_buy_kwh,
        "sell_kwh": total_sell_kwh,
        "soc": soc_history,
        "buy_flow": grid_buy_history,
        "sell_flow": grid_sell_history,
        "cost_history": cost_history
    }
```

코드 구현 포인트 (요청사항 반영 확인)

1. **순부하 계산:** `net_load = P_load - P_pv` 로 정확히 계산했습니다.
2. **전략 A (자가소비):** `min()` 함수를 사용하여 잉여 시 충전, 부족 시 방전 로직을 간결하게 구현했습니다.
3. **전략 B (가격기반):**
   - `price < 2.5` : 최대 속도로 충전 (`P_load + P_ch - P_pv` 구매).

- `price > 4.0` : **"배터리 판매 금지"** 조건을 위해 `deficit = max(0.0, net_load)` 로 제한을 걸어두었습니다.

4. **차분 방정식:** `E + (eta_ch * P_ch - P_dis / eta_dis) * DT` 공식을 그대로 적용했습니다.
5. **비용 계산:** `P_grid` 의 부호(+/-)에 따라 구매/판매 가격을 각각 곱해서 처리했습니다.