SAR - SAR 시뮬레이션 RCS 활용

좋아. 지금 구조(phase history → RC → BP) 그대로 두고,

1. **coarse BP(저해상도/작은 그리드)** 로 타겟 위치를 대충 찾고
2. 그 위치를 기준으로 **pulse 전체에 대해 ΔR 범위를 계산**해서
3. **range gating(ΔR min/max + margin)** 을 자동으로 잡은 뒤
4. high-res BP를 gate 걸어서 다시 돌리는

파이프라인을 코드로 붙여줄게.

아래 코드는 **당신의 기존 BP 정의**(`rng = |sensor - pixel| - r0`)에 맞춰서 "gate"를 잡는다.
(즉, gate는 **ΔR 공간(range_window 좌표)** 에서 적용됨)

---

1) 핵심 아이디어

Coarse 이미지로 타겟 추정

- `nx_c, ny_c` 를 작게(예: 96×96) 잡아 빠르게 BP
- `|img|` 가 최대인 위치를 타겟 후보로 선택

그 후보 위치로부터 pulse별 예상 ΔR 계산

$$\Delta R_p = \| s_p - x_{\text{est}} \| - r_0$$

- 여기서 `r0 = self.radius` (당신 코드의 range_center와 동일 의미)

gate 자동 생성

- pulse 전체에서 `ΔR_p` 가 변화하므로
- 전 pulse를 포함하는 범위로

[r_\min, r_\max] = [\min(\Delta R_p)-m, \max(\Delta R_p)+m]

- margin `m` 은 range resolution과 보간 안전 여유로 (예: 2~10 m) 추천

---

2) 클래스에 추가할 함수들 (그대로 붙여넣기)

아래는 `SARSimulate` (혹은 현재 클래스)에 추가하면 됨.
`backprojection_image_from_rc()` 또는 `backprojection_image_fused()` 를 이미 갖고 있다는 전제.

python

```python
import numpy as np
from scipy.constants import speed_of_light

def _to_cpu(a, xp):
    # cupy/numpy 모두 대응
    try:
        import cupy as cp
        if xp.__name__ == "cupy" and isinstance(a, cp.ndarray):
            return cp.asnumpy(a)
    except Exception:
        pass
    return np.asarray(a)

def _range_resolution_m(bandwidth_hz: float) -> float:
    # ΔR ≈ c / (2B)
    return speed_of_light / (2.0 * float(bandwidth_hz))

def find_target_coarse_and_make_range_gate(
    self,
    rc, range_window,
    z0=0.0,
    nx_coarse=96, ny_coarse=96,
    gate_margin_m=5.0,
    # clutter/노이즈가 심하면 작은 box blur 같은 후처리로 peak가 안정적
    use_power_peak=True,
):
    """
    1) Coarse BP로 target 위치 (x_est, y_est, z0) 추정
    2) 그 위치 기준 pulse별 ΔR_p 계산
    3) 전체 pulse를 커버하는 range_gate=(rmin, rmax) 생성 후 반환

    rc: (na, fft_len) complex (GPU or CPU 가능)
    range_window: (fft_len,) meters
    """

    # 1) coarse BP (빠르게)
    # - fused를 쓰든, from_rc를 쓰든 상관 없지만 nx/ny가 작으니 from_rc도 충분히 빠름
    img_c, xi_c, yi_c = self.backprojection_image_from_rc(
        rc=rc, range_window=range_window,
        nx=int(nx_coarse), ny=int(ny_coarse),
        z0=float(z0),
```

```
    range_gate=None  # coarse는 gate 없이
)

img_c_cpu = np.asarray(img_c)
xi_c_cpu = np.asarray(xi_c)
yi_c_cpu = np.asarray(yi_c)

# 2) peak 찾기
if use_power_peak:
    metric = np.abs(img_c_cpu) ** 2
else:
    metric = np.abs(img_c_cpu)

iy, ix = np.unravel_index(np.argmax(metric), metric.shape)
x_est = float(xi_c_cpu[ix])
y_est = float(yi_c_cpu[iy])
z_est = float(z0)

# 3) pulse별 ΔR_p 계산 → gate 생성
# sensor_xyz는 generate_phase_history에서 설정됨 (na,3)
sensor_xyz_cpu = _to_cpu(self.sensor_xyz, self.xp)  # (na,3) in CPU

r0 = float(self.radius)
p_est = np.array([x_est, y_est, z_est], dtype=np.float64)

dR = np.linalg.norm(sensor_xyz_cpu - p_est[None, :], axis=1) - r0  # (na,)

rmin = float(np.min(dR) - gate_margin_m)
rmax = float(np.max(dR) + gate_margin_m)

# 4) gate가 range_window 범위를 벗어나면 clamp (보간 0 방지)
rw_cpu = _to_cpu(range_window, self.xp)
rw_min = float(np.min(rw_cpu))
rw_max = float(np.max(rw_cpu))
rmin = max(rmin, rw_min)
rmax = min(rmax, rw_max)

# (추가) 너무 좁아지는 경우 안전장치
if rmax <= rmin:
    # 최소 폭 2*res 정도 확보
    res = _range_resolution_m(self.bandwidth)
    mid = 0.5 * (rmin + rmax)
```

```
        rmin = mid - 2.0 * res
        rmax = mid + 2.0 * res

    info = {
        "coarse_peak_xy": (x_est, y_est),
        "coarse_peak_ij": (int(iy), int(ix)),
        "dR_minmax": (float(np.min(dR)), float(np.max(dR))),
        "range_gate": (rmin, rmax),
    }
    return (x_est, y_est, z_est), (rmin, rmax), info
```

---

3) 실제 사용 예시 (전체 파이프라인)

아래는 "자동 coarse → gate → refined BP"를 한 번에 실행하는 흐름 예시야.

python

```
# 1) phase history (GPU 유지 권장)
sig, freq_gpu, ref_gpu = sim.generate_phase_history_optimized(use_farfield=False,
chunk_targets=200000)

# 2) window (GPU)
sig_w = sim.postprocess_gpu(sig)

# 3) range compression (batched, GPU)
rc, range_window = sim.range_compress_batched(sig_w)

# 4) coarse → target 추정 + range gate 자동 생성
# z0는 타겟이 있는 높이(평면 BP면 중요)
target_est, range_gate, info = sim.find_target_coarse_and_make_range_gate(
    rc=rc, range_window=range_window,
    z0=0.0,
    nx_coarse=96, ny_coarse=96,
    gate_margin_m=5.0,
    use_power_peak=True
)
print("[AUTO] target_est =", target_est)
print("[AUTO] range_gate =", range_gate)
print("[AUTO] info =", info)

# 5) refined BP (gate 적용)
```

```
# fused BP가 있으면 이걸 쓰는 게 더 빠름
img, xi, yi = sim.backprojection_image_fused(
    rc=rc, range_window=range_window,
    nx=500, ny=500,
    z0=0.0,
    range_gate=range_gate
)
```

---

4) 파라미터 튜닝 가이드 (진짜 중요한 부분)

gate_margin_m 추천

- range resolution $\Delta R = c/(2B)$ 기준으로 잡는 게 합리적
  - 예: B=300 MHz → ΔR≈0.5 m
- 그런데 BP에서는 보간/근사/자세 오차/scene mismatch 때문에 여유가 필요해서 보통:
  - **5 m ~ 30 m** 사이로 많이 둠

추천 시작값:

- **object만 / 단일점 검증**: 3~10 m
- **terrain+object / clutter 있음**: 10~50 m

coarse 해상도

- `nx_coarse=64~128` 정도면 충분
- 너무 작으면 peak가 grid quantization에 묻힐 수 있음

z0

- 타겟이 지형 위에 있으면 `z0=0`이 틀릴 수 있어요.
- 타겟 STL 중심 높이 또는 terrain height로 맞추면 coarse peak가 훨씬 안정적

---

5) 이 방식의 한계(알아두면 디버깅 빨라짐)

- 타겟이 여러 개면 coarse peak가 “가장 강한 타겟”으로 잡혀요.
- clutter가 훨씬 강하면 peak가 terrain에 잡힐 수 있어요.
  이때는 coarse 단계에서:
  - (a) ROI를 대충 타겟 예상 영역으로 제한하거나
  - (b) range bin에서 terrain 억제(예: 고도/경사 기반) 같은 전처리가 필요합니다.

원하면, 여기서 한 단계 더 “industry스럽게” 해서:

- coarse에서 peak 하나만 쓰지 말고 **상위 K개 peak 후보**를 뽑아서
  각 후보에 대한 gate를 평가(에너지 합이 최대인 gate 선택)

까지 자동화해줄게요. (멀티 타겟/클러터 상황에서 훨씬 견고해집니다)