Mixed HIL-PID Optimization:
Detailed Workflow for Ackermann Robot

Implementation Analysis & Debugging Guide

January 23, 2026

Contents

1 Introduction

This document provides a comprehensive technical analysis of three Human-in-the-Loop (HIL) PID optimization approaches for the Ackermann steering robot. It includes detailed workflows, code examples, parameter calculations, and debugging checkpoints.

1.1 Ackermann Robot Configuration

Listing 1: Ackermann Configuration from config.yaml

```yaml
ackermann:
  control_mode: ackermann
  wheelbase: 0.32            # meters
  wheel_radius: 0.05         # meters
  constant_speed: 5.0        # m/s
  steering_rate_limit: 2.0   # rad/s
  steering_alpha: 0.3        # smoothing factor

  # PID Control Parameters
  pid_output_limit: 0.6      # radians (steering angle limit)
  pid_bounds:
    kp: [0.1, 10.0]
    ki: [0.01, 10.0]
    kd: [0.01, 10.0]

  # Performance Targets
  pid_max_overshoot_pct: 5.0     # percent
  pid_max_rise_time: 4           # seconds
  pid_max_settling_time: 6       # seconds

  # Constraint Penalties
  pid_sat_penalty: 0.001
  pid_strict_output_limit: true
  pid_sat_hard_penalty: 100.0
```

1.2 Key Parameters Explained

- **pid_output_limit: 0.6 rad** $\approx 34.4°$ - Maximum steering angle
- **Target**: $\theta_{target} = 90°$ (turn 90 degrees)
- **Simulation**: 7500 steps $\times$ 0.00417s = 31.25 seconds
- **PID Search Space**: $K_p \in [0.1, 10], K_i \in [0.01, 10], K_d \in [0.01, 10]$

2 PID Control for Ackermann Steering

2.1 Control Loop

The PID controller adjusts the **steering angle** to achieve the target yaw:

$$\delta(t) = K_p e(t) + K_i \int_0^t e(\tau) d\tau + K_d \frac{de(t)}{dt} \tag{1}$$

where:

- $\delta(t)$ = steering angle (PID output)

- $e(t) = \theta_{target} - \theta_{current}$ = yaw error
- K_p, K_i, K_d = PID gains (optimized parameters)

Constraint: $|\delta(t)| \leq 0.6$ rad (physical steering limit)

2.2 Implementation Snippet

Listing 2: PID Evaluation in simulator.py

```python
def evaluate(self, params, label_text="", return_history=False,
    realtime=False):
    """
    Evaluate PID parameters on Ackermann robot.

    Args:
        params: [Kp, Ki, Kd] - PID gains

    Returns:
        fitness: sum of squared yaw errors
        history: trajectory data
        sat_info: {max_abs_raw_output, sat_count}
    """
    kp, ki, kd = params
    error_integral = 0.0
    prev_error = 0.0
    max_abs_output = 0.0
    sat_count = 0

    for step in range(simulation_steps):
        # Get current yaw
        _, orn = p.getBasePositionAndOrientation(robot_id)
        current_yaw = p.getEulerFromQuaternion(orn)[2] * 180/pi

        # PID calculation
        error = target_yaw - current_yaw
        error_integral += error * dt
        error_derivative = (error - prev_error) / dt

        # PID output (steering angle)
        raw_output = kp * error + ki * error_integral + kd *
            error_derivative
        max_abs_output = max(max_abs_output, abs(raw_output))

        # Apply limit (0.6 radians)
        steering_angle = np.clip(raw_output, -0.6, 0.6)
        if abs(raw_output) > 0.6:
            sat_count += 1

        # Apply to steering joints
        p.setJointMotorControl2(robot_id, steering_joints[0],
                                p.POSITION_CONTROL,
                                targetPosition=steering_angle)
        # ... (apply to other joints, step simulation)

    fitness = sum_squared_errors
    sat_info = {"max_abs_raw_output": max_abs_output, "sat_count":
        sat_count}
```

```
    return fitness, history, sat_info
```

2.3 Metrics Calculation

Listing 3: Performance Metrics from metrics.py

```
def calculate_metrics(history, target_val):
    """Calculate overshoot, rise time, settling time."""
    time_arr = np.array(history["time"])
    actual_arr = np.array(history["actual"])  # yaw values

    # Overshoot
    max_val = np.max(actual_arr)
    overshoot = ((max_val - target_val) / target_val) * 100 if max_val
        > target_val else 0.0

    # Rise time (10% to 90%)
    t_10_idx = np.where(actual_arr >= 0.1 * target_val)[0][0]
    t_90_idx = np.where(actual_arr >= 0.9 * target_val)[0][0]
    rise_time = time_arr[t_90_idx] - time_arr[t_10_idx]

    # Settling time (within 5% of target)
    tolerance = 0.05 * target_val
    out_of_bounds = np.where((actual_arr > target_val + tolerance) |
                             (actual_arr < target_val - tolerance))[0]
    settling_time = time_arr[out_of_bounds[-1] + 1] if len(
        out_of_bounds) > 0 else 0.0

    return {"overshoot": overshoot, "rise_time": rise_time, "
        settling_time": settling_time}

def meets_pid_targets(metrics, max_overshoot=5.0, max_rise_time=4,
    max_settling_time=6):
    """Check if metrics meet Ackermann targets."""
    if metrics["overshoot"] > max_overshoot:
        return False
    if metrics["rise_time"] <= 0 or metrics["rise_time"] >
        max_rise_time:
        return False
    if metrics["settling_time"] <= 0 or metrics["settling_time"] >
        max_settling_time:
        return False
    return True

def violation_from_sat(sat_info, output_limit=0.6):
    """Calculate constraint violation."""
    return sat_info["max_abs_raw_output"] - output_limit
```

3 Mixed HIL Workflow

3.1 Algorithm Overview

Mixed HIL combines **Differential Evolution (DE)** and **Bayesian Optimization (BO)**, presenting both candidates to the user for comparison.

Algorithm: Mixed HIL for Ackermann

1. Initialize DE population (6 candidates), BO surrogate
2. Warm-start BO with DE population
3. **While** iteration < max_iterations:
   (a) $\text{cand}_{\text{DE}} \leftarrow$ DE.evolve()
   (b) $\text{cand}_{\text{BO}} \leftarrow$ BO.propose_location()
   (c) Evaluate both on simulator → (fitness, history, sat_info)
   (d) Calculate metrics for both candidates
   (e) **Check auto-termination:**
   (f) **If** ($\text{target\_ok}_{\text{DE}}$ AND $\text{viol}_{\text{DE}} \leq 0$) OR ($\text{target\_ok}_{\text{BO}}$ AND $\text{viol}_{\text{BO}} \leq 0$):
      i. **TERMINATE** → Save best, exit
   (g) Show comparison GUI → User chooses: [Prefer DE — Prefer BO — TIE — REJECT]
   (h) Update algorithms based on user feedback

3.2 Detailed Code Walkthrough

Listing 4: Mixed HIL Iteration from experiment_executor.py

```python
def run_mixed_hil_experiment(self, run_index, total_runs, batch_id, robot_type="ackermann"):
    # Initialize
    de = DifferentialEvolutionOptimizer(bounds=PID_BOUNDS, pop_size=6, mutation_factor=0.5)
    bo = ConstrainedBayesianOptimizer(bounds=PID_BOUNDS, pof_min=0.95)
    pref_model = PreferenceModel(PID_BOUNDS, lr=0.3)

    # Get Ackermann-specific parameters
    robot_config = get_robot_config(_CONFIG, "ackermann")
    pid_output_limit = robot_config['pid_output_limit']  # 0.6 rad
    max_overshoot = robot_config['pid_max_overshoot_pct']  # 5.0
    max_rise_time = robot_config['pid_max_rise_time']  # 4 s
    max_settling_time = robot_config['pid_max_settling_time']  # 6 s

    # Warm-start BO with DE population
    for cand in de.population:
        fit, _, sat = self.sim.evaluate(cand)
        bo.update(cand, fit, violation_from_sat(sat, pid_output_limit))

    iteration = 0
    while iteration < _CONFIG['max_iterations']:
        iteration += 1

        # === STEP 1: Generate Candidates ===
        # DE evolves
        fitness_wrapper = lambda p: (lambda f, _, s: (f, violation_from_sat(s, pid_output_limit)))(*self.sim.evaluate(p))
        cand_a, fit_a_fast, viol_a_fast = de.evolve(fitness_wrapper)

        # BO proposes
        cand_b = bo.propose_location()
        fit_b_fast, _, sat_b_fast = self.sim.evaluate(cand_b)
        viol_b_fast = violation_from_sat(sat_b_fast, pid_output_limit)
```

```
        # Update BO with both
        bo.update(cand_b, fit_b_fast, viol_b_fast)
        bo.update(cand_a, fit_a_fast, viol_a_fast)

        # === STEP 2: Full Evaluation with History ===
        fit_a, hist_a, sat_a = self.sim.evaluate(cand_a, label_text="DE",
                                                 return_history=True, realtime=True)
        fit_b, hist_b, sat_b = self.sim.evaluate(cand_b, label_text="BO",
                                                 return_history=True, realtime=True)

        viol_a = violation_from_sat(sat_a, pid_output_limit)
        viol_b = violation_from_sat(sat_b, pid_output_limit)

        # === STEP 3: Calculate Metrics ===
        metrics_a = calculate_metrics(hist_a, 90.0)  # target_yaw_deg
        metrics_b = calculate_metrics(hist_b, 90.0)

        target_ok_a = meets_pid_targets(metrics_a, max_overshoot, max_rise_time, max_settling_time)
        target_ok_b = meets_pid_targets(metrics_b, max_overshoot, max_rise_time, max_settling_time)

        # === DEBUG OUTPUT ===
        print(f"\n[Iter {iteration}] Termination Status for ackermann:")
        print(f"  DE: Overshoot={metrics_a['overshoot']:.2f}% (limit {max_overshoot}), "
              f"Rise={metrics_a['rise_time']:.3f}s (limit {max_rise_time}), "
              f"Settle={metrics_a['settling_time']:.3f}s (limit {max_settling_time}), "
              f"Viol={viol_a:.3f} -> target_ok={target_ok_a}, feasible={viol_a<=0}")
        print(f"  BO: Overshoot={metrics_b['overshoot']:.2f}% (limit {max_overshoot}), "
              f"Rise={metrics_b['rise_time']:.3f}s (limit {max_rise_time}), "
              f"Settle={metrics_b['settling_time']:.3f}s (limit {max_settling_time}), "
              f"Viol={viol_b:.3f} -> target_ok={target_ok_b}, feasible={viol_b<=0}")

        # === STEP 4: Check Auto-Termination ===
        if (target_ok_a and viol_a <= 0) or (target_ok_b and viol_b <= 0):
            print(f"\n[SUCCESS] [Auto-terminate] Target met at iteration {iteration}!")
            # Log and save
            break

        # === STEP 5: Get User Feedback ===
        choice = gui.show_comparison(hist_a, hist_b, cand_a, cand_b, metrics_a, metrics_b)
```

```python
        # 1: Prefer DE, 2: Prefer BO, 3: TIE (Refine), 4: REJECT (Expand)

        # === STEP 6: Update Based on Feedback ===
        if choice == 1:  # Prefer DE
            pref_model.update_towards(cand_a, cand_b)
            de.inject_candidate(pref_model.anchor_params(), eval_func=fitness_wrapper, protect_best=True)
            bo.nudge_with_preference(cand_a, fit_a, fit_b, viol_a)
        elif choice == 2:  # Prefer BO
            pref_model.update_towards(cand_b, cand_a)
            de.inject_candidate(cand_b, eval_func=fitness_wrapper, protect_best=True)
            de.inject_candidate(pref_model.anchor_params(), eval_func=fitness_wrapper, protect_best=True)
            bo.nudge_with_preference(cand_b, fit_b, fit_a, viol_b)
        elif choice == 3:  # Refine
            avg = (cand_a + cand_b) / 2.0
            de.refine_search_space(avg)
            bo.refine_bounds(avg)
        elif choice == 4:  # Expand
            de.expand_search_space()
            bo.expand_bounds()
        else:
            break  # User closed GUI
```

3.3 Termination Condition Analysis

Auto-termination triggers when:

$$(\text{target\_ok}_{\text{DE}} \wedge \text{viol}_{\text{DE}} \leq 0) \vee (\text{target\_ok}_{\text{BO}} \wedge \text{viol}_{\text{BO}} \leq 0) \tag{2}$$

Where:

$$\text{target\_ok} = (\text{overshoot} \leq 5.0) \wedge (0 < \text{rise\_time} \leq 4) \wedge (0 < \text{settling\_time} \leq 6) \tag{3}$$

$$\text{viol} = \max(|\delta(t)|) - 0.6 \leq 0 \tag{4}$$

Common Failures:

- `rise_time = -1` → Never reached 90% of target
- `settling_time = -1` → Never settled (always oscillating)
- `viol > 0` → Steering exceeded 0.6 rad
- `overshoot > 5%` → Oversteered past 90°by ¿4.5°

4 BO HIL Workflow

4.1 Algorithm Overview

BO HIL uses only Bayesian Optimization with 2-option user feedback.

Algorithm: BO HIL for Ackermann

1. Initialize BO with 5 random samples
2. **While** iteration < max_iterations:

(a) cand ← BO.propose_location()
(b) Evaluate on simulator → (fitness, history, sat_info)
(c) Calculate metrics
(d) BO.update(cand, fitness, violation)
(e) **Check auto-termination:**
(f) **If** target_ok AND viol ≤ 0:
    i. **TERMINATE**
(g) Show candidate GUI → User chooses: [ACCEPT — REJECT]
(h) **If** ACCEPT:
    i. BO.refine_bounds(cand) // *Narrow search around good region*
(i) **Else**:
    i. BO.expand_bounds() // *Broaden search*

4.2 Key Code Sections

Listing 5: BO HIL Auto-Termination Logic

```
# After evaluating candidate
metrics = calculate_metrics(hist, 90.0)
target_ok = meets_pid_targets(metrics, 5.0, 4, 6)

# Debug output
print(f"\n[Iter {iteration}] Termination Status for ackermann:")
print(f"  BO: Overshoot={metrics['overshoot']:.2f}% (limit 5.0), "
      f"Rise={metrics['rise_time']:.3f}s (limit 4), "
      f"Settle={metrics['settling_time']:.3f}s (limit 6), "
      f"Viol={viol:.3f} -> target_ok={target_ok}, feasible={viol<=0}")

# Termination check
if target_ok and viol <= 0:
    print(f"\n[SUCCESS] [Auto-terminate] Target met at iteration {
        iteration}!")
    # Save logs and exit
    break
```

5 DE HIL Workflow

5.1 Algorithm Overview

DE HIL uses only Differential Evolution with 2-option user feedback.

Algorithm: DE HIL for Ackermann

1. Initialize DE population (6 candidates)
2. **While** iteration < max_iterations:
    (a) cand ← DE.evolve()
    (b) Evaluate on simulator → (fitness, history, sat_info)
    (c) Calculate metrics
    (d) **Check auto-termination:**

(e) **If** target_ok AND viol ≤ 0:

  i. **TERMINATE**

(f) Show candidate GUI → User chooses: [ACCEPT — REJECT]

(g) **If** ACCEPT:

  i. DE.refine_search_space(cand) // *Reduce bounds around good region*

(h) **Else**:

  i. DE.expand_search_space() // *Widen bounds*

6 Debugging Checklist

6.1 If Experiments Don't Auto-Terminate

Step 1: Check Debug Output

Look for patterns in console:

```
[Iter 5] Termination Status for ackermann:
  DE: Overshoot=3.2% (limit 5.0), Rise=2.1s (limit 4),
      Settle=-1.000s (limit 6), Viol=-0.05 -> target_ok=False, feasible=True
```

→ **Problem**: `settling_time = -1` (never settles)

Step 2: Identify Which Criterion Fails

Symptom	Cause	Fix
`rise_time = -1`	Never reaches 90% of 90°	Increase `max_rise_time`
`settling_time = -1`	Oscillates continuously	Increase `max_settling_time`
`viol > 0`	Steering $>$ 0.6 rad	Relax `pid_output_limit`
`overshoot > 5%`	Overshoots $>$ 94.5	Increase `max_overshoot_pct`

Step 3: Verify Ackermann Config Loaded

Add debug print at experiment start:

```python
robot_config = get_robot_config(_CONFIG, "ackermann")
print(f"Loaded Ackermann config: {robot_config}")
```

Ensure outputs:

```
pid_output_limit: 0.6
pid_max_overshoot_pct: 5.0
pid_max_rise_time: 4
pid_max_settling_time: 6
```

6.2 If Metrics Are Always Invalid

Check 1: Verify Yaw Target

```python
target_yaw_deg = _CONFIG['target_yaw_deg']
print(f"Target yaw: {target_yaw_deg}\textdegree")  # Should be 90.0
```

Check 2: Inspect History Data

```python
print(f"History length: {len(hist['time'])}")
print(f"Final yaw: {hist['actual'][-1]:.2f}\textdegree")
print(f"Max yaw: {max(hist['actual']):.2f}\textdegree")
```

Check 3: Validate Metrics Calculation

```python
metrics = calculate_metrics(hist, 90.0)
print(f"Raw metrics: {metrics}")
# Look for rise_time=-1 or settling_time=-1
```

6.3 Violation Analysis

Track Maximum Steering Output:

```python
sat_info = {"max_abs_raw_output": 0.834}  # Example
pid_output_limit = 0.6
viol = sat_info["max_abs_raw_output"] - pid_output_limit
# viol = 0.834 - 0.6 = 0.234 (INFEASIBLE)
```

If `viol > 0` consistently:

- PID gains too aggressive (high K_p, K_d)
- May need to increase `pid_output_limit` (but check physical robot limits!)
- Or accept slower convergence with more conservative PID

7 Example Termination Scenarios

7.1 Scenario 1: Successful Termination

```
[Iter 12] Termination Status for ackermann:
  DE: Overshoot=2.3% (limit 5.0), Rise=3.2s (limit 4),
      Settle=4.8s (limit 6), Viol=-0.12 -> target_ok=True, feasible=True
  BO: Overshoot=6.1% (limit 5.0), Rise=2.9s (limit 4),
      Settle=5.2s (limit 6), Viol=0.05 -> target_ok=False, feasible=False

[SUCCESS] [Auto-terminate] Target met at iteration 12!
[OK] Best solution saved to logs/ackermann_logs/MixedHIL_ackermann/...
```

→ **DE candidate met all criteria**, experiment ends.

7.2 Scenario 2: Close But No Termination

```
[Iter 8] Termination Status for ackermann:
  DE: Overshoot=4.8% (limit 5.0), Rise=3.9s (limit 4),
      Settle=6.2s (limit 6), Viol=-0.08 -> target_ok=False, feasible=True
  BO: Overshoot=3.2% (limit 5.0), Rise=3.5s (limit 4),
      Settle=5.8s (limit 6), Viol=0.02 -> target_ok=True, feasible=False
```

→ **DE fails**: settling time $6.2 > 6$
→ **BO fails**: violation $0.02 > 0$ (steering exceeded limit)
→ **Continue optimization**

7.3 Scenario 3: Never Settles

```
[Iter 45] Termination Status for ackermann:
  DE: Overshoot=2.1% (limit 5.0), Rise=2.8s (limit 4),
      Settle=-1.000s (limit 6), Viol=-0.15 -> target_ok=False, feasible=True
```

→ **Problem**: Robot oscillates around target, never stabilizes
→ **Debug**: Check if K_i too high (integral windup) or K_d too low (no damping)

8 Summary

8.1 Critical Implementation Points

1. **Robot-specific parameters must be loaded** from `config['robots']['ackermann']`
2. **PID output = steering angle** (radians), limited to ± 0.6 rad
3. **Violation** $= \max(|\delta(t)|) - 0.6$; must be ≤ 0 for feasibility
4. **Metrics** depend on trajectory reaching and stabilizing at 90°
5. **Auto-termination** requires `target_ok AND viol` $\leq$ `0`
6. **Debug output** shows exactly which criteria pass/fail each iteration

8.2 Common Error Sources

- **Wrong limits used** - Check global vs robot-specific config loading
- **Metrics calculation fails** - Verify trajectory data format
- **Overly strict targets** - Ackermann needs looser time constraints than Husky
- **GUI closes early** - User feedback loop broken before auto-termination