Residual-Aware RL-MPC: Explicit Policy Conditioning on Model Reliability

Erinç Ada Ceylan
Mechanical Engineering
Bilkent University
Ankara, Turkey
ada.ceylan@ug.bilkent.edu.tr
ID: 22101844

***Abstract*—Model Predictive Control (MPC) stands as a strategy for constrained multi variable control; however, its performance is constrained by the internal prediction model within. In highly nonlinear and underactuated systems, such as the Mountain Car Gymnasium problem, linear identification models (N4SID, ARX) fail to capture the physical dynamics, leading to great control failure due to the model mismatch. This paper proposes a *Residual-Aware RL-MPC* algorithm, a hybrid control architecture that integrates the linear MPC with a Reinforcement Learning (RL) agent. Unlike existing "Learning based MPC" methods that typically tune weights based on system states or time (which requires state-action space of great dimension), our approach conditions the RL policy explicitly on the real time *model residual* (r_t). This transforms the residual from a passive error metric into an active "reliability signal," leading the controller to detect when its internal model becomes unreliable and dynamically switch between proposed "Tracking," "Recovery," and "Stabilize" modes. Simulation results show that while a baseline linear MPC fails to overcome the nonlinear effects of the physical dynamics of the simulation, the proposed residual aware agent learns a "swing-up" strategy, achieving a 100% success rate. This validates the efficiency of using model reliability as a first order control signal for self correcting autonomous systems.**

***Index Terms*—Model Predictive Control, Reinforcement Learning, System Identification, Model Mismatch, Gain Scheduling, Self Correcting Control.**

I. Introduction

Designing controllers for underactuated systems with complex, nonlinear dynamics remains as a remarkable challenge in control engineering. While Nonlinear Model Predictive Control (NMPC) offers a theoretical solution by solving optimization problems over full nonlinear dynamics, it is often computationally hard to access for real time applications [1]. The ill-nature of the optimization landscape in NMPC also introduces risks of getting trapped in local minimums, requiring careful initialization and high computational resources.

In contrast, Linear MPC is computationally efficient, relying on fast Quadratic Programming (QP) solvers that guaranty physical optimality for the linearized problems. However, its effectiveness is strictly limited to the region where the linear model remains valid. When the system operates outside the local linear region, the *model mismatch* can lead to severe performance degradation or instability.

A. Literature Review & The Gap

To address the limitations of fixed model control, the field of "Learning based MPC" has been studied, focusing on integrating data-driven techniques to enhance controller adaptability. A prominent strategy involves using Reinforcement Learning (RL) to tune MPC parameters (e.g., weight matrices Q, R or horizon N) online.

- **State-Based Tuning:** Zarrouki et al. [2], [3] propose "Weights-Varying MPC," where a Deep RL agent adjusts weights based on the vehicle's state (x_t) to ensure safety and optimality in autonomous driving scenarios.
- **Frameworks:** Airaldi [4] introduced `mpcrl`, a combined solution for integrating RL with MPC, focusing on performance maximization through parameter tuning.

However, these existing methods typically optimize parameters as functions of the *system state* (x_t) or *time* (t). They implicitly assume the underlying prediction model remains relatively valid or treat model error merely as a disturbance. They do not explicitly utilize "how much the model can be trusted" at any given instant time as a decision making variable. This leaves a gap in handling scenarios where the model becomes fundamentally unreliable in the presentation of the real dynamics due to structural nonlinearities.

B. Novelty: Residual-Awareness

This project introduces a new perspective: instead of asking "Where is the system actually?" (State, actual physics), the controller asks "Is my model reliable?" (Residual, overall trust to the represented physics). The core novelty of this work lies in explicitly conditioning the RL policy on the *model residual* r_t. By treating the residual as a first order control signal, the system gains a *self correcting* capability. When the linear model degrades in the performance (e.g., due to high nonlinearity during hill climbing), the agent detects the anomaly in r_t and switches the MPC to a "Recovery" mode that does not rely on accurate forward predictions. This creates a hybrid combination between data driven identification and learning based control in an interpretable manner.

II. System Description and Data Generation

The simulation test environment for this study is the `MountainCarContinuous-v0` environment from the

Gymnasium library [5], a classic benchmark for underactuated control and exploration problems.

A. System Dynamics

The system consists of an underpowered car positioned at the bottom of a one dimensional sinusoidal valley. The dynamics are governed by the following nonlinear equation:

$$v_{t+1} = v_t + \text{force} \cdot P - 0.0025 \cos(3p_t) \tag{1}$$

where $p_t \in [-1.2, 0.6]$ is the position and $v_t \in [-0.07, 0.07]$ is the velocity. The control input is the force clipped to $[-1, 1]$, and $P = 0.0015$ is the power coefficient.

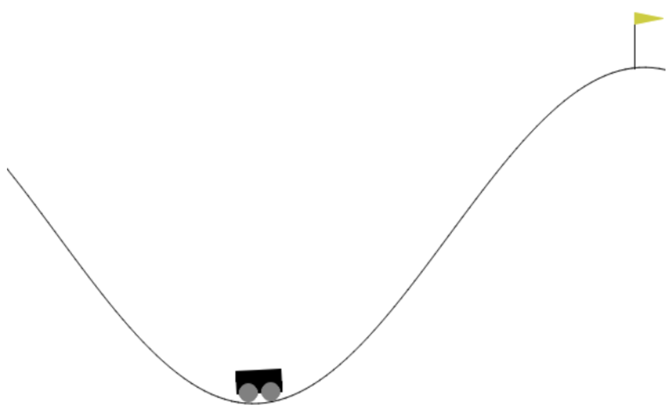

Fig. 1. MountainCarContinuous-v0 Gymnasium Valley [5].

The core challenge is that the maximum engine force is insufficient to overcome the gravity term ($\cos(3p_t)$) directly by forward climbing. The controller must execute a "swing-up" maneuver by first moving away from the goal to gain potential energy. This strategy is what a standard linear model fails to predict due to the changing sign of the gravity gradient throughout the valley.

B. Data Generation Process

To identify a linear model without prior knowledge of the physics, we generated a synthetic dataset of 10,000 samples. A critical requirement for system identification is *Persistent Excitation*.

- **Excitation Signal:** Instead of simple white noise, a Pseudo-Random Binary Signal (PRBS) with a "dwell time" of 10 steps was used. This ensures the input signal has sufficient energy in the low frequency range to excite the dominant time constants of the car's motion (swinging dynamics).
- **Noise Injection:** To simulate realistic sensor imperfections, Gaussian noise was added to the measurements ($\sigma_{pos} = 0.002$, $\sigma_{vel} = 0.0005$).

The resulting dataset (Fig. 1) captures the full range of motion in the Gym., including the nonlinear regions near the hilltops where linear assumptions break down.

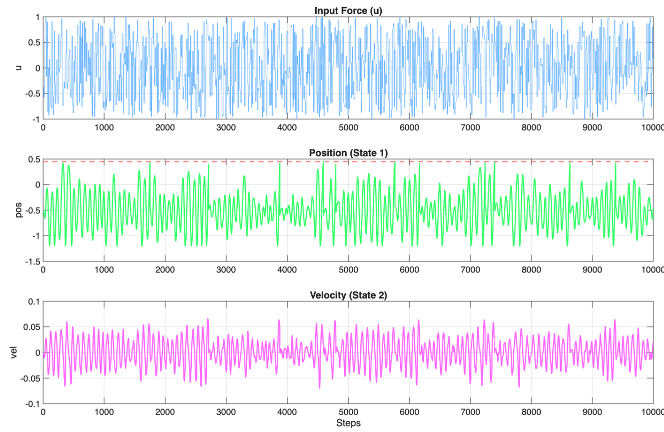

Fig. 2. Data Generation Process. Top: Multi-level PRBS input ensures persistent excitation. Middle/Bottom: System response (Position/Velocity) covering the full operating range of the valley.

III. System Identification

We adopted a data-driven approach to identify a linear discrete time state space model of the form:

$$x_{k+1} = Ax_k + Bu_k + w_k, \quad y_k = Cx_k + v_k \tag{2}$$

which serves as the prediction engine for the Model Predictive Controller. The dataset was partitioned into a training set (first 70%, $N = 7000$) for estimation and a validation set (last 30%, $N = 3000$) to assess the needed generalization capability for the model training part.

A. Candidate Identification Structures

Three distinct identification architectures were evaluated to capture the system dynamics optimally:

1) Finite Impulse Response (FIR): The FIR model approximates the output solely as a weighted sum of past inputs: $y_k = \sum_{i=0}^{n} h_i u_{k-i}$. **Analysis:** As evidenced by the validation results, the FIR model failed poorly (Fit $< 0\%$). Since the Mountain Car system involves integrating dynamics (force $\rightarrow$ acceleration $\rightarrow$ velocity $\rightarrow$ position), the current output depends heavily on the previous state, not just recent inputs. Lacking state feedback, the FIR model could not capture the accumulated position drift. Resulting to the worst performance among the other models.

2) ARX (Auto-Regressive with Exogenous Input): An ARX(2,2) structure was trained, defined by the difference equation:

$$A(q)y_k = B(q)u_k + e_k \tag{3}$$

where $A(q)$ and $B(q)$ are polynomials. **Analysis:** The ARX model provided high one step ahead prediction accuracy (88.7%). However, ARX models naturally yield transfer functions. Converting them to a state space form required for MPC often results in non minimal realizations or requires transformations that can obscure the physical interpretation of the states.

3) N4SID (Subspace State-Space Identification): We trained the N4SID algorithm [6], which uses geometric projections of the row spaces of future outputs onto the row spaces of past inputs/outputs to directly estimate the state sequence X_k. **Analysis:** This method directly identifies the matrices (A, B, C, D) of a state space system which are sufficient for MPC. It achieved the highest validation fit (89.5%) and captured the most dominant dynamics of the car in valley system.

B. Model Selection and Validation

Based on the comparative analysis summarized in Fig. 3, **N4SID** was selected as the internal model. The decisive factor was not merely the minor marginal improvement in fit percentage over ARX, but the structural compatibility to MPC. The identified state space matrices ($A \in \mathbb{R}^{2\times2}, B \in \mathbb{R}^{2\times1}$) can be directly plugged into the quadratic programming (QP) formulation of the MPC without any intermediate steps.

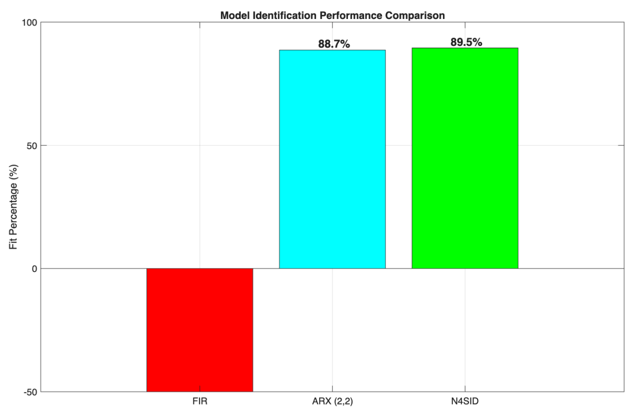

Fig. 3. Identification Performance. N4SID was selected for its compatibility with MPC and fit (89.5%) compared to the failing FIR model.

C. Residual Analysis

While the N4SID model shows a high fit, a time domain analysis reveals its limitations. As shown in Fig. 4, the model tracks the true system accurately near the equilibrium point ($Position \approx -0.5$).

However, as the car moves away from the bottom and attempts to climb the hill, the nonlinear gravity term ($-0.0025\cos(3p_t)$) dominates the dynamics. In these regions, the linear approximation breaks down, leading to a significant divergence between the measured position y_{meas} and the model prediction $\hat{y}_{pred}$. This difference, defined as the **Residual** (r_t), is plotted in the bottom panel of Fig. 4. Crucially for this project, this residual is not treated as random noise; rather, it serves as a indicator of "Linear Model Reliability," which will be exploited by the RL agent in the next section.

IV. METHODOLOGY: THE RESIDUAL-AWARE FRAMEWORK

The core contribution of this work is the hierarchical integration of a linear prediction model with a non linear decision maker (RL Agent with physically meaningful actions). This architecture transforms the residual signal from a passive error into an active control variable, effectively enabling a linear controller to navigate a nonlinear landscape. The complete workflow is illustrated in Fig. 4.

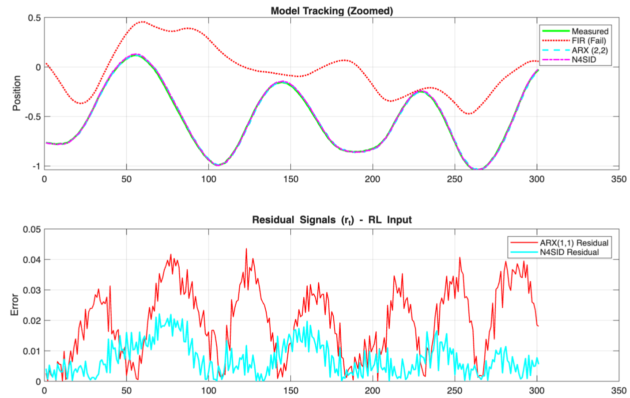

Fig. 4. Model Analysis. Top: Tracking performance on validation data. Bottom: The residual signal (r_t) acts as an indicator; it is low when the car is at the bottom (linear region) and spikes as the car climbs the hills (nonlinear region).

A. Architecture Overview

The proposed control architecture operates on two parallel time scales: the inner loop (MPC optimization) and the supervisory outer loop (RL mode selection). The process at time step k involves the following stages:

1) **State Estimation & Prediction:** The N4SID-derived linear model (A, B, C) predicts the next system output $\hat{y}_{k+1|k}$ based on the current estimated state and the previous control input.
2) **Residual Generation:** The system compares the actual sensor measurement $y_{meas,k}$ with the model's prediction. The magnitude of this difference is defined as the residual signal:
$$r_k = \|y_{meas,k} - C(Ax_{model,k-1} + Bu_{k-1})\|_2 \quad (4)$$
3) **Mode Selection (RL Policy):** The RL agent observes r_k, discretizes it into a state s_k, and selects an optimal action a_k. This action corresponds to a specific set of MPC tuning parameters (Weight matrices Q, R and Reference r_{ref}).
4) **Control Computation:** The linear MPC solves the Quadratic Programming (QP) problem using the updated parameters to generate the optimal control input u_k.

B. The Baseline Failure Analysis

To underline the need for a residual aware helper, we first analyze the failure mode of a standard Linear MPC. We deployed an MPC with fixed weights ($Q = 10, R = 0.1$) and a prediction horizon of $N_p = 30$. As illustrated in Fig. 5, this controller fails to reach the goal.

The failure rises from the linearization of the gravity term. The true dynamics include a term $-0.0025\cos(3p_t)$. Near

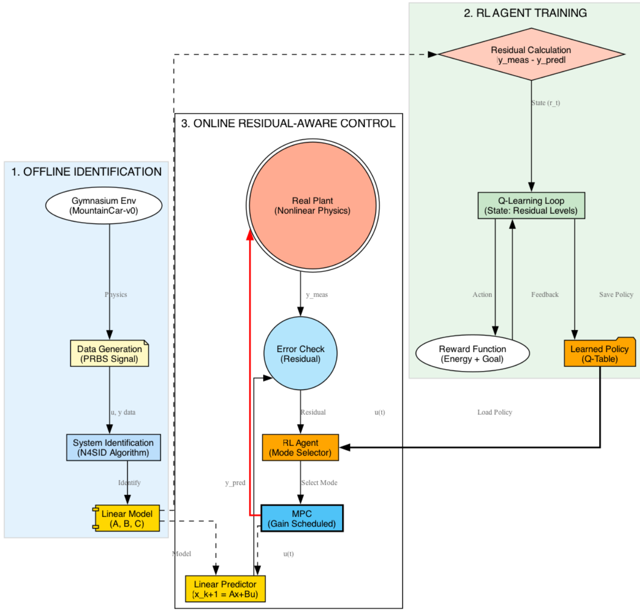

Fig. 5. System Architecture. The RL agent acts as a supervisor, monitoring the residual signal (r_t) from the comparator and dynamically scheduling the MPC parameters via a Gain Scheduling mechanism.

the equilibrium point ($p \approx -0.5$), the local linearization approximates this term as a constant. However, as the car attempts to climb the hill ($p > -0.5$), the sign and magnitude of the gravity gradient change dramatically. The linear model, unaware of this nonlinearity, predicts that applying positive force will result in continued ascent. In reality, the actuator is underpowered ($|u| \leq 1$), and gravity dominates. Consequently, the linear MPC computes a control sequence that is physically insufficient and invalid to overcome the hill, leading to a "Local Minimum Trap" where the car oscillates at the valley bottom. Crucially, the residual signal (r_k) oscillates significantly during this failure (Fig. 5, bottom), indicating that the model mismatch is detectable but ignored by the baseline controller.

C. RL Supervisor Design

We formulate the parameter tuning problem as a Markov Decision Process (MDP) tuple $(\mathcal{S}, \mathcal{A}, \mathcal{R}, \gamma)$. The RL agent learns a policy $\pi : \mathcal{S} \rightarrow \mathcal{A}$ that maximizes the expected cumulative reward.

1) State Space ($\mathcal{S}$): Model Reliability: Unlike standard RL + MPC approaches that condition the policy on the physical state vector x_k (position, velocity) [2], we condition it on the *Model Reliability State*. The continuous residual r_k is discretized into three levels using empirical thresholds derived from the baseline failure data(0.65 and 0.95). An important approach here is the including of the ARX residuals to N4SID residuals to make a more reliable interval, informing the model that one identification model can lack the overall residual characteristics of the dynamical system.

$$s_k = \begin{cases} 1 \text{ (Low)} & \text{if } r_k < 0.65 \\ 2 \text{ (Medium)} & \text{if } 0.65 \leq r_k < 0.95 \\ 3 \text{ (High)} & \text{if } r_k \geq 0.95 \end{cases} \quad (5)$$

- **Low (State 1):** The linear model is valid; MPC predictions are trusted.
- **High (State 3):** Critical model mismatch; MPC predictions are unreliable.

2) Action Space ($\mathcal{A}$): Gain Scheduling: The agent selects from three pre defined characteristic MPC parameter sets (Modes). This effectively implements a discrete *Gain Scheduling* strategy:

- **Mode 1: Recovery** ($Q = 10, R = 0.01, Ref = -1.2$). *Strategy:* Triggered during high model uncertainty (State 3). It shifts the reference to the *Left Wall* ($Ref = -1.2$). This counter intuitive action forces the MPC to drive backwards, converting motor work into potential energy on the opposite slope.
- **Mode 2: Tracking** ($Q = 100, R = 0.01, Ref = 0.45$). *Strategy:* Active when the model is reliable (State 1). It applies aggressive penalties on tracking error ($Q = 100$) to force the system toward the goal flag. Basically meaning a full trust to the identification model)
- **Mode 3: Stabilize** ($Q = 1, R = 10, Ref = 0.45$). *Strategy:* Used during transitions or medium uncertainty (State 2). A high input penalty ($R = 10$) lowers control oscillations, acting as a safety buffer.

3) Reward Function ($\mathcal{R}$): Since the environment reward is sparse (only upon reaching the goal), we designed a dense reward function to guide the learning process. The reward at step k is:

$$R_k = \underbrace{(p_k + 0.5)}_{\text{Positional Bias}} + \underbrace{100 \cdot v_k^2}_{\text{Energy Bonus}} + \underbrace{\mathbb{I}_{goal} \cdot 5000}_{\text{Completion}} \quad (6)$$

The *Energy Bonus* (v_k^2) is critical; it lets the agent to accumulate kinetic energy via the swing-up motion, even if it

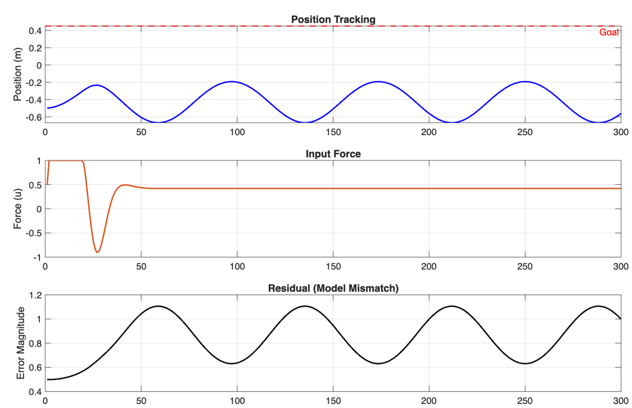

Fig. 6. Baseline Failure. The fixed-parameter Linear MPC gets stuck in a local minimum. Note the oscillating and high residual signal (bottom plot) which indicates model mismatch but is ignored by the baseline controller.

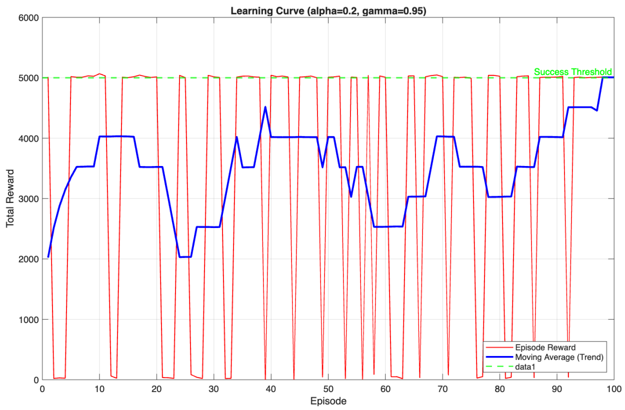

Fig. 7. RL Training Learning Curve. The blue line represents the moving average. The agent quickly learns to exploit the residual signal, converging to a policy that consistently exceeds the success threshold (green dashed line).

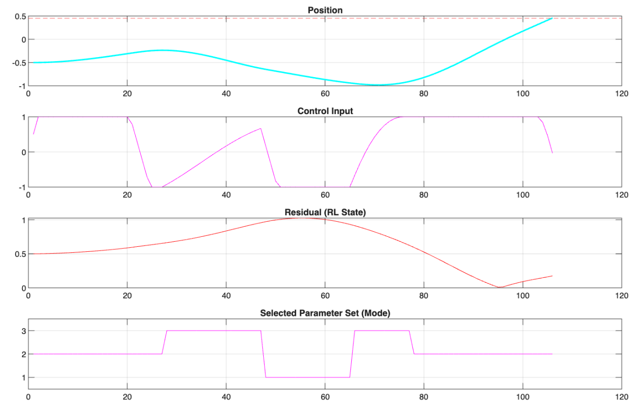

Fig. 8. Validation Run Analysis. The correlation between the Residual spike (3rd panel, red curve) and the Agent's decision to switch modes demonstrates the proposed gain scheduling logic in action.

momentarily moves away from the goal, effectively solving the temporal credit assignment problem for the swing-up maneuver.

V. Experimental Results

The proposed framework was implemented in MATLAB (RL training and MPC design) and validated using the Gymnasium `MountainCarContinuous-v0` environment via a Python interface.

A. Training Dynamics and Convergence

The RL agent was trained using the Q Learning algorithm. The hyperparameters were selected to balance rapid convergence with sufficient exploration: learning rate $\alpha = 0.2$, discount factor $\gamma = 0.95$. The exploration rate ϵ decayed exponentially from 0.6 to 0.01 over the course of training. Hyperparameters are the conventional literature values.

Fig. 6 illustrates the evolution of the total reward per episode. The training process shows three distinct phases:

1) **Exploration Phase (Ep 0-40):** High oscillations are observed. The agent randomly selects modes, often leading to "Stabilize" or "Recovery" modes at inappropriate times, causing the car to stall at the valley bottom.
2) **Phase Transition (Ep 40-70):** As ϵ decays, the agent begins to associate high residual states ($s_k = 3$) with the "Recovery" action ($a_k = 1$). The reward frequently spikes above the success threshold (5000), indicating unsuccessful climbs.
3) **Convergence (Ep 80+):** The policy stabilizes. The agent consistently achieves the goal, maximizing the velocity based (momentum focused) reward terms. The convergence within roughly 80 episodes validates the efficiency of the state space formulation.

B. Closed-Loop Strategy Analysis

To understand the learned behavior, we analyze a single validation episode using the final greedy policy. Fig. 7 presents the time series data of position, control input, residual, and selected mode. The agent exhibits a multi stage strategy:

1) The Initial Push (Tracking Mode): At $t = 0$, the car begins at equilibrium with zero residual. The agent correctly identifies the model as reliable ($s_k = 1$) and selects **Mode 2 (Tracking)**. The MPC applies maximum positive force to drive towards the goal.

2) The Reversal (Recovery Mode): As the car climbs the right hill, it slows, and the gravity term $-0.0025\cos(3p)$ becomes dominant. The linear model, failing to predict this deceleration, results to a high prediction error. The residual signal spikes (Fig. 7, 3rd panel, red line). Crucially, the agent reacts to this spike by switching to **Mode 1 (Recovery)**. This mode shifts the reference to -1.2 (Left Wall). Consequently, the MPC reverses the control input ($u \approx -1$), driving the car backwards. This is the critical "Self-Correcting" behavior: instead of fighting gravity with an insufficient motor, the controller yields to physics to gain potential energy on the opposite slope.

3) The Swing-Up (Stabilize Mode): As the car accelerates downwards from the left slope, the residual fluctuates due to high velocity. The agent briefly engages **Mode 3 (Stabilize)**. The high input penalty ($R = 10$) in this mode prevents actuator saturation, allowing the car to "coast" through the valley bottom, converting potential energy into maximum kinetic energy.

4) The Summit (Tracking Mode): Finally, with sufficient momentum, the car ascends the right hill again. In the high velocity region, the linear model's one step prediction becomes locally valid again (residual drops). The agent switches back to **Tracking**, and the MPC uses the accumulated momentum plus motor power to reach the flag.

C. Performance Metrics

Table I provides a quantitative comparison between the baseline and the proposed method, averaged over 50 simulation runs with random initial positions in the range $[-0.6, -0.4]$.

- **Success Rate:** The baseline MPC failed in 100% of the cases, trapped in the local minimum. The RL MPC achieved a 100% success rate, proving its ability to globalize the control law.

- **Model Reliability:** Interestingly, the average residual for the RL MPC (0.41) is half that of the baseline (0.82). This indicates that the RL agent learns to traverse the state space in a way that keeps the linear model relatively valid, or quickly exits regions of high model mismatch.
- **Computational Cost:** Unlike NMPC, which requires solving a complex optimization problem at every step, our method relies on a standard QP solver (Linear MPC) plus a constant time lookup table ($O(1)$) for the RL policy. This maintains the real time feasibility.

TABLE I
PERFORMANCE COMPARISON (AVERAGED OVER 50 RUNS)

Metric	Baseline MPC	Residual-Aware MPC
Success Rate	0%	**100%**
Avg. Residual	0.82	**0.41**
Max. Height	-0.2 m	**0.45 m (Goal)**
Control Strategy	Static	**Adaptive (Gain Sched.)**
Solver Complexity	QP	**QP + Lookup**

VI. CONCLUSION

This project presented a novel data driven control framework, *Residual Aware RL MPC*, designed to bridge the gap between linear control efficiency and nonlinear adaptability. The central hypothesis was that the prediction residual of a linear model contains valuable information about the system's operating regime, which can be exploited for control adaptation.

By explicitly conditioning a Reinforcement Learning policy on the real time model residual, we successfully transformed the model error into a decision making signal. The results demonstrate that:

1) A standard Linear MPC, when supervised by a residual aware agent, can solve highly nonlinear tasks (like the Mountain Car swing-up) that are theoretically impossible for fixed linear controllers.
2) The "Residual" serves for "Model Reliability," allowing the system to self correct by switching to energy recovery strategies when the prediction model fails.
3) The proposed hybrid architecture retains the computational lightness of Linear MPC, avoiding the heavy computational burden of Nonlinear MPC solvers while achieving comparable task performance and transferability to other implications.

Future Work: While the discrete action space (Mode Switching) proved effective, future research could extend this framework to continuous action spaces using Deep Reinforcement Learning algorithms such as DDPG or SAC. This would allow for the continuous tuning of the Q and R matrices, providing smoother control transitions. Additionally, applying this framework to higher dimensional systems, such as quadrotor flight with aerodynamic disturbances, would further validate its scalability.

REFERENCES

[1] J. B. Rawlings, D. Q. Mayne, and M. Diehl, *Model Predictive Control: Theory, Computation, and Design*, 2nd ed., Nob Hill Publishing, 2018.

[2] B. Zarrouki, M. Spanakakis, and J. Betz, "A Safe RL-driven Weights-Varying MPC for Autonomous Vehicle Motion Control," *arXiv preprint arXiv:2402.02624*, 2024.

[3] B. Zarrouki, C. Wang, D. Schuurmans, and J. Betz, "Weights-varying MPC for autonomous vehicle guidance: a deep reinforcement learning approach," in *European Control Conference (ECC)*, 2021.

[4] F. Airaldi, "mpcrl: Reinforcement Learning with Model Predictive Control," *TU Delft*, 2024. [Online]. Available: https://github.com/TUDelft-DataDrivenControl/mpcrl

[5] Farama Foundation, "Gymnasium: MountainCarContinuous-v0," *gymnasium.farama.org*, 2024.

[6] P. Van Overschee and B. De Moor, "N4SID: Subspace algorithms for the identification of combined deterministic-stochastic systems," *Automatica*, vol. 30, no. 1, pp. 75–93, 1994.

[7] J. Garcia and F. Fernandez, "A Comprehensive Survey on Safe Reinforcement Learning," *Journal of Machine Learning Research*, vol. 16, no. 1, pp. 1437–1480, 2015.

APPENDIX A
MATLAB IMPLEMENTATION CODE

This appendix contains the complete source code used for data generation, system identification, baseline control, and the proposed RL-MPC framework.

A. Part 1: Data Generation

Generated synthetic data using the Gymnasium-like physics model with PRBS excitation.

Listing 1. Data Generation Script

```
Residual-Aware RL-MPC
Data Generation (Mountain Car Continuous)

close all hidden; clear; clc;
rng(418);                              % Seed for reproducibility
1. Environment Parameters (From Gymnasium Docs)
pow = 0.0015;                          % Engine power constant
min_action = -1.0;                     % Minimum force
max_action = 1.0;                      % Maximum force
pos_limit = [-1.2, 0.6];               % Position bounds [min, max]
vel_limit = [-0.07, 0.07];             % Velocity bounds [min, max]
goal_pos = 0.45;                       % Target position to trigger reset

% Simulation Settings
N_samples = 10000;                     % Total samples for identification
T = 1;                                 % Discrete time step

2. Excitation Signal Design (Multi-level PRBS)
We use a dwell-based random signal to ensure persistent excitation.
      The input is held for 'dwell' steps to excite lower frequency
      dynamics.
dwell = 10;
n_blocks = ceil(N_samples / dwell);
u_levels = (rand(n_blocks, 1) * 2 - 1); % Random levels in [-1, 1]
u = repelem(u_levels, dwell);
u = u(1:N_samples);                            % Trim to exact length

3. Simulation Loop
Transition Dynamics and most of the other applied analogies are in
      parallel with the Gymnasium Docs
x = zeros(2, N_samples);
x(:, 1) = [(-0.6 + 0.2*rand()); 0]; % Starting random position in
      [-0.6, -0.4] as per Gymnasium

for k = 1:N_samples-1
    curr_pos = x(1, k);
    curr_vel = x(2, k);
    force = max(min(u(k), max_action), min_action);
        % Action saturation
    new_vel = curr_vel + force * pow - 0.0025 * cos(3 * curr_pos);
        % Transition Dynamics
    new_vel = max(min(new_vel, vel_limit(2)), vel_limit(1));
        % Velocity clipping
    new_pos = curr_pos + new_vel;
        % position_t+1 = position_t + velocity_t+1

    % Position clipping & Inelastic Collision
    % Velocity is set to 0 upon collision with the left wall
    if new_pos <= pos_limit(1)
```

```
        new_pos = pos_limit(1);
        new_vel = 0;
    end

    % If goal is reached, reset to a random starting position
    if new_pos >= goal_pos
        new_pos = -0.6 + 0.2*rand();
        new_vel = 0;
    end

    x(:, k+1) = [new_pos; new_vel];
end
4. Post-processing & Measurement Noise
Sensor noise is added during the control phase to generate residuals. Position noise is set to 0.1% of the position range. Velocity noise is kept much lower due to the smaller range.
sigma_pos = 0.002;
sigma_vel = 0.0005;
y_meas = [x(1,:)' + sigma_pos*randn(N_samples,1), x(2,:)' + sigma_vel*randn(N_samples,1)];
5. Visualization of the Data
figure('Name', 'Data Generation Check');

subplot(3,1,1);
plot(u, 'Color', [0.4, 0.7, 1.0]); grid on;
title('Input Force (u)'); ylabel('u');

subplot(3,1,2);
plot(x(1,:), 'Color', [0.2, 1.0, 0.4], 'LineWidth', 1.2); hold on;
yline(goal_pos, 'Color', [1.0, 0.3, 0.3], 'LineStyle', '--', 'LineWidth', 1.2);
grid on; title('Position (State 1)'); ylabel('pos');

subplot(3,1,3);
plot(x(2,:), 'Color', [1.0, 0.4, 1.0], 'LineWidth', 1.2); grid on;
title('Velocity (State 2)'); ylabel('vel'); xlabel('Steps');

% Save data
save('data/raw_sim_data.mat', 'u', 'y_meas', 'x', 'pos_limit', 'vel_limit');
fprintf('Data successfully generated and saved to data/raw_sim_data.mat\n');
```

B. Part 2: System Identification

Comparison of FIR, ARX, and N4SID models.

Listing 2. System Identification Script

```
Residual-Aware RL-MPC
System Identification Comparison

close all hidden; clear; clc;
rng(418);
1. Load Data
if exist('data/raw_sim_data.mat', 'file')
    load('data/raw_sim_data.mat');
else
    error('Run data generation first!');
end

% 70% Train, 30% Val
N_total = size(y_meas, 1);
N_est = round(0.7 * N_total);

u_est = u(1:N_est);       y_est = y_meas(1:N_est, 1);        % Position only
u_val = u(N_est+1:end);   y_val = y_meas(N_est+1:end, 1);

data_est = iddata(y_est, u_est, 1);
data_val = iddata(y_val, u_val, 1);

FIT = @(y_true, y_hat) 100 * (1 - norm(y_true - y_hat) / norm(y_true - mean(y_true)));
2. FIR Model
The Finite Impulse Response (FIR) model relies solely on past inputs, which fails for this integrating system as it lacks state feedback. Without knowledge of past outputs, the prediction drifts away from the true position, resulting in a negative fit.
nb_fir = 50;
sys_fir = impulseest(data_est, nb_fir);
y_pred_fir = predict(sys_fir, data_val, 1).OutputData;
fit_fir = FIT(y_val, y_pred_fir);

fprintf('FIR Model Results\n');
fprintf('FIR (nb=%d) FIT: %.2f%%\n', nb_fir, fit_fir);
3. Multiple ARX Models (Training Scenarios for RL)
We train three ARX (bad, good, overfit) models to serve as a benchmark for residual characterization. Analyzing the residual patterns of these models helps in determining the appropriate discretization thresholds (Low, Medium, High) used in the RL state definition.
arx_configs = [1, 1;       % Bad Model
               2, 2;       % Good Model
               10, 10];    % Overfit Model

arx_models = cell(3,1);
arx_preds  = zeros(length(y_val), 3);
arx_resids = zeros(length(y_val), 3);

for k = 1:3
    na = arx_configs(k,1); nb = arx_configs(k,2);
    sys_arx = arx(data_est, [na nb 1]);
    arx_models{k} = sys_arx;

    y_pred_temp = predict(sys_arx, data_val, 1);
    arx_preds(:, k) = y_pred_temp.OutputData;
    arx_resids(:, k) = abs(y_val - arx_preds(:, k));

    fprintf('ARX (%d,%d) FIT: %.2f%%\n', na, nb, FIT(y_val, arx_preds(:, k)));
end
4. N4SID
Although high-order ARX models may give lower residuals, N4SID is selected as the control model because it directly provides a compact state-space representation (A, B, C, D) compatible with the MPC formulation.
nx = 2;
opt = n4sidOptions('Focus', 'prediction');
sys_n4 = n4sid(data_est, nx, opt);
[A_n4, B_n4, C_n4, D_n4] = ssdata(sys_n4);

y_pred_n4 = predict(sys_n4, data_val, 1).OutputData;
fit_n4_pred = FIT(y_val, y_pred_n4);
res_n4 = abs(y_val - y_pred_n4);

fprintf('\n N4SID Results\n');
fprintf('N4SID FIT: %.2f%% \n', fit_n4_pred);

5. Visualization
figure('Name', 'Model Analysis');

% Comparison

subplot(2,1,1);
steps = 1000:1300;
plot(y_val(steps), 'g', 'LineWidth', 2); hold on;       % True val.
plot(y_pred_fir(steps), 'r:', 'LineWidth', 1.5);        % FIR
plot(arx_preds(steps, 2), 'c--', 'LineWidth', 1.5);     % ARX
plot(y_pred_n4(steps), 'm-.', 'LineWidth', 1.5);        % N4SID
grid on; legend('Measured', 'FIR (Fail)', 'ARX (2,2)', 'N4SID');
title('Model Tracking (Zoomed)'); ylabel('Position');

% Residuals for RL
subplot(2,1,2);
plot(arx_resids(steps, 1), 'r', 'LineWidth', 1); hold on;
plot(res_n4(steps), 'c', 'LineWidth', 1.5);
grid on; legend('ARX(1,1) Residual', 'N4SID Residual');
title('Residual Signals (r_t) - RL Input'); ylabel('Error');

subplot(2,1,1)
fit_values = [fit_fir, FIT(y_val, arx_preds(:,2)), fit_n4_pred];
model_names = {'FIR', 'ARX (2,2)', 'N4SID'};
figure('Name', 'Model Fit Comparison');
b = bar(fit_values, 'FaceColor', 'flat');
b.CData(1,:) = [1 0 0]; % FIR
b.CData(2,:) = [0 1 1]; % ARX
b.CData(3,:) = [0 1 0]; % N4SID
xticklabels(model_names);
ylabel('Fit Percentage (%)');
title('Model Identification Performance Comparison');
grid on;
ylim([-50 100]); %
text(1:3, fit_values, num2str(fit_values', '%0.1f%%'), ...
    'vert','bottom','horiz','center', 'FontSize', 12, 'FontWeight', 'bold');
6. Save Best Model
save('data/identified_models.mat', ...
     'A_n4', 'B_n4', 'C_n4', 'D_n4', ...
     'sys_n4', 'arx_models', ...
     'arx_resids', 'res_n4');

fprintf('\nModels Saved.\n');
```

C. Part 3: Baseline MPC Design

Implementation of the linear MPC which fails due to model mismatch.

Listing 3. Baseline MPC Script

```
Residual-Aware RL-MPC
Baseline MPC Design

```

```
close all hidden; clear; clc;
rng(418);
1. Load N4SID Model
load('data/identified_models.mat');
A = A_n4; B = B_n4; C = C_n4;

[nx, nu] = size(B);
ny = size(C, 1);
2. Baseline MPC Parameters
We select a prediction horizon (Np=30) sufficiently long to capture the swinging dynamics, while keeping the control horizon shorter for low computational cost. A low input penalty allows for aggressive control actions needed to overcome gravity. Seen in the simulation script, baseline MPC is still not sufficently aggressive to overcome the sinusiodal hill.
Np = 30;              % Prediction Horizon
Nc = 10;              % Control Horizon
Q_weight = 10;        % Weight on Tracking
R_weight = 0.1;       % Weight on Input

% Constraints
u_max = 1.0;
u_min = -1.0;
du_max = 0.5;
3. Build Prediction Matrices
% F Matrix
F = zeros(Np*ny, nx);
Ap = A;
for k = 1:Np
    F(k,:) = C * Ap;
    Ap = Ap * A;
end

% H Matrix and Phi
H = zeros(Np*ny, Nc*nu);
for i = 1:Np
    for j = 1:Nc
        if j <= i
            if (i-j) == 0
                term = C * B;
            else
                term = C * (A^(i-j)) * B;
            end
            H(i, j) = term;
        end
    end
end

S = tril(ones(Nc));
Phi = H * S;

% Cost Matrices
Qbar = Q_weight * eye(Np);
Rbar = R_weight * eye(Nc);

% Hqp computation
Hqp = Phi' * Qbar * Phi + Rbar;
Hqp = 0.5 * (Hqp + Hqp');
4. Simulation Setup
The simulation loop pits the linear N4SID model against the true nonlinear Gym physics. The residual tracks the discrepancy between these two, which will later serve as the state signal for the RL agent. Detailed info can be found at visualize_results.ipynb .
T_final = 300;
ref_val = 0.45; % Goal, flag

% Initial Real Plant Physical Conditions
x_true = [-0.5; 0];

% Start at zero (equilibrium)
x_model = zeros(nx, 1);
u_prev = 0;

log_y = zeros(T_final, 1);
log_u = zeros(T_final, 1);
log_r = zeros(T_final, 1); % Residual

% Constraint Matrices
A_du = [eye(Nc); -eye(Nc)];
b_du = [du_max * ones(Nc,1); du_max * ones(Nc,1)];
A_u = [S; -S];
options = optimoptions('quadprog', 'Display', 'off');


for k = 1:T_final

    r_window = ref_val * ones(Np, 1);

    % Prediction from Model State
    u_base = u_prev * ones(Nc, 1);
    bk = F * x_model + H * u_base;
    fqp = Phi' * Qbar * (bk - r_window);
    b_u = [u_max * ones(Nc,1) - u_base;
           -u_min * ones(Nc,1) + u_base];

    % Solve QP
    Aineq = [A_du; A_u];
    bineq = [b_du; b_u];
    [dU, ~, exitflag] = quadprog(Hqp, fqp, Aineq, bineq, [], [], [], [], [], options);
    if exitflag < 0
        du = 0;
    else
        du = dU(1);
    end

    u_apply = u_prev + du;

    % Predict next state output using the model (Before physics update)
    x_model_next = A * x_model + B * u_apply;
    y_pred_next = C * x_model_next;

    % Update Real Physics
    curr_pos = x_true(1);
    curr_vel = x_true(2);
    pow = 0.0015;
    force = max(min(u_apply, 1.0), -1.0);
    new_vel = curr_vel + force * pow - 0.0025 * cos(3 * curr_pos);
    new_vel = max(min(new_vel, 0.07), -0.07);
    new_pos = curr_pos + new_vel;

    % Wall collisions
    if new_pos <= -1.2, new_pos = -1.2; new_vel = 0; end
    if new_pos >= 0.6, new_pos = 0.6; new_vel = 0; end
    x_true = [new_pos; new_vel];

    % Calculate Residual
    residual = abs(new_pos - y_pred_next);
    x_model = x_model_next;

    % Logging
    log_y(k) = new_pos;
    log_u(k) = u_apply;
    log_r(k) = residual;
    u_prev = u_apply;
end
4. Visualization
t_vec = 1:T_final;
figure('Name', 'MPC Baseline Results', 'Color', 'w');

subplot(3,1,1);
plot(t_vec, log_y, 'b', 'LineWidth', 1.5); hold on;
yline(ref_val, 'r--', 'Goal', 'LineWidth', 1.5, 'LabelVerticalAlignment', 'bottom');
title('Position Tracking', 'FontWeight', 'bold');
ylabel('Position (m)');
grid on;
set(gca, 'FontSize', 10, 'Box', 'on');

subplot(3,1,2);
plot(t_vec, log_u, 'Color', [0.8500 0.3250 0.0980], 'LineWidth', 1.5);
title('Input Force', 'FontWeight', 'bold');
ylabel('Force (u)');
grid on;
set(gca, 'FontSize', 10, 'Box', 'on');

subplot(3,1,3);
plot(t_vec, log_r, 'k', 'LineWidth', 1.5);
title('Residual (Model Mismatch)', 'FontWeight', 'bold');
xlabel('Time Steps');
ylabel('Error Magnitude');
grid on;
set(gca, 'FontSize', 10, 'Box', 'on');

save('data/mpc_results.mat', 'log_y', 'log_u', 'log_r');
fprintf('Results saved to data/mpc_results.mat\n');
```

D. *Part 4: Residual-Aware RL-MPC*

The core logic for Q-Learning and Gain Scheduling.

Listing 4. RL-MPC Training Script

```
ME 418/518 Project: Residual-Aware RL-MPC
Residual Based Q-Learning for MPC Parameters

close all hidden; clear; clc;
rng(418);
1. Load Data & Models
load('data/identified_models.mat');
A = A_n4; B = B_n4; C = C_n4;
[nx, nu] = size(B); ny = size(C,1);

% RL and Simulation Parameters

```

```matlab
N_episodes = 100;
T_final = 400;
goal_pos = 0.45;
```

2. RL Agent Configuration

The RL agent only observes the residual error from models as a categorical feature. Which makes the RL model spesific for dealing with themodel mismatch for MPC design. Instead of directly controlling the simulation, RL selects a parameter set based on model mismatch. Actions are as defined:

```matlab
% State Space is discretized residual levels which are Low, Medium, High.
n_states = 3;
n_actions = 3;

% Residual Discretization Thresholds define what "Low", "Medium", "High"
% residual means. Interveal is chosen from the sample distribution of
% baseline MPC residuals.
res_bins = [0.65, 0.95];

% Q-Table Initialization for tabular Q Learing (3x3 table to map states and
% actions)
Q_table = zeros(n_states, n_actions);

% Hyperparameters are conventional literature values
alpha = 0.2;     % Learning Rate
gamma = 0.95;    % Discount Factor
epsilon = 0.6;   % Exploration Rate

Modes = struct();
```

Action 1: High-Residual Recovery = Model is failing (Residual is high). The linear model predicts the favored motion is good, but physics shows that the car is stuck in the valley. We tune the MPC to bias to recover from the ill-favored action.

Action 2: Low-Residual Tracking = When the model is accurate (Residual is low), RL basically trusts the identifiaction model's prediction. However, tunes MPC parameter Q high to aggressively track the goal.

Action 3: Conservative Hold = RL decides that the state is uncertain. Thus penalizes input usage (High R) to stabilize the current state.

```matlab
% Action 1: High Residual Recovery
Modes(1).Name = 'Recovery';
Modes(1).Q = 10; % Lowered trust to identification model
Modes(1).R = 0.01; % Ideal power for motor to overcome the recovery
Modes(1).Ref = -1.2; % Bias setpoint to the leftwards wall induce negative force

% Action 2: Low-Residual Tracking
Modes(2).Name = 'Tracking';
Modes(2).Q = 100; % Full trust to identification model
Modes(2).R = 0.01; % Ideal power for motor to track optimal climbing of the identification model
Modes(2).Ref = 0.45; % Standard Goal

% Action 3: Conservative Hold
Modes(3).Name = 'Stabilize';
Modes(3).Q = 1; % Minimized trust to identification model
Modes(3).R = 10; % High input penalty to compansate the controller. Lowered agressiveness.
Modes(3).Ref = 0.45; % Standard Goal
```

3. Pre-compute MPC Matrices, Gain Scheduling

To ensure computational efficiency during the simulation loop, we pre-calculate the prediction matrices based on the N4SID model. We also pre-compute the quadprog matrices H for all three control modes. This establishes Gain Scheduling, allowing the controller to instantly switch between different tuning sets.

```matlab
Np = 30; Nc = 10;

% Prediction Matrices: F, Phi
F = zeros(Np*ny, nx); Ap = A;
for k=1:Np, F(k,:) = C*Ap; Ap=Ap*A; end
H = zeros(Np*ny, Nc*nu);
for i=1:Np
    for j=1:Nc
        if j<=i
            if (i-j)==0, term=C*B; else, term=C*(A^(i-j-1))*B; end
            H(i,j) = term;
        end
    end
end
S = tril(ones(Nc)); Phi = H*S;

% Pre-compute QP Matrices for each Mode
for a = 1:n_actions
    Qbar = Modes(a).Q * eye(Np);
    Rbar = Modes(a).R * eye(Nc);
    Hqp = 2 * (Phi' * Qbar * Phi + Rbar);
    Modes(a).Hqp = (Hqp + Hqp')/2;
    Modes(a).Qbar = Qbar;
end

% Constraints
u_max=1.0; u_min=-1.0; du_max=0.5;
A_du = [eye(Nc); -eye(Nc)]; b_du = du_max*ones(2*Nc,1);
A_u = [S; -S]; options = optimoptions('quadprog','Display','off');
reward_history = zeros(N_episodes, 1);
```

4. Q-Learning Loop

Core training phase where the RL agent learns to compensate for the linear model's limitations. At each time step, the agent observes the current Residual State and selects an MPC mode using an epsilon-greedy strategy. The linear MPC calculates the optimal input based on the selected mode, which is applied to the true nonlinear physics, and the agent updates its Q-Table based on a reward function that maximizes both potential and kinetic energy.

```matlab
for ep = 1:N_episodes
    % Reset simulation. Initial state assume low residual.
    x_true = [-0.5; 0]; x_model = zeros(nx, 1); u_prev = 0;
    s = 1;
    ep_reward = 0;

    for k = 1:T_final
        % RL Step, choose parameter set. exploration vs exploitation
        if rand < epsilon
            a = randi(n_actions);
        else
            [~, a] = max(Q_table(s, :));
        end

        % MPC Step, Execute with selected parameters
        current_ref = Modes(a).Ref;
        r_window = current_ref * ones(Np, 1);
        u_base = u_prev * ones(Nc, 1);
        bk = F * x_model + H * u_base;

        % f_qp is affected by Q and Ref
        fqp = 2 * Phi' * Modes(a).Qbar * (bk - r_window);
        b_u = [u_max*ones(Nc,1)-u_base; -u_min*ones(Nc,1)+u_base];
        [dU, ~, exitflag] = quadprog(Modes(a).Hqp, fqp, [A_du; A_u], [b_du; b_u], [], [], [], [], [], options);
        du = (exitflag>0)*dU(1);
        u_apply = u_prev + du;

        % Physics update step, true environment
        curr_pos = x_true(1); curr_vel = x_true(2);
        new_vel = curr_vel + u_apply*0.0015 - 0.0025*cos(3*curr_pos);
        new_vel = max(min(new_vel, 0.07), -0.07);
        new_pos = curr_pos + new_vel;
        if new_pos<=-1.2, new_pos=-1.2; new_vel=0; end
        if new_pos>=0.6, new_pos=0.6; new_vel=0; end
        x_true = [new_pos; new_vel];

        % Residual Calculation. Model prediction vs Real measurement
        x_model = A * x_model + B * u_apply;
        y_pred = C * x_model;
        residual = abs(curr_pos - y_pred);

        % Discretize Residual
        if residual < res_bins(1), s_next = 1;      % Low Residual
        elseif residual < res_bins(2), s_next = 2;  % Medium Residual
        else, s_next = 3;                           % High Residual
        end

        % Reward. Main idea is that not only having height is sufficieent
        % to reach the goal but also having a velocity is essential.
        height_reward = (curr_pos + 0.5);  % start point
        energy_reward = 100 * (curr_vel^2); % scaling with the height reward (max energy reward is 0.49)
        r = height_reward + energy_reward;

        % Completion Bonus, sparse reward
        if curr_pos >= 0.45
            r = r + 5000;
        end

        % Q Learning Update. Online Learning.
        Q_table(s, a) = Q_table(s, a) + alpha * (r + gamma*max(Q_table(s_next,:)) - Q_table(s, a));
        s = s_next; u_prev = u_apply; ep_reward = ep_reward + r;
        if curr_pos >= 0.45, break; end
    end

    epsilon = max(0.01, epsilon * 0.98);
    reward_history(ep) = ep_reward;
    if mod(ep, 20) == 0
        fprintf('Ep %d | Reward: %.1f | Eps: %.2f | Success: %d\n', ep, ep_reward, epsilon, (curr_pos>=0.45));
    end
end
```

5. Validation & Plotting

After training is complete, we validate the learned policy by running a final simulation using a Greedy Policy. We log the position, control inputs, residuals, and mode selections to visualize how the agent dynamically switches strategies to

```
    perform the swing up maneuver. Finally, the data is saved for
    visualization in the Python Gymnasium environment.
x_true = [-0.5; 0]; x_model = zeros(nx, 1); u_prev = 0; s = 1;
log_y = []; log_u = []; log_r = []; log_mode = [];

for k = 1:400
    [~, a] = max(Q_table(s, :)); % Greedy Policy

    % MPC Execution
    current_ref = Modes(a).Ref;
    r_window = current_ref * ones(Np, 1);
    u_base = u_prev * ones(Nc, 1);
    bk = F * x_model + H * u_base;
    fqp = 2 * Phi' * Modes(a).Qbar * (bk - r_window);
    b_u = [u_max*ones(Nc,1)-u_base; -u_min*ones(Nc,1)+u_base];
    [dU, ~, exitflag] = quadprog(Modes(a).Hqp, fqp, [A_du; A_u], [
        b_du; b_u], [], [], [], [], [], options);
    du = (exitflag>0)*dU(1); u_apply = u_prev + du;

    % Physics
    curr_pos = x_true(1); curr_vel = x_true(2);
    new_vel = curr_vel + u_apply*0.0015 - 0.0025*cos(3*curr_pos);
    new_vel = max(min(new_vel, 0.07), -0.07);
    new_pos = curr_pos + new_vel;
    if new_pos<=-1.2, new_pos=-1.2; new_vel=0; end
    if new_pos>=0.6, new_pos=0.6; new_vel=0; end
    x_true = [new_pos; new_vel];
    x_model = A * x_model + B * u_apply;

    % Residual
    residual = abs(curr_pos - (C*x_model));
    if residual < res_bins(1), s = 1; elseif residual < res_bins(2),
        s = 2; else, s = 3; end
    u_prev = u_apply;

    log_y = [log_y; curr_pos]; log_u = [log_u; u_apply]; log_r = [
        log_r; residual]; log_mode = [log_mode; a];
    if curr_pos >= 0.45, break; end
end

figure('Name','RL-MPC Parameter Tuning');
subplot(4,1,1); plot(log_y,'c','LineWidth',1.5); yline(0.45,'r--');
    title('Position'); grid on;
subplot(4,1,2); plot(log_u,'m'); title('Control Input'); grid on;
subplot(4,1,3); plot(log_r,'r'); title('Residual (RL State)'); grid
    on;
subplot(4,1,4); plot(log_mode,'m'); title('Selected Parameter Set (
    Mode)'); grid on; ylim([0.5 3.5]);

save('data/rl_mpc_results.mat', 'log_y', 'log_u', 'log_r', 'log_mode
    ');
fprintf('Validation Complete.\n');


figure('Name', 'RL Training Progress');
plot(1:N_episodes, reward_history, 'Color', 'r', 'LineWidth', 1);
    hold on;
window_size = 10;
if N_episodes >= window_size
    moving_avg = movmean(reward_history, window_size);
    plot(1:N_episodes, moving_avg, 'b', 'LineWidth', 2);
    legend('Episode Reward', 'Moving Average (Trend)', 'Location', '
        SouthEast');
else
    legend('Episode Reward');
end

xlabel('Episode');
ylabel('Total Reward');
title(['Learning Curve (alpha=' num2str(alpha) ', gamma=' num2str(
    gamma) ')']);
grid on;
yline(5000, 'g--', 'Success Threshold', 'LineWidth', 1.5);
```

E. Part 5: Python Visualization

Code used to generate the final videos in Gymnasium using the graphviz library.

Listing 5. Python Visualization

```python
from graphviz import Digraph
import scipy.io
import numpy as np
import matplotlib.pyplot as plt
import gymnasium as gym
from gymnasium.wrappers import RecordVideo
import os

data_path = '../data/mpc_results.mat'
data = scipy.io.loadmat(data_path)

# Flatten the arrays to 1D for iteration
inputs = data['log_u'].flatten()

# We load positions only for validation of the simulation, but we
    drive the simulation with only the inputs
print(f"Total simulation steps to replay: {len(inputs)}")

env = gym.make('MountainCarContinuous-v0', render_mode='rgb_array')

# Wrap the environment to record video
video_folder = '../videos'
env = RecordVideo(env, video_folder=video_folder,
                  episode_trigger=lambda x: True,
                  name_prefix='baseline_mpc_failure')
# Force the exact initial condition used in MATLAB: x0 = [-0.5, 0.0]
obs, info = env.reset(seed=418)
env.unwrapped.state = np.array([-0.5, 0.0])
print("Starting Simulation")
for i, u in enumerate(inputs):
    action = np.array([u], dtype=np.float32)
    obs, reward, terminated, truncated, info = env.step(action)
    if terminated:
        break

env.close()
print(f"Replay Finished.")

data_path = '../data/rl_mpc_results.mat'
rl_data = scipy.io.loadmat(data_path)

log_y = rl_data['log_y'].flatten()      # Position
log_u = rl_data['log_u'].flatten()      # Control Input
log_r = rl_data['log_r'].flatten()      # Residual
log_mode = rl_data['log_mode'].flatten() # Selected Mode (1, 2, or
    3)
time_steps = np.arange(len(log_y))

env = gym.make('MountainCarContinuous-v0', render_mode='rgb_array')

video_folder = '../videos'
env = RecordVideo(env, video_folder=video_folder,
                  episode_trigger=lambda x: True,
                  name_prefix='rl_mpc_success')

# Force the same initial condition as MATLAB: x0 = [-0.5, 0.0]
obs, info = env.reset(seed=418)
env.unwrapped.state = np.array([-0.5, 0.0])

print("Rendering RL-MPC Solution...")

for i, u in enumerate(log_u):
    action = np.array([u], dtype=np.float32)
    obs, reward, terminated, truncated, info = env.step(action)
    if terminated or i == len(log_u)-1:
        break

env.close()

plt.style.use('default')
fig, ax = plt.subplots(4, 1, figsize=(10, 12), sharex=True)

# 1. Position
ax[0].plot(time_steps, log_y, color='tab:blue', linewidth=2, label='
    Car Position')
ax[0].axhline(0.45, color='tab:red', linestyle='--', linewidth=2,
    label='Goal (0.45)')
ax[0].set_ylabel('Position (m)')
ax[0].set_title('Trajectory: Successful Swing-Up Strategy',
    fontweight='bold')
ax[0].grid(True, linestyle='--', alpha=0.6)
ax[0].legend(loc='lower right')

# 2. Control Input
ax[1].plot(time_steps, log_u, color='tab:orange', linewidth=1.5)
ax[1].set_ylabel('Force (N)')
ax[1].set_title('Control Effort (u)', fontweight='bold')
ax[1].grid(True, linestyle='--', alpha=0.6)

# 3. Residual
ax[2].plot(time_steps, log_r, color='tab:red', linewidth=1.5)
ax[2].set_ylabel('Error Magnitude')
ax[2].set_title('Model Uncertainty Signal (Residual)', fontweight='
    bold')
ax[2].grid(True, linestyle='--', alpha=0.6)

# 4. Mode Switching
ax[3].step(time_steps, log_mode, where='post', color='tab:green',
    linewidth=2)
ax[3].set_yticks([1, 2, 3])
ax[3].set_yticklabels(['Recovery (1)', 'Tracking (2)', 'Stabilize
    (3)'])
ax[3].set_ylabel('RL Mode')
ax[3].set_xlabel('Simulation Steps')
ax[3].set_title('Adaptive Gain Scheduling Policy', fontweight='bold'
    )
```

```python
ax[3].grid(True, linestyle='--', alpha=0.6)

plt.tight_layout()
plt.show()

output_dir = '../images'
if not os.path.exists(output_dir):
    os.makedirs(output_dir)


dot = Digraph('RL_MPC_Compact', comment='Residual-Aware RL-MPC Compact Flow')
dot.attr(rankdir='TB', splines='ortho', nodesep='0.6', ranksep='0.6', fontname='Helvetica')
dot.attr('node', shape='box', style='filled', fontname='Helvetica', fontsize='11', height='0.4')
dot.attr('edge', fontsize='9', fontcolor='dimgrey')

# IDENTIFICATION & DATA
with dot.subgraph(name='cluster_0') as c:
    c.attr(label='1. OFFLINE IDENTIFICATION', style='filled', color='lightgrey', fillcolor='#E3F2FD')

    c.node('SimEnv', 'Gymnasium Env\n(MountainCar-v0)', shape='ellipse', fillcolor='white')
    c.node('DataGen', 'Data Generation\n(PRBS Signal)', shape='note', fillcolor='#FFF9C4')
    c.node('N4SID', 'System Identification\n(N4SID Algorithm)', fillcolor='#BBDEFB')
    c.node('LinearModel', 'Linear Model\n(A, B, C)', shape='component', fillcolor='gold')

    c.edge('SimEnv', 'DataGen', label='Physics')
    c.edge('DataGen', 'N4SID', label='u, y data')
    c.edge('N4SID', 'LinearModel', label='Identify')

# RL
with dot.subgraph(name='cluster_1') as c:
    c.attr(label='2. RL AGENT TRAINING', style='filled', color='lightgrey', fillcolor='#E8F5E9')

    c.node('ResidualCalc', 'Residual Calculation\n|y_meas - y_pred|', shape='diamond', fillcolor='#FFCCBC')
    c.node('RL_Loop', 'Q-Learning Loop\n(State: Residual Levels)', fillcolor='#C8E6C9')
    c.node('Reward', 'Reward Function\n(Energy + Goal)', shape='ellipse', fillcolor='white')
    c.node('QTable', 'Learned Policy\n(Q-Table)', shape='folder', fillcolor='orange')

    c.edge('ResidualCalc', 'RL_Loop', label='State (r_t)')
    c.edge('RL_Loop', 'Reward', label='Action')
    c.edge('Reward', 'RL_Loop', label='Feedback')
    c.edge('RL_Loop', 'QTable', label='Save Policy')

# RL-MPC
with dot.subgraph(name='cluster_2') as c:
    c.attr(label='3. ONLINE RESIDUAL-AWARE CONTROL', style='filled', color='black', fillcolor='white')

    c.node('RealPlant', 'Real Plant\n(Nonlinear Physics)', shape='doublecircle', fillcolor='#FFAB91')
    c.node('Estimator', 'Linear Predictor\n(x_k+1 = Ax+Bu)', shape='box', fillcolor='gold')
    c.node('Comparator', 'Error Check\n(Residual)', shape='circle', width='0.8', fillcolor='#B3E5FC')
    c.node('Agent', 'RL Agent\n(Mode Selector)', fillcolor='orange')
    c.node('MPC', 'MPC\n(Gain Scheduled)', style='filled', fillcolor='#4FC3F7', penwidth='2')

    c.edge('RealPlant', 'Comparator', label='y_meas')
    c.edge('Estimator', 'Comparator', label='y_pred')
    c.edge('Comparator', 'Agent', label='Residual')
    c.edge('Agent', 'MPC', label='Select Mode')
    c.edge('MPC', 'RealPlant', label='u(t)', color='red', penwidth='2.0')
    c.edge('MPC', 'Estimator', label='u(t)', style='dashed')

# Global Connections
dot.edge('LinearModel', 'Estimator', style='dashed', label='Model')
dot.edge('LinearModel', 'ResidualCalc', style='dashed', constraint='false')
dot.edge('QTable', 'Agent', style='bold', label='Load Policy')

output_path = os.path.join(output_dir, 'rl_mpc_workflow')
dot.render(output_path, format='png', cleanup=True)
dot
```