A collection of selected research and engineering projects demonstrating a strong foundation in **Optimal Control**, **Learning-based Dynamics**, and **Robotic Simulation**. These works highlight my ability to bridge the gap between rigorous mathematical formulations and real-world system implementation. Github Link

Selected Projects

01. Fast Differential Dynamic Programming for Time-Optimal Trajectory Planning (P2-P6)

> **Focus:** Optimal Control, Trajectory Optimization, Robotics
> **Tech Stack:** C++, Control Theory, Numerical Optimization

- **Innovation:** Addressed the limitations of fixed-horizon formulations in DDP/iLQR by introducing a novel method to jointly optimize control sequences and the planning horizon (T).
- **Result:** Achieved global optimality for time-optimal tasks significantly faster than current SOTA approximations (OnePass methods), avoiding local minima in complex cost landscapes.

02. Neural ODE for Planar Pushing Dynamics (P7-P11)

> **Focus:** Deep Learning, Dynamics Modeling, Contact-Rich Manipulation
> **Tech Stack:** PyTorch, Neural ODEs, Model-Based RL

- **Innovation:** Investigated continuous-time dynamics modeling for contact-rich planar pushing tasks using Neural Ordinary Differential Equations (Neural ODEs).
- **Result:** Demonstrated superior performance over discrete Residual Networks in multi-step trajectory prediction, effectively capturing the complex, non-smooth dynamics of robotic manipulation.

03. SimpleNet Extension: Adversarial Training for Anomaly Detection (P12-P14)

> **Focus:** Computer Vision, Unsupervised Learning, Adversarial Attacks
> **Tech Stack:** Python, Deep Learning, Adversarial Training

- **Innovation:** Enhanced the SimpleNet architecture by replacing random Gaussian noise with targeted adversarial noise during training to force tighter decision boundaries.
- **Result:** Improved anomaly localization performance (P-AUROC) by **3.6%** on the MVTec AD dataset, significantly reducing false positives in industrial defect detection.

04. Urban Construction Zone Generator for AV Testing (P15-P19)

> **Focus:** Software Engineering, Autonomous Driving Simulation, Procedural Generation
> **Tech Stack:** Python, SUMO, TraCI, TomTom API

- **Innovation:** Designed a scalable simulation pipeline to automatically generate MUTCD-compliant construction zones, addressing the data scarcity of corner cases in AV testing.
- **Result:** Successfully deployed within the **Mcity TeraSim** platform, enabling safety testing for autonomous vehicles in diverse, procedurally generated work zone scenarios.

Fast Differential Dynamic Programming for Time-Optimal Trajectory Planning

Research Portfolio

Miaomiao Dai

https://github.com/dmmsjtu-umich/time-opt-ilqr

1 Overview

Time-optimal trajectory planning—generating motions that complete a task in the minimum possible time—is a fundamental requirement for agile robotic systems. From autonomous drone racing to emergency collision avoidance in self-driving cars, the ability to jointly optimize the control sequence and the total maneuver duration T is critical for pushing physical limits.

While Differential Dynamic Programming (DDP) and its variant, the iterative Linear Quadratic Regulator (iLQR), have become standard tools for high-dimensional trajectory optimization, they typically assume a fixed planning horizon. Extending these methods to time-optimal control introduces a discrete-continuous optimization challenge: the solver must determine the optimal integer horizon T^* alongside the continuous control inputs.

2 Problem Formulation

We consider the discrete-time optimal control problem with variable horizon:

$$\begin{aligned} \min_{U_T, T} \quad & J = \phi(x_T) + \sum_{k=0}^{T-1} \ell(x_k, u_k) + wT \\ \text{s.t.} \quad & x_{k+1} = f(x_k, u_k), \quad k = 0, \dots, T-1 \\ & x_0 = \bar{x}_0, \quad T \in \{1, 2, \dots, N\} \end{aligned} \tag{1}$$

where the decision variables include both the control sequence $U_T = \{u_0, \dots, u_{T-1}\}$ and the finish time T. The term wT penalizes the planning horizon, encouraging time-optimal behaviour of the system.

3 The Challenge

A fundamental bottleneck lies in the structure of the standard Riccati recursion. In the LQR backward pass, the Value Function V_k is computed recursively starting from a terminal cost anchored at the final time step T (i.e., $P_T = Q_T$). Consequently, changing the horizon from T to $T+1$ shifts the boundary condition, invalidating the entire sequence of previously computed Cost-to-Go matrices. This structural dependency prevents the reuse of historical computations across different horizons, forcing the solver to restart the backward pass from scratch for each candidate T, resulting in a prohibitive $\mathcal{O}(N^2)$ complexity.

4 Our Approach: Time-Varying Propagator

To address the loss of reusability in time-varying systems, we shift from *reusing values* to *reusing mappings*.

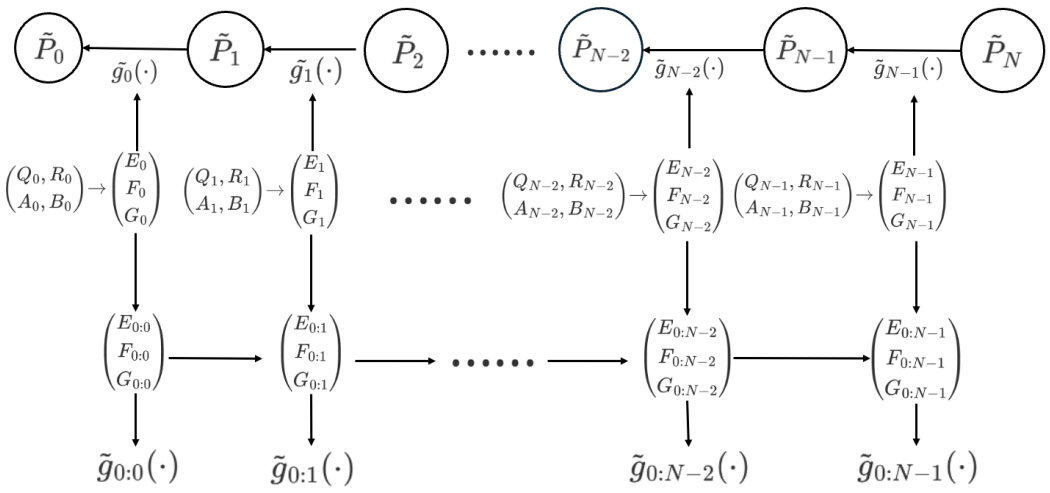


Figure 1: **Time-varying propagator.** When g_k varies with k, we switch to inverse form where each stage is an LFT $\tilde{g}_k$. The composed map $\tilde{g}_{0:k}$ remains an LFT with prefix parameters $(E_{0:k}, F_{0:k}, G_{0:k})$. This enables cheap horizon queries by reusing the composed mapping $\tilde{g}_{0:k}$ instead of reusing $\tilde{P}_k$ values.

Our key idea (Fig. 1) is to rewrite the map g_k as a new linear fractional transformation (LFT) form $\tilde{g}_{0:k}$, and some of the matrices that help compute $\tilde{g}_{0:k}$ can be reused. As a result, these matrices only need to be computed once for all possible horizons $k = 1, 2, \cdots, N$, as opposed to be repetitively computed for each possible horizon, which thus saves computational effort.

4.1 Linear Fractional Transformation Form

Let $\tilde{P}_k := P_k^{-1}$ denote the inverse matrix of P_k. Let notation $\tilde{g}_{0:k} = \tilde{g}_0 \circ \cdots \circ \tilde{g}_k$ denote a *composed map* that composes the maps $g_0, g_1, \cdots, g_k$ sequentially.

Theorem 1 (LFT Form)**.** *There exist matrices $(E_{0:k}, F_{0:k}, G_{0:k})$, $k = 0, 1, 2, \cdots, N$ such that*

$$\tilde{g}_{0:k}(\tilde{P}) = E_{0:k} - F_{0:k}(\tilde{P} + G_{0:k})^{-1} F_{0:k}^\top, \tag{2}$$

where the prefix parameters obey the recursion (for $k \geq 1$):

$$\begin{aligned} W_k &= (E_k + G_{0:k-1})^{-1}, \\ E_{0:k} &= E_{0:k-1} - F_{0:k-1} W_k F_{0:k-1}^\top, \\ F_{0:k} &= F_{0:k-1} W_k F_k, \\ G_{0:k} &= G_k - F_k^\top W_k F_k, \end{aligned} \tag{3}$$

with base case $E_{0:0} = E_0$, $F_{0:0} = F_0$, $G_{0:0} = G_0$, and stage parameters:

$$E_k = Q_k^{-1}, \quad F_k = Q_k^{-1} A_k^\top, \quad G_k = A_k Q_k^{-1} A_k^\top + B_k R_k^{-1} B_k^\top. \tag{4}$$

4.2 Efficient Query for All Arrival Times

Given the prefix triple at $t-1$, the initial inverse for any candidate arrival time t is:

$$\tilde{P}_0^{(t)} = E_{0:t-1} - F_{0:t-1}(\tilde{P}_t + G_{0:t-1})^{-1}F_{0:t-1}^\top. \tag{5}$$

Then $P_0^{(t)} = (\tilde{P}_0^{(t)})^{-1}$ and the cost for horizon t is:

$$J_t = \tfrac{1}{2}x_0^\top P_0^{(t)} x_0 + wt. \tag{6}$$

Thus all $\{J_t\}_{t=1}^N$ are obtained from a single forward prefix build plus N terminal updates.

Complexity. The propagator has the same order as a single LQR backward sweep, $\mathcal{O}(Nn^3)$. Brute forcing all horizons by re-solving Riccati is $\mathcal{O}(N^2n^3)$.

4.3 Augmented State Formulation for iLQR

To extend the propagator to nonlinear iLQR, we introduce an **Augmented State Formulation** that absorbs the time-varying affine linearization terms into a homogeneous coordinate system:

$$z_k = \begin{bmatrix} \delta x_k \\ 1 \end{bmatrix}. \tag{7}$$

The augmented system matrices become:

$$A_k^{\text{aug}} = \begin{bmatrix} A_k - B_k\ell_{uu,k}^{-1}\ell_{ux,k} & -B_k\ell_{uu,k}^{-1}\ell_{u,k} \\ 0 & 1 \end{bmatrix}, \quad B_k^{\text{aug}} = \begin{bmatrix} B_k \\ 0 \end{bmatrix}. \tag{8}$$

This unifies the treatment of linear and nonlinear problems, allowing the propagator to compute the *exact* LQR cost for all horizons in a single $\mathcal{O}(N)$ pass.

5 Main Contributions

1. **Propagator-based Horizon Selection:** We develop an LFT-based solver that enables the reuse of backward pass computations, reducing the complexity of horizon selection from $\mathcal{O}(N^2n^3)$ to $\mathcal{O}(Nn^3)$.

2. **Augmented State Formulation:** We propose a state augmentation technique that embeds affine linearization terms into a homogeneous coordinate system, extending the efficient propagator method to general nonlinear iLQR problems.

3. **Performance and Robustness:** We validate our algorithm on four benchmark systems, including a 12-DOF Quadrotor. Experimental results show that our method achieves speedups of up to **43**$\times$ compared to brute-force search while guaranteeing global optimality with respect to the linearized model.

6 Results

We validate our proposed Propagator-based iLQR on four benchmark systems: Double Integrator, Segway Balance, Cartpole Swing-Up, and a 12-DOF Quadrotor. We compare three horizon-selection strategies:

- **Ours (Propagator):** The proposed method using the Augmented State Propagator.
- **Baseline-1 (Bruteforce):** Evaluating all horizons via standard backward Riccati sweeps. This serves as the ground truth but has $\mathcal{O}(N^2)$ complexity.
- **Baseline-2 (OnePass):** The current state-of-the-art approximate method, which estimates costs for neighboring horizons by reusing the value function from a single nominal backward pass. While efficient, it is only an *approximation* that can introduce significant errors for time-varying systems.

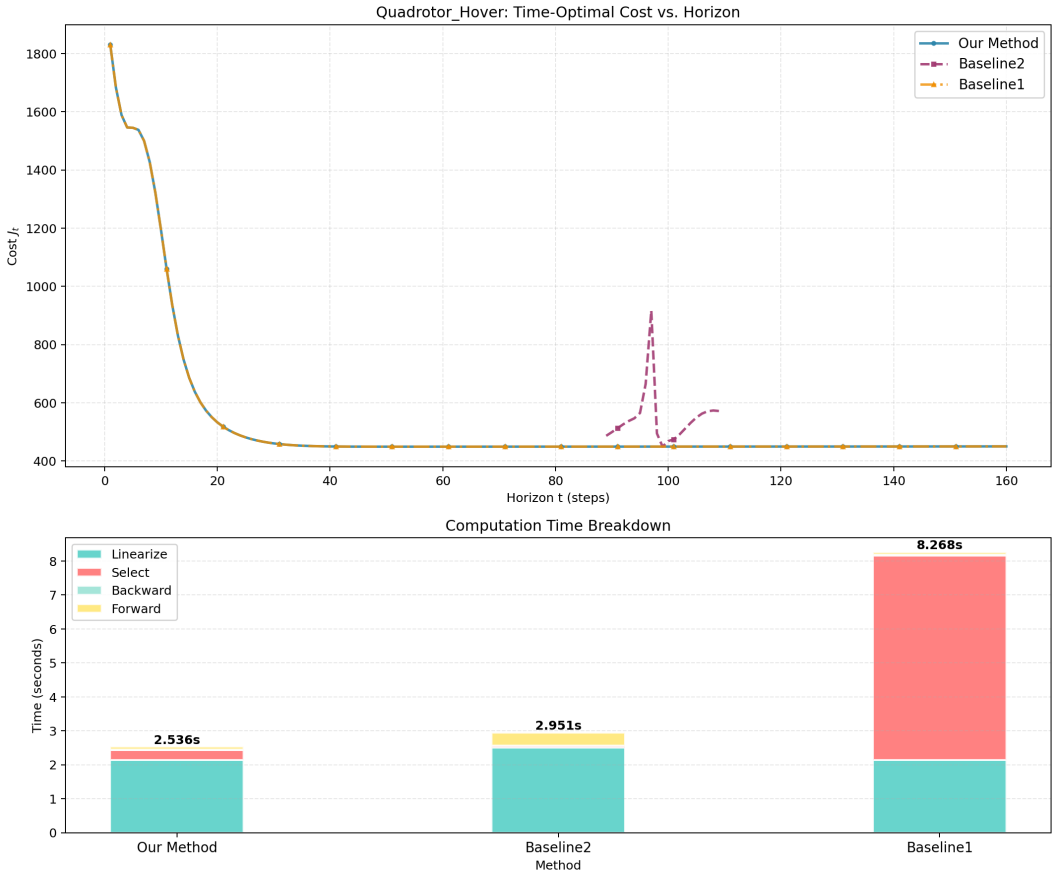


Figure 2: **Case Study on Quadrotor Hover.** (Top) Comparison of cost landscapes (J_t) computed by different methods. (Bottom) Breakdown of total runtime into linearization, selection, backward, and forward phases.

Cost Landscape (Top Panel). The top panel of Fig. 2 compares the cost curves (J_t vs. horizon t). The OnePass method (Purple), as the current SOTA, approximates the cost landscape by

projecting the value function from a single nominal horizon. However, because it is merely an *estimation* method, it suffers from severe distortions when the system dynamics vary significantly over time (e.g., the artifact spike near $t = 82$). Consequently, OnePass converges to a wrong local minimum, failing to find the true optimal horizon.

In contrast, our Propagator curve (Blue) overlaps *perfectly* with the Bruteforce ground truth (Yellow). This confirms that our augmented formulation correctly captures the *exact* time-varying LQR cost—not an approximation—allowing the solver to locate the true global optimum.

Runtime Breakdown (Bottom Panel). The bottom panel decomposes the runtime to reveal the source of efficiency. The Bruteforce method (Baseline-1) computes the exact cost but at prohibitive expense—the massive "Select" phase (Red) represents the $\mathcal{O}(N^2)$ cost of repeated Riccati sweeps, taking **8.21s** total.

The OnePass method (Baseline-2) is faster (**2.88s**) but, as shown above, produces incorrect results due to its approximate nature.

Our Propagator method achieves the best of both worlds: it computes the *exact* cost like Bruteforce while running even faster than the approximate OnePass (**2.46s**). By compressing the horizon evaluation into an $\mathcal{O}(N)$ LFT propagation, we effectively eliminate the selection bottleneck while maintaining rigorous optimality.

Neural ODE for Planar Pushing Dynamics

Siyuan Li, Miaomiao Dai
GitHub: https://github.com/dmmsjtu-umich/Neural-ODE.git

***Abstract*—We tune the hyperparameter such as intergration step/method and hidden dimensions/layers of Neural ODE–based model for planar pushing dynamics. We use the best hyperparameter to train a single step and mutiple step Neural ODE Model. Compared to the baseline residual neural network, our model achieves a lower validation loss and generates a trajectory that more closely matches the ground truth. Its effectiveness is also demonstrated through closed-loop control with an MPPI controller to push block towards the target pose.**

I. Introduction

Accurate modeling of contact-rich interactions, such as planar push, is crucial for robotic manipulation tasks in cluttered or dynamic environments. These dynamic models are usually hard to derive a analytic form due to the complex relationship and discontinuous dynamics. Learning-based approaches, such as residual neural networks, have been shown to effectively learn dynamic models and provide accurate predictions. However, these models discretize dynamics and implicitly assume that each step corresponds to a fixed time interval jump, which is not true for a continuous-time dynamics. Also, the progression over time cannot be explicitly controlled which makes it lack flexibility. Neural ODEs solve these problems by learning a differential equationwhich operates in continuous time and allows flexible control over the evolution of the dynamics.

In this work, we tune the hyperparameters of Neural ODE-based dynamics model and train a single step dynamic model and multiple step dynamic models based on the best hyperparameters. We compare our Neural ODE Model with the baseline model in different aspects and find that our Neural ODE Model has higher accuracy. Also, we demonstrate its effectiveness through closed-loop control with an MPPI controller.

II. Implementation

A. Planar Pushing Environment

For our planar pushing setup, we will use PyBullet and a custom Gym environment to simulate a Franka Panda robot arm that pushes a block on top of a table. The pushing block, the goal pose and the obstacle are visualized in white, green and brown respectively.

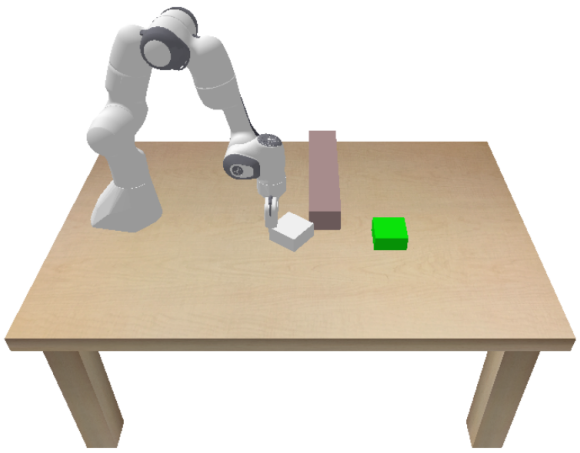


Fig. 1. Planar Pushing Environment

B. Model Architecture

Figure 2 illustrates the core Neural ODE architecture for the planar pushing task:

1) **Input $[x_t, u_t]$:**
   - $x_t \in \mathrm{SE}(2)$: the object's current pose (x, y, θ).
   - $u_t \in \mathbb{R}^3$: the push action parameters, consisting of:
     - $p \in [-1, 1]$: pushing location along the lower block edge.
     - $\phi \in [-\frac{\pi}{2}, \frac{\pi}{2}]$ pushing angle.
     - $\ell \in [0, 1]$ pushing length as a fraction of the maximum pushing length. The maximum pushing length is 0.1 m
2) **ODEFunc $f_\theta(x, u)$:**
   - A multilayer perceptron that maps the concatenated state and action $[x_t, u_t] \in \mathbb{R}^6$ to the instantaneous state derivative $\dot{X} \in \mathbb{R}^3$.
3) **ODE Solver (`odeint_adjoint`):**
   - Numerically integrates $\dot{x} = f_\theta(x, u)$ over the time interval $[0, \Delta t]$ using a chosen method (e.g., Euler, RK4, Bilinear Optimal Search Heuristic 3, Dormand-Prince 5). The input ac-

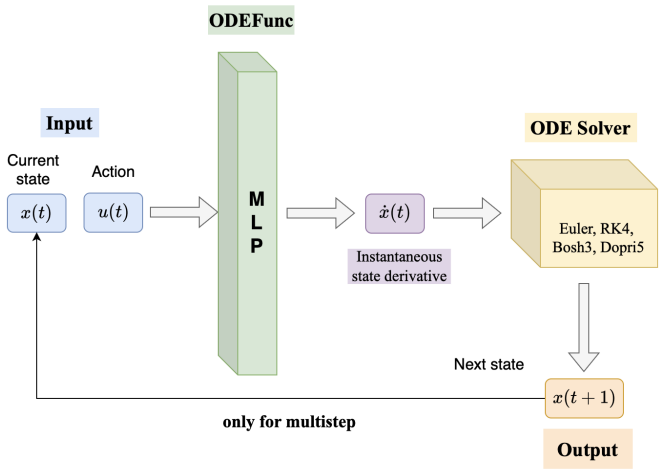


Fig. 2. Neural ODE architecture for planar pushing.

tion u is kept fixed because, in a Gym environment, the action remains constant throughout each step.

- Returns the evolved state at $t = \Delta t$, which is our prediction x_{t+1}.

4) **Output x_{t+1}:**

- The solver's final state, representing the model's prediction of the object's next pose.

During training, the adjoint sensitivity method is used automatically for backpropagation, keeping memory usage constant with respect to the number of integration steps and allowing flexible control over numerical precision and network complexity.

C. ODEFunc $f_\theta(x, u)$

We define

$$f_\theta : \mathbb{R}^6 \to \mathbb{R}^3, \quad \dot{x} = f_\theta(x_t, u_t)$$

via a multilayer perceptron:

$$\begin{aligned} h_1 &= \mathrm{ReLU}(W_1[x_t; u_t] + b_1)\,, \\ h_i &= \mathrm{ReLU}(W_i h_{i-1} + b_i) \quad (i = 2 \dots L)\,, \\ \dot{x} &= W_{L+1} h_L + b_{L+1}\,. \end{aligned}$$

Note that the input of ODEFunc actually only needs $x_t \in \mathbb{R}^3$ since it will be concatenated with a fixed action during Δt. This network directly outputs the instantaneous velocity $\dot{x}$.

D. Training

We collect dataset by sampling trajectories in Panda Pushing Environment. For single step dataloader, the input are current state and action and the target is the next state. For mutiple step dataloader, the input are current state and following actions, and the target are the following states. We use Adam Optimizer with fixed learning rate as the optimizer. For hyperparameters, we use the method of controlling variables to tune them one by one in order to find the optimal set of hyperparameters.

To train the dynamics model, we minimize a weighted quadratic loss between the predicted and ground-truth next state, which we call SE2PoseLoss. For predicted next state $\mathbf{q}_p = \begin{bmatrix} x_p & y_p & \theta_p \end{bmatrix}^\top$ and ground-truth next state $\mathbf{q}_g = \begin{bmatrix} x_g & y_g & \theta_g \end{bmatrix}^\top$, the loss function is

$$\mathcal{L}(\mathbf{q}_p, \mathbf{q}_g) = \mathrm{MSE}(x_p, x_g) + \mathrm{MSE}(y_p, y_g) + r\mathrm{MSE}(\theta_p, \theta_g)$$

$$r = \sqrt{\frac{w^2 + l^2}{12}}$$

The multiple step loss is defined as follow

$$\mathcal{L} = \sum_{i=0}^{i=H} \gamma^i SE2PoseLoss(q_{pi}, q_{gi})$$

where $\gamma \in (0, 1)$ is the discount factor.

E. Control

To demonstrate the precision of our Neural ODE Model in real world situation. We use our Neural ODE Model as the dynaic model and MPPI as the controller to direct the robot arm to push block towards the target pose. The cost function is defined as follow:

$$\begin{aligned} \mathrm{Cost}(\mathbf{x}_1, ..., \mathbf{x}_T) = \sum_{t=1}^{T} \big(&(\mathbf{x}_t - \mathbf{x}_{goal})^T Q (\mathbf{x}_t - \mathbf{x}_{goal}) \\ &+ 100 \texttt{in\_collision}(\mathbf{x}_t) \end{aligned}$$

where the weighting matrix Q is given by

$$Q = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0.1 \end{bmatrix}.$$

III. Results and Discussion

A. Integration Step

When dealing with non-adaptive integration methods, the time interval Δt needs to be divided into many small steps so that the numerical methods approximate the solution step by step moving forward in small increments. We investigate the influence of the number of integration steps on the validation loss. We use Euler as the integration method since it's much faster than RK4 and easier to do experiments. We use 0.005 as the learning rate to accelerate training, 128 as the hidden

dimension and 2 as the hidden layers. Fig. 3 displays validation loss with different number of integration steps.

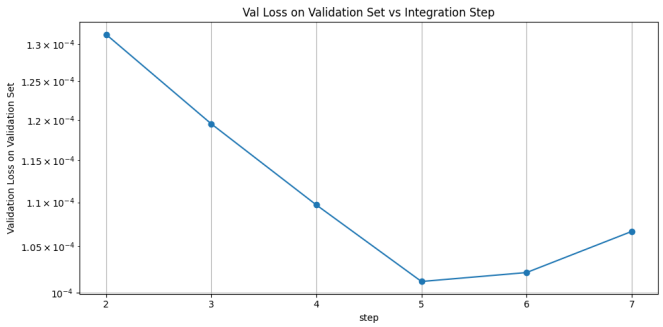


Fig. 3. Validation Loss vs Number of Integration Steps

From step=2 to step=5, the validation loss decreases. This is because more steps lead to smaller errors per step and the ode solver can have a better prediction of the state at the end of the time interval. And the validation loss begins to increase slightly starting from Step = 5. This may because the model now needs to be extremely precises over very short time intervals, and give accurate estimation of dynamic derivative for inputs that don't differ much. In this case, the model becomes to sensitive to small errors and thus the performance drops. Overall, the best integration step is 5.

B. Network Expressivity

Hidden dimensions and the number of hidden layers are the most important factors that influence the network expressivity. We study the effect on validation loss by changing the number of hidden layers [2,3] and varying the hidden dimension among [16,64,128,256]. We use Euler method and the best number of integration of steps obtained from section A for all architectures. Learning rate is set to 0.005 to accelerate training. Fig. 4 displays the validation loss using different hidden dimensions and number of hidden layers.

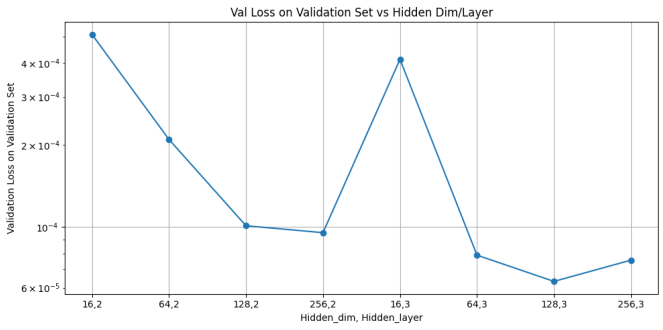


Fig. 4. Validation Loss vs Hidden Dimensions/Layers

When hidden layer=2, the validation loss decreases as the hidden dimension increases. This is because larger hidden dimension can capture more complex relationship and has stronger network expressivity. When hidden layer=3, the validation loss decreases as the hidden dimension increases from 16 to 128. However, the validation loss increases as the hidden dimension increases from 128 to 256. Our explanation for this is when that the number of parameters of the neural network architecture reaches a certain amount, the model begins to overfit. The best hidden dimension is 128 and the best number of hidden layers is 3.

C. Integration Method

We use four different integration methods (Euler, RK4, BOSH3, DoPri5) as the method of ode solver and compare their final validation loss. Euler and RK4 are non-adaptive methods and we use the best number of integration steps obtained from section A. BOSH3 and DoPri5 use adaptive algorithms and so the number of integration steps don't matter. We use 0.005 as the learning rate to accelerate training, and the best hidden dimension and layers obtained from section B for all integration methods. TABLE I shows lowest validation loss using different integration methods.

Integration Method	**Validation Loss**
Euler	1.1812×10^{-4}
RK4	1.1026×10^{-4}
BOSH3	1.1238×10^{-4}
DoPri5	9.7306×10^{-5}

TABLE I
VALIDATION LOSS FOR DIFFERENT INTEGRATION METHODS.

It turns out that DoPri5 has the lowest validation loss, RK4 and BOSH3 have the similar validation loss and Euler has the largest validation loss. The best integration method is DoPri5. However, the performance of different integration methods don't differ much. So we will use Euler method as our chosen model for the following sections to enable faster training.

D. Learning Rate

Different learning rate are investigated to choose the best one. For learning rate in [0.01, 0.005, 0.003, 0.001, 0.0005, 0.0001], we find the corresponding number of epochs to get the lowest validation loss is [500, 800, 1200, 2000, 3000, 8000]. Euler is used as the integration method. We use the best hidden dimensions and the best number of hidden layers obtained from section B for all learning rates. Fig. 5 shows the best validation loss that different learning rates can achieve.

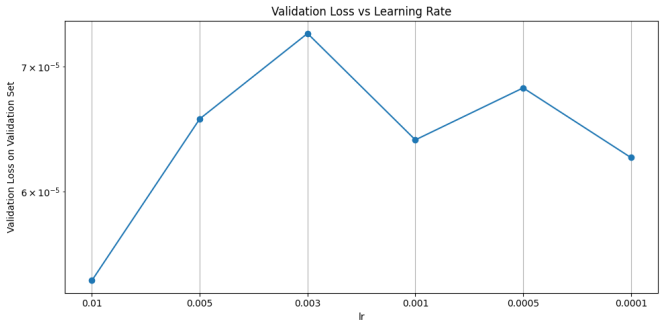


Fig. 5. Validation Loss vs Learning Rate

It turns out that when learning rate is 0.01, it has the lowest validation loss. And from 0.005 to 0.0001, the lowest validation loss don't vary much. Actually, This is inconsistent with our expectations. We had expected that smaller learning rate will contribute to lower validation loss. Our explanation for this is that small learning rates can get stuck in the local minimum. And 0.01 is the right choice since it is able to jump out of local minimum and is not too large to avoid overshooting.

E. Baseline Single Step Model

We use the residual neural network with hidden dimension = 100 and hidden layers = 2 as our baseline model. Learning rates and the number of epochs have been tuned so that the baseline model can achieve its lowest validation loss. They are 0.001 and 1000 respectively.

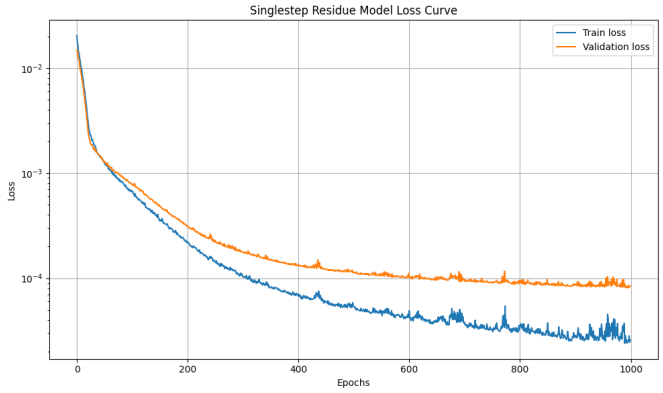


Fig. 6. Loss Curve of Residue Dynamic Mode

The baseline model achieves a validation loss of 9.525×10^{-5}.

F. Best Tuned Single Step Neural ODE Model

The best hyperparameters obtained from previous sections are used to train the single step ode model. They are listed in TABLE II.

Integration Step	Method	Hidden Dim	Hidden Layer	LR
5	Euler	128	3	0.01

TABLE II
CHOSEN HYPERPARAMETER

The training and validation loss curves are shown in Fig. 7.

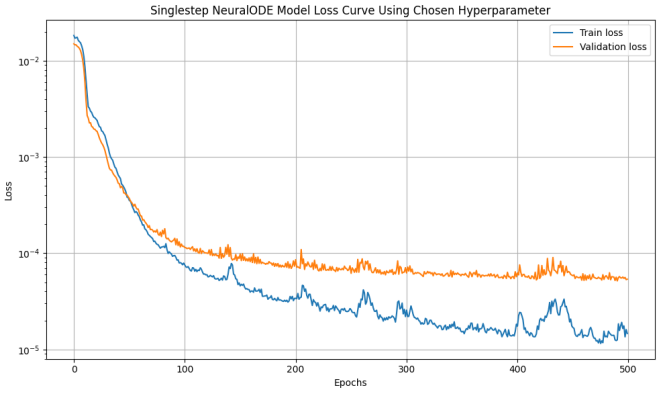


Fig. 7. Loss Curves of Single Step ODE Model

It achieves a validation loss of 5.892×10^{-5}, which is lower than our baseline model's 9.525×10^{-5}.

To evaluate the real-world accuracy of next-state prediction by the Single-Step Neural ODE model, we integrate it as the dynamics model within an MPPI controller and deploy MPPI to push the box toward a target pose in an obstacle PandaPushingEnv environment. It turns out that the goal is reached in all of the ten trials.

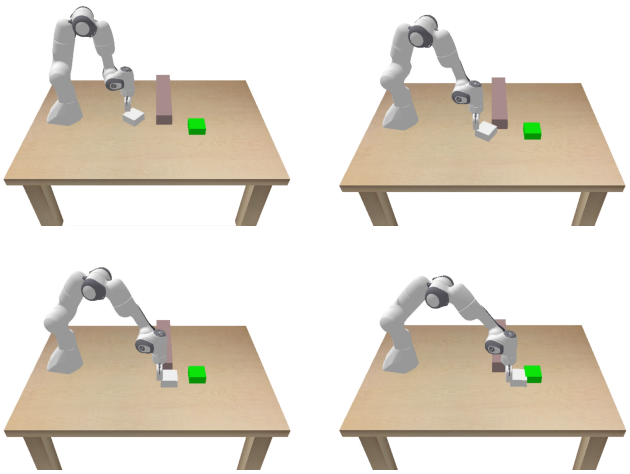


Fig. 8. Pushing Box

G. Mutiple Step Neural ODE Model

The performance of mutiple step Neural ODE Model are evaluated and compared with mutiple step Residue Model. We use the best hyperparameters from TABLE II except for a smaller learning rate to guarantee stable gradient descent since it's mutiple step loss involves more complex relationship. Then we evaluate both Mutistep Neural ODE Model and Mutistep Residue Model on a held out validation set. The results of comparison are shown in Fig. 9. The validation loss uses mutiple step loss.

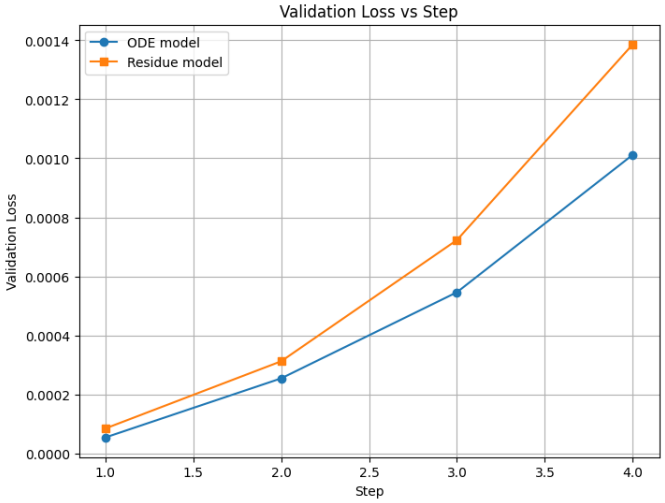


Fig. 9. Compare Mutistep Neural ODE Model and Mutistep Residue Model

At each step in [1,2,3,4], Mutistep Neural ODE Model achieves a lower validation loss than Mutistep Residue Model. This demonstrates the Neural ODE model's ability to perform multi-step predictions.

H. Trajectory Comparison

To visualize the distinct performance of Neural ODE Model and Residue Model in predicting next state, we use PandaPushingEnv to sample actions and generate a groud truth trajectory. At each step, we use Mutistep Neural ODE Model and Mutistep Residue Model to predict next state given the current state and sampled action. Their predicted trajectories and the ground truth trajectory are shown in Fig. 10.

For x coordinate, Neural ODE Model has more accurate prediction. For y coordinate, two models have similar prediction accuray. For θ coordinate, the Neural ODE model achieves significantly higher precision. These observations offer insight into why Neural ODE Model can achieve lower validation loss than Residue Model.

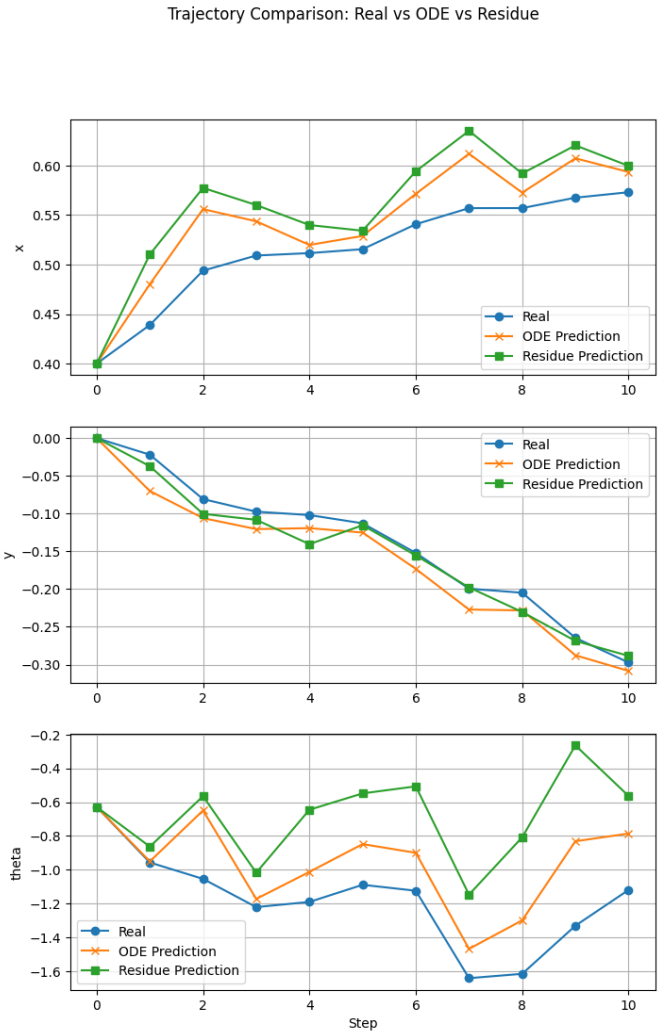


Fig. 10. Compare Trajectory

IV. Conclusion

We use the method of controlling variables to tune hyperparameters one by one in order to find the optimal set. Based on the best hyperparameter, we trained single step and mutiple step Neural ODE Model. It achieves a lower validation loss than the baseline residue neural network model. Its effectiveness as a dynamics model is validated by successfully using an MPPI controller to push the block to the target.

References

[1] R. T. Q. Chen, Y. Rubanova, J. Bettencourt, and D. Duvenaud, "Neural ordinary differential equations," in *Advances in Neural Information Processing Systems*, 2018, pp. 6571–6583.

SimpleNet Extension: Adversarial Training for Anomaly Detection

Miaomiao Dai
Role: Designed and implemented adversarial noise strategy for SimpleNet
University of Michigan **dmmsjtu@umich.edu**
Project Link: https://github.com/dmmsjtu-umich/SimpleNet_Extension

Abstract

We introduce an adversarial training strategy to SimpleNet to improve anomaly localization. By replacing random Gaussian noise with targeted adversarial noise, we force the discriminator to learn tighter boundaries. In this project, experimental results on MVTec AD show that this approach achieves **+3.6% P-AUROC** (87.8% → 91.4%) improvement with reduced training epochs, demonstrating better spatial precision in anomaly detection.

1. Motivation and Related Work

What SimpleNet Does SimpleNet (Liu et al. 2023) generates synthetic anomalies by adding **random Gaussian noise** to all 1536 feature dimensions equally.
Problem: Random noise explores all directions equally, but only directions **near the decision boundary** are informative for training.

Evidence from Adversarial Learning Goodfellow, Shlens, and Szegedy (2015) showed that adversarial examples (found via gradients) are more effective for robustness training than random perturbations. Madry et al. (2018) demonstrated that PGD adversarial training creates tighter decision boundaries. Miyato et al. (2018) proposed Virtual Adversarial Training (VAT) for semi-supervised learning.
Key insight: If adversarial directions improve classifier boundaries, can they improve anomaly discriminator boundaries?

2. Method

Approach Overview To address the limitations of random exploration, we propose replacing completely random noise with **targeted noise** that moves features toward the decision boundary.
After the Feature Adaptor outputs feature vector q, our process is as follows:

1. Compute gradient: $g = \nabla_q D(q)$, which represents the direction that most significantly changes the discriminator score.
2. Generate adversarial anomalies: $q^- = q + \alpha \cdot \text{sign}(g)$.

Why it matters:

- Random noise explores all 1536 directions equally, often wasting capacity on irrelevant areas.
- Adversarial noise focuses on the "hardest" directions where the discriminator is most uncertain.
- This forces the discriminator to learn tighter boundaries by automatically targeting the weakest parts of the current boundary.

To prevent overfitting to discriminator weaknesses, we employ a **mixed strategy**: 50% adversarial noise (exploitation of weak boundaries) and 50% random Gaussian noise (exploration for diversity).

Training Algorithm The detailed training procedure is described in Algorithm 1. We modify the noise generation step to probabilistically select between Gaussian noise and Gradient-based adversarial noise.

Algorithm 1 Mixed Adversarial Noise Strategy

Require: Feature vector q, Discriminator D, magnitude α, ratio γ
$p \sim \mathcal{U}(0, 1)$
if $p < \gamma$ **then**
  {Adversarial Perturbation}
  $g \leftarrow \nabla_q D(q)$
  $\delta \leftarrow \alpha \cdot \text{sign}(g)$
else
  {Gaussian Perturbation}
  $\delta \sim \mathcal{N}(0, I) \cdot \alpha$
end if
$q^- \leftarrow q + \delta$
$\mathcal{L} \leftarrow \text{BCE}(D(q), 0) + \text{BCE}(D(q^-), 1)$
return $\mathcal{L}$

Key difference: 50% of training uses gradients to find challenging perturbations (FGSM-like attack), which sharpens the decision boundaries compared to the pure Gaussian baseline.

3. Experiments and Results

Experiment Setup **Dataset**: MVTec AD (15 categories, ∼3500 test images)

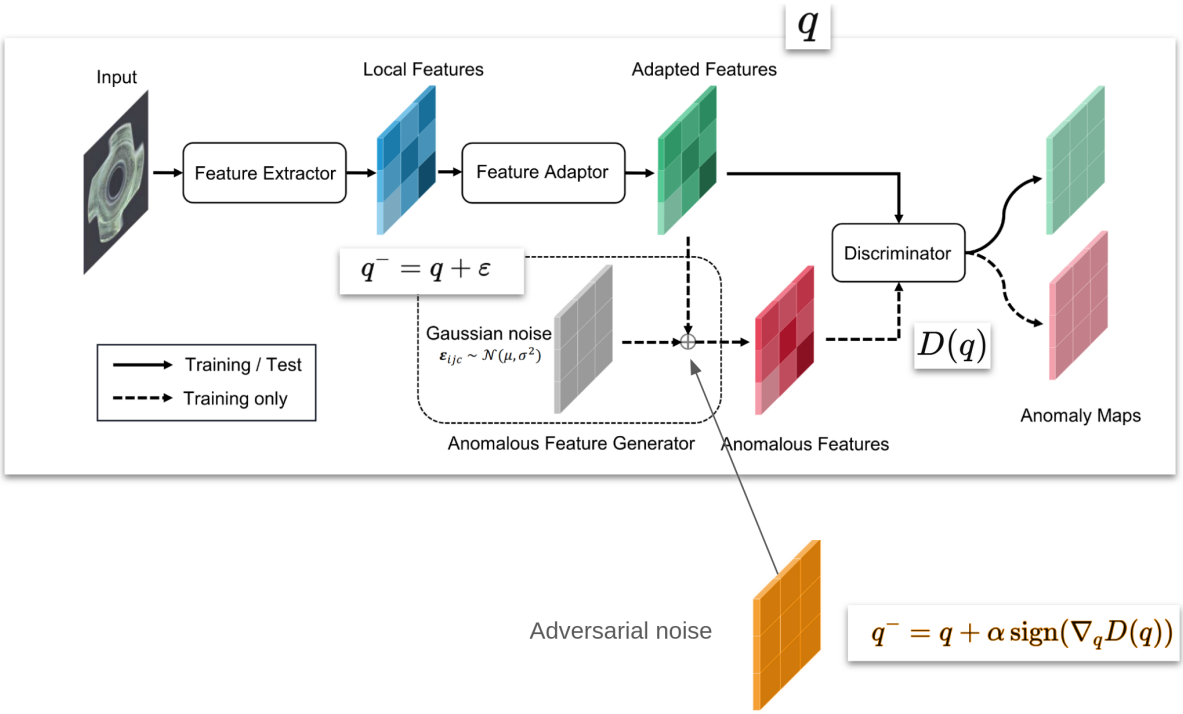


Figure 1: Approach Overview: Adding adversarial noise into the Anomalous Feature Generator.

Hardware: NVIDIA RTX 3070ti
Hyperparameters: We reduce the training epochs from 160 to 40 for both methods while keeping other parameters unchanged. This modification serves two purposes: (1) it increases experimental efficiency, and (2) it demonstrates that Extension 2 can improve model performance in early training stages.
Evaluation Metrics: We use two standard metrics to evaluate performance. **I-AUROC (Image-level AUROC)** measures the model's ability to classify entire images as normal or anomalous, evaluating overall detection capability. **P-AUROC (Pixel-level AUROC)** measures pixel-wise anomaly localization accuracy, assessing how precisely the model identifies defect locations. Both metrics range from 0-100%, with higher values indicating better performance. P-AUROC is particularly important for industrial applications requiring precise defect localization.

Table 1: Average performance across all MVTec categories

Metric	Baseline	Extension 2	Improvement
I-AUROC	95.81%	**96.76%**	**+0.96%**
P-AUROC	87.80%	**91.40%**	**+3.60%**

Overall Performance **Key finding**: Extension 2 achieves consistent improvements on both metrics, with the most significant gain in pixel-level localization (P-AUROC: +3.60%), indicating that mixed adversarial noise substantially enhances spatial precision in anomaly detection.

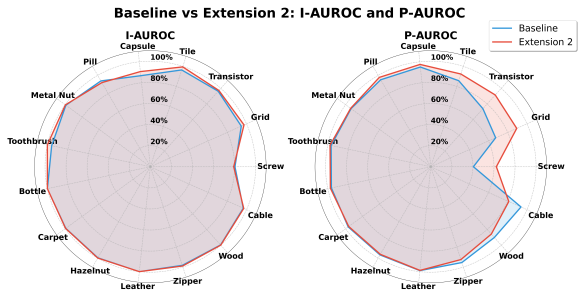


Figure 2: Category-wise performance comparison. Left: I-AUROC (image-level), Right: P-AUROC (pixel-level).

Category-wise Comparison Figure 2 presents radar charts comparing performance across all 15 categories, where each axis represents one category and the radial distance indicates AUROC percentage. At the image level (I-AUROC, left), both methods achieve strong performance exceeding 80% for most categories. Extension 2 shows notable improvements on Grid (+2.84%), Tile (+3.03%), and Transistor (+1.00%), with slight decreases on Pill (-1.86%) and Screw (-1.19%) that do not significantly impact overall effectiveness.

The pixel-level comparison (P-AUROC, right) reveals more substantial differences. Extension 2 demonstrates dramatic improvements on object categories with localized defects: Screw (+21.79%), Grid (+21.92%), Transistor (+17.51%), and Tile (+6.64%). However, moderate degradation appears on texture categories including Wood (-4.32%),

Zipper (-2.96%), and Cable (-12.70%). Despite these decreases, Extension 2 achieves higher average P-AUROC (91.40% vs 87.80%), validating the effectiveness of mixed adversarial noise for precise anomaly localization.

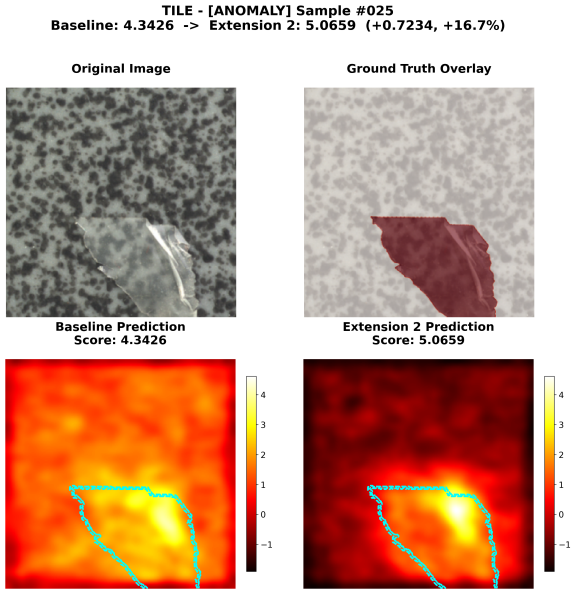


Figure 3: Tile #25 visualization. Top: Original and ground truth (red overlay). Bottom: Baseline and Extension 2 heatmaps.

Case Study: Tile #25 (Crack Defect) Figure 3 provides a qualitative comparison on a **glue strip defect**. In the heatmaps, pixel intensity correlates with anomaly confidence: darker regions are normal, while brighter regions suggest anomalies.

While the Baseline model correctly detects the presence of the defect, its activation map is **scattered and imprecise**, extending into the clean texture regions. Extension 2, however, generates a **highly concentrated activation map** that tightly hug the contours of the glue strip, matching the ground truth mask (top-right). This result substantiates our quantitative findings: the adversarial noise forces the discriminator to learn tighter decision boundaries, resulting in a significant boost in P-AUROC (+6.64%) by minimizing false positives at the pixel level.

References

Goodfellow, I. J.; Shlens, J.; and Szegedy, C. 2015. Explaining and Harnessing Adversarial Examples. In *International Conference on Learning Representations*.

Liu, Z.; Zhou, Y.; Xu, Y.; and Wang, Z. 2023. SimpleNet: A Simple Network for Image Anomaly Detection and Localization. *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 20402–20411.

Madry, A.; Makelov, A.; Schmidt, L.; Tsipras, D.; and Vladu, A. 2018. Towards Deep Learning Models Resistant to Adversarial Attacks. In *International Conference on Learning Representations*.

Miyato, T.; Maeda, S.-i.; Koyama, M.; and Ishii, S. 2018. Virtual Adversarial Training: A Regularization Method for Supervised and Semi-Supervised Learning. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 41(8): 1979–1993.

Urban Construction Zone Generator

Autonomous Vehicle Testing Simulation Module

> **Research Project** | University of Michigan - Mcity
> **Role**: Software Developer & Algorithm Designer
> **Tech Stack**: Python, SUMO, TraCI, TomTom API

GitHub Repository: github.com/dmmsjtu-umich/Terasim_Urban_Construction_Zone

Problem Statement

Construction zones represent one of the most challenging scenarios for autonomous vehicles:

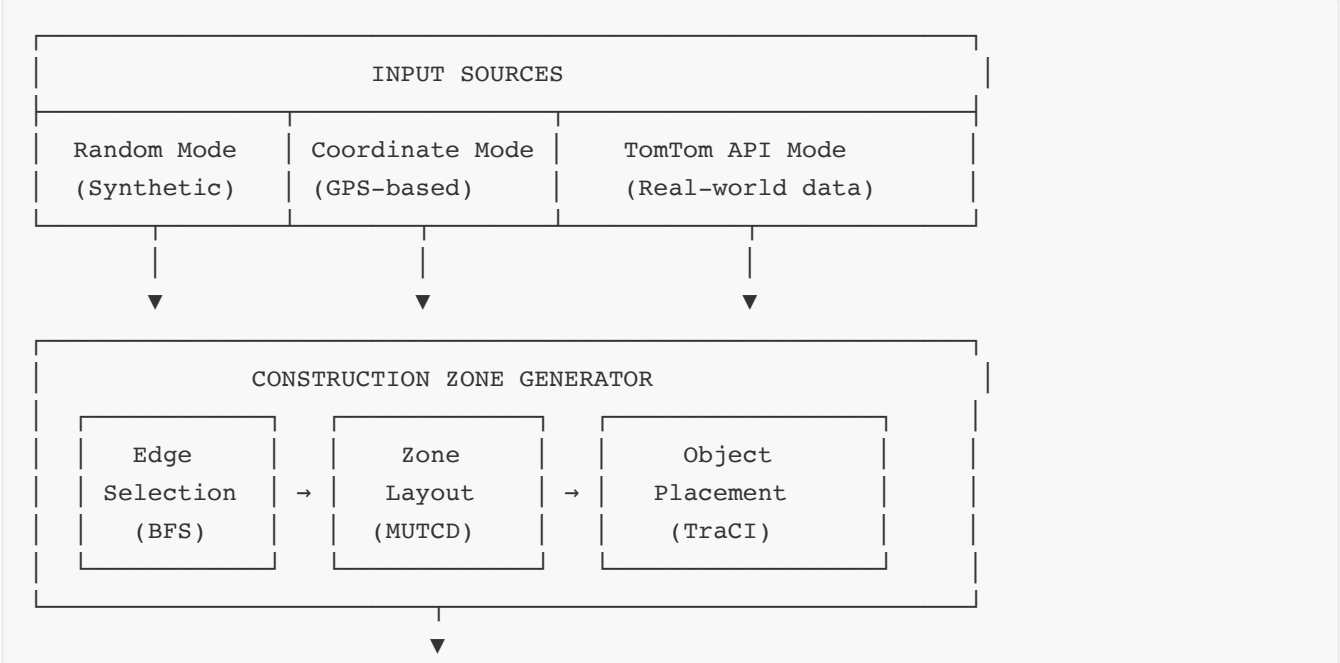
- **23% of highway fatalities** occur in work zones (NHTSA, 2023)
- Non-standard signage and unexpected lane closures
- Complex merging behaviors with human drivers
- Limited real-world testing data available

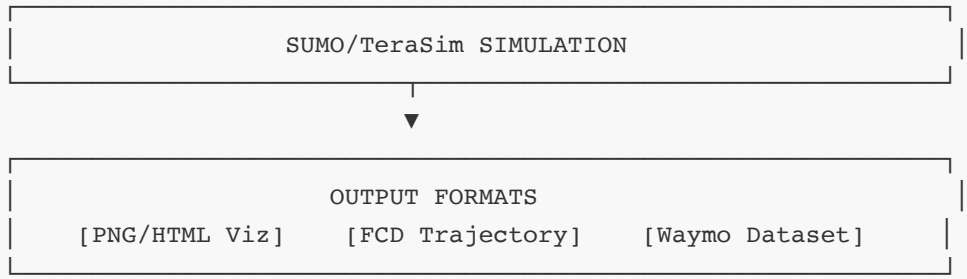
Challenge: How can we generate diverse, realistic construction zone scenarios at scale to improve AV safety testing?

My Solution

I designed and implemented a **scalable simulation pipeline** that automatically generates MUTCD-compliant construction zones for autonomous vehicle testing.

System Architecture

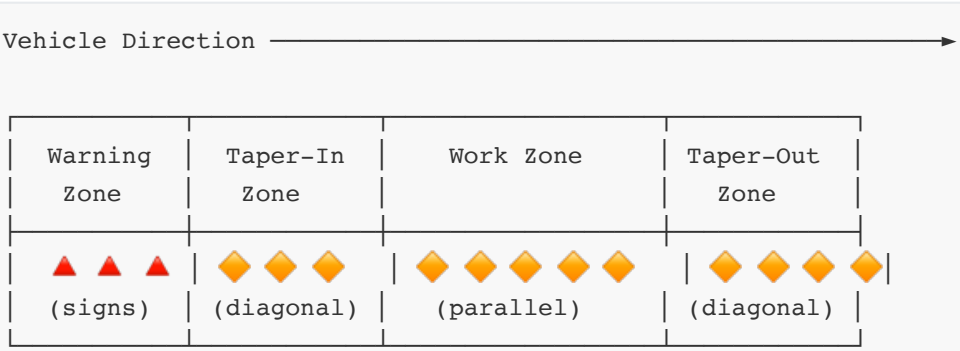




Technical Implementation

1. MUTCD-Compliant Zone Layout

I implemented U.S. federal highway standards for work zone design:



Zone Length Formulas:

$L = W \times S^2 / 60$ (S ≤ 40 mph)

$L = W \times S$ (S ≥ 45 mph)

Where: W = lane width, S = speed limit

2. Multi-Source Data Integration

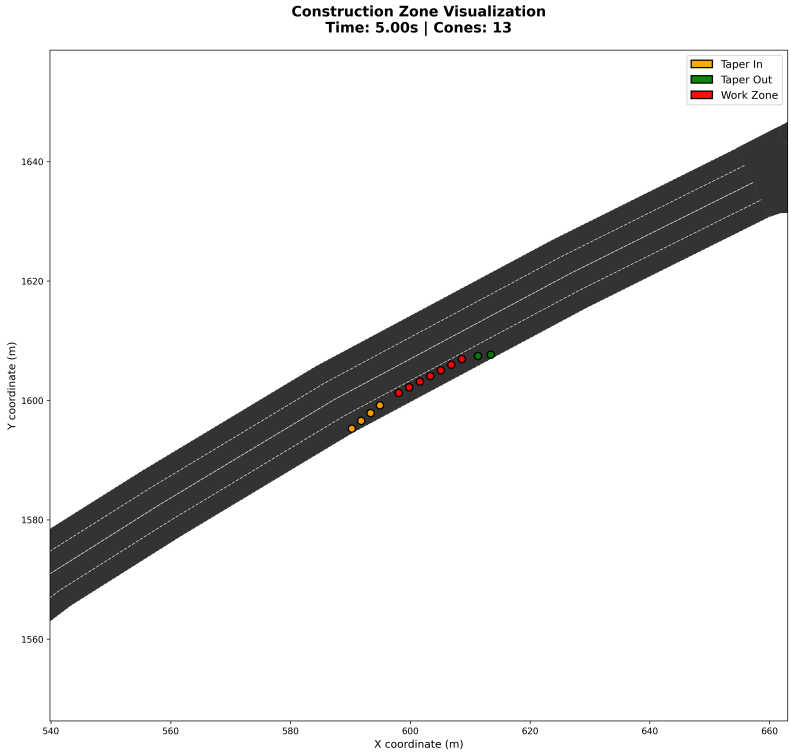
Generation Mode	Data Source	Use Case
Random	SUMO road network	Batch synthetic testing
Coordinate	User GPS input	Specific location testing
TomTom API	Real traffic data	Real-world scenario replication

3. Key Algorithms

Component	Algorithm	Complexity
Edge Selection	BFS Graph Search	O(V + E)
Coordinate Matching	KD-Tree + Projection	O(log n)
Zone Calculation	MUTCD Formulas	O(1)
Object Placement	TraCI API calls	O(n)

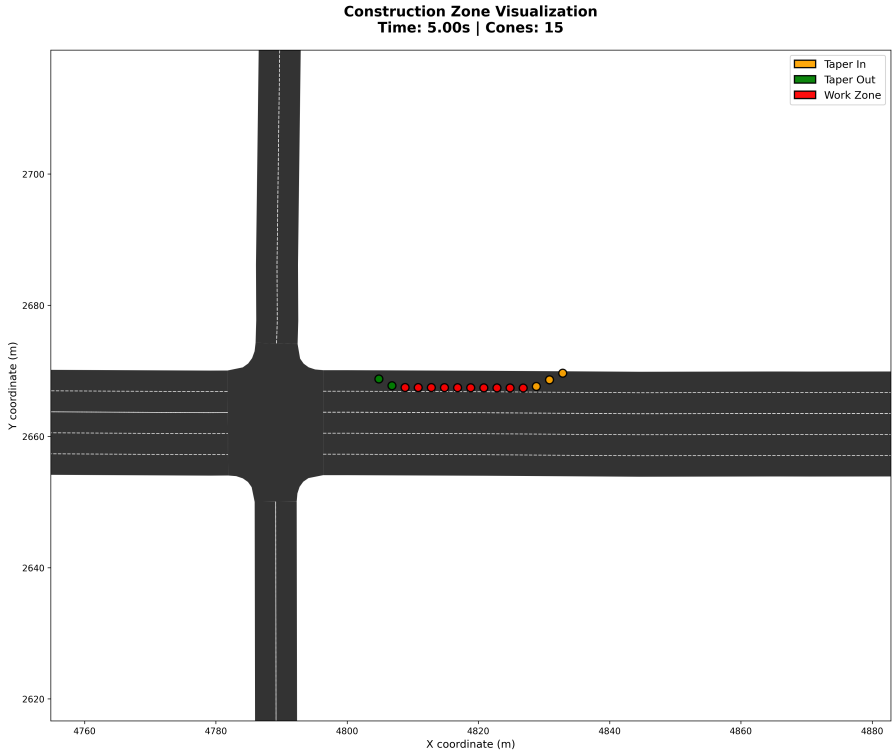
Visualization Results

Construction Zone Generation



Generated construction zone showing taper-in (orange), work zone (red), and taper-out (green) regions with precise cone placement.

Real-World Data Integration



Construction zone generated from live TomTom Traffic API data - Denver, CO highway work zone.

Interactive HTML Map

The system generates zoomable satellite maps for detailed inspection:

- Google satellite tiles (zoom level 21)
- Click-to-inspect cone details
- Multi-layer map options

Quantitative Results

Metric	Achievement
Generation Speed	~3 scenarios/minute
Zone Length Range	30m - 10km
Coordinate Accuracy	±2m (GPS to lane)
API Coverage	27+ major US cities
Output Formats	4 (PNG, HTML, Waymo, FCD)

Skills Demonstrated

Problem Solving

- Identified gap in AV testing infrastructure for work zone scenarios
- Designed modular architecture supporting multiple data sources
- Implemented federal highway standards in simulation environment

Visualization

- Multi-format output (static PNG, interactive HTML, satellite maps)
- Color-coded zone visualization for quick comprehension
- Segmented views for long construction corridors

Software Engineering

- Clean, documented Python codebase (~3000 lines)
- API integration (TomTom Traffic, OpenStreetMap)
- Export to industry-standard formats (Waymo Dataset)

Domain Knowledge

- MUTCD highway safety standards
- Traffic simulation (SUMO/TraCI)
- Autonomous vehicle testing methodologies

What I Learned

1. **Systems Thinking**: Designing modular pipelines that connect data sources → processing → simulation → visualization
2. **Standards Compliance**: Translating regulatory documents (MUTCD) into working algorithms
3. **API Integration**: Working with external data sources to enhance simulation realism
4. **Research Context**: Contributing to open-source AV safety research at a major university lab

---

This project was developed as part of my research work at University of Michigan Mcity, contributing to the TeraSim autonomous vehicle testing platform.