Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed
Computing

Using Physics-Informed Neural Networks to Predict Railway Irregularities

Master's Thesis

Dingran Feng

`difeng@ethz.ch`

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Andreas Plesner
Prof. Dr. Roger Wattenhofer

September 24, 2024

Acknowledgements

I would like to express my deepest gratitude to my thesis supervisors, Andreas Plesner and Professor Roger Wattenhofer, for their guidance and support throughout my master's thesis. I am also thankful to the Distributed Computing Group (DISCO) and the Computer Engineering and Networks Laboratory (TIK) at ETH Zurich, for providing high-performance computing resources such as GPUs and CPUs.

Many thanks and love to my family and friends, for their encouragement and love all the time. Additionally, I extend my appreciation to the city of Zurich, a beautiful place where I spent my past two years studying, working, and enjoying life.

Thank you very much for your significant contributions to this work.

Abstract

In this thesis, we studied the application of Physics-Informed Neural Networks (PINNs) on the rail geometry's prediction. Using the simulated data of train vehicles as well as the prior knowledge about its dynamic system, we can effectively detect the irregularities based on deep learning models. We first constructed a physical model of the train vehicle by ordinary differential equations (ODEs), focusing on wheel-rail interactions for external forces and the suspension systems for internal forces.

We used a mass-spring-damper model as an analogue of the suspension system, to compare the performance of PINNs against conventional neural networks. By penalizing the deviations of predicted accelerations from the physical system, PINNs shows higher accuracy and efficiency compared with traditional Convolutional Neural Networks (CNNs). Our results also demonstrated their better performance to train effectively under limited data volume.

Additionally, we developed a specialized PINN training algorithm tailored for the train vehicle system, with the design of mixed loss function and loss type switching mechanism. Implementing a 3-layer PINN on the vehicle sensor data generated by numerical simulation, we successfully reduced the prediction error of a traditional CNN by 14.9%, attaining a mean absolute error (MAE) of 0.377 mm for irregularity prediction. We also tuned the hyperparameters, including physics weights, loss thresholds, and network architecture, to optimize the PINN model's performance.

Contents

Chapter 1

Introduction

1.1 Background and Motivation

Rail irregularities, which refer to deviations from the nominal or designed positions of rails in both lateral and vertical directions, are critical concerns, particularly for high-speed railways (HSRs). Accurate detection of these irregularities is crucial to repair rail deformations that exceed tolerances, ensuring both the safety of train operations and the comfort of passengers at high speeds driving. Traditional anomaly detection methods rely on either portable specific monitoring devices or comprehensive inspection trains (CITs), which are not only costly in terms of time and money, but could also take up operational rails or postpone regular train services due to different speed requirements. Additionally, given the extensive mileage of HSRs, the limited number of measurement devices and vehicles makes it challenging to maintain frequent detection and continuous conservation of the rails.

A practical and effective solution to these challenges is to equip the in-service trains with low-cost sensors, such as inertial measurement units (IMUs) or optical monitors. These sensors can quickly and continuously gather data from railway tracks in an economical manner, an approach often known as performance-based track geometry (PBTG). Furthermore, enhanced with odometers and GPS systems, trains can also report precise locations of significant displacements directly to the control center in real time.

Given the volume of data generated daily by in-service trains, employing data-driven methods like deep learning to process onboard monitoring data presents a viable option. Nowadays we have seen the great power deep neural networks show in processing large scale datasets. Some research also has begun to explore the application of machine learning in assessing track quality and rail roughness profiles [1][2][3]. Nevertheless, there has been limited research on integrating prior dynamic knowledge with deep learning models to enhance the accuracy of track geometry predictions and to reduce the data demand for model training. This gap represents the primary motivation for our research project.

1.2 Literature Review

In the master thesis of Christiansen [4], a Ordinary Differential Equation (ODE) system of a railway vehicle is developed to investigate the vehicle's dynamic behavior. Considering the complexity of this system, the compuatiion was implemented with the help of some numerical method, such as SDIRK integrator [5] and RSGEO contact point calculator [6]. His work also placed significant emphasis on the calculation of wheel penetration and the forces generated at the wheel-rail interface, which gives us much inspiration on integrating the accelerations derived from forces into our PINN model. We plan to modify his programs for generating dynamics simulation dataset, for training and testing neural networks in the following work. Additionally, other research has leveraged tools like SIMPACK and VAMPIRE for numerical simulation.

There are some other studies that apply deep learning models in rail geometry prediction. Plesner et al. [7] employed convolutional neural networks (CNNs) and conformal prediction to predict track irregularities with uncertainty intervals from high-fidelity sensor data of an ETR500 passenger train. In Plesner's master thesis [8], he also compares the CNN model with classic machine learning models, such as decision tree, bootstrapping, and random forest, highlighting the superior predictive capabilities and continuous monitoring potential of CNNs. He also mentioned the mean unsigned error for railway industry requirements at 0.1 mm, and the error for state-of-the-art (SOTA) at 0.33 mm, which can be the benchmarks for our model assessment.

Wang et al.[3] constructed a Branch Fourier Neural Operator (BFNO) model to estimate the behavior of vehicle-track coupled system, which can successfully handle the prediction tasks in a wide range of detection resolutions. They also demonstrated the efficiency of the model by comparison with the conventional CNN-GRU model. Their work gives us an inspiration that we can consider the irregularity prediction question in a functional mapping form in the time domain, using neural operator methods like Deep Operator Networks (DeepONet) [9], Fourier Neural Operators[10], Convolutional Neural Operators[11], and Koopman Neural Operators[12], etc.

Many studies have been done in applying Physics-Informed Neural Networks (PINNs) on differential equations related fields, including system simulation, data-driven equation exploration, and inverse problem solution [13] [14]. As a new form of meshless method, PINNs can be easily extended to different resolutions and irregular geometries without retraining the model [15], which shows its significant characteristics in representation learning and transfer learning. PINN models have shown their efficacy in solving inverse problems [16], even with multi-fidelity and stochastic datasets.

Extended Physics-Informed Neural Networks (XPINNs) have facilitated the decomposition of the time-space domain in arbitrary forms and resolutions [17], thereby enhancing the utilization of GPUs' parallel computing capabilities and improving the training efficiency. This represents a significant advancement over Conservative PINNs (cPINNs) and worth further application or exploration in our future work. However, it is important to note that all these PINN models have limitations, particularly when addressing certain problems, such as abrupt changes and chaotic systems, as well as constraints in model optimization [18] [19]. Considering that the geometry of the railway is a continuous quantity and has a standardized description in dynamics, PINNs are applicable to our system.

Chapter 2

Dynamic Model of Railway Vehicles

2.1 Coordinate System and Vehicle Structure

To clearly describe the motion of each component, we first set up the coordinate system as follows: the x-axis (longitudinal direction) aligns with the direction of train's driving; the y-axis (lateral direction) is perpendicular to the driving direction and parallel to the sleepers; the z-axis (vertical direction) is perpendicular to the field plane and pointing to the sky. Within this coordinate system, each rigid body has defined rotational angles to measure its orientation relative to its center of mass: roll angle ϕ (rotation around the x-axis), yaw angle ψ (rotation around the z-axis), and pitch angle θ (rotation around the y-axis).

We adopt the structure and parameters from Cooperrider's bogie [20] to model our dynamic system, with linearity in Hooke's Law and Damping Law. Our research concentrates on the front part (leading section) of a train vehicle, which includes four primary components: the leading front wheelset, leading rear wheelset, leading bogie frame, and car body. Both the front and rear wheelsets are connected to the bogie frame via the primary suspension system (6 springs for each side), while the bogie frame and the car body are interconnected by the secondary suspension system (7 springs and 7 dampers). Though each component theoretically possesses six degrees of freedom (DOF), denoted as $[x, y, z, \phi, \psi, \theta]$, our dynamic model in practice only considers specific DOFs: $[y, z, \phi, \psi]$ for both front and rear wheelsets, $[y, z, \phi, \psi, \theta]$ for the bogie frame, and $[\phi]$ for the car body. The model also incorporates the first and second derivatives of these variables, capturing corresponding translational or angular velocities and accelerations of these DOFs. The coordinate system and the vehicle's structure are depicted in Figure 2.1 in detail.

The vehicle operates on a track where each wheelset contacts the left and right rails. The motion of the vehicle is significantly influenced by its current dynamic state as well as the lateral and vertical irregularities of these rails. Here, lateral irregularity refers to deviations from the nominal y-direction position of

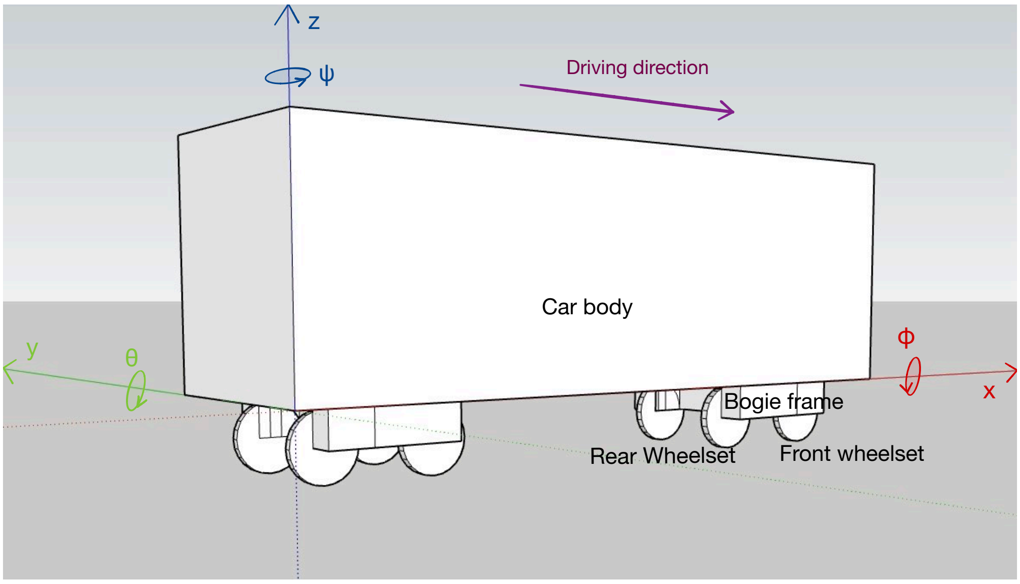

Figure 2.1: Schematic diagram of the coordinate system and vehicle structure.

the rail, and vertical irregularity refers to deviations from the nominal z-direction position of the rail.

2.2 Mass-Spring-Damper Model

The contact points serve as the sole source of external forces (normal forces and creepage forces), while the suspension systems exclusively provide internal forces (spring forces and damper forces). Therefore, investigating how the parameters of springs and dampers in the suspension systems affect the kinetic characteristics of the system's components is crucial. To this end, we analyze a simple damped harmonic oscillator consisting of a single spring and a single damper connecting a ball to a rail. This basic system is commonly referred to as the mass-spring-damper (MSD) model.

The MSD model is presumed to be positioned on a completely smooth ground i.e. not considering frictions. The small ball is enclosed within a completely smooth pipe, which is always kept perpendicular to the constant driving speed, ν, along the x-axis. Within the pipe, a spring and a damper are installed, each connecting one end to the ball and the other end to the rail via a smooth loop. The schematic diagram of this MSD model is depicted in Figure 2.2. Let m represent the mass of the ball, k the spring constant, c the damping coefficient, and u the position of the ball. In the absence of rail irregularity (i.e., when the rail is perfectly straight), the motion of the ball can be modeled by the following second-

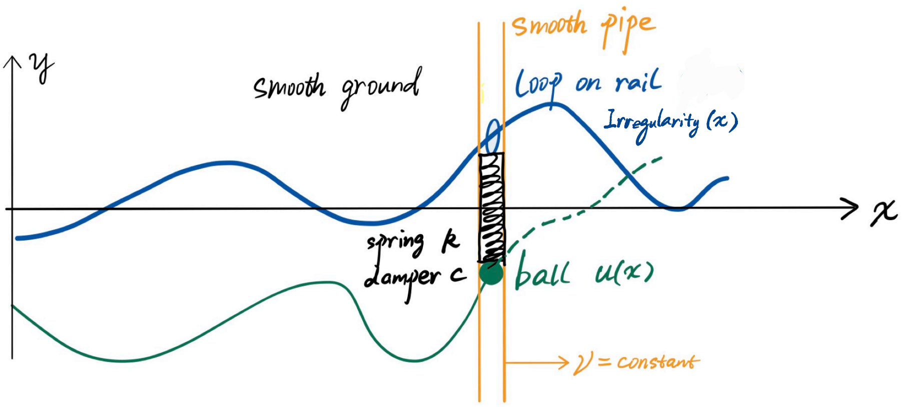

Figure 2.2: Schematic diagram of the mass-spring-damper model. The blue curve represents the geometry of rail (lateral irregularity), and the green curve represents the motion trajectory of green ball.

order Ordinary Differential Equation (ODE) derived from Newton's Second Law:

$$m\frac{d^2u}{dt^2} + c\frac{du}{dt} + ku = 0 \tag{2.1}$$

After introducing the driving speed ν, and the rail's lateral irregularity $\phi(x)$, the original ODE is adjusted to:

$$m\frac{d^2[u(\nu t) - \phi(\nu t)]}{dt^2} + c\frac{d[u(\nu t) - \phi(\nu t)]}{dt} + k[u(\nu t) - \phi(\nu t)] = 0 \tag{2.2}$$

The dynamic characteristics of the system are significantly influenced by the relative magnitudes of the spring and damper coefficients. This relationship is quantified by the damping ratio, which has three ranges of damping conditions (underdamping, critical damping, and overdamping):

$$\zeta = \frac{c}{c_c} = \frac{c}{2\sqrt{mk}} \begin{cases} < 1 & \text{Underdamping} \\ = 1 & \text{Critical damping} \\ > 1 & \text{Overdamping} \end{cases} \tag{2.3}$$

where $c_c = 2\sqrt{mk}$ denotes the critical damping coefficient. With the help of ODE solver and linear interpolation, we conducted numerical simulations of the ball's trajectory utilizing the real data of left rail's lateral irregularity. The outcomes across three different damping conditions are outlined as follows: Figure 2.3 illustrates the underdamping condition with a ratio of 0.0025, where we can observe significant oscillations in the ball's motion; Figure 2.4 illustrates the critical damping condition with a ratio of 1, where ball's trajectory closely aligns with the rail geometry, as the system quickly returns to its equilibrium state in a

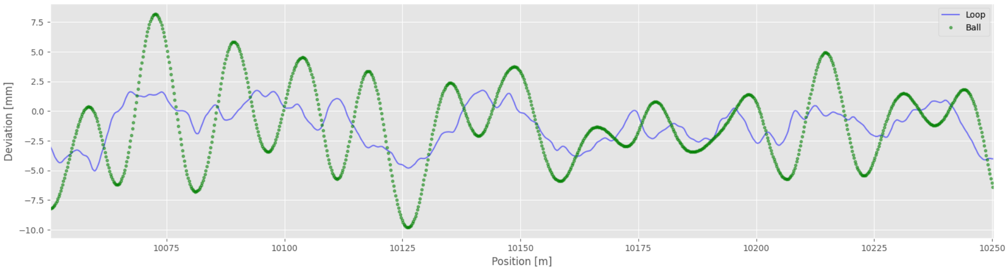

Figure 2.3: Underdamping $\zeta = 0.025$. Blue curve: rail's lateral irregularity. Green curve: ball's movement trajectory.

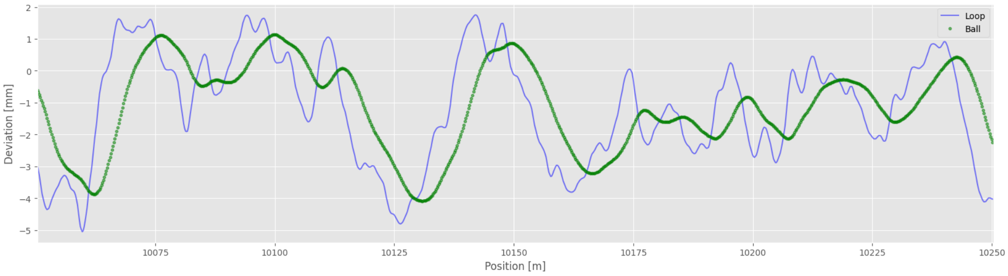

Figure 2.4: Critical damping $\zeta = 1$. Blue curve: rail's lateral irregularity. Green curve: ball's movement trajectory.

minimal amount of time.; Figure 2.5 illustrates the overdamping condition with a ratio of 5, where the ball's trajectory loses much of the geometric information from the rail, because the high damping coefficient significantly slows its return to the equilibrium state.

We apply CNNs and PINNs on this MSD model to verify their prediction performance in predicting rail irregularities in section 4.2.

2.3 Wheel-Rail Interaction on the Contact point

The most challenging aspect of calculating the normal and friction forces at the contact point is the nonlinear relationship among the deviation position, rotation angle, and penetration depth, which arises directly from the irregular shape and material properties of both the rails and wheels. Calculating the rail-wheel interaction at the contact patches is crucial for updating the external forces.

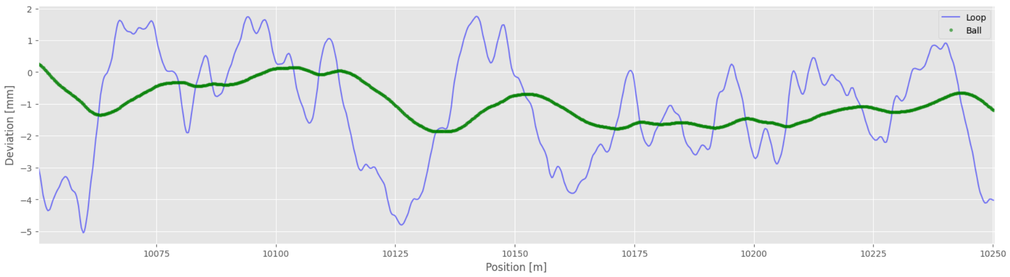

Figure 2.5: Overdamping $\zeta = 5$. Blue curve: rail's lateral irregularity. Green curve: ball's movement trajectory.

2.3.1 Rail Profile

Our model utilizes the UIC60 rail profile, which consists of arcs with several radii (300 mm, 80 mm, and 13 mm). The cross-sectional view of the rail surface and its corresponding curvature are shown in Figure 2.6. Each half of the profile is made up of three circular arcs, with the outermost section forming a steep inclined plane with slant angle 87.14°. Additionally, we incorporate actual lateral and vertical irregularities data to represent the 3D shape of the rail, as illustrated in Figure 2.7.

The roll angle of the wheelset influences the distance between the contact points on the left and right rails, as shown in Figure 2.8. As the roll angle increases, the distance decreases from the nominal value of 1500 mm, stabilizing around 1464 mm when the roll angle exceeds 0.5°. This underscores a factor contributing to the contact area's nonlinearity: even a slight deviation from the wheelset's zero-roll state significantly changes the positions of contact points.

2.3.2 Wheel Profile

In this project, we use the DSB97-1 profile as the surface shape of the four wheels we are considering. The tread and flange (blue line in Figure 2.9) of a railway wheel form the area where it contacts the track. When the wheelset's roll angle is small, the contact point lies on the tread; as the roll angle increases, the contact point shifts to the flange. The flange plays a critical role in maintaining the balance of the wheelset and preventing derailment, since it provide sufficient centripetal force during train turning (track cant $\alpha \neq 0$ and turning radius $R \neq \infty$). It is also the primary cause of the nonlinearity between the contact point position, wheelset roll angle, and penetration depth — which makes analytical solutions for these relationships nearly impossible. The 3D surface profile of the wheel is displayed in Figure 2.10. The positions of the contact points on the wheel have a nonlinear relationship with the wheelset's roll angle (Fig 2.11).

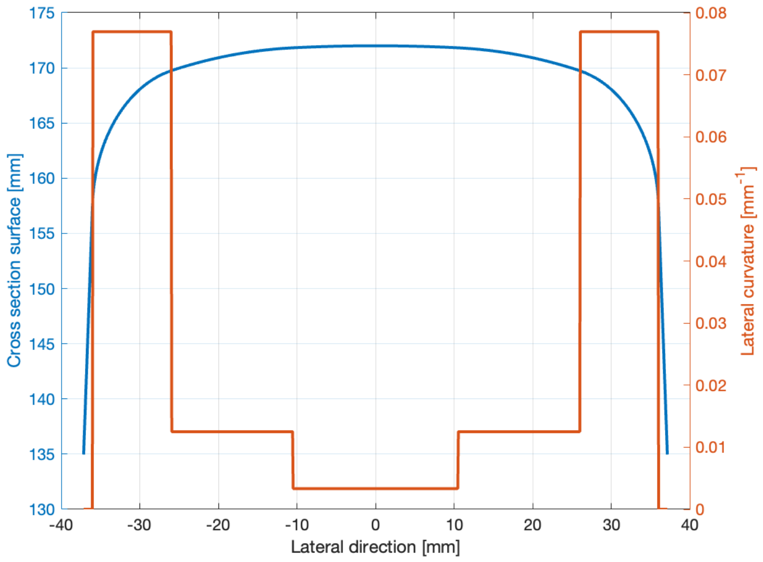

Figure 2.6: Cross-sectional view of the rail surface. Blue curve: rail shape. Red curve: curvature.

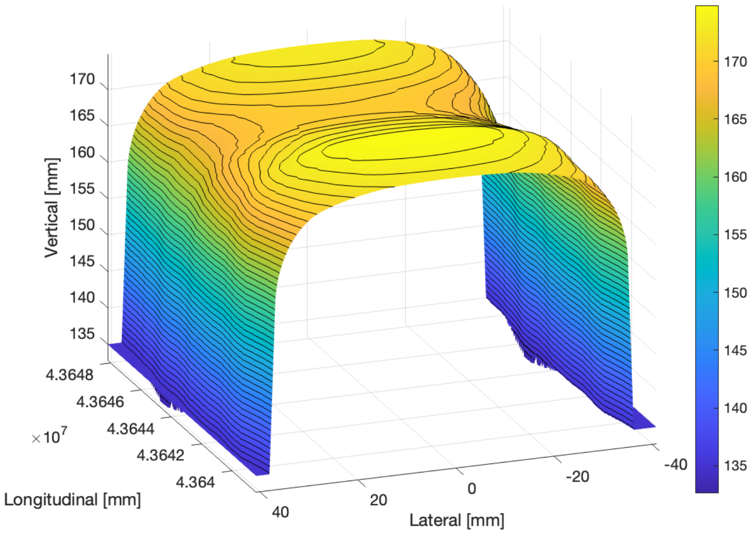

Figure 2.7: 3D representation of the rail surface, including lateral and vertical irregularities. The contour lines and color bar indicate the vertical height.

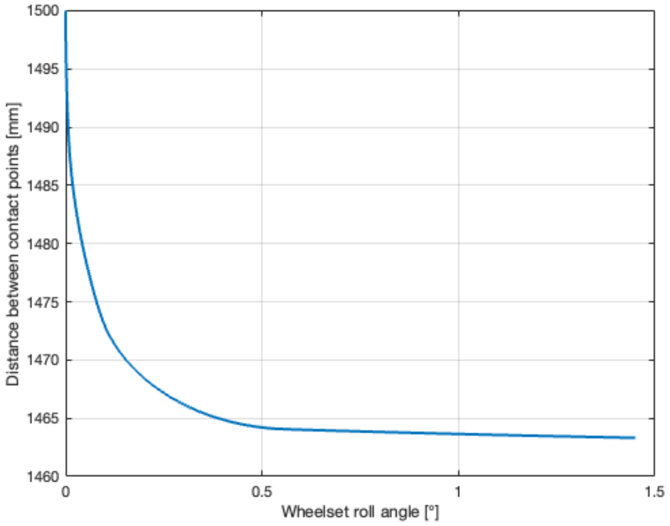

Figure 2.8: Contact point distance under different wheelset roll angle.

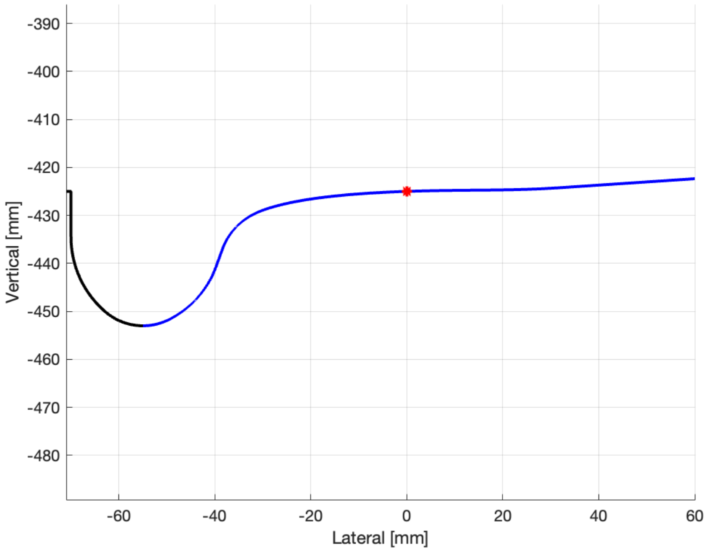

Figure 2.9: Profile of the right wheel. The blue area represents the effective contact region, while the red point indicates the nominal wheel-rail contact point in a static state and zero irregularities.

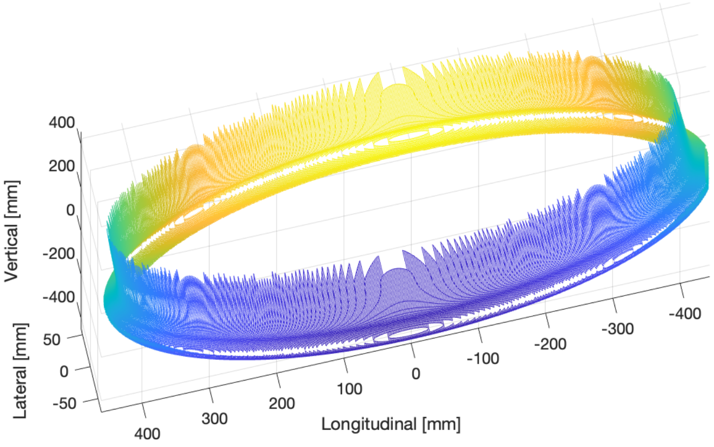

Figure 2.10: The 3D shape of wheel surface.

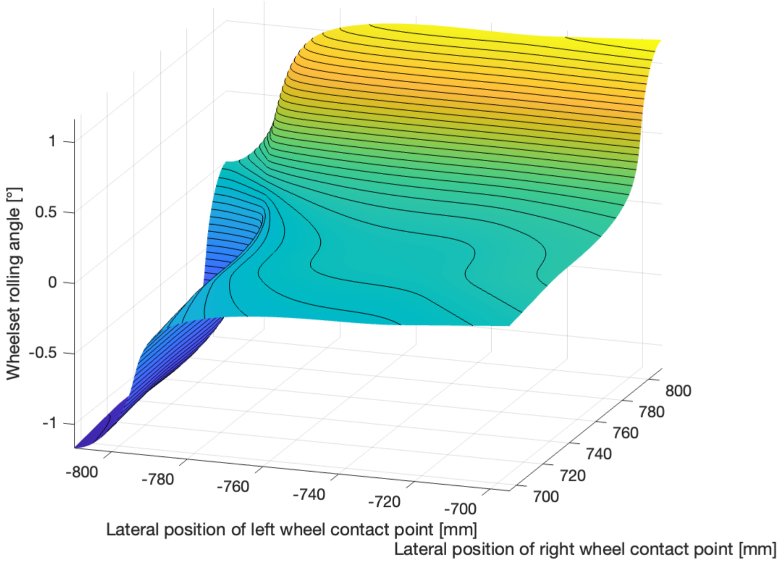

Figure 2.11: Wheelset roll angle at different contact point positions on the left and right wheels.

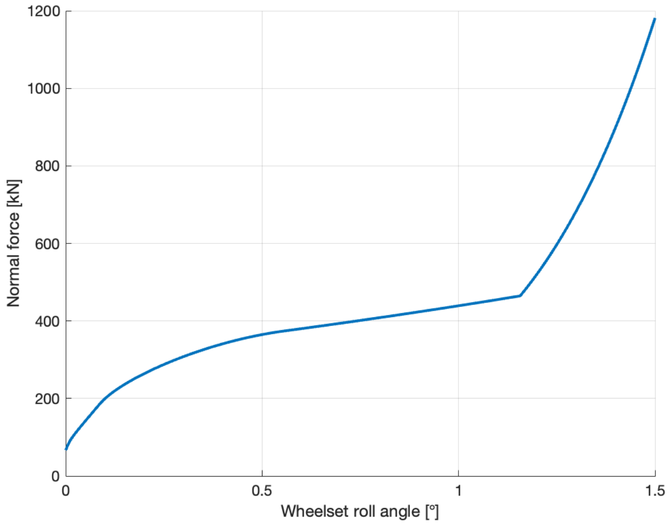

Figure 2.12: The relationship between normal force and wheelset roll angle.

2.3.3 Penetration Depths and Contact Point Forces

Based on the previous work by Christiansen on contact points [4], the normal force is proportional to the penetration depth raised to the power of 1.5:

$$N \propto p^{\frac{3}{2}} \tag{2.4}$$

Using the initial penetration depth p_0 and normal force N_0 values calculated from RSGEO [6], we can express the normal force as:

$$N = N_0(1 + \frac{\Delta p}{p_0})^{\frac{3}{2}} \tag{2.5}$$

By integrating the previously mentioned wheel and rail profiles, the normal force can be calculated for different wheelset roll angles with some simplifications on the wheelset yawing state. We can notice that there is a mutation point around 1.15°.

2.4 Differential Equations of Vehicle Dynamics

The wheelset experiences various forces, including normal force, creepage force, centrifugal force, and gravitational force. Normal force N_{ijk} and creepage force F_{ijk} result from penetration at the rail-wheel contact point. Here, subscript $i \in \{l, r\}$ specifies the left or right rail, subscript $j \in \{x, y, z\}$ specifies the direction along the three axes, and subscript $k \in \{1, 2\}$ specifies the front or rear wheelset. The bogie frame is subjected to centrifugal force and gravitational force, alongside forces from the springs and dampers, because of its links to both the primary and secondary suspension systems.

Table 2.1: Definition of the variables in ODEs

Variable	Description
q_1	Lateral position of front wheelset
q_2	Lateral linear velocity of front wheelset
q_3	Yaw angle of front wheelset
q_4	Yaw angular velocity of front wheelset
q_5	Lateral position of rear wheelset
q_6	Lateral linear velocity of rear wheelset
q_7	Yaw angle of rear wheelset
q_8	Yaw angular velocity of rear wheelset
q_9	Lateral position of bogie frame
q_{10}	Lateral linear velocity of bogie frame
q_{11}	Yaw angle of bogie frame
q_{12}	Yaw angular velocity of bogie frame
q_{13}	Roll angle of bogie frame
q_{14}	Roll angular velocity of bogie frame
q_{15}	Roll angle of car body
q_{16}	Roll angular velocity of car body
q_{17}	Vertical position of front wheelset
q_{18}	Vertical linear velocity of front wheelset
q_{19}	Vertical position of rear wheelset
q_{20}	Vertical linear velocity of rear wheelset
q_{21}	Roll angle of front wheelset
q_{22}	Roll angular velocity of front wheelset
q_{23}	Roll angle of rear wheelset
q_{24}	Roll angular velocity of rear wheelset
q_{25}	Vertical position of bogie frame
q_{26}	Vertical linear velocity of bogie frame
q_{27}	Pitch angle of bogie frame
q_{28}	Pitch angular velocity of bogie frame
q_{29}	Rolling constraint of front wheelset
q_{30}	Rolling constraint of rear wheelset
q_{31}	Driving distance of the vehicle

Utilizing the Newton-Euler equations, we can characterize the translational motion of a rigid body using Newton's second law and its rotational motion using Euler's formula. Consequently, the dynamic system of the vehicle is depicted by a set of second-order differential equations, which describe the movement of each component and the modifications of two rolling constraints. To streamline the calculations and decrease the model's complexity, we have also incorporated the velocities and angular velocities of each component, approximately doubling the number of variables (Table 2.1). Based on previous work [4] [8], the ODEs for these state variables are presented below, where we convert the 2nd-order ODEs into 1st-order ODEs:

$$\frac{\mathrm{d}}{\mathrm{d}t}q_i = q_{i+1}, \quad i \in \{1, 3, \dots, 27\} \tag{2.6}$$

$$\begin{aligned} m_w \frac{\mathrm{d}}{\mathrm{d}t}q_2 = {} & F_{ly1} + F_{ry1} + N_{ly1} + N_{ry1} \\ & - 2k_1(q_1 - q_9 - bq_{11} - h_1 q_{13}) - m_x g\theta + m_w \frac{V^2}{R} \end{aligned} \tag{2.7}$$

$$\begin{aligned} I_{wy} \frac{\mathrm{d}}{\mathrm{d}t}q_4 = {} & a_{r1}[F_{rx1} + (F_{ry1} + N_{ry1})q_3] - a_{l1}[F_{lx1} + (F_{ly1} + N_{ly1})q_3] \\ & - 2k_2 d_1^2 (q_3 - q_{11}) \end{aligned} \tag{2.8}$$

$$\begin{aligned} m_w \frac{\mathrm{d}}{\mathrm{d}t}q_6 = {} & F_{ly2} + F_{ry2} + N_{ly2} + N_{ry2} \\ & - 2k_1(q_5 - q_9 - bq_{11} - h_1 q_{13}) - m_x g\theta + m_w \frac{V^2}{R} \end{aligned} \tag{2.9}$$

$$\begin{aligned} I_{wx} \frac{\mathrm{d}}{\mathrm{d}t}q_8 = {} & a_{r2}[F_{rx2} + (F_{ry2} + N_{ry2})q_7] - a_{l2}[F_{lx2} + (F_{ly2} + N_{ly2})q_7] \\ & - 2k_2 d_1^2 (q_7 - q_{11}) \end{aligned} \tag{2.10}$$

$$\begin{aligned} m_b \frac{\mathrm{d}}{\mathrm{d}t}q_{10} = {} & 2k_1(q_1 + q5 - 2q_9 - 2h_1 q_{13}) + 2k_4(h_2 q_{13} + h_3 q_{15} - q_9) \\ & + 2D_2(h_2 q_{14} + h_3 q_{16} - q_{10}) - (\frac{1}{2}m_{cb} + m_b)g\theta + m_b \frac{V^2}{R} \end{aligned} \tag{2.11}$$

$$\begin{aligned} I_{bz} \frac{\mathrm{d}}{\mathrm{d}t}q_{12} = {} & 2d_1^2 k_2(q_3 + q_7 - 2q_{11}) - k_6 q_{11} - D_6 q_{12} \\ & + 2bk_1(q_1 - q_5 - 2bq_{11}) \end{aligned} \tag{2.12}$$

$$\begin{aligned} I_{bx} \frac{\mathrm{d}}{\mathrm{d}t}q_{14} = {} & 2k_1 h_1(q_1 + q_5 - 2q_9 - 2h_1 q_{13}) + 2k_4 h_2(q_9 - h_2 q_{13} - h_3 q_{15}) \\ & + 2D_2 h_2(q_{10} - h_2 q_{14} - h_3 q_{16}) - 2d_2^2[k_5(q_{13} - q_{15}) \\ & + D_1(q_{14} - q_{16})] - 2d_1^2 k_3(2q_{13} - q_{21} - q_{23}) \end{aligned} \tag{2.13}$$

$$I_{cx}\frac{\mathrm{d}}{\mathrm{d}t}q_{16} = -2d_2^2[k_5(q_{15}-q_{13}) + D_1(q_{16}-q_{14})] \tag{2.14}$$

$$\begin{aligned} m_w\frac{\mathrm{d}}{\mathrm{d}t}q_{18} &= F_{lz1} + F_{rz1} + N_{lz1} + N_{rz1} \\ &\quad + 2k_3(q_{25}-q_{17}) - m_x g - m_w\theta\frac{V^2}{R} \end{aligned} \tag{2.15}$$

$$\begin{aligned} m_w\frac{\mathrm{d}}{\mathrm{d}t}q_{20} &= F_{lz2} + F_{rz2} + N_{lz2} + N_{rz2} \\ &\quad + 2k_3(q_{25}-q_{19}) - m_x g - m_w\theta\frac{V^2}{R} \end{aligned} \tag{2.16}$$

$$\begin{aligned} I_{wx}\frac{\mathrm{d}}{\mathrm{d}t}q_{22} &= -a_{r1}[F_{rz1} + N_{rz1} - (F_{ry1} + N_{ry1})q_{21}] \\ &\quad + a_{l1}[F_{lz1} + N_{lz1} - (F_{ly1} + N_{ly1})q_{21}] - 2k_3 d_1^2(q_{21}-q_{13}) \end{aligned} \tag{2.17}$$

$$\begin{aligned} I_{wx}\frac{\mathrm{d}}{\mathrm{d}t}q_{24} &= -a_{r2}[F_{rz2} + N_{rz2} - (F_{ry2} + N_{ry2})q_{23}] \\ &\quad + a_{l2}[F_{lz2} + N_{lz2} - (F_{ly2} + N_{ly2})q_{23}] - 2k_3 d_1^2(q_{23}-q_{13}) \end{aligned} \tag{2.18}$$

$$m_b\frac{\mathrm{d}}{\mathrm{d}t}q_{26} = -2k_3(2q_{25}-q_{17}-q_{19}) - 2k_5 q_{25} - 2D_1 q_{26} - m_b\theta\frac{V^2}{R} \tag{2.19}$$

$$I_{by}\frac{\mathrm{d}}{\mathrm{d}t}q_{28} = -2bk_3(2bq_{27} + q_{17} - q_{19}) \tag{2.20}$$

$$\begin{aligned} I_{wy}\frac{\mathrm{d}}{\mathrm{d}t}q_{29} &= -r_{r1}(F_{rx1} + F_{ry1}q_3 + N_{ry1}q_3) - r_{l1}(F_{lx1} + F_{ly1}q_3 + N_{ly1}q_3) \\ &\quad - 2d_1^2 k_3 q_3 q_{13} \end{aligned} \tag{2.21}$$

$$\begin{aligned} I_{wy}\frac{\mathrm{d}}{\mathrm{d}t}q_{30} &= -r_{r2}(F_{rx2} + F_{ry2}q_7 + N_{ry2}q_7) - r_{l2}(F_{lx2} + F_{ly2}q_7 + N_{ly2}q_7) \\ &\quad - 2d_1^2 k_3 q_7 q_{13} \end{aligned} \tag{2.22}$$

$$\frac{\mathrm{d}}{\mathrm{d}t}q_{31} = V \tag{2.23}$$

where $m_\bullet$ represents the mass, and $I_\bullet$ denotes the moment of inertia. The static load of mass on each wheelset, $m_x = \frac{1}{4}m_c + \frac{1}{2}m_b + m_w$, is calculated by considering the masses of the car body (m_c), bogie frame (m_b), and wheelset (m_w) respectively. The driving velocity of the train, $V = V(t)$, can vary over time in general settings. However, for our model and experiments, we only consider the vehicle's movement at several constant speeds, i.e., $V = \bar{V}$. The radius of the rail curve at present, $R = R(q_{31}) = R(\bar{V}t)$, which is also the turning radius of the train, is assumed to be infinite to simplify our computations, implying that the nominal centerline of the rails is a straight line, and therefore, the corresponding cant angle α is always zero. The parameters a_{ik} and r_{ik} represent the lateral and vertical distances from the wheelset's mass center to the contact points, respectively. The values and meanings of other parameters in this ODE system are detailed in Table B.1.

2.5 Numerical Simulation of Vehicle Dynamics

In this project, we utilize the simulation code authored by Christiansen [4] in C++ to generate a comprehensive dataset for training, validating, and testing neural networks. We have further modified this code in Python to include both a point-to-point and a tensor-to-tensor version, which facilitate the calculation of physics loss in Physics-Informed Neural Networks (PINNs) based on the parallel computing on GPUs.

The structure of the numerical simulation program in C++ is depicted in Figure 2.13. Initially, the program generates 160 km of lateral and vertical irregularities using Vector Auto-Regression (VAR). It then loads this irregularity data along with rail profile data computed by the RSGEO program [6]. Upon initializing the system state, the algorithm proceeds through loops of numerical integration of the state variables until the target driving distance is reached. Each time step within these loops consists of two main tasks: updating penetration depths and solving the ODEs using the Runge-Kutta method (RK56). It is important to note that the adjustment of contact forces (normal forces and creepage forces) through penetration updates is necessary. Because N_{ijk} and F_{ijk} only appear as exogenous variables in the above ODEs, rather than as endogenous state variables.

Let h denote the time step and $\bar{V}$ the constant driving speed. The recording and detection resolution, γ, can be computed as follows:

$$\gamma = h \cdot \bar{V} \tag{2.24}$$

In this thesis, we consider three driving speeds (120 km/h, 160 km/h, and 200 km/h) and assume a constant resolution of 0.16 m for the convenience of comparison across different scenarios. Accordingly, we can calculate their respective time step values, as detailed in Table 2.2.

Table 2.2: Time step values under constant driving speeds.

Driving speed [km/h]	Time step [s]	Resolution [m]
120	0.00480	0.16
160	0.00360	0.16
200	0.00288	0.16

2.5.1 Penetration Depths and Contact Forces Update

The motivation of updating penetration depths is to recalculate normal forces N_{ijk} and creepage forces F_{ijk} for their subsequent application in solving the ODEs at each time step of integration loop. This process begins by determining the

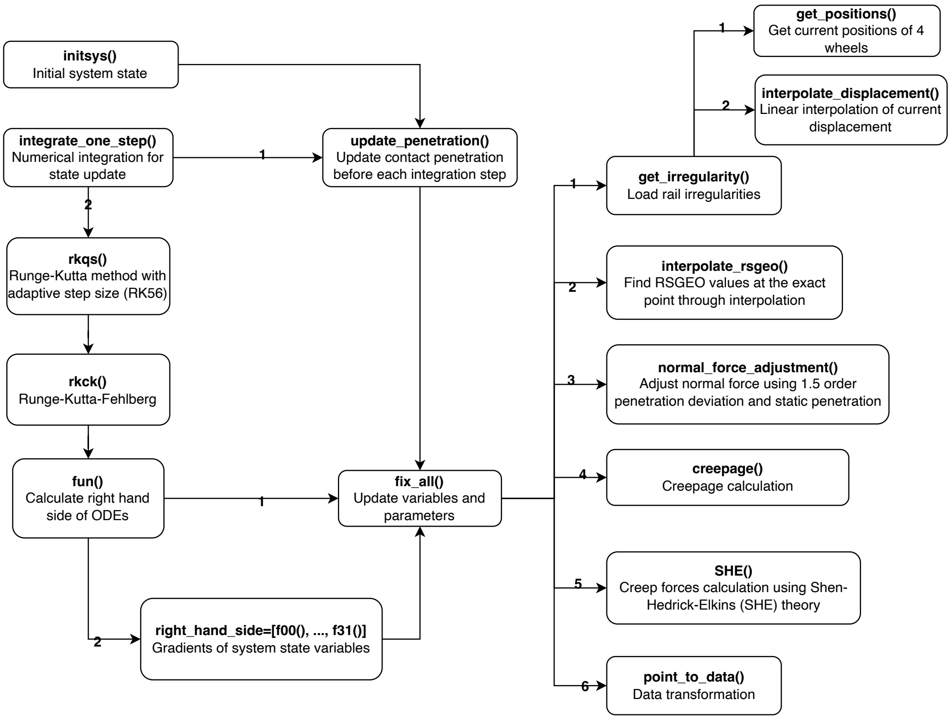

Figure 2.13: Flowchart of the Numerical Simulation Program. Arrows indicate the call relationships between functions, with numbers on the arrows showing the sequence of execution.

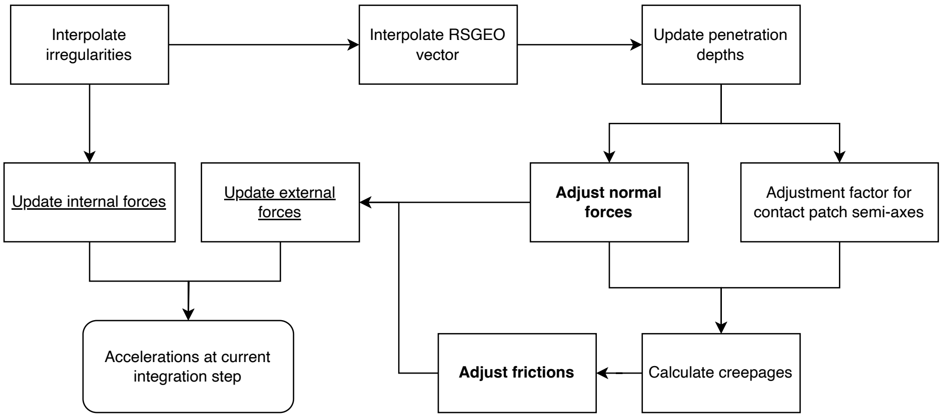

Figure 2.14: Algorithm of forces update and acceleration calculation.

current driven distance and calculating the displacements of the left and right rails using linear interpolation based on the irregularity table. Subsequently, linear interpolation is applied to the 13 contact point parameters listed in the RSGEO table, including the semi-axes of contact patch ellipses, the actual rolling radii of the wheels, and the coordinates of the contact points, etc. Adjustments to the normal forces on both sides are then made using Equation 2.5. The longitudinal, lateral, and yaw-direction creepages are computed based on the Shen-Hedrick-Elkins (SHE) theory [21], and then longitudinal friction f_x and lateral friction f_y are calculated based on updated creepage data.

At this point, all updates to the external forces have been completed. Next, we only need to update the internal forces of the springs and dampers based on the relative positions of the components within the primary and secondary suspension systems.

Figure 2.14 illustrates the force update algorithm in our acceleration approximation program specially designed for the tensor data in physics loss functions.

2.5.2 Explicit Runge-Kutta Method in Each Time Step

The aforementioned ODEs can be expressed again in the form of vectors succinctly as:

$$\frac{\mathrm{d}}{\mathrm{d}t}\vec{q} = \frac{\mathrm{d}}{\mathrm{d}t}(q_1, \cdots, q_{31})^\top = (f_1(\vec{q}, t), \cdots, f_{31}(\vec{q}, t))^\top = \vec{f}(\vec{q}, t) \tag{2.25}$$

Here, $\vec{q}$ represents the system state vector, and $\vec{f}(\bullet)$ denotes the vector of functions on all right-hand-sides of the ODEs. Using the state vector $\vec{q}_n$ obtained from the previous time step, we employ the explicit 5th-order-6th-order Runge-Kutta method (RK56) to compute the state $\vec{q}_{n+1}$ for the current iteration. This method is a numerical integration technique of order $O(h^5)$ with an error estimator of order $O(h^6)$, which adaptively adjusts the integration step size h based on the discrepancy between the 5th-order and 6th-order approximations:

$$\vec{q}_{n+1}^{(5)} = \vec{q}_n + h \sum_{i=1}^{s} b_i^{(5)} \vec{k}_i \tag{2.26}$$

$$\vec{q}_{n+1}^{(6)} = \vec{q}_n + h \sum_{i=1}^{s} b_i^{(6)} \vec{k}_i \tag{2.27}$$

where $\vec{k}_i, \quad i \in \{1, \cdots, s\}$ represents the slope vectors (or called intermediate stages) estimated at various points on the right-hand-side vector function $\vec{f}(\bullet)$, within a single time step. If the error $||\vec{q}_{n+1}^{(5)} - \vec{q}_{n+1}^{(6)}||$ is too large, the algorithm decreases the step size h, and vise versa.

This method is analogous to the Runge–Kutta–Fehlberg method (RK45) [22], automatically keeping the trade-off between computational efficiency and approximation accuracy. Compared to RK45, RK56 theoretically achieves higher solution precision but at a cost of higher computational complexity.

CHAPTER 3

The Theory of Deep Learning and Physics-Informed Neural Networks

---

3.1 The Architecture of Multi-Layer Perceptrons

The Multi-Layer Perceptron (MLP), also known as a Feedforward Neural Network (FNN), consists of multiple fully-connected neuron layers. There are three types of layers in this architecture: input layers, hidden layers, and output layers. Every neuron in a hidden layer is linked to all neurons in the preceding and following layers, making the information spread smoothly throughout the network. Every hidden layer implements an affine transformation followed by an activation function, whereas the output layer typically involves only the affine transformation. The operations of an MLP can be described mathematically as:

$$h^{(0)} = \text{Inputs} \tag{3.1}$$

$$h^{(i)} = \sigma(W^{(i)}h^{(i-1)} + b^{(i)}), \quad i \in \{1, \cdots, L\} \tag{3.2}$$

$$h^{(L+1)} = W^{(L+1)}h^{(L)} + b^{(L+1)} = \text{Outputs} \tag{3.3}$$

Here L denotes the number of hidden layers, $h^{(i)}$ the output of each layer, and $\sigma(\cdot)$ the activation function. The $h^{(0)}$ is the input layer, primarily for data reception and not for parameter learning. The $h^{(L)}$ is the output layer, which ultimately produces the network's prediction. The intervening $L - 1$ layers, termed hidden layers, contain trainable weights and biases, which can be expressed by weight matrix $W^{(i)}$ and bias vector $b^{(i)}$ as a whole. The activation functions within these layers introduce non-linearity, enabling the MLP to learn complex functional relationships. Figure 3.1 depicts a simple two-hidden-layer MLP.

The fitting capability of MLPs is proved by the Universal Approximation Theorem (UAT) [23] [24]. It sates that a MLP model is able to fit any continuous function in the Euclidean space, as long as it has enough layers and neurons.

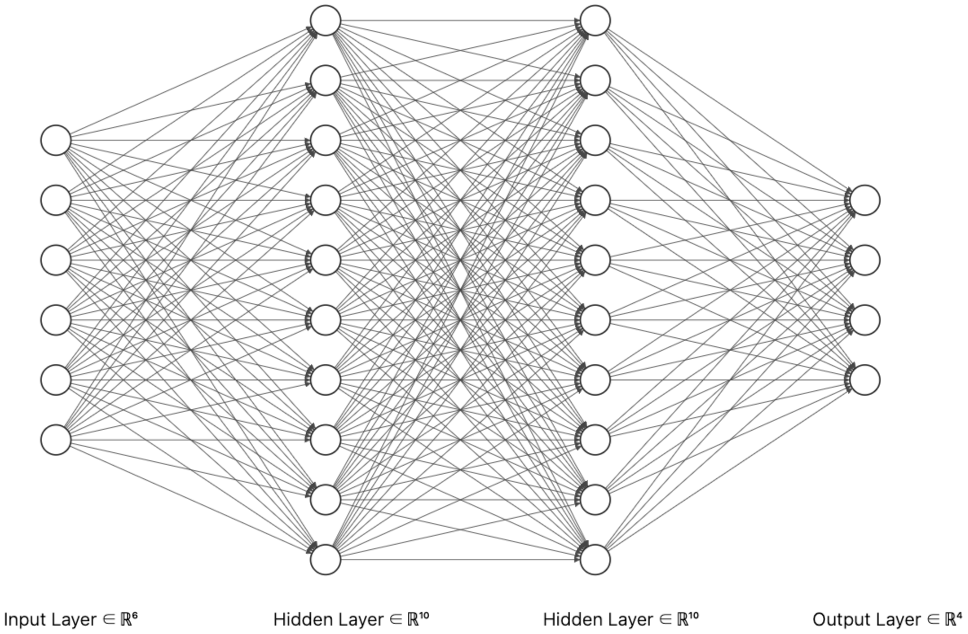

Figure 3.1: Diagram of a 2-layer MLP, featuring two hidden layers and one output layer.

3.2 The Architecture of Convolutional Neural Networks

A Convolutional Neural Network (CNN) is an architecture specifically tailored to process matrix or tensor-structured data, such as images, videos, and audio. A well-known implementation, AlexNet [25], demonstrated the exceptional accuracy of CNNs in image recognition and highlighted the significance of network depth in enhancing model performance. Each layer in a CNN includes a convolution operation and an activation function, and occasionally, a pooling operation. This layered convolutional structure enables the network to extract different levels of features from the input data. As the convolution kernels are learned automatically, explicit inversion is unnecessary. In this case, convolution is equivalent to cross-correlation operation:

$$Y_{i,j} = \sum_{p=-\lfloor k/2 \rfloor}^{\lfloor k/2 \rfloor} \sum_{q=-\lfloor k/2 \rfloor}^{\lfloor k/2 \rfloor} X_{i+p,j+q} K_{p,q} \tag{3.4}$$

where K is the convolutional kernel, X is the input matrix, and Y is the output matrix.

For this project, we focus on the One-Dimensional CNN (1D-CNN), where convolution occurs solely along one dimension, such as the time dimension in time-series datasets. The above operation now can be expressed as

$$Y_i = \sum_{p=-\lfloor k/2 \rfloor}^{\lfloor k/2 \rfloor} X_{i+p} K_p \tag{3.5}$$

We do not incorporate pooling for undersampling in this project because our target is to achieve a mapping for irregularity prediction at the same position as the input sensor data.

Compared with MLP, CNN generally has less parameters to consider, which makes its training faster. We applied the CNN model on the mass-spring-damper system to predict the rail geometry. Figure 3.2 and Figure 3.3 show the comparison between CNN and baseline models, under different damping ratios and moving speeds respectively.

3.3 The Training of Neural Networks

Training neural networks involves three critical processes: forward-propagation, back-propagation [26][27], and parameter updates. Initially, the input data traverse the entire network to generate prediction values. These predictions are subsequently used to compute the gradients (partial derivatives) of the loss function with respect to all learnable parameters of the network. Following this, the

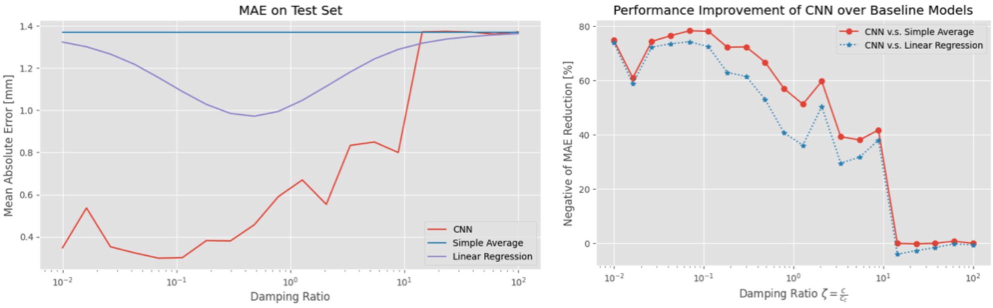

Figure 3.2: Comparison of CNN and baseline models under different damping ratios.

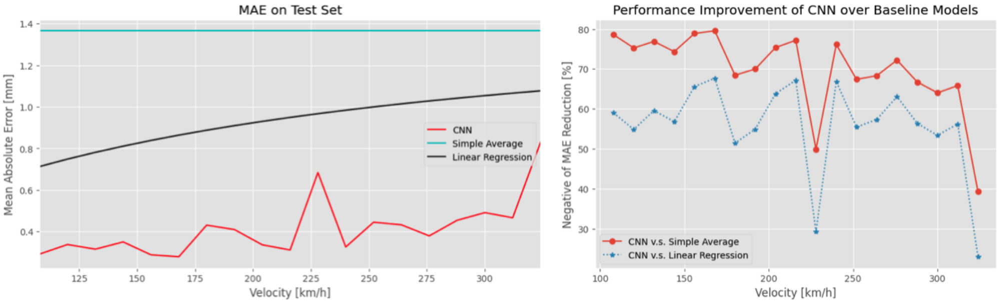

Figure 3.3: Comparison of CNN and baseline models under different moving speeds.

parameters (weights and biases) are adjusted in specific patterns based on these gradients and according to chosen optimization strategies to minimize the loss function. Commonly used optimizers include Stochastic Gradient Descent (SGD) and Adaptive Moment Estimation (Adam)[28].

We also implemented an early stopping mechanism during the network's training phase to prevent overfitting and save training time, the pseudocode for which is detailed in Algorithm 1.

3.4 Physics-Informed Neural Networks

Physics-Informed Neural Networks (PINNs) integrate neural network architectures with the principles of physics, as described in the foundational work [29]. This approach incorporates a regularization term—referred to as physics loss—into the original loss function. This term quantifies the deviation of predictions from governing differential equations (ODEs or PDEs), so as to encourage the network to learn prior physical knowledge about the system. Consequently, PINNs are also known as Theory-Training Deep Neural Networks (TTNs) [30].

The loss function of a PINN is given by:

$$L = L_{data} + \vec{\lambda}_{physics} \cdot \vec{L}_{physics} \tag{3.6}$$

where L_{data} is the same data loss as in conventional deep learning models, $\vec{L}_{physics}$ is the vector of different types of physics loss (such as observation points residuals, initial conditions, or boundary conditions), and $\vec{\lambda}_{physics}$ is the corresponding weight vector for the physics loss. Data loss is often defined as the mean square error (MSE) between the network's predictions and the target values:

$$L_{data} = \frac{1}{N} \sum_{i=1}^{N} ||\hat{y}_i - y_i||^2 \tag{3.7}$$

Here, N denotes the number of samples. And $\vec{L}_{physics}$ is defined as the MSE vector of deviations about some terms $\mathrm{DE}_i(x, y)$ of the differential equation sets (e.g. the squares of ODEs' or PDEs' left-hand-side values when keeping all right-hand-side values being 0):

$$L_{physics,j} = \frac{1}{M} \sum_{i=1}^{M} ||\hat{\mathrm{DE}}_j(x_i, \hat{y}_i) - \mathrm{DE}_j(x_i, y_i)||^2 \tag{3.8}$$

Here M is the count of terms in the differential equations considered when calculating physics loss. Note that although MSE is used to measure prediction errors during training, the network's fitting performance on validation and testing phases is evaluated using mean absolute error (MAE).

Algorithm 1 Neural Network Training with Early Stopping

```
Initialize neural network model and datasets
Set learning rate, batch size, number of epochs, patience, saving gap
Initialize optimizer and scheduler
$best\_val\_mae \leftarrow \infty$, $patience\_counter \leftarrow 0$, $saving\_counter \leftarrow 0$
for epoch in max_epochs do
    Set to training mode
    for each batch in training dataset do
        Forward-propagation to compute prediction
        Compute MSE on the batch
        Loss Back-propagation by automatic differentiation (autograd)
        Update network's parameters using optimizer
        Update optimizer's learning rate using scheduler
    end for
    Set to evaluation mode
    $val\_mae\_sum \leftarrow 0$
    for each batch in validation dataset do
        Forward-propagation to compute prediction
        Compute MAE on the batch and add to $val\_mae\_sum$
    end for
    $val\_mae \leftarrow val\_mae\_sum/num\_validation\_batches$
    $saving\_counter \leftarrow saving\_counter + 1$
    if $val\_mae < best\_val\_mae$ then
        $best\_val\_mae \leftarrow val\_mae$
        $patience\_counter \leftarrow 0$
        if $saving\_counter > saving\_gap$ then
            Save network parameters
            $saving\_counter \leftarrow 0$
        end if
    else
        $patience\_counter \leftarrow patience\_counter + 1$
        if $patience\_counter \geq patience$ then
            Early stopping triggered. Break training loop.
        end if
    end if
end for
Load the last saved parameters as the best model
```

The architecture typically includes two simultaneous applications of the network's outputs to calculate total loss, which imposes constraints from both data and physics perspectives. By incorporating prior information into the neural network, PINNs can narrow down the feasible domain of inverse problem's solutions, thus thus improving the accuracy of predictions. With adequate prior knowledge and accurate modeling of the system primarily through differential equations, it is possible to achieve good model performance even with limited training samples. Moreover, as a non-grid method, PINNs can more easily manage complex geometries or high-dimensional problems.

However, there are also some difficulties and disadvantages when using PINNs. For example, the performance of PINNs depends on an exact understanding of the physical system, particularly in the scenarios of limited data volume. If the physical model we construct not conforms well with the real-world system, then there will be much misleading for the neural networks' learning. In addition, keeping a balance between data loss and physics loss often requires many experiments to optimize $\vec{\lambda}_{physics}$, as bad values can easily make the model converge to local optimum. Sometimes solving this problem even needs automatic adjustment of loss weights or switching of loss modes.

Chapter 4

PINNs for Solving Inverse Problems

4.1 Inverse Problem for Irregularities Prediction

In our project, the inverse problem is defined as predicting the values of irregularities from dynamic data collected by various sensors on the train, denoted by the inverse process of ODEs simulation, $\mathcal{H}^{-1}(\bullet)$. There are two primary types of prediction for this task. The first is the point-to-point methods, where sensor data from a specific time point is used to predict irregularities at that exact moment. e.g. the MLP model where we input a sensor vector and out put an irregularity vector. Assuming there are n_{sens} sensors on the vehicle and n_{irr} irregularities to predict, the point-to-point method can be mathematically represented as a mapping $\mathbb{R}^{n_{sens}} \to \mathbb{R}^{n_{irr}}$, irrespective of the specific fitting models or neural networks used:

$$\text{Irregularities}(x_i) = \mathcal{H}^{-1}(\text{Sensors}(x_i)) \tag{4.1}$$

The second type of prediction method, known as the segment-to-segment approaches, utilize a segment of time-series sensor data to produce outputs that correspond to irregularities at the same or approximately the same locations. e.g. the CNN model where we input a sensor matrix and output an irregularity matrix. This relationship can be loosely defined as a mapping $\mathbb{R}^{T_{seg} \times n_{sens}} \to \mathbb{R}^{T_{seg} \times n_{irr}}$:

$$[\text{Irregularities}(x_k)]_{k=i-s}^{i} = \mathcal{H}^{-1}([\text{Sensors}(x_k)]_{k=i-s}^{i}) \tag{4.2}$$

where T_{seg} represents the number of time steps in each time-series segment. The comparative analysis of the point-to-point and segment-to-segment methods is illustrated in the following Figure 4.1.

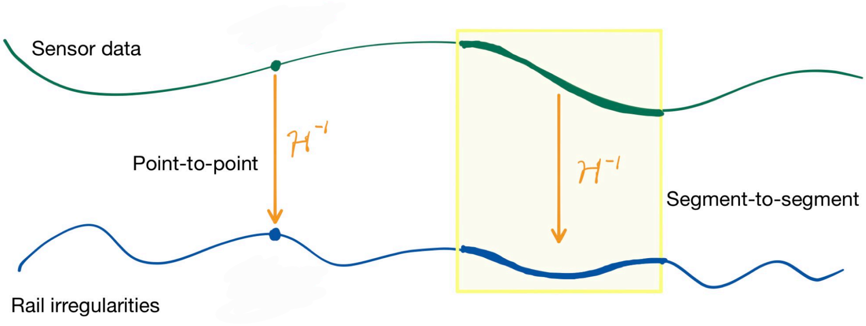

Figure 4.1: The comparison between point-to-point and segment-to-segment methods.

4.2 Testing PINNs on the Mass-Spring-Damper System

To preliminarily assess the applicability of Physics-Informed Neural Networks (PINNs) to inverse problems, we employed a simple physics-informed CNN to predict the lateral irregularities of a rail in the previously mentioned mass-spring-damper system under the critical damping condition (section 2.2). The input to the PINN is a two-dimensional vector consisting of the lateral position and lateral velocity of the ball measured by sensors, while the output is a scalar representing the rail's lateral irregularity we wish to predict.

Through our various experiments, we found that even a 2-layer or 3-layer CNN could achieve very promising predictive performance. The 3-layer CNN architecture we ultimately selected is illustrated in Figure 4.2. The kernel sizes for each layer are [3, 11, 61], and the number of kernels for each layer are [28, 8, 32], with all strides being 1 and without pooling operations. We utilized the Rectified Linear Unit (ReLU) as the activation function across all three layers. The optimizer we use is Adam.

As depicted in the algorithm flowchart (Figure 4.3), the position and velocity data are first input into the CNN to calculate the predicted irregularity. This prediction is then used to compute the mean squared error (MSE) with the target irregularity. Simultaneously, the predicted irregularity is fed into the ODE system along with the input-side vector and other dynamic parameters. Leveraging our prior knowledge of this ODE model, we can readily predict the lateral acceleration. By comparing this predicted acceleration with actual acceleration data, we calculate the physics loss. The final loss, which is a weighted sum of

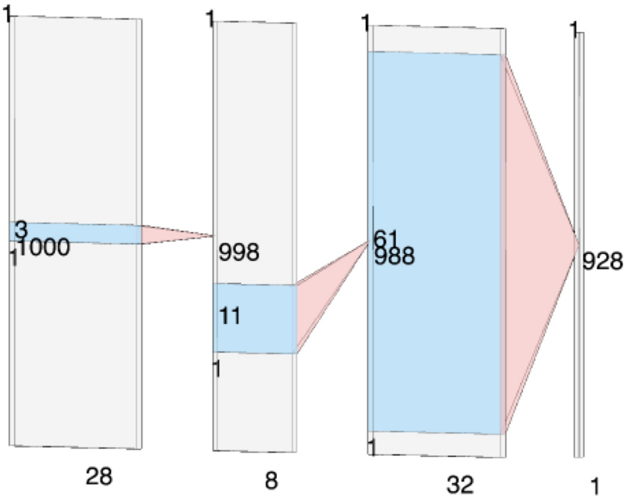

Figure 4.2: Architecture of the Physics-Informed CNN.

data loss and physics loss, is computed as follows:

$$\begin{aligned} L &= \|u - z\|_\Gamma + \lambda \cdot \|f\|_\Omega \\ &= \frac{1}{n_\Gamma} \sum_i \|NN(x_i) - \text{Irregularity}(x_i)\|_2^2 \\ &\quad + \frac{\lambda}{n_\Omega} \sum_j \|\frac{F(\mathcal{H}^{-1}(u_j, v_j), u_j, v_j)}{m} - a_j\|_2^2 \end{aligned} \tag{4.3}$$

where Γ and Ω are the observation domains for data loss and physics loss respectively, and n_Γ and n_Ω are the number of observation samples on them. $NN(\bullet)$ represent the forward-propagation of the CNN. Function $F(\bullet)$ denotes the prior knowledge we have to calculate the ball's external forces from its velocity, acceleration, and current rail irregularity. And L is the final loss that we calculate the gradient in compuation graph and used for back-propagation for the 3-layer CNN.

To demonstrate the high prediction accuracy and low data requirement of PINNs, we compared the performance of four models: a CNN trained with all samples, a CNN trained with half the samples, a PINN trained with all samples, and a PINN trained with half the samples. Except for the loss functions and the usage volume of training data, all other hyperparameters were kept the same. The predictions of lateral irregularity by these models are shown in Figure 4.4.

From the mean absolute error (MAE) comparison in Figure 4.5, it is evident that prediction errors have been significantly reduced for both 50% and 100% data usage scenarios. Remarkably, the performance of the PINN trained with 50% of the data even slightly surpasses that of the conventional CNN trained

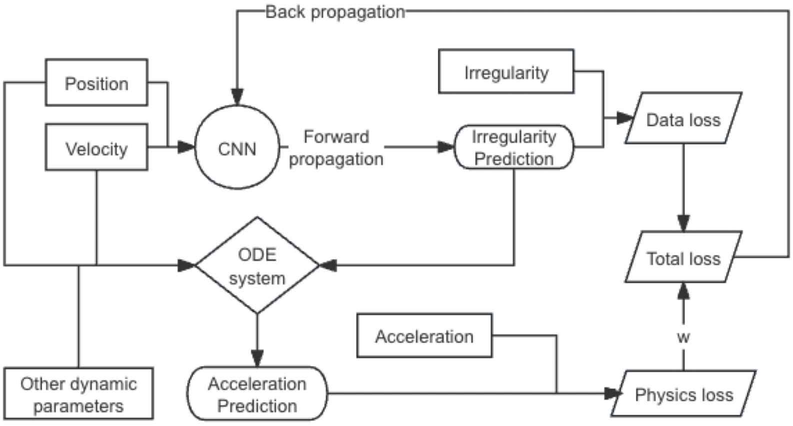

Figure 4.3: Flowchart of PINN's Training Process.

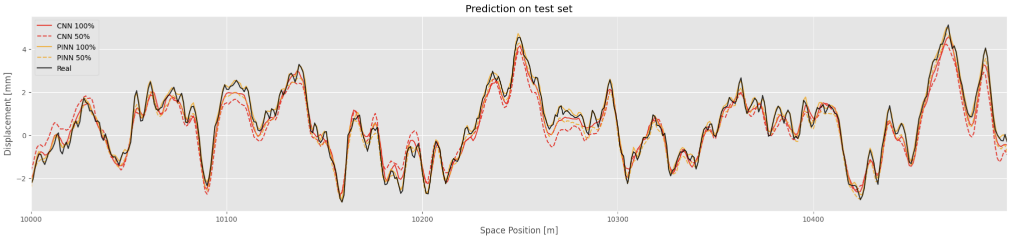

Figure 4.4: Predictions of Lateral Irregularity by CNNs and PINNs.

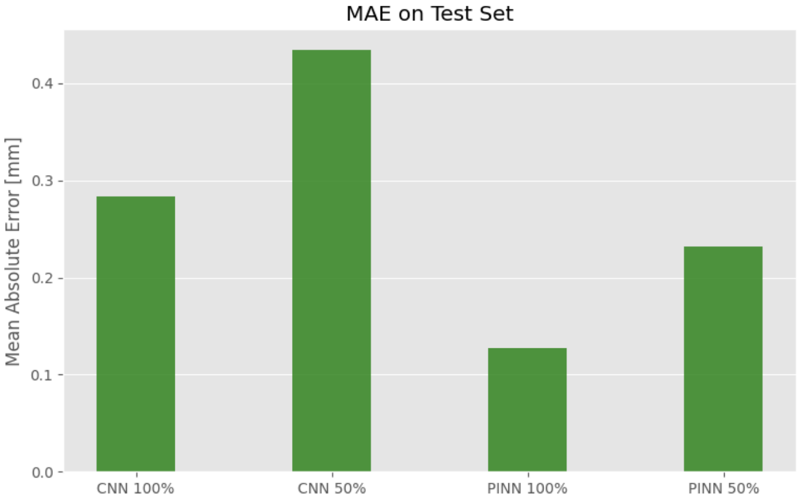

Figure 4.5: Prediction Errors of CNN and PINN on Test Set.

with 100% of the data by approximately 0.5 mm. Furthermore, the loss function curves indicates that after 1000 training epochs, the validation set MAE of the PINN with 100% data usage quickly decreases to approximately 2.5 mm, whereas the other three models only achieve around 7.5 mm. Additionally, the PINN model utilizing 50% of the data also showcases its slightly higher learning efficiency after 1500 epochs, in comparison to the other two CNN models. Consequently, we can conclude that under a good alignment between dynamic system's training data and prior knowledge of its differential equations, PINNs not only achieve higher generalization capabilities with fewer training samples, but also demonstrate faster convergence of loss compared to conventional CNNs.

4.3 Dataset Generation and Separation

We utilized MATLAB and C++ programs [8] [4] to generate track data and vehicle dynamics data, respectively. Numerical simulations were implemented at three driving speeds: 120 km/h, 160 km/h, and 200 km/h. For each speed, 50 random seeds were used to generate rail irregularities based on vector autoregression. The length of each railway is 152 km. So theoretically there should be $50 \times 152 = 7600$ km length of vehicle dynamics data. However, not all random seeds yield valid results; some lead to scenarios where the train would derail during simulations. Additionally, we discarded the first 2 km of data from each simulation to eliminate transient state effects before the system reached a steady state. Consequently, the amount of usable data for each speed we consider ranges from approximately 2000 km to 3000 km.

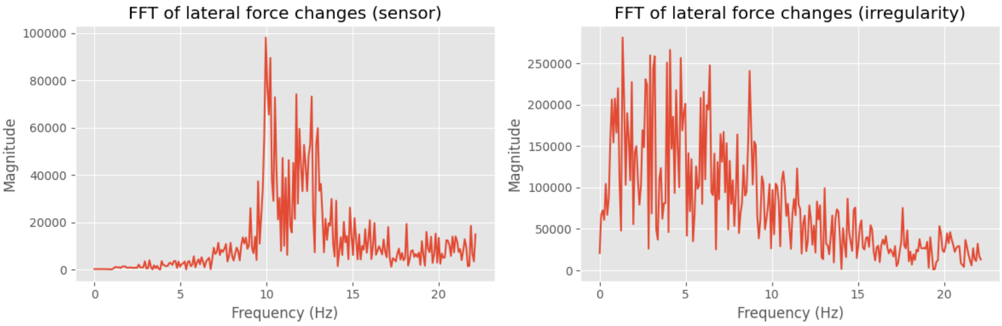

Figure 4.6: Fast Fourier Transform of force data on the rear wheelset.

The segment length is set at 500 meters to ensure coverage of all rail geometry wavelength domains ranging from 3 m to 200 m, as discussed in the literature [8] [7]. Table 4.1 presents the varying ranges of geometric wavelengths and measurement frequencies associated with different driving speeds. Specifically, D1, D2, and D3 correspond to high-frequency, medium-frequency, and low-frequency geometry information, respectively. Our Fast Fourier Transform (FFT) analysis of the forces on rear wheelset (Figure 4.6) further highlights this information across various frequency bands.

Table 4.1: Wavelength ranges of rail geometry.

Range name	Wavelength [m]	f_{120kmh} [Hz]	f_{160kmh} [Hz]	f_{200kmh} [Hz]
D1	3 - 25	1.33 - 11.11	1.78 - 14.81	2.22 - 18.52
D2	25 - 70	0.48 - 1.33	0.63 - 1.78	0.79 - 2.22
D3	70 - 200	0.17 - 0.48	0.22 - 0.63	0.28 - 0.79

Each simulation dataset contains 47 columns, which include metrics such as distance driven, integration data, state variables, and irregularities. The column names are detailed in the Appendix.

4.4 Validity of Acceleration Estimation and Physics Loss Calculation

Similar to the application of PINNs to the mass-spring-damper system, we utilize both translational and angular accelerations as criterion variables to quantify physical deviations and compute the corresponding physics losses. Although we could replicate the acceleration computation method used in the original C++ simulation code, doing so would considerably slow down the training due to the high computational complexity.

Actually, in our previous numerical simulation work, calculating a complete time-series of dynamic states for 152 km typically required over 10 hours on an Intel Xeon E5-2690 CPU. This slow computation is attributed to several factors: First, the accuracy of the numerical simulation depends heavily on frequent updates of parameters such as contact patch semi-axes and penetration depths. Second, simulating vehicle kinematics is mainly a serial computing task because current state variables depend on those from the previous time step as well as current rail irregularities, which needs to be calculated step by step, making it challenging to divide the task for parallel processing across multiple cores or CPUs. Our experiments with multi-core processors and increased RAM confirm that merely enhancing hardware capabilities does little to accelerate the dynamics simulation. A third contributing factor to the slow simulation speed may be the adaptive step size adjustment feature of the Runge-Kutta method (RK56). When estimations from 5th and 6th order RK calculations diverge significantly, the algorithm automatically reduces the step size to maintain accuracy, which, while preserving precision, consumes considerable computation time and minimally advances the time period t, thus reducing overall computational efficiency.

Consequently, it is crucial and beneficial to modify and simplify the acceleration computation logic within the physics loss calculation component of PINNs' training to avoid extremely slow training processes across thousands of training epochs and mini-batch computations. We firstly streamlined the state update process in the serial computing mode of acceleration computation and adjusted variable processing to better suit neural network training. We replaced the Runge-Kutta method with a fixed-step differential representation, because in neural networks' training phase, the real state variables at each step are known; thus, errors can only arise from the network's irregularity predictions, as well as the computation in current time step using the completely precise data of last time step's system state and current irregularities. The modified acceleration calculation algorithm we wrote in Python aligns very well with accelerations generated by the original numerical simulation algorithm in C++, while significantly reducing computation time. Figures 4.7 and 4.8 show comparisons of lateral and vertical accelerations on the front wheelset between the original simulation in C++ and our modified calculations in Python.

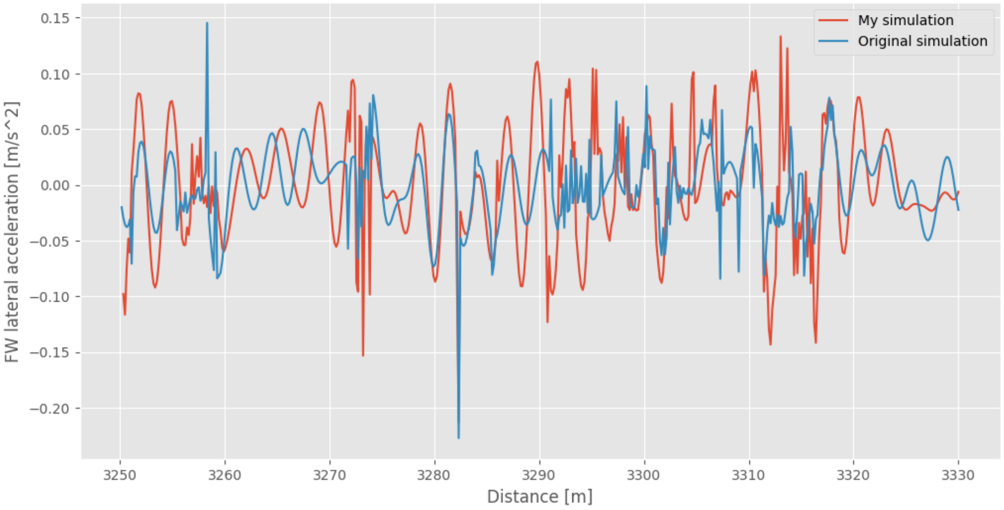

Figure 4.7: Comparison of Lateral Accelerations between the Original and Modified Calculations.

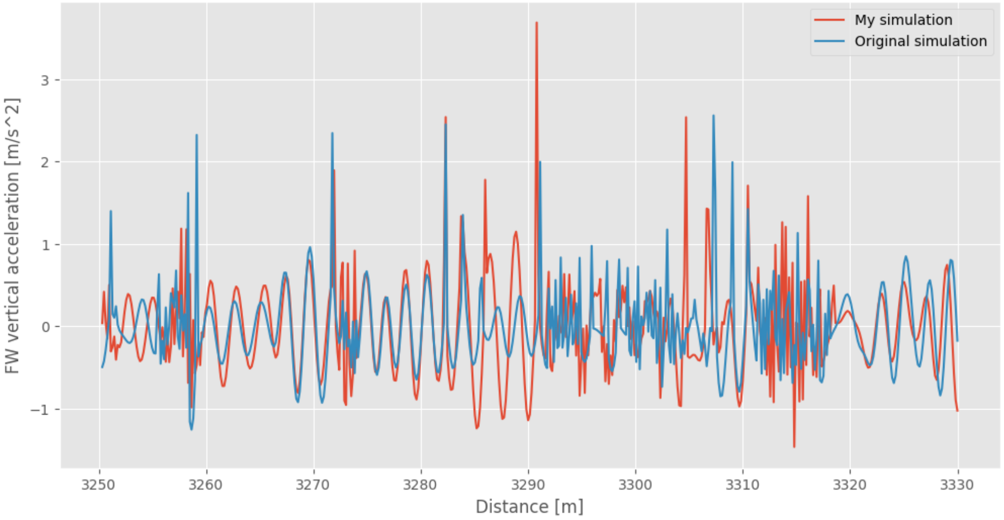

Figure 4.8: Comparison of Vertical Accelerations between the Original and Modified Calculations.

To make the best use of the capabilities of PyTorch's autograd for gradient calculation, we improved our code to be compatible with PyTorch tensor operations, which can further enhance model's training efficiency.

The total loss is calculated as:

$$L = L_{data} + \lambda_{lateral} \cdot L_{lateral} + \lambda_{vertical} \cdot L_{vertical} \tag{4.4}$$

where $L_{lateral}$ is the physical loss of wheelsets' lateral accelerations, and $L_{vertical}$ is the physical loss of wheelsets' vertical accelerations.

Note that physics loss is used only during the training phase, and the input of state variables is obtained precisely from system's numerical simulation. In the validation and testing phases, forward-propagation is implemented in the identical way as conventional methods, i.e., domain knowledge only influences model's training through the design of loss function, but does not directly participate in the evaluation of model's performance.

Based on the physics loss calculation method, we have designed the training process for the PINN as depicted in Algorithm 2.

Algorithm 2 Training of PINN for Vehicle Dynamics

```
Initialize neural network model and datasets
Set hyperparameters
Calculate target accelerations $a_{lateral}$ and $a_{vertical}$
$loss\_mode \leftarrow$ "mixed"
for epoch in max_epochs do
    Set to training mode
    Forward-propagation to compute prediction
    Compute data MSE
    if $loss\_type =$ "mixed" then
        Compute estimated acceleration $\hat{a}_{lateral}$ and $\hat{a}_{vertical}$ from the irregularity prediction and state variables
        Compute physics MSEs $||\hat{a}_{lateral} - a_{lateral}||^2$ and $||\hat{a}_{vertical} - a_{vertical}||^2$
        $Loss \leftarrow$ weighted average of data MSE and physics MSEs
    else if $loss\_type =$ "data" then
        $Loss \leftarrow$ data MSE
    end if
    Loss Back-propagation by automatic differentiation (autograd)
    Update network's parameters using optimizer
    Update optimizer's learning rate using scheduler
    Set to evaluation mode
    Forward-propagation to compute prediction
    Compute MAEs for irregularities, lateral accelerations, and vertical accelerations
    Compute estimated acceleration $\hat{a}_{lateral}$ and $\hat{a}_{vertical}$ from the irregularity prediction and state variables
    Compute physics MAEs $||\hat{a}_{lateral} - a_{lateral}||$ and $||\hat{a}_{vertical} - a_{vertical}||$
    if $||\hat{a}_{lateral} - a_{lateral}|| <$ lateral_threshold or $||\hat{a}_{vertical} - a_{vertical}|| <$ vertical_threshold then
        $loss\_mode \leftarrow$ "data"
    end if
end for
```

Chapter 5

Network Tuning and Relevant Experiments

5.1 Comparison of Loss Function Types

We currently employ several types of loss functions to train our CNN: The first is data loss, which measures the discrepancy between predicted and target values, akin to traditional methods. Another type is mixed loss, used in PINNs, which is defined as the weighted average between data error and physics error. The physics error comprises deviations in lateral and vertical accelerations of the front wheelset and the rear wheelset within our vehicle dynamic system. Additionally, we have the option to train the network solely on physics loss, whether by focusing on lateral acceleration, vertical acceleration, or a combination of both.

Given our limited computational resources and the extensive training time required for all the dynamics data we have simulated, we opted for a relatively simple model structure and a lightweight dataset to implement the experiments. We use a 3-layer CNN, with each layer containing 8 channels of the same kernel size 19, to capture the rail geometry within the small wavelength domain (high-frequency information). The input to the networks is a 30-dimensional vector representing sensor measurements, and the output is an 8-dimensional vector representing rail irregularities at the positions of 8 considered wheels. For hyper-parameter tuning, we selectively use part of the dynamics data under a driving speed of 120 km/h.

We trained the aforementioned CNN architecture with both data loss and mixed loss respectively. Figure 5.1 illustrates their mean absolute error (MAE) convergence curves on the validation set during training phase. Predictions of rail irregularities on the test set are shown in Figure 5.2 for lateral irregularities and Figure 5.3 for vertical irregularities. The CNN model ultimately achieved a prediction MAE of 0.443 mm, while the PINN model reduced this error by 14.9% to 0.377 mm.

From the data loss curve, we observe that training with mixed loss enhances

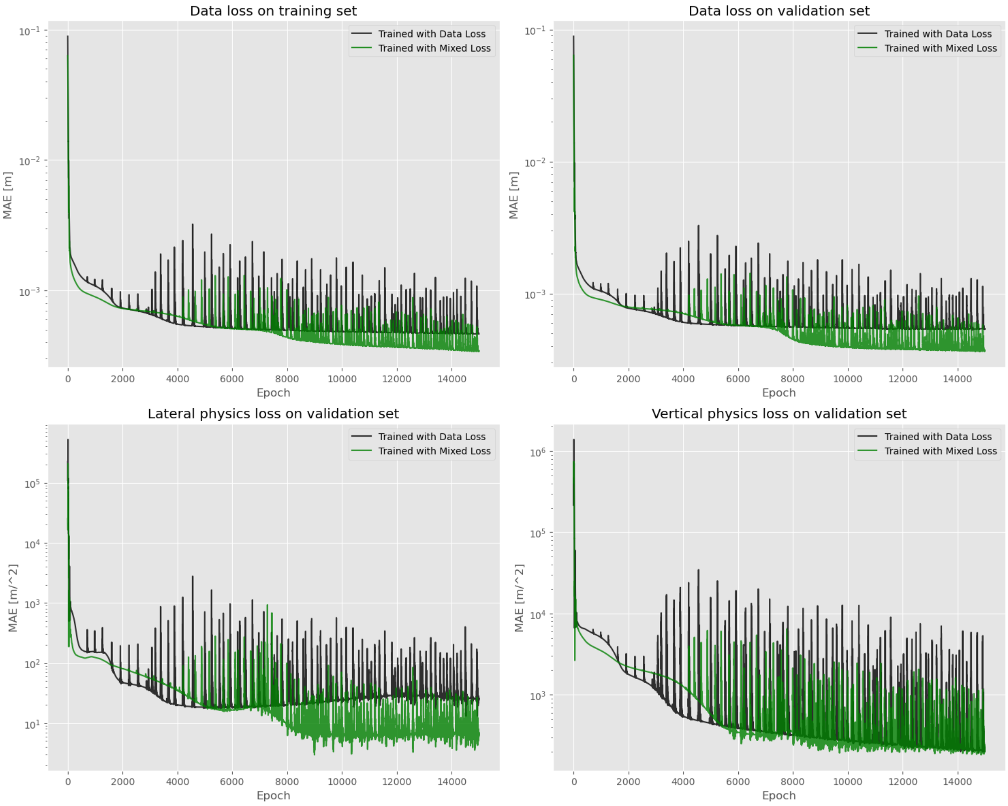

Figure 5.1: Data and physics MAE convergence comparison between CNN (data loss mode) and PINN (mixed loss mode).

model convergence speed in the initial period, primarily due to the higher efficacy of physics information compared to data information in this stage. The physics information also contributes to higher prediction accuracy, as evidenced by the data loss curves, for the reason that our prior knowledge about the vehicle dynamics system provides additional information to the neural network that the training dataset alone cannot offer.

Furthermore, the physics information in the PINN model demonstrates a stronger ability to learn lateral geometry information, significantly enhancing accuracy concerning lateral wheelset acceleration. This improvement explains why, in Figure 5.2, the mixed loss mode curve more closely aligns with the ground truth, whereas in Figure 5.3, both types perform well in learning vertical accelerations.

It is also important to note that our experiments using complete physics loss for training did not yield favorable prediction results. This may be due to the limited physics indices used to construct the physics loss, as we have only

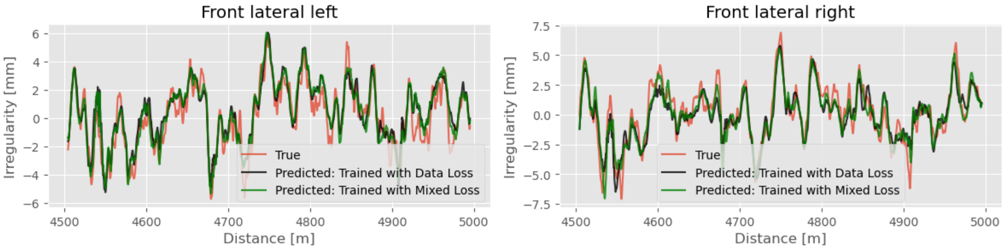

Figure 5.2: Lateral irregularities prediction under the data loss (CNN) and mixed loss (PINN) training modes.

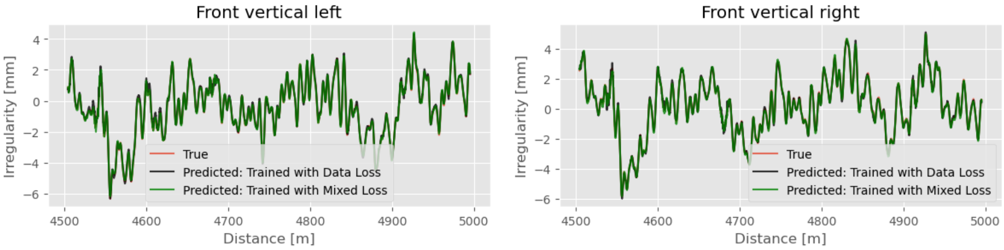

Figure 5.3: Vertical irregularities prediction under the data loss (CNN) and mixed loss (PINN) training modes.

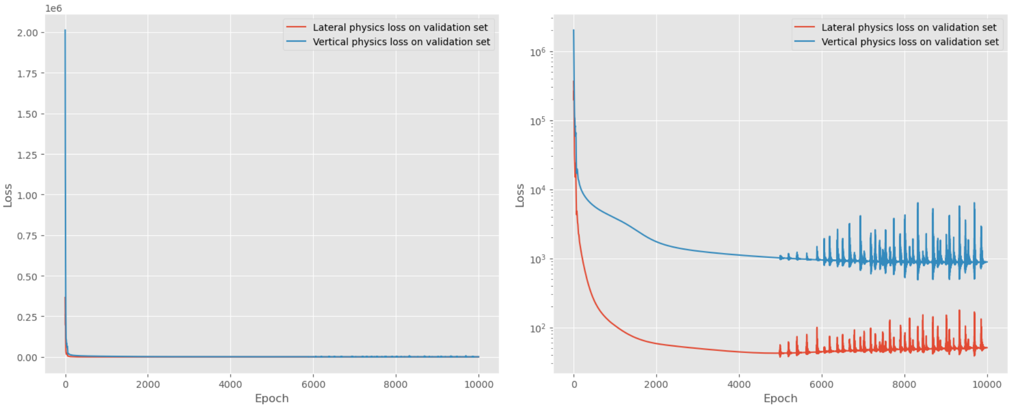

Figure 5.4: Lateral and Vertical Losses on the Validation Set.

considered lateral and vertical accelerations of wheelsets so far. It is difficult for neural networks to deduce the system's operation based solely on measurements from just two 2-axis accelerometers, since the solution to the inverse problem may not be unique, i.e., many other railway geometries could also yield the same accelerometer data. For this reason, the prior physics knowledge serves merely as an auxiliary factor to constrain the solution space, rather than the primary determinant in searching the direction of the solution vector. Therefore, the mixed loss, comprising weighted data loss and physics loss, proves to be a more effective approach for our subsequent experiments.

5.2 Tuning of Physics Weights and Thresholds

5.2.1 Ratio between Lateral Weight and Vertical Weight

After some experiments with different network architectures, we observed the difference between the magnitude orders in prediction MAEs for lateral and vertical accelerations. The MAE for vertical acceleration is roughly an order of magnitude larger than that for lateral acceleration in the validation set, as shown in the convergence values in Figure 5.4. To adjust these weighted physics losses to a same level, considering that mean squared error (MSE) is used to represent loss on the training set, we have:

$$\log \sqrt{\frac{\lambda_{lateral}}{\lambda_{vertical}}} \approx 1 \tag{5.1}$$

After a grid search within neighboring range, with 15000 epochs for each test (as shown in Figure 5.5 and Figure 5.6), we determined the optimal ratio to be

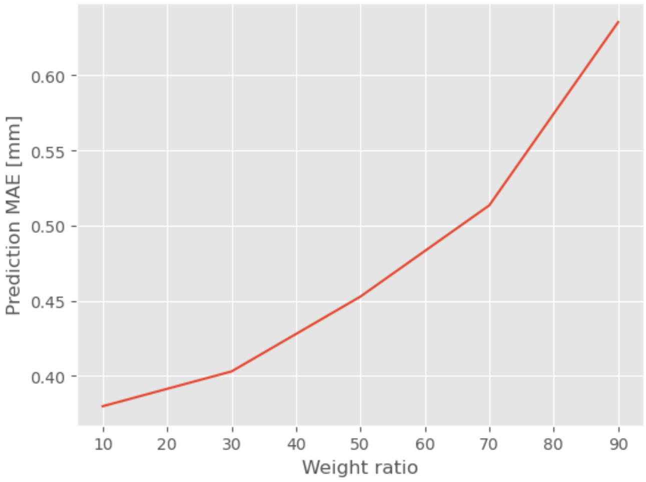

Figure 5.5: Prediction MAE comparison for physics weight ratios in [10, 100].

16 approximately, which means $\lambda_{lateral}/\lambda_{vertical} = 16^2 = 256$. Figure 5.7 displays the MAE convergence curves under different physics weight ratios. We can see that small variations within this range do not significantly affect model's final performance.

5.2.2 Tuning Lateral Weight's Order of Magnitude

Through several experiments, we found that the order of magnitude of the lateral weight does not significantly affect prediction accuracy (Figure 5.8), provided that the number of training epochs exceeds 15000. However, it does impact the loss convergence speed during the initial phase of model training. This effect arises because it modulates the balance between given data and prior knowledge. Consequently, we have set this hyperparameter to 10^{-15} for our subsequent implementations.

5.2.3 Optimal Physics Weight Thresholds

Assuming that the lateral and vertical acceleration MAEs are of the same order of magnitude in terms of their convergence rates, then the ratio between the vertical and lateral MAE thresholds should also match the ratio of $\frac{\lambda_{lateral}}{\lambda_{vertical}}$, which is 16. This is supported by observing the plateau values of the physics MAE after training for 50000 epochs (Figure 5.9). We did the grid search in the range [20, 200], with each test running for 20000 epochs. The result is shown in Figure 5.10. We can find that the prediction MAE is kept in a relatively low value in the range of about [40, 150]. Thus we choose the lateral physics threshold to be 100 and the vertical physics threshold to be 1600 in our following PINN models.

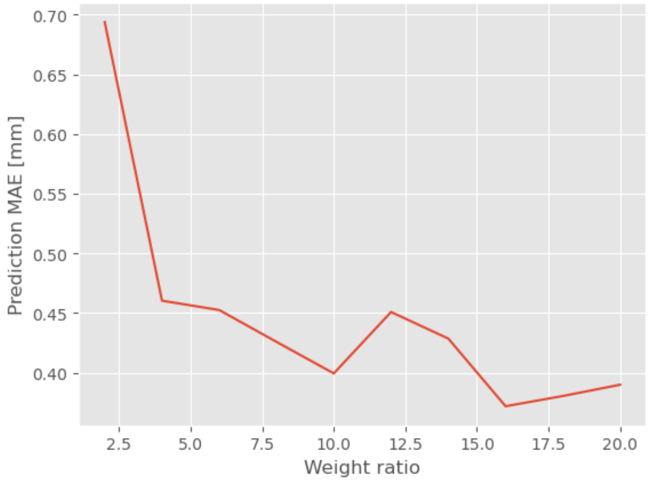

Figure 5.6: Prediction MAE comparison for physics weight ratios in [2, 20].

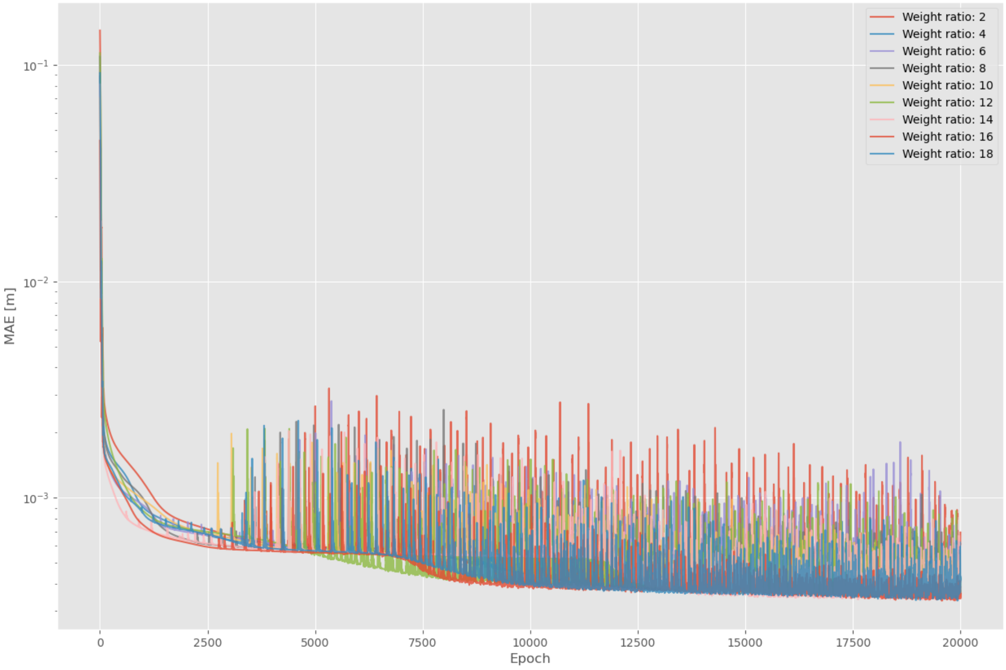

Figure 5.7: Validation set MAE convergence under different physics weight ratios.

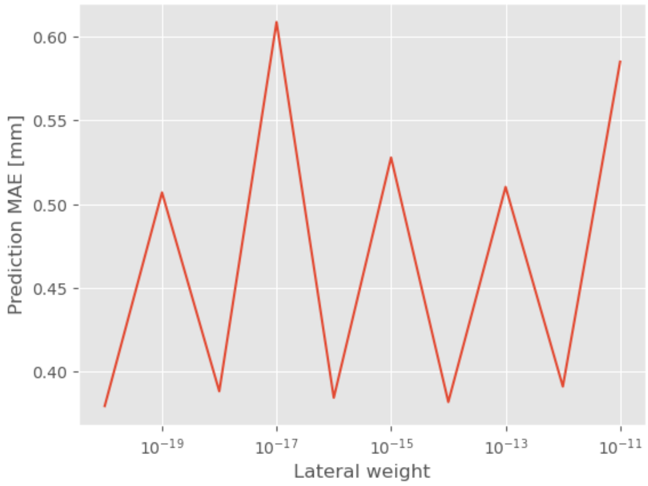

Figure 5.8: Prediction MAEs across aarious orders of magnitude for the lateral physics weight.

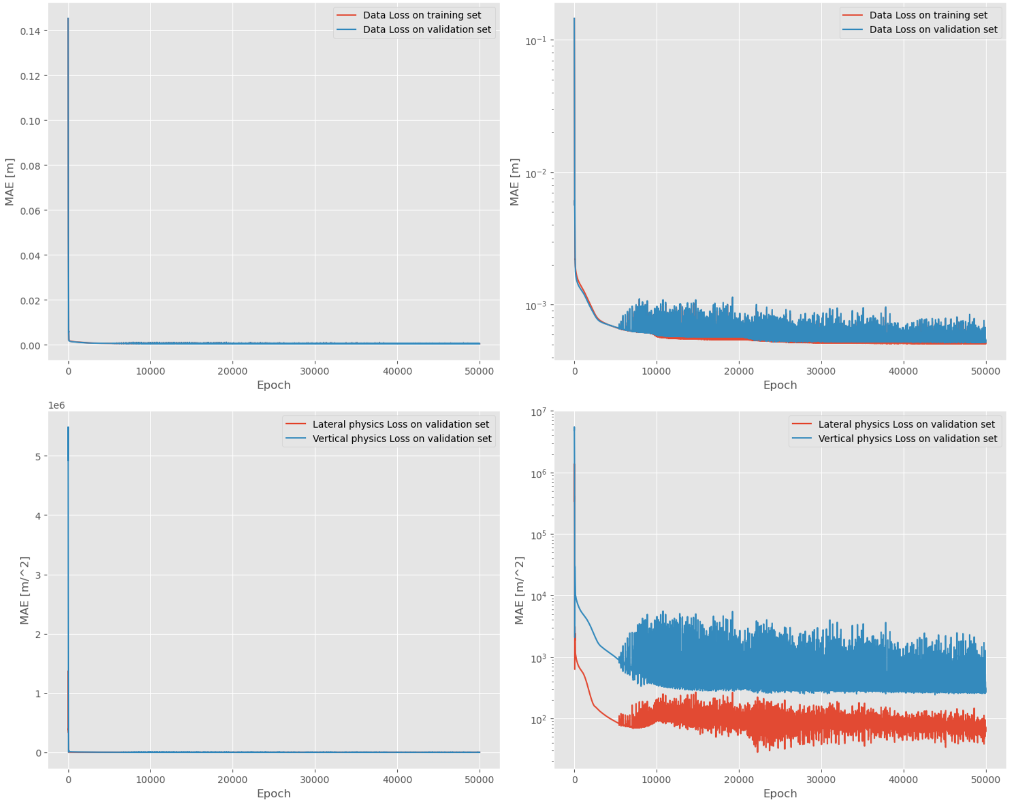

Figure 5.9: Losses following 50000 training epochs.

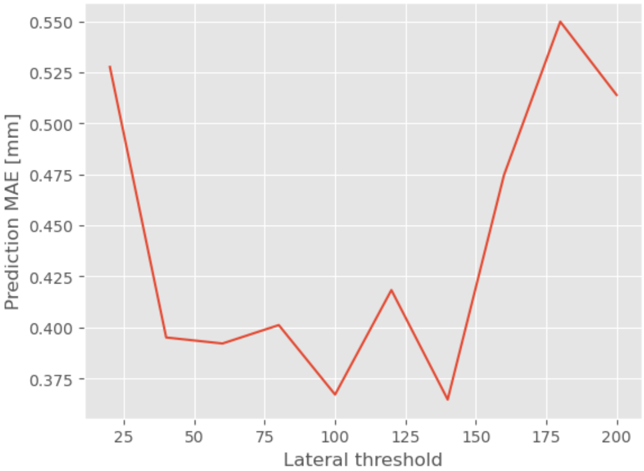

Figure 5.10: Prediction MAE comparison for lateral physics threshold [20, 200].

5.3 Tuning CNN Architecture and Other Hyperparameters

We have tested the training of PINNs with several different network architectures, and the final hyperparameters we have chosen is listed in Table 5.1. The loss convergence curves and prediction results of the final model are shown in Figure 5.11 and Figure 5.12. Our final model achieved a prediction MAE of 0.329 mm on the testing set.

Table 5.1: Hyperparameters in the final PINN model.

Hyperparameter	Value
Physics weight (lateral)	1.0e-15 $(m/s)^{-2}$
Physics weight (vertical)	3.9e-18 $(m/s)^{-2}$
Physics loss threshold (lateral)	100 $(m/s)^{-1}$
Physics loss threshold (vertical)	1600 $(m/s)^{-1}$
Number of hidden layers	3
Kernel number	[16, 64, 16]
Kernel size	[5, 21, 21]
Output layer kernel size	3
Optimizer	Adam
Scheduler	None
Number of epochs	20000
Activation function	ELU
Learning rate	1.0e-3
Resolution	0.16 m

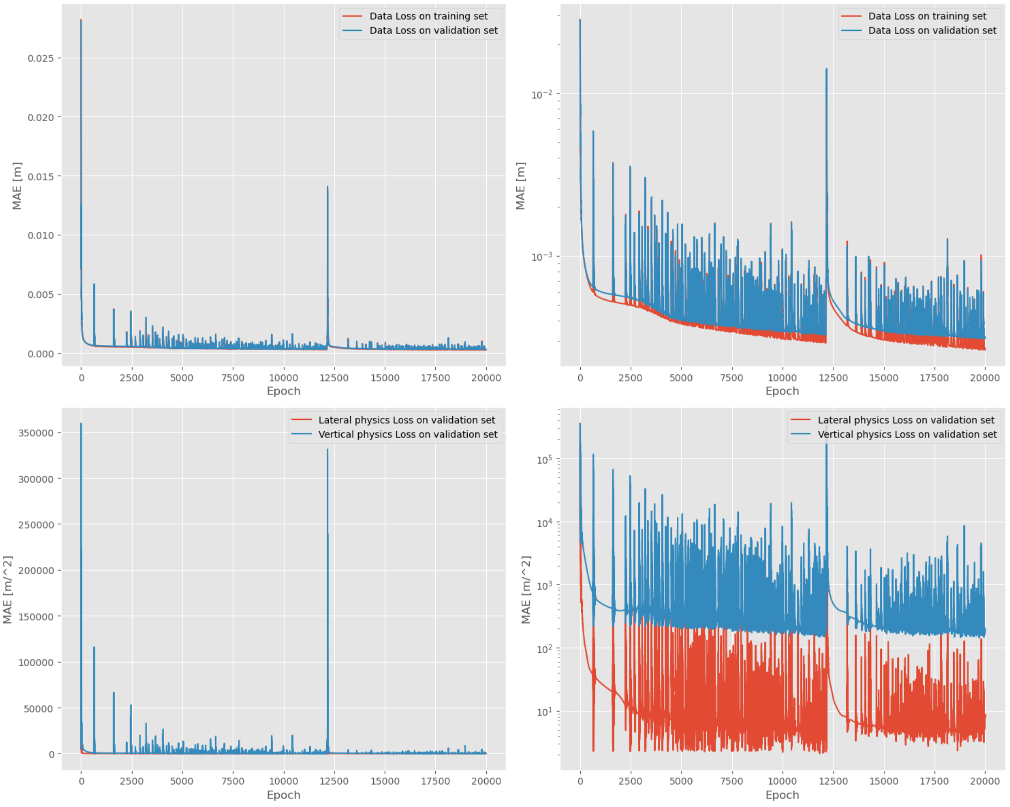

Figure 5.11: Loss convergence of the final model.

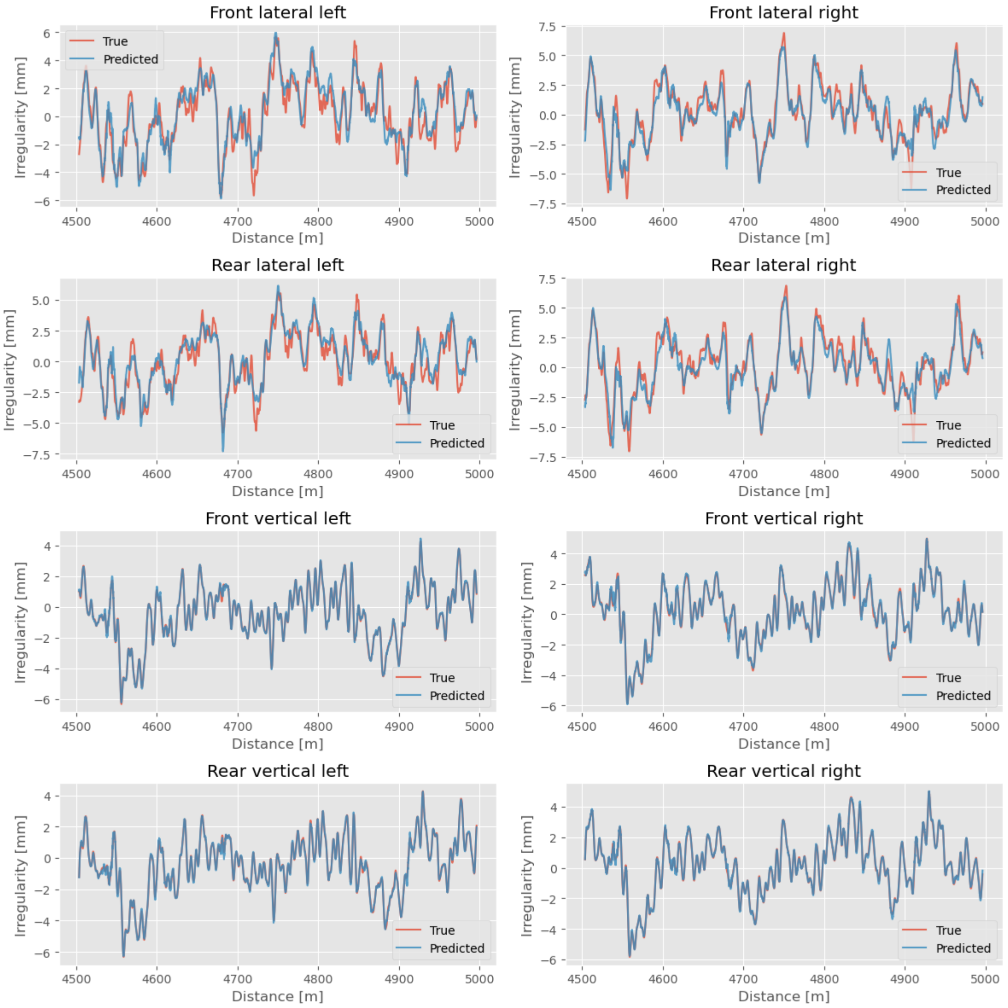

Figure 5.12: Irregularity prediction of the final model.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

In this thesis, we explored the use of Physics-Informed Neural Networks (PINNs) to predict railway track irregularities utilizing on-board sensor data from in-service train vehicles. By integrating the system's governing differential equations in to the loss function and developing an algorithm for the automatic switching between loss types, we enhanced the neural network's learning capabilities. The inclusion of physics-based loss during the training phase proved that PINNs can outperform traditional deep learning approaches, such as CNNs, in terms of prediction accuracy and training efficiency.

Our experiment results on both the mass-spring-damper model and the dynamic railway vehicle system demonstrated that PINNs not only enhance prediction accuracy but also accelerate the convergence of loss, compared to conventional models with identical network structures. Additionally, our findings indicate that PINNs are adept at handling time-variant complex systems where data might be scarce or costly to obtain.

This research highlights the considerable potential of PINNs as a powerful tool for real-time detection and predictive maintenance of railway infrastructure, showing their broader applicability in other scientific and engineering fields.

6.2 Research Limitations and Future Work

In this project we only used simulated vehicle data for neural networks training. But It would be more beneficial to use the real measurement data from train sensors, such as accelerometers or gyroscopes, to verify the effectiveness of the PINN models in a real engineering scenario.

PINN models have a strict requirement for domain knowledge to align with the real physical system, including the structure of differential equations and system parameters. Future investigations should assess the robustness of PINNs under physical knowledge deviations. For instance, if a spring or damper in the suspension systems is experiencing wear and tear, whose coefficient is not updated in the physics loss calculation, how will this influence the training results of PINNs?

Moreover, since we have only considered wheelset accelerations to calculate the physics loss so far, other work can also be done in integrating more physical indices, to further improve the prediction accuracy with more domain knowledge.

Bibliography

[1] S. Ma, L. Gao, X. Liu, and J. Lin, “Deep learning for track quality evaluation of high-speed railway based on vehicle-body vibration prediction,” in *IEEE Access*, 2019.

[2] C. D. Stoura, V. K. Dertimanis, C. Hoelzl, C. Kossmann, A. Cigada, and E. N. Chatzi, “Stoura c d, dertimanis v k, hoelzl c, et al. a model-based bayesian inference approach for on-board monitoring of rail roughness profiles: Application on field measurement data of the swiss federal railways network,” in *Structural Control and Health Monitoring*, 2023.

[3] Q. Wang, W. Ding, Q. He, and P. Wang, “Estimation of railway vehicle response for track geometry evaluation using branch fourier neural operator,” in *arXiv preprint arXiv:2402.18366*, 2024.

[4] L. Engbo, “The dynamics of a railway vehicle on a disturbed track – modelling of lateral irregularities,” in *Master’s Thesis, Department of Physics, The Technical University of Denmark*, Jun. 2001.

[5] E. Østergaard, “Documentation for the sdirk c++ solver,” in *IMM, The Technical University of Denmark (DTU), Denmark*, 1998.

[6] W. Kik, “Rsgeo and rsprof programs for the simulation of the wheel-rail or the wheelset-roller kinematics, translation j,” in *Litzenburger, DTU*, 2000.

[7] A. Plesner, A. P. Engsig-Karup, and H. True, “Advanced cnn and conformal prediction techniques for railway track irregularity prediction,” in *The Technical University of Denmark (DTU), Denmark*, Jun. 2000.

[8] A. Plesner, “Using data-driven state-of-the-art machine learning and conformal prediction for track irregularities from observed dynamics of in-service railway vehicles,” in *Master’s Thesis, Department of Applied Mathematics and Computer Science, The Technical University of Denmark*, 2022.

[9] L. Lu, P. Jin, and G. E. Karniadakis, “Deeponet: Learning nonlinear operators for identifying differential equations based on the universal approximation theorem of operators,” in *Nature Machine Intelligence*, Mar. 2021.

[10] Z. Li, N. Kovachki, K. Azizzadenesheli, B. Liu, K. Bhattacharya, A. Stuart, and A. Anandkumar, “Fourier neural operator for parametric partial differential equations,” in *arXiv preprint arXiv:2010.08895*, 2020.

[11] B. Raonic, R. Molinaro, T. Rohner, S. Mishra, and E. de Bezenac, "Convolutional neural operators," in *ICLR 2023 Workshop on Physics for Machine Learning,*, 2023.

[12] W. Xiong, X. Huang, Z. Zhang, R. Deng, P. Sun, and Y. Tian, "Koopman neural operator as a mesh-free solver of non-linear partial differential equations," in *arXiv preprint arXiv:2301.10022*, 2023.

[13] M. Raissi, P. Perdikaris, and G. Karniadakis, "Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations," in *Journal of Computational physics*, 2019.

[14] T. D. Ryck, S. Mishra, R. Molinaro, and T. K. Rusch, "Deep learning in scientific computing lecture notes," in *Department of Mathematics, ETH Zürich, Switzerland*, Feb. 2022.

[15] S. Markidis, "Physics-informed deep-learning for scientific computing," in *KTH Royal Institute of Technology, Sweden*, 2021.

[16] P. Thanasutives, T. Morita, M. Numao, and K. ichi Fukui, "Noise-aware physics-informed machine learning for robust pde discovery," in *Machine Learning: Science and Technology*, 2023.

[17] A. D. Jagtap and G. E. Karniadakis, "Extended physics-informed neural networks (xpinns): A generalized space-time domain decomposition based deep learning framework for nonlinear partial differential equations," in *Communications in Computational Physics*, 2020.

[18] E. A. Antonelo, E. Camponogara, L. O. Seman, J. P. Jordanou, E. R. de Souza, and J. F. Hübner, "Physics-informed neural nets for control of dynamical systems," in *Neurocomputing*, 2024.

[19] S. Rout, V. Dwivedi, and B. Srinivasan, "Numerical approximation in cfd problems using physics informed machine learning," in *arXiv preprint arXiv:2111.02987*, 2021.

[20] P. Isaksen and H. True, "On the ultimate transition to chaos in the dynamics of cooperrider's bogie," in *Chaos, Solitons and Fractals*, 1997.

[21] Z. Y. Shen, J. K. Hedrick, and J. A. Elkins, "A comparison of alternative creep force models for rail vehicle dynamic analysis," in *Vehicle System Dynamics*, 1983.

[22] E. Fehlberg, "Low-order classical runge-kutta formulas with stepsize control and their application to some heat transfer problems," in *National aeronautics and space administration*, 1969.

[23] K. Hornik, M. Stinchcombe, and H. White, “Multilayer feedforward networks are universal approximators,” in *Neural Networks*, 1989.

[24] B. C. Csáji, “Approximation with artificial neural networks,” in *Faculty of Sciences, Eötvös Loránd University, Hungary*, 2001.

[25] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” in *Advances in Neural Information Processing Systems*, 2012.

[26] H. J. KELLEY, “Gradient theory of optimal flight paths,” in *Ars Journal*, 1960.

[27] A. E. Bryson, “A gradient method for optimizing multi-stage allocation processes,” in *Harvard Univ. Symposium on digital computers and their applications*, 1961.

[28] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” in *arXiv preprint arXiv:1412.6980*, 2014.

[29] M. Raissi, P. Perdikaris, and G. E. Karniadakis, “Physics informed deep learning (part i): Data-driven solutions of nonlinear partial differential equations,” in *arXiv preprint arXiv:1711.10561*, 2017.

[30] M. T. Rad, A. Viardin, G. Schmitz, and M. Apel, “Theory-training deep neural networks for an alloy solidification benchmark problem,” in *Computational Materials Science*, 2020.

APPENDIX A

Shorthand Notation

Table A.1: Shorthand notations used in the thesis.

Shorthand	Full name
MSE	Mean Squared Error
MAE	Mean Absolute Error
VAR	Vector Auto-Regression
ODE	Ordinary Differential Equation
PDE	Partial Differential Equation
RK	Runge-Kutta method
RK45	Runge–Kutta–Fehlberg method
RK56	5th-6th-order Runge-Kutta method
SHE	Shen-Hedrick-Elkins theory
NN	Neural Network
MLP	Multi-Layer Perceptron
CNN	Convolutional Neural Network
PINN	Physics-Informed Neural Network
cPINN	Conservative Physics-Informed Neural Network
XPINN	Extended Physics-Informed Neural Network
TTN	Theory-Trained Neural Network
SGD	Stochastic Gradient Descent
ReLU	Rectified Linear Unit
HSR	High-Speed Railway
IMU	Inertial Measurement Unit
CIT	Comprehensive Inspection Train
VBA	Vehicle-Body Acceleration
PBTG	Performance-Based Track Geometry
TRV	Track Recording Vehicles
MSD	Mass-Spring-Damper model
FNO	Fourier Neural Operator
FFT	Fast Fourier Transform
SOTA	State-Of-The-Art
DeepONet	Deep Operator Networks
AD	Automatic Differentiation

Appendix B

Parameters of the Dynamic System

Table B.1: Definitions and values of the parameters in ODEs.

Parameter	Description	Value	Unit
k_1, k_2, k_3	Primary suspension spring coefficient	1823.0, 3646.0, 3646.0	kN/m
k_4, k_5, k_6	Secondary suspension spring coefficient	182.3, 333.3, 2710.0	kN/m
D_1, D_2, D_6	Secondary suspension damper coefficient	20.0, 29.2, 500.0	kNs/m
b	Longitudinal characteristic length	1.074	m
d_1, d_2	Lateral characteristic length	0.62, 0.68	m
h_1, h_2, h_3	Vertical characteristic length	0.0762, 0.6584, 0.8654	m
h_1, h_2, h_3	Vertical characteristic length	0.0762, 0.6584, 0.8654	m
m_w	Wheelset's mass	1022.0	kg
I_{wx}, I_{wy}, I_{wz}	Wheelset's moment of inertia around axes	678.0 80.0 678.0	kgm^2
m_b	Bogie frame's mass	2918.9	kg
I_{bx}, I_{by}, I_{bz}	Bogie frame's moment of inertia around axes	6780.0 6780.0 6780.0	kgm^2
m_c	Car body's mass	44388.0	kg
I_{cx}	Car body's moment of inertia around x-axis	280000.0	kgm^2
g	Gravitational acceleration	9.81	m/s^2

APPENDIX C

Programming Environment

C.1 Programming Languages

The generation of track data is implemented in MATLAB R2024a. Numerical simulation of vehicle's dynamic system is realized using C++11. Deep Learning models and main data processing are employed in Python 3.8.19.

C.2 Packages and Toolboxes

Table C.1: Python packages used in this thesis.

Package name	Version
pandas	1.4.4
numpy	1.19.2
matplotlib	3.6.2
scipy	1.6.2
tqdm	4.66.4
torch	2.3.1
sklearn	1.2.1
IPython	8.12.2
seaborn	0.12.2
os	built-in
math	built-in
time	built-in

C.3 Computational Hardware

We mainly use the hardware resources from the Computer Engineering and Networks Laboratory, ETH Zurich.

Table C.2: MATLAB toolboxes used in this thesis.

Toolbox name	Version
Optimization Toolbox	24.1
Statistics and Machine Learning Toolbox	24.1
Econometrics Toolbox	24.1
Signal Processing Toolbox	24.1

In numerical simulations for the train dynamic system, we use the Dual Octa-Core Intel Xeon E5-2690 CPU (node arton03 in the lab cluster). We allocate 1 GB memory and 1 core to each simulation on a 152 km track data. Thus, according to the resources constraints by the laboratory clusters, we can run 16 simulations at the same time, with the allocated 16 GB memory and 16 CPU cores.

It the work related to deep learning models, the GPU we are using is NVIDIA TITAN Xp with CUDA 12.2 (node tikgpu02 in the lab cluster), allocated with 64 GB memory. Before large-scale training on the full formal dataset, I also do some light-weight demo verification on my MacBook Air with M3 chip, using the Metal Performance Shaders (MPS) 14.6.1 framework.

APPENDIX D

Extra Results and Extra Plots

D.1 Numerical Simulation

Table D.1: Valid random seeds for rail geometry generation.

Driving speed	Valid seeds from 1 to 50
120 km/h	4, 5, 6, 7, 8, 12, 13, 14, 15, 19, 42, 44, 45, 48
160 km/h	3, 5, 6, 7, 11, 14, 19, 22, 23, 24, 28, 34, 42, 47
200 km/h	3, 4, 10, 11, 14 19, 22, 23, 24, 27, 28, 35, 47

D.2 PINNs Application on Mass-Spring-Damper Dynamics

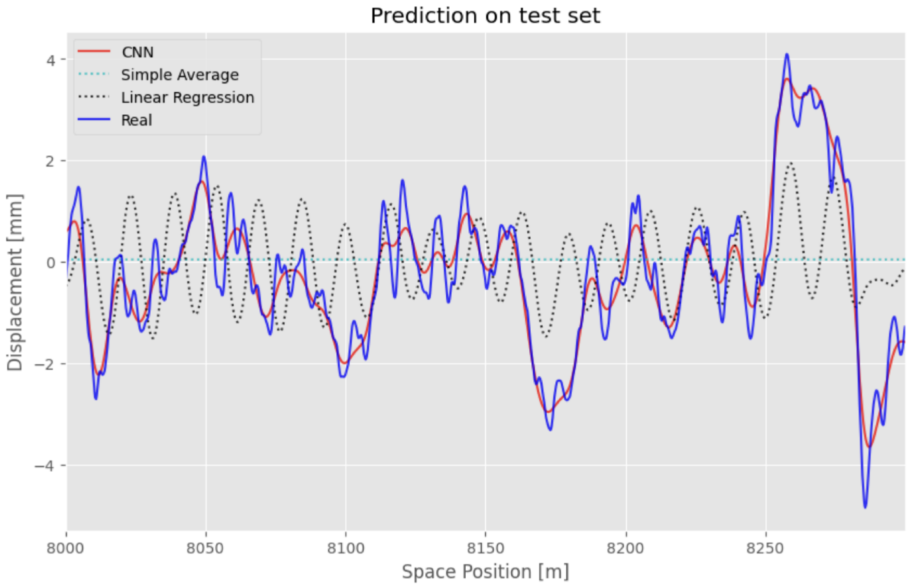

Figure D.1: CNN prediction in underdamping condition, with damping ratio 0.025.

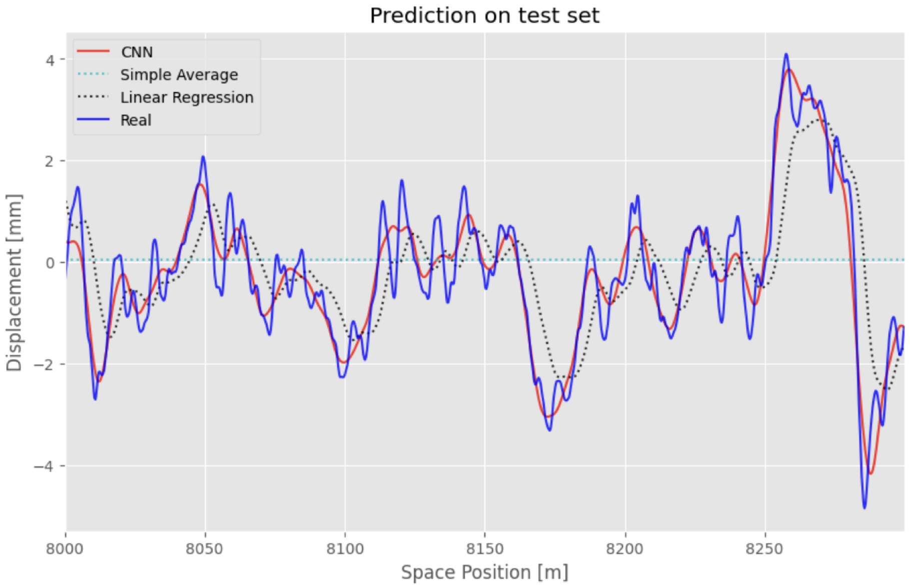

Figure D.2: CNN prediction in critical damping condition, with damping ratio 1.0.

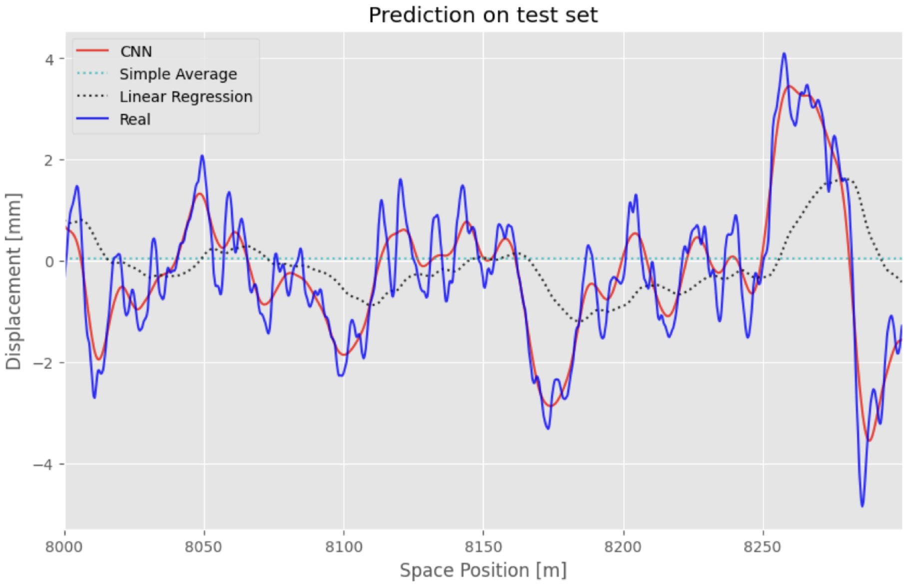

Figure D.3: CNN prediction in overdamping condition, with damping ratio 5.0.

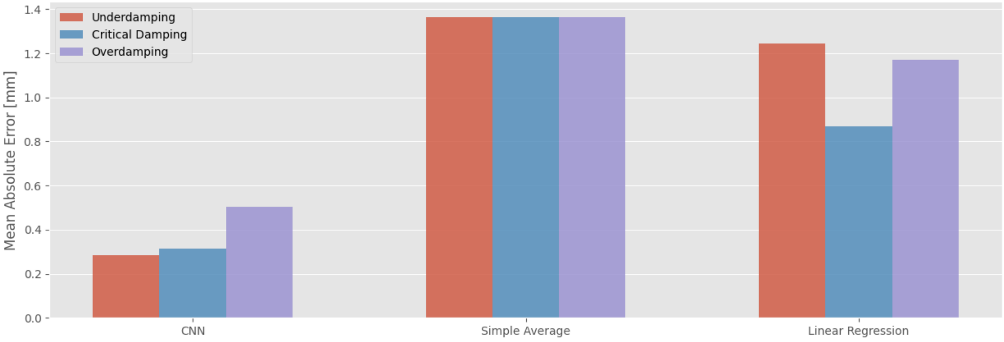

Figure D.4: Comparison of CNN's performance with baseline models (simple average and linear regression) on the MSD system.

D.3 PINNs Application on Vehicle Dynamics

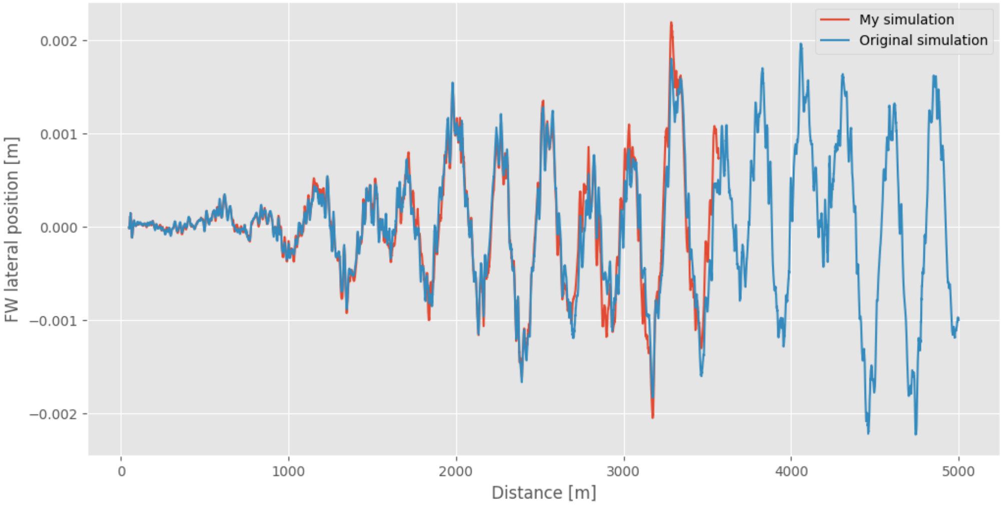

Figure D.5: The consistency of front wheelset's lateral position.

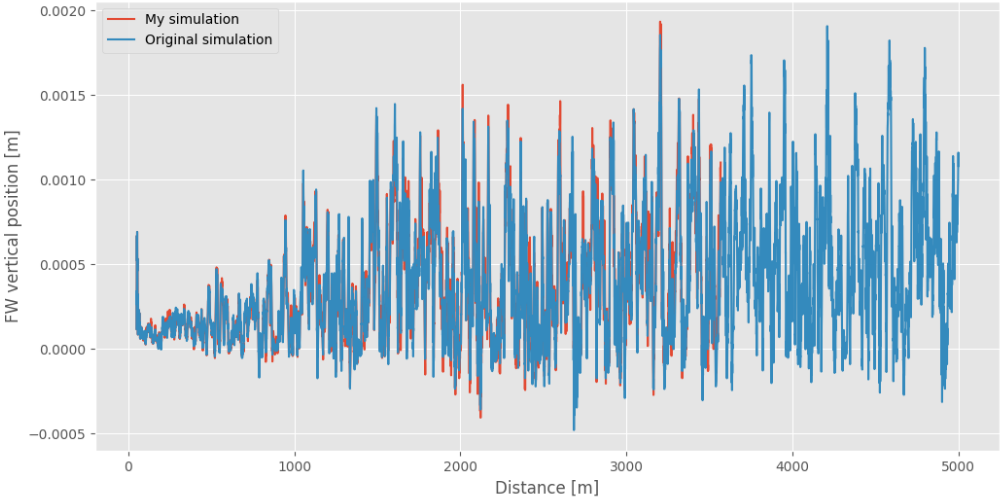

Figure D.6: The consistency of front wheelset's vertical position.

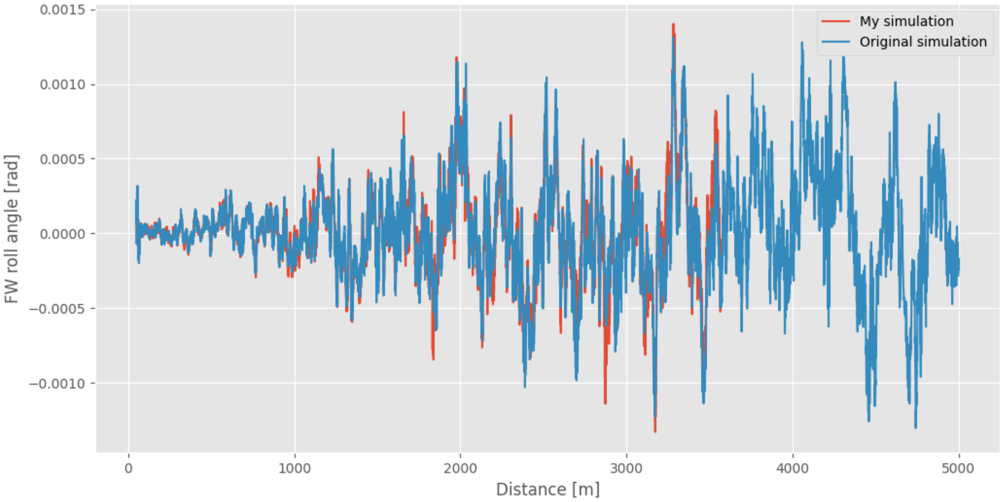

Figure D.7: The consistency of front wheelset's roll angle.

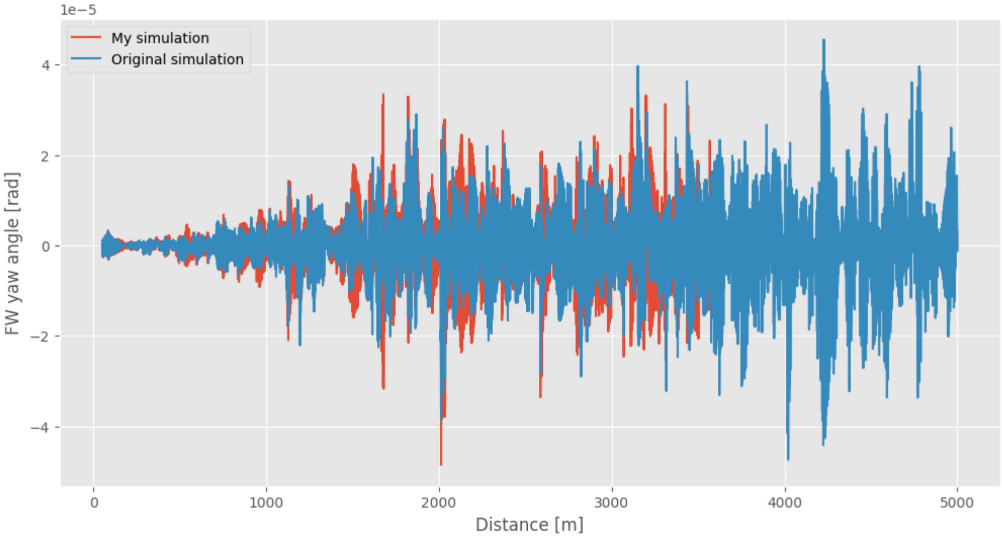

Figure D.8: The consistency of front wheelset's yaw angle.

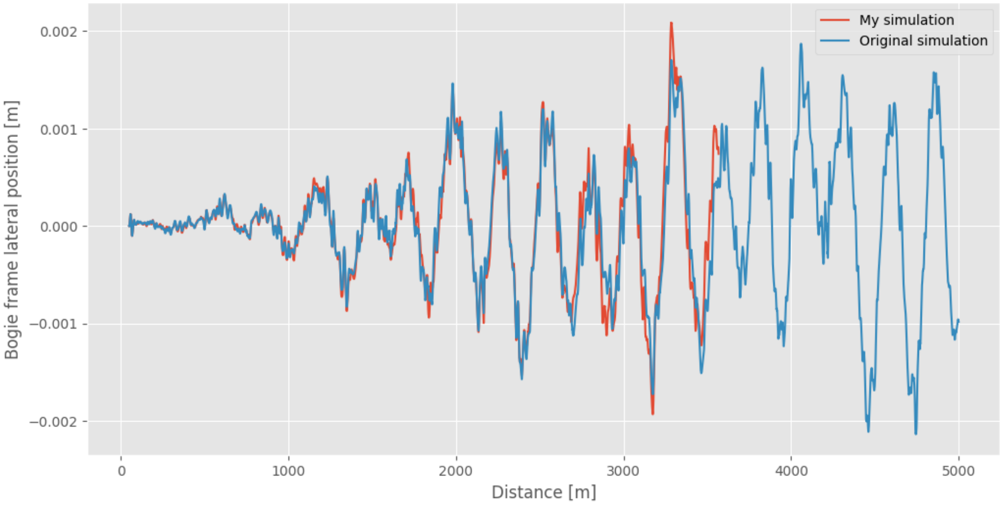

Figure D.9: The consistency of bogie frame's lateral position.

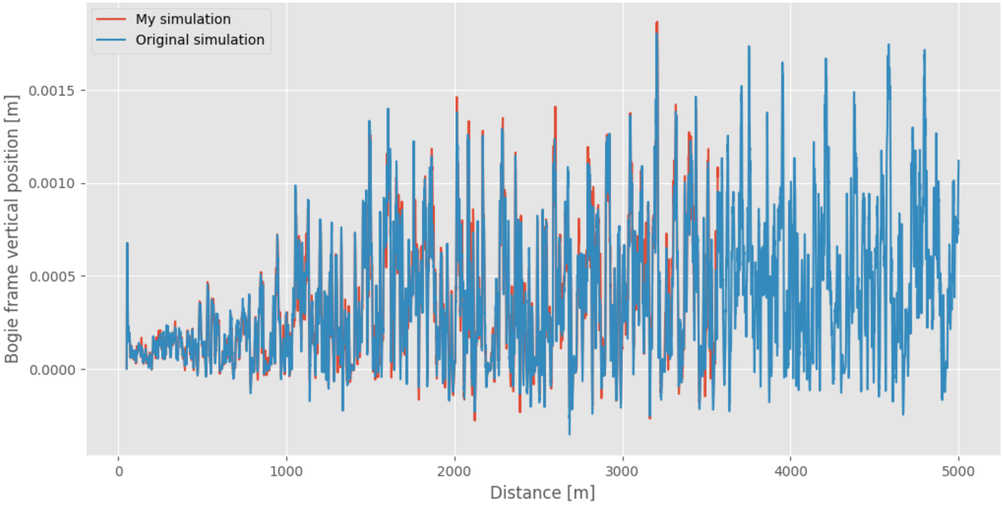

Figure D.10: The consistency of bogie frame's vertical position.

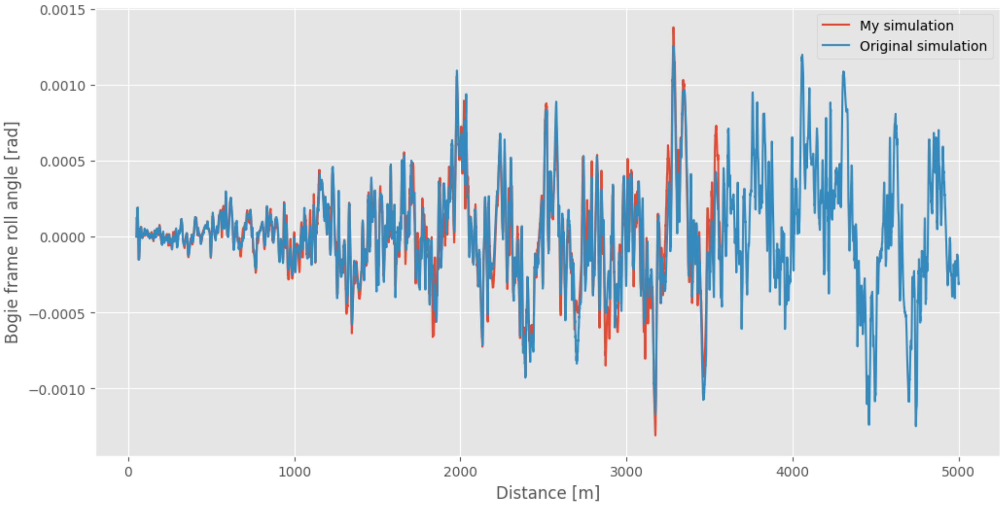

Figure D.11: The consistency of bogie frame's roll angle.

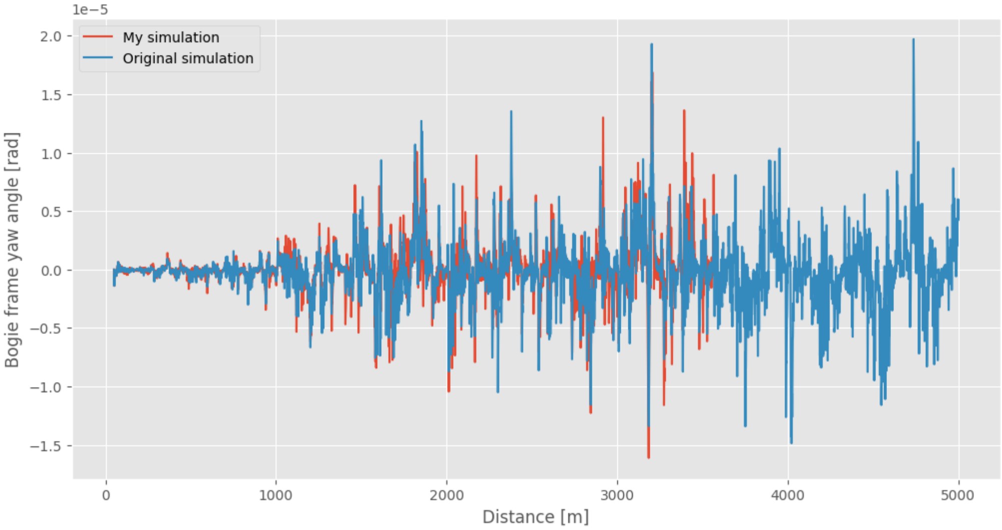

Figure D.12: The consistency of bogie frame's yaw angle.

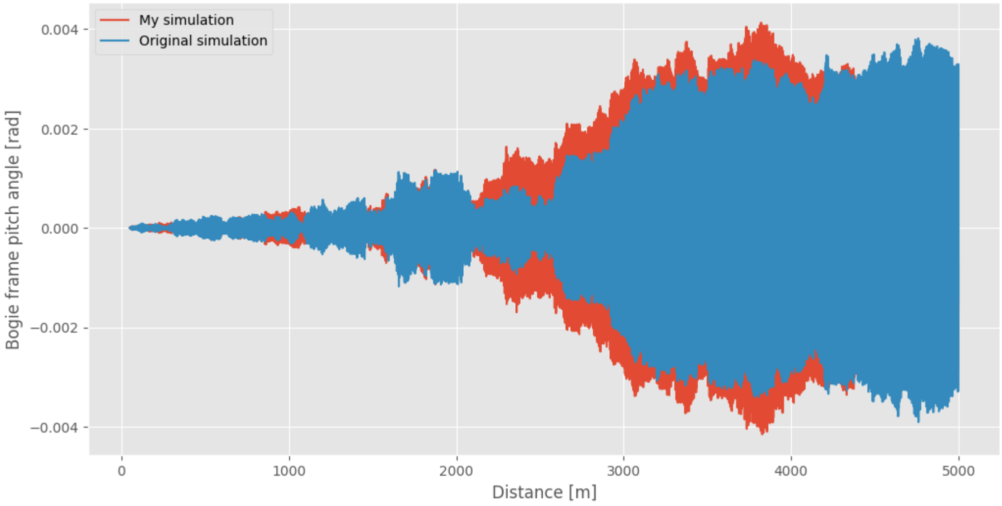

Figure D.13: The consistency of bogie frame's pitch angle.

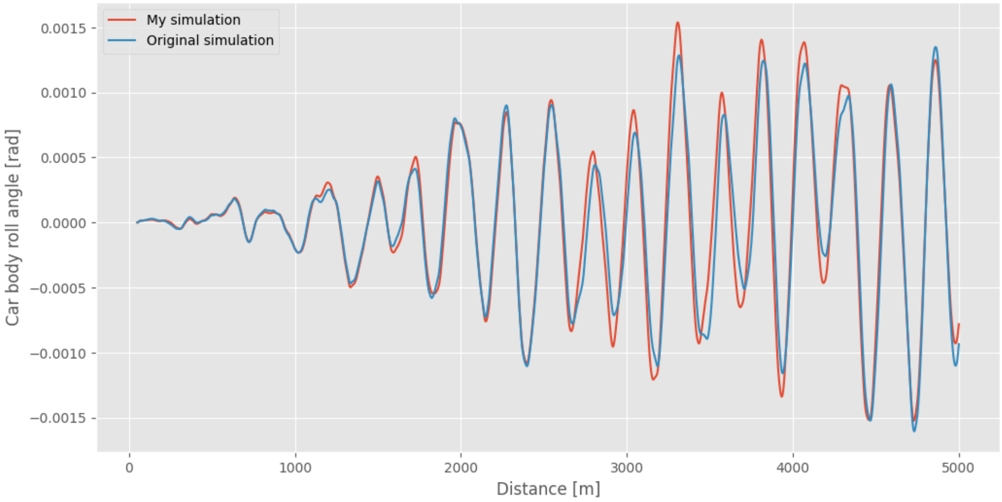

Figure D.14: The consistency of car body's roll angle.

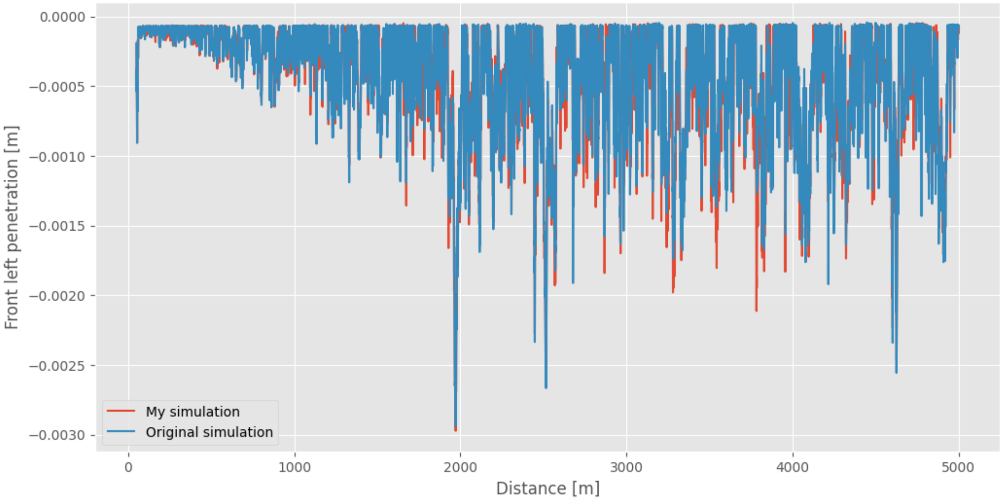

Figure D.15: The consistency of left penetration depth on front wheelset.

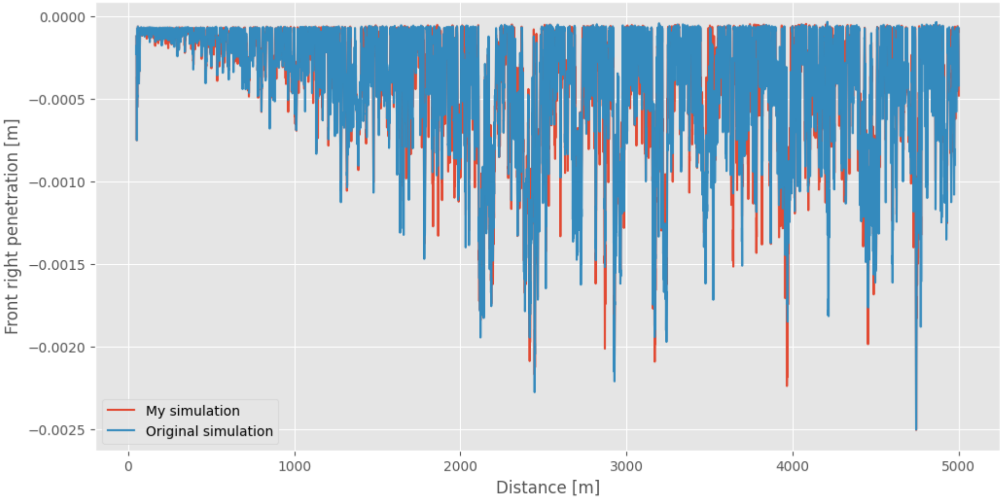

Figure D.16: The consistency of right penetration depth on front wheelset.

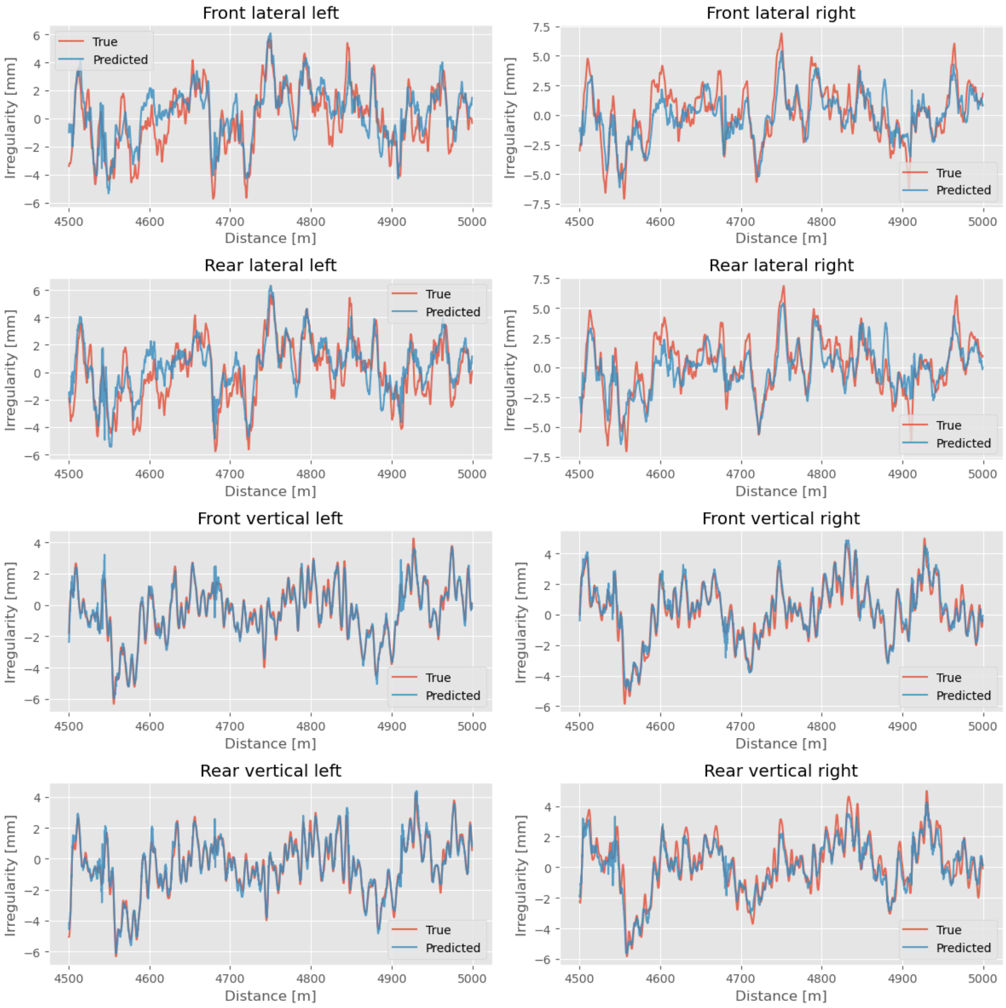

Figure D.17: Rail irregularities Prediction by a 3-layer CNN.

APPENDIX E

Important Code

E.1 cnn.py

```python
import numpy as np
import pandas as pd
from tqdm import tqdm
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
# from p2p_acc_calc_tensor import get_acc_tensor
from convolutional_neural_networks.p2p_acc_calc_tensor import get_acc_tensor

class CNNModel(nn.Module):
    def __init__(self, input_dim, output_dim,
                 conv_kernel_numbers, conv_kernel_sizes, device):
        super(CNNModel, self).__init__()
        assert len(conv_kernel_numbers) == len(conv_kernel_sizes), "The length of conv_kernel_numbers not equals to the length of conv_kernel_sizes."
        num_layers = len(conv_kernel_numbers)
        assert conv_kernel_numbers[-1] == output_dim, "The last element of conv_kernel_numbers should be equal to output_dim."
        for kernel_size in conv_kernel_sizes:
            assert kernel_size % 2 == 1, "All kernel sizes should be odd numbers."

        layers = []
        for i in range(num_layers-1):
            layers.append(nn.Conv1d(in_channels=input_dim if i == 0 else conv_kernel_numbers[i-1], out_channels=conv_kernel_numbers[i],
                                    kernel_size=conv_kernel_sizes[i], stride=1, padding=0, bias=True))
            layers.append(nn.ELU())

```

```
        self.conv_layers = nn.Sequential(*layers)
        self.output_layer =
            nn.Conv1d(in_channels=conv_kernel_numbers[-2] if
            num_layers > 1 else input_dim,
                                      out_channels=output_dim,
                                          kernel_size=conv_kernel_sizes[-1],
                                          stride=1, padding=0,
                                          bias=True)

        self.alignment_index = np.sum(np.array(conv_kernel_sizes)
            // 2)
        self.device = device
        self.to(device)
        self.criterion_train = nn.MSELoss()
        self.criterion_eval = nn.L1Loss()

    def forward(self, x):
        x = self.conv_layers(x)
        x = self.output_layer(x)
        return x

    def train_model(self, train_feature_seg_tensor,
       train_label_seg_tensor,
                     val_feature_seg_tensor, val_label_seg_tensor,
                     optimizer, scheduler, epochs,
                     rsgeo_table_tensor, velocity, time_step,
                     loss_mode, data_weight,
                         physics_weight_lateral,
                         physics_weight_vertical):
        # (batch_size, sequence_length, channels) -> (batch_size,
            channels, sequence_length)
        train_feature = train_feature_seg_tensor.permute(0, 2, 1)
        train_label = train_label_seg_tensor.permute(0, 2, 1)[:,
            :, self.alignment_index:-self.alignment_index]
        val_feature = val_feature_seg_tensor.permute(0, 2, 1)
        val_label = val_label_seg_tensor.permute(0, 2, 1)[:, :,
            self.alignment_index:-self.alignment_index]

        train_acceleration_seg_tensor =
            get_acc_tensor(train_label_seg_tensor,
            train_feature_seg_tensor, rsgeo_table_tensor,
            velocity, time_step, self.device)
        val_acceleration_seg_tensor =
            get_acc_tensor(val_label_seg_tensor,
            val_feature_seg_tensor, rsgeo_table_tensor, velocity,
            time_step, self.device)

        train_losses = []
        val_losses = []
        val_losses_lateral = []
        val_losses_vertical = []
        for epoch in range(epochs):
            self.train()
            optimizer.zero_grad()
```

```
            output = self(train_feature)

            if loss_mode == 'data':
                data_loss = self.criterion_train(output,
                    train_label)
                loss = data_loss
            elif loss_mode == 'lateral':
                acc_calculated = get_acc_tensor(output.permute(0,
                    2, 1), train_feature_seg_tensor[:,
                    self.alignment_index:-self.alignment_index,
                    :], rsgeo_table_tensor, velocity, time_step,
                    self.device)
                physics_loss_lateral =
                    self.criterion_train(acc_calculated[:, :, [0,
                    2]], train_acceleration_seg_tensor[:,
                    self.alignment_index:-self.alignment_index,
                    [0, 2]])
                loss = physics_loss_lateral
            elif loss_mode == 'vertical':
                acc_calculated = get_acc_tensor(output.permute(0,
                    2, 1), train_feature_seg_tensor[:,
                    self.alignment_index:-self.alignment_index,
                    :], rsgeo_table_tensor, velocity, time_step,
                    self.device)
                physics_loss_vertical =
                    self.criterion_train(acc_calculated[:, :, [1,
                    3]], train_acceleration_seg_tensor[:,
                    self.alignment_index:-self.alignment_index,
                    [1, 3]])
                loss = physics_loss_vertical
            elif loss_mode == 'pinn' or loss_mode == 'pinn_auto':
                data_loss = self.criterion_train(output,
                    train_label)
                acc_calculated = get_acc_tensor(output.permute(0,
                    2, 1), train_feature_seg_tensor[:,
                    self.alignment_index:-self.alignment_index,
                    :], rsgeo_table_tensor, velocity, time_step,
                    self.device)
                physics_loss_lateral =
                    self.criterion_train(acc_calculated[:, :, [0,
                    2]], train_acceleration_seg_tensor[:,
                    self.alignment_index:-self.alignment_index,
                    [0, 2]])
                physics_loss_vertical =
                    self.criterion_train(acc_calculated[:, :, [1,
                    3]], train_acceleration_seg_tensor[:,
                    self.alignment_index:-self.alignment_index,
                    [1, 3]])
                loss = data_weight * data_loss +
                    physics_weight_lateral * physics_loss_lateral
                    + physics_weight_vertical *
                    physics_loss_vertical

            loss.backward()
```

```
            optimizer.step()
            # scheduler.step()

            # Validation
            self.eval()
            with torch.no_grad():
                # Calculate data loss for training set
                output = self(train_feature)
                data_loss_train = self.criterion_eval(output, train_label)
                train_losses.append(data_loss_train.item())

                # Calculate data loss for validation set
                output = self(val_feature)
                data_loss_val = self.criterion_eval(output, val_label)
                val_losses.append(data_loss_val.item())

                # Calculate physics loss for validation set
                acc_calculated = get_acc_tensor(output.permute(0, 2, 1), val_feature_seg_tensor[:, self.alignment_index:-self.alignment_index, :], rsgeo_table_tensor, velocity, time_step, self.device)
                physics_loss_lateral = self.criterion_eval(acc_calculated[:, :, [0, 2]], val_acceleration_seg_tensor[:, self.alignment_index:-self.alignment_index, [0, 2]])
                val_losses_lateral.append(physics_loss_lateral.item())
                physics_loss_vertical = self.criterion_eval(acc_calculated[:, :, [1, 3]], val_acceleration_seg_tensor[:, self.alignment_index:-self.alignment_index, [1, 3]])
                val_losses_vertical.append(physics_loss_vertical.item())

                # Adjust physics weight
                if loss_mode == 'pinn_auto':
                    physics_weight_lateral = data_loss_val / physics_loss_lateral
                    physics_weight_vertical = data_loss_val / physics_loss_vertical

                print(f'Epoch {epoch+1}/{epochs}, Training set data MSE: {loss}, Training set data MAE: {train_losses[-1]}, \
                      Validation set data MAE: {val_losses[-1]}, Physics lateral MAE: {val_losses_lateral[-1]}, Physics vertical MAE: {val_losses_vertical[-1]}')

        return train_losses, val_losses, val_losses_lateral, val_losses_vertical
```

Listing E.1: cnn.py

E.2 p2p_acc_calc.py

```python
import numpy as np
import pandas as pd
import math

NOV = 31    # Number of variables
NOC = 13    # Number of constants (in rsgeo_table)
NOPO = 3401 # Number of points in rsgeo datafile
epsilon = 1e-9

k1, k2, k3, k4, k5, k6 = 1823000., 3646000., 3646000., 182300., 333300., 2710000.
D1, D2, D6 = 20000., 29200., 500e3
d1, d2 = 0.620, 0.680
b = 1.074
h1, h2, h3 = 0.0762, 0.6584, 0.8654
d1d1 = d1 * d1
d2d2 = d2 * d2
Iwy, Iwx, Iwz = 80., 678., 678.
Ifz, Ifx, Ify = 6780., 6780., 6780.
Icx = 2.80e5
mw, mf, mrc = 1022., 2918.9, 44388.
mx = 0.25 * mrc + 0.5 * mf + mw
g = 9.82

min   = -0.01700 # Lower bound of the irregularity
max   = 0.01700  # Upper bound of the irregularity
step  = NOPO - 1
delta = (max - min) / step # Interval of the irregularity

r0 = 0.4248828 # nominal rotational radius of wheel
z0 = 0.424827690183942 # Initial vertical position of wheel
y0d = 9.999748926159402e-05 # Initial lateral displacement of wheel
Nz_static = 133343.0 # Static normal force
# Nz_static = 0.0 # Static normal force

G  = 2.1e11 / (2 * (1 - 0.27)) # Shear Modulus
nu = 0.15 # Coefficient of friction

def update_state(y, data, velocity, time_step):
        dy = np.zeros(NOV + 1)
        dy[1] = y[2]
        dy[2] = (data[0, 5] + data[1, 5] + data[0, 2] + data[1, 2] -
                2 * k1 * (y[1] - y[9] - b * y[11] - h1 * y[13])) / mw
```

```
        dy[3] = y[4]
        dy[4] = (data[1, 0] * (data[1, 4] + (data[1, 5] + data[1, 2]) * y[3]) -
                data[0, 0] * (data[0, 4] + (data[0, 5] + data[0, 2]) * y[3]) -
                2 * k2 * d1d1 * (y[3] - y[11])) / Iwz
        dy[5] = y[6]
        dy[6] = (data[2, 5] + data[3, 5] + data[2, 2] + data[3, 2] -
                2 * k1 * (y[5] - y[9] + b * y[11] - h1 * y[13])) / mw
        dy[7] = y[8]
        dy[8] = (data[3, 0] * (data[3, 4] + (data[3, 5] + data[3, 2]) * y[7]) -
                data[2, 0] * (data[2, 4] + (data[2, 5] + data[2, 2]) * y[7]) -
                2 * k2 * d1d1 * (y[7] - y[11])) / Iwz
        dy[9] = y[10]
        dy[10] = (2 * k1 * (y[1] + y[5] - 2 * y[9] - 2 * h1 * y[13]) + 2 * k4 *
                (h2 * y[13] + h3 * y[15] - y[9]) +
                2 * D2 * (h2 * y[14] + h3 * y[16] - y[10])) / mf
        dy[11] = y[12]
        dy[12] = (2 * d1d1 * k2 * (y[3] + y[7] - 2 * y[11]) - k6 * y[11] - D6 * y[12] +
                2 * b * k1 * (y[1] - y[5] - 2 * b * y[11])) / Ifz
        dy[13] = y[14]
        dy[14] = (2 * k1 * h1 * (y[1] + y[5] - 2 * y[9] - 2 * h1 * y[13]) +
                2 * k4 * h2 * (y[9] - h2 * y[13] - h3 * y[15]) + 2 * D2 * h2 *
                (y[10] - h2 * y[14] - h3 * y[16]) -
                2 * d2d2 * (k5 * (y[13] - y[15]) + D1 * (y[14] - y[16])) -
                2 * d1d1 * k3 * (2 * y[13] - y[21] - y[23])) / Ifx
        dy[15] = y[16]
        dy[16] = (-2 * d2d2 * (k5 * (y[15] - y[13]) + D1 * (y[16] - y[14]))) / Icx
        dy[17] = y[18]
        dy[18] = (data[0, 6] + data[1, 6] + data[0, 3] + data[1, 3] - Nz_static +
                2 * k3 * (y[25] - y[17])) / mw
        dy[19] = y[20]
        dy[20] = (data[2, 6] + data[3, 6] + data[2, 3] + data[3, 3] - Nz_static +
                2 * k3 * (y[25] - y[19])) / mw
        dy[21] = y[22]
        dy[22] = (-data[1, 0] *
                (data[1, 6] + data[1, 3] - (data[1, 5] + data[1, 2]) * y[21]) +
                data[0, 0] *
                (data[0, 6] + data[0, 3] - (data[0, 5] + data[0, 2]) * y[21]) -
                2 * k3 * d1d1 * (y[21] - y[13])) / Iwx
```

```python
        dy[23] = y[24]
        dy[24] = (-data[3, 0] *
                (data[3, 6] + data[3, 3] - (data[3, 5] + data[3, 2]) * y[23]) +
                data[2, 0] *
                (data[2, 6] + data[2, 3] - (data[2, 5] + data[2, 2]) * y[23]) -
                2 * k3 * d1d1 * (y[23] - y[13])) / Iwx
        dy[25] = y[26]
        dy[26] = (-2 * k3 * (2 * y[25] - y[17] - y[19]) - 2 * k5 * y[25] - 2 * D1 *
                y[26]) / mf
        dy[27] = y[28]
        dy[28] = (-2 * b * k3 * (2 * b * y[27] + y[17] - y[19])) / Ify
        dy[29] = (-data[1, 1] * (data[1, 4] + (data[1, 5] + data[1, 2]) * y[3]) +
                -data[0, 1] * (data[0, 4] + (data[0, 5] + data[0, 2]) * y[3]) -
                2 * d1d1 * k3 * y[3] * y[13]) / Iwy
        dy[30] = (-data[3, 1] * (data[3, 4] + (data[3, 5] + data[3, 2]) * y[7]) +
                -data[2, 1] * (data[2, 4] + (data[2, 5] + data[2, 2]) * y[7]) -
                2 * d1d1 * k3 * y[7] * y[13]) / Iwy
        dy[31] = velocity

        return y + 0.5 * dy * time_step

def get_acc(irregularity, y, rsgeo_table, velocity, time_step):
    Omega = r0 / velocity

    # Get irregularity
    # irreg = np.zeros((4, 2))
    # irreg[0, 0] = irregularity[0] # fw left lateral
    # irreg[1, 0] = irregularity[1] # fw right lateral
    # irreg[2, 0] = irregularity[2] # rw right lateral
    # irreg[3, 0] = irregularity[3] # rw right lateral
    # irreg[0, 1] = irregularity[4] # fw left vertical
    # irreg[1, 1] = irregularity[5] # fw right vertical
    # irreg[2, 1] = irregularity[6] # rw left vertical
    # irreg[3, 1] = irregularity[7] # rw right vertical
    irreg = np.array(irregularity).reshape(2,4).T

    # Interpolate RSGEO
    point = np.zeros((4, NOC)) # Contact point states
    # point[0, 0] = y[1] - irreg[0, 0]        # F l lateral displacement
    # point[1, 0] = -(y[1] - irreg[1, 0])     # F r lateral displacement
    # point[2, 0] = y[5] - irreg[2, 0]        # R l lateral displacement
    # point[3, 0] = -(y[5] - irreg[3, 0])     # R r lateral displacement
```

```
    point[:, 0] = [y[1] - irreg[0, 0], -(y[1] - irreg[1, 0]), y[5]
        - irreg[2, 0], -(y[5] - irreg[3, 0])]

    # for i in range(4):
    #     tmp = np.clip(point[i, 0], -0.01699, 0.01699)
    #     # point[i, 0] = tmp
    #     index = int(np.floor((tmp - min) / delta))
    #     frac  = (tmp - rsgeo_table[index, 0]) / delta
    #     for j in range(1, NOC):
    #         point[i, j] = rsgeo_table[index, j] * (1 - frac) +
        rsgeo_table[index + 1, j] * frac
    tmp = np.clip(point[:, 0], -0.01699, 0.01699)
    index = np.floor((tmp - min) / delta).astype(int)
    frac = (tmp - rsgeo_table[index, 0]) / delta
    for j in range(1, NOC):
        point[:, j] = rsgeo_table[index, j] * (1 - frac) +
            rsgeo_table[index + 1, j] * frac

    # Update penetration
    dp = np.zeros(4) # Penetration
    for i in [0, 2]:
        # left wheels:
        dp[i] = ((point[i, 12] + irreg[i, 0] - y[1 + 2 * i] -
            point[i, 5] - y[21 + i] *
                point[i, 6] - y0d) * np.sin(point[i, 2] + y[21 +
                    i]) +
                np.cos(point[i, 2] + y[21 + i]) *
                (point[i, 10] - z0 - y[17 + i] - y[21 + i] *
                    point[i, 5] +
                point[i, 6] + irreg[i, 1]))
        # right wheels:
        dp[i + 1] = ((point[i + 1, 12] - irreg[i + 1, 0] + y[1 + 2
            * i] - point[i + 1, 5] +
                y[21 + i] * point[i + 1, 6] - y0d) *
                np.sin(point[i + 1, 2] - y[21 + i]) +
                np.cos(point[i + 1, 2] - y[21 + i]) *
                (point[i + 1, 10] - z0 - y[17 + i] + y[21 + i] *
                    point[i + 1, 5] +
                point[i + 1, 6] + irreg[i + 1, 1]))
        # print(dp)

    # Adjust normal forces
#     for i in range(4):
#         if point[i, 11] == 0 or point[i, 1] == 0:
#                 print('Division by zero')
    #     if dp[i] > -point[i, 11]:
    #         N = point[i, 1] * math.pow(1 + dp[i] / point[i, 11],
        1.5)
    #     else:
    #         N = epsilon
    #     N3 = math.pow(N / point[i, 1], 1. / 3.) # Adjustment
        factor for semi axes

    #     point[i, 3] *= N3
```

```
    #     point[i, 4] *= N3
    #     point[i, 1] = N
#     print('A:', point[:, 11])
#     print('B:', 1 + dp / point[:, 11])
    N = point[:, 1] * np.power(np.maximum(1 +  dp / point[:, 11], 0), 1.5)
#     print('N:', N)
    N3 = np.power(N / point[:, 1], 1/3)
    point[:, 3] = point[:, 3] * N3
    point[:, 4] = point[:, 4] * N3
    point[:, 1] = N

    # for i in [0, 2]:
    #     if point[i, 1] == epsilon and point[i + 1, 1] != epsilon:
    #         point[i + 1, 3] *= math.pow(epsilon / point[i, 1], 1. / 3.)
    #         point[i + 1, 4] *= math.pow(epsilon / point[i, 1], 1. / 3.)
    #         point[i + 1, 1] = epsilon
    #     elif point[i + 1, 1] == epsilon and point[i, 1] != epsilon:
    #         point[i, 3] *= math.pow(epsilon / point[i + 1, 1], 1. / 3.)
    #         point[i, 4] *= math.pow(epsilon / point[i + 1, 1], 1. / 3.)
    #         point[i, 1] = epsilon

    # Calculate creepages on longitudinal, lateral, and yaw directions
    creep = np.zeros((4, 3))
    for i in [0, 2]:
        tsin  = np.sin(point[i, 2])
        tsin2 = np.sin(point[i + 1, 2])
        tcos  = np.cos(point[i, 2])
        tcos2 = np.cos(point[i + 1, 2])
        y21i  = y[21 + i]
        y22i  = y[22 + i]
        y29i2 = y[29 + i // 2]
        y2p2i = y[2 + 2 * i]
        y3p2i = y[3 + 2 * i]

        creep[i, 0] = ((velocity - point[i, 6] * (Omega + y29i2 - y3p2i * y22i) -
                        y[4 + 2 * i] * point[i, 5] + y3p2i * y2p2i) / velocity) # chi_xL longitutinal creepage of left wheel
        creep[i + 1, 0] = ((velocity - point[i + 1, 6] * (Omega + y29i2 - y3p2i * y22i) +
                            y[4 + 2 * i] * point[i + 1, 5] + y3p2i * y2p2i) / velocity) # chi_yL longitutinal creepage of right wheel

```

```
        creep[i, 1] = (((y2p2i - y3p2i * velocity + y21i * y[18 + i] + point[i, 6] * y22i) * tcos +
                       (y[18 + i] - y2p2i * y21i + point[i, 5] * y22i) * tsin) / velocity) # chi_yL lateral creepage of left wheel
        creep[i + 1, 1] = (((y2p2i - y3p2i * velocity + y21i * y[18 + i] + point[i + 1, 6] * y22i) * tcos2 -
                           (y[18 + i] - y2p2i * y21i - point[i + 1, 5] * y22i) * tsin2) / velocity) # chi_yR lateral creepage of right wheel

        creep[i, 2] = ((-(Omega + y29i2 - y3p2i * y22i) * tsin +
                       y[4 + 2 * i] * tcos) / velocity) # chi_spL spin creepage of left wheel
        creep[i + 1, 2] = (((Omega + y29i2 - y3p2i * y22i) * tsin2 +
                           y[4 + 2 * i] * tcos2) / velocity) # chi_spR spin creepage of right wheel

    # Calculate frictions based on Shen-Hedrick-Elkins (SHE) model
    fx = np.zeros(4) # Longitudinal friction
    fy = np.zeros(4) # Lateral friction
    for i in range(4):
        fx[i] = -point[i, 3] * point[i, 4] * G * point[i, 7] * creep[i, 0]
        fy[i] = -point[i, 3] * point[i, 4] * G * (point[i, 8] * creep[i, 1] +
                                              np.sqrt(point[i, 3] * point[i, 4]) *
                                              point[i, 9] * creep[i, 2])

        fnorm = np.sqrt(fx[i] * fx[i] + fy[i] * fy[i]) / (nu * point[i, 1]) # Normalized friction

        if fnorm < 3:
            factor = 1 - fnorm / 3. + fnorm * fnorm / 27.
            fx[i] *= factor
            fy[i] *= factor
        else:
            fx[i] *= 1 / fnorm
            fy[i] *= 1 / fnorm

    # Transfter from point[] to data[]
    data = np.zeros((4, 8))
    data[:4, 0] = point[:4, 5]  # K_wy
    data[:4, 1] = point[:4, 6]  # K_wz
    data[:4, 7] = point[:4, 10] # K_rz

    for i in [0, 2]:
        phi = y[21 + i]
        if point[i, 1] > 0:
            tsin       = np.sin(point[i, 2] + phi)
```

```
            tcos       = np.cos(point[i, 2] + phi)
            data[i, 2] = -point[i, 1] * tsin # N_y
            data[i, 3] = point[i, 1] * tcos  # N_z
            data[i, 4] = fx[i]                # F_wx
            data[i, 5] = fy[i] * tcos         # F_y
            data[i, 6] = fy[i] * tsin         # F_z
        else:
            for j in range(2, 7):
                data[i, j] = 0

        if point[i + 1, 1] > 0:
            tsin           = np.sin(point[i + 1, 2] - phi)
            tcos           = np.cos(point[i + 1, 2] - phi)
            data[i + 1, 2] = point[i + 1, 1] * tsin # N_y
            data[i + 1, 3] = point[i + 1, 1] * tcos # N_z
            data[i + 1, 4] = fx[i + 1]              # F_wx
            data[i + 1, 5] = fy[i + 1] * tcos       # F_y
            data[i + 1, 6] = -fy[i + 1] * tsin      # F_z
        else:
            for j in range(2, 7):
                data[i + 1, j] = 0

    # y = update_state(y, data, velocity, time_step)


    # Iterated calculation of accelerations
#     for _ in range(2):
#         N = point[:, 1] * np.power(np.maximum(1 +  dp / point[:, 11], 0), 1.5)
#         N3 = np.power(N / point[:, 1], 1/3)
#         point[:, 3] = point[:, 3] * N3
#         point[:, 4] = point[:, 4] * N3
#         point[:, 1] = N

#         for i in range(4):
#             fx[i] = -point[i, 3] * point[i, 4] * G * point[i, 7] * creep[i, 0]
#             fy[i] = -point[i, 3] * point[i, 4] * G * (point[i, 8] * creep[i, 1] +
#                                                  np.sqrt(point[i, 3] * point[i, 4]) *
#                                                  point[i, 9] * creep[i, 2])

#             fnorm = np.sqrt(fx[i] * fx[i] + fy[i] * fy[i]) / (nu * point[i, 1]) # Normalized friction

#             if fnorm < 3:
#                 factor = 1 - fnorm / 3. + fnorm * fnorm / 27.
#                 fx[i] *= factor
#                 fy[i] *= factor
#             else:
#                 fx[i] *= 1 / fnorm
#                 fy[i] *= 1 / fnorm
```

```

#         for i in [0, 2]:
#             phi = y[21 + i]
#             if point[i, 1] > 0:
#                 tsin       = np.sin(point[i, 2] + phi)
#                 tcos       = np.cos(point[i, 2] + phi)
#                 data[i, 2] = -point[i, 1] * tsin # N_y
#                 data[i, 3] = point[i, 1] * tcos  # N_z
#                 data[i, 4] = fx[i]               # F_wx
#                 data[i, 5] = fy[i] * tcos        # F_y
#                 data[i, 6] = fy[i] * tsin        # F_z
#             else:
#                 for j in range(2, 7):
#                     data[i, j] = 0

#             if point[i + 1, 1] > 0:
#                 tsin           = np.sin(point[i + 1, 2] - phi)
#                 tcos           = np.cos(point[i + 1, 2] - phi)
#                 data[i + 1, 2] = point[i + 1, 1] * tsin # N_y
#                 data[i + 1, 3] = point[i + 1, 1] * tcos # N_z
#                 data[i + 1, 4] = fx[i + 1]              # F_wx
#                 data[i + 1, 5] = fy[i + 1] * tcos       # F_y
#                 data[i + 1, 6] = -fy[i + 1] * tsin      # F_z
#             else:
#                 for j in range(2, 7):
#                     data[i + 1, j] = 0


    bound_lateral = 50
    bound_vertical = 200
    acc_FW_lateral = np.clip((data[0, 5] + data[1, 5] + data[0, 2]
        + data[1, 2] -
                2 * k1 * (y[1] - y[9] - b * y[11] - h1 * y[13])) /
                    mw, -bound_lateral, bound_lateral)
    acc_FW_vertical = np.clip((data[0, 6] + data[1, 6] + data[0,
        3] + data[1, 3] - Nz_static +
                2 * k3 * (y[25] - y[17])) / mw, -bound_vertical,
                    bound_vertical)
    acc_RW_lateral = np.clip((data[2, 5] + data[3, 5] + data[2, 2]
        + data[3, 2] -
                2 * k1 * (y[5] - y[9] + b * y[11] - h1 * y[13])) /
                    mw, -bound_lateral, bound_lateral)
    acc_RW_vertical = np.clip((data[2, 6] + data[3, 6] + data[2,
        3] + data[3, 3] - Nz_static +
                2 * k3 * (y[25] - y[19])) / mw, -bound_vertical,
                    bound_vertical)
    # acc_B_lateral = (2 * k1 * (y[1] + y[5] - 2 * y[9] - 2 * h1 *
        y[13]) + 2 * k4 *
    #             (h2 * y[13] + h3 * y[15] - y[9]) +
    #             2 * D2 * (h2 * y[14] + h3 * y[16] - y[10])) / mf
    # acc_B_vertical = (-2 * k3 * (2 * y[25] - y[17] - y[19]) - 2
        * k5 * y[25] - 2 * D1 *
    #             y[26]) / mf
```

```
    # acc_C_roll = (-2 * d2d2 * (k5 * (y[15] - y[13]) + D1 *
        (y[16] - y[14]))) / Icx


    acc = np.array([acc_FW_lateral, acc_FW_vertical,
        acc_RW_lateral, acc_RW_vertical])

    return acc
```

Listing E.2: p2p_acc_calc.py

E.3 p2p_acc_calc_tensor.py

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from tqdm import tqdm
import math
import torch

NOV = 31     # Number of variables
NOC = 13     # Number of constants (in rsgeo_table)
NOPO = 3401 # Number of points in rsgeo datafile
epsilon = 1e-9

k1, k2, k3, k4, k5, k6 = 1823000., 3646000., 3646000., 182300.,
    333300., 2710000.
D1, D2, D6 = 20000., 29200., 500e3
d1, d2 = 0.620, 0.680
b = 1.074
h1, h2, h3 = 0.0762, 0.6584, 0.8654
d1d1 = d1 * d1
d2d2 = d2 * d2
Iwy, Iwx, Iwz = 80., 678., 678.
Ifz, Ifx, Ify = 6780., 6780., 6780.
Icx = 2.80e5
mw, mf, mrc = 1022., 2918.9, 44388.
mx = 0.25 * mrc + 0.5 * mf + mw
g = 9.82

min   = -0.01700 # Lower bound of the irregularity
max   = 0.01700  # Upper bound of the irregularity
step  = NOPO - 1
delta = (max - min) / step # Interval of the irregularity

r0 = 0.4248828 # nominal rotational radius of wheel
z0 = 0.424827690183942 # Initial vertical position of wheel
y0d = 9.999748926159402e-05 # Initial lateral displacement of wheel
Nz_static = 133343.0 # Static normal force
# Nz_static = 0.0 # Static normal force

```

```
G  = 2.1e11 / (2 * (1 - 0.27)) # Shear Modulus
nu = 0.15 # Coefficient of friction

bound_lateral = 50
bound_vertical = 200

def get_acc_tensor(irregularity, y, rsgeo_table_tensor, velocity, time_step, device):
        Omega = r0 / velocity
        batch_size = irregularity.shape[0]
        segment_length = irregularity.shape[1]

        zeros_tensor = torch.zeros(batch_size, segment_length, 1, device=device)
        y = torch.cat((zeros_tensor, y), dim=2)

        irreg = irregularity.view(batch_size, segment_length, 2, 4).permute(0, 1, 3, 2).to(device)

        point = torch.zeros((batch_size, segment_length, 4, NOC), device=device)
        point[:, :, 0, 0] = y[:, :, 1] - irreg[:, :, 0, 0]  # F l lateral displacement
        point[:, :, 1, 0] = -(y[:, :, 1] - irreg[:, :, 1, 0])  # F r lateral displacement
        point[:, :, 2, 0] = y[:, :, 5] - irreg[:, :, 2, 0]  # R l lateral displacement
        point[:, :, 3, 0] = -(y[:, :, 5] - irreg[:, :, 3, 0])  # R r lateral displacement
        tmp = torch.clip(point[:, :, :, 0].clone(), -0.01699, 0.01699)
        index = torch.floor((tmp - min) / delta).long()
        frac = (tmp - rsgeo_table_tensor[index, 0]) / delta
        for j in range(1, NOC):
                point[:, :, :, j] = rsgeo_table_tensor[index, j] * (1 - frac) + rsgeo_table_tensor[index + 1, j] * frac

        dp = torch.zeros((batch_size, segment_length, 4), dtype=torch.float32, device=device)
        for i in [0, 2]:
                # left wheels:
                dp[:, :, i] = ((point[:, :, i, 12] + irreg[:, :, i, 0] - y[:, :, 1 + 2 * i] - point[:, :, i, 5] - y[:, :, 21 + i] *
                        point[:, :, i, 6] - y0d) * torch.sin(point[:, :, i, 2] + y[:, :, 21 + i]) +
                        torch.cos(point[:, :, i, 2] + y[:, :, 21 + i]) *
                        (point[:, :, i, 10] - z0 - y[:, :, 17 + i] - y[:, :, 21 + i] * point[:, :, i, 5] +
                        point[:, :, i, 6] + irreg[:, :, i, 1]))
                # right wheels:
```

```
                dp[:, :, i + 1] = ((point[:, :, i + 1, 12] -
                    irreg[:, :, i + 1, 0] + y[:, :, 1 + 2 * i] -
                    point[:, :, i + 1, 5] +
                        y[:, :, 21 + i] * point[:, :, i + 1, 6] -
                            y0d) *
                        torch.sin(point[:, :, i + 1, 2] - y[:, :,
                            21 + i]) +
                        torch.cos(point[:, :, i + 1, 2] - y[:, :,
                            21 + i]) *
                        (point[:, :, i + 1, 10] - z0 - y[:, :, 17
                            + i] + y[:, :, 21 + i] * point[:, :, i
                            + 1, 5] +
                        point[:, :, i + 1, 6] + irreg[:, :, i + 1,
                            1]))

        N = point[:, :, :, 1].clone() * torch.pow(torch.clamp(1 +
            dp / point[:, :, :, 11].clone(), min=epsilon), 1.5)
        N3 = torch.pow((N / point[:, :, :, 1].clone()), 1/3)
        point[:, :, :, 3] = point[:, :, :, 3].clone() * N3
        point[:, :, :, 4] = point[:, :, :, 4].clone() * N3
        point[:, :, :, 1] = N

        creep = torch.zeros((batch_size, segment_length, 4, 3),
            device=device)
        for i in [0, 2]:
                tsin = torch.sin(point[:, :, i, 2])
                tsin2 = torch.sin(point[:, :, i + 1, 2])
                tcos = torch.cos(point[:, :, i, 2])
                tcos2 = torch.cos(point[:, :, i + 1, 2])

                y21i = y[:, :, 21 + i]
                y22i = y[:, :, 22 + i]
                y29i2 = y[:, :, 29 + i // 2]
                y2p2i = y[:, :, 2 + 2 * i]
                y3p2i = y[:, :, 3 + 2 * i]

                creep[:, :, i, 0] = (velocity - point[:, :, i, 6]
                    * (Omega + y29i2 - y3p2i * y22i) -
                                        y[:, :, 4 + 2 * i] *
                                            point[:, :, i, 5] +
                                            y3p2i * y2p2i) /
                                            velocity
                creep[:, :, i + 1, 0] = (velocity - point[:, :, i
                    + 1, 6] * (Omega + y29i2 - y3p2i * y22i) +
                                        y[:, :, 4 + 2 * i] *
                                            point[:, :, i + 1, 5]
                                            + y3p2i * y2p2i) /
                                            velocity

                creep[:, :, i, 1] = ((y2p2i - y3p2i * velocity +
                    y21i * y[:, :, 18 + i] + point[:, :, i, 6] *
                    y22i) * tcos +
                                        (y[:, :, 18 + i] - y2p2i *
                                            y21i + point[:, :, i,
```

```
                                            5] * y22i) * tsin) /
                                            velocity
                creep[:, :, i + 1, 1] = ((y2p2i - y3p2i * velocity
                    + y21i * y[:, :, 18 + i] + point[:, :, i + 1,
                    6] * y22i) *
                                        tcos2 - (y[:, :, 18 + i] -
                                            y2p2i * y21i -
                                            point[:, :, i + 1, 5]
                                            * y22i) * tsin2) /
                                            velocity

                creep[:, :, i, 2] = (-(Omega + y29i2 - y3p2i *
                    y22i) * tsin + y[:, :, 4 + 2 * i] * tcos) /
                    velocity
                creep[:, :, i + 1, 2] = ((Omega + y29i2 - y3p2i *
                    y22i) * tsin2 + y[:, :, 4 + 2 * i] * tcos2) /
                    velocity
        # if torch.isnan(creep).any():
        #         print("creep contains NaN")
        # elif torch.isinf(creep).any():
        #         print("creep contains Inf")
        # else:
        #         print("creep does not contain NaN or Inf")

        fx = -point[:, :, :, 3] * point[:, :, :, 4] * G * point[:,
            :, :, 7] * creep[:, :, :, 0]
        fy = -point[:, :, :, 3] * point[:, :, :, 4] * G *
            (point[:, :, :, 8] * creep[:, :, :, 1] +
                                                torch.sqrt(point[:,
                                                    :, :, 3] *
                                                    point[:, :, :,
                                                    4]) *
                                                point[:, :, :, 9]
                                                    * creep[:, :,
                                                    :, 2])
        # if torch.isnan(fx).any():
        #         print("fx contains NaN")
        # elif torch.isinf(fx).any():
        #         print("fx contains Inf")
        # else:
        #         print("fx does not contain NaN or Inf")
        # if torch.isnan(fy).any():
        #         print("fy contains NaN")
        # elif torch.isinf(fy).any():
        #         print("fy contains Inf")
        # else:
        #         print("fy does not contain NaN or Inf")
        fnorm = torch.sqrt(fx**2 + fy**2) / (nu * point[:, :, :,
            1])
        # if torch.isnan(fnorm).any():
        #         print("fnorm contains NaN")
        # elif torch.isinf(fnorm).any():
        #         print("fnorm contains Inf")
        # else:
```

```
        #         print("fnorm does not contain NaN or Inf")
        factor = torch.where(fnorm < 3, 1 - fnorm / 3. + fnorm**2 / 27., 1 / fnorm)
        fx = fx * factor
        fy = fy * factor
        # if torch.isnan(fx).any():
        #         print("fx contains NaN")
        # elif torch.isinf(fx).any():
        #         print("fx contains Inf")
        # else:
        #         print("fx does not contain NaN or Inf")
        # if torch.isnan(fy).any():
        #         print("fy contains NaN")
        # elif torch.isinf(fy).any():
        #         print("fy contains Inf")
        # else:
        #         print("fy does not contain NaN or Inf")

        data = torch.zeros((batch_size, segment_length, 4, 8), dtype=torch.float32, device=device)
        data[:, :, :, 0] = point[:, :, :, 5]
        data[:, :, :, 1] = point[:, :, :, 6]
        data[:, :, :, 7] = point[:, :, :, 10]
        for i in [0, 2]:
                phi = y[:, :, 21 + i]

                mask = point[:, :, i, 1] > 0
                tsin = torch.sin(point[:, :, i, 2] + phi)
                tcos = torch.cos(point[:, :, i, 2] + phi)
                data[:, :, i, 2] = (-point[:, :, i, 1] * tsin[:, :])
                data[:, :, i, 3] = (point[:, :, i, 1] * tcos[:, :])
                data[:, :, i, 4] = (fx[:, :, i])
                data[:, :, i, 5] = (fy[:, :, i] * tcos[:, :])
                data[:, :, i, 6] = (fy[:, :, i] * tsin[:, :])
                # data[:, :, i, 2][mask] = (-point[:, :, i, 1] * tsin[:, :])[mask]
                # data[:, :, i, 3][mask] = (point[:, :, i, 1] * tcos[:, :])[mask]
                # data[:, :, i, 4][mask] = (fx[:, :, i])[mask]
                # data[:, :, i, 5][mask] = (fy[:, :, i] * tcos[:, :])[mask]
                # data[:, :, i, 6][mask] = (fy[:, :, i] * tsin[:, :])[mask]

                mask = point[:, :, i + 1, 1] > 0
                tsin = torch.sin(point[:, :, i + 1, 2] - phi)
                tcos = torch.cos(point[:, :, i + 1, 2] - phi)
                data[:, :, i + 1, 2] = (point[:, :, i + 1, 1] * tsin[:, :])
                data[:, :, i + 1, 3] = (point[:, :, i + 1, 1] * tcos[:, :])
                data[:, :, i + 1, 4] = (fx[:, :, i + 1])
```

```
                data[:, :, i + 1, 5] = (fy[:, :, i + 1] * tcos[:,
                    :])
                data[:, :, i + 1, 6] = (-fy[:, :, i + 1] * tsin[:,
                    :])
                # data[:, :, i + 1, 2][mask] = (point[:, :, i + 1,
                    1] * tsin[:, :])[mask]
                # data[:, :, i + 1, 3][mask] = (point[:, :, i + 1,
                    1] * tcos[:, :])[mask]
                # data[:, :, i + 1, 4][mask] = (fx[:, :, i +
                    1])[mask]
                # data[:, :, i + 1, 5][mask] = (fy[:, :, i + 1] *
                    tcos[:, :])[mask]
                # data[:, :, i + 1, 6][mask] = (-fy[:, :, i + 1] *
                    tsin[:, :])[mask]
        # if torch.isnan(data).any():
        #         print("data contains NaN")
        # elif torch.isinf(data).any():
        #         print("data contains Inf")
        # else:
        #         print("data does not contain NaN or Inf")

        acc_FW_lateral = (data[:, :, 0, 5] + data[:, :, 1, 5] +
            data[:, :, 0, 2] + data[:, :, 1, 2] -
                2 * k1 * (y[:, :, 1] - y[:, :, 9] - b * y[:, :,
                    11] - h1 * y[:, :, 13])) / mw
        acc_FW_vertical = (data[:, :, 0, 6] + data[:, :, 1, 6] +
            data[:, :, 0, 3] + data[:, :, 1, 3] - Nz_static +
                2 * k3 * (y[:, :, 25] - y[:, :, 17])) / mw
        acc_RW_lateral = (data[:, :, 2, 5] + data[:, :, 3, 5] +
            data[:, :, 2, 2] + data[:, :, 3, 2] -
                2 * k1 * (y[:, :, 5] - y[:, :, 9] + b * y[:, :,
                    11] - h1 * y[:, :, 13])) / mw
        acc_RW_vertical = (data[:, :, 2, 6] + data[:, :, 3, 6] +
            data[:, :, 2, 3] + data[:, :, 3, 3] - Nz_static +
                2 * k3 * (y[:, :, 25] - y[:, :, 19])) / mw

        acc_calculated = torch.stack([acc_FW_lateral,
            acc_FW_vertical, acc_RW_lateral, acc_RW_vertical],
            dim=2)
        # if torch.isnan(acc_calculated).any():
        #         print("acc_calculated contains NaN")
        # elif torch.isinf(acc_calculated).any():
        #         print("acc_calculated contains Inf")
        # else:
        #         print("acc_calculated does not contain NaN or
            Inf")
        return acc_calculated
```

Listing E.3: p2p_acc_calc_tensor.py

E.4 spring_damper_system.py

```python
import numpy as np
import pandas as pd
import scipy
from scipy.interpolate import interp1d
from scipy.integrate import odeint
import matplotlib.pyplot as plt
import seaborn as sns
from plotnine import ggplot, aes, geom_point, theme, labs
from tqdm import tqdm
import time
from IPython.display import display, clear_output

import sklearn
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import KFold
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, mean_absolute_error

import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset
import torch.nn.functional as F
from torch.optim.lr_scheduler import StepLR

class CNN(nn.Module):
    '''
    This class defines a 1D CNN model with a variable number of convolutional layers, kernel numbers, and kernel sizes, etc.
    Keep all kernel sizes odd numbers and with at least one kernel size larger than 200m/resolution.
    '''
    def __init__(self, resolution, device, input_dim, output_dim, num_layers, conv_kernel_numbers,
                                  conv_kernel_sizes, activation, dropout):
        super(CNN, self).__init__()
        assert len(conv_kernel_numbers) == num_layers, "The length of conv_kernel_numbers not equals to num_layers."
        assert len(conv_kernel_sizes) == num_layers, "The length of conv_kernel_sizes not equals to num_layers."
        assert conv_kernel_numbers[-1] == output_dim, "The last element of conv_kernel_numbers should be equal to output_dim."
        for kernel_size in conv_kernel_sizes:
            assert kernel_size % 2 == 1, "All kernel sizes should be odd numbers."

        self.resolution = resolution
        self.device = device
        self.input_dim = input_dim
```

```
        self.output_dim = output_dim
        self.num_layers = num_layers
        self.conv_kernel_numbers = conv_kernel_numbers
        self.conv_kernel_sizes = conv_kernel_sizes

        layers = []
        for i in range(num_layers-1):
            layers.append(nn.Conv1d(in_channels=input_dim if i ==
               0 else conv_kernel_numbers[i-1],
               out_channels=conv_kernel_numbers[i],
                                    kernel_size=conv_kernel_sizes[i],
                                       stride=1, padding=0,
                                       bias=True))

            # layers.append(nn.BatchNorm1d(conv_kernel_numbers[i]))

            if activation == 'relu':
                layers.append(nn.ReLU())
            elif activation == 'elu':
                layers.append(nn.ELU())
            elif activation == 'leaky_relu':
                layers.append(nn.LeakyReLU())
            elif activation == 'sigmoid':
                layers.append(nn.Sigmoid())
            elif activation == 'tanh':
                layers.append(nn.Tanh())
            else:
                raise ValueError('Activation function not
                   supported.')

            # layers.append(nn.Dropout(p=dropout))

        self.conv_layers = nn.Sequential(*layers)
        self.output_layer =
           nn.Conv1d(in_channels=conv_kernel_numbers[-2] if
           num_layers > 1 else input_dim,
                                      out_channels=output_dim,
                                         kernel_size=conv_kernel_sizes[-1],
                                         stride=1, padding=0,
                                         bias=True)

    def forward(self, x):
        '''
        Define the forward pass of the 1D CNN model.
        '''
        x = self.conv_layers(x)
        x = self.output_layer(x)

        return x

    def dataset_reshape(self, segment_length, segment_step,
       X_train, y_train, X_val, y_val, X_test, y_test):
        '''
```

```
        Reshape the input and target datasets into 3D arrays for the 1D CNN model.
        The reshaping pattern depends on the network architecture.
        '''
        print("Segment length:", segment_length / 1000, "km")
        self.segment_size_X = int(segment_length / self.resolution)
        print("Segment size of X:", self.segment_size_X)
        self.segment_size_y = self.segment_size_X
        for i in range(self.num_layers):
            self.segment_size_y = self.segment_size_y - (self.conv_kernel_sizes[i] - 1)
        print("Segment size of y:", self.segment_size_y)

        # Reshape the training dataset into a new 3D tensor (segments, channels, time series)
        num_segments_train = (X_train.shape[0] - self.segment_size_X) // segment_step + 1
        X_train_3d = np.zeros((num_segments_train, self.input_dim, self.segment_size_X))
        y_train_3d = np.zeros((num_segments_train, self.output_dim, self.segment_size_y))
        for i in tqdm(range(num_segments_train)):
            X_train_3d[i, :, :] = X_train.iloc[i * segment_step : i * segment_step + self.segment_size_X, :].T
            y_train_3d[i, :, :] = y_train.iloc[
                (self.segment_size_X - self.segment_size_y) // 2 + i * segment_step :
                (self.segment_size_X - self.segment_size_y) // 2 + i * segment_step + self.segment_size_y, :].T
        X_train_3d = torch.from_numpy(X_train_3d).float()
        y_train_3d = torch.from_numpy(y_train_3d).float()

        # Reshape the validation dataset into a new 3D tensor (segments, channels, time series)
        num_segments_val = (X_val.shape[0] - self.segment_size_X) // segment_step + 1
        X_val_3d = np.zeros((num_segments_val, self.input_dim, self.segment_size_X))
        y_val_3d = np.zeros((num_segments_val, self.output_dim, self.segment_size_y))
        for i in tqdm(range(num_segments_val)):
            X_val_3d[i, :, :] = X_val.iloc[i * segment_step : i * segment_step + self.segment_size_X, :].T
            y_val_3d[i, :, :] = y_val.iloc[
                (self.segment_size_X - self.segment_size_y) // 2 + i * segment_step :
                (self.segment_size_X - self.segment_size_y) // 2 + i * segment_step + self.segment_size_y, :].T
        X_val_3d = torch.from_numpy(X_val_3d).float()
        y_val_3d = torch.from_numpy(y_val_3d).float()

        # Reshape the test dataset into a new 3D tensor (segments, channels, time series)
        # Without overlapping and gap between y's segments
```

```
        num_segments_test = (X_test.shape[0] -
            (self.segment_size_X - self.segment_size_y)) //
            self.segment_size_y
        X_test_3d = np.zeros((num_segments_test, self.input_dim,
            self.segment_size_X))
        for i in tqdm(range(num_segments_test)):
            X_test_3d[i, :, :] = X_test.iloc[i *
                self.segment_size_y : i * self.segment_size_y +
                self.segment_size_X, :].T
        X_test_cut = X_test.iloc[(self.segment_size_X -
            self.segment_size_y) // 2 :
                                    (self.segment_size_X -
                                        self.segment_size_y) // 2
                                        + num_segments_test *
                                        self.segment_size_y, :
                                    ].values
        y_test_cut = y_test.iloc[(self.segment_size_X -
            self.segment_size_y) // 2 :
                                    (self.segment_size_X -
                                        self.segment_size_y) // 2
                                        + num_segments_test *
                                        self.segment_size_y, :
                                    ].values
        X_test_3d = torch.from_numpy(X_test_3d).float()
        X_test_cut = torch.from_numpy(X_test_cut).float()
        y_test_cut = torch.from_numpy(y_test_cut).float()

        return X_train_3d, y_train_3d, X_val_3d, y_val_3d,
            X_test_3d, X_test_cut, y_test_cut

    def set_optimizer(self, criterion_type="mse",
                      optimizer_type="adam", learning_rate=1e-3,
                          weight_decay=0.0):
        '''
        Set the optimizer for the CNN model.
        '''
        self.criteron_type = criterion_type
        if criterion_type == 'mse':
            self.criterion = nn.MSELoss() # MSE
        elif criterion_type == 'l1':
            self.criterion = nn.L1Loss() # MAE
        else:
            raise ValueError('Criterion not supported.')

        if optimizer_type == 'adam':
            self.optimizer = optim.Adam(self.parameters(),
                lr=learning_rate, weight_decay=weight_decay)
        elif optimizer_type == 'sgd':
            self.optimizer = optim.SGD(self.parameters(),
                lr=learning_rate, weight_decay=weight_decay)
        elif optimizer_type == 'rmsprop':
            self.optimizer = optim.RMSprop(self.parameters(),
                lr=learning_rate, weight_decay=weight_decay)
        else:
```

```
            raise ValueError('Optimizer not supported.')

        self.scheduler = StepLR(self.optimizer, step_size=100,
            gamma=0.8)

    def train_model(self, X_train_3d, y_train_3d, X_val_3d,
        y_val_3d,
                    epochs=1000,
                    patience=10, save_gap=10):

        train_losses = []
        val_losses = []
        best_val_loss = float('inf')
        epochs_since_save = 0
        epochs_no_improve = 0
        estimation_criterion = nn.L1Loss()

        for epoch in range(epochs):
            epochs_since_save += 1

            self.train()
            self.optimizer.zero_grad()
            output = self(X_train_3d)
            loss = self.criterion(output, y_train_3d)
            loss.backward()
            self.optimizer.step()

            self.eval()
            with torch.no_grad():
                output_train = self(X_train_3d)
                train_loss_epoch =
                    estimation_criterion(output_train,
                    y_train_3d).item()
                output_val = self(X_val_3d)
                val_loss_epoch = estimation_criterion(output_val,
                    y_val_3d).item()

            train_losses.append(train_loss_epoch)
            val_losses.append(val_loss_epoch)

            print(f"Epoch {epoch+1}/{epochs}, Training Loss:
                {train_losses[-1] * 1e3} mm, Validation Loss:
                {val_losses[-1] *1e3} mm, Learning Rate:
                {self.scheduler.get_last_lr()[0]}")

            # Early stopping logic
            if val_losses[-1] < best_val_loss:
                print("Better results found!")
                best_val_loss = val_losses[-1]
                epochs_no_improve = 0
                if epochs_since_save > save_gap or epoch == 0:
                    print(f'Model and optimizer state_dict saved
                        after {epoch+1} epochs!')
```

```
                    torch.save(self.state_dict(), 'model_state_dict.pth')
                    torch.save(self.optimizer.state_dict(), 'optimizer_state_dict.pth')
                    torch.save(val_losses[-1], 'best_val_loss.pth')
                    epochs_since_save = 0

            else:
                epochs_no_improve += 1
                if epochs_no_improve > patience:
                    print(f'Early stopping triggered after {epoch+1} epochs!')
                    break

            # self.scheduler.step()

        # Load the best model and optimizer state_dict
        self.load_state_dict(torch.load('model_state_dict.pth'))
        self.optimizer.load_state_dict(torch.load('optimizer_state_dict.pth'))

        best_val_loss = torch.load('best_val_loss.pth')
        print("Best validation loss:", best_val_loss * 1e3, "mm")

        return train_losses, val_losses

    def evaluate_model(self, X_test_3d, X_test_cut, y_test_cut, X_train, y_train):
        '''
        Evaluate the CNN model on the test dataset.
        '''
        # Make predictions on the test set
        self.eval()
        with torch.no_grad():
            y_test_pred = self(X_test_3d[0, :, :].unsqueeze(0)).squeeze(0).T
            for i in range(1, X_test_3d.shape[0]):
                y_test_pred = torch.cat((y_test_pred, self(X_test_3d[i, :, :].unsqueeze(0)).squeeze(0).T), dim=0)

        # Baseline of simple average: the mean of training set's target values
        pred_baseline_sa = torch.tensor(np.array(y_train.mean(axis=0))).float().to(self.device) * torch.ones_like(y_test_cut)

        # Baseline of linear regression: fit a linear regression model on the training set
        model_lr = LinearRegression()
        model_lr.fit(X_train, y_train)
        pred_baseline_lr = torch.tensor(model_lr.predict(X_test_cut.cpu().numpy())).float().to(self.

        # Calculate the error of model and the baselines
```

```

        # estimation_criterion = nn.L1Loss()
        errors_model = np.array([mean_absolute_error(y_test_pred[:,i].cpu().numpy(), y_test_cut[:, i].cpu().numpy()) for i in range(self.output_dim)])
        # errors_model = np.array([estimation_criterion(y_test_pred[:,i], y_test_actual[:, i]).item() for i in range(self.output_dim)])
        print("Model errors on test set:", errors_model * 1e3, "mm")
        print("Model mean error on test set:", errors_model.mean() * 1e3, "mm")

        errors_baseline_avg = np.array([mean_absolute_error(pred_baseline_sa[:,i].cpu().numpy(), y_test_cut[:, i].cpu().numpy()) for i in range(self.output_dim)])
        # errors_baseline = np.array([estimation_criterion(baseline[:,i], y_test_actual[:, i]).item() for i in range(self.output_dim)])
        print("Baseline errors on test set (simple average):", errors_baseline_avg * 1e3, "mm")
        print("Baseline mean error on test set (simple average):", errors_baseline_avg.mean() * 1e3, "mm")

        errors_baseline_lr = np.array([mean_absolute_error(pred_baseline_lr[:,i].cpu().numpy(), y_test_cut[:, i].cpu().numpy()) for i in range(self.output_dim)])
        print("Baseline errors on test set (linear regression):", errors_baseline_lr * 1e3, "mm")
        print("Baseline mean error on test set (linear regression):", errors_baseline_lr.mean() * 1e3, "mm")

        return y_test_pred, pred_baseline_sa, pred_baseline_lr, errors_model, errors_baseline_avg, errors_baseline_lr


class PINN(nn.Module):
    '''
    This class defines a 1D CNN model with a variable number of convolutional layers, kernel numbers, and kernel sizes, etc.
    Keep all kernel sizes odd numbers and with at least one kernel size larger than 200m/resolution.
    '''
    def __init__(self, resolution, device, input_dim, output_dim, num_layers, conv_kernel_numbers,
                                  conv_kernel_sizes, activation, dropout):
        super(PINN, self).__init__()
```

```
        assert len(conv_kernel_numbers) == num_layers, "The length
            of conv_kernel_numbers not equals to num_layers."
        assert len(conv_kernel_sizes) == num_layers, "The length
            of conv_kernel_sizes not equals to num_layers."
        assert conv_kernel_numbers[-1] == output_dim, "The last
            element of conv_kernel_numbers should be equal to
            output_dim."
        for kernel_size in conv_kernel_sizes:
            assert kernel_size % 2 == 1, "All kernel sizes should
                be odd numbers."

        self.resolution = resolution
        self.device = device
        self.input_dim = input_dim
        self.output_dim = output_dim
        self.num_layers = num_layers
        self.conv_kernel_numbers = conv_kernel_numbers
        self.conv_kernel_sizes = conv_kernel_sizes

        layers = []
        for i in range(num_layers-1):
            layers.append(nn.Conv1d(in_channels=input_dim if i ==
                0 else conv_kernel_numbers[i-1],
                out_channels=conv_kernel_numbers[i],
                                    kernel_size=conv_kernel_sizes[i],
                                        stride=1, padding=0,
                                        bias=True))

            # layers.append(nn.BatchNorm1d(conv_kernel_numbers[i]))

            if activation == 'relu':
                layers.append(nn.ReLU())
            elif activation == 'elu':
                layers.append(nn.ELU())
            elif activation == 'leaky_relu':
                layers.append(nn.LeakyReLU())
            elif activation == 'sigmoid':
                layers.append(nn.Sigmoid())
            elif activation == 'tanh':
                layers.append(nn.Tanh())
            else:
                raise ValueError('Activation function not
                    supported.')

            # layers.append(nn.Dropout(p=dropout))

        self.conv_layers = nn.Sequential(*layers)
        self.output_layer =
            nn.Conv1d(in_channels=conv_kernel_numbers[-2] if
            num_layers > 1 else input_dim,
                                      out_channels=output_dim,
                                          kernel_size=conv_kernel_sizes[-1],
                                          stride=1, padding=0,
                                          bias=True)
```

```python

    def forward(self, x):
        '''
        Define the forward pass of the 1D CNN model.
        '''
        x = self.conv_layers(x)
        x = self.output_layer(x)

        return x

    def dataset_reshape(self, segment_length, segment_step, X_train, y_train, X_val, y_val, X_test, y_test):
        '''
        Reshape the input and target datasets into 3D arrays for the 1D CNN model.
        The reshaping pattern depends on the network architecture.
        '''
        print("Segment length:", segment_length / 1000, "km")
        self.segment_size_X = int(segment_length / self.resolution)
        print("Segment size of X:", self.segment_size_X)
        self.segment_size_y = self.segment_size_X
        for i in range(self.num_layers):
            self.segment_size_y = self.segment_size_y - (self.conv_kernel_sizes[i] - 1)
        print("Segment size of y:", self.segment_size_y)

        # Reshape the training dataset into a new 3D tensor (segments, channels, time series)
        num_segments_train = (X_train.shape[0] - self.segment_size_X) // segment_step + 1
        X_train_3d = np.zeros((num_segments_train, self.input_dim, self.segment_size_X))
        y_train_3d = np.zeros((num_segments_train, self.output_dim, self.segment_size_y))
        for i in tqdm(range(num_segments_train)):
            X_train_3d[i, :, :] = X_train.iloc[i * segment_step : i * segment_step + self.segment_size_X, :].T
            y_train_3d[i, :, :] = y_train.iloc[
                (self.segment_size_X - self.segment_size_y) // 2 + i * segment_step :
                (self.segment_size_X - self.segment_size_y) // 2 + i * segment_step + self.segment_size_y, :].T
        X_train_3d = torch.from_numpy(X_train_3d).float()
        y_train_3d = torch.from_numpy(y_train_3d).float()

        # Reshape the validation dataset into a new 3D tensor (segments, channels, time series)
        num_segments_val = (X_val.shape[0] - self.segment_size_X) // segment_step + 1
        X_val_3d = np.zeros((num_segments_val, self.input_dim, self.segment_size_X))
        y_val_3d = np.zeros((num_segments_val, self.output_dim, self.segment_size_y))
        for i in tqdm(range(num_segments_val)):
```

```
            X_val_3d[i, :, :] = X_val.iloc[i * segment_step : i * segment_step + self.segment_size_X, :].T
            y_val_3d[i, :, :] = y_val.iloc[
                (self.segment_size_X - self.segment_size_y) // 2 + i * segment_step :
                (self.segment_size_X - self.segment_size_y) // 2 + i * segment_step + self.segment_size_y, :].T
        X_val_3d = torch.from_numpy(X_val_3d).float()
        y_val_3d = torch.from_numpy(y_val_3d).float()

        # Reshape the test dataset into a new 3D tensor (segments, channels, time series)
        # Without overlapping and gap between y's segments
        num_segments_test = (X_test.shape[0] - (self.segment_size_X - self.segment_size_y)) // self.segment_size_y
        X_test_3d = np.zeros((num_segments_test, self.input_dim, self.segment_size_X))
        for i in tqdm(range(num_segments_test)):
            X_test_3d[i, :, :] = X_test.iloc[i * self.segment_size_y : i * self.segment_size_y + self.segment_size_X, :].T
        X_test_cut = X_test.iloc[(self.segment_size_X - self.segment_size_y) // 2 :
                                    (self.segment_size_X - self.segment_size_y) // 2 + num_segments_test * self.segment_size_y, :
                                    ].values
        y_test_cut = y_test.iloc[(self.segment_size_X - self.segment_size_y) // 2 :
                                    (self.segment_size_X - self.segment_size_y) // 2 + num_segments_test * self.segment_size_y, :
                                    ].values
        X_test_3d = torch.from_numpy(X_test_3d).float()
        X_test_cut = torch.from_numpy(X_test_cut).float()
        y_test_cut = torch.from_numpy(y_test_cut).float()

        return X_train_3d, y_train_3d, X_val_3d, y_val_3d, X_test_3d, X_test_cut, y_test_cut

    def set_optimizer(self, criterion_type="mse",
                      optimizer_type="adam", learning_rate=1e-3, weight_decay=0.0):
        '''
        Set the optimizer for the CNN model.
        '''
        self.criteron_type = criterion_type
        if criterion_type == 'mse':
            self.criterion = nn.MSELoss() # MSE
        elif criterion_type == 'l1':
            self.criterion = nn.L1Loss() # MAE
```

```
        else:
            raise ValueError('Criterion not supported.')

        if optimizer_type == 'adam':
            self.optimizer = optim.Adam(self.parameters(),
                lr=learning_rate, weight_decay=weight_decay)
        elif optimizer_type == 'sgd':
            self.optimizer = optim.SGD(self.parameters(),
                lr=learning_rate, weight_decay=weight_decay)
        elif optimizer_type == 'rmsprop':
            self.optimizer = optim.RMSprop(self.parameters(),
                lr=learning_rate, weight_decay=weight_decay)
        else:
            raise ValueError('Optimizer not supported.')

        self.scheduler = StepLR(self.optimizer, step_size=100,
            gamma=0.8)

    def phy_loss(self, m, k, c, pos, irr, vel, acc):
        '''
        Define the physics loss for the CNN model.
        '''
        # ODE: m * acc + k * (pos - irr) + c * v = 0
        return np.mean((m * acc + k * (pos - irr) + c * vel) ** 2)

    def train_model(self, X_train_3d, y_train_3d, X_val_3d,
        y_val_3d,
                    phy_weight, m, k, c,
                    epochs=1000,
                    patience=10, save_gap=10):

        train_losses = []
        estimation_losses = []
        physics_losses = []
        val_losses = []
        best_val_loss = float('inf')
        epochs_since_save = 0
        epochs_no_improve = 0
        estimation_criterion = nn.L1Loss()

        for epoch in range(epochs):
            epochs_since_save += 1

            self.train()
            self.optimizer.zero_grad()
            output = self(X_train_3d)
            loss_estimation = estimation_criterion(output,
                y_train_3d)
            # print("Loss of estimation:", loss_estimation.item())
            estimation_losses.append(loss_estimation.item())
            irr_matching = output[:, 0, :].detach().cpu().numpy()
            pos_matching = X_train_3d[:, 0, (X_train_3d.size(2) -
                output.size(2)) // 2 : (X_train_3d.size(2) -
                output.size(2)) // 2 +
```

```
                output.size(2)].detach().cpu().numpy()
            vel_matching = X_train_3d[:, 1, (X_train_3d.size(2) -
                output.size(2)) // 2 : (X_train_3d.size(2) -
                output.size(2)) // 2 +
                output.size(2)].detach().cpu().numpy()
            vel_matching_forward = X_train_3d[:, 1,
                (X_train_3d.size(2) - output.size(2)) // 2 + 1 :
                (X_train_3d.size(2) - output.size(2)) // 2 + 1 +
                output.size(2)].detach().cpu().numpy()
            acc_matching = vel_matching_forward - vel_matching
            loss_physics_info = self.phy_loss(m, k, c,
                pos_matching, irr_matching, vel_matching,
                acc_matching)
            # print("Loss of physics:", loss_physics_info)
            physics_losses.append(loss_physics_info)
            loss = loss_estimation + phy_weight * loss_physics_info
            loss.backward()
            self.optimizer.step()

            self.eval()
            with torch.no_grad():
                output_train = self(X_train_3d)
                train_loss_epoch =
                    estimation_criterion(output_train,
                    y_train_3d).item()
                output_val = self(X_val_3d)
                val_loss_epoch = estimation_criterion(output_val,
                    y_val_3d).item()

            train_losses.append(train_loss_epoch)
            val_losses.append(val_loss_epoch)

            print(f"Epoch {epoch+1}/{epochs}, Training Loss:
                {train_losses[-1] * 1e3} mm, Validation Loss:
                {val_losses[-1] *1e3} mm, Learning Rate:
                {self.scheduler.get_last_lr()[0]}")

            # Early stopping logic
            if val_losses[-1] < best_val_loss:
                print("Better results found!")
                best_val_loss = val_losses[-1]
                epochs_no_improve = 0
                if epochs_since_save > save_gap or epoch == 0:
                    print(f'Model and optimizer state_dict saved
                        after {epoch+1} epochs!')
                    torch.save(self.state_dict(),
                        'model_state_dict.pth')
                    torch.save(self.optimizer.state_dict(),
                        'optimizer_state_dict.pth')
                    torch.save(val_losses[-1], 'best_val_loss.pth')
                    epochs_since_save = 0

            else:
                epochs_no_improve += 1
```

```
                if epochs_no_improve > patience:
                    print(f’Early stopping triggered after {epoch+1} epochs!’)
                    break

            # self.scheduler.step()

        # Load the best model and optimizer state_dict
        self.load_state_dict(torch.load(’model_state_dict.pth’))
        self.optimizer.load_state_dict(torch.load(’optimizer_state_dict.pth’))

        best_val_loss = torch.load(’best_val_loss.pth’)
        print("Best validation loss:", best_val_loss * 1e3, "mm")

        return train_losses, val_losses, estimation_losses, physics_losses

    def evaluate_model(self, X_test_3d, X_test_cut, y_test_cut, X_train, y_train):
        ’’’
        Evaluate the CNN model on the test dataset.
        ’’’
        # Make predictions on the test set
        self.eval()
        with torch.no_grad():
            y_test_pred = self(X_test_3d[0, :, :].unsqueeze(0)).squeeze(0).T
            for i in range(1, X_test_3d.shape[0]):
                y_test_pred = torch.cat((y_test_pred, self(X_test_3d[i, :, :].unsqueeze(0)).squeeze(0).T), dim=0)

        # Baseline of simple average: the mean of training set’s target values
        pred_baseline_sa = torch.tensor(np.array(y_train.mean(axis=0))).float().to(self.device) * torch.ones_like(y_test_cut)

        # Baseline of linear regression: fit a linear regression model on the training set
        model_lr = LinearRegression()
        model_lr.fit(X_train, y_train)
        pred_baseline_lr = torch.tensor(model_lr.predict(X_test_cut.cpu().numpy())).float().to(self.

        # Calculate the error of model and the baselines

        # estimation_criterion = nn.L1Loss()
        errors_model = np.array([mean_absolute_error(y_test_pred[:,i].cpu().numpy(), y_test_cut[:, i].cpu().numpy()) for i in range(self.output_dim)])
        # errors_model = np.array([estimation_criterion(y_test_pred[:,i],
```

```
            y_test_actual[:, i]).item() for i in
            range(self.output_dim)])
        print("Model errors on test set:", errors_model * 1e3,
            "mm")
        print("Model mean error on test set:", errors_model.mean()
            * 1e3, "mm")

        errors_baseline_avg =
            np.array([mean_absolute_error(pred_baseline_sa[:,i].cpu().numpy(),
            y_test_cut[:, i].cpu().numpy()) for i in
            range(self.output_dim)])
        # errors_baseline =
            np.array([estimation_criterion(baseline[:,i],
            y_test_actual[:, i]).item() for i in
            range(self.output_dim)])
        print("Baseline errors on test set (simple average):",
            errors_baseline_avg * 1e3, "mm")
        print("Baseline mean error on test set (simple average):",
            errors_baseline_avg.mean() * 1e3, "mm")

        errors_baseline_lr =
            np.array([mean_absolute_error(pred_baseline_lr[:,i].cpu().numpy(),
            y_test_cut[:, i].cpu().numpy()) for i in
            range(self.output_dim)])
        print("Baseline errors on test set (linear regression):",
            errors_baseline_lr * 1e3, "mm")
        print("Baseline mean error on test set (linear
            regression):", errors_baseline_lr.mean() * 1e3, "mm")

        return y_test_pred, pred_baseline_sa, pred_baseline_lr,
            errors_model, errors_baseline_avg, errors_baseline_lr
```

Listing E.4: spring_damper_system.py