Learning Dynamics of a Charged Particle in an Electromagnetic Field

Eugenio Macias Banner ID: B01830220

Problem Statement

We aim to predict the future states of a charged electron in an electromagnetic field (EMF) using three distinct deep learning methods: Symplectic Neural Networks (SympNets), Poisson Neural Networks (PNNs), and Physics Informed Neural Networks (PINNs). The dynamics of this system are governed by the Lorentz force , which says that the acceleration of the particle is determined by both electric and magnetic fields:

$$m\ddot{x} = q\left(E(x) + \dot{x} \times B(x)\right),$$

where m is the mass of the particle, q its electric charge, $x \in \mathbb{R}^3$ is its position, and $\dot{x} = v$ is its velocity. The electromagnetic fields are derived from scalar and vector potentials as $B(x) = \nabla \times A(x)$ and $E(x) = -\nabla\varphi(x)$ respectively. This second-order system can be rewritten as a first order system in Poisson form by introducing the state vector $z = (v, x) \in \mathbb{R}^6$. The equations then become:

$$\begin{pmatrix} \dot{v} \\ \dot{x} \end{pmatrix} = \begin{pmatrix} -\frac{q}{m^2}\hat{B}(x) & -\frac{1}{m}I \\ \frac{1}{m}I & 0 \end{pmatrix} \nabla H(v, x),$$

where the Hamiltonian is given by

$$H(v, x) = \frac{1}{2} m v^\top v + q\varphi(x),$$

and $\hat{B}(x)$ is the matrix representation of the cross product with the magnetic field $B(x)$:

$$\hat{B}(x) = \begin{pmatrix} 0 & -B_3(x) & B_2(x) \\ B_3(x) & 0 & -B_1(x) \\ -B_2(x) & B_1(x) & 0 \end{pmatrix}.$$

In this project, we fix $m = 1$ and $q = 1$, and define the vector potential $A(x)$ and scalar potential $\varphi(x)$ to be:

$$A(x) = \frac{1}{3}\sqrt{x_1^2 + x_2^2} \cdot (-x_2, x_1, 0), \qquad \varphi(x) = \frac{1}{100\sqrt{x_1^2 + x_2^2}},$$

for $x = (x_1, x_2, x_3)^\top \in \mathbb{R}^3$. From these potentials, we obtain the EMF:

$$B(x) = (\nabla \times A)(x) = (0, 0, \sqrt{x_1^2 + x_2^2}), \qquad E(x) = -(\nabla \varphi)(x) = \frac{(x_1, x_2, 0)}{100(x_1^2 + x_2^2)^{3/2}}.$$

We consider a dataset of 1500 data points theoretically generated using a störmer-verlet integrator (volume-preserving), each representing the state of the particle in $\mathbb{R}^4$ by concatenating position and velocity vectors. The first 1200 points are used to train the models, while the remaining 300 are used as test to evaluate extrapolation performance. The objective is to explore three model architectures for predicting the trajectory of the charged particle. In the first task, we train a SympNet to learn the discrete flow map $(v_{i-1}, x_{i-1}) \mapsto (v_i, x_i)$ and apply this map recursively to generate multistep predictions. The predicted trajectory is then compared to the ground truth in the x_1–x_2 plane, and the mean squared error (MSE) is computed as a function of time. In the second task, we replace the SympNet with a Poisson Neural Network (PNN), which learns a nonlinear invertible coordinate transformation to latent symplectic space and learns the dynamics using a symplectic map. In the third task we formulate an inverse problem and use a Physics-Informed Neural Network (PINN) to estimate the unknown physical parameters m and q. The learned parameters are used to forward-integrate the Lorentz system by using an integrator, and the resulting predictions are compared to the true trajectory. Each method is evaluated on its ability to capture the long term dynamics of the dynamical system.

Methodology

Theoretical Background

A matrix $H \in \mathbb{R}^{2d \times 2d}$ is said to be symplectic if it preserves the canonical symplectic form, that is,

$$H^\top J H = J,$$

where $J \in \mathbb{R}^{2d \times 2d}$ is the standard symplectic matrix defined as

$$J = \begin{bmatrix} 0 & I_d \\ -I_d & 0 \end{bmatrix}.$$

Similarly, a differentiable map $\phi : U \subseteq \mathbb{R}^{2d} \to \mathbb{R}^{2d}$ is symplectic if its Jacobian matrix satisfies

$$(D\phi(x))^\top J D\phi(x) = J \quad \text{for all } x \in U.$$

A dynamical system is said to be Hamiltonian if its evolution is governed by the differential equation

$$\dot{z} = J^{-1} \nabla H(z),$$

where $z \in \mathbb{R}^{2d}$ is the state vector and $H : \mathbb{R}^{2d} \to \mathbb{R}$ is a smooth function known as the Hamiltonian. Hamiltonian systems preserve the symplectic form, and by Liouville's theorem, they also preserve volume in phase space . A more general class of structure-preserving systems is given by Poisson systems. These take the form

$$\dot{z} = B(z) \nabla H(z),$$

where $B(z)$ is a smooth, state-dependent, skew-symmetric matrix field that satisfies the Jacobi identity in terms of the associated Poisson bracket. For smooth functions $f, g : \mathbb{R}^n \to \mathbb{R}$, the Poisson bracket is defined as

$$\{f, g\}(z) = \nabla f(z)^\top B(z) \nabla g(z).$$

Poisson systems preserve this bracket structure under time . Two important theorems characterize the behavior of such systemss. Poincaré's Theorem says that the flow generated by a Hamiltonian system is symplectic. That is, if ϕ_t denotes the time-t flow map associated with the system $\dot{z} = J^{-1}\nabla H(z)$, then for all $z \in \mathbb{R}^{2d}$,

$$(D\phi_t(z))^\top J D\phi_t(z) = J.$$

This implies that Hamiltonian flows preserve the symplectic geometry of phase space over time. The Darboux-Lie Theorem states that any Poisson system of constant rank can be locally transformed into canonical Hamiltonian form. More precisely, if $B(z)$ defines a Poisson bracket of constant rank $2d$ in a neighborhood of a point $z_0 \in \mathbb{R}^n$, then there exists a local coordinate transformation (a diffeomorphism) $\psi : \mathbb{R}^n \to \mathbb{R}^{2d}$ such that, in the new coordinates, the system takes the canonical Hamiltonian form

$$\dot{P}_i = -\frac{\partial H}{\partial Q_i}, \quad \dot{Q}_i = \frac{\partial H}{\partial P_i}, \quad \text{for } i = 1, \ldots, d.$$

This result allows us to interpret Poisson systems locally (in a latent space) as Hamiltonian systems in appropriate coordinates, and thus motivates learning such coordinate transformations when modeling non-canonical dynamics.

Model Architecture and Intuition

The three neural netowkrs in this project are designed to exploit the geometric structure of the dynamical system it models. The architectural design choices for the three NN's models were chosen by implementation examples provided in Presentations 7 and 8 from the Deep Learning for Scientists and Engineers course [4,5]. The models were adapted and inspired from examples in [4,5] and then I refined through iterative experimentation and tuning using the T4 GPU on Google Collab. 1) The SympNN, is designed to learn structure-preserving flows in canonical Hamiltonian systems. It achieves this by composing nonlinear symplectic maps of the form $\psi = v_k \circ \cdots \circ v_1$, where each v_j is a parameterized symplectic transformation. The model is constructed from a sequence of modules, including linear modules $\mathcal{M}_L$, activation modules $\mathcal{M}_A$, and gradient modules $\mathcal{M}_G$, so that they can guarantee the symplecticity of the learned map. As a result, SympNN imitates the structure of Hamiltonian flows by encoding a discrete-time integrator that respects the symplectic form. 2) The PNN extends the capabilities of SympNN to a submanifold of lower dimension by the Darboux-Lie Theorem, which ensures that Poisson systems of constant rank can be locally transformed into Hamiltonian systems. The PNN architecture ten learns a smooth, invertible transformation $\theta : \mathbb{R}^n \to \mathbb{R}^{2d}$ that maps the original coordinates into a latent symplectic space. In this transformed space, a symplectic map Φ evolves the dynamics, and the inverse transformation θ^{-1} maps the result back to the original coordinate space (a piecewise composition). Thus, the overall learned flow is given by the composition $\hat{z}_{t+1} = \theta^{-1} \circ \Phi \circ \theta(z_t)$, where Φ is learned using a SympNet. The encoder θ and decoder θ^{-1} are parameterized using invertible neural networks (INNs). While the learned symplectic map Φ is guaranteed to preserve volume due to its symplecticity, the volume preserving property of the overall flow depends on whether the coordinate transformation θ itself is volume-preserving. In my implementation, I did not enforce volume preservation in the inverse neural network θ, so the model can be classified as a non-volume preserving PNN (NVP-PNN). In contrast, the V-PNN introduced by Jin et al. [1] ensures that the learned transformation θ preserves volume, thereby making sure that the entire composition $\theta^{-1} \circ \Phi \circ \theta$ also preserves volume. To verify that the transformation θ is volume-preserving, I must check that the determinant of its Jacobian satisfies $\det(D\theta(z)) = 1$ for all z which is a point of improvement for next implementations. The VP-PNN would likely have better long-term generalization beyond this horizon by maintaining volume preserving phase flow.

3) The PINN, was an inverse problem of recovering unknown physical parameters from trajectory data. The PINN then outputs predicted states $\hat{y}_i = \mathrm{NN}(x_i)$ and is trained using a hybrid loss that uses observed data and physics laws. The total loss is defined as (I used $\lambda = 0.50$)

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{data}} + \lambda_{\text{phys}} \cdot \mathcal{L}_{\text{physics}},$$

where the supervised data loss is given by

$$\mathcal{L}_{\text{data}} = \frac{1}{N} \sum_{i=1}^{N} \|\hat{y}_i - y_i\|^2,$$

and the physics loss penalizes deviations from the Lorentz force dynamics,

$$\mathcal{L}_{\text{physics}} = \frac{1}{N} \sum_{i=1}^{N} \left\| \frac{d}{dt}\hat{v}_i - \left(\frac{q}{m} E(x_i) + \frac{q}{m} \hat{v}_i \times B(x_i) \right) \right\|^2 .$$

The derivative $\frac{d}{dt}\hat{v}_i$ was approximated using finite differences over predicted velocity sequences. The training data was generated using the Störmer-Verlet method, a second-order symplectic integrator that preserves volume in phase space. In contrast, the predictions generated by the PINN are produced using a high-order Runge-Kutta integrator (specifically, the DOP853 method), which provides higher accuracy but does not preserve volume.

Table 1: Neural Architecture Details

Feature	**SympNN**	**PNN**	**PINN**
Layers	6 SympBlocks	INN + SymplecticMap + INN^{-1}	FC Net (3 layers)
Hidden Size	64/block	64/layer	64/layer
Modules	$F(x), G(v)$	Invertible + symplectic	Physics loss + learnable m, q
Loss	MSE	MSE	MSE + physics residual
Prediction	300 steps	300 steps	ODE (DOP853) with learned m, q
Epochs	15500	20000	10000
Optimizer	Adam (1e-4)	Adam (1e-5)	Adam (1e-5)
Parameters	Trajectory map	Coord. transform + SympNet	m, q only

Results

The performance of each model was evaluated by comparing predicted trajectories against ground, using both visualizations in phase space and MSE. The SYMP-NN had strong short-time prediction accuracy. The phase-space trajectory tracked the true orbit closely at the beginning, but errors accumulated gradually as the system evolved. It had a training MSE of 1.052×10^{-5} and an average trajectory MSE of 0.017. As seen in the phase-plane plots, the predicted orbit begins to diverge structurally beyond step 150, due the geometric mismatch. The PNN achieved the best long-term trajectory prediction among all models. By learning to accurately embed the non-canonical Poisson structure. The training MSE was significantly lower at 2.8987×10^{-6}, with an average trajectory MSE of 6.7×10^{-3}. The MSE over time remained mor stable compared to SympNN. The PINN accurately recovered the parameters, yielding $m \approx 1.019$ and $q \approx 0.981$, both within 2% of the true values. However, the model suffered from higher prediction error over time. The total loss was $\mathcal{L}_{\text{total}} = 1.623 \times 10^{-4}$, with data loss $\mathcal{L}_{\text{data}} = 5.293 \times 10^{-5}$ and physics loss $\mathcal{L}_{\text{physics}} = 2.188 \times 10^{-4}$. The trajectory MSE averaged 5.189×10^{-2}, and the prediction error curve increased rapidly due to the integrator mismatch. The PINN predictions generated using the DOP853 Runge-Kutta method, unlike the symplectic Störmer-Verlet resulted in artificial energy drift, leading to divergence in longer timestep predictions and shows the importance of using symplectic integration schemes when modeling conservative dynamical systems. In summary, SympNN captures short term dynamics well but lacks structural generality; PINN effectively recovers physical laws but suffers from integration error; and PNN achieves the best balance between structure and accuracy, especially for long term rollout..

Conclusion

In summary, I found the SympNN to be effective for canonical Hamiltonian systems but struggled to generalize to Poisson dynamics due to its structural assumptions. The PNN, by learning a latent symplectic representation through the diffeomorphism and SympNet evolution, successfully captured the non-canonical structure and had the most accurate long-term predictions. The PINN demonstrated to be great for solving inverse problems and accurately recovering physical parameters, though its predictions were hindered by the use of a non-symplectic integrator. While the PNN performed well, enforcing volume preservation in the coordinate transformation would further improve stability and accuracy, particularly for long trajectory extrapolation. Some directions I did not follow and consider for improvement include performing a full grid search across all three architectures to identify optimal hyperparameters and model structures. For the PNN specifically, ensuring that the learned diffeomorphism θ is volume-preserving. In the case of the PINN, an addition would be to treat the hybrid loss weight λ_{phys} as a learnable parameter rather than fixing it. All experiments were conducted in PyTorch, but future work might benefit from implementing models in JAX or DeepXDE for faster automatic differentiation and training. Additionally, access to a higher-performance GPU would enable more extensive training and have more robust models.

References

[1] P. Jin, Z. Zhang, I. G. Kevrekidis, and G. E. Karniadakis. *Learning Poisson systems and trajectories of autonomous systems via Poisson neural networks.* arXiv preprint arXiv:2012.03133, 2020.

[2] P. Jin, Z. Zhang, A. Zhu, Y. Tang, and G. E. Karniadakis. *SympNets: Intrinsic structure-preserving symplectic networks for identifying Hamiltonian systems.* Neural Networks, 132:166–179, 2020.

[3] M. Raissi, P. Perdikaris, and G. E. Karniadakis. *Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations.* Journal of Computational Physics, 378:686–707, 2019.

[4] K. Shukla, A. Bora. *Lecture 7: Data-Driven Dynamical Systems*, Brown University - Deep Learning for Scientists and Engineers (DLI).

[5] K. Shukla, A. Bora. *Lecture 8: Physics-Informed Neural Networks (PINNs)*, Brown University - Deep Learning for Scientists and Engineers (DLI).

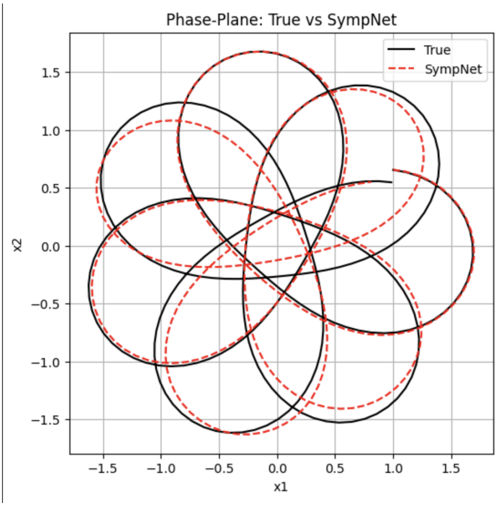
[6] K. Shukla, A. Bora. *Lecture 8: Physics-Informed Neural Networks (PINNs) – Part II*, Brown University - Deep Learning for Scientists and Engineers (DLI), developed in collaboration with NVIDIA Deep Learning Institute and MathWorks.

[7] Wikipedia contributors. *Symplectomorphism.* Retrieved from `https://en.wikipedia.org/wiki/Symplectomorphism`

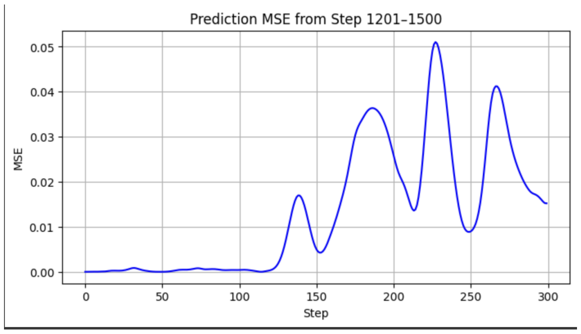
[8] Wikipedia contributors. *Verlet integration.* Retrieved from `https://en.wikipedia.org/wiki/Verlet_integration`

[9] Wikipedia contributors. *Symplectic integrator.* Retrieved from `https://en.wikipedia.org/wiki/Symplectic_integrator`

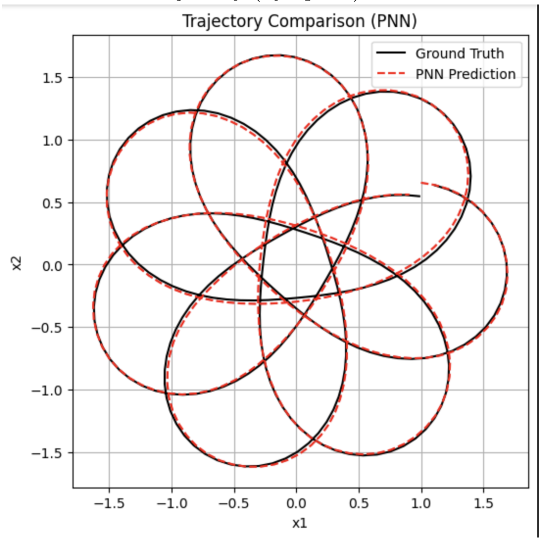
Appendix



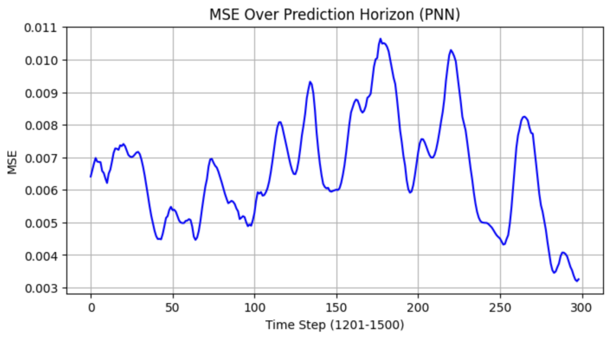
Trajectory (SympNet)



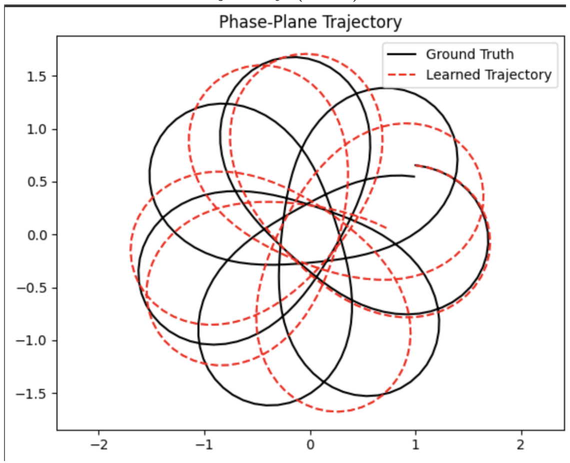
MSE (SympNet)



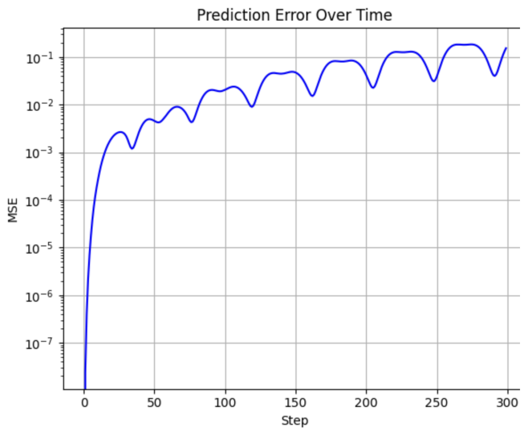
Trajectory (PNN)



MSE (PNN)



Trajectory (PINN)



MSE (PINN)

Metric	SympNet	PNN	PINN
MSE Training	1.052×10^{-5}	2.8987×10^{-6}	—
Avg. MSE Trajectory	0.017	6.7×10^{-3}	5.189×10^{-2}
Total MSE	—	—	1.623×10^{-4}
Data MSE	—	—	5.293×10^{-5}
Physics MSE	—	—	2.188×10^{-4}
Learned m	—	—	1.019234
Learned q	—	—	0.981129

Model Code Listings

```
# SYMP-NN
import numpy as np
import torch
import torch.nn as nn
import matplotlib.pyplot as plt

#T4 GPU
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("Using device:", device)

#Load data:
data = np.loadtxt('train.txt')
X_all = np.column_stack([data[:-1, :2], data[:-1, 2:]])
Y_all = np.column_stack([data[1:,  :2], data[1:,  2:]])

#Split into training (first 1200) and test (remaining 300) pairs
N_train = 1200
X_train = torch.tensor(X_all[:N_train], dtype=torch.float32, device=device
    )
Y_train = torch.tensor(Y_all[:N_train], dtype=torch.float32, device=device
    )
initial = torch.tensor(X_all[N_train-1], dtype=torch.float32, device=
    device).unsqueeze(0)
Y_test_np = Y_all[N_train-1:N_train-1+300, 2:]

# SYMP-block
class SympBlock(nn.Module):
    def __init__(self, dim=4, hidden_dim=128):
        super().__init__()
        self.F = nn.Sequential(
            nn.Linear(dim//2, hidden_dim), nn.Tanh(), nn.Linear(hidden_dim
    , dim//2))
        self.G = nn.Sequential(
            nn.Linear(dim//2, hidden_dim), nn.Tanh(), nn.Linear(hidden_dim
    , dim//2))
        self._init_weights()

    def _init_weights(self):
        # Xavier initialization from class notes (achieved faster
    convergence when tested)
```

```
        for layer in list(self.F) + list(self.G):
            if isinstance(layer, nn.Linear):
                nn.init.xavier_uniform_(layer.weight)
                nn.init.zeros_(layer.bias)

    def forward(self, z):
        v, x = z[:, :2], z[:, 2:]
        v = v + self.F(x)
        x = x + self.G(v)
        return torch.cat([v, x], dim=1)

class SympNet(nn.Module):
    def __init__(self, dim=4, hidden_dim=64, n_blocks=6):
        super().__init__()
        self.blocks = nn.Sequential(*[SympBlock(dim, hidden_dim) for _ in
    range(n_blocks)])

    def forward(self, z):
        return self.blocks(z)

# Optimizer and parameters tweaked by hand after some trial and error
model = SympNet().to(device)
criterion = nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.0001, weight_decay=1
    e-6)

# Train
for epoch in range(1, 15501):
    model.train()
    optimizer.zero_grad()
    loss = criterion(model(X_train), Y_train)
    loss.backward()
    optimizer.step()
    if epoch % 100 == 0:
        print(f"Epoch {epoch}/15500     Train MSE: {loss.item():.3e}")

# Recursive prediction starting from step 1200
model.eval()
preds = []
z = initial.clone()
with torch.no_grad():
    for _ in range(300):
        z = model(z)
        preds.append(z.cpu().numpy()[0, 2:])
preds = np.vstack(preds)

#Plot phase plane and MSE
plt.figure(figsize=(6,6))
plt.plot(Y_test_np[:, 0], Y_test_np[:, 1], 'k-', label='True')
plt.plot(preds[:, 0], preds[:, 1], 'r--', label='SympNet')
plt.xlabel('x1'); plt.ylabel('x2')
plt.title('Phase-Plane: True vs SympNet')
plt.grid(); plt.axis('equal'); plt.legend()

```

```python
mse = np.mean((preds - Y_test_np)**2, axis=1)
plt.figure(figsize=(8,4))
plt.plot(mse, 'b-')
plt.xlabel('Step'); plt.ylabel('MSE')
plt.title('Prediction MSE from Step 1201  1500  ')
plt.grid()
plt.show()
```

Listing 1: SympNN Code

```python
import numpy as np
import torch
import matplotlib.pyplot as plt
from torch.utils.data import DataLoader, TensorDataset

data = np.loadtxt('train.txt')
train_data = data[:1200]
test_data = data[1200:]

def create_sequences(data):
    inputs = data[:-1, [2,3,0,1]]
    targets = data[1:, [2,3,0,1]]
    return inputs, targets

X_train, Y_train = create_sequences(train_data)
X_test, Y_test = create_sequences(test_data)
true_x = test_data[1:, 2:4]

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
X_train = torch.tensor(X_train, dtype=torch.float32, device=device)
Y_train = torch.tensor(Y_train, dtype=torch.float32, device=device)
X_test = torch.tensor(X_test, dtype=torch.float32, device=device)
Y_test = torch.tensor(Y_test, dtype=torch.float32, device=device)

train_loader = DataLoader(TensorDataset(X_train, Y_train), batch_size=64, shuffle=True)


#Diffeomorphism-theta (Invertible NN) (How to check if it has Jacobian = 1?)
class INN(torch.nn.Module):
    def __init__(self, dim=4, hidden_dim=64):
        super().__init__()
        self.fc1 = torch.nn.Linear(dim//2, hidden_dim)
        self.fc2 = torch.nn.Linear(hidden_dim, dim//2)
        self.fc3 = torch.nn.Linear(dim//2, hidden_dim)
        self.fc4 = torch.nn.Linear(hidden_dim, dim//2)

    def forward(self, x, reverse=False):
        x1, x2 = x[:, :2], x[:, 2:]
        if not reverse:
            x2 = x2 + self.fc2(torch.relu(self.fc1(x1)))
            x1 = x1 + self.fc4(torch.relu(self.fc3(x2)))
        else: #theta^-1
```

```
            x1 = x1 - self.fc4(torch.relu(self.fc3(x2)))
            x2 = x2 - self.fc2(torch.relu(self.fc1(x1)))
        return torch.cat([x1, x2], dim=1)

#Darboux Lie Theorem applies here: Symplectic in latent space
class SymplecticMap(torch.nn.Module):
    def __init__(self, dim=4, hidden_dim=64):
        super().__init__()
        self.sympnet = torch.nn.Sequential(
            torch.nn.Linear(dim, hidden_dim),
            torch.nn.Tanh(),
            torch.nn.Linear(hidden_dim, dim))

    def forward(self, z):
        return self.sympnet(z)
#PNN module
class PNN(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.theta = INN()
        self.phi = SymplecticMap()

    def forward(self, x): # composition of theta(phi)theta^-1
        z = self.theta(x)
        z = self.phi(z)
        return self.theta(z, reverse=True)

model = PNN().to(device)

#  Train: add some regularization to prevent overfit
optimizer = torch.optim.Adam(model.parameters(), lr=1e-5, weight_decay=1e
    -6)
criterion = torch.nn.MSELoss()

for epoch in range(20000):
    model.train()
    total_loss = 0.0
    for inputs, targets in train_loader:
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, targets)
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
    if epoch % 100 == 0:
        print(f"Epoch {epoch}, Loss: {total_loss/len(train_loader):.4e}")

#Recursive prediction starting from step 1200
def predict_trajectory(model, initial_state, steps):
    model.eval()
    current_state = initial_state.unsqueeze(0)
    predictions = []
    with torch.no_grad():
        for _ in range(steps):
```

```python
            next_state = model(current_state)
            predictions.append(next_state.cpu().numpy())
            current_state = next_state
    return np.vstack(predictions)
initial_state = torch.tensor(train_data[-1, [2,3,0,1]], dtype=torch.
    float32, device=device)
preds = predict_trajectory(model, initial_state, steps=299)
pred_x = preds[:, :2]

#Plot phase plane and MSE
plt.figure(figsize=(6,6))
plt.plot(true_x[:,0], true_x[:,1], 'k-', label='Ground Truth')
plt.plot(pred_x[:,0], pred_x[:,1], 'r--', label='PNN Prediction')
plt.xlabel('x1'); plt.ylabel('x2'); plt.legend()
plt.title('Trajectory Comparison (PNN)'); plt.grid(True); plt.axis('equal'
    )
plt.show()

mse = np.mean((pred_x - true_x)**2, axis=1)
plt.figure(figsize=(8,4))
plt.plot(mse, 'b-')
plt.xlabel('Time Step (1201-1500)'); plt.ylabel('MSE')
plt.title('MSE Over Prediction Horizon (PNN)'); plt.grid(True)
plt.show()
```

Listing 2: PNN Code

```python
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
from scipy.integrate import solve_ivp
import matplotlib.pyplot as plt
torch.set_default_dtype(torch.float64) #for extra precision as we are
    integrating the dynamical system
# Hyperparameters: (Should lambda be dynamic?)
h = 0.1
epochs = 10000
lr = 1e-5
lambda_phys = 0.50
batch_size = 128
data = np.loadtxt("train.txt")
train_data = data[:1200]
test_data = data[1200:]
X = train_data[:-1]
Y = train_data[1:]
X_train = torch.tensor(X, dtype=torch.float64)
Y_train = torch.tensor(Y, dtype=torch.float64)

# Physics-Informed-NN
class PINN_Learn_mq(nn.Module):
    def __init__(self, input_dim=4, hidden_dim=64, output_dim=4):
        super().__init__()
        self.log_m = nn.Parameter(torch.tensor([0.0]))
        self.log_q = nn.Parameter(torch.tensor([0.0]))
        self.net = nn.Sequential(
            nn.Linear(input_dim, hidden_dim),
            nn.Tanh(),
            nn.Linear(hidden_dim, hidden_dim),
            nn.Tanh(),
            nn.Linear(hidden_dim, output_dim)
        )

    def forward(self, x):
        return self.net(x)

    def get_params(self):
        return torch.exp(self.log_m), torch.exp(self.log_q) #Initialize them as x+ = exp^ln(x)

#  Calculate the residual for the Loss_physics
def net_residual(model, x, h):
    m, q = model.get_params()
    with torch.set_grad_enabled(True):
        x = x.clone().detach().requires_grad_(True)
        y_pred = model(x)
        v = x[:, :2]
        pos = x[:, 2:]
        r = torch.norm(pos, dim=1, keepdim=True) + 1e-9
        Bz = r
        grad_Hv = m * v
        B_force = torch.cat([-Bz * grad_Hv[:, 1:2], Bz * grad_Hv[:, 0:1]], dim=1)
        grad_phi = pos / (100 * r**3)
        grad_Hx = q * grad_phi
        dvdt_rhs = - q / m**2 * B_force - grad_Hx / m
        dxdt_rhs = grad_Hv / m
        rhs = torch.cat([dvdt_rhs, dxdt_rhs], dim=1)
        v_pred = y_pred[:, :2]
        pos_pred = y_pred[:, 2:]
        dvdt_pred = (v_pred - v) / h
        dxdt_pred = (pos_pred - pos) / h
        lhs = torch.cat([dvdt_pred, dxdt_pred], dim=1)

    return lhs - rhs

# Hybrid loss
def loss_fn(model, x, y, h, lambda_phys):
    pred = model(x)
    data_loss = nn.functional.mse_loss(pred, y)
    physics_loss = torch.mean(net_residual(model, x, h)**2)
    return data_loss + lambda_phys * physics_loss, data_loss.item(), physics_loss.item()

# Train
model = PINN_Learn_mq()
optimizer = torch.optim.Adam(model.parameters(), lr=lr)
train_loader = torch.utils.data.DataLoader(torch.utils.data.TensorDataset(
    X_train, Y_train), batch_size=batch_size, shuffle=True)

for ep in range(1, epochs + 1):
    for xb, yb in train_loader:
        optimizer.zero_grad()
        loss, data_l, phys_l = loss_fn(model, xb, yb, h, lambda_phys)
        loss.backward()
        optimizer.step()
    if ep % 200 == 0 or ep == 1:
        m_val, q_val = model.get_params()
        print(f"Ep {ep:4d} | total={loss.item():.3e} | data={data_l:.3e} |
     phys={phys_l:.3e} | m={m_val.item():.6f} | q={q_val.item():.6f}")

m_learned, q_learned = model.get_params()
print(f"\n>>> Learned m = {m_learned.item():.6f}, q = {q_learned.item():.6
    f}")

# ODE Integration part
def lorentz_rhs(t, state, m, q):
    v1, v2, x1, x2 = state
    r = np.sqrt(x1**2 + x2**2) + 1e-8
    Bz = r
    grad_H_v = m * np.array([v1, v2])
    B_force = np.array([-Bz * grad_H_v[1], Bz * grad_H_v[0]])
    grad_phi = np.array([x1, x2]) / (100 * r**3)
    grad_H_x = q * grad_phi
    dvdt = - q / m**2 * B_force - grad_H_x / m
    dxdt = grad_H_v / m
    return np.concatenate([dvdt, dxdt])

init_state = test_data[0]
t_eval = np.linspace(0, (len(test_data)-1)*h, len(test_data))

sol = solve_ivp(
    lorentz_rhs,
    [0, t_eval[-1]],
    init_state,
    t_eval=t_eval,
    args=(m_learned.item(), q_learned.item()),
    method="DOP853" #I read this is the most precise one
)

# Evaluate and plott
pred_traj = sol.y[2:,:].T
true_traj = test_data[:, 2:]

plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.plot(true_traj[:, 0], true_traj[:, 1], 'k-', label="Ground Truth")
plt.plot(pred_traj[:, 0], pred_traj[:, 1], 'r--', label="Learned
    Trajectory")
plt.axis('equal'); plt.legend(); plt.title("Phase-Plane Trajectory")

mse_t = np.mean((pred_traj - true_traj)**2, axis=1)
```

```
plt.subplot(1, 2, 2)
plt.semilogy(mse_t, 'b-')
plt.xlabel("Step"); plt.ylabel("MSE"); plt.grid(True)
plt.title("Prediction Error Over Time")

plt.tight_layout()
plt.show()
print(f"\nAverage MSE over horizon: {mse_t.mean():.3e}")
```

Listing 3: PINN Code