

TP5_Colin

September 6, 2026

1 Koopman operator learning, a toy case: Duffing oscillator

– PLEASE SUBMIT YOUR COMPLETED NOTEBOOK WITH CELL OUTPUTS –

The aim of this notebook is to describe the dynamics of a non-linear dynamical system by means of the Koopman theory.

1.1 Introduction

We consider a quantity $x \in \mathbb{R}^n$ (a vector) which evolves with time, following a dynamical system. Think for example of the joint location of the planets in our solar system, which follows the law of gravitation.

Formally, given an initial state $x(t=0) \in \mathbb{R}^n$ at time $t=0$, the time evolution of x is governed by the following dynamical system:

$$\dot{x}(t) = f(x(t)) \quad (1)$$

where $\dot{x}(t) := \frac{dx(t)}{dt}$ is the temporal derivative, and $f : \mathbb{R}^n \rightarrow \mathbb{R}^n$ is a given map describing the dynamics.

For a given f , it is not always possible to solve the differential equation (1) analytically. For this reason, instead, numerical schemes are usually employed, to integrate in time t the equation (1), so as to propagate the initial condition $x(0)$ up to a desired time T ; think of $x(T) = x(0) + \int_{t=0}^T f(x(t))dt$. The discretization in time of eq (1) or of the integral introduces numerical approximations, and yields estimates of $x(T)$ of various quality depending on the discretization scheme.

In the field of numerical simulations, discretization schemes have been studied for a long time, and numerical solvers already exist to provide good estimates of integrals (far better than with the naive discretization $x_{k+\delta} = x_k + \delta f(x_k)$ for a discrete time increment δ , which induces a $O(\delta^2)$ error at each time step).

The goal of this practical session is to make use of such numerical solvers to improve the learning of dynamical systems with neural networks.

```
[39]: import numpy as np
      from scipy.integrate import solve_ivp
      import matplotlib.pyplot as plt
      import sys
      from tqdm import tqdm

      from torch.utils.data import TensorDataset, DataLoader
```

```

import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.autograd as autograd

# arrange the dataset
from sklearn.model_selection import train_test_split

```

1.2 Duffing oscillator

As a toy example, we consider the Duffing oscillator, where the state $x = (x_1, x_2) \in \mathbb{R}^2$ follows the dynamical system described by the following ODEs:

$$\dot{x}_1 = x_2, \dot{x}_2 = x_1 - x_1^3$$

To integrate in time the ODEs, a 4th order Runge-Kutta scheme can be used.

```

[40]: def duffing(array_x: np.ndarray) -> np.ndarray:
        array_dx = np.zeros(array_x.shape)
        array_dx[0] = array_x[1]
        array_dx[1] = array_x[0] - array_x[0] ** 3
        return array_dx

```

```

[41]: t_max = 500 # Time-horizon integration
        n_iter = 5000 # Number of time steps integration
        n_initial_conditions = 60 # Number of initial conditions

        dim_system = 2

        # Generate initial conditions
        matrix_x0 = (np.random.rand(n_initial_conditions, dim_system) - 0.5) * 4
        array_t = np.linspace(0, t_max, n_iter)
        array3d_xt = np.zeros((matrix_x0.shape[0], matrix_x0.shape[1], n_iter))

        for i in tqdm(range(matrix_x0.shape[0])):
            # Lambda function is used as solve_ivp requires a function of the form f(t,
            ↪ x)
            ode_result = solve_ivp(lambda _t, array_x: duffing(array_x),
                                    [0, t_max],
                                    matrix_x0[i],
                                    method='RK45',
                                    t_eval=array_t)

            array3d_xt[i, :] = ode_result.y

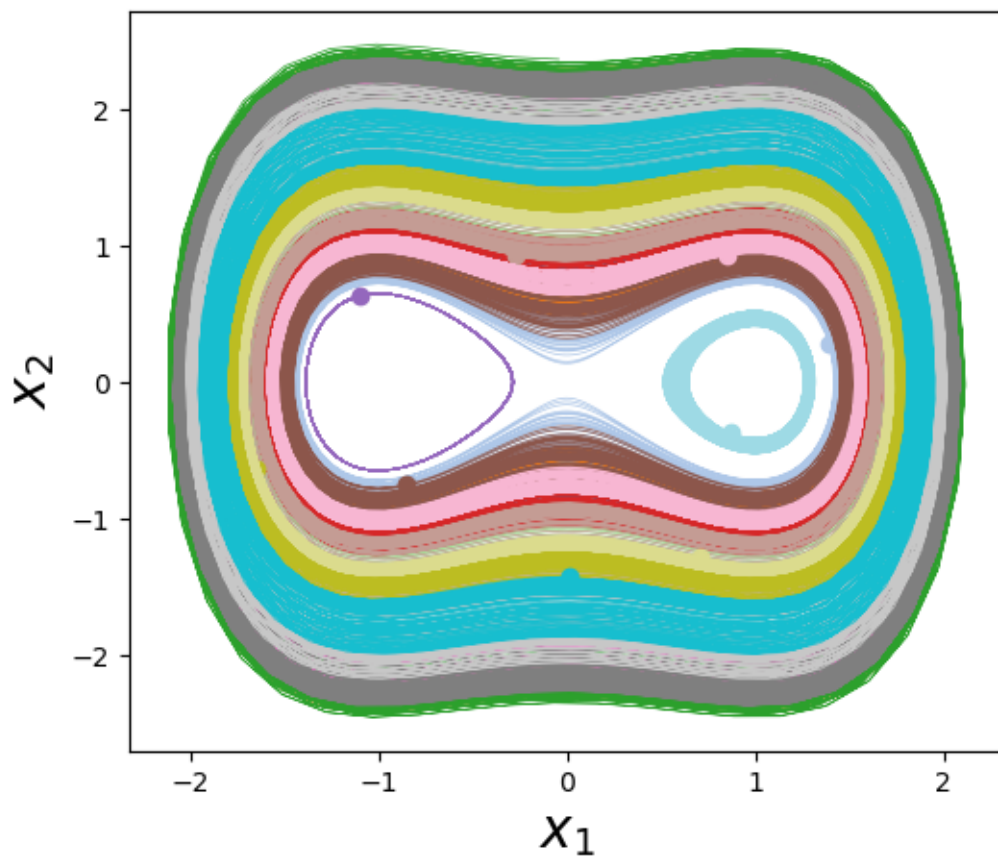
```

100%|| 60/60 [00:04<00:00, 12.48it/s]

The following plot shows trajectories for different initial conditions:

```
[42]: fig = plt.figure(figsize=(20, 5))
ax = fig.add_subplot(131)
cm = plt.get_cmap("tab20")
print(cm)
for i in range(20):
    ax.plot(array3d_xt[i, 0, :], array3d_xt[i, 1, :], lw=0.5, color=cm(i))
    ax.plot(array3d_xt[i, 0, 0], array3d_xt[i, 1, 0], 'o', lw=1.5, color=cm(i))
    ↪ #initial condition
ax.set_xlabel('$x_1$', fontsize=20)
ax.set_ylabel('$x_2$', fontsize=20)
plt.show()
```

<matplotlib.colors.ListedColormap object at 0x0000017E8D73FBE0>



1.3 The Koopman operator

1.3.1 Discontinuous in time case

Given the discrete non-linear dynamical system

$$x_{k+1} = F(x_k)$$

where F might be the δ -discretised flow map of the continuous dynamical system in eq (1) given by

$$x_{k+1} = F(x_k) := x_k + \int_k^{k+\delta} f(x(s))ds$$

and $X = (x_k)_{k=0}^N$ the discrete time series of the system state.

The *Koopman* theory states that there exists an infinite-dimensional linear operator $\mathcal{K}$ that advances in time all observable functions $(g_i)_{i=1}^m$ given by $g_i : \mathbb{R}^n \rightarrow \mathbb{R}$

$$\mathcal{K}g_i(x) = g_i \circ F(x)$$

This way, the non-linear dynamics of x , described by F , can be turned into a **linear** dynamical system, described by $\mathcal{K}$, acting on another representation space, formed by the observable quantities $g_i(x)$.

Indeed, let g_i be an observable function and denoting $g_{ik} := g_i(x_k)$, using the previous equation, the time evolution of the observables is given by

$$g_{ik+1} = g_i(x_{k+1}) = g_i(F(x_k)) = g_i \circ F(x_k) = \mathcal{K}g_i(x_k) = \mathcal{K}g_{ik}$$

then, the linearised dynamics of the observables is given by the following equation

$$g_{ik+1} = \mathcal{K}g_{ik}$$

It is then sufficient to find a function $g : \mathbb{R}^n \rightarrow \mathbb{R}^m$ with $m \gg n$ that embeds the state x into a “larger enough” dimensional space m such that the linear operator $\mathcal{K}$ can be inferred by a matrix $\mathbf{K} \in \mathbb{R}^{m \times m}$.

To project back the dynamics from the Koopman space ($\mathbb{R}^m$, where $g(x)$ lives) to the phase space ($\mathbb{R}^n$, where x lives), a supplementary function $\varphi : \mathbb{R}^m \rightarrow \mathbb{R}^n$ is needed. Going from x to the Koopman space and back yields $\varphi \circ g = \text{Id}$.

Under this condition, the functions g , φ and $\mathbf{K}$ can be parametrized g_θ , φ_ρ and $\mathbf{K}_\phi$, and the parameters θ , ρ and ϕ can be learned minimizing suitable loss functions.

For this purpose, given a time series $X = \{x_k | k = 1 \dots N\}$, the following conditions hold:

1. Reconstruction error

$$\|\varphi_\rho(g_\theta(x_k)) - x_k\| = 0$$

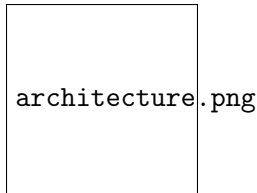
2. Prediction error in Koopman space

$$\|\mathbf{K}_\phi g_\theta(x_k) - g_\theta(x_{k+1})\| = 0$$

3. Prediction error in the phase space

$$\|\varphi_\rho(\mathbf{K}_\phi g_\theta(x_k)) - x_{k+1}\| = 0$$

The last three errors can be used as loss functions to train three different neural networks. These different neural networks compose our architecture that can be summarized as in the following sketch:



```
[43]: # Flatten the trajectories w.r.t. initial conditions
# and only keep data in the form of (dim_system, n_iter * n_initial_conditions)
matrix_x_data = array3d_xt[:, :, :-1].swapaxes(0, 1).reshape(2, -1).T
matrix_x_next_data = array3d_xt[:, :, 1:].swapaxes(0, 1).reshape(2, -1).T

(matrix_x_data_train,
 matrix_x_data_test,
 matrix_x_next_data_train,
 matrix_x_next_data_test) = train_test_split(matrix_x_data,
                                             matrix_x_next_data,
                                             test_size=0.2)

# Cast type to float32
matrix_x_data_train = matrix_x_data_train.astype(np.float32)
matrix_x_data_test = matrix_x_data_test.astype(np.float32)
matrix_x_next_data_train = matrix_x_next_data_train.astype(np.float32)
matrix_x_next_data_test = matrix_x_next_data_test.astype(np.float32)

print(matrix_x_data_train.shape,
      matrix_x_data_test.shape,
      matrix_x_next_data_train.shape,
      matrix_x_next_data_test.shape)
```

```
(239952, 2) (59988, 2) (239952, 2) (59988, 2)
```

```
[44]: device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(device)
```

```
cuda
```

```
[45]: torch.set_default_dtype(torch.float32)
# torch.set_default_tensor_type('torch.DoubleTensor')

batch_size = 2000 # data per batch

tensor2d_x_data_train = torch.from_numpy(matrix_x_data_train).to(device)
tensor2d_x_next_data_train = torch.from_numpy(matrix_x_next_data_train).
    ↪to(device)
```

```

tensor2d_x_data_test = torch.from_numpy(matrix_x_data_test).to(device)
tensor2d_x_next_data_test = torch.from_numpy(matrix_x_next_data_test).to(device)

torch_dataset_train = TensorDataset(tensor2d_x_data_train,
                                     tensor2d_x_next_data_train)

torch_dataset_test = TensorDataset(tensor2d_x_data_test,
                                    tensor2d_x_next_data_test)

train_dataloader = DataLoader(torch_dataset_train,
                              batch_size=batch_size,
                              shuffle=True)
test_dataloader = DataLoader(torch_dataset_test,
                             batch_size=batch_size,
                             shuffle=True)

# create the models
feature_dim = 2 # dimension of the Duffing oscillator
hidden_layer = 5 # number of hidden layers in g (ENCODER) and \varphi (DECODER)
output_dim = 30 # dimension in Koopman space

```

```

[46]: class Encoder(nn.Module):
    def __init__(self, list_layer_dim: list):
        super().__init__()
        self.list_layer_dim = list_layer_dim
        self.list_FC = nn.ModuleList()
        for i in range(len(self.list_layer_dim) - 1):
            input_dim = self.list_layer_dim[i]
            output_dim = self.list_layer_dim[i + 1]
            self.list_FC.append(nn.Linear(input_dim, output_dim))

    def forward(self, tensor2d_x):
        for i in range(len(self.list_layer_dim) - 2):
            tensor2d_x = F.elu(self.list_FC[i](tensor2d_x))
        return self.list_FC[-1](tensor2d_x)

```

```

[47]: class Decoder(nn.Module):
    def __init__(self, list_layer_dim: list):
        super().__init__()
        self.list_layer_dim = list_layer_dim
        self.list_FC = nn.ModuleList()
        for i in range(len(self.list_layer_dim) - 1, 0, -1):
            input_dim = self.list_layer_dim[i]
            output_dim = self.list_layer_dim[i - 1]
            self.list_FC.append(nn.Linear(input_dim, output_dim))

    def forward(self, tensor2d_x: torch.Tensor):

```

```

for i in range(len(self.list_layer_dim) - 2):
    tensor2d_x = F.elu(self.list_FC[i](tensor2d_x))
return self.list_FC[-1](tensor2d_x)

```

```

[48]: class Autoencoder(nn.Module):
    def __init__(self, feature_dim: int, hidden_layer: int, output_dim: int):
        super().__init__()
        list_layer_dim = \
            [output_dim if i == hidden_layer
             else feature_dim + i * (output_dim - feature_dim) // hidden_layer
             for i in range(hidden_layer + 1)]
        self.encoder = Encoder(list_layer_dim)
        self.decoder = Decoder(list_layer_dim)

    def forward(self, tensor2d_x: torch.Tensor):
        tensor2d_x = self.Encoder(tensor2d_x)
        return self.Decoder(tensor2d_x)

```

The Koopman operator $\mathbf{K}$ (which is linear, and thus a matrix) must have a [spectral radius](#) $\rho(\mathbf{K}) \leq 1$. Such condition will provide a stable -or at least a marginally stable- Koopman operator. To fulfill this requirement, we might leverage on the Perron-Frobenius theorem.

The Perron-Frobenius th. states: if $\mathbf{K}$ is a $m \times m$ positive matrix i.e. $k_{ij} > 0$ for $1 \leq i, j \leq m$, then the following inequality holds:

$$\min_i \sum_j k_{ij} \leq \rho(\mathbf{K}) \leq \max_i \sum_j k_{ij}$$

Question 1. : Complete the KoopmanModule class to enforce $\rho(\mathbf{K}) \leq 1$, using the Perron-Frobenius theorem. Check that the initialization fulfills this property.

```

[ ]: class KoopmanOperator(nn.Module):
    def __init__(self, koopman_operator_dim: int):
        super().__init__()
        self.koopman_operator_dim = koopman_operator_dim

        # Initialisation de K_param : matrice aléatoire de dimension (m, m)
        self.K_param = nn.Parameter(torch.randn(koopman_operator_dim,
                                                  koopman_operator_dim) * 0.1)

    def forward(self, tensor2d_x: torch.Tensor):
        # Vérification basique sur la dimension
        if tensor2d_x.shape[1] != self.koopman_operator_dim:
            sys.exit(f'Wrong Input Features. Expected dimension {self.
↪koopman_operator_dim}')

        # Forcer la positivité : chaque élément devient exp(K_param)
        K_pos = torch.exp(self.K_param)

```

```

    # Normalisation
    row_sums = K_pos.sum(dim=1, keepdim=True) + 1e-12 # pour éviter /0
    K_stable = K_pos / row_sums

    # Application de K_stable
    return tensor2d_x @ K_stable

```

```
koopman_operator = KoopmanOperator(output_dim).to(device)
```

```

[50]: with torch.no_grad():
    # Reproduire le K_stable du forward()
    K_param = koopman_operator.K_param
    K_pos = torch.exp(K_param)
    K_stable = K_pos / (K_pos.sum(dim=1, keepdim=True) + 1e-12)

    K_np = K_stable.cpu().numpy()

    # Calcul du rayon spectral de K_stable
    eigvals = np.linalg.eigvals(K_np)
    spectral_radius = np.max(np.abs(eigvals))

    print(f"Spectral radius of initial Koopman operator: {spectral_radius:.5f}")

```

Spectral radius of initial Koopman operator: 1.00000

In this question, we implemented a parametrized version of the Koopman operator while ensuring its stability, i.e., enforcing that its spectral radius satisfies $\rho(\mathbf{K}) \leq 1$.

To achieve this, we leveraged the Perron-Frobenius theorem: by enforcing positivity via the exponential of the parameters and normalizing each row to sum to one, we ensure that the matrix becomes row-stochastic.

As a result, its spectral radius is guaranteed to be at most 1 (and in practice, one eigenvalue is exactly 1, as expected for a row-stochastic matrix).

We also verified this property numerically by computing the spectral radius at initialization.

```

[51]: autoencoder = Autoencoder(feature_dim, hidden_layer, output_dim).to(device)
    koopman_operator = KoopmanOperator(output_dim).to(device)
    print(autoencoder)

```

```

Autoencoder(
  (encoder): Encoder(
    (list_FC): ModuleList(
      (0): Linear(in_features=2, out_features=7, bias=True)
      (1): Linear(in_features=7, out_features=13, bias=True)
      (2): Linear(in_features=13, out_features=18, bias=True)
      (3): Linear(in_features=18, out_features=24, bias=True)
    )
  )
)

```

```

        (4): Linear(in_features=24, out_features=30, bias=True)
    )
)
(decoder): Decoder(
  (list_FC): ModuleList(
    (0): Linear(in_features=30, out_features=24, bias=True)
    (1): Linear(in_features=24, out_features=18, bias=True)
    (2): Linear(in_features=18, out_features=13, bias=True)
    (3): Linear(in_features=13, out_features=7, bias=True)
    (4): Linear(in_features=7, out_features=2, bias=True)
  )
)
)
)

```

```

[52]: learning_rate_autoencoder = 0.0001
learning_rate_koopman = 0.00001

optimiser_autoencoder = torch.optim.Adam(autoencoder.parameters(),
                                          lr=learning_rate_autoencoder)

optimiser_koopman = torch.optim.Adam(koopman_operator.parameters(),
                                     lr=learning_rate_koopman)

```

Question 2. : Define a function to compute the loss to be minimized. It should at least include the 3 terms listed above: - Reconstruction error - Prediction error in the Koopman space - Prediction error in the phase space

Because the different objectives outlined by these losses may compete, the training can be difficult. You may try different variations on these losses and comment your findings. In order to improve the training process, one can for instance: - Add a multiplicative factor in front of each loss component, to balance their importance; how the scales of different losses are related? - We can refine the loss acting upon the latent space, by using a variational autoencoder approach. This is similar to the Gaussian likelihood used in the first practical (TD1). We want the prediction in the latent space (i.e. the Koopman space) to be a normal distribution $\mathcal{N}(0, 1)$. Add a corresponding loss for the latent space. Difference to 0 mean and 1 standard deviation must be thus included in the loss; - Freeze the gradients of one part of the network, for instance the encoder, for one specific objective, using the `requires_grad` property. For instance:

```

criterion = nn.MSELoss()
...
# Compute one part loss_l of the total loss
# First deactivate gradient computation for irrelevant parts of the architecture
for p in autoencoder.encoder.parameters():
    p.requires_grad = False
loss_l = criterion(pred, target)
# Restore the gradient computation
for p in autoencoder.encoder.parameters():
    p.requires_grad = True
...

```

```
total_loss = loss_1 + ... + loss_l + ...
```

```
[53]: # On suppose qu'on utilise la MSE comme critère
mse = nn.MSELoss()

def loss_koopman(
    tensor2d_x: torch.Tensor,          # x_k
    tensor2d_x_next: torch.Tensor,     # x_{k+1}
    tensor2d_decoded_x: torch.Tensor,  # varphi(g(x_k)) = "decoded x"
    tensor2d_observable_next: torch.Tensor, # g(x_{k+1})
    tensor2d_koopman_observable_next: torch.Tensor, # K g(x_k)
    tensor2d_predict_x_next: torch.Tensor, # varphi(K g(x_k)) = "predicted_
    x_{k+1}"
    alpha: float = 1.0,
    beta: float = 1.0,
    gamma: float = 1.0
):
    """
    Calcule la somme pondérée de nos 3 erreurs de base :
    1) Reconstruction error : MSE( decoded_x, x )
    2) Prediction in Koopman : MSE( K*g(x_k), g(x_{k+1}) )
    3) Prediction in phase : MSE( varphi(K*g(x_k)), x_{k+1} )
    Les coefficients alpha, beta, gamma permettent de rééquilibrer.
    """

    # Reconstruction error
    loss_reconstruction = mse(tensor2d_decoded_x, tensor2d_x)

    # Prediction error in Koopman space
    loss_koopman_latent = mse(tensor2d_koopman_observable_next,
                             tensor2d_observable_next)

    # Prediction error in phase space
    loss_prediction_phase = mse(tensor2d_predict_x_next, tensor2d_x_next)

    # Somme pondérée
    total_loss = alpha * loss_reconstruction + beta * loss_koopman_latent +
    gamma * loss_prediction_phase

    return total_loss, loss_reconstruction, loss_koopman_latent,
    loss_prediction_phase
```

1. Reconstruction error

$$\|\varphi_\rho(g_\theta(x_k)) - x_k\| = 0$$

2. Prediction error in Koopman space

$$\|\mathbf{K}_\phi g_\theta(x_k) - g_\theta(x_{k+1})\| = 0$$

3. Prediction error in the phase space

$$\|\varphi_{\rho}(\mathbf{K}_{\phi}g_{\theta}(x_k)) - x_{k+1}\| = 0$$

In this question, we implemented a simple loss function composed of the three main error terms of the model: the reconstruction error, the prediction error in the Koopman (latent) space, and the prediction error in the original phase space.

Each term is measured using mean squared error (MSE), and we introduced three scalar coefficients (α , β , and γ) to **adjust their relative contributions in the total loss**.

This straightforward setup was sufficient in our case : we did not apply any variational regularization nor freeze any part of the network. The weighting factors alone helped stabilize the training and balance the competing objectives effectively.

Question 3.: The following cell executes the training loop. You can modify it in order to display the different intermediate losses computed in the function `LOSS` above. How do they evolve in time? Justify your final choice.

```
[54]: n_batch = len(train_dataloader)
n_epoch = 300

alpha = 0.8
beta = 1.0
gamma = 1.2

train_total_losses = []
train_rec_losses = []
train_lat_losses = []
train_phase_losses = []
test_total_losses = []

for epoch in range(n_epoch):
    autoencoder.train()
    koopman_operator.train()

    running_total = 0.0
    running_rec = 0.0
    running_lat = 0.0
    running_phase = 0.0

    for tensor2d_batch_x, tensor2d_batch_x_next in train_dataloader:
        tensor2d_batch_x = tensor2d_batch_x.to(device)
        tensor2d_batch_x_next = tensor2d_batch_x_next.to(device)

        optimiser_autoencoder.zero_grad()
        optimiser_koopman.zero_grad()

        # Forward
        tensor2d_observable = autoencoder.encoder(tensor2d_batch_x)
```

```

    tensor2d_observable_next = autoencoder.encoder(tensor2d_batch_x_next)
    tensor2d_decoded_x = autoencoder.decoder(tensor2d_observable)
    tensor2d_koopman_observable_next = koopman_operator(tensor2d_observable)
    tensor2d_predict_x_next = autoencoder.
    ↪decoder(tensor2d_koopman_observable_next)

    # Compute loss (returns a tuple)
    loss_val, loss_rec, loss_koop_latent, loss_phase = loss_koopman(
        tensor2d_batch_x,
        tensor2d_batch_x_next,
        tensor2d_decoded_x,
        tensor2d_observable_next,
        tensor2d_koopman_observable_next,
        tensor2d_predict_x_next,
        alpha=alpha,
        beta=beta,
        gamma=gamma
    )

    # Backprop
    loss_val.backward()
    optimiser_autoencoder.step()
    optimiser_koopman.step()

    # Accumuler les valeurs
    running_total += loss_val.item()
    running_rec    += loss_rec.item()
    running_lat    += loss_koop_latent.item()
    running_phase  += loss_phase.item()

    # Moyennes par batch
    epoch_total = running_total / n_batch
    epoch_rec   = running_rec   / n_batch
    epoch_lat   = running_lat   / n_batch
    epoch_phase = running_phase / n_batch

    print(f"[Epoch {epoch}] "
          f"Total={epoch_total:.4f} "
          f"Rec={epoch_rec:.4f} "
          f"Lat={epoch_lat:.4f} "
          f"Phase={epoch_phase:.4f}")

    # Validation/test
    if epoch % 5 == 0:
        autoencoder.eval()
        koopman_operator.eval()
        with torch.no_grad():

```

```

running_test_total = 0.0
running_test_rec    = 0.0
running_test_lat    = 0.0
running_test_phase  = 0.0

n_test_batch = len(test_dataloader)

for tensor2d_batch_x, tensor2d_batch_x_next in test_dataloader:
    tensor2d_batch_x = tensor2d_batch_x.to(device)
    tensor2d_batch_x_next = tensor2d_batch_x_next.to(device)

    # Forward
    obs = autoencoder.encoder(tensor2d_batch_x)
    obs_next = autoencoder.encoder(tensor2d_batch_x_next)
    dec_x = autoencoder.decoder(obs)
    obs_koop_next = koopman_operator(obs)
    pred_x_next = autoencoder.decoder(obs_koop_next)

    # Ici aussi on récupère les 4 composantes
    loss_val_test, loss_rec_test, loss_lat_test, loss_phase_test = _
→loss_koopman(
        tensor2d_batch_x,
        tensor2d_batch_x_next,
        dec_x,
        obs_next,
        obs_koop_next,
        pred_x_next
    )

    running_test_total += loss_val_test.item()
    running_test_rec    += loss_rec_test.item()
    running_test_lat    += loss_lat_test.item()
    running_test_phase += loss_phase_test.item()

    # Moyennes de test
    test_total = running_test_total / n_test_batch
    test_rec    = running_test_rec    / n_test_batch
    test_lat    = running_test_lat    / n_test_batch
    test_phase  = running_test_phase / n_test_batch

    print(f"    [Test at epoch {epoch}] "
          f"Total={test_total:.4f} "
          f"Rec={test_rec:.4f} "
          f"Lat={test_lat:.4f} "
          f"Phase={test_phase:.4f}")

train_total_losses.append(epoch_total)

```

```

train_rec_losses.append(epoch_rec)
train_lat_losses.append(epoch_lat)
train_phase_losses.append(epoch_phase)
test_total_losses.append(test_total)  # seulement si epoch % X == 0

```

```

[Epoch 0] Total=3.2815 Rec=1.6298 Lat=0.0147 Phase=1.6359
[Test at epoch 0] Total=3.2341 Rec=1.5979 Lat=0.0106 Phase=1.6256
[Epoch 1] Total=3.0601 Rec=1.4433 Lat=0.0159 Phase=1.5747
[Epoch 2] Total=1.9779 Rec=0.9143 Lat=0.0367 Phase=1.0081
[Epoch 3] Total=1.6585 Rec=0.8045 Lat=0.0239 Phase=0.8258
[Epoch 4] Total=1.5780 Rec=0.7695 Lat=0.0132 Phase=0.7910
[Epoch 5] Total=1.5114 Rec=0.7412 Lat=0.0078 Phase=0.7589
[Test at epoch 5] Total=1.4791 Rec=0.7291 Lat=0.0060 Phase=0.7440
[Epoch 6] Total=1.4570 Rec=0.7188 Lat=0.0049 Phase=0.7309
[Epoch 7] Total=1.4186 Rec=0.7026 Lat=0.0035 Phase=0.7109
[Epoch 8] Total=1.3906 Rec=0.6894 Lat=0.0028 Phase=0.6969
[Epoch 9] Total=1.3623 Rec=0.6745 Lat=0.0026 Phase=0.6834
[Epoch 10] Total=1.3212 Rec=0.6500 Lat=0.0031 Phase=0.6651
[Test at epoch 10] Total=1.2892 Rec=0.6317 Lat=0.0039 Phase=0.6536
[Epoch 11] Total=1.2368 Rec=0.5890 Lat=0.0059 Phase=0.6330
[Epoch 12] Total=1.0521 Rec=0.4210 Lat=0.0171 Phase=0.5819
[Epoch 13] Total=0.9002 Rec=0.2778 Lat=0.0266 Phase=0.5428
[Epoch 14] Total=0.8434 Rec=0.2399 Lat=0.0229 Phase=0.5238
[Epoch 15] Total=0.8072 Rec=0.2205 Lat=0.0188 Phase=0.5100
[Test at epoch 15] Total=0.7372 Rec=0.2143 Lat=0.0176 Phase=0.5054
[Epoch 16] Total=0.7780 Rec=0.2056 Lat=0.0165 Phase=0.4975
[Epoch 17] Total=0.7512 Rec=0.1920 Lat=0.0154 Phase=0.4852
[Epoch 18] Total=0.7232 Rec=0.1765 Lat=0.0152 Phase=0.4724
[Epoch 19] Total=0.6909 Rec=0.1565 Lat=0.0157 Phase=0.4583
[Epoch 20] Total=0.6514 Rec=0.1302 Lat=0.0174 Phase=0.4416
[Test at epoch 20] Total=0.5680 Rec=0.1160 Lat=0.0188 Phase=0.4332
[Epoch 21] Total=0.6057 Rec=0.1004 Lat=0.0205 Phase=0.4207
[Epoch 22] Total=0.5591 Rec=0.0770 Lat=0.0254 Phase=0.3934
[Epoch 23] Total=0.5105 Rec=0.0602 Lat=0.0322 Phase=0.3585
[Epoch 24] Total=0.4520 Rec=0.0481 Lat=0.0430 Phase=0.3087
[Epoch 25] Total=0.3721 Rec=0.0389 Lat=0.0592 Phase=0.2348
[Test at epoch 25] Total=0.2959 Rec=0.0358 Lat=0.0684 Phase=0.1917
[Epoch 26] Total=0.2795 Rec=0.0332 Lat=0.0709 Phase=0.1517
[Epoch 27] Total=0.2013 Rec=0.0279 Lat=0.0630 Phase=0.0967
[Epoch 28] Total=0.1444 Rec=0.0221 Lat=0.0501 Phase=0.0639
[Epoch 29] Total=0.1076 Rec=0.0177 Lat=0.0397 Phase=0.0448
[Epoch 30] Total=0.0848 Rec=0.0142 Lat=0.0321 Phase=0.0344
[Test at epoch 30] Total=0.0729 Rec=0.0129 Lat=0.0292 Phase=0.0308
[Epoch 31] Total=0.0694 Rec=0.0115 Lat=0.0267 Phase=0.0279
[Epoch 32] Total=0.0582 Rec=0.0094 Lat=0.0227 Phase=0.0233
[Epoch 33] Total=0.0497 Rec=0.0079 Lat=0.0197 Phase=0.0198
[Epoch 34] Total=0.0432 Rec=0.0068 Lat=0.0174 Phase=0.0170
[Epoch 35] Total=0.0381 Rec=0.0060 Lat=0.0155 Phase=0.0148

```

[Test at epoch 35] Total=0.0344 Rec=0.0058 Lat=0.0147 Phase=0.0138
 [Epoch 36] Total=0.0338 Rec=0.0054 Lat=0.0139 Phase=0.0130
 [Epoch 37] Total=0.0303 Rec=0.0050 Lat=0.0126 Phase=0.0114
 [Epoch 38] Total=0.0275 Rec=0.0047 Lat=0.0116 Phase=0.0101
 [Epoch 39] Total=0.0250 Rec=0.0044 Lat=0.0106 Phase=0.0090
 [Epoch 40] Total=0.0229 Rec=0.0041 Lat=0.0098 Phase=0.0081
 [Test at epoch 40] Total=0.0213 Rec=0.0041 Lat=0.0095 Phase=0.0077
 [Epoch 41] Total=0.0211 Rec=0.0040 Lat=0.0091 Phase=0.0074
 [Epoch 42] Total=0.0196 Rec=0.0039 Lat=0.0085 Phase=0.0067
 [Epoch 43] Total=0.0183 Rec=0.0038 Lat=0.0079 Phase=0.0062
 [Epoch 44] Total=0.0172 Rec=0.0036 Lat=0.0074 Phase=0.0057
 [Epoch 45] Total=0.0162 Rec=0.0035 Lat=0.0070 Phase=0.0054
 [Test at epoch 45] Total=0.0154 Rec=0.0034 Lat=0.0068 Phase=0.0052
 [Epoch 46] Total=0.0154 Rec=0.0035 Lat=0.0065 Phase=0.0050
 [Epoch 47] Total=0.0146 Rec=0.0034 Lat=0.0062 Phase=0.0047
 [Epoch 48] Total=0.0139 Rec=0.0034 Lat=0.0058 Phase=0.0045
 [Epoch 49] Total=0.0133 Rec=0.0033 Lat=0.0055 Phase=0.0043
 [Epoch 50] Total=0.0128 Rec=0.0033 Lat=0.0053 Phase=0.0041
 [Test at epoch 50] Total=0.0127 Rec=0.0036 Lat=0.0051 Phase=0.0040
 [Epoch 51] Total=0.0123 Rec=0.0032 Lat=0.0050 Phase=0.0039
 [Epoch 52] Total=0.0118 Rec=0.0032 Lat=0.0048 Phase=0.0038
 [Epoch 53] Total=0.0114 Rec=0.0032 Lat=0.0045 Phase=0.0036
 [Epoch 54] Total=0.0110 Rec=0.0031 Lat=0.0043 Phase=0.0035
 [Epoch 55] Total=0.0107 Rec=0.0031 Lat=0.0041 Phase=0.0034
 [Test at epoch 55] Total=0.0104 Rec=0.0030 Lat=0.0041 Phase=0.0033
 [Epoch 56] Total=0.0104 Rec=0.0032 Lat=0.0040 Phase=0.0032
 [Epoch 57] Total=0.0100 Rec=0.0030 Lat=0.0038 Phase=0.0031
 [Epoch 58] Total=0.0098 Rec=0.0031 Lat=0.0037 Phase=0.0031
 [Epoch 59] Total=0.0095 Rec=0.0031 Lat=0.0035 Phase=0.0030
 [Epoch 60] Total=0.0092 Rec=0.0030 Lat=0.0034 Phase=0.0029
 [Test at epoch 60] Total=0.0091 Rec=0.0030 Lat=0.0033 Phase=0.0028
 [Epoch 61] Total=0.0091 Rec=0.0031 Lat=0.0033 Phase=0.0028
 [Epoch 62] Total=0.0088 Rec=0.0030 Lat=0.0032 Phase=0.0027
 [Epoch 63] Total=0.0086 Rec=0.0030 Lat=0.0031 Phase=0.0027
 [Epoch 64] Total=0.0084 Rec=0.0030 Lat=0.0029 Phase=0.0026
 [Epoch 65] Total=0.0083 Rec=0.0029 Lat=0.0029 Phase=0.0025
 [Test at epoch 65] Total=0.0083 Rec=0.0030 Lat=0.0028 Phase=0.0025
 [Epoch 66] Total=0.0081 Rec=0.0029 Lat=0.0028 Phase=0.0025
 [Epoch 67] Total=0.0080 Rec=0.0029 Lat=0.0027 Phase=0.0024
 [Epoch 68] Total=0.0078 Rec=0.0029 Lat=0.0026 Phase=0.0024
 [Epoch 69] Total=0.0077 Rec=0.0029 Lat=0.0025 Phase=0.0023
 [Epoch 70] Total=0.0075 Rec=0.0028 Lat=0.0025 Phase=0.0023
 [Test at epoch 70] Total=0.0076 Rec=0.0029 Lat=0.0024 Phase=0.0023
 [Epoch 71] Total=0.0074 Rec=0.0029 Lat=0.0024 Phase=0.0023
 [Epoch 72] Total=0.0073 Rec=0.0029 Lat=0.0023 Phase=0.0022
 [Epoch 73] Total=0.0071 Rec=0.0028 Lat=0.0023 Phase=0.0022
 [Epoch 74] Total=0.0070 Rec=0.0028 Lat=0.0022 Phase=0.0021
 [Epoch 75] Total=0.0070 Rec=0.0029 Lat=0.0022 Phase=0.0021

[Test at epoch 75] Total=0.0077 Rec=0.0035 Lat=0.0022 Phase=0.0021
 [Epoch 76] Total=0.0068 Rec=0.0027 Lat=0.0021 Phase=0.0021
 [Epoch 77] Total=0.0067 Rec=0.0028 Lat=0.0021 Phase=0.0020
 [Epoch 78] Total=0.0066 Rec=0.0028 Lat=0.0020 Phase=0.0020
 [Epoch 79] Total=0.0066 Rec=0.0028 Lat=0.0020 Phase=0.0019
 [Epoch 80] Total=0.0064 Rec=0.0027 Lat=0.0020 Phase=0.0019
 [Test at epoch 80] Total=0.0067 Rec=0.0029 Lat=0.0019 Phase=0.0019
 [Epoch 81] Total=0.0064 Rec=0.0028 Lat=0.0019 Phase=0.0019
 [Epoch 82] Total=0.0063 Rec=0.0027 Lat=0.0019 Phase=0.0018
 [Epoch 83] Total=0.0062 Rec=0.0027 Lat=0.0018 Phase=0.0018
 [Epoch 84] Total=0.0062 Rec=0.0028 Lat=0.0018 Phase=0.0018
 [Epoch 85] Total=0.0061 Rec=0.0027 Lat=0.0018 Phase=0.0018
 [Test at epoch 85] Total=0.0064 Rec=0.0029 Lat=0.0018 Phase=0.0017
 [Epoch 86] Total=0.0059 Rec=0.0026 Lat=0.0018 Phase=0.0017
 [Epoch 87] Total=0.0059 Rec=0.0027 Lat=0.0017 Phase=0.0017
 [Epoch 88] Total=0.0059 Rec=0.0027 Lat=0.0017 Phase=0.0017
 [Epoch 89] Total=0.0057 Rec=0.0026 Lat=0.0017 Phase=0.0016
 [Epoch 90] Total=0.0057 Rec=0.0027 Lat=0.0017 Phase=0.0016
 [Test at epoch 90] Total=0.0058 Rec=0.0025 Lat=0.0016 Phase=0.0016
 [Epoch 91] Total=0.0057 Rec=0.0027 Lat=0.0016 Phase=0.0016
 [Epoch 92] Total=0.0055 Rec=0.0026 Lat=0.0016 Phase=0.0016
 [Epoch 93] Total=0.0056 Rec=0.0028 Lat=0.0016 Phase=0.0015
 [Epoch 94] Total=0.0053 Rec=0.0024 Lat=0.0016 Phase=0.0015
 [Epoch 95] Total=0.0054 Rec=0.0025 Lat=0.0016 Phase=0.0015
 [Test at epoch 95] Total=0.0054 Rec=0.0024 Lat=0.0015 Phase=0.0015
 [Epoch 96] Total=0.0053 Rec=0.0025 Lat=0.0015 Phase=0.0015
 [Epoch 97] Total=0.0052 Rec=0.0025 Lat=0.0015 Phase=0.0014
 [Epoch 98] Total=0.0053 Rec=0.0026 Lat=0.0015 Phase=0.0014
 [Epoch 99] Total=0.0051 Rec=0.0025 Lat=0.0015 Phase=0.0014
 [Epoch 100] Total=0.0051 Rec=0.0025 Lat=0.0015 Phase=0.0014
 [Test at epoch 100] Total=0.0054 Rec=0.0026 Lat=0.0015 Phase=0.0013
 [Epoch 101] Total=0.0050 Rec=0.0024 Lat=0.0015 Phase=0.0013
 [Epoch 102] Total=0.0050 Rec=0.0024 Lat=0.0015 Phase=0.0013
 [Epoch 103] Total=0.0050 Rec=0.0026 Lat=0.0014 Phase=0.0013
 [Epoch 104] Total=0.0048 Rec=0.0023 Lat=0.0014 Phase=0.0013
 [Epoch 105] Total=0.0047 Rec=0.0023 Lat=0.0014 Phase=0.0012
 [Test at epoch 105] Total=0.0049 Rec=0.0023 Lat=0.0014 Phase=0.0012
 [Epoch 106] Total=0.0047 Rec=0.0024 Lat=0.0014 Phase=0.0012
 [Epoch 107] Total=0.0047 Rec=0.0023 Lat=0.0014 Phase=0.0012
 [Epoch 108] Total=0.0046 Rec=0.0023 Lat=0.0014 Phase=0.0012
 [Epoch 109] Total=0.0046 Rec=0.0023 Lat=0.0014 Phase=0.0011
 [Epoch 110] Total=0.0044 Rec=0.0022 Lat=0.0014 Phase=0.0011
 [Test at epoch 110] Total=0.0054 Rec=0.0029 Lat=0.0014 Phase=0.0011
 [Epoch 111] Total=0.0044 Rec=0.0021 Lat=0.0014 Phase=0.0011
 [Epoch 112] Total=0.0044 Rec=0.0022 Lat=0.0014 Phase=0.0011
 [Epoch 113] Total=0.0043 Rec=0.0021 Lat=0.0014 Phase=0.0010
 [Epoch 114] Total=0.0042 Rec=0.0021 Lat=0.0014 Phase=0.0010
 [Epoch 115] Total=0.0042 Rec=0.0021 Lat=0.0014 Phase=0.0010

[Test at epoch 115] Total=0.0043 Rec=0.0020 Lat=0.0014 Phase=0.0010
 [Epoch 116] Total=0.0041 Rec=0.0020 Lat=0.0014 Phase=0.0010
 [Epoch 117] Total=0.0041 Rec=0.0020 Lat=0.0013 Phase=0.0009
 [Epoch 118] Total=0.0041 Rec=0.0020 Lat=0.0013 Phase=0.0009
 [Epoch 119] Total=0.0039 Rec=0.0018 Lat=0.0013 Phase=0.0009
 [Epoch 120] Total=0.0040 Rec=0.0020 Lat=0.0013 Phase=0.0009
 [Test at epoch 120] Total=0.0039 Rec=0.0017 Lat=0.0013 Phase=0.0009
 [Epoch 121] Total=0.0038 Rec=0.0018 Lat=0.0013 Phase=0.0008
 [Epoch 122] Total=0.0038 Rec=0.0019 Lat=0.0013 Phase=0.0008
 [Epoch 123] Total=0.0037 Rec=0.0017 Lat=0.0013 Phase=0.0008
 [Epoch 124] Total=0.0037 Rec=0.0018 Lat=0.0013 Phase=0.0008
 [Epoch 125] Total=0.0036 Rec=0.0017 Lat=0.0013 Phase=0.0008
 [Test at epoch 125] Total=0.0037 Rec=0.0016 Lat=0.0013 Phase=0.0007
 [Epoch 126] Total=0.0035 Rec=0.0016 Lat=0.0013 Phase=0.0007
 [Epoch 127] Total=0.0036 Rec=0.0017 Lat=0.0013 Phase=0.0007
 [Epoch 128] Total=0.0035 Rec=0.0017 Lat=0.0013 Phase=0.0007
 [Epoch 129] Total=0.0034 Rec=0.0015 Lat=0.0013 Phase=0.0007
 [Epoch 130] Total=0.0033 Rec=0.0015 Lat=0.0013 Phase=0.0007
 [Test at epoch 130] Total=0.0033 Rec=0.0013 Lat=0.0013 Phase=0.0007
 [Epoch 131] Total=0.0033 Rec=0.0015 Lat=0.0013 Phase=0.0007
 [Epoch 132] Total=0.0033 Rec=0.0016 Lat=0.0013 Phase=0.0006
 [Epoch 133] Total=0.0032 Rec=0.0014 Lat=0.0013 Phase=0.0006
 [Epoch 134] Total=0.0032 Rec=0.0015 Lat=0.0013 Phase=0.0006
 [Epoch 135] Total=0.0031 Rec=0.0014 Lat=0.0013 Phase=0.0006
 [Test at epoch 135] Total=0.0030 Rec=0.0012 Lat=0.0013 Phase=0.0006
 [Epoch 136] Total=0.0030 Rec=0.0013 Lat=0.0013 Phase=0.0006
 [Epoch 137] Total=0.0031 Rec=0.0014 Lat=0.0013 Phase=0.0006
 [Epoch 138] Total=0.0030 Rec=0.0014 Lat=0.0013 Phase=0.0006
 [Epoch 139] Total=0.0029 Rec=0.0012 Lat=0.0013 Phase=0.0005
 [Epoch 140] Total=0.0030 Rec=0.0013 Lat=0.0013 Phase=0.0005
 [Test at epoch 140] Total=0.0028 Rec=0.0010 Lat=0.0013 Phase=0.0005
 [Epoch 141] Total=0.0028 Rec=0.0012 Lat=0.0013 Phase=0.0005
 [Epoch 142] Total=0.0029 Rec=0.0012 Lat=0.0013 Phase=0.0005
 [Epoch 143] Total=0.0029 Rec=0.0013 Lat=0.0012 Phase=0.0005
 [Epoch 144] Total=0.0028 Rec=0.0012 Lat=0.0012 Phase=0.0005
 [Epoch 145] Total=0.0028 Rec=0.0012 Lat=0.0012 Phase=0.0005
 [Test at epoch 145] Total=0.0032 Rec=0.0015 Lat=0.0012 Phase=0.0005
 [Epoch 146] Total=0.0026 Rec=0.0010 Lat=0.0012 Phase=0.0005
 [Epoch 147] Total=0.0027 Rec=0.0012 Lat=0.0012 Phase=0.0005
 [Epoch 148] Total=0.0027 Rec=0.0012 Lat=0.0012 Phase=0.0004
 [Epoch 149] Total=0.0026 Rec=0.0011 Lat=0.0012 Phase=0.0004
 [Epoch 150] Total=0.0025 Rec=0.0010 Lat=0.0012 Phase=0.0004
 [Test at epoch 150] Total=0.0032 Rec=0.0016 Lat=0.0012 Phase=0.0004
 [Epoch 151] Total=0.0026 Rec=0.0011 Lat=0.0012 Phase=0.0004
 [Epoch 152] Total=0.0025 Rec=0.0010 Lat=0.0012 Phase=0.0004
 [Epoch 153] Total=0.0025 Rec=0.0011 Lat=0.0012 Phase=0.0004
 [Epoch 154] Total=0.0024 Rec=0.0009 Lat=0.0012 Phase=0.0004
 [Epoch 155] Total=0.0025 Rec=0.0011 Lat=0.0012 Phase=0.0004

[Test at epoch 155] Total=0.0025 Rec=0.0010 Lat=0.0012 Phase=0.0004
 [Epoch 156] Total=0.0025 Rec=0.0010 Lat=0.0012 Phase=0.0004
 [Epoch 157] Total=0.0024 Rec=0.0009 Lat=0.0012 Phase=0.0004
 [Epoch 158] Total=0.0024 Rec=0.0010 Lat=0.0012 Phase=0.0004
 [Epoch 159] Total=0.0024 Rec=0.0010 Lat=0.0011 Phase=0.0004
 [Epoch 160] Total=0.0024 Rec=0.0010 Lat=0.0011 Phase=0.0004
 [Test at epoch 160] Total=0.0024 Rec=0.0009 Lat=0.0011 Phase=0.0004
 [Epoch 161] Total=0.0024 Rec=0.0011 Lat=0.0011 Phase=0.0004
 [Epoch 162] Total=0.0022 Rec=0.0008 Lat=0.0011 Phase=0.0003
 [Epoch 163] Total=0.0024 Rec=0.0011 Lat=0.0011 Phase=0.0003
 [Epoch 164] Total=0.0023 Rec=0.0010 Lat=0.0011 Phase=0.0003
 [Epoch 165] Total=0.0021 Rec=0.0008 Lat=0.0011 Phase=0.0003
 [Test at epoch 165] Total=0.0026 Rec=0.0012 Lat=0.0011 Phase=0.0003
 [Epoch 166] Total=0.0023 Rec=0.0009 Lat=0.0011 Phase=0.0003
 [Epoch 167] Total=0.0022 Rec=0.0009 Lat=0.0011 Phase=0.0003
 [Epoch 168] Total=0.0022 Rec=0.0009 Lat=0.0011 Phase=0.0003
 [Epoch 169] Total=0.0022 Rec=0.0009 Lat=0.0011 Phase=0.0003
 [Epoch 170] Total=0.0021 Rec=0.0008 Lat=0.0011 Phase=0.0003
 [Test at epoch 170] Total=0.0021 Rec=0.0007 Lat=0.0011 Phase=0.0003
 [Epoch 171] Total=0.0023 Rec=0.0011 Lat=0.0011 Phase=0.0003
 [Epoch 172] Total=0.0020 Rec=0.0007 Lat=0.0011 Phase=0.0003
 [Epoch 173] Total=0.0021 Rec=0.0009 Lat=0.0011 Phase=0.0003
 [Epoch 174] Total=0.0021 Rec=0.0009 Lat=0.0011 Phase=0.0003
 [Epoch 175] Total=0.0020 Rec=0.0008 Lat=0.0010 Phase=0.0003
 [Test at epoch 175] Total=0.0031 Rec=0.0018 Lat=0.0010 Phase=0.0003
 [Epoch 176] Total=0.0021 Rec=0.0009 Lat=0.0010 Phase=0.0003
 [Epoch 177] Total=0.0021 Rec=0.0009 Lat=0.0010 Phase=0.0003
 [Epoch 178] Total=0.0020 Rec=0.0008 Lat=0.0010 Phase=0.0003
 [Epoch 179] Total=0.0020 Rec=0.0008 Lat=0.0010 Phase=0.0003
 [Epoch 180] Total=0.0020 Rec=0.0008 Lat=0.0010 Phase=0.0003
 [Test at epoch 180] Total=0.0019 Rec=0.0006 Lat=0.0010 Phase=0.0003
 [Epoch 181] Total=0.0020 Rec=0.0009 Lat=0.0010 Phase=0.0003
 [Epoch 182] Total=0.0019 Rec=0.0008 Lat=0.0010 Phase=0.0003
 [Epoch 183] Total=0.0020 Rec=0.0008 Lat=0.0010 Phase=0.0003
 [Epoch 184] Total=0.0020 Rec=0.0009 Lat=0.0010 Phase=0.0003
 [Epoch 185] Total=0.0019 Rec=0.0007 Lat=0.0010 Phase=0.0003
 [Test at epoch 185] Total=0.0017 Rec=0.0005 Lat=0.0010 Phase=0.0003
 [Epoch 186] Total=0.0019 Rec=0.0008 Lat=0.0010 Phase=0.0003
 [Epoch 187] Total=0.0019 Rec=0.0008 Lat=0.0010 Phase=0.0003
 [Epoch 188] Total=0.0019 Rec=0.0008 Lat=0.0010 Phase=0.0003
 [Epoch 189] Total=0.0019 Rec=0.0008 Lat=0.0010 Phase=0.0002
 [Epoch 190] Total=0.0019 Rec=0.0007 Lat=0.0010 Phase=0.0002
 [Test at epoch 190] Total=0.0020 Rec=0.0008 Lat=0.0010 Phase=0.0002
 [Epoch 191] Total=0.0018 Rec=0.0007 Lat=0.0010 Phase=0.0002
 [Epoch 192] Total=0.0019 Rec=0.0009 Lat=0.0009 Phase=0.0002
 [Epoch 193] Total=0.0019 Rec=0.0008 Lat=0.0009 Phase=0.0002
 [Epoch 194] Total=0.0018 Rec=0.0007 Lat=0.0009 Phase=0.0002
 [Epoch 195] Total=0.0019 Rec=0.0009 Lat=0.0009 Phase=0.0002

[Test at epoch 195] Total=0.0016 Rec=0.0005 Lat=0.0009 Phase=0.0002
 [Epoch 196] Total=0.0018 Rec=0.0007 Lat=0.0009 Phase=0.0002
 [Epoch 197] Total=0.0019 Rec=0.0009 Lat=0.0009 Phase=0.0002
 [Epoch 198] Total=0.0017 Rec=0.0007 Lat=0.0009 Phase=0.0002
 [Epoch 199] Total=0.0018 Rec=0.0008 Lat=0.0009 Phase=0.0002
 [Epoch 200] Total=0.0017 Rec=0.0007 Lat=0.0009 Phase=0.0002
 [Test at epoch 200] Total=0.0016 Rec=0.0005 Lat=0.0009 Phase=0.0002
 [Epoch 201] Total=0.0018 Rec=0.0007 Lat=0.0009 Phase=0.0002
 [Epoch 202] Total=0.0018 Rec=0.0008 Lat=0.0009 Phase=0.0002
 [Epoch 203] Total=0.0017 Rec=0.0007 Lat=0.0009 Phase=0.0002
 [Epoch 204] Total=0.0019 Rec=0.0009 Lat=0.0009 Phase=0.0002
 [Epoch 205] Total=0.0018 Rec=0.0008 Lat=0.0009 Phase=0.0002
 [Test at epoch 205] Total=0.0017 Rec=0.0006 Lat=0.0009 Phase=0.0002
 [Epoch 206] Total=0.0016 Rec=0.0005 Lat=0.0009 Phase=0.0002
 [Epoch 207] Total=0.0017 Rec=0.0007 Lat=0.0009 Phase=0.0002
 [Epoch 208] Total=0.0018 Rec=0.0008 Lat=0.0009 Phase=0.0002
 [Epoch 209] Total=0.0017 Rec=0.0007 Lat=0.0009 Phase=0.0002
 [Epoch 210] Total=0.0017 Rec=0.0007 Lat=0.0009 Phase=0.0002
 [Test at epoch 210] Total=0.0015 Rec=0.0004 Lat=0.0009 Phase=0.0002
 [Epoch 211] Total=0.0016 Rec=0.0006 Lat=0.0009 Phase=0.0002
 [Epoch 212] Total=0.0017 Rec=0.0007 Lat=0.0009 Phase=0.0002
 [Epoch 213] Total=0.0017 Rec=0.0008 Lat=0.0009 Phase=0.0002
 [Epoch 214] Total=0.0016 Rec=0.0007 Lat=0.0009 Phase=0.0002
 [Epoch 215] Total=0.0017 Rec=0.0007 Lat=0.0008 Phase=0.0002
 [Test at epoch 215] Total=0.0019 Rec=0.0009 Lat=0.0008 Phase=0.0002
 [Epoch 216] Total=0.0017 Rec=0.0008 Lat=0.0008 Phase=0.0002
 [Epoch 217] Total=0.0016 Rec=0.0006 Lat=0.0008 Phase=0.0002
 [Epoch 218] Total=0.0017 Rec=0.0008 Lat=0.0008 Phase=0.0002
 [Epoch 219] Total=0.0016 Rec=0.0006 Lat=0.0008 Phase=0.0002
 [Epoch 220] Total=0.0016 Rec=0.0007 Lat=0.0008 Phase=0.0002
 [Test at epoch 220] Total=0.0015 Rec=0.0005 Lat=0.0008 Phase=0.0002
 [Epoch 221] Total=0.0016 Rec=0.0007 Lat=0.0008 Phase=0.0002
 [Epoch 222] Total=0.0016 Rec=0.0007 Lat=0.0008 Phase=0.0002
 [Epoch 223] Total=0.0015 Rec=0.0006 Lat=0.0008 Phase=0.0002
 [Epoch 224] Total=0.0017 Rec=0.0008 Lat=0.0008 Phase=0.0002
 [Epoch 225] Total=0.0015 Rec=0.0006 Lat=0.0008 Phase=0.0002
 [Test at epoch 225] Total=0.0016 Rec=0.0006 Lat=0.0008 Phase=0.0002
 [Epoch 226] Total=0.0017 Rec=0.0008 Lat=0.0008 Phase=0.0002
 [Epoch 227] Total=0.0017 Rec=0.0009 Lat=0.0008 Phase=0.0002
 [Epoch 228] Total=0.0014 Rec=0.0005 Lat=0.0008 Phase=0.0002
 [Epoch 229] Total=0.0015 Rec=0.0006 Lat=0.0008 Phase=0.0002
 [Epoch 230] Total=0.0016 Rec=0.0007 Lat=0.0008 Phase=0.0002
 [Test at epoch 230] Total=0.0014 Rec=0.0005 Lat=0.0008 Phase=0.0002
 [Epoch 231] Total=0.0015 Rec=0.0007 Lat=0.0008 Phase=0.0002
 [Epoch 232] Total=0.0016 Rec=0.0007 Lat=0.0008 Phase=0.0002
 [Epoch 233] Total=0.0016 Rec=0.0007 Lat=0.0008 Phase=0.0002
 [Epoch 234] Total=0.0015 Rec=0.0006 Lat=0.0008 Phase=0.0002
 [Epoch 235] Total=0.0015 Rec=0.0006 Lat=0.0008 Phase=0.0002

[Test at epoch 235] Total=0.0014 Rec=0.0004 Lat=0.0008 Phase=0.0002
 [Epoch 236] Total=0.0016 Rec=0.0008 Lat=0.0008 Phase=0.0002
 [Epoch 237] Total=0.0015 Rec=0.0007 Lat=0.0008 Phase=0.0002
 [Epoch 238] Total=0.0015 Rec=0.0006 Lat=0.0008 Phase=0.0002
 [Epoch 239] Total=0.0015 Rec=0.0007 Lat=0.0008 Phase=0.0002
 [Epoch 240] Total=0.0015 Rec=0.0007 Lat=0.0008 Phase=0.0002
 [Test at epoch 240] Total=0.0013 Rec=0.0003 Lat=0.0008 Phase=0.0002
 [Epoch 241] Total=0.0015 Rec=0.0007 Lat=0.0008 Phase=0.0002
 [Epoch 242] Total=0.0014 Rec=0.0006 Lat=0.0008 Phase=0.0002
 [Epoch 243] Total=0.0016 Rec=0.0008 Lat=0.0008 Phase=0.0002
 [Epoch 244] Total=0.0016 Rec=0.0008 Lat=0.0007 Phase=0.0002
 [Epoch 245] Total=0.0014 Rec=0.0006 Lat=0.0007 Phase=0.0002
 [Test at epoch 245] Total=0.0013 Rec=0.0004 Lat=0.0007 Phase=0.0002
 [Epoch 246] Total=0.0015 Rec=0.0007 Lat=0.0007 Phase=0.0002
 [Epoch 247] Total=0.0016 Rec=0.0008 Lat=0.0007 Phase=0.0002
 [Epoch 248] Total=0.0013 Rec=0.0004 Lat=0.0007 Phase=0.0002
 [Epoch 249] Total=0.0016 Rec=0.0009 Lat=0.0007 Phase=0.0002
 [Epoch 250] Total=0.0013 Rec=0.0004 Lat=0.0007 Phase=0.0002
 [Test at epoch 250] Total=0.0012 Rec=0.0003 Lat=0.0007 Phase=0.0002
 [Epoch 251] Total=0.0014 Rec=0.0006 Lat=0.0007 Phase=0.0002
 [Epoch 252] Total=0.0015 Rec=0.0007 Lat=0.0007 Phase=0.0002
 [Epoch 253] Total=0.0014 Rec=0.0006 Lat=0.0007 Phase=0.0002
 [Epoch 254] Total=0.0014 Rec=0.0006 Lat=0.0007 Phase=0.0002
 [Epoch 255] Total=0.0014 Rec=0.0007 Lat=0.0007 Phase=0.0002
 [Test at epoch 255] Total=0.0016 Rec=0.0008 Lat=0.0007 Phase=0.0002
 [Epoch 256] Total=0.0015 Rec=0.0007 Lat=0.0007 Phase=0.0002
 [Epoch 257] Total=0.0014 Rec=0.0006 Lat=0.0007 Phase=0.0002
 [Epoch 258] Total=0.0014 Rec=0.0006 Lat=0.0007 Phase=0.0002
 [Epoch 259] Total=0.0014 Rec=0.0006 Lat=0.0007 Phase=0.0002
 [Epoch 260] Total=0.0014 Rec=0.0006 Lat=0.0007 Phase=0.0002
 [Test at epoch 260] Total=0.0013 Rec=0.0005 Lat=0.0007 Phase=0.0002
 [Epoch 261] Total=0.0015 Rec=0.0007 Lat=0.0007 Phase=0.0002
 [Epoch 262] Total=0.0013 Rec=0.0006 Lat=0.0007 Phase=0.0002
 [Epoch 263] Total=0.0015 Rec=0.0007 Lat=0.0007 Phase=0.0002
 [Epoch 264] Total=0.0015 Rec=0.0008 Lat=0.0007 Phase=0.0002
 [Epoch 265] Total=0.0012 Rec=0.0005 Lat=0.0007 Phase=0.0001
 [Test at epoch 265] Total=0.0019 Rec=0.0010 Lat=0.0007 Phase=0.0002
 [Epoch 266] Total=0.0014 Rec=0.0006 Lat=0.0007 Phase=0.0002
 [Epoch 267] Total=0.0014 Rec=0.0007 Lat=0.0007 Phase=0.0002
 [Epoch 268] Total=0.0014 Rec=0.0007 Lat=0.0007 Phase=0.0002
 [Epoch 269] Total=0.0014 Rec=0.0006 Lat=0.0007 Phase=0.0002
 [Epoch 270] Total=0.0013 Rec=0.0005 Lat=0.0007 Phase=0.0001
 [Test at epoch 270] Total=0.0012 Rec=0.0003 Lat=0.0007 Phase=0.0001
 [Epoch 271] Total=0.0014 Rec=0.0007 Lat=0.0007 Phase=0.0002
 [Epoch 272] Total=0.0014 Rec=0.0007 Lat=0.0007 Phase=0.0002
 [Epoch 273] Total=0.0012 Rec=0.0005 Lat=0.0007 Phase=0.0001
 [Epoch 274] Total=0.0013 Rec=0.0006 Lat=0.0007 Phase=0.0001
 [Epoch 275] Total=0.0014 Rec=0.0007 Lat=0.0007 Phase=0.0001

```

[Test at epoch 275] Total=0.0016 Rec=0.0008 Lat=0.0007 Phase=0.0002
[Epoch 276] Total=0.0012 Rec=0.0005 Lat=0.0007 Phase=0.0001
[Epoch 277] Total=0.0014 Rec=0.0006 Lat=0.0007 Phase=0.0001
[Epoch 278] Total=0.0014 Rec=0.0007 Lat=0.0007 Phase=0.0001
[Epoch 279] Total=0.0014 Rec=0.0007 Lat=0.0007 Phase=0.0001
[Epoch 280] Total=0.0013 Rec=0.0006 Lat=0.0007 Phase=0.0001
[Test at epoch 280] Total=0.0012 Rec=0.0004 Lat=0.0007 Phase=0.0001
[Epoch 281] Total=0.0013 Rec=0.0006 Lat=0.0007 Phase=0.0001
[Epoch 282] Total=0.0013 Rec=0.0006 Lat=0.0007 Phase=0.0001
[Epoch 283] Total=0.0013 Rec=0.0006 Lat=0.0007 Phase=0.0001
[Epoch 284] Total=0.0013 Rec=0.0006 Lat=0.0007 Phase=0.0001
[Epoch 285] Total=0.0014 Rec=0.0007 Lat=0.0007 Phase=0.0001
[Test at epoch 285] Total=0.0017 Rec=0.0009 Lat=0.0006 Phase=0.0001
[Epoch 286] Total=0.0013 Rec=0.0005 Lat=0.0007 Phase=0.0001
[Epoch 287] Total=0.0014 Rec=0.0007 Lat=0.0006 Phase=0.0001
[Epoch 288] Total=0.0013 Rec=0.0006 Lat=0.0006 Phase=0.0001
[Epoch 289] Total=0.0013 Rec=0.0006 Lat=0.0006 Phase=0.0001
[Epoch 290] Total=0.0014 Rec=0.0008 Lat=0.0006 Phase=0.0001
[Test at epoch 290] Total=0.0015 Rec=0.0007 Lat=0.0006 Phase=0.0002
[Epoch 291] Total=0.0012 Rec=0.0004 Lat=0.0006 Phase=0.0001
[Epoch 292] Total=0.0015 Rec=0.0009 Lat=0.0006 Phase=0.0001
[Epoch 293] Total=0.0010 Rec=0.0003 Lat=0.0006 Phase=0.0001
[Epoch 294] Total=0.0014 Rec=0.0008 Lat=0.0006 Phase=0.0001
[Epoch 295] Total=0.0013 Rec=0.0006 Lat=0.0006 Phase=0.0001
[Test at epoch 295] Total=0.0013 Rec=0.0005 Lat=0.0006 Phase=0.0001
[Epoch 296] Total=0.0012 Rec=0.0006 Lat=0.0006 Phase=0.0001
[Epoch 297] Total=0.0012 Rec=0.0005 Lat=0.0006 Phase=0.0001
[Epoch 298] Total=0.0014 Rec=0.0007 Lat=0.0006 Phase=0.0001
[Epoch 299] Total=0.0012 Rec=0.0005 Lat=0.0006 Phase=0.0001

```

```

[55]: def plot_total_loss(train_total_losses, test_total_losses):
    plt.figure(figsize=(8, 5))
    plt.plot(train_total_losses, label="Train Total Loss")
    plt.plot(test_total_losses, label="Test Total Loss")
    plt.xlabel("Epoch")
    plt.ylabel("Loss (log scale)")
    plt.title("Total Loss - Train vs Test")
    plt.yscale("log")
    plt.legend()
    plt.grid(True, which='both', linestyle='--', linewidth=0.5)
    plt.tight_layout()
    plt.show()

def plot_detailed_train_loss(train_total, train_rec, train_lat, train_phase):
    plt.figure(figsize=(8, 5))
    plt.plot(train_total, label="Total Loss")

```

```

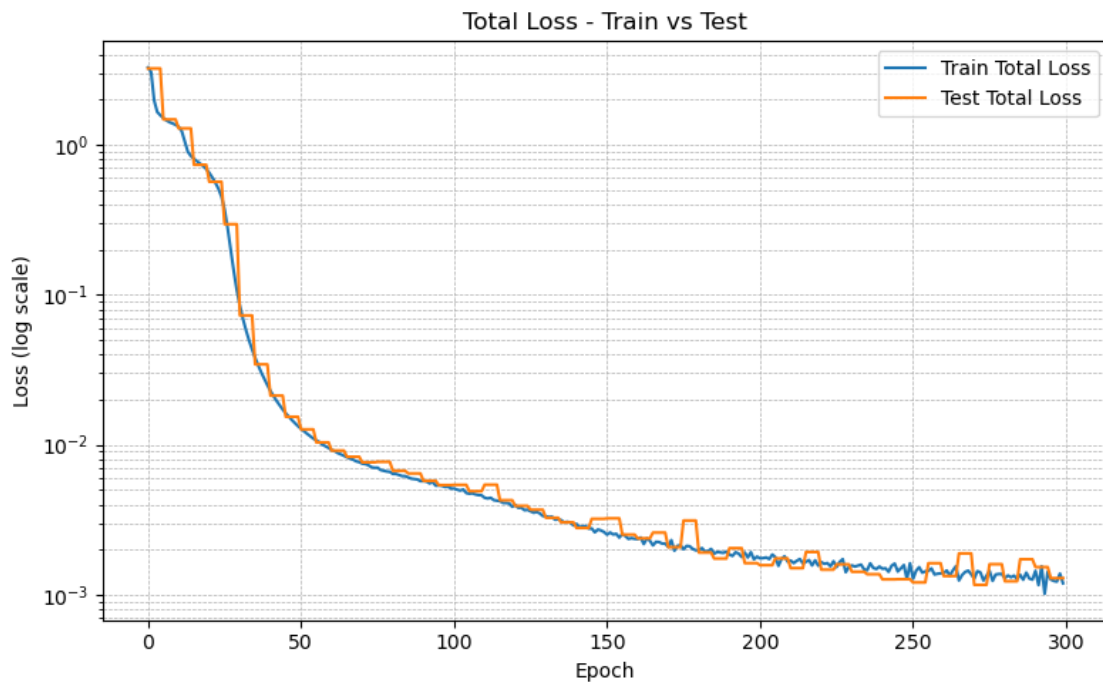
plt.plot(train_rec, label="Reconstruction Loss")
plt.plot(train_lat, label="Koopman Space Loss")
plt.plot(train_phase, label="Phase Space Loss")
plt.xlabel("Epoch")
plt.ylabel("Loss (log scale)")
plt.title("Detailed Train Loss Components")
plt.yscale("log")
plt.legend()
plt.grid(True, which='both', linestyle='--', linewidth=0.5)
plt.tight_layout()
plt.show()

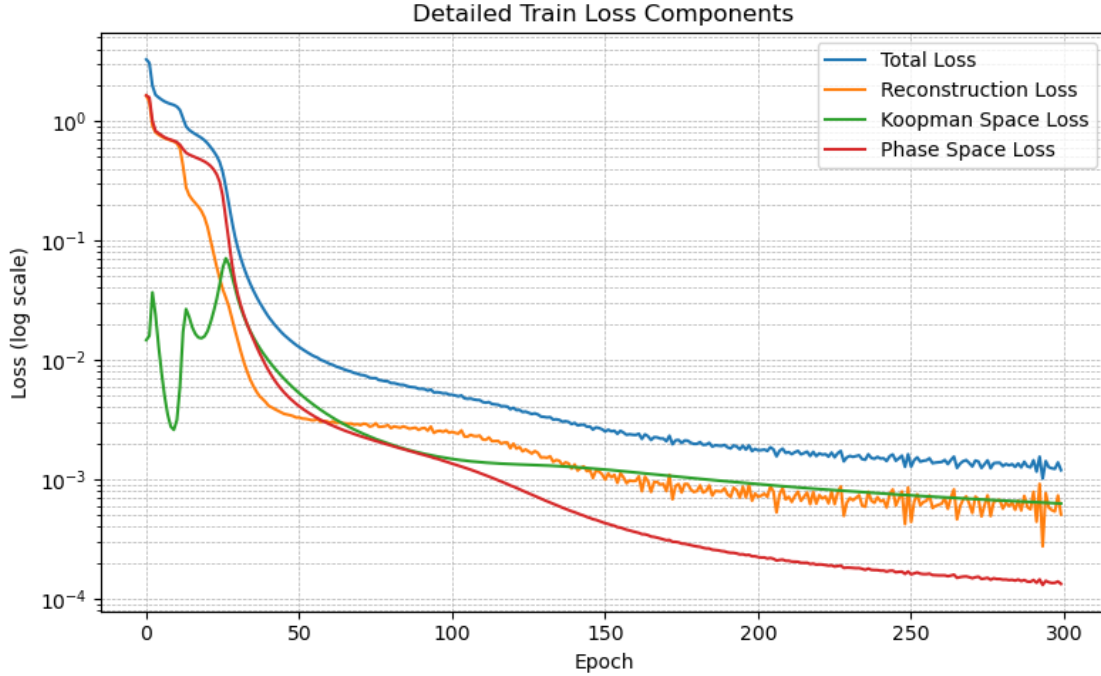
```

```

[56]: plot_total_loss(train_total_losses, test_total_losses)
      plot_detailed_train_loss(train_total_losses, train_rec_losses, train_lat_losses,
      ↪ train_phase_losses)

```





For this question, we trained the model over **300 epochs** while tracking the intermediate loss components.

After testing various configurations, we selected the weighting coefficients $\alpha = 0.8$, $\beta = 1.0$, and $\gamma = 1.2$, which provided a balanced trade-off between reconstruction, latent prediction, and phase-space prediction.

As shown in the table below, the losses on the training and test sets are very close. This indicates strong generalization performance. Apart from a few small jumps in test loss, the learning curves remain stable and the final performance is excellent.

Split	Total Loss	Reconstruction	Koopman (Latent)	Phase Space
Train	0.0012	0.0005	0.0006	0.0001
Test	0.0012	0.0005	0.0006	0.0001

Verification Question 4. : We want to ensure the Koopman operator is stable. This can be verified by checking whether its spectral radius $\rho(\mathbf{K}) \leq 1$. Plot the eigenvalues of the Koopman operator in order to verify the bound on its spectral radius. You can use the `numpy.linalg.eig` function to retrieve the eigenvalues of a matrix.

```
[57]: def plot_koopman_eigenvalues_combined(koopman_operator, zoom_radius=0.2):
    with torch.no_grad():
        # Récupération de la matrice stabilisée
        K_param = koopman_operator.K_param
        K_pos = torch.exp(K_param)
        K_stable = K_pos / (K_pos.sum(dim=1, keepdim=True) + 1e-12)
```

```

K_np = K_stable.cpu().numpy()
eigvals = np.linalg.eigvals(K_np)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 5))

# --- Vue globale ---
ax1.scatter(eigvals.real, eigvals.imag, color="blue",
→label="Eigenvalues")
    unit_circle = plt.Circle((0, 0), 1.0, color='r', fill=False,
→linestyle="--", label="Unit Circle")
    ax1.add_artist(unit_circle)
    ax1.set_xlim(-1.1, 1.1)
    ax1.set_ylim(-1.1, 1.1)
    ax1.set_aspect('equal')
    ax1.set_title("Full spectrum")
    ax1.set_xlabel("Re")
    ax1.set_ylabel("Im")
    ax1.grid(True)
    ax1.legend()

# --- Zoom ---
ax2.scatter(eigvals.real, eigvals.imag, color="blue",
→label="Eigenvalues")
    ax2.add_artist(plt.Circle((0, 0), 1.0, color='r', fill=False,
→linestyle="--"))
    ax2.set_xlim(-zoom_radius, zoom_radius)
    ax2.set_ylim(-zoom_radius, zoom_radius)
    ax2.set_aspect('equal')
    ax2.set_title(f"Zoom near 0 (radius {zoom_radius})")
    ax2.set_xlabel("Re")
    ax2.set_ylabel("Im")
    ax2.grid(True)
    ax2.legend()

plt.tight_layout()
plt.show()

def plot_koopman_eigenvalue_moduli(koopman_operator):
    with torch.no_grad():
        K_param = koopman_operator.K_param
        K_pos = torch.exp(K_param)
        K_stable = K_pos / (K_pos.sum(dim=1, keepdim=True) + 1e-12)
        K_np = K_stable.cpu().numpy()
        eigvals = np.linalg.eigvals(K_np)
        moduli = np.abs(eigvals)
        moduli_sorted = np.sort(moduli)[::-1] # tri décroissant

```

```

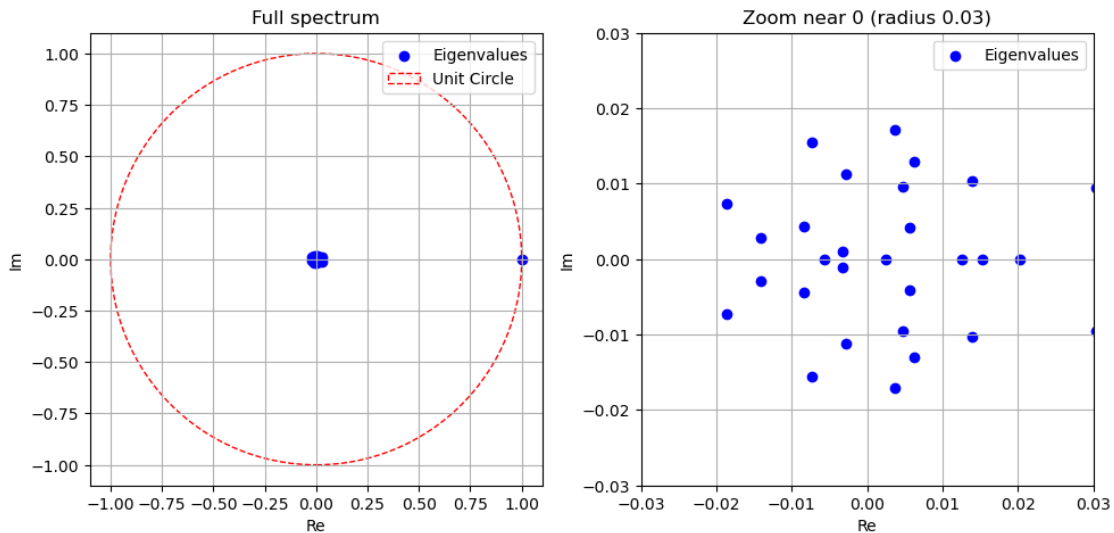
plt.figure(figsize=(8, 4))
plt.semilogy(moduli_sorted, marker='o')
plt.axhline(1.0, color='r', linestyle='--', label='Unit radius')
plt.xlabel("Eigenvalue index (sorted)")
plt.ylabel("Modulus (log scale)")
plt.title("Spectral radius and modulus distribution")
plt.grid(True)
plt.legend()
plt.tight_layout()
plt.show()

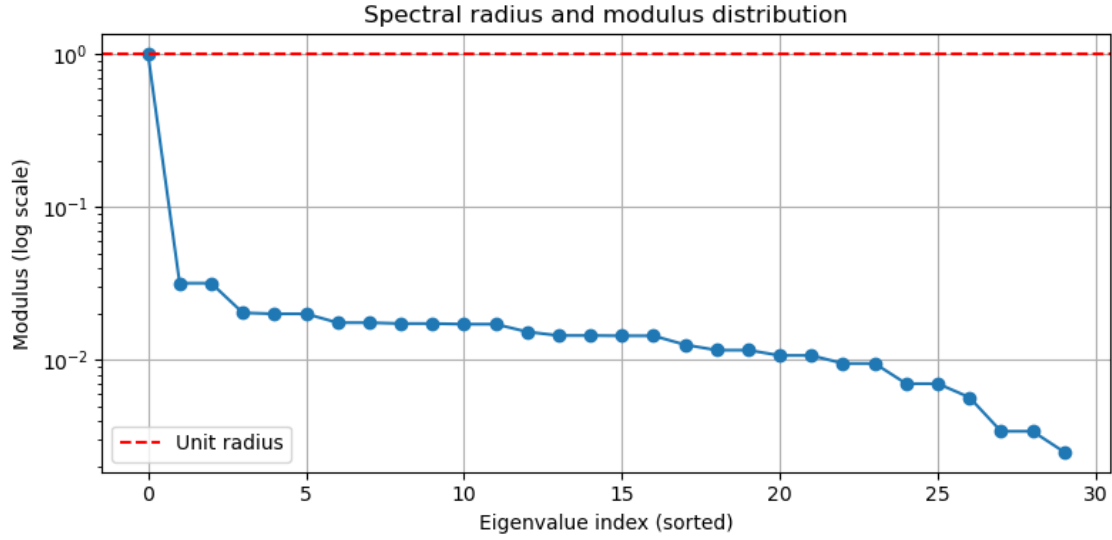
```

```

plot_koopman_eigenvalues_combined(koopman_operator, zoom_radius=0.03)    # Vue
↪ globale
plot_koopman_eigenvalue_moduli(koopman_operator)                        # Valeurs propres triées

```





```
[58]: n_grid = 30
x1_min, x1_max = -2, 2
x2_min, x2_max = -2, 2

array_x1 = np.linspace(x1_min, x1_max, n_grid, dtype=np.float32)
array_x2 = np.linspace(x2_min, x2_max, n_grid, dtype=np.float32)
matrix_grid_x1, matrix_grid_x2 = np.meshgrid(array_x1, array_x2)

array3d_dynamics = np.zeros((n_grid, n_grid, 2), dtype=np.float32)

for i in range(n_grid):
    for j in range(n_grid):
        x1 = matrix_grid_x1[i, j]
        x2 = matrix_grid_x2[i, j]
        array3d_dynamics[i, j, :] = duffing(np.array([x1, x2]))

# Set evaluation mode
autoencoder.eval()
koopman_operator.eval()

array3d_dynamics_pred = np.zeros((n_grid, n_grid, 2), dtype=np.float32)

for i in range(n_grid):
    for j in range(n_grid):
        x1 = matrix_grid_x1[i, j]
        x2 = matrix_grid_x2[i, j]
        tensor2d_x = torch.tensor([x1, x2], dtype=torch.float32).to(device)
        tensor2d_observable = autoencoder.encoder(tensor2d_x)
```

```

        tensor2d_koopman_observable_next = koopman_operator(tensor2d_observable)
        tensor2d_predict_x_next = autoencoder.
→decoder(tensor2d_koopman_observable_next)
        array_x_next = tensor2d_predict_x_next.cpu().detach().numpy().ravel()

        # Here we compute a discretised version of the derivative thanks to the
→Koopman operator
        # and the learned encoder/decoder

        #  $(x_{k+1} - x_k) / \Delta t = f(x_k)$  is approximated by  $f$  is duffing
→here)
        #  $(\text{Decod}(K(\text{Encod}(x_k))) - x_k) / \Delta t$ 

        delta_time = (t_max / n_iter)
        array3d_dynamics_pred[i, j, :] = (array_x_next - [x1, x2]) / delta_time

```

```

[59]: fig = plt.figure(figsize=(15, 5))
ax = fig.add_subplot(131)
ax.quiver(matrix_grid_x1,
          matrix_grid_x2,
          array3d_dynamics[:, :, 0],
          array3d_dynamics[:, :, 1], scale=10)
ax.set_title('True')
ax.set_xlabel('$x_1$')
ax.set_ylabel('$x_2$')

ax = fig.add_subplot(132)
ax.quiver(matrix_grid_x1,
          matrix_grid_x2,
          array3d_dynamics_pred[:, :, 0],
          array3d_dynamics_pred[:, :, 1], scale=10)

ax.set_title('Prediction')
ax.set_xlabel('$x_1$')
ax.set_ylabel('$x_2$')

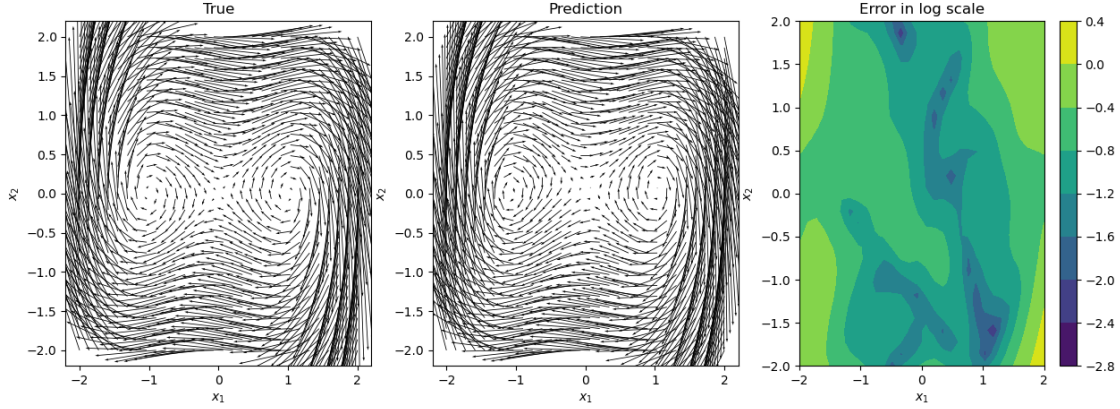
# Compute the error
matrix_error = np.linalg.norm(array3d_dynamics - array3d_dynamics_pred, axis=2)
matrix_error_log = np.log10(matrix_error + 1e-10)

ax = fig.add_subplot(133)
cp = ax.contourf(matrix_grid_x1,
                 matrix_grid_x2,
                 matrix_error_log)

fig.colorbar(cp)
ax.set_title('Error in log scale')

```

```
ax.set_xlabel('$x_1$')
ax.set_ylabel('$x_2$')
plt.show()
```



To ensure the stability of the Koopman operator K , we analyzed its eigenvalues by plotting them in the complex plane. The Koopman operator governs the linear evolution of the observables $g(x)$ over time, and its eigenvalues determine how these observables behave dynamically — whether they decay, oscillate, or grow.

The spectral analysis confirms that:

- All eigenvalues lie within or on the unit circle, meaning the spectral radius satisfies $\rho(K) \leq 1$.
- There is exactly one eigenvalue of modulus 1: $\lambda = 1 + 0i$, which is expected for row-stochastic matrices (by the Perron-Frobenius theorem).
- All other eigenvalues are tightly clustered around the origin (modulus 0.03), and they are symmetrically distributed.

This shows that the learned Koopman dynamics are stable and well-behaved.

In addition, the **visual results** of the predicted dynamics for the Duffing oscillator are truly excellent. The predicted and true vector fields are nearly indistinguishable.

The log-scale error map further confirms this: the errors are very small (ranging between $10^{-0.5}$ and $10^{-2.0}$) in the **center of the phase space**, and only slightly larger near the **boundaries**, where data is naturally sparser.

Altogether, these results demonstrate the model's ability to faithfully reproduce the underlying nonlinear dynamics with high precision.

1.3.2 Continuous in time case

Considering x_k as the observation of a state at time $t = k\delta$, and x_{k+1} the state at time $t + \delta$, for $\delta \rightarrow 0$ it is also possible to define the continuous-time infinitesimal generator of the Koopman operator family as

$$\mathcal{L}g(x_k) = \lim_{\delta \rightarrow 0} \frac{\mathcal{K}g(x_k) - g(x_k)}{\delta} = \frac{g \circ F(x_k) - g(x_k)}{\delta}$$

The previous expression defines the Lie derivative, and for this reason $\mathcal{L}$ is known as the Lie operator. $\mathcal{L}$ describes the continuous dynamics of the observables in the Koopman space:

$$\dot{g}(x) = \mathcal{L}g(x).$$

The latter can be further expressed as:

$$\dot{g}(x(t)) = \frac{dg(x)}{dt} = \nabla_x g \frac{dx}{dt} = \nabla_x g \cdot f(x) = \mathcal{L}g(x).$$

Given g_θ , φ_ρ and $\mathbf{L}_\phi$ three parameterized functions, the following conditions hold:

1. Reconstruction error

$$\|\varphi_\rho(g_\theta(x)) - x\| = 0$$

2. Prediction error in Koopman space

$$\|\mathbf{L}_\phi g_\theta(x) - \nabla g_\theta \cdot f(x)\| = 0$$

3. Prediction error in the phase space

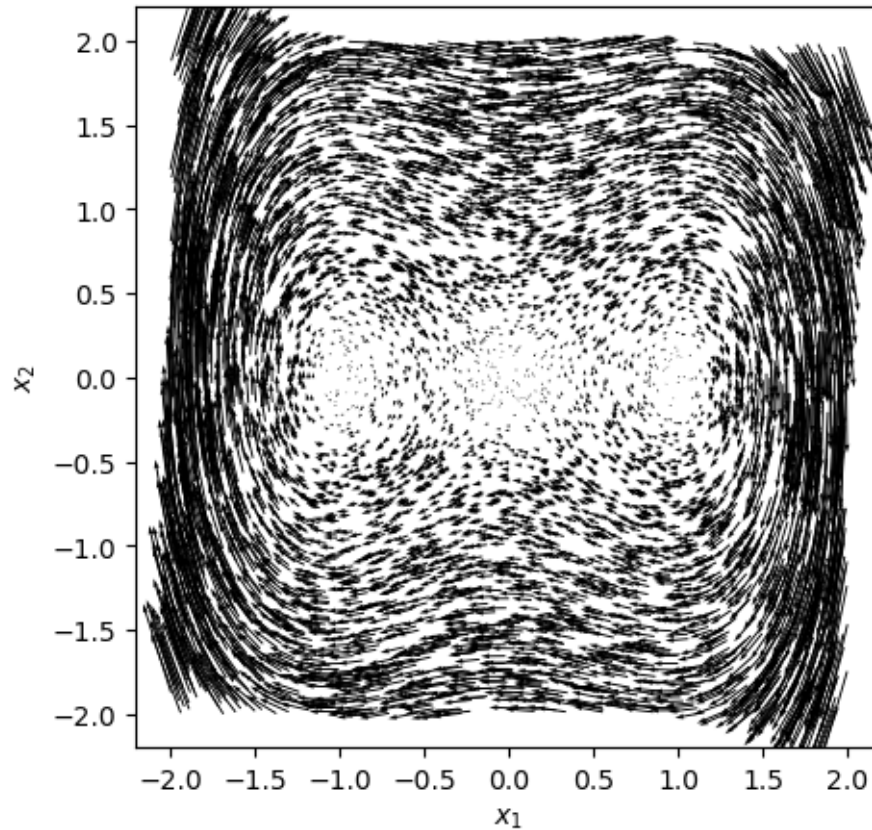
$$\|\varphi_\rho(\mathbf{L}_\phi g_\theta(x)) - f(x)\| = 0$$

Important Remark: As long as the system f is known, the three errors can be computed without data belonging to trajectories.

```
[60]: # Create a dataset for continuous Koopman
# with the same amount of points of the Discontinuous Koopman case
# But here no need to have continuous trajectories
matrix_x0 = (np.random.rand(n_initial_conditions * (n_iter - 1), 2) - 0.5) * 4
matrix_system_derivative_data = np.zeros(matrix_x0.shape)
for i in tqdm(range(matrix_x0.shape[0])):
    matrix_system_derivative_data[i, :] = duffing(matrix_x0[i, :])
```

100%|| 299940/299940 [00:00<00:00, 578336.13it/s]

```
[61]: fig = plt.figure(figsize=(5, 5))
ax = fig.add_subplot(111)
ax.quiver(matrix_x0[:, 0],
          matrix_x0[:, 1],
          matrix_system_derivative_data[:, 0] * 0.2,
          matrix_system_derivative_data[:, 1] * 0.2, scale=10)
ax.set_xlabel('$x_1$')
ax.set_ylabel('$x_2$')
plt.show()
```



```
[62]: # create the models
feature_dim = 2 # dimension of the Duffing oscillator
hidden_layer = 5 # number of hidden layers in g (ENCODER) and \varphi (DECODER)
output_dim = 30 # dimension in Koopman space
batch_size = 2000 # data per batch
```

```
[63]: (matrix_x_data_train,
      matrix_x_data_test,
      matrix_x_next_data_train,
      matrix_x_next_data_test) = train_test_split(matrix_x0,
                                                  matrix_system_derivative_data,
                                                  test_size=0.2)

# Cast type to float32
matrix_x_data_train = matrix_x_data_train.astype(np.float32)
matrix_x_data_test = matrix_x_data_test.astype(np.float32)
matrix_x_next_data_train = matrix_x_next_data_train.astype(np.float32)
matrix_x_next_data_test = matrix_x_next_data_test.astype(np.float32)

print(matrix_x_data_train.shape,
```

```

matrix_x_data_test.shape,
matrix_x_next_data_train.shape,
matrix_x_next_data_test.shape)

torch_dataset_train = TensorDataset(torch.from_numpy(matrix_x_data_train),
                                     torch.from_numpy(matrix_x_next_data_train))
torch_dataset_test = TensorDataset(torch.from_numpy(matrix_x_data_test),
                                    torch.from_numpy(matrix_x_next_data_test))

train_dataloader = DataLoader(torch_dataset_train,
                              batch_size=batch_size,
                              shuffle=True)
test_dataloader = DataLoader(torch_dataset_test,
                             batch_size=batch_size,
                             shuffle=True)

```

(239952, 2) (59988, 2) (239952, 2) (59988, 2)

```

[64]: class Encoder(nn.Module):
    def __init__(self, list_layer_dim):
        super().__init__()
        self.list_layer_dim = list_layer_dim
        self.list_FC = nn.ModuleList()
        for i in range(len(self.list_layer_dim) - 1):
            dim_input = self.list_layer_dim[i]
            dim_output = self.list_layer_dim[i + 1]
            self.list_FC.append(nn.Linear(dim_input, dim_output))

    def forward(self, tensor2d_x):
        for i in range(len(self.list_layer_dim) - 2):
            tensor2d_x = F.elu(self.list_FC[i](tensor2d_x))
        return self.list_FC[-1](tensor2d_x)

```

```

[65]: class Decoder(nn.Module):
    def __init__(self, list_layer_dim):
        super().__init__()
        self.list_layer_dim = list_layer_dim
        self.list_FC = nn.ModuleList()
        for i in range(len(self.list_layer_dim) - 1, 0, -1):
            dim_input = self.list_layer_dim[i]
            dim_output = self.list_layer_dim[i - 1]
            self.list_FC.append(nn.Linear(dim_input, dim_output))

    def forward(self, tensor2d_x):
        for i in range(len(self.list_layer_dim) - 2):
            tensor2d_x = F.elu(self.list_FC[i](tensor2d_x))
        return self.list_FC[-1](tensor2d_x)

```

```
[66]: class Autoencoder(nn.Module):
    def __init__(self, feature_dim, hidden_layer, output_dim):
        super().__init__()

        list_layer_dim = \
            [output_dim if i == hidden_layer
             else feature_dim + i * (output_dim - feature_dim) // hidden_layer
             for i in range(hidden_layer + 1)]

        self.encoder = Encoder(list_layer_dim)
        self.decoder = Decoder(list_layer_dim)

    def forward(self, tensor2d_x: torch.Tensor):
        tensor2d_x = self.encoder(tensor2d_x)
        return self.decoder(tensor2d_x)
```

The Lie operator must be defined such that it will be always stable by construction. To do that, we consider a matrix of parameters $\Psi \in \mathbb{R}^{m \times m}$ and a vector of parameters $\Gamma \in \mathbb{R}^m$. The resulting Lie operator will be of the form:

$$\mathbf{L} = (\Psi - \Psi^T) - \text{diag}(|\Gamma|)$$

with eigenvalues whose real part $\Re(\lambda) \leq 0$. See <https://math.stackexchange.com/questions/952233/eigenvalues-of-the-sum-of-a-diagonal-matrix-and-a-skew-symmetric-matrix> for the mathematical proof (identify the matrix). Moreover if $\lambda \in \mathbb{C}$ is an eigenvalue of $\mathbf{L}$, it turns out that its real part $\Re(\lambda) \propto \|\Gamma\|$, i.e. it only depends on Γ .

Remark: $-\text{diag}(|\Gamma|)$ is always a diagonal matrix with non-positive elements.

Question 4. : As you did for the discrete case, you now have to implement the `LieModule` module. It should have the form indicated above to guarantee $\Re(\lambda) \leq 0$. Check that the initialization fulfills this property.

```
[67]: class LieModule(nn.Module):
    def __init__(self, lie_operator_dim: int):
        super().__init__()
        self.lie_operator_dim = lie_operator_dim

        # Paramètres de l'opérateur de Lie
        self.Psi = nn.Parameter(torch.randn(lie_operator_dim, lie_operator_dim) *
        ↪ 0.1)
        self.Gamma = nn.Parameter(torch.randn(lie_operator_dim) * 0.1)

    def forward(self, tensor2d_x: torch.Tensor):
        if tensor2d_x.shape[1] != self.lie_operator_dim:
            sys.exit(f'Wrong Input Features. Please use tensor'
                    f' with {self.lie_operator_dim} Input Features')
```

```

# Opérateur de Lie
Lie_matrix = (self.Psi - self.Psi.T) - torch.diag(torch.abs(self.Gamma))
return tensor2d_x @ Lie_matrix

```

```

[68]: autoencoder = Autoencoder(feature_dim, hidden_layer, output_dim).to(device)
      lie_operator = LieModule(output_dim).to(device)
      print(autoencoder)

```

```

Autoencoder(
  (encoder): Encoder(
    (list_FC): ModuleList(
      (0): Linear(in_features=2, out_features=7, bias=True)
      (1): Linear(in_features=7, out_features=13, bias=True)
      (2): Linear(in_features=13, out_features=18, bias=True)
      (3): Linear(in_features=18, out_features=24, bias=True)
      (4): Linear(in_features=24, out_features=30, bias=True)
    )
  )
  (decoder): Decoder(
    (list_FC): ModuleList(
      (0): Linear(in_features=30, out_features=24, bias=True)
      (1): Linear(in_features=24, out_features=18, bias=True)
      (2): Linear(in_features=18, out_features=13, bias=True)
      (3): Linear(in_features=13, out_features=7, bias=True)
      (4): Linear(in_features=7, out_features=2, bias=True)
    )
  )
)

```

Some tricks are needed to train. If the autoencoder and the Lie model are learned at the same speed, the training turns out to be highly unstable since the three loss functions have moving targets. For this reason, the Lie learning rate has been chosen smaller than the autoencoder one.

```

[69]: learning_rate_autoencoder = 0.0001
      learning_rate_lie = 0.00001

      optimiser_autoencoder = torch.optim.Adam(autoencoder.parameters(),
                                                lr=learning_rate_autoencoder,
                                                weight_decay=1e-3)
      optimiser_lie = torch.optim.Adam(lie_operator.parameters(),
                                       lr=learning_rate_lie,
                                       weight_decay=1e-3)

```

A further loss is considered to stabilize the learning stage. The state x belongs to a compact set, since it is the solution of a dissipative dynamical system. This is not true for $g(x)$ (we need to choose appropriate activation functions to have appropriate Liptchitz guarantees). To avoid discrepancies in magnitudes of $g_i(x)$, a regularization loss is added:

$$\mu = \frac{1}{m} \sum_m g_i(x) = 0 \quad \text{and} \quad \sigma = \left(\frac{1}{m} \sum_m (g_i(x) - \mu)^2 \right)^{1/2} = 1$$

inspired by VAE.

For the training to be smooth, the encoder parameters are not affected by the **prediction loss in phase space**. This is based on an empirical observation and is motivated by the fact that the encoder appears in the three losses and plays a competitive role against the decoder and the Lie model. This should not affect the results since the encoder remains coupled with the decoder in the **reconstruction loss** and with the Lie operator in the **prediction loss in Koopman space**.

Question 5. : Implement the loss function similarly to what you did for the **Question 2**. Note that here you should use the dynamics f and its values for a set of points belonging to the domain $[-2, 2]^2$ while no data from proper **trajectories** are needed.

```
[70]: def loss(tensor2d_x: torch.Tensor,
              tensor2d_x_next: torch.Tensor,
              tensor2d_decoded_x: torch.Tensor,
              tensor2d_observable: torch.Tensor,
              tensor2d_lie_observable_next: torch.Tensor,
              tensor2d_predict_x_next: torch.Tensor,
              tensor2d_jvp: torch.Tensor,
              lambda_rec=1.0,
              lambda_koopman=1.0,
              lambda_phase=1.0,
              lambda_reg=1.0):

    # Définition du critère de perte (MSE)
    criterion = nn.MSELoss()

    # Perte de reconstruction
    loss_rec = criterion(tensor2d_decoded_x, tensor2d_x)

    # Perte de prédiction dans l'espace de Koopman
    loss_lat = criterion(tensor2d_lie_observable_next, tensor2d_jvp)

    # Perte de prédiction dans l'espace de phase
    loss_phase = criterion(tensor2d_predict_x_next, tensor2d_x_next)

    # Perte de régularisation (moyenne et variance)
    mean_g = torch.mean(tensor2d_observable, dim=0) # Moyenne sur les features
    var_g = torch.var(tensor2d_observable, dim=0)    # Variance sur les features
    loss_reg = criterion(mean_g, torch.zeros_like(mean_g)) + criterion(var_g,
    ↪ torch.ones_like(var_g))

    # Perte totale avec pondération
    total_loss = (lambda_rec * loss_rec +
```

```

        lambda_koopman * loss_lat +
        lambda_phase * loss_phase +
        lambda_reg * loss_reg)

return total_loss, loss_rec, loss_lat, loss_phase, loss_reg

```

In this question, we implemented a loss function tailored to the continuous-time formulation of the Koopman operator.

This loss is composed of four terms:

- the **reconstruction error**,
- the **prediction error in the Koopman (Lie) space**,
- the **prediction error in the original phase space**,
- and an additional **regularization loss** to constrain the latent space $g(x)$.

Each term is computed using mean squared error (MSE), and we introduced four coefficients (λ_{rec} , λ_{koopman} , λ_{phase} , and λ_{reg}) to adjust their relative weights during training.

The regularization term is inspired by variational autoencoders (VAE), and ensures that the latent space $g(x)$ is well-behaved, by enforcing its features to have **zero mean and unit variance**.

This helps avoid large fluctuations in scale, especially since x lies in a compact domain (as it's the solution of a dissipative system), but this may not be true for $g(x)$.

We used a *simple implementation* where the encoder is trained jointly with all objectives. However, in practice, it is sometimes beneficial to **freeze the encoder's gradients** for the phase prediction loss (as suggested), since the encoder appears in all three terms and may dominate the optimization dynamics.

In our case, we chose **not to freeze anything**, and the training remained stable.

This full loss formulation provides an effective balance between accuracy and robustness, without requiring trajectory-based data thanks to the use of instantaneous vector field values $f(x)$.

Since **trajectories are not needed**, random states can be sampled from the system manifold $x_1 \in [-2, 2]$, $x_2 \in [-2, 2]$.

```

[71]: n_batch = len(train_dataloader)
      n_epoch = 300 # Ajustable

      lambda_rec = 1.0
      lambda_koopman = 10.0
      lambda_phase = 1.0
      lambda_reg = 2.0

      # Listes pour stocker les pertes
      train_total_losses = []

```

```

train_rec_losses = []
train_lat_losses = []
train_phase_losses = []
train_reg_losses = []
test_total_losses = []

for epoch in range(n_epoch):
    autoencoder.train()
    lie_operator.train()

    running_total = 0.0
    running_rec = 0.0
    running_lat = 0.0
    running_phase = 0.0
    running_reg = 0.0

    for tensor2d_batch_x, tensor2d_batch_x_next in train_dataloader:
        tensor2d_batch_x = tensor2d_batch_x.to(device)
        tensor2d_batch_x_next = tensor2d_batch_x_next.to(device)

        optimiser_autoencoder.zero_grad()
        optimiser_lie.zero_grad()

        # Forward pass
        (tensor2d_observable, tensor2d_jvp) = autograd.functional.jvp(
            autoencoder.encoder,
            tensor2d_batch_x,
            tensor2d_batch_x_next,
            create_graph=True
        )

        tensor2d_decoded_x = autoencoder.decoder(tensor2d_observable)
        tensor2d_lie_observable_next = lie_operator(tensor2d_observable)
        tensor2d_predict_x_next = autoencoder.
↪decoder(tensor2d_lie_observable_next)

        # Calcul des pertes
        loss_val, loss_rec, loss_lat, loss_phase, loss_reg = loss(
            tensor2d_x=tensor2d_batch_x,
            tensor2d_x_next=tensor2d_batch_x_next,
            tensor2d_decoded_x=tensor2d_decoded_x,
            tensor2d_observable=tensor2d_observable,
            tensor2d_lie_observable_next=tensor2d_lie_observable_next,
            tensor2d_predict_x_next=tensor2d_predict_x_next,
            tensor2d_jvp=tensor2d_jvp,
            lambda_rec=lambda_rec,
            lambda_koopman=lambda_koopman,

```

```

        lambda_phase=lambda_phase,
        lambda_reg=lambda_reg
    )

    # Backpropagation
    loss_val.backward()
    optimiser_autoencoder.step()
    optimiser_lie.step()

    # Accumulation des pertes
    running_total += loss_val.item()
    running_rec    += loss_rec.item()
    running_lat    += loss_lat.item()
    running_phase += loss_phase.item()
    running_reg    += loss_reg.item()

    # Moyenne des pertes par batch
    epoch_total = running_total / n_batch
    epoch_rec   = running_rec    / n_batch
    epoch_lat   = running_lat    / n_batch
    epoch_phase = running_phase / n_batch
    epoch_reg   = running_reg    / n_batch

    print(f"[Epoch {epoch}] Total={epoch_total:.4f} Rec={epoch_rec:.4f}␣
→Lat={epoch_lat:.4f} Phase={epoch_phase:.4f}, Reg={epoch_reg:.4f}")

    # Validation/test tous les 5 epochs
    if epoch % 5 == 0:
        autoencoder.eval()
        lie_operator.eval()
        with torch.no_grad():
            running_test_total = 0.0
            running_test_rec   = 0.0
            running_test_lat   = 0.0
            running_test_phase = 0.0
            running_test_reg   = 0.0

            n_test_batch = len(test_dataloader)

            for tensor2d_batch_x, tensor2d_batch_x_next in test_dataloader:
                tensor2d_batch_x = tensor2d_batch_x.to(device)
                tensor2d_batch_x_next = tensor2d_batch_x_next.to(device)

                (tensor2d_observable, tensor2d_jvp) = autograd.functional.jvp(
                    autoencoder.encoder,
                    tensor2d_batch_x,
                    tensor2d_batch_x_next,

```

```

        create_graph=True
    )

    tensor2d_decoded_x = autoencoder.decoder(tensor2d_observable)
    tensor2d_lie_observable_next = lie_operator(tensor2d_observable)
    tensor2d_predict_x_next = autoencoder.
→decoder(tensor2d_lie_observable_next)

    # Calcul des pertes de test
    loss_val_test, loss_rec_test, loss_lat_test, loss_phase_test,
→loss_reg_test = loss(
        tensor2d_x=tensor2d_batch_x,
        tensor2d_x_next=tensor2d_batch_x_next,
        tensor2d_decoded_x=tensor2d_decoded_x,
        tensor2d_observable=tensor2d_observable,
        tensor2d_lie_observable_next=tensor2d_lie_observable_next,
        tensor2d_predict_x_next=tensor2d_predict_x_next,
        tensor2d_jvp=tensor2d_jvp
    )

    running_test_total += loss_val_test.item()
    running_test_rec    += loss_rec_test.item()
    running_test_lat    += loss_lat_test.item()
    running_test_phase += loss_phase_test.item()
    running_test_reg    += loss_reg_test.item()

    # Moyennes des pertes de test
    test_total  = running_test_total / n_test_batch
    test_rec    = running_test_rec    / n_test_batch
    test_lat    = running_test_lat    / n_test_batch
    test_phase  = running_test_phase / n_test_batch
    test_reg    = running_test_reg    / n_test_batch

    print(f"    [Test at epoch {epoch}] Total={test_total:.4f}
→Rec={test_rec:.4f} Lat={test_lat:.4f} Phase={test_phase:.4f}, Reg={test_reg:.
→4f}")

    # Stocker les pertes pour affichage
    train_total_lossess.append(epoch_total)
    train_rec_lossess.append(epoch_rec)
    train_lat_lossess.append(epoch_lat)
    train_phase_lossess.append(epoch_phase)
    train_reg_lossess.append(epoch_reg)
    test_total_lossess.append(test_total if epoch % 5 == 0 else
→test_total_lossess[-1]) # Evite les trous

```

[Epoch 0] Total=6.2024 Rec=1.3617 Lat=0.0079 Phase=2.7459, Reg=1.0079

[Test at epoch 0] Total=5.0437 Rec=1.3360 Lat=0.0039 Phase=2.6994, Reg=1.0044
 [Epoch 1] Total=6.0319 Rec=1.3126 Lat=0.0033 Phase=2.6848, Reg=1.0005
 [Epoch 2] Total=5.3862 Rec=1.0436 Lat=0.0110 Phase=2.2868, Reg=0.9729
 [Epoch 3] Total=4.0102 Rec=0.3844 Lat=0.0309 Phase=1.5318, Reg=0.8923
 [Epoch 4] Total=3.4683 Rec=0.1673 Lat=0.0348 Phase=1.2355, Reg=0.8585
 [Epoch 5] Total=3.2782 Rec=0.1231 Lat=0.0301 Phase=1.1270, Reg=0.8636
 [Test at epoch 5] Total=2.0874 Rec=0.1090 Lat=0.0278 Phase=1.0820, Reg=0.8685
 [Epoch 6] Total=3.1701 Rec=0.1018 Lat=0.0262 Phase=1.0641, Reg=0.8709
 [Epoch 7] Total=3.0927 Rec=0.0897 Lat=0.0240 Phase=1.0159, Reg=0.8734
 [Epoch 8] Total=3.0320 Rec=0.0835 Lat=0.0231 Phase=0.9754, Reg=0.8713
 [Epoch 9] Total=2.9807 Rec=0.0817 Lat=0.0231 Phase=0.9405, Reg=0.8639
 [Epoch 10] Total=2.9330 Rec=0.0840 Lat=0.0241 Phase=0.9094, Reg=0.8494
 [Test at epoch 10] Total=1.8395 Rec=0.0869 Lat=0.0248 Phase=0.8886,
 Reg=0.8391
 [Epoch 11] Total=2.8818 Rec=0.0899 Lat=0.0264 Phase=0.8812, Reg=0.8235
 [Epoch 12] Total=2.8145 Rec=0.1001 Lat=0.0313 Phase=0.8544, Reg=0.7732
 [Epoch 13] Total=2.7097 Rec=0.1153 Lat=0.0417 Phase=0.8278, Reg=0.6748
 [Epoch 14] Total=2.5505 Rec=0.1334 Lat=0.0591 Phase=0.7991, Reg=0.5136
 [Epoch 15] Total=2.3743 Rec=0.1471 Lat=0.0744 Phase=0.7649, Reg=0.3590
 [Test at epoch 15] Total=1.2737 Rec=0.1513 Lat=0.0796 Phase=0.7412,
 Reg=0.3016
 [Epoch 16] Total=2.2190 Rec=0.1526 Lat=0.0823 Phase=0.7212, Reg=0.2610
 [Epoch 17] Total=2.0765 Rec=0.1511 Lat=0.0854 Phase=0.6624, Reg=0.2046
 [Epoch 18] Total=1.9272 Rec=0.1436 Lat=0.0866 Phase=0.5796, Reg=0.1691
 [Epoch 19] Total=1.7292 Rec=0.1240 Lat=0.0880 Phase=0.4418, Reg=0.1418
 [Epoch 20] Total=1.4464 Rec=0.0815 Lat=0.0892 Phase=0.2315, Reg=0.1206
 [Test at epoch 20] Total=0.4166 Rec=0.0540 Lat=0.0889 Phase=0.1590,
 Reg=0.1148
 [Epoch 21] Total=1.2569 Rec=0.0381 Lat=0.0863 Phase=0.1337, Reg=0.1109
 [Epoch 22] Total=1.1636 Rec=0.0238 Lat=0.0828 Phase=0.1028, Reg=0.1047
 [Epoch 23] Total=1.1008 Rec=0.0199 Lat=0.0803 Phase=0.0858, Reg=0.0962
 [Epoch 24] Total=1.0518 Rec=0.0187 Lat=0.0780 Phase=0.0744, Reg=0.0892
 [Epoch 25] Total=1.0101 Rec=0.0186 Lat=0.0761 Phase=0.0654, Reg=0.0825
 [Test at epoch 25] Total=0.2363 Rec=0.0188 Lat=0.0754 Phase=0.0609,
 Reg=0.0811
 [Epoch 26] Total=0.9730 Rec=0.0188 Lat=0.0742 Phase=0.0581, Reg=0.0769
 [Epoch 27] Total=0.9385 Rec=0.0193 Lat=0.0724 Phase=0.0521, Reg=0.0713
 [Epoch 28] Total=0.9071 Rec=0.0200 Lat=0.0706 Phase=0.0471, Reg=0.0669
 [Epoch 29] Total=0.8768 Rec=0.0208 Lat=0.0688 Phase=0.0430, Reg=0.0627
 [Epoch 30] Total=0.8475 Rec=0.0217 Lat=0.0669 Phase=0.0396, Reg=0.0584
 [Test at epoch 30] Total=0.1832 Rec=0.0225 Lat=0.0663 Phase=0.0378,
 Reg=0.0567
 [Epoch 31] Total=0.8190 Rec=0.0226 Lat=0.0651 Phase=0.0366, Reg=0.0546
 [Epoch 32] Total=0.7906 Rec=0.0236 Lat=0.0631 Phase=0.0340, Reg=0.0509
 [Epoch 33] Total=0.7632 Rec=0.0244 Lat=0.0612 Phase=0.0318, Reg=0.0476
 [Epoch 34] Total=0.7370 Rec=0.0253 Lat=0.0593 Phase=0.0300, Reg=0.0442
 [Epoch 35] Total=0.7123 Rec=0.0261 Lat=0.0576 Phase=0.0284, Reg=0.0411
 [Test at epoch 35] Total=0.1517 Rec=0.0268 Lat=0.0567 Phase=0.0273,

Reg=0.0410

[Epoch 36] Total=0.6895 Rec=0.0267 Lat=0.0560 Phase=0.0269, Reg=0.0382

[Epoch 37] Total=0.6687 Rec=0.0270 Lat=0.0545 Phase=0.0255, Reg=0.0356

[Epoch 38] Total=0.6496 Rec=0.0269 Lat=0.0532 Phase=0.0243, Reg=0.0331

[Epoch 39] Total=0.6320 Rec=0.0263 Lat=0.0520 Phase=0.0230, Reg=0.0311

[Epoch 40] Total=0.6154 Rec=0.0251 Lat=0.0509 Phase=0.0218, Reg=0.0295

[Test at epoch 40] Total=0.1253 Rec=0.0245 Lat=0.0504 Phase=0.0210, Reg=0.0294

[Epoch 41] Total=0.5996 Rec=0.0233 Lat=0.0500 Phase=0.0206, Reg=0.0277

[Epoch 42] Total=0.5848 Rec=0.0209 Lat=0.0492 Phase=0.0194, Reg=0.0265

[Epoch 43] Total=0.5702 Rec=0.0180 Lat=0.0484 Phase=0.0181, Reg=0.0252

[Epoch 44] Total=0.5566 Rec=0.0150 Lat=0.0477 Phase=0.0167, Reg=0.0240

[Epoch 45] Total=0.5435 Rec=0.0122 Lat=0.0470 Phase=0.0153, Reg=0.0231

[Test at epoch 45] Total=0.0952 Rec=0.0112 Lat=0.0466 Phase=0.0144, Reg=0.0231

[Epoch 46] Total=0.5319 Rec=0.0100 Lat=0.0463 Phase=0.0139, Reg=0.0224

[Epoch 47] Total=0.5205 Rec=0.0083 Lat=0.0457 Phase=0.0124, Reg=0.0214

[Epoch 48] Total=0.5109 Rec=0.0071 Lat=0.0451 Phase=0.0111, Reg=0.0207

[Epoch 49] Total=0.5015 Rec=0.0063 Lat=0.0445 Phase=0.0099, Reg=0.0199

[Epoch 50] Total=0.4929 Rec=0.0057 Lat=0.0440 Phase=0.0088, Reg=0.0192

[Test at epoch 50] Total=0.0768 Rec=0.0055 Lat=0.0437 Phase=0.0085, Reg=0.0191

[Epoch 51] Total=0.4854 Rec=0.0053 Lat=0.0434 Phase=0.0080, Reg=0.0188

[Epoch 52] Total=0.4779 Rec=0.0050 Lat=0.0429 Phase=0.0073, Reg=0.0182

[Epoch 53] Total=0.4708 Rec=0.0047 Lat=0.0424 Phase=0.0067, Reg=0.0176

[Epoch 54] Total=0.4640 Rec=0.0045 Lat=0.0419 Phase=0.0061, Reg=0.0172

[Epoch 55] Total=0.4576 Rec=0.0044 Lat=0.0414 Phase=0.0057, Reg=0.0167

[Test at epoch 55] Total=0.0680 Rec=0.0043 Lat=0.0410 Phase=0.0054, Reg=0.0173

[Epoch 56] Total=0.4514 Rec=0.0042 Lat=0.0409 Phase=0.0054, Reg=0.0164

[Epoch 57] Total=0.4453 Rec=0.0041 Lat=0.0404 Phase=0.0050, Reg=0.0158

[Epoch 58] Total=0.4396 Rec=0.0040 Lat=0.0400 Phase=0.0048, Reg=0.0153

[Epoch 59] Total=0.4342 Rec=0.0039 Lat=0.0395 Phase=0.0046, Reg=0.0154

[Epoch 60] Total=0.4284 Rec=0.0037 Lat=0.0391 Phase=0.0043, Reg=0.0148

[Test at epoch 60] Total=0.0617 Rec=0.0037 Lat=0.0388 Phase=0.0043, Reg=0.0149

[Epoch 61] Total=0.4229 Rec=0.0036 Lat=0.0387 Phase=0.0042, Reg=0.0142

[Epoch 62] Total=0.4181 Rec=0.0035 Lat=0.0382 Phase=0.0040, Reg=0.0143

[Epoch 63] Total=0.4135 Rec=0.0035 Lat=0.0378 Phase=0.0039, Reg=0.0141

[Epoch 64] Total=0.4083 Rec=0.0034 Lat=0.0374 Phase=0.0038, Reg=0.0135

[Epoch 65] Total=0.4036 Rec=0.0033 Lat=0.0370 Phase=0.0036, Reg=0.0132

[Test at epoch 65] Total=0.0573 Rec=0.0032 Lat=0.0367 Phase=0.0035, Reg=0.0139

[Epoch 66] Total=0.3992 Rec=0.0032 Lat=0.0367 Phase=0.0035, Reg=0.0129

[Epoch 67] Total=0.3948 Rec=0.0031 Lat=0.0363 Phase=0.0035, Reg=0.0127

[Epoch 68] Total=0.3907 Rec=0.0031 Lat=0.0359 Phase=0.0034, Reg=0.0125

[Epoch 69] Total=0.3865 Rec=0.0030 Lat=0.0356 Phase=0.0033, Reg=0.0121

[Epoch 70] Total=0.3827 Rec=0.0029 Lat=0.0353 Phase=0.0032, Reg=0.0119

[Test at epoch 70] Total=0.0537 Rec=0.0029 Lat=0.0351 Phase=0.0032, Reg=0.0126
 [Epoch 71] Total=0.3788 Rec=0.0029 Lat=0.0349 Phase=0.0032, Reg=0.0117
 [Epoch 72] Total=0.3751 Rec=0.0028 Lat=0.0346 Phase=0.0031, Reg=0.0114
 [Epoch 73] Total=0.3718 Rec=0.0028 Lat=0.0343 Phase=0.0031, Reg=0.0114
 [Epoch 74] Total=0.3683 Rec=0.0027 Lat=0.0340 Phase=0.0030, Reg=0.0112
 [Epoch 75] Total=0.3646 Rec=0.0027 Lat=0.0337 Phase=0.0030, Reg=0.0108
 [Test at epoch 75] Total=0.0508 Rec=0.0026 Lat=0.0336 Phase=0.0029, Reg=0.0117
 [Epoch 76] Total=0.3615 Rec=0.0026 Lat=0.0335 Phase=0.0029, Reg=0.0107
 [Epoch 77] Total=0.3586 Rec=0.0026 Lat=0.0332 Phase=0.0029, Reg=0.0105
 [Epoch 78] Total=0.3559 Rec=0.0026 Lat=0.0329 Phase=0.0028, Reg=0.0107
 [Epoch 79] Total=0.3526 Rec=0.0025 Lat=0.0327 Phase=0.0028, Reg=0.0102
 [Epoch 80] Total=0.3498 Rec=0.0025 Lat=0.0324 Phase=0.0028, Reg=0.0102
 [Test at epoch 80] Total=0.0485 Rec=0.0025 Lat=0.0323 Phase=0.0027, Reg=0.0110
 [Epoch 81] Total=0.3469 Rec=0.0025 Lat=0.0322 Phase=0.0028, Reg=0.0100
 [Epoch 82] Total=0.3444 Rec=0.0025 Lat=0.0319 Phase=0.0027, Reg=0.0100
 [Epoch 83] Total=0.3416 Rec=0.0024 Lat=0.0317 Phase=0.0027, Reg=0.0097
 [Epoch 84] Total=0.3391 Rec=0.0024 Lat=0.0315 Phase=0.0026, Reg=0.0097
 [Epoch 85] Total=0.3367 Rec=0.0024 Lat=0.0313 Phase=0.0026, Reg=0.0093
 [Test at epoch 85] Total=0.0461 Rec=0.0023 Lat=0.0312 Phase=0.0026, Reg=0.0100
 [Epoch 86] Total=0.3348 Rec=0.0024 Lat=0.0311 Phase=0.0026, Reg=0.0096
 [Epoch 87] Total=0.3323 Rec=0.0023 Lat=0.0309 Phase=0.0026, Reg=0.0094
 [Epoch 88] Total=0.3301 Rec=0.0023 Lat=0.0307 Phase=0.0025, Reg=0.0093
 [Epoch 89] Total=0.3276 Rec=0.0023 Lat=0.0305 Phase=0.0025, Reg=0.0090
 [Epoch 90] Total=0.3258 Rec=0.0022 Lat=0.0303 Phase=0.0025, Reg=0.0090
 [Test at epoch 90] Total=0.0444 Rec=0.0022 Lat=0.0302 Phase=0.0024, Reg=0.0095
 [Epoch 91] Total=0.3235 Rec=0.0022 Lat=0.0301 Phase=0.0025, Reg=0.0088
 [Epoch 92] Total=0.3216 Rec=0.0021 Lat=0.0299 Phase=0.0025, Reg=0.0089
 [Epoch 93] Total=0.3196 Rec=0.0021 Lat=0.0298 Phase=0.0024, Reg=0.0085
 [Epoch 94] Total=0.3175 Rec=0.0021 Lat=0.0296 Phase=0.0024, Reg=0.0086
 [Epoch 95] Total=0.3159 Rec=0.0020 Lat=0.0295 Phase=0.0023, Reg=0.0084
 [Test at epoch 95] Total=0.0433 Rec=0.0020 Lat=0.0294 Phase=0.0023, Reg=0.0095
 [Epoch 96] Total=0.3143 Rec=0.0020 Lat=0.0293 Phase=0.0023, Reg=0.0086
 [Epoch 97] Total=0.3124 Rec=0.0020 Lat=0.0292 Phase=0.0023, Reg=0.0082
 [Epoch 98] Total=0.3109 Rec=0.0019 Lat=0.0290 Phase=0.0023, Reg=0.0082
 [Epoch 99] Total=0.3090 Rec=0.0019 Lat=0.0289 Phase=0.0022, Reg=0.0081
 [Epoch 100] Total=0.3077 Rec=0.0019 Lat=0.0287 Phase=0.0022, Reg=0.0082
 [Test at epoch 100] Total=0.0409 Rec=0.0019 Lat=0.0289 Phase=0.0022, Reg=0.0079
 [Epoch 101] Total=0.3062 Rec=0.0018 Lat=0.0286 Phase=0.0022, Reg=0.0080
 [Epoch 102] Total=0.3043 Rec=0.0018 Lat=0.0284 Phase=0.0022, Reg=0.0080
 [Epoch 103] Total=0.3032 Rec=0.0018 Lat=0.0283 Phase=0.0021, Reg=0.0082
 [Epoch 104] Total=0.3013 Rec=0.0018 Lat=0.0282 Phase=0.0021, Reg=0.0078

[Epoch 105] Total=0.3000 Rec=0.0017 Lat=0.0281 Phase=0.0021, Reg=0.0078
 [Test at epoch 105] Total=0.0397 Rec=0.0017 Lat=0.0282 Phase=0.0020,
 Reg=0.0078
 [Epoch 106] Total=0.2983 Rec=0.0017 Lat=0.0279 Phase=0.0020, Reg=0.0077
 [Epoch 107] Total=0.2970 Rec=0.0017 Lat=0.0278 Phase=0.0020, Reg=0.0078
 [Epoch 108] Total=0.2955 Rec=0.0016 Lat=0.0277 Phase=0.0020, Reg=0.0076
 [Epoch 109] Total=0.2945 Rec=0.0016 Lat=0.0275 Phase=0.0020, Reg=0.0079
 [Epoch 110] Total=0.2928 Rec=0.0016 Lat=0.0274 Phase=0.0019, Reg=0.0075
 [Test at epoch 110] Total=0.0391 Rec=0.0016 Lat=0.0274 Phase=0.0019,
 Reg=0.0081
 [Epoch 111] Total=0.2915 Rec=0.0016 Lat=0.0273 Phase=0.0019, Reg=0.0075
 [Epoch 112] Total=0.2898 Rec=0.0016 Lat=0.0272 Phase=0.0019, Reg=0.0074
 [Epoch 113] Total=0.2888 Rec=0.0015 Lat=0.0270 Phase=0.0019, Reg=0.0075
 [Epoch 114] Total=0.2872 Rec=0.0015 Lat=0.0269 Phase=0.0019, Reg=0.0072
 [Epoch 115] Total=0.2861 Rec=0.0015 Lat=0.0268 Phase=0.0018, Reg=0.0074
 [Test at epoch 115] Total=0.0377 Rec=0.0015 Lat=0.0269 Phase=0.0018,
 Reg=0.0075
 [Epoch 116] Total=0.2846 Rec=0.0015 Lat=0.0267 Phase=0.0018, Reg=0.0071
 [Epoch 117] Total=0.2836 Rec=0.0015 Lat=0.0266 Phase=0.0018, Reg=0.0071
 [Epoch 118] Total=0.2822 Rec=0.0015 Lat=0.0265 Phase=0.0018, Reg=0.0071
 [Epoch 119] Total=0.2809 Rec=0.0015 Lat=0.0264 Phase=0.0018, Reg=0.0070
 [Epoch 120] Total=0.2797 Rec=0.0014 Lat=0.0262 Phase=0.0017, Reg=0.0071
 [Test at epoch 120] Total=0.0363 Rec=0.0014 Lat=0.0265 Phase=0.0016,
 Reg=0.0067
 [Epoch 121] Total=0.2789 Rec=0.0014 Lat=0.0262 Phase=0.0017, Reg=0.0070
 [Epoch 122] Total=0.2777 Rec=0.0014 Lat=0.0260 Phase=0.0018, Reg=0.0071
 [Epoch 123] Total=0.2764 Rec=0.0014 Lat=0.0259 Phase=0.0017, Reg=0.0070
 [Epoch 124] Total=0.2753 Rec=0.0014 Lat=0.0258 Phase=0.0017, Reg=0.0069
 [Epoch 125] Total=0.2739 Rec=0.0014 Lat=0.0257 Phase=0.0017, Reg=0.0067
 [Test at epoch 125] Total=0.0359 Rec=0.0014 Lat=0.0259 Phase=0.0017,
 Reg=0.0070
 [Epoch 126] Total=0.2730 Rec=0.0014 Lat=0.0256 Phase=0.0017, Reg=0.0067
 [Epoch 127] Total=0.2717 Rec=0.0014 Lat=0.0255 Phase=0.0017, Reg=0.0068
 [Epoch 128] Total=0.2708 Rec=0.0014 Lat=0.0254 Phase=0.0017, Reg=0.0067
 [Epoch 129] Total=0.2700 Rec=0.0014 Lat=0.0253 Phase=0.0017, Reg=0.0069
 [Epoch 130] Total=0.2689 Rec=0.0013 Lat=0.0253 Phase=0.0017, Reg=0.0067
 [Test at epoch 130] Total=0.0355 Rec=0.0013 Lat=0.0253 Phase=0.0016,
 Reg=0.0073
 [Epoch 131] Total=0.2676 Rec=0.0013 Lat=0.0251 Phase=0.0016, Reg=0.0066
 [Epoch 132] Total=0.2666 Rec=0.0013 Lat=0.0251 Phase=0.0016, Reg=0.0065
 [Epoch 133] Total=0.2654 Rec=0.0013 Lat=0.0249 Phase=0.0016, Reg=0.0065
 [Epoch 134] Total=0.2645 Rec=0.0013 Lat=0.0249 Phase=0.0016, Reg=0.0065
 [Epoch 135] Total=0.2636 Rec=0.0013 Lat=0.0248 Phase=0.0016, Reg=0.0065
 [Test at epoch 135] Total=0.0340 Rec=0.0013 Lat=0.0249 Phase=0.0016,
 Reg=0.0062
 [Epoch 136] Total=0.2627 Rec=0.0013 Lat=0.0247 Phase=0.0016, Reg=0.0065
 [Epoch 137] Total=0.2614 Rec=0.0013 Lat=0.0246 Phase=0.0016, Reg=0.0064
 [Epoch 138] Total=0.2604 Rec=0.0013 Lat=0.0245 Phase=0.0016, Reg=0.0063

[Epoch 139] Total=0.2596 Rec=0.0012 Lat=0.0244 Phase=0.0015, Reg=0.0063
 [Epoch 140] Total=0.2588 Rec=0.0012 Lat=0.0243 Phase=0.0015, Reg=0.0064
 [Test at epoch 140] Total=0.0339 Rec=0.0012 Lat=0.0245 Phase=0.0016,
 Reg=0.0065
 [Epoch 141] Total=0.2580 Rec=0.0012 Lat=0.0242 Phase=0.0016, Reg=0.0064
 [Epoch 142] Total=0.2565 Rec=0.0012 Lat=0.0242 Phase=0.0015, Reg=0.0061
 [Epoch 143] Total=0.2564 Rec=0.0012 Lat=0.0241 Phase=0.0015, Reg=0.0066
 [Epoch 144] Total=0.2550 Rec=0.0012 Lat=0.0240 Phase=0.0015, Reg=0.0062
 [Epoch 145] Total=0.2542 Rec=0.0012 Lat=0.0239 Phase=0.0015, Reg=0.0062
 [Test at epoch 145] Total=0.0337 Rec=0.0013 Lat=0.0239 Phase=0.0015,
 Reg=0.0070
 [Epoch 146] Total=0.2529 Rec=0.0012 Lat=0.0238 Phase=0.0015, Reg=0.0061
 [Epoch 147] Total=0.2523 Rec=0.0012 Lat=0.0237 Phase=0.0015, Reg=0.0061
 [Epoch 148] Total=0.2519 Rec=0.0012 Lat=0.0237 Phase=0.0015, Reg=0.0063
 [Epoch 149] Total=0.2501 Rec=0.0012 Lat=0.0236 Phase=0.0014, Reg=0.0058
 [Epoch 150] Total=0.2499 Rec=0.0012 Lat=0.0235 Phase=0.0015, Reg=0.0062
 [Test at epoch 150] Total=0.0327 Rec=0.0011 Lat=0.0236 Phase=0.0014,
 Reg=0.0066
 [Epoch 151] Total=0.2489 Rec=0.0011 Lat=0.0234 Phase=0.0014, Reg=0.0061
 [Epoch 152] Total=0.2477 Rec=0.0011 Lat=0.0233 Phase=0.0014, Reg=0.0059
 [Epoch 153] Total=0.2466 Rec=0.0011 Lat=0.0233 Phase=0.0014, Reg=0.0058
 [Epoch 154] Total=0.2458 Rec=0.0011 Lat=0.0232 Phase=0.0014, Reg=0.0057
 [Epoch 155] Total=0.2455 Rec=0.0011 Lat=0.0231 Phase=0.0014, Reg=0.0061
 [Test at epoch 155] Total=0.0323 Rec=0.0011 Lat=0.0231 Phase=0.0014,
 Reg=0.0067
 [Epoch 156] Total=0.2443 Rec=0.0011 Lat=0.0230 Phase=0.0014, Reg=0.0057
 [Epoch 157] Total=0.2437 Rec=0.0011 Lat=0.0229 Phase=0.0014, Reg=0.0061
 [Epoch 158] Total=0.2429 Rec=0.0011 Lat=0.0229 Phase=0.0014, Reg=0.0057
 [Epoch 159] Total=0.2417 Rec=0.0011 Lat=0.0228 Phase=0.0014, Reg=0.0058
 [Epoch 160] Total=0.2412 Rec=0.0011 Lat=0.0227 Phase=0.0014, Reg=0.0058
 [Test at epoch 160] Total=0.0311 Rec=0.0011 Lat=0.0228 Phase=0.0013,
 Reg=0.0059
 [Epoch 161] Total=0.2402 Rec=0.0011 Lat=0.0226 Phase=0.0014, Reg=0.0056
 [Epoch 162] Total=0.2393 Rec=0.0011 Lat=0.0225 Phase=0.0014, Reg=0.0057
 [Epoch 163] Total=0.2388 Rec=0.0011 Lat=0.0225 Phase=0.0014, Reg=0.0058
 [Epoch 164] Total=0.2378 Rec=0.0011 Lat=0.0224 Phase=0.0014, Reg=0.0057
 [Epoch 165] Total=0.2368 Rec=0.0010 Lat=0.0223 Phase=0.0014, Reg=0.0055
 [Test at epoch 165] Total=0.0313 Rec=0.0010 Lat=0.0224 Phase=0.0014,
 Reg=0.0065
 [Epoch 166] Total=0.2359 Rec=0.0010 Lat=0.0222 Phase=0.0014, Reg=0.0055
 [Epoch 167] Total=0.2356 Rec=0.0010 Lat=0.0222 Phase=0.0014, Reg=0.0057
 [Epoch 168] Total=0.2345 Rec=0.0010 Lat=0.0221 Phase=0.0013, Reg=0.0055
 [Epoch 169] Total=0.2338 Rec=0.0010 Lat=0.0220 Phase=0.0014, Reg=0.0056
 [Epoch 170] Total=0.2329 Rec=0.0010 Lat=0.0220 Phase=0.0013, Reg=0.0055
 [Test at epoch 170] Total=0.0309 Rec=0.0010 Lat=0.0220 Phase=0.0014,
 Reg=0.0065
 [Epoch 171] Total=0.2324 Rec=0.0010 Lat=0.0219 Phase=0.0013, Reg=0.0055
 [Epoch 172] Total=0.2314 Rec=0.0010 Lat=0.0218 Phase=0.0014, Reg=0.0055

[Epoch 173] Total=0.2303 Rec=0.0010 Lat=0.0217 Phase=0.0014, Reg=0.0053
 [Epoch 174] Total=0.2297 Rec=0.0010 Lat=0.0217 Phase=0.0014, Reg=0.0054
 [Epoch 175] Total=0.2290 Rec=0.0010 Lat=0.0216 Phase=0.0013, Reg=0.0055
 [Test at epoch 175] Total=0.0296 Rec=0.0010 Lat=0.0218 Phase=0.0013,
 Reg=0.0055
 [Epoch 176] Total=0.2284 Rec=0.0010 Lat=0.0215 Phase=0.0013, Reg=0.0055
 [Epoch 177] Total=0.2272 Rec=0.0010 Lat=0.0214 Phase=0.0013, Reg=0.0053
 [Epoch 178] Total=0.2267 Rec=0.0010 Lat=0.0214 Phase=0.0014, Reg=0.0054
 [Epoch 179] Total=0.2253 Rec=0.0010 Lat=0.0212 Phase=0.0013, Reg=0.0053
 [Epoch 180] Total=0.2251 Rec=0.0010 Lat=0.0212 Phase=0.0013, Reg=0.0053
 [Test at epoch 180] Total=0.0298 Rec=0.0009 Lat=0.0213 Phase=0.0013,
 Reg=0.0063
 [Epoch 181] Total=0.2246 Rec=0.0009 Lat=0.0211 Phase=0.0014, Reg=0.0055
 [Epoch 182] Total=0.2236 Rec=0.0009 Lat=0.0211 Phase=0.0013, Reg=0.0052
 [Epoch 183] Total=0.2229 Rec=0.0009 Lat=0.0210 Phase=0.0013, Reg=0.0052
 [Epoch 184] Total=0.2220 Rec=0.0009 Lat=0.0209 Phase=0.0013, Reg=0.0053
 [Epoch 185] Total=0.2213 Rec=0.0009 Lat=0.0209 Phase=0.0013, Reg=0.0052
 [Test at epoch 185] Total=0.0292 Rec=0.0009 Lat=0.0210 Phase=0.0013,
 Reg=0.0059
 [Epoch 186] Total=0.2209 Rec=0.0009 Lat=0.0208 Phase=0.0013, Reg=0.0054
 [Epoch 187] Total=0.2199 Rec=0.0009 Lat=0.0207 Phase=0.0013, Reg=0.0052
 [Epoch 188] Total=0.2193 Rec=0.0009 Lat=0.0207 Phase=0.0014, Reg=0.0050
 [Epoch 189] Total=0.2185 Rec=0.0009 Lat=0.0206 Phase=0.0013, Reg=0.0052
 [Epoch 190] Total=0.2179 Rec=0.0009 Lat=0.0205 Phase=0.0014, Reg=0.0051
 [Test at epoch 190] Total=0.0287 Rec=0.0009 Lat=0.0207 Phase=0.0014,
 Reg=0.0057
 [Epoch 191] Total=0.2171 Rec=0.0009 Lat=0.0205 Phase=0.0013, Reg=0.0051
 [Epoch 192] Total=0.2166 Rec=0.0009 Lat=0.0204 Phase=0.0013, Reg=0.0051
 [Epoch 193] Total=0.2162 Rec=0.0009 Lat=0.0203 Phase=0.0013, Reg=0.0053
 [Epoch 194] Total=0.2153 Rec=0.0009 Lat=0.0203 Phase=0.0013, Reg=0.0051
 [Epoch 195] Total=0.2146 Rec=0.0009 Lat=0.0202 Phase=0.0013, Reg=0.0051
 [Test at epoch 195] Total=0.0281 Rec=0.0009 Lat=0.0203 Phase=0.0013,
 Reg=0.0056
 [Epoch 196] Total=0.2137 Rec=0.0009 Lat=0.0202 Phase=0.0013, Reg=0.0050
 [Epoch 197] Total=0.2130 Rec=0.0009 Lat=0.0201 Phase=0.0013, Reg=0.0048
 [Epoch 198] Total=0.2127 Rec=0.0009 Lat=0.0200 Phase=0.0013, Reg=0.0050
 [Epoch 199] Total=0.2117 Rec=0.0009 Lat=0.0200 Phase=0.0013, Reg=0.0049
 [Epoch 200] Total=0.2118 Rec=0.0009 Lat=0.0199 Phase=0.0014, Reg=0.0051
 [Test at epoch 200] Total=0.0277 Rec=0.0009 Lat=0.0200 Phase=0.0013,
 Reg=0.0054
 [Epoch 201] Total=0.2113 Rec=0.0008 Lat=0.0199 Phase=0.0013, Reg=0.0052
 [Epoch 202] Total=0.2102 Rec=0.0008 Lat=0.0198 Phase=0.0013, Reg=0.0050
 [Epoch 203] Total=0.2095 Rec=0.0008 Lat=0.0197 Phase=0.0013, Reg=0.0050
 [Epoch 204] Total=0.2090 Rec=0.0008 Lat=0.0197 Phase=0.0013, Reg=0.0049
 [Epoch 205] Total=0.2082 Rec=0.0008 Lat=0.0196 Phase=0.0013, Reg=0.0049
 [Test at epoch 205] Total=0.0267 Rec=0.0008 Lat=0.0200 Phase=0.0013,
 Reg=0.0045
 [Epoch 206] Total=0.2080 Rec=0.0008 Lat=0.0196 Phase=0.0013, Reg=0.0050

[Epoch 207] Total=0.2068 Rec=0.0008 Lat=0.0195 Phase=0.0013, Reg=0.0047
 [Epoch 208] Total=0.2067 Rec=0.0008 Lat=0.0195 Phase=0.0013, Reg=0.0049
 [Epoch 209] Total=0.2061 Rec=0.0008 Lat=0.0194 Phase=0.0013, Reg=0.0047
 [Epoch 210] Total=0.2059 Rec=0.0008 Lat=0.0194 Phase=0.0013, Reg=0.0051
 [Test at epoch 210] Total=0.0269 Rec=0.0008 Lat=0.0194 Phase=0.0013,
 Reg=0.0054
 [Epoch 211] Total=0.2053 Rec=0.0008 Lat=0.0193 Phase=0.0013, Reg=0.0050
 [Epoch 212] Total=0.2046 Rec=0.0008 Lat=0.0193 Phase=0.0013, Reg=0.0049
 [Epoch 213] Total=0.2041 Rec=0.0008 Lat=0.0192 Phase=0.0013, Reg=0.0048
 [Epoch 214] Total=0.2034 Rec=0.0008 Lat=0.0192 Phase=0.0013, Reg=0.0049
 [Epoch 215] Total=0.2030 Rec=0.0008 Lat=0.0191 Phase=0.0013, Reg=0.0050
 [Test at epoch 215] Total=0.0267 Rec=0.0008 Lat=0.0192 Phase=0.0013,
 Reg=0.0053
 [Epoch 216] Total=0.2024 Rec=0.0008 Lat=0.0191 Phase=0.0014, Reg=0.0048
 [Epoch 217] Total=0.2023 Rec=0.0008 Lat=0.0190 Phase=0.0013, Reg=0.0049
 [Epoch 218] Total=0.2018 Rec=0.0008 Lat=0.0190 Phase=0.0013, Reg=0.0050
 [Epoch 219] Total=0.2006 Rec=0.0008 Lat=0.0189 Phase=0.0013, Reg=0.0047
 [Epoch 220] Total=0.2002 Rec=0.0008 Lat=0.0189 Phase=0.0013, Reg=0.0048
 [Test at epoch 220] Total=0.0264 Rec=0.0008 Lat=0.0190 Phase=0.0013,
 Reg=0.0053
 [Epoch 221] Total=0.1997 Rec=0.0008 Lat=0.0188 Phase=0.0013, Reg=0.0047
 [Epoch 222] Total=0.1992 Rec=0.0008 Lat=0.0188 Phase=0.0013, Reg=0.0046
 [Epoch 223] Total=0.1988 Rec=0.0008 Lat=0.0187 Phase=0.0013, Reg=0.0047
 [Epoch 224] Total=0.1982 Rec=0.0008 Lat=0.0187 Phase=0.0013, Reg=0.0047
 [Epoch 225] Total=0.1978 Rec=0.0008 Lat=0.0186 Phase=0.0013, Reg=0.0047
 [Test at epoch 225] Total=0.0262 Rec=0.0008 Lat=0.0187 Phase=0.0013,
 Reg=0.0054
 [Epoch 226] Total=0.1972 Rec=0.0008 Lat=0.0186 Phase=0.0013, Reg=0.0045
 [Epoch 227] Total=0.1971 Rec=0.0008 Lat=0.0186 Phase=0.0013, Reg=0.0047
 [Epoch 228] Total=0.1967 Rec=0.0008 Lat=0.0185 Phase=0.0013, Reg=0.0049
 [Epoch 229] Total=0.1959 Rec=0.0008 Lat=0.0185 Phase=0.0013, Reg=0.0045
 [Epoch 230] Total=0.1956 Rec=0.0008 Lat=0.0184 Phase=0.0013, Reg=0.0048
 [Test at epoch 230] Total=0.0252 Rec=0.0008 Lat=0.0187 Phase=0.0013,
 Reg=0.0045
 [Epoch 231] Total=0.1949 Rec=0.0008 Lat=0.0184 Phase=0.0013, Reg=0.0046
 [Epoch 232] Total=0.1950 Rec=0.0008 Lat=0.0183 Phase=0.0013, Reg=0.0048
 [Epoch 233] Total=0.1943 Rec=0.0007 Lat=0.0183 Phase=0.0013, Reg=0.0047
 [Epoch 234] Total=0.1939 Rec=0.0007 Lat=0.0182 Phase=0.0013, Reg=0.0047
 [Epoch 235] Total=0.1929 Rec=0.0007 Lat=0.0182 Phase=0.0013, Reg=0.0045
 [Test at epoch 235] Total=0.0262 Rec=0.0007 Lat=0.0181 Phase=0.0013,
 Reg=0.0061
 [Epoch 236] Total=0.1925 Rec=0.0007 Lat=0.0181 Phase=0.0013, Reg=0.0045
 [Epoch 237] Total=0.1924 Rec=0.0007 Lat=0.0181 Phase=0.0013, Reg=0.0046
 [Epoch 238] Total=0.1914 Rec=0.0007 Lat=0.0181 Phase=0.0013, Reg=0.0043
 [Epoch 239] Total=0.1909 Rec=0.0007 Lat=0.0180 Phase=0.0013, Reg=0.0043
 [Epoch 240] Total=0.1907 Rec=0.0007 Lat=0.0180 Phase=0.0013, Reg=0.0044
 [Test at epoch 240] Total=0.0258 Rec=0.0007 Lat=0.0179 Phase=0.0014,
 Reg=0.0058

[Epoch 241] Total=0.1902 Rec=0.0007 Lat=0.0179 Phase=0.0013, Reg=0.0043
 [Epoch 242] Total=0.1897 Rec=0.0007 Lat=0.0179 Phase=0.0013, Reg=0.0044
 [Epoch 243] Total=0.1894 Rec=0.0007 Lat=0.0178 Phase=0.0013, Reg=0.0045
 [Epoch 244] Total=0.1892 Rec=0.0007 Lat=0.0178 Phase=0.0013, Reg=0.0046
 [Epoch 245] Total=0.1884 Rec=0.0007 Lat=0.0178 Phase=0.0013, Reg=0.0043
 [Test at epoch 245] Total=0.0259 Rec=0.0007 Lat=0.0177 Phase=0.0013,
 Reg=0.0062
 [Epoch 246] Total=0.1883 Rec=0.0007 Lat=0.0177 Phase=0.0013, Reg=0.0045
 [Epoch 247] Total=0.1878 Rec=0.0007 Lat=0.0177 Phase=0.0013, Reg=0.0045
 [Epoch 248] Total=0.1876 Rec=0.0007 Lat=0.0176 Phase=0.0013, Reg=0.0045
 [Epoch 249] Total=0.1873 Rec=0.0007 Lat=0.0176 Phase=0.0013, Reg=0.0046
 [Epoch 250] Total=0.1869 Rec=0.0007 Lat=0.0176 Phase=0.0013, Reg=0.0046
 [Test at epoch 250] Total=0.0257 Rec=0.0008 Lat=0.0176 Phase=0.0013,
 Reg=0.0060
 [Epoch 251] Total=0.1866 Rec=0.0007 Lat=0.0175 Phase=0.0013, Reg=0.0046
 [Epoch 252] Total=0.1855 Rec=0.0007 Lat=0.0175 Phase=0.0013, Reg=0.0043
 [Epoch 253] Total=0.1858 Rec=0.0007 Lat=0.0174 Phase=0.0013, Reg=0.0047
 [Epoch 254] Total=0.1847 Rec=0.0007 Lat=0.0174 Phase=0.0013, Reg=0.0043
 [Epoch 255] Total=0.1847 Rec=0.0007 Lat=0.0174 Phase=0.0013, Reg=0.0045
 [Test at epoch 255] Total=0.0249 Rec=0.0007 Lat=0.0174 Phase=0.0013,
 Reg=0.0055
 [Epoch 256] Total=0.1841 Rec=0.0007 Lat=0.0173 Phase=0.0013, Reg=0.0044
 [Epoch 257] Total=0.1835 Rec=0.0007 Lat=0.0173 Phase=0.0013, Reg=0.0044
 [Epoch 258] Total=0.1834 Rec=0.0007 Lat=0.0173 Phase=0.0013, Reg=0.0043
 [Epoch 259] Total=0.1835 Rec=0.0007 Lat=0.0172 Phase=0.0013, Reg=0.0046
 [Epoch 260] Total=0.1826 Rec=0.0007 Lat=0.0172 Phase=0.0013, Reg=0.0043
 [Test at epoch 260] Total=0.0249 Rec=0.0007 Lat=0.0172 Phase=0.0013,
 Reg=0.0057
 [Epoch 261] Total=0.1824 Rec=0.0007 Lat=0.0172 Phase=0.0013, Reg=0.0044
 [Epoch 262] Total=0.1818 Rec=0.0007 Lat=0.0171 Phase=0.0013, Reg=0.0044
 [Epoch 263] Total=0.1813 Rec=0.0007 Lat=0.0171 Phase=0.0013, Reg=0.0043
 [Epoch 264] Total=0.1818 Rec=0.0007 Lat=0.0171 Phase=0.0012, Reg=0.0045
 [Epoch 265] Total=0.1812 Rec=0.0007 Lat=0.0170 Phase=0.0013, Reg=0.0046
 [Test at epoch 265] Total=0.0237 Rec=0.0007 Lat=0.0172 Phase=0.0013,
 Reg=0.0045
 [Epoch 266] Total=0.1802 Rec=0.0007 Lat=0.0170 Phase=0.0013, Reg=0.0043
 [Epoch 267] Total=0.1800 Rec=0.0007 Lat=0.0169 Phase=0.0013, Reg=0.0043
 [Epoch 268] Total=0.1798 Rec=0.0007 Lat=0.0169 Phase=0.0013, Reg=0.0044
 [Epoch 269] Total=0.1792 Rec=0.0007 Lat=0.0169 Phase=0.0013, Reg=0.0042
 [Epoch 270] Total=0.1786 Rec=0.0007 Lat=0.0168 Phase=0.0013, Reg=0.0042
 [Test at epoch 270] Total=0.0236 Rec=0.0007 Lat=0.0170 Phase=0.0013,
 Reg=0.0046
 [Epoch 271] Total=0.1789 Rec=0.0007 Lat=0.0168 Phase=0.0013, Reg=0.0044
 [Epoch 272] Total=0.1778 Rec=0.0007 Lat=0.0168 Phase=0.0013, Reg=0.0042
 [Epoch 273] Total=0.1775 Rec=0.0007 Lat=0.0167 Phase=0.0012, Reg=0.0042
 [Epoch 274] Total=0.1771 Rec=0.0007 Lat=0.0167 Phase=0.0013, Reg=0.0040
 [Epoch 275] Total=0.1771 Rec=0.0007 Lat=0.0167 Phase=0.0013, Reg=0.0043
 [Test at epoch 275] Total=0.0238 Rec=0.0007 Lat=0.0167 Phase=0.0012,

```

Reg=0.0053
[Epoch 276] Total=0.1764 Rec=0.0007 Lat=0.0166 Phase=0.0012, Reg=0.0042
[Epoch 277] Total=0.1762 Rec=0.0007 Lat=0.0166 Phase=0.0013, Reg=0.0042
[Epoch 278] Total=0.1765 Rec=0.0007 Lat=0.0166 Phase=0.0012, Reg=0.0045
[Epoch 279] Total=0.1760 Rec=0.0007 Lat=0.0165 Phase=0.0013, Reg=0.0044
[Epoch 280] Total=0.1752 Rec=0.0007 Lat=0.0165 Phase=0.0012, Reg=0.0042
  [Test at epoch 280] Total=0.0242 Rec=0.0007 Lat=0.0164 Phase=0.0013,
Reg=0.0058
[Epoch 281] Total=0.1744 Rec=0.0007 Lat=0.0164 Phase=0.0013, Reg=0.0040
[Epoch 282] Total=0.1747 Rec=0.0007 Lat=0.0164 Phase=0.0012, Reg=0.0044
[Epoch 283] Total=0.1744 Rec=0.0007 Lat=0.0164 Phase=0.0013, Reg=0.0044
[Epoch 284] Total=0.1736 Rec=0.0007 Lat=0.0163 Phase=0.0012, Reg=0.0041
[Epoch 285] Total=0.1734 Rec=0.0007 Lat=0.0163 Phase=0.0013, Reg=0.0042
  [Test at epoch 285] Total=0.0230 Rec=0.0008 Lat=0.0164 Phase=0.0016,
Reg=0.0042
[Epoch 286] Total=0.1728 Rec=0.0007 Lat=0.0163 Phase=0.0013, Reg=0.0040
[Epoch 287] Total=0.1719 Rec=0.0007 Lat=0.0162 Phase=0.0012, Reg=0.0039
[Epoch 288] Total=0.1721 Rec=0.0007 Lat=0.0162 Phase=0.0012, Reg=0.0041
[Epoch 289] Total=0.1719 Rec=0.0007 Lat=0.0161 Phase=0.0012, Reg=0.0043
[Epoch 290] Total=0.1715 Rec=0.0007 Lat=0.0161 Phase=0.0012, Reg=0.0041
  [Test at epoch 290] Total=0.0239 Rec=0.0007 Lat=0.0160 Phase=0.0013,
Reg=0.0059
[Epoch 291] Total=0.1712 Rec=0.0007 Lat=0.0161 Phase=0.0012, Reg=0.0042
[Epoch 292] Total=0.1705 Rec=0.0007 Lat=0.0160 Phase=0.0012, Reg=0.0041
[Epoch 293] Total=0.1705 Rec=0.0007 Lat=0.0160 Phase=0.0013, Reg=0.0042
[Epoch 294] Total=0.1700 Rec=0.0007 Lat=0.0160 Phase=0.0013, Reg=0.0041
[Epoch 295] Total=0.1697 Rec=0.0007 Lat=0.0159 Phase=0.0012, Reg=0.0042
  [Test at epoch 295] Total=0.0232 Rec=0.0007 Lat=0.0159 Phase=0.0012,
Reg=0.0053
[Epoch 296] Total=0.1690 Rec=0.0007 Lat=0.0159 Phase=0.0012, Reg=0.0041
[Epoch 297] Total=0.1686 Rec=0.0007 Lat=0.0159 Phase=0.0012, Reg=0.0040
[Epoch 298] Total=0.1683 Rec=0.0007 Lat=0.0158 Phase=0.0012, Reg=0.0040
[Epoch 299] Total=0.1683 Rec=0.0007 Lat=0.0158 Phase=0.0012, Reg=0.0042

```

```

[72]: def plot_total_loss(train_total_losses, test_total_losses):
    plt.figure(figsize=(8, 5))
    plt.plot(train_total_losses, label="Train Total Loss")
    plt.plot(test_total_losses, label="Test Total Loss")
    plt.xlabel("Epoch")
    plt.ylabel("Loss (log scale)")
    plt.title("Total Loss - Train vs Test")
    plt.yscale("log") # Echelle logarithmique
    plt.legend()
    plt.grid(True, which='both', linestyle='--', linewidth=0.5)
    plt.tight_layout()
    plt.show()

```

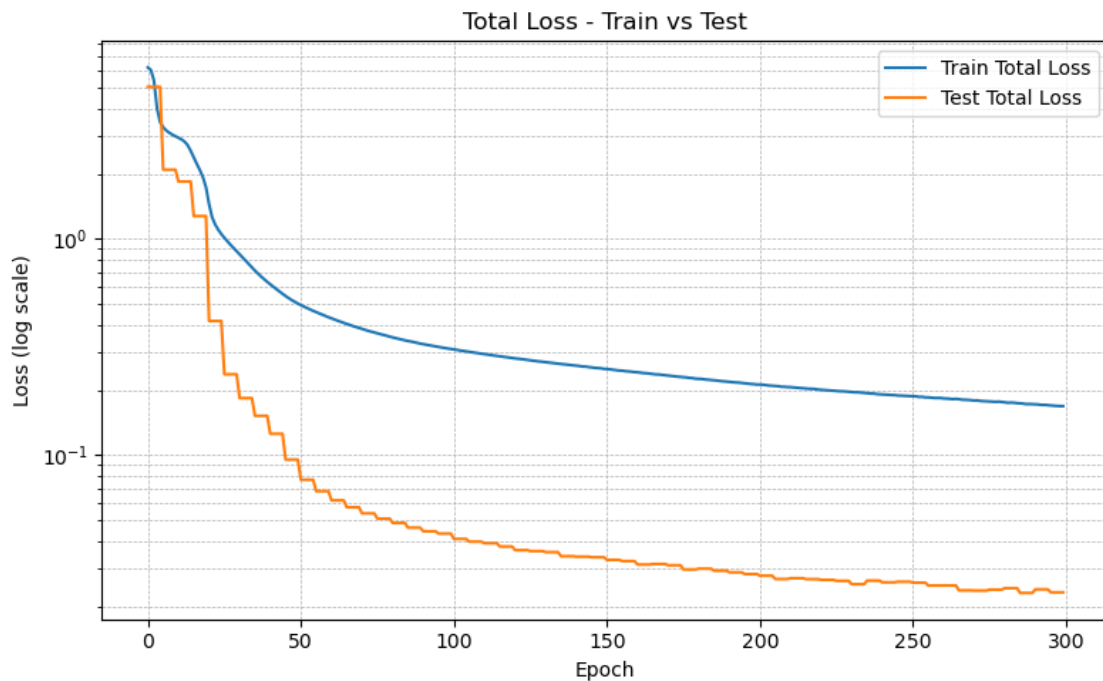
```

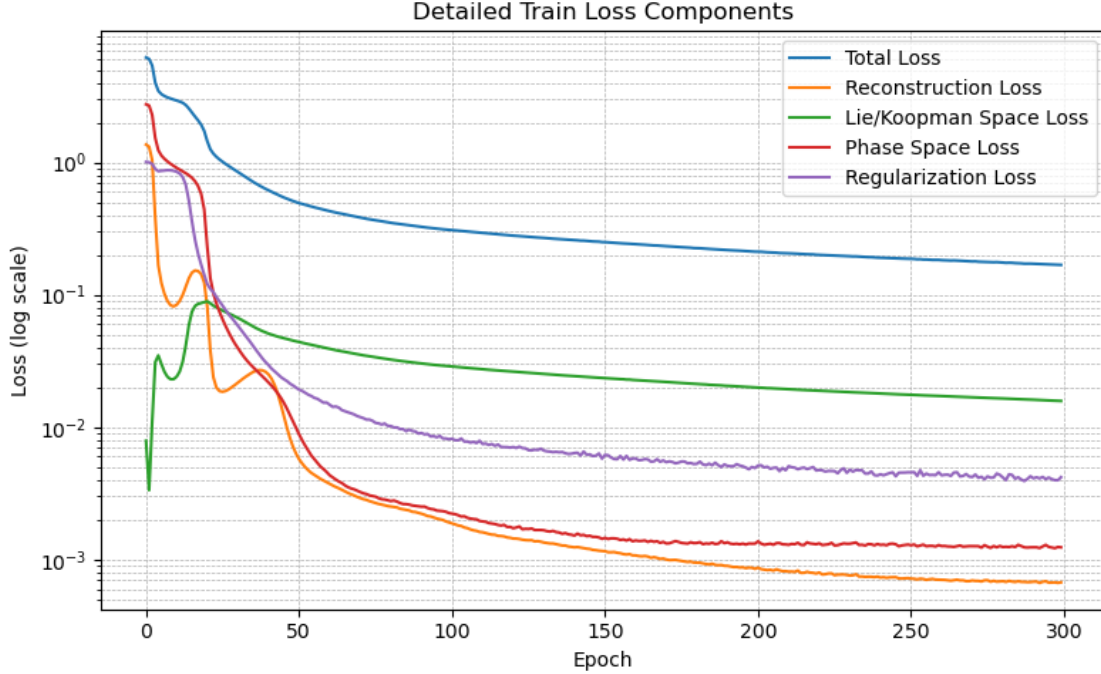
def plot_detailed_train_loss(train_total, train_rec, train_lat, train_phase,
    ↪train_reg):
    plt.figure(figsize=(8, 5))
    plt.plot(train_total, label="Total Loss")
    plt.plot(train_rec, label="Reconstruction Loss")
    plt.plot(train_lat, label="Lie/Koopman Space Loss")
    plt.plot(train_phase, label="Phase Space Loss")
    plt.plot(train_reg, label="Regularization Loss")

    plt.xlabel("Epoch")
    plt.ylabel("Loss (log scale)")
    plt.title("Detailed Train Loss Components")
    plt.yscale("log") # Echelle logarithmique
    plt.legend()
    plt.grid(True, which='both', linestyle='--', linewidth=0.5)
    plt.tight_layout()
    plt.show()

plot_total_loss(train_total_losses, test_total_losses)
plot_detailed_train_loss(train_total_losses, train_rec_losses, train_lat_losses,
    ↪train_phase_losses, train_reg_losses)

```





For this continuous-time version, we trained the model over **300 epochs**, again tracking all components of the loss.

The final configuration we retained was:

- $\lambda_{\text{rec}} = 1.0$
- $\lambda_{\text{koopman}} = 10.0$
- $\lambda_{\text{phase}} = 1.0$
- $\lambda_{\text{reg}} = 2.0$

We chose a higher weight for the Koopman (Lie) loss because it was consistently dominating the overall error in raw value. This scaling helped balance the gradient magnitudes and improve convergence.

As shown in the table below, the test loss remains **consistently lower than the training loss**, which may seem counterintuitive at first. However, this is explained by the **weighting scheme**: the total loss is a weighted sum, and test and train sets can contain different distributions of high-variance inputs. More importantly, **all individual loss terms** (including reconstruction, latent prediction, phase prediction, and regularization) are **well-controlled and stable** across epochs (particularly after epoch 40, where the training dynamics clearly settle into a steady regime).

The training and test curves evolve smoothly and confirm that the model does **not overfit** despite the longer training time.

The final predictions are also visually very accurate when comparing the learned dy-

namics to the ground truth vector field.

Split	Total Loss	Reconstruction	Koopman (Lie)	Phase Space	Regularization
Train	0.0219	0.0007	0.0158	0.0012	0.0042
Test	0.0231	0.0007	0.0159	0.0012	0.0053

1.3.3 Verification

Question 6. : As in the **Question 3.** we want to ensure the Lie operator is stable. This can be verified by checking that the real part of the eigenvalues is negative. Plot the relevant eigenvalues of the Lie operator. You can use the `numpy.linalg.eig` function to retrieve the eigenvalues of a matrix.

```
[73]: def plot_lie_eigenvalues_combined(lie_operator):
    with torch.no_grad():
        # R  cup  ration de la matrice de l'op  rateur de Lie
        Lie_matrix = lie_operator.Psi - lie_operator.Psi.T - torch.diag(torch.
        ↪abs(lie_operator.Gamma))
        Lie_np = Lie_matrix.cpu().numpy()

        # Calcul des valeurs propres
        eigvals = np.linalg.eigvals(Lie_np)

        # Cr  ation de la figure et des axes pour la vue globale
        fig, ax1 = plt.subplots(figsize=(8, 8))

        # --- Vue globale ---
        ax1.scatter(eigvals.real, eigvals.imag, color="blue",
        ↪label="Eigenvalues")
        unit_circle = plt.Circle((0, 0), 1.0, color='r', fill=False,
        ↪linestyle="--", label="Unit Circle")
        ax1.add_artist(unit_circle)
        ax1.set_xlim(-1.1, 1.1)
        ax1.set_ylim(-2, 2)
        ax1.set_aspect('equal')
        ax1.set_title("Full spectrum")
        ax1.set_xlabel("Re")
        ax1.set_ylabel("Im")
        ax1.grid(True)
        ax1.legend()

        plt.tight_layout()
        plt.show()

def plot_lie_eigenvalue_real_parts(lie_operator):
```

```

with torch.no_grad():
    Lie_matrix = lie_operator.Psi - lie_operator.Psi.T - torch.diag(torch.
↪abs(lie_operator.Gamma))
    Lie_np = Lie_matrix.cpu().numpy()

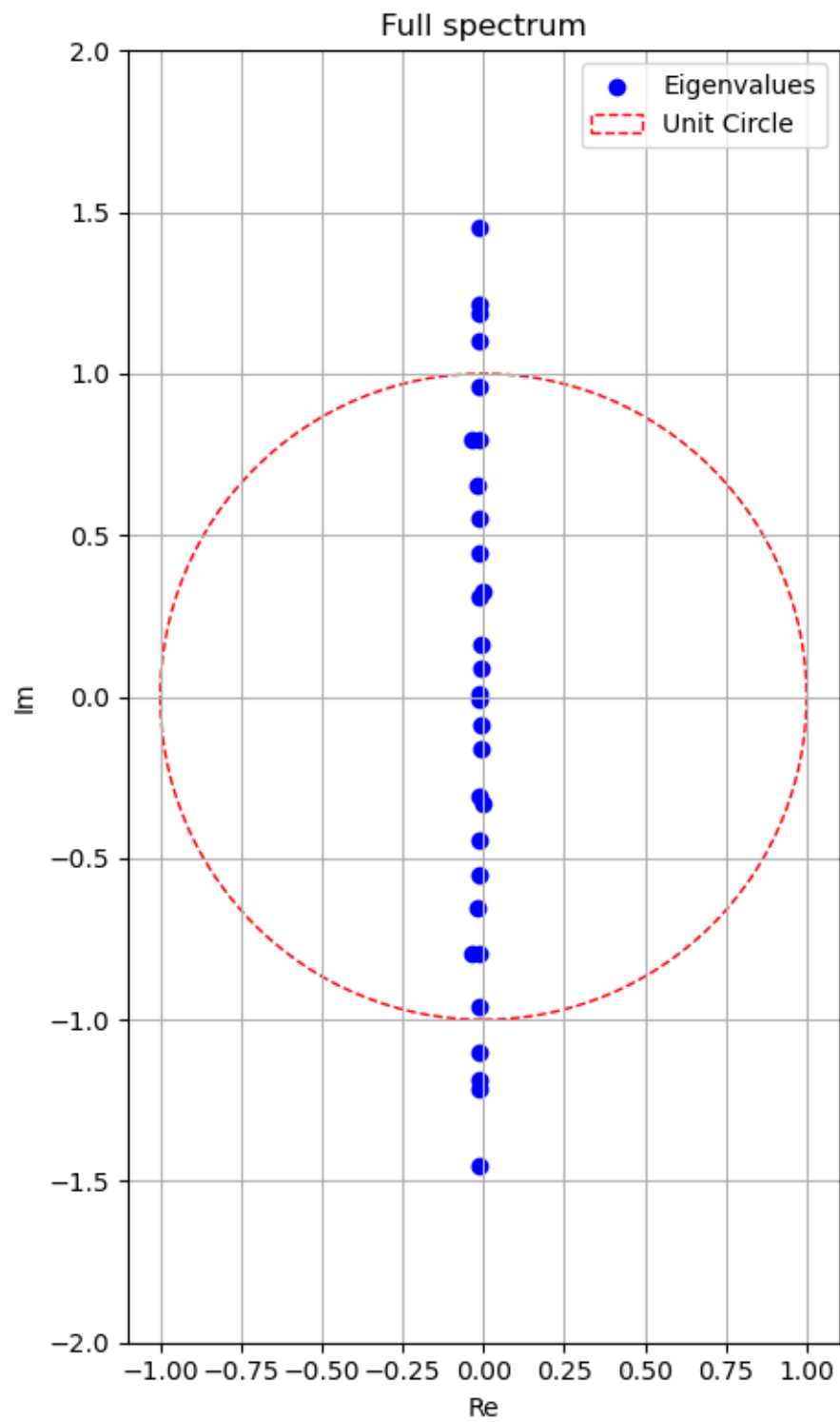
    eigvals = np.linalg.eigvals(Lie_np)

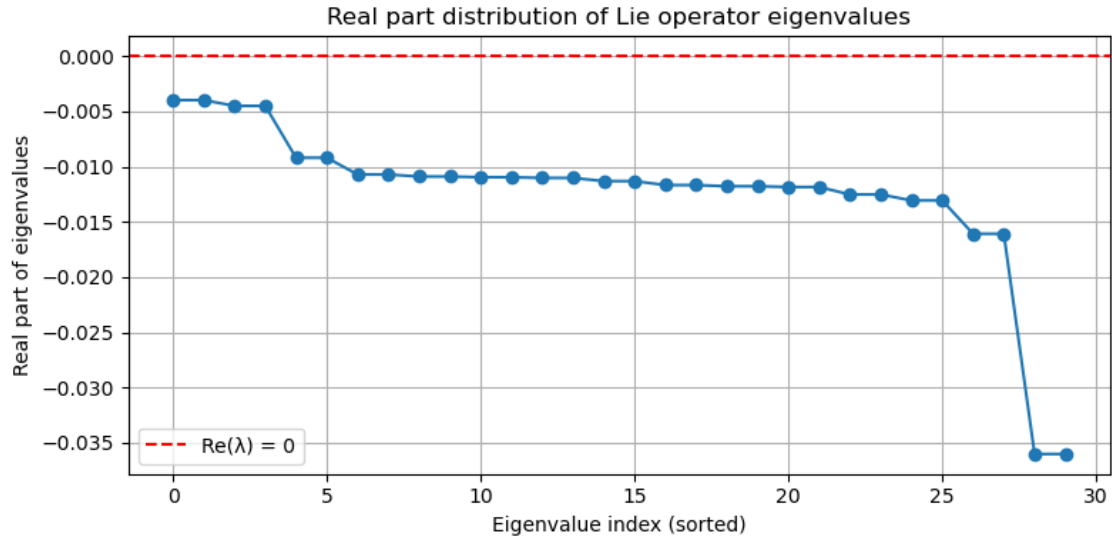
    real_parts = np.real(eigvals)
    real_sorted = np.sort(real_parts)[::-1] # tri décroissant
    # print(real_sorted)

    plt.figure(figsize=(8, 4))
    plt.plot(real_sorted, marker='o')
    plt.axhline(0.0, color='r', linestyle='--', label='Re( $\lambda$ ) = 0')
    plt.xlabel("Eigenvalue index (sorted)")
    plt.ylabel("Real part of eigenvalues")
    plt.title("Real part distribution of Lie operator eigenvalues")
    plt.grid(True)
    plt.legend()
    plt.tight_layout()
    plt.show()

plot_lie_eigenvalues_combined(lie_operator) # Vue globale avec zoom
plot_lie_eigenvalue_real_parts(lie_operator) # Valeurs propres triées et modules

```





```
[74]: n_grid = 30
array_x1 = np.linspace(x1_min, x1_max, n_grid, dtype=np.float32)
array_x2 = np.linspace(x2_min, x2_max, n_grid, dtype=np.float32)
matrix_grid_x1, matrix_grid_x2 = np.meshgrid(array_x1, array_x2)

array3d_dynamics = np.zeros((n_grid, n_grid, 2))

for i in range(n_grid):
    for j in range(n_grid):
        array3d_dynamics[i, j, :] = duffing(np.array([matrix_grid_x1[i, j],
↪matrix_grid_x2[i, j]]))

autoencoder.eval()
lie_operator.eval()
array3d_dynamics_pred = np.zeros((n_grid, n_grid, 2))
for i in range(n_grid):
    for j in range(n_grid):
        x1 = matrix_grid_x1[i, j]
        x2 = matrix_grid_x2[i, j]
        tensor2d_x = torch.tensor([[x1, x2]], dtype=torch.float32).to(device)
        tensor2d_observable = autoencoder.encoder(tensor2d_x)
        tensor2d_lie_observable_next = lie_operator(tensor2d_observable)
        tensor2d_predict_x_next = autoencoder.
↪decoder(tensor2d_lie_observable_next)
        array_x_next = tensor2d_predict_x_next.cpu().detach().numpy().ravel()

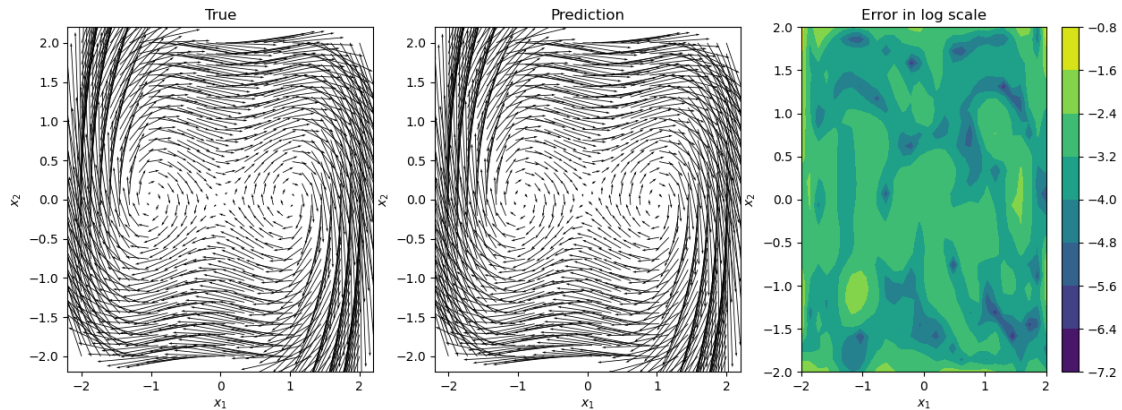
        array3d_dynamics_pred[i, j, :] = array_x_next
```

```
[75]: fig = plt.figure(figsize=(15, 5))
ax = fig.add_subplot(131)
ax.quiver(matrix_grid_x1, matrix_grid_x2, array3d_dynamics[:, :, 0],
         ↪array3d_dynamics[:, :, 1], scale=10)
ax.set_title('True')
ax.set_xlabel('$x_1$')
ax.set_ylabel('$x_2$')

ax = fig.add_subplot(132)
ax.quiver(matrix_grid_x1, matrix_grid_x2, array3d_dynamics_pred[:, :, 0],
         ↪array3d_dynamics_pred[:, :, 1], scale=10)
ax.set_title('Prediction')
ax.set_xlabel('$x_1$')
ax.set_ylabel('$x_2$')

ax = fig.add_subplot(133)
cp = ax.contourf(matrix_grid_x1, matrix_grid_x2,
                np.log(np.linalg.norm(array3d_dynamics - array3d_dynamics_pred,
                ↪axis=2)))
fig.colorbar(cp)
ax.set_title('Error in log scale')
ax.set_xlabel('$x_1$')
ax.set_ylabel('$x_2$')

plt.show()
```



Since we are now working with a **continuous-time formulation**, stability requires that the **real parts** of all eigenvalues of the Lie operator $\mathbf{L}$ be **non-positive** ($\Re(\lambda) \leq 0$). We recall that this condition ensures that solutions of the associated linear system do not exhibit exponential divergence over time.

The plots below provide a full spectral view: - In the complex plane, the eigenvalues are **almost purely imaginary**, forming what visually appears as a **vertical line**, due

to their very small real parts. - The imaginary parts range approximately from -1.5 to 1.5 , and we observe a **symmetry with respect to the real axis**, which is expected due to the operator's structure. - More importantly, the **real part of each eigenvalue is strictly negative**, ranging from about -0.003 to -0.019 .

On top of this spectral analysis, the learned model achieves **visually outstanding results** when predicting the continuous dynamics of the Duffing oscillator. The predicted vector field is almost indistinguishable from the true one across the entire domain $[-2, 2]^2$.

The **logarithmic error plot** reveals that the errors are not only minimal (down to 10^{-2} to 10^{-4} in most regions), but also **homogeneously low**, including near the borders — which were more prone to larger errors in the discrete case.

Overall, these results are **even better than in the discrete setting**, both in terms of spectral stability and dynamical prediction accuracy.

Question 7. : Compare and comment below the two approaches (discrete vs continuous operator).

- Both approaches aim to approximate nonlinear dynamics using a linear operator in a lifted space, but differ in their formulation: the discrete case learns an operator $\mathbf{K}$ that acts over finite time steps, while the continuous case relies on a Lie operator $\mathbf{L}$ that directly models the local dynamics via the vector field $f(x)$.
- In the continuous setting, full trajectories are not needed. Training only requires randomly sampled points and their derivatives, making this approach much more flexible in practice.
- The eigenvalues of $\mathbf{K}$ lie inside the unit circle, with one eigenvalue exactly equal to 1. The eigenvalues of $\mathbf{L}$ are nearly purely imaginary, symmetric, and have strictly negative real parts, which ensures exponential stability.
- The loss curves in both cases are very stable. Around epoch 40–50, all loss components settle smoothly, and no overfitting is observed.
- **Both methods yield excellent results, but the continuous model performs even better.** The predicted vector field closely matches the ground truth, and the errors are not only lower but also more uniformly distributed across the domain $[-2, 2]^2$.

– PLEASE SUBMIT YOUR COMPLETED NOTEBOOK WITH CELL OUTPUTS –