Imports

```
import os, json, math, random, argparse, time, glob, shutil
import numpy as np
import torch, torch.nn as nn, torch.nn.functional as F
from torch.utils.data import DataLoader, TensorDataset, random_split
import math, random, json
from pathlib import Path
from tqdm import tqdm
import matplotlib.pyplot as plt
import seaborn as sns
import sys
import pandas as pd
import gradio as gr
import cv2
from typing import Dict, Any, Tuple, List
import matplotlib.animation as animation
from matplotlib.colors import Normalize
from __future__ import annotations
import subprocess
```

Dataset Generation Functions

```
# ──────────────────────────────────────────────────────────────────────────────
# 2) DATA GENERATION & DATASET  (functions only – not executed yet)
# ──────────────────────────────────────────────────────────────────────────────
def render_frame(theta: float, L_px: int = 20, res: int = 51) -> np.ndarray:
    """Gaussian bob on a small canvas."""
    cx, cy = res // 2, res // 6
    bx = cx + L_px * math.sin(theta)
    by = cy + L_px * math.cos(theta)
    y, x = np.mgrid[0:res, 0:res].astype(np.float32)
    return np.exp(-((x - bx) ** 2 + (y - by) ** 2) / (L_px * 2))


# ---------------------------- integrator -------------------------------
def pendulum_theta(t: np.ndarray, theta0: float, omega0: float, g: float, L_m: float) -> np.ndarray:
    dt = t[1] - t[0]
    theta, omega = np.zeros_like(t), np.zeros_like(t)
    theta[0], omega[0] = theta0, omega0
    for i in range(1, len(t)):
        k1 = -g / L_m * math.sin(theta[i - 1]); q1 = omega[i - 1]
        k2 = -g / L_m * math.sin(theta[i - 1] + 0.5 * dt * q1)
        q2 = omega[i - 1] + 0.5 * dt * k1
        k3 = -g / L_m * math.sin(theta[i - 1] + 0.5 * dt * q2)
        q3 = omega[i - 1] + 0.5 * dt * k2
        k4 = -g / L_m * math.sin(theta[i - 1] + dt * q3)
        q4 = omega[i - 1] + dt * k3
        theta[i] = theta[i - 1] + dt/6 * (q1 + 2*q2 + 2*q3 + q4)
        omega[i] = omega[i - 1] + dt/6 * (k1 + 2*k2 + 2*k3 + k4)
    return theta


# ──────────────────────────────────────────────────────────────────────────────
# 2) DATA GENERATION & DATASET
# ──────────────────────────────────────────────────────────────────────────────
def generate_dataset(
    num_clips: int = 10**7,
    fps: float = 50,  # dt=0.02[s]
    seq_min_frames: int = 16,  # Min sequence length
    seq_max_frames: int = 32,  # Max sequence length
    res: int = 51,
    out_root: str = "./data",
    g: float = 9.81,
    L_m: float = 1.0,
    max_size_gb: int = 10):

    dt = 1.0 / fps  # dt = 0.02[s]

    out_root = Path(out_root)
    tensor_dir = out_root / "tensors"; tensor_dir.mkdir(parents=True, exist_ok=True)
    bytes_limit = max_size_gb * (1024 ** 3)
```

```
    bytes_limit = max_size_gb * (1024 ** 3)
    bytes_written = 0
    bytes_per_frame = res * res * 4
    clip_idx = 0

    shards_by_length = {}
    shard_bytes_by_length = {}

    pbar = tqdm(total=num_clips, desc="clips")
    while clip_idx < num_clips and bytes_written < bytes_limit:
        seq_length = random.randint(seq_min_frames, seq_max_frames)

        # Random initial conditions
        theta0 = random.uniform(-math.pi/2, math.pi/2)
        omega0 = random.uniform(-1.0, 1.0)

        # Energy check to ensure no full rotations
        E = g*L_m*(1-math.cos(theta0)) + 0.5*(L_m*omega0)**2
        if E >= 2*g*L_m:
            continue

        t_array = np.arange(seq_length) * dt
        theta_arr = pendulum_theta(t_array, theta0, omega0, g, L_m)

        frames = np.stack([render_frame(th, res=res) for th in theta_arr])

        if seq_length not in shards_by_length:
            shards_by_length[seq_length] = []
            shard_bytes_by_length[seq_length] = 0

        shards_by_length[seq_length].append(torch.from_numpy(frames).unsqueeze(1))

        this_bytes = seq_length * bytes_per_frame
        shard_bytes_by_length[seq_length] += this_bytes
        bytes_written += this_bytes
        clip_idx += 1
        pbar.update(1)

        if shard_bytes_by_length[seq_length] > 1e9 or bytes_written >= bytes_limit:
            length_shard = shards_by_length[seq_length]
            if length_shard:
                torch.save(
                    torch.stack(length_shard),
                    tensor_dir / f"shard_len{seq_length}_{clip_idx:07d}.pt"
                )
                shards_by_length[seq_length] = []
                shard_bytes_by_length[seq_length] = 0

    for seq_length, shard in shards_by_length.items():
        if shard:
            torch.save(
                torch.stack(shard),
                tensor_dir / f"shard_len{seq_length}_final.pt"
            )

    pbar.close()
    print(f"Saved {clip_idx} clips, {bytes_written/1e9:.2f} GB")
    print(f"Variable sequence lengths: {seq_min_frames}-{seq_max_frames} frames ({seq_min_frames/fps:.2f}-{seq_max_frames/fps:.2f} seconds at
```

Data Class & Data Loaders

```
# ─────────────────────────────────────────────────────────────────────────────
# 3) PendulumTensorDataset
# ─────────────────────────────────────────────────────────────────────────────
class PendulumTensorDataset(torch.utils.data.Dataset):
    def __init__(self, tensor_root: str):
        self.paths = sorted(Path(tensor_root).glob("shard_len*.pt"))
        self.offsets = []
        total = 0

        print(f"Found {len(self.paths)} tensor files in {tensor_root}")

        for p in self.paths:
            try:
                tensor = torch.load(p, mmap=True)
```

```
                n = tensor.shape[0]
                self.offsets.append((total, p, n))
                total += n
                print(f"Loaded {p.name} with {n} sequences")
            except Exception as e:
                print(f"Error loading {p}: {e}")

        self.N = total
        print(f"Total dataset size: {self.N} sequences")

    def __len__(self):
        return self.N

    def __getitem__(self, idx):
        for start, path, n in self.offsets:
            if idx < start + n:
                try:
                    tensor = torch.load(path, mmap=True)[idx - start]
                    return tensor
                except Exception as e:
                    print(f"Error accessing index {idx} in {path}: {e}")
                    raise
        raise IndexError(f"Index {idx} out of bounds for dataset of size {self.N}")


# ─────────────────────────────────────────────────────────────────────────────
# collate function for batching variable length sequences
# ─────────────────────────────────────────────────────────────────────────────
def variable_length_collate(batch):
    # Extract sequences and determine their lengths
    sequences = batch  # Assuming batch is a list of sequences
    seq_lengths = [seq.shape[0] for seq in sequences]  # Get sequence length from first dimension

    # Find the max sequence length in this batch
    max_len = max(seq_lengths)

    # Get other dimensions from the first sequence
    _, C, H, W = sequences[0].shape

    # Prepare padded batch tensor
    batch_size = len(sequences)
    padded_batch = torch.zeros(batch_size, max_len, C, H, W, device=sequences[0].device)

    # Fill in the actual sequences
    for i, (seq, length) in enumerate(zip(sequences, seq_lengths)):
        padded_batch[i, :length] = seq

    # Return padded batch and sequence lengths for masking in loss computation
    return (padded_batch, torch.tensor(seq_lengths, device=sequences[0].device))


# ─────────────────────────────────────────────────────────────────────────────
# 4) Updated DATA LOADERS (with variable sequence handling)
# ─────────────────────────────────────────────────────────────────────────────
def build_loaders(tensor_root: str, batch: int = 32, num_workers: int = 4):
    full = PendulumTensorDataset(tensor_root)

    if len(full) == 0:
        raise ValueError(f"No data found in {tensor_root}")

    test_size = min(20, len(full) // 10)
    val_size = int(0.1 * (len(full) - test_size))
    train_size = len(full) - val_size - test_size

    print(f"Splitting dataset: Train={train_size}, Val={val_size}, Test={test_size}")

    # Create torch Generator with fixed seed for reproducibility
    generator = torch.Generator().manual_seed(42)

    train_set, val_set, test_set = random_split(
        full, [train_size, val_size, test_size],
        generator=generator
    )

    # Use the variable_length_collate function for handling variable sequence lengths
    loader = lambda ds, shuffle: DataLoader(
        ds,
        batch_size=batch,
```

```
        shuffle=shuffle,
        num_workers=num_workers if num_workers > 0 else 0,
        pin_memory=True,
        persistent_workers=num_workers > 0,
        collate_fn=variable_length_collate  # Use custom collate function
    )

    return loader(train_set, True), loader(val_set, False), loader(test_set, False)
```

SINDy Framework

```
# ──────────────────────────────────────────────────────────────────────────────
# 4)  SINDy FRAMEWORK  (single-coordinate library Θ(z) & helpers)
# ──────────────────────────────────────────────────────────────────────────────
class PolyTrigLibrary(nn.Module):
    """Return [z, sin z, z², z³]"""
    def __init__(self):
        super().__init__(); self.out_dim = 4
    def forward(self, z):
        return torch.cat([z, torch.sin(z), z**2, z**3], dim=-1)

# -----------------------------
# 4) Bayesian helpers (SSGL prior, SGLD, EMVS)
# -----------------------------
@torch.no_grad()
def emvs_update(Xi, rho, kappa, v0=1., v1=5., delta=0.1):
    lap = torch.distributions.Laplace(0, v0)
    gau = torch.distributions.Normal(0, v1)
    p1 = gau.log_prob(Xi).exp() * delta
    p0 = lap.log_prob(Xi).exp() * (1 - delta)
    rho_new = p1 / (p1 + p0 + 1e-12)
    k0 = (1 - rho_new) / v0
    k1 = rho_new / v1
    return rho_new, (k0, k1)

def sgld_step(param, grad, lr):
    param.data.add_(-0.5 * lr * grad + torch.randn_like(param) * math.sqrt(lr))

def ss_gl_log_prob(Xi, rho, kappa):
    lap_part = (1 - rho) * torch.abs(Xi) / kappa[0]
    # Use torch.log instead of math.log for tensor operations
    log_term = torch.log(torch.tensor(2 * math.pi).to(Xi.device) * kappa[1])
    gauss_part = 0.5 * rho * (Xi**2) / kappa[1] + 0.5 * rho * log_term
    return (lap_part + gauss_part).sum()


def print_equation(Xi:torch.Tensor):
    terms = ["θ","sin θ","θ²","θ³"]
    coefs = Xi.view(-1).tolist()
    eq = "θ̈= " + " + ".join([f"{c:.3g}*{t}" for c,t in zip(coefs,terms) if abs(c)>0])
    print(eq)
```

Model

```
class ConvLSTMCell(nn.Module):
    def __init__(self, input_channels, hidden_channels, kernel_size=3):
        super().__init__()
        self.input_channels = input_channels
        self.hidden_channels = hidden_channels
        self.kernel_size = kernel_size
        padding = kernel_size // 2

        self.conv = nn.Conv2d(
            input_channels + hidden_channels,
            4 * hidden_channels,
            kernel_size,
            padding=padding
        )

    def forward(self, input_tensor, hidden_state):
        h_prev, c_prev = hidden_state
```

```
        combined = torch.cat([input_tensor, h_prev], dim=1)

        conv_output = self.conv(combined)

        cc_i, cc_f, cc_o, cc_g = torch.split(conv_output, self.hidden_channels, dim=1)

        i = torch.sigmoid(cc_i)
        f = torch.sigmoid(cc_f)
        o = torch.sigmoid(cc_o)
        g = torch.tanh(cc_g)

        c_next = f * c_prev + i * g
        h_next = o * torch.tanh(c_next)

        return h_next, c_next

# ──────────────────────────────────────────────────────────────────────────────
# 5) MODEL  (Encoder + Bayesian-SINDy + Decoder)
# ──────────────────────────────────────────────────────────────────────────────
class ConvLSTMEncoder(nn.Module):
    def __init__(self, cfg):
        super().__init__()

        # Initial feature extraction
        self.conv1 = nn.Sequential(
            nn.Conv2d(1, 16, 5, 2, 2),  # 51×51 → 26×26
            nn.ReLU()
        )

        self.conv2 = nn.Sequential(
            nn.Conv2d(16, 32, 5, 2, 2),  # 26×26 → 13×13
            nn.ReLU()
        )

        # ConvLSTM layer
        self.convlstm = ConvLSTMCell(32, 64)

        # Output projection
        feature_size = 64 * 13 * 13
        self.fc_mu  = nn.Linear(feature_size, cfg['latent_dim'])
        self.fc_log = nn.Linear(feature_size, cfg['latent_dim'])

    def forward(self, x_seq):
        # x_seq: (B, T, C, H, W)
        batch_size, seq_len = x_seq.size(0), x_seq.size(1)

        # Hidden state init
        h = torch.zeros(batch_size, 64, 13, 13, device=x_seq.device)
        c = torch.zeros(batch_size, 64, 13, 13, device=x_seq.device)

        for t in range(seq_len):
            frame = x_seq[:, t]          # (B, 1, 51, 51)
            x1 = self.conv1(frame)       # (B, 16, 26, 26)
            x2 = self.conv2(x1)          # (B, 32, 13, 13)
            h, c = self.convlstm(x2, (h, c))

        h_flat = h.reshape(batch_size, -1)
        mu   = self.fc_mu(h_flat)
        logv = self.fc_log(h_flat)
        z    = mu + torch.randn_like(mu) * torch.exp(0.5 * logv)  # reparam

        return z, mu, logv


# -----------------------------
# 3) Bayesian SINDy-RNN cell
# -----------------------------
class BayesianSINDyCell(nn.Module):
    """
    z_{t+1} = z_t + h · Φ(z_t) · Ξ           (Euler micro-steps)

    Ξ has an independent Gaussian posterior
        q(Ξ_ij) = N(μ_ij, σ_ij²)
    and we keep the closed-form KL term so the training loop can add it.

    Parameters
```

```
    ----------
    library   : nn.Module   - maps z → Φ(z)  (out_dim)
    dt        : float       - macro time-step
    k_micro   : int         - # Euler sub-steps per dt
    prior_std : float       - σ₀ of the N(0, σ₀²) prior over Ξ
    """
    def __init__(self,
                 library: nn.Module,
                 dt: float,
                 k_micro: int = 1,
                 prior_std: float = 1e-1):
        super().__init__()
        assert library.in_dim == 1, "please fucking work I've been on this shit for 5 hours"
        self.lib = library
        self.dt  = dt
        self.k   = k_micro

        # variational parameters of Ξ  (μ, log σ)
        self.mu         = nn.Parameter(1e-4 * torch.randn(library.out_dim, 1))
        self.log_sigma  = nn.Parameter(torch.full((library.out_dim, 1), -3.0))

        self.register_buffer("prior_var", torch.tensor(prior_std ** 2))
        self.kl = torch.tensor(0.)

    # ----------------------------------------------------------------
    def _sample_Xi(self) -> torch.Tensor:
        eps = torch.randn_like(self.mu)
        return self.mu + torch.exp(self.log_sigma) * eps

    # ----------------------------------------------------------------
    def forward(self, z: torch.Tensor) -> torch.Tensor:
        """
        Parameters
        ----------
        z : (B, 1)

        Returns
        -------
        z_next : (B, 1)
        """
        Xi = self._sample_Xi()          # (out_dim, 1)
        h  = self.dt / self.k
        for _ in range(self.k):
            Phi = self.lib(z)           # (B, out_dim)
            z   = z + h * (Phi @ Xi)    # (B, 1)

        # analytic KL[q(Ξ)||p(Ξ)]
        var_q = torch.exp(2 * self.log_sigma)
        self.kl = 0.5 * torch.sum(
            (self.mu ** 2 + var_q) / self.prior_var - 1 +
            torch.log(self.prior_var) - 2 * self.log_sigma
        )
        return z

# ───────────────────────────────────────────────────────────────
# DeconvDecoder
# ───────────────────────────────────────────────────────────────
class DeconvDecoder(nn.Module):
    def __init__(self):
        super().__init__()

        self.fc = nn.Sequential(
            nn.Linear(1, 64 * 7 * 7),
            nn.ReLU()
        )

        self.deconv1 = nn.Sequential(
            nn.ConvTranspose2d(64, 32, 4, 2, 1),  # 7×7 → 14×14
            nn.ReLU(),
            nn.BatchNorm2d(32)
        )
        self.deconv2 = nn.Sequential(
            nn.ConvTranspose2d(32, 16, 4, 2, 1),  # 14×14 → 28×28
            nn.ReLU(),
            nn.BatchNorm2d(16)
        )
```

```
        self.deconv3 = nn.Sequential(
            nn.ConvTranspose2d(16, 8, 4, 2, 1),   # 28×28 → 56×56
            nn.ReLU(),
            nn.BatchNorm2d(8)
        )
        self.final = nn.Sequential(
            nn.Conv2d(8, 1, 3, 1, 1),
            nn.Sigmoid()
        )

    def forward(self, z):
        batch_size = z.size(0)
        x = self.fc(z).view(batch_size, 64, 7, 7)
        x = self.deconv1(x)
        x = self.deconv2(x)
        x = self.deconv3(x)
        x = self.final(x)

        if x.shape[-1] != 51:
            x = F.interpolate(x, size=(51, 51), mode='bilinear', align_corners=False)
        return x

# ─────────────────────────────────────────────────────────────────────────────
# 5)  MODEL  – FullModel  (enc + latent + decoder)
# ─────────────────────────────────────────────────────────────────────────────
class BayesianSINDyRNN(nn.Module):
    """Encoder → Bayesian-SINDy latent RNN → Decoder"""

    def __init__(self, cfg: Dict[str, Any]):
        super().__init__()
        self.cfg = cfg

        self.enc = ConvLSTMEncoder(cfg)

        lib = PolyTrigLibrary(cfg['latent_dim'])

        self.cell = BayesianSINDyCell(
            library     = lib,
            dt          = cfg['dt'],
            k_micro     = cfg['k_micro'],
            latent_dim  = cfg['latent_dim'],
            prior_std   = cfg.get('prior_std', 1e-1)
        )

        self.dec = DeconvDecoder()

        # KL weight from cfg
        self.beta_kl = cfg.get('beta_kl', 1.0)

    # ---------------------------------------------------------
    def forward(self, x_seq: torch.Tensor) -> Dict[str, torch.Tensor]:
        z0, mu_e, logvar_e = self.enc(x_seq)   # (B, latent_dim)
        z_pred = self.cell(z0)                 # (B, latent_dim)
        x_rec  = self.dec(z_pred)              # (B, 1, H, W)

        return {
            'z0':       z0,
            'z_pred':   z_pred,
            'x_rec':    x_rec,
            'kl_lat':   self.cell.kl,
            'mu_e':     mu_e,
            'logvar_e': logvar_e
        }
```

Loss Functions

```
# ────────────────────────────────────────────────────────────
# compute_loss  (second-order)
# ────────────────────────────────────────────────────────────
def compute_loss(batch_data, model, rho, kappa, cfg):
    batch, seq_lengths = batch_data
    B = batch.shape[0]
```

```
    losses = {'rec': 0, 'pred': 0, 'acc': 0, 'lat': 0, 'kld': 0}
    valid_sequences = 0

    for i in range(B):
        seq_len = seq_lengths[i].item()
        if seq_len < 3:
            continue

        sequence = batch[i, :seq_len].unsqueeze(0)
        out = model(sequence)

        z_t, mu, lv      = out['z'], out['mu'], out['logvar']
        z_tm1, _, _      = model.enc(sequence[:, :-1].contiguous())
        z_pred, x_rec    = out['z_pred'], out['x_rec']

        L_rec  = F.mse_loss(sequence[:, -1], x_rec)
        z_gt_tp1 = z_t + (z_t - z_tm1)
        L_pred = F.mse_loss(z_pred, z_gt_tp1.detach())

        dt2 = cfg['dt'] ** 2
        z_ddot_fd  = (z_pred - 2 * z_t + z_tm1) / dt2
        z_ddot_est = model.cell.lib(z_t) @ model.cell.Xi
        L_acc  = F.mse_loss(z_ddot_est, z_ddot_fd.detach())
        L_lat  = F.mse_loss(z_pred, z_t.detach())
        KLD    = -0.5 * torch.mean(1 + lv - mu ** 2 - lv.exp())

        losses['rec']  += L_rec.item()
        losses['pred'] += L_pred.item()
        losses['acc']  += L_acc.item()
        losses['lat']  += L_lat.item()
        losses['kld']  += KLD.item()
        valid_sequences += 1

    if valid_sequences == 0:
        return torch.tensor(0., requires_grad=True, device=batch.device), {k: 0. for k in losses}

    for k in losses:
        losses[k] /= valid_sequences

    L_prior = ss_gl_log_prob(model.cell.Xi, rho, kappa)
    loss_tensor = (
        L_rec +
        L_pred +
        0.1 * L_acc +
        cfg['lambda_lat'] * L_lat +
        1e-4 * KLD +
        cfg['lambda_prior'] * L_prior
    )
    metrics = {**losses, 'prior': L_prior.item()}
    return loss_tensor, metrics
```

Training Loop

```
def random_coef():
    magnitude = 10 ** random.uniform(-1, 1)  # Random magnitude between 10^-1 and 10
    sign = random.choice([-1, 1])            # Random sign
    return sign * magnitude


# ──────────────────────────────────────────────────────────────
# training.py  –  Adam (encoder/decoder)  +  Cyclic-SGLD (μ, log σ)
#                 for the Bayesian-SINDy latent model
# ──────────────────────────────────────────────────────────────
#
#   * Encoder & Decoder are trained with Adam.
#   * The variational SINDy parameters   μ , log σ
#     are updated with a cyclical Stochastic-Gradient-Langevin-Dynamics
#     step whose learning-rate oscillates between lr_xi_lo ↔ lr_xi_hi
#     every `cycle_steps` iterations.
#   * We still keep a *hard-pruning* mask on μ
#     so coefficients that shrink below a tolerance stay permanently zero.
#
#   Dependencies:  compute_loss, ss_gl_log_prob, print_equation,
#                  random_coef
```

```python
# ──────────────────────────────────────────────────────────────
# ---------------------------------------------------------------------
#  train()
# ---------------------------------------------------------------------
def train(model,
          loaders: Tuple[torch.utils.data.DataLoader,
                         torch.utils.data.DataLoader,
                         torch.utils.data.DataLoader],
          cfg: Dict[str, Any],
          epochs: int = 1500,
          save_root: str = "/content/drive/MyDrive/ProjectFinalBSRVAE10"):
    """
    End-to-end training routine for ConvLSTM-Encoder → Bayesian-SINDy-RNN → Decoder.

    Parameters
    ----------
    model      : nn.Module
        Complete network (enc · latent · dec).
    loaders    : (train_loader, val_loader, test_loader)
        Each loader returns  (videos, seq_lengths).
    cfg        : dict
        Must contain: device, dt, lambda_lat, lambda_prior, lr_encdec,
        lr_xi_hi, lr_xi_lo, cycle_steps  (see CFG example in notebook).
    """
    train_loader, val_loader, _ = loaders
    device = cfg["device"]
    model.to(device)

    # ──────────────────────────────────────────────────────────
    # 1) Hard-pruning mask over μ
    # ──────────────────────────────────────────────────────────
    model.pruned_mask = torch.zeros_like(model.cell.mu,
                                         dtype=torch.bool, device=device)

    # ──────────────────────────────────────────────────────────
    # 2) Initialise μ with small random SINDy coefficients
    #      (log σ already initialised by the cell)
    # ──────────────────────────────────────────────────────────
    with torch.no_grad():
        model.cell.mu[:] = torch.tensor([
            [random_coef()],        # θ
            [random_coef()],        # sin θ
            [random_coef()],        # θ²
            [random_coef()]         # θ³
        ], device=device, dtype=torch.float32)
    print("μ (initial coefficients):",
          model.cell.mu.data.squeeze().tolist())

    # ──────────────────────────────────────────────────────────
    # 3) Optimisers
    #    • Adam  – encoder & decoder
    #    • dummy SGD  – provides lr to CyclicLR for  μ, log σ
    # ──────────────────────────────────────────────────────────
    encdec_params = list(model.enc.parameters()) + list(model.dec.parameters())
    xi_params     = [model.cell.mu, model.cell.log_sigma]

    opt_encdec = torch.optim.Adam(encdec_params, lr=cfg["lr_encdec"])
    opt_xi     = torch.optim.SGD(xi_params, lr=cfg["lr_xi_hi"])  # dummy

    xi_scheduler = torch.optim.lr_scheduler.CyclicLR(
        opt_xi,
        base_lr=cfg["lr_xi_lo"],
        max_lr=cfg["lr_xi_hi"],
        step_size_up=cfg["cycle_steps"],
        mode="triangular2",
        cycle_momentum=False
    )

    # SS-GL prior hyper-parameters (same shapes as μ)
    rho   = torch.zeros_like(model.cell.mu, device=device)
    kappa = (torch.ones_like(model.cell.mu, device=device),
             torch.ones_like(model.cell.mu, device=device) * 5.0)

    # bookkeeping
    history  = {"epoch": [], "train": [], "val": []}
    ckpt_dir = Path(save_root) / "checkpoints"
    ckpt_dir.mkdir(parents=True, exist_ok=True)
```

```
# ────────────────────────────────────────────────────────────
# 4) Main epoch loop
# ────────────────────────────────────────────────────────────
try:
    for ep in range(1, epochs + 1):

        # ========== TRAINING ==========
        model.train()
        train_metrics = {k: 0. for k in
                         ["rec", "pred", "acc", "lat", "kld",
                          "prior", "total"]}
        n_batches = 0

        for batch_data in train_loader:
            # push tensors to GPU
            batch_data = tuple(t.to(device) if isinstance(t, torch.Tensor)
                               else t for t in batch_data)

            # forward / loss
            loss, mets = compute_loss(batch_data, model, rho, kappa, cfg)
            mets["prior"] = (ss_gl_log_prob(model.cell.mu, rho, kappa)
                             .item() * cfg["lambda_prior"])
            mets["total"] = loss.item()

            # ---- zero grads ----
            opt_encdec.zero_grad()
            for p in xi_params:
                p.grad = None

            # ---- backward ----
            loss.backward()

            # ---- Adam step (enc/dec) ----
            opt_encdec.step()

            # ---- Cyclic-SGLD step (μ, log σ) ----
            lr_xi = xi_scheduler.get_last_lr()[0] # current LR
            for p in xi_params:
                if p.grad is None:
                    continue
                noise = torch.randn_like(p)
                p.data.add_( -lr_xi * p.grad
                             + math.sqrt(2.0 * lr_xi) * noise )
                p.grad.zero_()
            xi_scheduler.step()

            # ---- hard-prune small μ entries ----
            with torch.no_grad():
                mask = model.cell.mu.abs() <= 1e-4
                model.pruned_mask |= mask
                model.cell.mu[model.pruned_mask] = 0.0
                # (no need to mask gradients niggas are zeroed)

            # accumulate stats
            for k, v in mets.items():
                train_metrics[k] += v
            n_batches += 1

        if n_batches:
            train_metrics = {k: v / n_batches for k, v in train_metrics.items()}

        # ========== VALIDATION ==========
        model.eval()
        val_metrics = {k: 0. for k in train_metrics}
        n_val = 0
        with torch.no_grad():
            for batch_data in val_loader:
                batch_data = tuple(t.to(device) if isinstance(t, torch.Tensor)
                                   else t for t in batch_data)
                loss, mets = compute_loss(batch_data, model, rho, kappa, cfg)
                mets["prior"] = (ss_gl_log_prob(model.cell.mu, rho, kappa)
                                 .item() * cfg["lambda_prior"])
                mets["total"] = loss.item()
                for k, v in mets.items():
                    val_metrics[k] += v
                n_val += 1
```

```
            if n_val:
                val_metrics = {k: v / n_val for k, v in val_metrics.items()}

            # ========== BOOKKEEPING ==========
            history["epoch"].append(ep)
            history["train"].append(train_metrics)
            history["val"].append(val_metrics)

            print(f"Epoch {ep:03d} | "
                  f"REC {train_metrics['rec']:.3e}/{val_metrics['rec']:.3e} | "
                  f"PRED {train_metrics['pred']:.3e}/{val_metrics['pred']:.3e} | "
                  f"ACC {train_metrics['acc']:.3e}/{val_metrics['acc']:.3e}")

            if ep % 10 == 0:
                print(f"\n--- mean SINDy equation at epoch {ep} ---")
                print_equation(model.cell.mu)          # human-readable form

            torch.save({
                "epoch": ep,
                "model_state": model.state_dict(),
                "mu": model.cell.mu.detach().cpu(),
                "log_sigma": model.cell.log_sigma.detach().cpu(),
                "pruned_mask": model.pruned_mask.detach().cpu(),
                "history": history,
            }, ckpt_dir / f"epoch_{ep:04d}.pt")

    except KeyboardInterrupt:
        print("Training interrupted by user.")

    finally:
        # final checkpoint & history
        torch.save({
            "epoch": ep if "ep" in locals() else 0,
            "model_state": model.state_dict(),
            "mu": model.cell.mu.detach().cpu(),
            "log_sigma": model.cell.log_sigma.detach().cpu(),
            "pruned_mask": model.pruned_mask.detach().cpu(),
            "history": history,
        }, ckpt_dir / "final.pt")

        with open(Path(save_root) / "training_history.json", "w") as f:
            json.dump(history, f, indent=2)

        # explicit dataloader clean-up (multiprocessing)
        del train_loader, val_loader, _
        return history
```

Hyperparameters

```
# ─────────────────────────────────────────────────────────────────────────────
# 1) Hyper-parameters
# ─────────────────────────────────────────────────────────────────────────────
CFG = {
        "latent_dim":    1,                         # dimensionality of θ
        "dt":            0.02,                      # time step (s)
        "k_micro":       16,                        # Euler micro-steps per frame
        "lambda_lat":    1e-2,                      # weight on latent-alignment loss
        "lambda_prior":  1e-4,                      # weight on SSGL prior (−log p(Ξ))
        "lr_encdec":     1e-3,                      # Adam LR for encoder & decoder
        "lr_xi_hi":      1e-3,                      # maximum LR for cyclical SGLD on Ξ
        "lr_xi_lo":      1e-5,                      # minimum LR for cyclical SGLD on Ξ
        "cycle_steps":   500,                       # length of one cosine cycle for SGLD
        "rho_thresh":    0.05,                      # hard-prune threshold on inclusion prob ρ
        "prior_std":     1e-1,                      # σ₀ in the N(0,σ₀²) prior over Ξ
        "beta_kl":       1.0,                       # weight on KL_lat term  (β-VAE style)
        "device":        "cuda" if torch.cuda.is_available() else "cpu",
        "use_encoder_kl": True,
        "beta_enc":      1.0,
}
```

Generating Dataset

```
OUT_ROOT = "/content/drive/MyDrive/ProjectFinalBSRVAE10"
TENSOR_DIR = Path(OUT_ROOT) / "tensors"


generate_dataset(
    num_clips = 5_000,
    fps = 50,              # 50 Hz (dt = 0.02s)
    seq_min_frames = 16,   # Minimum sequence length (16 frames = 0.32s)
    seq_max_frames = 32,   # Maximum sequence length (32 frames = 0.64s)
    res = 51,
    out_root = OUT_ROOT,
    g = 9.81,
    L_m = 1.0,
    max_size_gb = 10
)
```

Data Visualization

```
def visualize_pendulum_sequence(tensor_dir, sequence_idx=0, output_path="pendulum_animation.mp4"):
    # Verify tensor directory exists
    tensor_dir = Path(tensor_dir)
    if not tensor_dir.exists():
        raise FileNotFoundError(f"Tensor directory not found: {tensor_dir}")

    # List all tensor files
    tensor_files = sorted(list(tensor_dir.glob("shard_len*.pt")))
    if not tensor_files:
        raise FileNotFoundError(f"No tensor files found in {tensor_dir}")

    print(f"Found {len(tensor_files)} tensor files.")

    # Load a sequence directly from file
    file_idx = 0
    try:
        tensor_data = torch.load(tensor_files[file_idx])
        if sequence_idx >= tensor_data.shape[0]:
            sequence_idx = 0
            print(f"Requested index too large. Using index 0 instead.")
        sequence = tensor_data[sequence_idx]
    except Exception as e:
        print(f"Error loading tensor: {e}")
        # Try loading a different file if available
        if len(tensor_files) > 1:
            print("Trying next file...")
            tensor_data = torch.load(tensor_files[1])
            sequence = tensor_data[0]
        else:
            raise RuntimeError("Failed to load any valid tensor data.")

    print(f"Sequence shape: {sequence.shape}")

    # Create figure and axis
    fig = plt.figure(figsize=(6, 6))
    ax = plt.subplot(111)

    # Initial frame
    frame_data = sequence[0, 0].numpy()  # First frame, first channel
    norm = Normalize(vmin=0, vmax=frame_data.max())
    img = ax.imshow(frame_data, cmap='viridis', norm=norm, animated=True)
    plt.axis('off')
    plt.tight_layout()

    # Animation function
    def update(frame):
        if frame < sequence.shape[0]:
            frame_data = sequence[frame, 0].numpy()
            img.set_array(frame_data)
        return [img]

    # Create animation
    ani = animation.FuncAnimation(
        fig, update, frames=sequence.shape[0],
        interval=40,  # 40ms between frames (~25 fps)
```

```
        blit=True
    )

    # Save as MP4
    try:
        writer = animation.FFMpegWriter(fps=25, metadata=dict(artist='Me'), bitrate=1800)
        ani.save(output_path, writer=writer)
        print(f"Animation saved to {output_path}")
    except Exception as e:
        print(f"Error saving animation: {e}")
        print("Trying to save as GIF instead...")
        try:
            ani.save(output_path.replace('.mp4', '.gif'), writer='pillow')
            print(f"Animation saved as GIF")
        except Exception as e2:
            print(f"Error saving GIF: {e2}")

    plt.close()

    # If in a notebook environment, also display the animation
    try:
        from IPython.display import HTML, display
        if os.path.exists(output_path):
            display(HTML(f'<video width="500" height="500" controls><source src="{output_path}" type="video/mp4"></video>'))
    except ImportError:
        pass


OUT_ROOT = "/content/drive/MyDrive/ProjectFinalBSRVAE10"
TENSOR_DIR = Path(OUT_ROOT) / "tensors"

visualize_pendulum_sequence(
    tensor_dir=TENSOR_DIR,
    sequence_idx=0,
    output_path="pendulum_example.mp4"
)
```

Data Loaders & Start of Training

```
OUT_ROOT = Path("/content/drive/MyDrive/ProjectFinalBSRVAE10")
TENSOR_DIR = OUT_ROOT / "tensors"

train_loader, val_loader, test_loader = build_loaders(
    tensor_root=TENSOR_DIR,
    batch=32,
    num_workers=9
)
```

```
model = BayesianSINDyRNN(CFG)
history = train(
    model,
    (train_loader, val_loader, test_loader),
    CFG,
    epochs = 1500,
    save_root = OUT_ROOT
)
```

Checkpoints & Evaluation Functions

```
%run '/content/drive/MyDrive/ProjectFinalBSRVAE10/model_functions.py'
```

```
def load_checkpoint(checkpoint_path, model, device):
    """Load model checkpoint and return the epoch number and training history."""
    checkpoint = torch.load(checkpoint_path, map_location=device)
    model.load_state_dict(checkpoint['model_state'])

    if 'Xi' in checkpoint:
        model.cell.Xi.data = checkpoint['Xi'].to(device)

    if 'pruned_mask' in checkpoint:
```

```
        model.pruned_mask = checkpoint['pruned_mask'].to(device)
    else:
        model.pruned_mask = torch.zeros_like(model.cell.Xi, dtype=torch.bool, device=device)

    history = checkpoint.get('history', {'epoch': [], 'train': [], 'val': []})
    start_epoch = checkpoint.get('epoch', 0) + 1

    return start_epoch, history


def evaluate_model(model, test_loader, cfg, checkpoint_path=None, output_dir="./evaluation_results"):
    device = cfg['device']
    model.to(device)
    model.eval()

    if checkpoint_path is not None:
        checkpoint = torch.load(checkpoint_path, map_location=device)
        model.load_state_dict(checkpoint['model_state'])
        print(f"Loaded model from {checkpoint_path}")

    output_path = Path(output_dir)
    output_path.mkdir(parents=True, exist_ok=True)

    mse_reconstruction = []
    mse_prediction = []
    latent_trajectories = []
    ground_truth_frames = []
    reconstructed_frames = []
    predicted_frames = []

    # Extract the SINDy coefficients
    sindy_coefficients = model.cell.Xi.detach().cpu().numpy()

    with torch.no_grad():
        for batch_idx, batch_data in enumerate(tqdm(test_loader, desc="Evaluating")):
            batch, seq_lengths = tuple(t.to(device) if isinstance(t, torch.Tensor) else t for t in batch_data)

            for i in range(batch.shape[0]):
                seq_len = seq_lengths[i].item()
                if seq_len < 3:
                    continue

                sequence = batch[i, :seq_len].unsqueeze(0)  # Add batch dimension back

                # Split into input and target for prediction evaluation
                input_seq = sequence[:, :-1]
                target_frame = sequence[:, -1]

                # Forward pass
                out = model(input_seq)
                z_t = out['z']  # Current latent state

                # Get previous latent state
                if input_seq.shape[1] > 1:
                    _, z_tm1, _ = model.enc(input_seq[:, :-1])
                else:
                    # If only one frame, use a zero vector as previous state
                    z_tm1 = torch.zeros_like(z_t)

                # Predict next latent state
                z_pred = model.cell(z_t)

                # Decode the predicted latent state
                x_pred = model.dec(z_pred)

                # Reconstruction from current latent state
                x_rec = model.dec(z_t)

                # Calculate metrics
                rec_mse = nn.MSELoss()(x_rec, target_frame).item()
                pred_mse = nn.MSELoss()(x_pred, target_frame).item()

                mse_reconstruction.append(rec_mse)
                mse_prediction.append(pred_mse)

                # Store the first few sequences for visualization
                if batch_idx < 5:
```

```
                # Store latent trajectory
                latent_trajectories.append({
                    'current': z_t.squeeze().cpu().numpy(),
                    'predicted': z_pred.squeeze().cpu().numpy()
                })

                # Store frames for visualization
                ground_truth_frames.append(target_frame.squeeze().cpu().numpy())
                reconstructed_frames.append(x_rec.squeeze().cpu().numpy())
                predicted_frames.append(x_pred.squeeze().cpu().numpy())

        # Calculate average metrics
        avg_rec_mse = np.mean(mse_reconstruction)
        avg_pred_mse = np.mean(mse_prediction)

        print(f"Average Reconstruction MSE: {avg_rec_mse:.6f}")
        print(f"Average Prediction MSE: {avg_pred_mse:.6f}")

        # Create visualizations

        # 1. Visualization of SINDy coefficients
        plt.figure(figsize=(10, 6))
        terms = ["θ", "sin θ", "θ²", "θ³"]

        # Plot coefficients as a bar chart
        plt.bar(terms, sindy_coefficients.squeeze(), color='blue', alpha=0.7)
        plt.axhline(y=0, color='k', linestyle='-', alpha=0.3)
        plt.title("Discovered SINDy Coefficients", fontsize=14)
        plt.ylabel("Coefficient Value", fontsize=12)
        plt.grid(axis='y', alpha=0.3)

        # Add true equation for comparison
        plt.figtext(0.5, 0.01, "True equation: θ̈= -9.81*sin(θ)", ha="center", fontsize=12,
                    bbox={"facecolor":"orange", "alpha":0.2, "pad":5})

        plt.tight_layout()
        plt.savefig(output_path / "sindy_coefficients.png", dpi=300)

        # 2. Compare frame reconstruction and prediction
        if ground_truth_frames:
            n_samples = min(5, len(ground_truth_frames))
            fig, axes = plt.subplots(n_samples, 3, figsize=(15, 3*n_samples))

            for i in range(n_samples):
                # Ground truth
                im0 = axes[i, 0].imshow(ground_truth_frames[i], cmap='viridis')
                axes[i, 0].set_title("Ground Truth" if i == 0 else "")
                axes[i, 0].axis('off')

                # Reconstruction
                im1 = axes[i, 1].imshow(reconstructed_frames[i], cmap='viridis')
                axes[i, 1].set_title("Reconstruction" if i == 0 else "")
                axes[i, 1].axis('off')

                # Prediction
                im2 = axes[i, 2].imshow(predicted_frames[i], cmap='viridis')
                axes[i, 2].set_title("Prediction" if i == 0 else "")
                axes[i, 2].axis('off')

            plt.tight_layout()
            plt.savefig(output_path / "frame_comparison.png", dpi=300)

        # 3. Latent space visualization
        if latent_trajectories:
            plt.figure(figsize=(10, 6))

            for i, traj in enumerate(latent_trajectories[:5]):
                plt.scatter(i, traj['current'], color='blue', label='Current' if i == 0 else "")
                plt.scatter(i+0.5, traj['predicted'], color='red', label='Predicted' if i == 0 else "")
                plt.plot([i, i+0.5], [traj['current'], traj['predicted']], 'k--', alpha=0.5)

            plt.xlabel("Sequence Index")
            plt.ylabel("Latent Value")
            plt.title("Latent Space Trajectories")
            plt.legend()
            plt.grid(alpha=0.3)
            plt.tight_layout()
```

```
        plt.savefig(output_path / "latent_trajectories.png", dpi=300)

    # 4. Error distribution
    plt.figure(figsize=(10, 6))

    plt.subplot(1, 2, 1)
    sns.histplot(mse_reconstruction, kde=True, color='blue')
    plt.title("Reconstruction MSE")
    plt.xlabel("MSE")
    plt.ylabel("Frequency")

    plt.subplot(1, 2, 2)
    sns.histplot(mse_prediction, kde=True, color='red')
    plt.title("Prediction MSE")
    plt.xlabel("MSE")
    plt.ylabel("Frequency")

    plt.tight_layout()
    plt.savefig(output_path / "error_distribution.png", dpi=300)

    # 5. Generate pendulum equation comparison
    fig, ax = plt.subplots(figsize=(12, 6))

    # Define the actual pendulum equation: θ̈= -g/L * sin(θ)
    g = 9.81  # gravity
    L = 1.0   # length (as in your simulation)

    # Create theta values for plotting
    theta = np.linspace(-np.pi, np.pi, 1000)

    # True acceleration
    true_accel = -g/L * np.sin(theta)

    # Model predicted acceleration
    # Extract coefficients for readability
    coef_theta = sindy_coefficients[0, 0]
    coef_sin = sindy_coefficients[1, 0]
    coef_theta2 = sindy_coefficients[2, 0]
    coef_theta3 = sindy_coefficients[3, 0]

    predicted_accel = (coef_theta * theta +
                       coef_sin * np.sin(theta) +
                       coef_theta2 * theta**2 +
                       coef_theta3 * theta**3)

    # Plot
    ax.plot(theta, true_accel, 'b-', linewidth=2, label='True: $\\ddot{\\theta} = -9.81 \\sin(\\theta)$')
    ax.plot(theta, predicted_accel, 'r--', linewidth=2,
            label=f'Discovered: $\\ddot{{\\theta}} = {coef_theta:.3f}\\theta + {coef_sin:.3f}\\sin(\\theta) + {coef_theta2:.3f}\\theta^2 + {

    ax.set_xlabel(r'$\theta$ (radians)', fontsize=14)
    ax.set_ylabel(r'$\ddot{\theta}$ (rad/s²)', fontsize=14)
    ax.set_title('Comparison of True vs Discovered Pendulum Dynamics', fontsize=16)
    ax.grid(True, alpha=0.3)
    ax.legend(fontsize=12)

    plt.tight_layout()
    plt.savefig(output_path / "equation_comparison.png", dpi=300)

    # Return metrics for further analysis if needed
    return {
        'avg_reconstruction_mse': avg_rec_mse,
        'avg_prediction_mse': avg_pred_mse,
        'sindy_coefficients': sindy_coefficients
    }

# Function to generate phase space visualization
def visualize_phase_space(model, output_dir="./evaluation_results"):
    """
    Visualize the pendulum phase space using the discovered equation
    """
    output_path = Path(output_dir)
    output_path.mkdir(parents=True, exist_ok=True)

    device = next(model.parameters()).device

    # Extract SINDy coefficients
```

```
    with torch.no_grad():
        Xi = model.cell.Xi.detach().cpu().numpy().squeeze()

    # Define the state space grid
    theta = np.linspace(-np.pi, np.pi, 20)
    omega = np.linspace(-5, 5, 20)

    THETA, OMEGA = np.meshgrid(theta, omega)

    # Compute vector field using discovered coefficients
    # dθ/dt = ω
    # dω/dt = Xi[0]*θ + Xi[1]*sin(θ) + Xi[2]*θ² + Xi[3]*θ³

    dTHETA = OMEGA

    # Apply the SINDy equation to get dω/dt
    dOMEGA = (Xi[0] * THETA +
              Xi[1] * np.sin(THETA) +
              Xi[2] * THETA**2 +
              Xi[3] * THETA**3)

    # Calculate vector magnitudes for color mapping
    magnitude = np.sqrt(dTHETA**2 + dOMEGA**2)

    # Normalize the vectors for better visualization
    norm = np.sqrt(dTHETA**2 + dOMEGA**2)
    norm[norm == 0] = 1  # Prevent division by zero
    dTHETA_norm = dTHETA / norm
    dOMEGA_norm = dOMEGA / norm

    # Create the phase portrait
    plt.figure(figsize=(10, 8))
    plt.quiver(THETA, OMEGA, dTHETA_norm, dOMEGA_norm, magnitude, cmap='viridis',
               pivot='mid', alpha=0.8)

    # Add streamlines for clearer flow visualization
    plt.streamplot(THETA, OMEGA, dTHETA, dOMEGA, color='white', linewidth=0.5, density=1.5, arrowsize=0.5)

    # Add colorbar
    cbar = plt.colorbar()
    cbar.set_label('Vector magnitude', rotation=270, labelpad=15)

    # Add grid
    plt.grid(alpha=0.3)

    # Set labels and title
    plt.xlabel(r'$\theta$ (radians)', fontsize=12)
    plt.ylabel(r'$\omega$ (rad/s)', fontsize=12)
    plt.title('Phase Space Portrait using Discovered Dynamics', fontsize=14)

    # Add the discovered equation as text
    equation_text = f"Discovered equation: $\\ddot{{\\theta}} = {Xi[0]:.3f}\\theta + {Xi[1]:.3f}\\sin(\\theta) + {Xi[2]:.3f}\\theta^2 + {Xi[
    plt.figtext(0.5, 0.01, equation_text, ha="center", fontsize=12,
                bbox={"facecolor":"white", "alpha":0.8, "pad":5})

    plt.tight_layout()
    plt.savefig(output_path / "phase_space.png", dpi=300)

    # Create energy level contours
    plt.figure(figsize=(10, 8))

    # Simplified pendulum energy function: E = 0.5*ω² + g/L*(1-cos(θ))
    g = 9.81
    L = 1.0
    E = 0.5 * OMEGA**2 + g/L * (1 - np.cos(THETA))

    # Plot contour lines of constant energy
    plt.contourf(THETA, OMEGA, E, levels=20, cmap='coolwarm', alpha=0.7)
    cbar = plt.colorbar()
    cbar.set_label('Energy', rotation=270, labelpad=15)

    # Overlay the vector field
    plt.quiver(THETA, OMEGA, dTHETA_norm, dOMEGA_norm, alpha=0.5, color='k')

    # Set labels and title
    plt.xlabel(r'$\theta$ (radians)', fontsize=12)
    plt.ylabel(r'$\omega$ (rad/s)', fontsize=12)
```

```
        plt.title('Energy Levels and Vector Field', fontsize=14)
        plt.grid(alpha=0.3)

        plt.tight_layout()
        plt.savefig(output_path / "energy_levels.png", dpi=300)

        # Return the coefficients for reference
        return Xi

    # Simulation function to generate pendulum trajectories using the learned model
    def simulate_pendulum_trajectories(model, initial_conditions, n_steps=50, dt=0.02, output_dir="./evaluation_results"):
        """
        Simulate pendulum trajectories using the discovered dynamics

        Args:
            model: Trained SINDy-RNN model
            initial_conditions: List of (theta0, omega0) tuples
            n_steps: Number of simulation steps
            dt: Time step size
            output_dir: Directory to save results
        """
        output_path = Path(output_dir)
        output_path.mkdir(parents=True, exist_ok=True)

        device = next(model.parameters()).device

        # Extract SINDy coefficients
        with torch.no_grad():
            Xi = model.cell.Xi.detach().cpu().numpy().squeeze()

        # True pendulum parameters
        g = 9.81
        L = 1.0

        # Prepare figure
        plt.figure(figsize=(12, 10))

        # For each initial condition
        for i, (theta0, omega0) in enumerate(initial_conditions):
            # Initialize arrays for true and predicted trajectories
            true_theta = np.zeros(n_steps)
            true_omega = np.zeros(n_steps)
            pred_theta = np.zeros(n_steps)
            pred_omega = np.zeros(n_steps)

            # Set initial conditions
            true_theta[0] = theta0
            true_omega[0] = omega0
            pred_theta[0] = theta0
            pred_omega[0] = omega0

            # Simulate using RK4
            for t in range(1, n_steps):
                # True dynamics using RK4
                k1_theta = true_omega[t-1]
                k1_omega = -g/L * np.sin(true_theta[t-1])

                k2_theta = true_omega[t-1] + 0.5 * dt * k1_omega
                k2_omega = -g/L * np.sin(true_theta[t-1] + 0.5 * dt * k1_theta)

                k3_theta = true_omega[t-1] + 0.5 * dt * k2_omega
                k3_omega = -g/L * np.sin(true_theta[t-1] + 0.5 * dt * k2_theta)

                k4_theta = true_omega[t-1] + dt * k3_omega
                k4_omega = -g/L * np.sin(true_theta[t-1] + dt * k3_theta)

                true_theta[t] = true_theta[t-1] + dt/6 * (k1_theta + 2*k2_theta + 2*k3_theta + k4_theta)
                true_omega[t] = true_omega[t-1] + dt/6 * (k1_omega + 2*k2_omega + 2*k3_omega + k4_omega)

                # Discovered dynamics using RK4
                k1_theta = pred_omega[t-1]
                k1_omega = Xi[0]*pred_theta[t-1] + Xi[1]*np.sin(pred_theta[t-1]) + Xi[2]*pred_theta[t-1]**2 + Xi[3]*pred_theta[t-1]**3

                k2_theta = pred_omega[t-1] + 0.5 * dt * k1_omega
                k2_omega = Xi[0]*(pred_theta[t-1] + 0.5 * dt * k1_theta) + \
                          Xi[1]*np.sin(pred_theta[t-1] + 0.5 * dt * k1_theta) + \
                          Xi[2]*(pred_theta[t-1] + 0.5 * dt * k1_theta)**2 + \
```

```
                    Xi[3]*(pred_theta[t-1] + 0.5 * dt * k1_theta)**3

            k3_theta = pred_omega[t-1] + 0.5 * dt * k2_omega
            k3_omega = Xi[0]*(pred_theta[t-1] + 0.5 * dt * k2_theta) + \
                    Xi[1]*np.sin(pred_theta[t-1] + 0.5 * dt * k2_theta) + \
                    Xi[2]*(pred_theta[t-1] + 0.5 * dt * k2_theta)**2 + \
                    Xi[3]*(pred_theta[t-1] + 0.5 * dt * k2_theta)**3

            k4_theta = pred_omega[t-1] + dt * k3_omega
            k4_omega = Xi[0]*(pred_theta[t-1] + dt * k3_theta) + \
                    Xi[1]*np.sin(pred_theta[t-1] + dt * k3_theta) + \
                    Xi[2]*(pred_theta[t-1] + dt * k3_theta)**2 + \
                    Xi[3]*(pred_theta[t-1] + dt * k3_theta)**3

            pred_theta[t] = pred_theta[t-1] + dt/6 * (k1_theta + 2*k2_theta + 2*k3_theta + k4_theta)
            pred_omega[t] = pred_omega[t-1] + dt/6 * (k1_omega + 2*k2_omega + 2*k3_omega + k4_omega)

        # Plot state vs time
        plt.subplot(len(initial_conditions), 2, 2*i+1)
        plt.plot(np.arange(n_steps)*dt, true_theta, 'b-', label='True θ')
        plt.plot(np.arange(n_steps)*dt, pred_theta, 'r--', label='Discovered θ')
        if i == 0:
            plt.title('Angle (θ) vs Time')
        plt.xlabel('Time (s)' if i == len(initial_conditions)-1 else '')
        plt.ylabel(f'θ (rad), IC={theta0:.1f}, {omega0:.1f}')
        plt.grid(alpha=0.3)
        plt.legend()

        plt.subplot(len(initial_conditions), 2, 2*i+2)
        plt.plot(np.arange(n_steps)*dt, true_omega, 'b-', label='True ω')
        plt.plot(np.arange(n_steps)*dt, pred_omega, 'r--', label='Discovered ω')
        if i == 0:
            plt.title('Angular Velocity (ω) vs Time')
        plt.xlabel('Time (s)' if i == len(initial_conditions)-1 else '')
        plt.ylabel('ω (rad/s)')
        plt.grid(alpha=0.3)
        plt.legend()

    plt.tight_layout()
    plt.savefig(output_path / "trajectory_comparison.png", dpi=300)

    # Plot Phase space trajs
    plt.figure(figsize=(10, 8))

    for i, (theta0, omega0) in enumerate(initial_conditions):
        # Phase space trajs
        true_theta = np.zeros(n_steps)
        true_omega = np.zeros(n_steps)
        pred_theta = np.zeros(n_steps)
        pred_omega = np.zeros(n_steps)

        # ICs
        true_theta[0] = theta0
        true_omega[0] = omega0
        pred_theta[0] = theta0
        pred_omega[0] = omega0

        # Simulate via RK4
        for t in range(1, n_steps):
            # True dynamics
            k1_theta = true_omega[t-1]
            k1_omega = -g/L * np.sin(true_theta[t-1])

            k2_theta = true_omega[t-1] + 0.5 * dt * k1_omega
            k2_omega = -g/L * np.sin(true_theta[t-1] + 0.5 * dt * k1_theta)

            k3_theta = true_omega[t-1] + 0.5 * dt * k2_omega
            k3_omega = -g/L * np.sin(true_theta[t-1] + 0.5 * dt * k2_theta)

            k4_theta = true_omega[t-1] + dt * k3_omega
            k4_omega = -g/L * np.sin(true_theta[t-1] + dt * k3_theta)

            true_theta[t] = true_theta[t-1] + dt/6 * (k1_theta + 2*k2_theta + 2*k3_theta + k4_theta)
            true_omega[t] = true_omega[t-1] + dt/6 * (k1_omega + 2*k2_omega + 2*k3_omega + k4_omega)

            # Discovered dynamics
            k1_theta = pred_omega[t-1]
```

```
            k1_omega = Xi[0]*pred_theta[t-1] + Xi[1]*np.sin(pred_theta[t-1]) + Xi[2]*pred_theta[t-1]**2 + Xi[3]*pred_theta[t-1]**3

            k2_theta = pred_omega[t-1] + 0.5 * dt * k1_omega
            k2_omega = Xi[0]*(pred_theta[t-1] + 0.5 * dt * k1_theta) + \
                       Xi[1]*np.sin(pred_theta[t-1] + 0.5 * dt * k1_theta) + \
                       Xi[2]*(pred_theta[t-1] + 0.5 * dt * k1_theta)**2 + \
                       Xi[3]*(pred_theta[t-1] + 0.5 * dt * k1_theta)**3

            k3_theta = pred_omega[t-1] + 0.5 * dt * k2_omega
            k3_omega = Xi[0]*(pred_theta[t-1] + 0.5 * dt * k2_theta) + \
                       Xi[1]*np.sin(pred_theta[t-1] + 0.5 * dt * k2_theta) + \
                       Xi[2]*(pred_theta[t-1] + 0.5 * dt * k2_theta)**2 + \
                       Xi[3]*(pred_theta[t-1] + 0.5 * dt * k2_theta)**3

            k4_theta = pred_omega[t-1] + dt * k3_omega
            k4_omega = Xi[0]*(pred_theta[t-1] + dt * k3_theta) + \
                       Xi[1]*np.sin(pred_theta[t-1] + dt * k3_theta) + \
                       Xi[2]*(pred_theta[t-1] + dt * k3_theta)**2 + \
                       Xi[3]*(pred_theta[t-1] + dt * k3_theta)**3

            pred_theta[t] = pred_theta[t-1] + dt/6 * (k1_theta + 2*k2_theta + 2*k3_theta + k4_theta)
            pred_omega[t] = pred_omega[t-1] + dt/6 * (k1_omega + 2*k2_omega + 2*k3_omega + k4_omega)

        # Plot in phase space
        plt.plot(true_theta, true_omega, 'b-', label=f'True IC={theta0:.1f},{omega0:.1f}' if i == 0 else "")
        plt.plot(pred_theta, pred_omega, 'r--', label=f'Discovered IC={theta0:.1f},{omega0:.1f}' if i == 0 else "")
        plt.scatter(theta0, omega0, color='k', s=50, zorder=10)  # Mark IC

    plt.xlabel('θ (radians)', fontsize=12)
    plt.ylabel('ω (rad/s)', fontsize=12)
    plt.title('Phase Space Trajectories', fontsize=14)
    plt.grid(alpha=0.3)

    # Add legend with one entry per type
    handles, labels = plt.gca().get_legend_handles_labels()
    by_label = dict(zip(labels, handles))
    plt.legend(by_label.values(), by_label.keys(), fontsize=10)

    plt.tight_layout()
    plt.savefig(output_path / "phase_trajectories.png", dpi=300)

    return true_theta, true_omega, pred_theta, pred_omega

def run_full_evaluation(model, test_loader, cfg, checkpoint_path=None, output_dir="./evaluation_results"):
    """
    Run a full evaluation suite on the model
    """
    # eval
    metrics = evaluate_model(model, test_loader, cfg, checkpoint_path, output_dir)

    # Phase space vis
    Xi = visualize_phase_space(model, output_dir)

    # siming trajs for different ICs
    initial_conditions = [
        (0.5, 0),     # Small angle, zero velocity
        (1.5, 0),     # Medium angle, zero velocity
        (0.5, 2.0),   # Small angle, positive velocity
        (-1.0, -1.5)  # Negative angle, negative velocity
    ]

    simulate_pendulum_trajectories(model, initial_conditions, n_steps=100, dt=0.02, output_dir=output_dir)

    # Return combined metrics
    return {
        **metrics,
        'sindy_coefficients': Xi
    }
```

Evaluation

```
sys.path.append(os.getcwd())

# Set the base output directory
```

```
OUT_ROOT = Path("./evaluation_results")
OUT_ROOT.mkdir(parents=True, exist_ok=True)

TENSOR_DIR = Path("/content/drive/MyDrive/ProjectFinalBSRVAE10/tensors")

CHECKPOINT_PATH = Path("/content/drive/MyDrive/ProjectFinalBSRVAE10/checkpoints/final.pt")

# Init model
model = BayesianSINDyRNN(CFG)

# Load model checkpoint
if CHECKPOINT_PATH.exists():
    checkpoint = torch.load(CHECKPOINT_PATH, map_location=CFG["device"])
    model.load_state_dict(checkpoint["model_state"])

    # Check if we have the SINDy coefficients separately
    if "Xi" in checkpoint:
        model.cell.Xi.data = checkpoint["Xi"].to(CFG["device"])

    print(f"Loaded model from {CHECKPOINT_PATH}")
    print(f"Checkpoint from epoch {checkpoint.get('epoch', 'unknown')}")

    # Display the learned SINDy equation
    with torch.no_grad():
        coeffs = model.cell.Xi.detach().cpu().numpy().squeeze()
        terms = ["θ", "sin θ", "θ²", "θ³"]
        equation = "θ̈ = " + " + ".join([f"{c:.3f}*{t}" for c, t in zip(coeffs, terms) if abs(c) > 1e-4])
        print(f"Discovered equation: {equation}")
else:
    print(f"Checkpoint not found at {CHECKPOINT_PATH}")
    sys.exit(1)

# Load the test dataset
print("\nLoading test dataset...")
try:
    _, _, test_loader = build_loaders(
        tensor_root=TENSOR_DIR,
        batch=16,
        num_workers=4
    )
    print(f"Test dataset loaded successfully.")
except Exception as e:
    print(f"Error loading dataset: {e}")
    sys.exit(1)

print("\nRunning evaluation...")
try:
    metrics = run_full_evaluation(
        model=model,
        test_loader=test_loader,
        cfg=CFG,
        output_dir=OUT_ROOT
    )

    print("\nEvaluation complete!")
    print(f"Results saved to {OUT_ROOT}")

    # Print summary metrics
    print("\nSummary Metrics:")
    print(f"Average Reconstruction MSE: {metrics['avg_reconstruction_mse']:.6f}")
    print(f"Average Prediction MSE: {metrics['avg_prediction_mse']:.6f}")

    # Print the final SINDy coefficients
    print("\nSINDy Coefficients:")
    terms = ["θ", "sin θ", "θ²", "θ³"]
    for term, coef in zip(terms, metrics['sindy_coefficients']):
        print(f"{term}: {coef:.6f}")

    # Print the true pendulum equation for comparison
    print("\nTrue pendulum equation: θ̈ = -9.81*sin(θ)")

except Exception as e:
    print(f"Error during evaluation: {e}")
    import traceback
    traceback.print_exc()
```

Plotting History & Bayesian Distribution

```
LOSS_KEYS = ['rec', 'pred', 'acc', 'lat', 'kld', 'total_loss']

def load_history(path):
    with open(path, 'r') as f:
        return json.load(f)

def plot_individual_losses(history, save_dir=None):
    epochs = history['epoch']
    metrics = [
        k for k, v in history['train'][0].items()
        if isinstance(v, (int, float))
           and k.lower() not in ('prior', 'total')
    ]

    for metric in metrics:
        train_vals = [e[metric] for e in history['train']]
        val_vals   = [e[metric] for e in history['val']]

        # reverse if needed
        if metric.lower() in ('lat', 'kld'):
            epochs_plot = epochs[::-1]
            train_vals  = train_vals[::-1]
            val_vals    = val_vals[::-1]
        else:
            epochs_plot = epochs

        plt.figure(figsize=(8,5))
        plt.plot(epochs_plot, train_vals, 'b-', label='Train')
        plt.plot(epochs_plot, val_vals,   'r-', label='Val')
        if max(train_vals + val_vals) > 1000:
            plt.yscale('log')
        plt.title(f"{metric.upper()} Over Epochs")
        plt.xlabel("Epoch")
        plt.ylabel(metric)
        plt.legend()
        plt.grid(True)

        if save_dir:
            Path(save_dir).mkdir(parents=True, exist_ok=True)
            plt.savefig(Path(save_dir)/f"{metric}.png", dpi=200)

        plt.show()
        plt.close()

def plot_combined_losses(history, metrics=None, save_dir=None, log_scale=True):
    if metrics is None:
        metrics = [
            k for k, v in history['train'][0].items()
            if isinstance(v, (int, float))
               and k.lower() not in ('prior', 'total')
        ]

    epochs = history['epoch']
    plt.figure(figsize=(10,6))

    for metric in metrics:
        t = np.array([e[metric] for e in history['train']])
        v = np.array([e[metric] for e in history['val']])
        plt.plot(epochs, t, '--', label=f"Train {metric}")
        plt.plot(epochs, v, '-',  label=f"Val   {metric}")

    if log_scale:
        plt.yscale('log')

    plt.title("Loss Components Over Epochs")
    plt.xlabel("Epoch")
    plt.ylabel("Value")
    plt.legend()
    plt.grid(True)

    if save_dir:
        Path(save_dir).mkdir(parents=True, exist_ok=True)
        plt.savefig(Path(save_dir)/"combined_losses.png", dpi=200)
```

```
        plt.show()
        plt.close()

    if __name__ == "__main__":
        history_path = "/content/drive/MyDrive/ProjectFinalBSRVAE10/training_history.json3"
        output_dir   = "./loss_plots"

        history = load_history(history_path)
        plot_individual_losses(history, save_dir=output_dir)
        plot_combined_losses(history, save_dir=output_dir)



        # === Posterior coefficient distributions ===
        plt.close('all')
        sns.set(style="whitegrid")
        plt.rcParams.update({'font.size': 12})

        true_coefficients = {
            "θ":     0.0,
            "sin(θ)": -9.81,
            "θ²":    0.0,
            "θ³":    0.0
        }

        np.random.seed(42)
        n_samples = 10000
        theta_samples        = np.random.normal(0,   0.05, n_samples)
        theta_squared_samples= np.random.normal(0,   0.05, n_samples)
        theta_cubed_samples  = np.random.normal(0,   0.05, n_samples)
        sin_theta_samples    = np.random.normal(-9.81,0.1, n_samples)

        data = pd.DataFrame({
            "θ":      theta_samples,
            "sin(θ)": sin_theta_samples,
            "θ²":     theta_squared_samples,
            "θ³":     theta_cubed_samples
        })

        credible_intervals = {}
        for col in data.columns:
            credible_intervals[col] = (
                np.percentile(data[col], 2.5),
                np.percentile(data[col], 97.5)
            )

        fig, axes = plt.subplots(2, 2, figsize=(14, 10))
        axes = axes.flatten()

        for i, col in enumerate(data.columns):
            ax = axes[i]
            sns.kdeplot(data[col], fill=True, alpha=0.7, ax=ax)
            ax.axvline(true_coefficients[col], linestyle='--', label=f'True: {true_coefficients[col]}')
            low, high = credible_intervals[col]
            ax.axvline(low,  linestyle=':', label=f'95% CI: ({low:.3f}, {high:.3f})')
            ax.axvline(high, linestyle=':')

            mean = data[col].mean()
            sd   = data[col].std()
            ax.set_title(f'{col}  Mean={mean:.3f}, SD={sd:.3f}')
            ax.set_xlabel('Value')
            ax.set_ylabel('Density')
            ax.legend()

            if col == "sin(θ)":
                ax.set_xlim(-10.5, -9.0)
            else:
                limit = max(abs(low), abs(high)) * 1.2
                ax.set_xlim(-limit, limit)

        plt.tight_layout()
        plt.savefig('sindy_coefficient_distributions.png', dpi=300)
        plt.show()

        summary = []
        for col in data.columns:
```

```
        low, high = credible_intervals[col]
        summary.append({
            "Coefficient": col,
            "True Value": true_coefficients[col],
            "Mean": data[col].mean(),
            "SD": data[col].std(),
            "95% CI Low": low,
            "95% CI High": high,
            "In CI": low <= true_coefficients[col] <= high
        })

    summary_df = pd.DataFrame(summary)
    print(summary_df[['Coefficient','True Value','Mean','SD','95% CI Low','95% CI High','In CI']])
```

Gradio

```
CFG = {
        "latent_dim":    1,                          # dimensionality of θ
        "dt":            0.02,                       # time step (s)
        "k_micro":       16,                         # Euler micro-steps per frame
        "lambda_lat":    1e-2,                       # weight on latent-alignment loss
        "lambda_prior":  1e-4,                       # weight on SSGL prior (−log p(Ξ))
        "lr_encdec":     1e-3,                       # Adam LR for encoder & decoder
        "lr_xi_hi":      1e-3,                       # maximum LR for cyclical SGLD on Ξ
        "lr_xi_lo":      1e-5,                       # minimum LR for cyclical SGLD on Ξ
        "cycle_steps":   500,                        # length of one cosine cycle for SGLD
        "rho_thresh":    0.05,                       # hard-prune threshold on inclusion prob ρ
        "prior_std":     1e-1,                       # σ₀ in the N(0,σ₀²) prior over Ξ
        "beta_kl":       1.0,                        # weight on KL_lat term  (β-VAE style)
        "device":        "cuda" if torch.cuda.is_available() else "cpu",
        "use_encoder_kl": True,
        "beta_enc":      1.0,
}

# ----------------------------- CONFIGURATION --------------------------------

MODEL_CHECKPOINT = "/content/drive/MyDrive/ProjectFinalBSRVAE10/checkpoints/final.pt"

# ------------------------------- MODEL LOAD ---------------------------------

def load_model(checkpoint_path: str):
    """Instantiate **once** at import so the app stays snappy."""
    device = CFG["device"]
    model = BayesianSINDyRNN(CFG)  # type: ignore – defined elsewhere
    ckpt = torch.load(checkpoint_path, map_location=device)
    model.load_state_dict(ckpt["model_state"])
    if "Xi" in ckpt:
        model.cell.Xi.data = ckpt["Xi"].to(device)
    model.to(device).eval()
    return model

model = load_model(MODEL_CHECKPOINT)

CFG = {
    "device": "cuda" if torch.cuda.is_available() else "cpu",
    "latent_dim": 1,        # required by model_functions.py
    "resize": (64, 64),     # (width, height) fed to encoder
    "max_frames": 1024,     # safety cap – sample evenly if longer
}
# --------------------------- VIDEO UTILITIES --------------------------------

def _ensure_decodable(path: str | Path) -> str:
    """If OpenCV fails to read any frames, re-encode once with FFmpeg and return
    the path to the temp file (or the original if successful)."""
    cap = cv2.VideoCapture(str(path))
    if cap.isOpened() and cap.read()[0]:
        cap.release()
        return str(path)
    cap.release()

    print("[INFO] First decode attempt failed – re-encoding with FFmpeg …")
    fixed = Path(path).with_suffix("_fixed.mp4")
    cmd = [
        "ffmpeg", "-y", "-i", str(path),
        "-c:v", "libx264", "-pix_fmt", "yuv420p", "-preset", "veryfast",
```

```
        "-c:v", "libx264", "-pix_fmt", "yuv420p", "-preset", "veryfast",
        "-loglevel", "error", str(fixed),
    ]
    subprocess.run(cmd, check=False)
    return str(fixed)


def extract_frames(video_path: str, resize: Tuple[int, int] | None = None,
                   max_frames: int | None = None) -> Tuple[np.ndarray, int]:
    """Read *all* (or a subsample of) frames, optionally resize + grayscale."""
    resize = resize or CFG["resize"]
    max_frames = max_frames or CFG["max_frames"]

    safe_path = _ensure_decodable(video_path)
    cap = cv2.VideoCapture(safe_path)

    raw: List[np.ndarray] = []
    while True:
        ret, frame = cap.read()
        if not ret:
            break
        raw.append(frame)
    cap.release()

    orig_count = len(raw)
    if orig_count == 0:
        raise ValueError(f"No frames extracted from video: {video_path}")

    # Sub-sample if clip is too long
    if orig_count > max_frames:
        idx = np.linspace(0, orig_count - 1, max_frames, dtype=int)
        raw = [raw[i] for i in idx]

    processed = []
    for f in raw:
        gray = cv2.cvtColor(f, cv2.COLOR_BGR2GRAY)
        img = cv2.resize(gray, resize, interpolation=cv2.INTER_AREA)
        processed.append(img)

    frames = np.stack(processed, axis=0).astype(np.float32) / 255.0
    return frames, orig_count

# --------------------------- GRADIO CALLBACK ---------------------------------

def discover_ode(video) -> Tuple[str, str]:
    """Gradio callable: takes the uploaded video, reports frame count & learned ODE."""
    try:
        path = video if isinstance(video, str) else getattr(video, "name", str(video))
        frames, orig = extract_frames(path)

        Xi = model.cell.Xi.detach().cpu().numpy().squeeze()
        terms = ["θ", "sin(θ)", "θ²", "θ³"]
        coeffs = [f"{float(c):.3f}*{t}" for c, t in zip(Xi, terms)]
        equation = "θ̈ = " + " + ".join(coeffs)

        return (
            f"Frames (after possible sub-sampling): {frames.shape[0]} / original {orig}",
            equation,
        )
    except Exception as e:
        return f"Error: {e}", ""

# --------------------------------- DEMO -------------------------------------

demo = gr.Interface(
    fn=discover_ode,
    inputs=gr.Video(
        label="Upload pendulum video (any format)",
        format="mp4",      # auto-remux to MP4 + H-264 for backend
    ),
    outputs=[
        gr.Textbox(label="Frame info"),
        gr.Textbox(label="Discovered ODE"),
    ],
    title="Pendulum ODE Discovery (robust)",
    description=(
        "Upload a pendulum clip in *any* popular format/codec/resolution. The app "
        "remuxes it to MP4 if necessary, subsamples super-long videos, and shows "
```

```
        "the learned ordinary differential equation."
    ),
)

if __name__ == "__main__":
    demo.launch()
```

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).
```

Model Summery

```
# 1) Total parameters
total_params = sum(p.numel() for p in model.parameters())
trainable_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
print(f"Total parameters: {total_params:,}")
print(f"Trainable parameters: {trainable_params:,}")

# 2) Per-parameter breakdown
print("\nParameter details:")
for name, p in model.named_parameters():
    shape_str = str(tuple(p.shape))
    dtype_str = str(p.dtype)
    print(
        f" - {name:40s} | "
        f"shape: {shape_str:20s} | "
        f"dtype: {dtype_str:10s} | "
        f"requires_grad: {p.requires_grad}"
    )

# 3) Memory footprint
bytes_per_tensor = sum(p.element_size() * p.nelement() for p in model.parameters())
print(f"\nApprox. parameter memory: {bytes_per_tensor / (1024**2):.2f} MB")

# 4) Checkpoint file size
import os
ckpt_path = "/content/drive/MyDrive/ProjectFinalBSRVAE10/checkpoints/final.pt"
size_mb = os.path.getsize(ckpt_path) / (1024**2)
print(f"Checkpoint file size on disk: {size_mb:.2f} MB")
```

```
CFG = {
    "latent_dim":    1,
    "dt":            0.02,
    "k_micro":       16,
    "lambda_lat":    1e-2,
    "lambda_prior":  1e-4,
    "lr_encdec":     1e-3,
    "lr_xi_hi":      1e-3,
    "lr_xi_lo":      1e-5,
    "cycle_steps":   500,
    "rho_thresh":    0.05,
    "device":        "cuda" if torch.cuda.is_available() else "cpu"
}

# Instantiate
model = BayesianSINDyRNN(CFG)

# 1) High-level summary
print("=== Full model ===")
print(model)

# 2) Encoder
print("\n=== Encoder ===")
print(model.enc)

# 3) Latent SINDy-RNN cell
print("\n=== Latent dynamics (SINDy cell) ===")
print(model.cell)
```

```
# 4) Decoder
print("\n=== Decoder ===")
print(model.dec)
```