# PendulumDataset — densification version

The class is **back-compatible** (old calls work).
Three optional knobs let you make phase-space samples denser or wider:

| knob | intent | how it works | effect |
|---|---|---|---|
| `sub_steps` (int, default `1`) | higher **temporal** resolution | does *N* internal `env.step()` calls before storing **one** frame | ≈ *N*× more (θ, ω) points per orbit & smoother ω estimates |
| `init_grid` (list of `(θ₀, ω₀)` tuples) | wider **phase-space** coverage | if provided, each tuple becomes one episode; `num_episodes` is ignored | training sees many energies/angles → better generalisation |
| `random_action` (bool, default `True`) | choose stochastic vs. pure swing | `True`: sample torque (old behaviour) `False`: always action = 0 | clean conservative orbits—nice for energy plots & HNN/LNN |

# Code

```
class PendulumDataset(Dataset):
    def __init__(self,
                 num_episodes: int = 100,
                 episode_length: int = 200,
                 img_size: int = 64,
                 seq_len: int = 3,
                 *,
                 sub_steps: int = 1,                        # knob ①
                 init_grid: list[tuple[float, float]] | None = None,  # knob ②
                 random_action: bool = True,                # knob ③
                 transform=None):
        """
        Returns sequences of length `seq_len` (T,C,H,W) plus (θ, ω).

        Parameters
        ----------
        sub_steps : int
            How many env.step() calls between stored frames.
        init_grid : list[(theta0, omega0)]
            Explicit start states; overrides num_episodes.
        random_action : bool
            True → random torque.  False → action = 0 (no external force).
        """
        assert seq_len >= 2, "seq_len must be ≥ 2 for physics loss"
        self.img_size, self.seq_len = img_size, seq_len
        self.sub_steps = max(1, sub_steps)
        self.init_grid = init_grid
        self.random_action = random_action
        self.transform = transform

        self.frames, self.states, self.indices = [], [], []
        self._generate(num_episodes, episode_length)

    # ── draw pendulum ─────────────────────────────────────────
    def _render_pendulum(self, theta):
        L, cx, cy = self.img_size * 0.4, self.img_size // 2, self.img_size // 2
        ex, ey = int(cx + L * np.sin(theta)), int(cy + L * np.cos(theta))
        img = Image.new("RGB", (self.img_size, self.img_size), "white")
        d   = ImageDraw.Draw(img)
```

```
        d.line([(cx, cy), (ex, ey)], fill="black", width=3)
        d.ellipse([(cx-5, cy-5), (cx+5, cy+5)], fill="red")
        d.ellipse([(ex-8, ey-8), (ex+8, ey+8)], fill="blue")
        return np.asarray(img)

    # ── rollout generator ──────────────────────────────────────
    def _generate(self, n_episodes, epi_len):
        print("Generating pendulum trajectories …")
        env = gym.make("Pendulum-v1")

        # choose seeds: grid or default random resets
        seeds = self.init_grid if self.init_grid is not None else [None] * n_episodes
        for seed in tqdm(seeds):
            # set initial state
            if seed is None:
                obs, _ = env.reset()
            else:
                theta0, omega0 = seed
                env.reset()
                env.unwrapped.state = np.array([theta0, omega0], dtype=np.float32)
                obs = env.unwrapped.state_to_obs()

            ep_imgs, ep_states = [], []
            for _ in range(epi_len):
                # finer Δt by sub-stepping
                for _ in range(self.sub_steps):
                    action = env.action_space.sample() if self.random_action else np.array
                    obs, _, done, trunc, _ = env.step(action)
                    if done or trunc:
                        break

                theta, omega = np.arctan2(obs[1], obs[0]), float(obs[2])
                ep_imgs.append(self._render_pendulum(theta))
                ep_states.append((theta, omega))

            # sliding window indexes
            for t0 in range(0, len(ep_imgs) - self.seq_len + 1):
                self.indices.append(len(self.frames) + t0)

            self.frames.extend(ep_imgs)
```

```
            self.states.extend(ep_states)

        env.close()
        print(f"Created {len(self.indices)} windows (seq_len={self.seq_len})")

    # ── PyTorch Dataset protocol ──────────────────────────────
    def __len__(self) -> int:
        return len(self.indices)

    def __getitem__(self, idx):
        start = self.indices[idx]
        end   = start + self.seq_len

        imgs = [torch.from_numpy(self.frames[i]).float().permute(2,0,1)/255.
                for i in range(start, end)]
        imgs = torch.stack(imgs)          # (T,C,H,W)

        states = torch.tensor(self.states[start:end], dtype=torch.float32)  # (T,2)
        if self.transform:
            imgs = self.transform(imgs)
        return imgs, states
```