Exercise 6: SINDy-Autoencoders

In [ ]:
```
import numpy as np
import pandas as pd
import pickle
import matplotlib.pyplot as plt

from tqdm import tqdm
from scipy.integrate import odeint
from sklearn import linear_model

import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.optim import Adam
```

In [ ]:
```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print(device)
```

```
cuda:0
```

In [ ]:
```
N_REPEAT = 10

OPTIMIZE_AUTOENCODER_HYPERPARAMETERS = False
TRAIN_SMALL_ANGLE_APPROXIMATION = False
```

1.1 SINDy in Ground Truth Coordinates z

1.1 Simulation

In [ ]:
```
T = 50  # number of steps of the simulation
DT = 0.02  # time in seconds of each step
```

Term Library

In [ ]:
```
terms_np =  {
    '1': lambda z, dz, device: np.ones_like(z),
    'z': lambda z, dz, _: z,
    'dz': lambda z, dz, _: dz,
    'sin(z)': lambda z, dz, _: np.sin(z),
    'z^2': lambda z, dz, _: z**2,
    'z dz': lambda z, dz, _: z * dz,
    'z sin(z)': lambda z, dz, _: z * z * np.sin(z),
    'dz^2': lambda z, dz, _: dz**2,
    'dz sin(z)': lambda z, dz, _: dz * dz * np.sin(z),
    'sin(z)^2': lambda z, dz, _: np.sin(z)**2,
}
```

In [ ]:
```
terms_torch =  {
    '1': lambda z, dz, device: torch.ones_like(z, device=device),
    'z': lambda z, dz, _: z,
    'dz': lambda z, dz, _: dz,
    'sin(z)': lambda z, dz, _: torch.sin(z),
    'z^2': lambda z, dz, _: z**2,
    'z dz': lambda z, dz, _: z * dz,
    'z sin(z)': lambda z, dz, _: z * torch.sin(z),
    'dz^2': lambda z, dz, _: dz**2,
    'dz sin(z)': lambda z, dz, _: dz * torch.sin(z),
    'sin(z)^2': lambda z, dz, _: torch.sin(z)**2,
}
```

In [ ]:
```
target_coefficients = {
    '1': 0,
    'z': 0,
    'dz': 0,
    'sin(z)': -1,
    'z^2': 0,
    'z dz': 0,
    'z sin(z)': 0,
    'dz^2': 0,
    'dz sin(z)': 0,
    'sin(z)^2': 0,
}
```

Simulation Functions

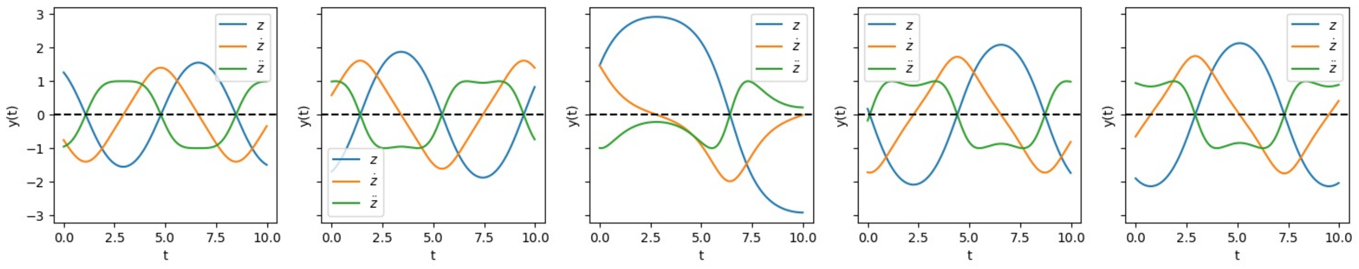
```
In [ ]: def pendulum_rhs(z, dz, coefficients, terms):
            """
            Compute the right hand side of the pendulum ODE
            """
            return np.sum([coef * term(z, dz, "cpu") for coef, term in zip(coefficients, terms)], axis=0)

        # The function that returns dy/dt
        def pendulum_ode_step(y, t, coefficients, terms):
            """
            Perform the integration step of the pendulum ODE
            """
            z, dz = y.T
            dydt = [dz, pendulum_rhs(z, dz, coefficients, terms)]
            return np.array(dydt).T

        def simulate_pendulum(z0, dz0, coefficients, terms, T, dt):
            """
            Simulate the pendulum ODE for the given initial conditions with the `pendulum_ode_step` integration step
            """
            # Create T time points dt apart
            t = np.arange(0, T) * dt

            # Solve ODE
            y = np.empty((z0.shape[0], t.shape[0], 2))

            for i in range(z0.shape[0]):
                y0 = [z0[i], dz0[i]]
                y[i,:,:] = odeint(pendulum_ode_step, y0, t, args=(coefficients, terms))

            return t, y[:,:,0], y[:,:,1]

        def create_pendulum_data(z0_min, z0_max, dz0_min, dz0_max, coefficients, terms, T, dt, N, embedding=None, rejec
            z = np.empty(N)
            dz = np.empty(N)

            i = 0
            rejections = 0
            MAX_REJECTIONS = N * 10
            while i < N:
                z0 = np.random.uniform(z0_min, z0_max)
                dz0 = np.random.uniform(dz0_min, dz0_max)

                # Check if the initial conditions are valid
                if np.abs(dz0**2/2. - np.cos(z0)) <= 0.99 or not reject_invalid:
                    z[i] = z0
                    dz[i] = dz0
                    i += 1
                else:
                    rejections += 1
                    if rejections > MAX_REJECTIONS:
                        raise ValueError("Too many rejections")

            t, z, dz = simulate_pendulum(z, dz, coefficients, terms, T, dt)
            ddz = pendulum_rhs(z, dz, coefficients, terms)

            if embedding is not None:
                x, dx, ddx = embedding(z, dz, ddz, t)
            else:
                x = None
                dx = None
                ddx = None

            return x, dx, ddx, z, dz, ddz, t
```

Verification

```
In [ ]: # Simulate
        x, dx, ddx, z, dz, ddz, t = create_pendulum_data(
            z0_min=-np.pi,
            z0_max=np.pi,
            dz0_min=-2.1,
            dz0_max=2.1,
            coefficients=[target_coefficients[term] for term in terms_np],
            terms=[terms_np[term] for term in terms_np],
            T=T * 10,
            dt=DT,
            N=500
        )

        print(f"{t.shape = }")
        print(f"{z.shape = }")
        print(f"{dz.shape = }")
        print(f"{ddz.shape = }")
```

```
t.shape = (500,)
z.shape = (500, 500)
dz.shape = (500, 500)
ddz.shape = (500, 500)
```

In [ ]:
```
fig, ax = plt.subplots(1, 5, figsize=(18, 3), sharey=True)

for i in range(5):
    ax[i].plot(t, z[i,:], label='$z$')
    ax[i].plot(t, dz[i,:], label='$\dot z$')
    ax[i].plot(t, ddz[i,:], label='$\ddot z$')

    ax[i].axhline(0, color='black', linestyle='--')

    ax[i].set_xlabel('t')
    ax[i].set_ylabel('y(t)')
    ax[i].legend()
```

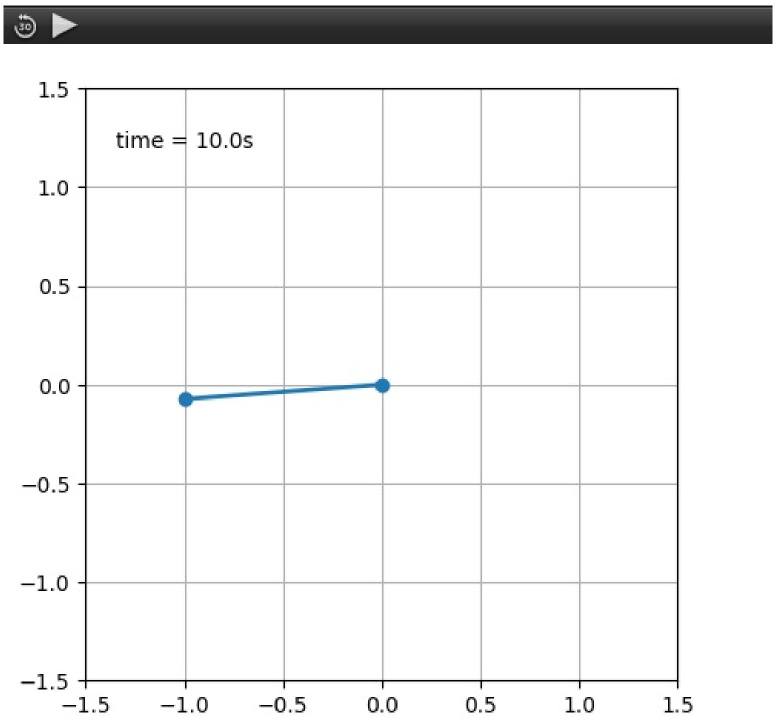


Animation

In [ ]:
```
tip_positions = np.stack([np.sin(z), -np.cos(z)]).transpose(1,2,0)

print(f"{tip_positions.shape = }")
```

```
tip_positions.shape = (500, 500, 2)
```

In [ ]:
```
# Animate the pendulum
from matplotlib.animation import FuncAnimation
from IPython.display import HTML

fig, ax = plt.subplots(figsize=(5, 5))
ax.set_xlim(-1.5, 1.5)
ax.set_ylim(-1.5, 1.5)
ax.set_aspect('equal')
ax.grid()

line, = ax.plot([], [], 'o-', lw=2)
time_template = 'time = %.1fs'
time_text = ax.text(0.05, 0.9, '', transform=ax.transAxes)

def init():
    line.set_data([], [])
    time_text.set_text('')
    return line, time_text

def animate(i):
    line.set_data([0, tip_positions[0, i, 0]], [0, tip_positions[0, i, 1]])
    time_text.set_text(time_template % (i*DT))
    return line, time_text

anim = FuncAnimation(fig, animate, init_func=init, frames=z.shape[1], interval=DT*1000, blit=True)
display(HTML(anim.to_html5_video()))
```



1.2 Implementation & Training

Library & SINDy classes

In [ ]:
```
class Library(nn.Module):
    def __init__(self, variables, dim, terms):
        super(Library, self).__init__()

        self.variables = variables
        self.dim = dim
        self.terms = terms
        self.L = len(self.terms)
```

In [ ]:
```
# For the RHS, consider terms 1, and all combinations of z, dz, sin(z) up to total order 2
class SINDy(nn.Module):
    def __init__(self, library, thresholder=None, init="ones"):
        super(SINDy, self).__init__()

        self.library = library
        self.thresholder = thresholder

        self.device = "cpu"

        match init:
            case "ones":
                self.coef = nn.Parameter(torch.ones((self.library.dim, self.library.L)))
                self.coef_mask = nn.Parameter(torch.ones((self.library.dim, self.library.L), dtype=bool), requi
            case "normal":
                self.coef = nn.Parameter(torch.randn((self.library.dim, self.library.L)))
                self.coef_mask = nn.Parameter(torch.ones((self.library.dim, self.library.L), dtype=bool), requi
            case _:
                raise ValueError(f"Unknown init: {init}")

    def to(self, device):
        super(SINDy, self).to(device)
        self.device = device
        self.coef = self.coef.to(device)
        self.coef_mask = self.coef_mask.to(device)

        if self.thresholder is not None:
            self.thresholder = self.thresholder.to(device)

        return self

    def compute_RHS(self, z, dz):
        # Compute the terms
        RHS = torch.empty((z.shape[0], z.shape[1], self.library.L), device=self.device)

        for i, (k, f) in enumerate(self.library.terms.items()):
            RHS[:,:,i] = f(z, dz, self.device)

        return RHS, list(self.library.terms.keys())

    def forward(self, z, dz):
        rhs, _ = self.compute_RHS(z, dz)

        # Compute the linear combination
        return torch.sum(rhs * self.coef * self.coef_mask, dim=-1)
```

Verification

In [ ]:
```
lib = Library(['z', 'dz'], 1, terms_torch)
sindy = SINDy(lib)
```

In [ ]:
```
for param in sindy.parameters():
    print(param)
```

```
Parameter containing:
tensor([[1., 1., 1., 1., 1., 1., 1., 1., 1., 1.]], requires_grad=True)
Parameter containing:
tensor([[True, True, True, True, True, True, True, True, True, True]])
```

In [ ]:
```
ddz_hat = sindy.forward(torch.tensor(z[0, :6]).view(-1, 1), torch.tensor(dz[0, :6]).view(-1, 1))
ddz_hat
```

Out[ ]:
```
tensor([[5.0668],
        [4.9622],
        [4.8574],
        [4.7524],
        [4.6473],
        [4.5422]], grad_fn=<SumBackward1>)
```

In [ ]:
```
loss_fn = nn.MSELoss()

# Check if backprop works
loss = loss_fn(ddz_hat, torch.tensor(ddz[0, :6]).float().view(-1, 1))

loss.backward()

for param in sindy.parameters():
    if param.grad is not None:
        print(param.grad)
```

```
tensor([[ 11.4900,  14.0799,  -9.2276,  10.8074,  17.2623, -11.2975,  13.2463,
           7.4226,  -8.6760,  10.1663]])
```

SINDy Training Loop

In [ ]:
```
def train_sindy(loss_history, sindy, optimizer, z_train, dz_train, ddz_train, z_val, dz_val, ddz_val, epochs, r
    loss_fn = nn.MSELoss()

    loss_history['train_sindy'] = []
    loss_history['train_l1'] = []
    loss_history['val_sindy'] = []
    loss_history['val_l1'] = []
    loss_history['coefficients'] = []
    loss_history['active_terms'] = []
    pbar = tqdm(range(epochs), disable=not verbose)

    for epoch in pbar:
        # Training
        sindy.train()

        for i in range(0, z_train.shape[0], batch_size):
            # Backpropagation
            optimizer.zero_grad()

            z_batch = z_train[i : i + batch_size]
            dz_batch = dz_train[i : i + batch_size]
            ddz_batch = ddz_train[i : i + batch_size]

            ddz_hat = sindy.forward(z_batch, dz_batch)

            sindy_loss = loss_fn(ddz_hat, ddz_batch)

            if epoch >= refinement_after_epochs:
                l1_loss = torch.Tensor([0]).to(device)
            else:
                l1_loss = l1_weight * torch.norm(sindy.coef * sindy.coef_mask, p=1)

            loss = (sindy_loss + l1_loss * z_batch.shape[0])

            loss.backward()
            optimizer.step()

            loss_history['train_sindy'].append([epoch + i / z_train.shape[0], sindy_loss.item() / z_train.shape
            loss_history['train_l1'].append([epoch + i / z_train.shape[0], l1_loss.item() / z_train.shape[0]])

        # Thresholding
        if sindy.thresholder is not None:
            sindy.coef_mask.data = sindy.thresholder(sindy.coef.data, sindy.coef_mask.data)
            sindy.coef.data = sindy.coef.data * sindy.coef_mask.data
            loss_history['active_terms'].append([epoch + i / z_train.shape[0], torch.sum(sindy.coef_mask).item(
        else:
            loss_history['active_terms'].append([epoch + i / z_train.shape[0], sindy.coef.shape[1]])

        # Store the coefficients
        loss_history['coefficients'].append([epoch + i / z_train.shape[0], sindy.coef.detach().cpu().numpy().co

        # Validation
        sindy.eval()
        with torch.no_grad():
            ddz_hat = sindy.forward(z_val, dz_val)

            sindy_loss = loss_fn(ddz_hat, ddz_val)
            l1_loss = l1_weight * torch.norm(sindy.coef * sindy.coef_mask, p=1)

            loss = (sindy_loss + l1_loss * z_batch.shape[0])

            loss_history['val_sindy'].append([epoch, sindy_loss.item() / z_val.shape[0]])
            loss_history['val_l1'].append([epoch, l1_loss.item() / z_val.shape[0]])

        if verbose:
            coefs = sindy.coef.detach().cpu().numpy().copy()[0]
            coef_mask = sindy.coef_mask.detach().cpu().numpy().copy()[0]
            terms = list(sindy.library.terms.keys())
            equation_string = " ".join([f"{'+ ' if coef >= 0 else '- '}{np.abs(coef):.3f} {term}" for coef, terr
            pbar.set_description(f"Train SINDy: {sindy_loss.item():.3e} | Train L1: {l1_loss.item():.3e} | Val S
```

SINDy Dataset

In [ ]:
```
_, _, _, z_train, dz_train, ddz_train, t_train = create_pendulum_data(
    z0_min=-np.pi,
    z0_max=np.pi,
    dz0_min=-2.1,
    dz0_max=2.1,
    coefficients=[target_coefficients[term] for term in terms_np],
    terms=[terms_np[term] for term in terms_np],
    T=T,
    dt=DT,
    N=80
)

_, _, _, z_val, dz_val, ddz_val, t_val = create_pendulum_data(
    z0_min=-np.pi,
    z0_max=np.pi,
    dz0_min=-2.1,
    dz0_max=2.1,
    coefficients=[target_coefficients[term] for term in terms_np],
    terms=[terms_np[term] for term in terms_np],
    T=T,
    dt=DT,
    N=20
)
```

In [ ]:

```
# Create tensors
z_train = torch.tensor(z_train).float().view(-1, 1)
dz_train = torch.tensor(dz_train).float().view(-1, 1)
ddz_train = torch.tensor(ddz_train).float().view(-1, 1)

# Shuffle the training data
idx = torch.randperm(z_train.shape[0])
z_train = z_train[idx]
dz_train = dz_train[idx]
ddz_train = ddz_train[idx]

z_val = torch.tensor(z_val).float().view(-1, 1)
dz_val = torch.tensor(dz_val).float().view(-1, 1)
ddz_val = torch.tensor(ddz_val).float().view(-1, 1)

print(f"{z_train.shape = }")
print(f"{dz_train.shape = }")
print(f"{z_val.shape = }")
print(f"{dz_val.shape = }")
```

```
z_train.shape = torch.Size([4000, 1])
dz_train.shape = torch.Size([4000, 1])
z_val.shape = torch.Size([1000, 1])
dz_val.shape = torch.Size([1000, 1])
```

SINDy with sklearn

In [ ]:

```
# Augment the data
theta_train = np.concatenate([f(z_train, dz_train, "cpu") for f in terms_np.values()], axis=1)
print(f"{theta_train.shape = }")
```

```
theta_train.shape = (4000, 10)
```

In [ ]:

```
sindy_sklearn = linear_model.Lasso(alpha=1e-4, fit_intercept=False, max_iter=10000, tol=1e-5)

# Fit the model
sindy_sklearn.fit(theta_train, ddz_train)

# Show the learned equation
equation_str_sklearn = " + ".join([f"{coef:.3f} {term}" for coef, term in zip(sindy_sklearn.coef_, terms_np.keys
print(f"Learned equation: $\ddot z = {equation_str_sklearn}$")
```

```
Learned equation: $\ddot z = -0.000 1 + -0.000 z + -0.000 dz + -0.999 sin(z) + -0.000 z^2 + -0.000 z dz + -0.000
z sin(z) + -0.000 dz^2 + -0.000 dz sin(z) + -0.000 sin(z)^2$
```

SINDy with pytorch

In [ ]:

```
lib = Library(['z', 'dz'], 1, terms_torch)
sindy = SINDy(lib).to(device)

optimizer = Adam(sindy.parameters(), lr=5e-2)
```

In [ ]:

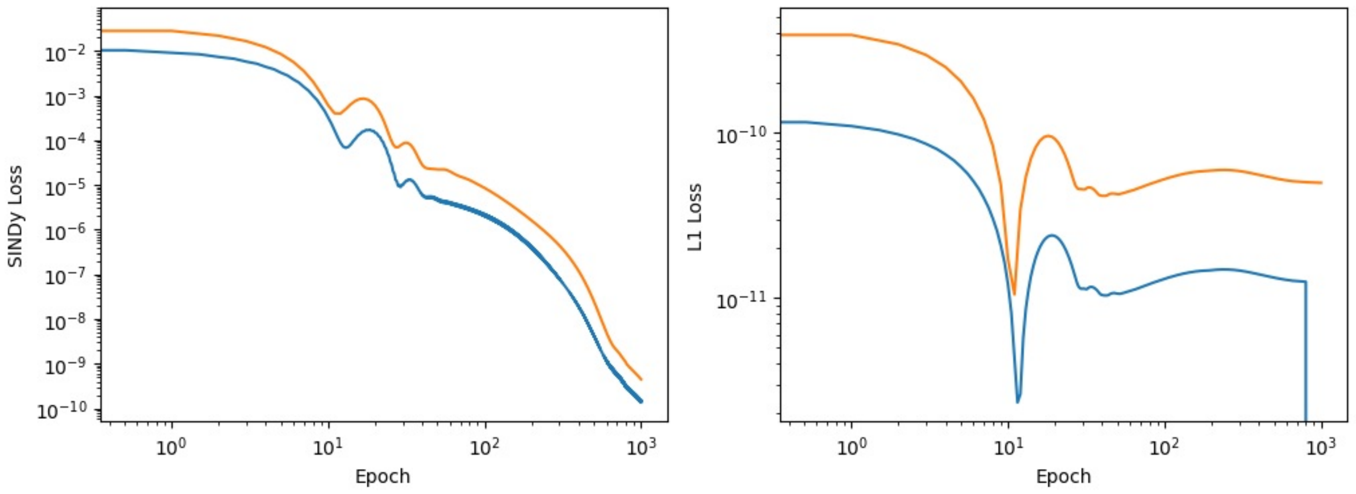
```
batch_size = 2048

loss_history = {}

train_sindy(loss_history, sindy, optimizer,
            z_train.to(device), dz_train.to(device), ddz_train.to(device),
            z_val.to(device), dz_val.to(device), ddz_val.to(device),
            epochs=1000, refinement_after_epochs=800, l1_weight=1e-4 / batch_size, batch_size=batch_size, verbo
```

```
Train SINDy: 4.485e-07 | Train L1: 4.988e-08 | Val SINDy: 4.485e-10 | Val L1: 4.988e-11 | Equation: + 0.001 1 -
0.001 z + 0.000 dz - 0.999 sin(z) - 0.002 z^2 - 0.000 z dz + 0.009 z sin(z) - 0.000 dz^2 + 0.001 dz sin(z) - 0.0
10 sin(z)^2: 100%|██████████| 1000/1000 [00:03<00:00, 315.97it/s]
```

In [ ]:

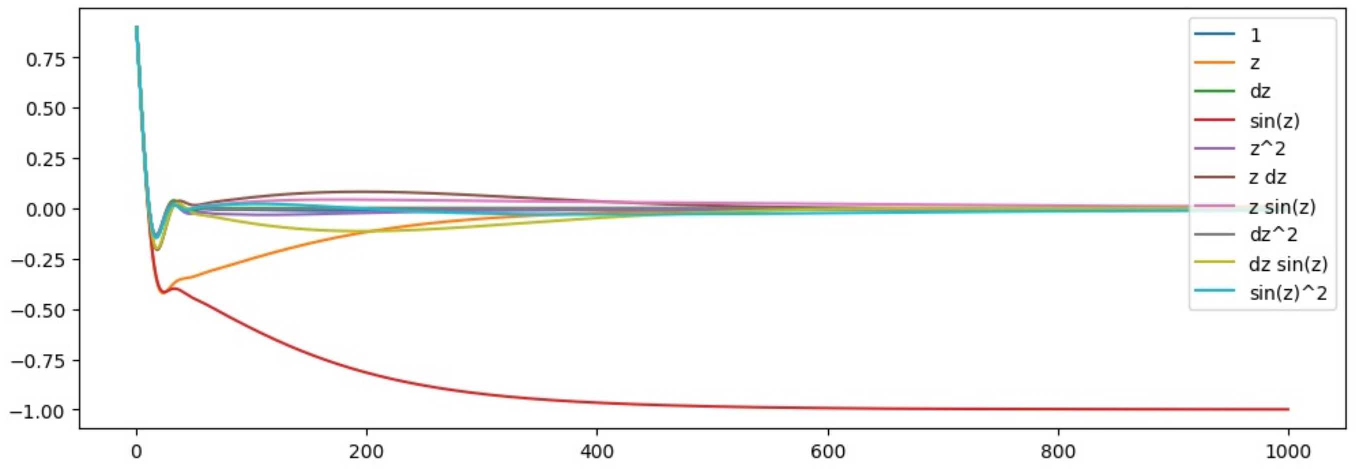
```
fig, ax = plt.subplots(1, 2, figsize=(12, 4))

ax[0].plot(*np.array(loss_history['train_sindy']).T, label='Train')
ax[0].plot(*np.array(loss_history['val_sindy']).T, label='Validation')

ax[0].set_xlabel('Epoch')
ax[0].set_ylabel('SINDy Loss')

ax[0].set_xscale('log')
ax[0].set_yscale('log')

ax[1].plot(*np.array(loss_history['train_l1']).T, label='Train')
```

```
ax[1].plot(*np.array(loss_history['val_l1']).T, label='Validation')

ax[1].set_xlabel('Epoch')
ax[1].set_ylabel('L1 Loss')

ax[1].set_xscale('log')
ax[1].set_yscale('log')
```



In [ ]:
```
# Plot the coefficient history
fig, ax = plt.subplots(figsize=(12, 4))

x = np.array([c[0] for c in loss_history['coefficients']])
Y = np.array([c[1] for c in loss_history['coefficients']])

for i in range(Y.shape[2]):
    ax.plot(x, Y[:, 0, i], label=list(terms_np.keys())[i])

ax.legend(loc='upper right')
```

Out[ ]:
```
<matplotlib.legend.Legend at 0x7f50905ebe10>
```



In [ ]:
```
# Show the learned equation
equation_str = " + ".join([f"{coef:.3f} {term}" for coef, term in zip(sindy.coef.detach().cpu().numpy().flatten
print(f"Learned equation: $\ddot z = {equation_str}$")
```

```
Learned equation: $\ddot z = 0.001 1 + -0.001 z + 0.000 dz + -0.999 sin(z) + -0.002 z^2 + -0.000 z dz + 0.009 z
sin(z) + -0.000 dz^2 + 0.001 dz sin(z) + -0.010 sin(z)^2$
```

1.3 Thresholding

Thresholding Classes

In [ ]:
```
class Thresholder():
    def __init__(self, library):
        self.library = library

    def to(self, device):
        return self

    def __call__(self, x):
        pass
```

```python
class SequentialThresholder(Thresholder):
    def __init__(self, library, threshold=0.1, interval=500):
        super(SequentialThresholder, self).__init__(library)
        self.threshold = threshold
        self.interval = interval

        self.step = 0

    def to(self, device):
        return self

    def __call__(self, coef, coef_mask):
        self.step += 1

        # If the the length of the history is a multiple of the interval
        if self.step % self.interval == 0:

            # Turn off coefficients that are below the threshold
            coef_mask_new = torch.abs(coef) > self.threshold

            # Keep disabled coefficients disabled
            coef_mask_new = coef_mask_new & coef_mask

            return coef_mask_new

        else:
            return coef_mask


class PatientTrendAwareThresholder(Thresholder):
    def __init__(self, library, threshold_a=0.1, threshold_b=0.01, patience=500):
        super(PatientTrendAwareThresholder, self).__init__(library)
        self.device = "cpu"

        self.threshold_a = threshold_a
        self.threshold_b = threshold_b
        self.patience = patience

        # Store the indices at which the thresholds were last exceeded
        self.exceeded_threshold = torch.zeros((self.library.dim, self.library.L), dtype=int)
        self.exceeded_trend_threshold = torch.zeros((self.library.dim, self.library.L), dtype=int)

        self.step = 0

        self.last_coef = torch.zeros((self.library.dim, self.library.L))

    def to(self, device):
        self.device = device
        self.exceeded_threshold = self.exceeded_threshold.to(device)
        self.exceeded_trend_threshold = self.exceeded_trend_threshold.to(device)
        self.last_coef = self.last_coef.to(device)

        return self

    def __call__(self, coef, coef_mask):
        self.step += 1

        # Where the coefficients are above the threshold, set the exceeded threshold index to the current index
        self.exceeded_threshold[torch.abs(coef) > self.threshold_a] = self.step

        # Where the coefficient trends (i.e. the difference between the current and previous coefficient) are a
        self.exceeded_trend_threshold[torch.abs(coef - self.last_coef) > self.threshold_b] = self.step

        # Turn off coefficients for which the index is longer ago than the patience
        coef_mask_new = ((self.step - self.exceeded_threshold) < self.patience) | ((self.step - self.exceeded_t

        # Keep disabled coefficients disabled
        coef_mask_new = coef_mask_new & coef_mask

        self.last_coef = coef

        return coef_mask_new
```

Apply both thresholding algorithms during training

In [ ]:
```python
lib = Library(['z', 'dz'], 1, terms_torch)
thresholder = SequentialThresholder(lib, threshold=0.1, interval=200).to(device)
sindy_st = SINDy(lib, thresholder=thresholder).to(device)

optimizer = Adam(sindy_st.parameters(), lr=5e-2)
```

In [ ]:
```
batch_size = 1000

loss_history_st = {}

train_sindy(loss_history_st, sindy_st, optimizer,
            z_train.to(device), dz_train.to(device), ddz_train.to(device),
            z_val.to(device), dz_val.to(device), ddz_val.to(device),
            epochs=1000, refinement_after_epochs=800, l1_weight=1e-4 / batch_size, batch_size=batch_size, verbos
```

```
Train SINDy: 9.734e-16 | Train L1: 1.000e-07 | Val SINDy: 9.734e-19 | Val L1: 1.000e-10 | Equation: - 1.000 sin(
z): 100%|██████████| 1000/1000 [00:04<00:00, 215.20it/s]
```

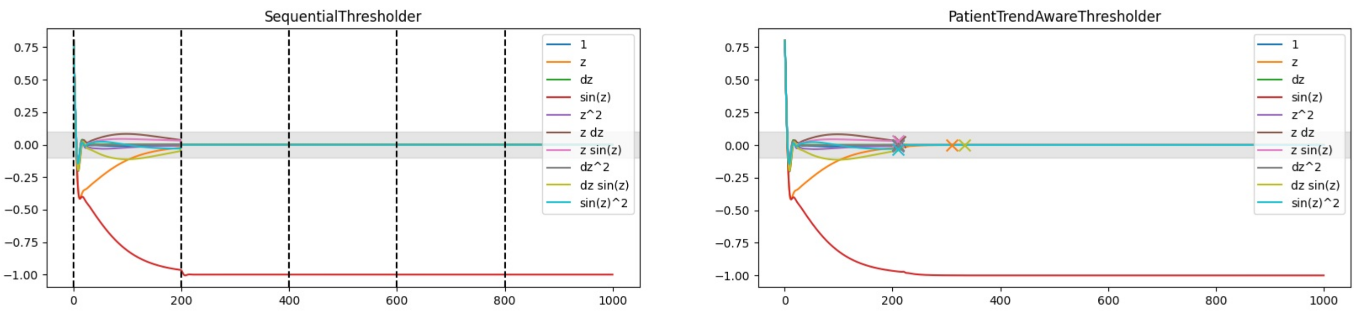
In [ ]:
```
lib = Library(['z', 'dz'], 1, terms_torch)
thresholder = PatientTrendAwareThresholder(lib, threshold_a=0.1, threshold_b=0.01, patience=200)
sindy_ptat = SINDy(lib, thresholder=thresholder).to(device)

optimizer = Adam(sindy_ptat.parameters(), lr=5e-2)
```

In [ ]:
```
loss_history_ptat = {}

train_sindy(loss_history_ptat, sindy_ptat, optimizer,
            z_train.to(device), dz_train.to(device), ddz_train.to(device),
            z_val.to(device), dz_val.to(device), ddz_val.to(device),
            epochs=1000, refinement_after_epochs=800, l1_weight=1e-4 / batch_size, batch_size=batch_size, verbos
```

```
Train SINDy: 9.734e-16 | Train L1: 1.000e-07 | Val SINDy: 9.734e-19 | Val L1: 1.000e-10 | Equation: - 1.000 sin(
z): 100%|██████████| 1000/1000 [00:05<00:00, 189.53it/s]
```

Inspect the coefficient histories with both thresholding algorithms

In [ ]:
```
# Plot the coefficient history
fig, axes = plt.subplots(1, 2, figsize=(20, 4))

for ax, loss_history, sindy in zip(axes, [loss_history_st, loss_history_ptat], [sindy_st, sindy_ptat]):
    x = np.array([c[0] for c in loss_history['coefficients']])
    Y = np.array([c[1] for c in loss_history['coefficients']])

    for i in range(Y.shape[2]):
        ax.plot(x, Y[:, 0, i], label=list(terms_np.keys())[i], color=f'C{i}')

    if isinstance(sindy.thresholder, PatientTrendAwareThresholder):
        threshold_epochs = sindy.thresholder.exceeded_threshold.detach().cpu().numpy().flatten()
        threshold_epochs += sindy.thresholder.patience

        for i in range(Y.shape[2]):
            if threshold_epochs[i] < x.shape[0]:
                ax.scatter([threshold_epochs[i]], [Y[threshold_epochs[i], 0, i]], color=f'C{i}', marker='x', s=

        ax.axhspan(-sindy.thresholder.threshold_a, sindy.thresholder.threshold_a, alpha=0.1, color='black')

    elif isinstance(sindy.thresholder, SequentialThresholder):
        threshold_epochs = np.arange(0, x.shape[0], sindy.thresholder.interval)
        for i in threshold_epochs:
            ax.axvline(i, color='black', linestyle='--')
        ax.axhspan(-sindy.thresholder.threshold, sindy.thresholder.threshold, alpha=0.1, color='black')

    ax.legend(loc='upper right')
    ax.set_title(type(sindy.thresholder).__name__)
```



Learned equations with both thresholding algorithms

In [ ]:
```
# Show the learned equation
for sindy in [sindy_st, sindy_ptat]:
    equation_str = " + ".join([f"{coef:.3f} {term}" for coef, mask, term in zip(
        sindy.coef.detach().cpu().numpy().flatten(),
        sindy.coef_mask.detach().cpu().numpy().flatten(),
        terms_np.keys()) if mask])
    print(f"Learned equation with {type(sindy.thresholder).__name__}: $\ddot z = {equation_str}$")
```

```
Learned equation with SequentialThresholder: $\ddot z = -1.000 sin(z)$
Learned equation with PatientTrendAwareThresholder: $\ddot z = -1.000 sin(z)$
```

1.4 Evaluation & Visualization

Test set

In [ ]:
```
_, _, _, z_test, dz_test, ddz_test, t_test = create_pendulum_data(
    z0_min=-np.pi,
    z0_max=np.pi,
    dz0_min=-2.1,
    dz0_max=2.1,
    coefficients=[target_coefficients[term] for term in terms_np],
    terms=[terms_np[term] for term in terms_np],
    T=T * 10,
    dt=DT,
    N=100
)

# Create tensors
z_test = torch.tensor(z_test).float().view(-1, 1)
dz_test = torch.tensor(dz_test).float().view(-1, 1)
ddz_test = torch.tensor(ddz_test).float().view(-1, 1)
```

ODE Prediction

In [ ]:
```
# Compute the test loss of sindy_sklearn and sindy
theta_test = np.concatenate([f(z_test, dz_test, "cpu") for f in terms_np.values()], axis=1)
ddz_hat_sklearn = sindy_sklearn.predict(theta_test)

ddz_hat = sindy_ptat.forward(z_test.to(device), dz_test.to(device)).detach().cpu().numpy()

print(f"{ddz_hat_sklearn.shape = }")
print(f"{ddz_hat.shape = }")
```

```
ddz_hat_sklearn.shape = (50000,)
ddz_hat.shape = (50000, 1)
```

ODE MSE losses

In [ ]:
```
# Compute the test loss
loss_fn = nn.MSELoss()

loss_sklearn = loss_fn(torch.tensor(ddz_hat_sklearn).float().view(-1, 1), ddz_test)
loss = loss_fn(torch.tensor(ddz_hat).float().view(-1, 1), ddz_test)

print(f"{loss_sklearn = :.3e}")
print(f"{loss = :.3e}")
```

```
loss_sklearn = 5.778e-08
loss = 1.018e-15
```

Resimulate the system with the learned equations and the initial conditions from the test set

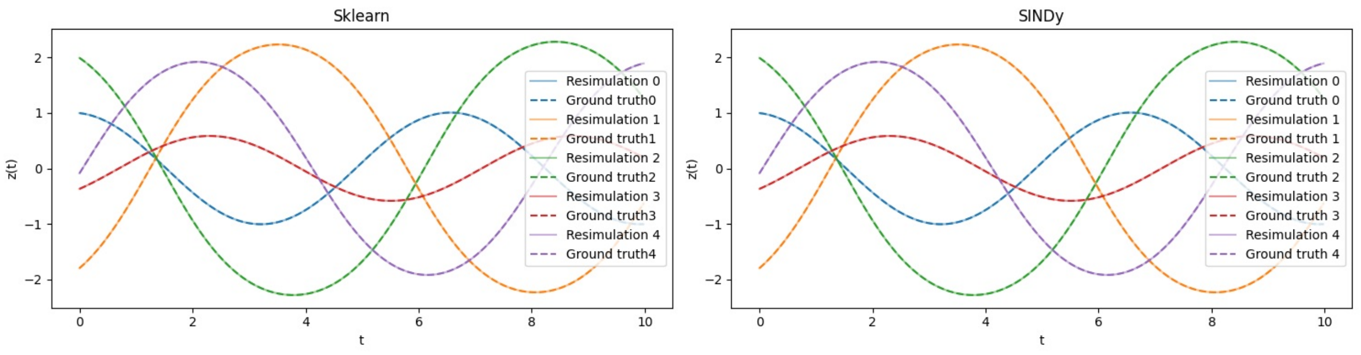
In [ ]:
```
z0 = z_test.reshape(100, T * 10)[:, 0].cpu().numpy().copy()
dz0 = dz_test.reshape(100, T * 10)[:, 0].cpu().numpy().copy()
```

In [ ]:
```
# Sklearn
t_sklearn, z_sklearn, dz_sklearn = simulate_pendulum(z0, dz0, sindy_sklearn.coef_, [terms_np[term] for term in t

# SINDy
t_sindy, z_sindy, dz_sindy = simulate_pendulum(z0, dz0, sindy_ptat.coef.detach().cpu().numpy().flatten(), [terms

print(f"{t_sklearn.shape = }")
print(f"{z_sklearn.shape = }")
print(f"{dz_sklearn.shape = }")

print(f"{t_sindy.shape = }")
print(f"{z_sindy.shape = }")
print(f"{dz_sindy.shape = }")
```

```
t_sklearn.shape = (500,)
z_sklearn.shape = (100, 500)
dz_sklearn.shape = (100, 500)
t_sindy.shape = (500,)
z_sindy.shape = (100, 500)
dz_sindy.shape = (100, 500)
```

In [ ]:
```
# Show a resimulated trajectory
```

```
fig, axes= plt.subplots(1, 2, figsize=(15, 4))

for i in range(5):
    axes[0].plot(t_sklearn, z_sklearn[i, :], label=f"Resimulation {i}", color=f'C{i}', alpha=0.5)
    axes[0].plot(t_sklearn, z_test.reshape(100, T * 10)[i, :], linestyle='--', label=f"Ground truth{i}", color=
    axes[0].set_title("Sklearn")

    axes[1].plot(t_sindy, z_sindy[i, :], label=f"Resimulation {i}", color=f'C{i}', alpha=0.5)
    axes[1].plot(t_sindy, z_test.reshape(100, T * 10)[i, :], linestyle='--', label=f"Ground truth {i}", color=f
    axes[1].set_title("SINDy")

    for ax in axes:
        ax.set_xlabel('t')
        ax.set_ylabel('z(t)')
        ax.legend()

fig.tight_layout()
```

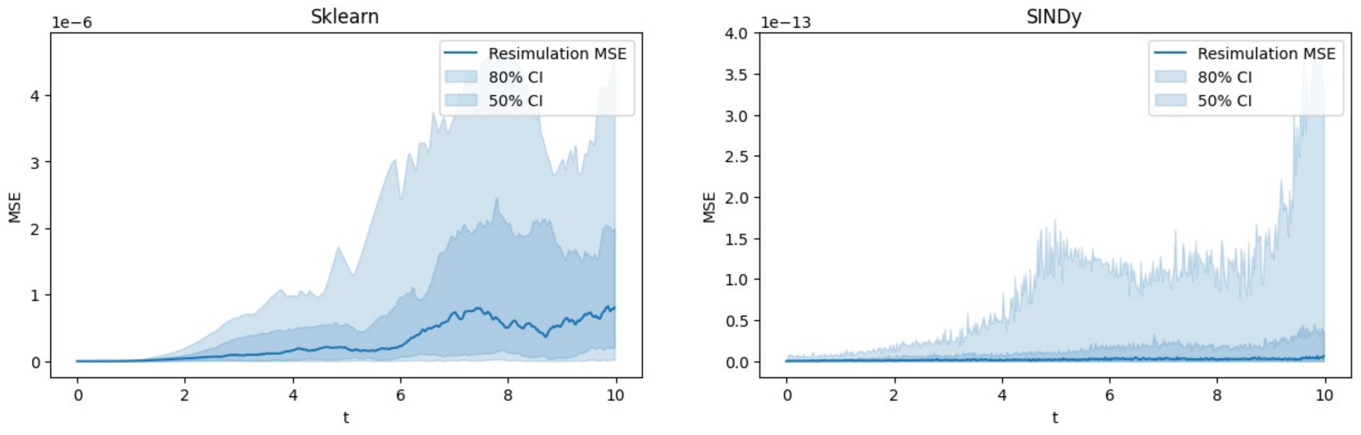


In [ ]:
```
resimulation_mse_z_median_sklearn = np.median((z_sklearn - z_test.reshape(100, T * 10).cpu().numpy())**2, axis=
resimulation_mse_z_quantiles_sklearn = np.quantile((z_sklearn - z_test.reshape(100, T * 10).cpu().numpy())**2,

resimulation_mse_z_median_sindy = np.median((z_sindy - z_test.reshape(100, T * 10).cpu().numpy())**2, axis=0)
resimulation_mse_z_quantiles_sindy = np.quantile((z_sindy - z_test.reshape(100, T * 10).cpu().numpy())**2, [0.1


print(f"{resimulation_mse_z_median_sklearn.shape = }")
print(f"{resimulation_mse_z_median_sindy.shape = }")
```

```
resimulation_mse_z_median_sklearn.shape = (500,)
resimulation_mse_z_median_sindy.shape = (500,)
```

In [ ]:
```
# Show the resimulation error over time
fig, axes = plt.subplots(1, 2, figsize=(15, 4))

axes[0].plot(t_test, resimulation_mse_z_median_sklearn, label='Resimulation MSE', color='C0')
axes[0].fill_between(t_test, resimulation_mse_z_quantiles_sklearn[0], resimulation_mse_z_quantiles_sklearn[-1],
axes[0].fill_between(t_test, resimulation_mse_z_quantiles_sklearn[1], resimulation_mse_z_quantiles_sklearn[-2],
axes[0].set_xlabel('t')
axes[0].set_ylabel('MSE')
axes[0].set_title('Sklearn')
axes[0].legend()

axes[1].plot(t_test, resimulation_mse_z_median_sindy, label='Resimulation MSE', color='C0')
axes[1].fill_between(t_test, resimulation_mse_z_quantiles_sindy[0], resimulation_mse_z_quantiles_sindy[-1], alp
axes[1].fill_between(t_test, resimulation_mse_z_quantiles_sindy[1], resimulation_mse_z_quantiles_sindy[-2], alp
axes[1].set_xlabel('t')
axes[1].set_ylabel('MSE')
axes[1].set_title('SINDy')
axes[1].legend()
```

Out[ ]:
```
<matplotlib.legend.Legend at 0x7f508b4e3110>
```



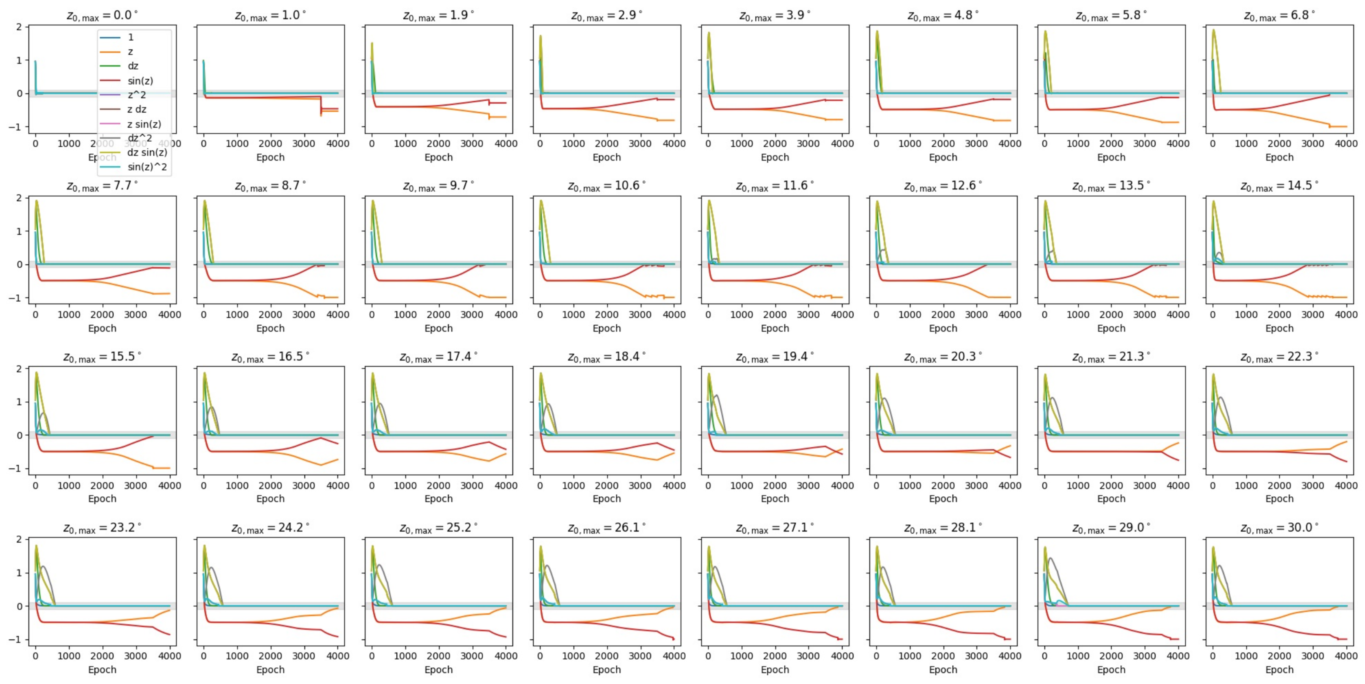
1.5 Small Angle Approximation

Scan through a range of smaller angles

In [ ]:
```
z0_small_list = np.linspace(0, np.pi / 6, 32)
dz0_small_list = z0_small_list / 10
```

In [ ]:
```
batch_size = 1000
if TRAIN_SMALL_ANGLE_APPROXIMATION:
    loss_histories = []
    coefficients = []

    for z0_small, dz0_small in zip(z0_small_list, dz0_small_list):
        # Create a new dataset with smaller initial conditions
        _, _, _, z_train_small, dz_train_small, ddz_train_small, t_train_small = create_pendulum_data(
            z0_min=-z0_small,
            z0_max=z0_small,
            dz0_min=0,
            dz0_max=0,
            coefficients=[target_coefficients[term] for term in terms_np],
            terms=[terms_np[term] for term in terms_np],
            T=T,
            dt=DT,
            N=100,
            reject_invalid=False
        )

        _, _, _, z_val_small, dz_val_small, ddz_val_small, t_val_small = create_pendulum_data(
            z0_min=-z0_small,
            z0_max=z0_small,
            dz0_min=0,
            dz0_max=0,
            coefficients=[target_coefficients[term] for term in terms_np],
            terms=[terms_np[term] for term in terms_np],
            T=T,
            dt=DT,
            N=20,
            reject_invalid=False
        )

        # Create tensors
        z_train_small = torch.tensor(z_train_small).float().view(-1, 1)
        dz_train_small = torch.tensor(dz_train_small).float().view(-1, 1)
        ddz_train_small = torch.tensor(ddz_train_small).float().view(-1, 1)

        # Shuffle the training data
        idx = torch.randperm(z_train_small.shape[0])
        z_train_small = z_train_small[idx]
        dz_train_small = dz_train_small[idx]
        ddz_train_small = ddz_train_small[idx]

        z_val_small = torch.tensor(z_val_small).float().view(-1, 1)
        dz_val_small = torch.tensor(dz_val_small).float().view(-1, 1)
        ddz_val_small = torch.tensor(ddz_val_small).float().view(-1, 1)

        # Fit SINDy on the small dataset
        lib = Library(['z', 'dz'], 1, terms_torch)
        thresholder = PatientTrendAwareThresholder(lib, threshold_a=0.1, threshold_b=0.01, patience=200)
        sindy_small = SINDy(lib, thresholder).to(device)

        optimizer = Adam(sindy_small.parameters(), lr=1e-2)

        loss_history = {}

        train_sindy(loss_history, sindy_small, optimizer,
                    z_train_small.to(device), dz_train_small.to(device), ddz_train_small.to(device),
                    z_val_small.to(device), dz_val_small.to(device), ddz_val_small.to(device),
                    epochs=4000, refinement_after_epochs=3500,
                    l1_weight=1e-4 / batch_size, batch_size=batch_size, verbose=True)

        loss_histories.append(loss_history)
        coefficients.append(sindy_small.coef.detach().cpu().numpy().flatten())

        # Save the loss history and coefficients
        with open('small_angle_loss_histories.pkl', 'wb') as f:
            pickle.dump(loss_histories, f)

        with open('small_angle_coefficients.pkl', 'wb') as f:
            pickle.dump(coefficients, f)

else:
    with open('small_angle_loss_histories.pkl', 'rb') as f:
        loss_histories = pickle.load(f)

    with open('small_angle_coefficients.pkl', 'rb') as f:
        coefficients = pickle.load(f)
```

In [ ]:
```
# Plot the coefficient histories
fig, axes = plt.subplots(4, 8, figsize=(20, 10), sharey=True)

for i, ax in enumerate(axes.flatten()):
    for j in range(Y.shape[2]):
        x = np.array([c[0] for c in loss_histories[i]['coefficients']])
        Y = np.array([c[1] for c in loss_histories[i]['coefficients']])
        ax.plot(x, Y[:, 0, j], label=list(terms_np.keys())[j])
        ax.set_title(f"$z_{{0, \\text{{max}}}} = {z0_small_list[i] / np.pi * 180:.1f}^\circ$")
        ax.set_xlabel('Epoch')
    ax.axhspan(-sindy.thresholder.threshold_a, sindy.thresholder.threshold_a, alpha=0.1, color='black')

axes[0, 0].legend(loc='upper right')

fig.tight_layout()
```

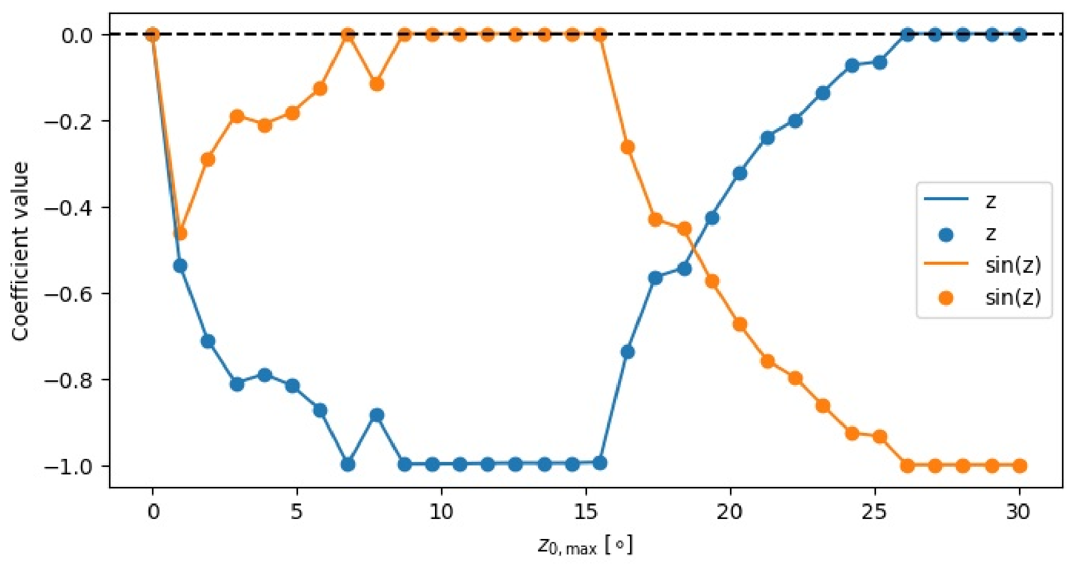


In [ ]:
```
# Extract a list of the z and sin(z) coefficient values for each initial condition
print(terms_np.keys())
z_coefs = np.array([coef[1] for coef in coefficients])
sinz_coefs = np.array([coef[3] for coef in coefficients])
```

```
dict_keys(['1', 'z', 'dz', 'sin(z)', 'z^2', 'z dz', 'z sin(z)', 'dz^2', 'dz sin(z)', 'sin(z)^2'])
```

In [ ]:
```
fig, ax = plt.subplots(1, 1, figsize=(8, 4))

ax.plot(z0_small_list / np.pi * 180, z_coefs, label='z')
ax.scatter(z0_small_list / np.pi * 180, z_coefs, label='z')
ax.plot(z0_small_list / np.pi * 180, sinz_coefs, label='sin(z)')
ax.scatter(z0_small_list / np.pi * 180, sinz_coefs, label='sin(z)')

ax.axhline(0, color='black', linestyle='--')

ax.set_xlabel('$z_{0, \\text{max}} \; [\circ]$')
ax.set_ylabel('Coefficient value')
ax.legend()
```

Out[ ]:
```
<matplotlib.legend.Legend at 0x7f507ec4bc50>
```



SINDy-Autoencoder

2.1 Artificial Embedding

```
In [ ]: def embed_cartesian(z, dz, ddz, t):
            """
            Artificially embed the point mass into 2d cartesian coordinates
            """
            # Artificially embed the point mass into 2d cartesian coordinates
            x = np.stack([np.sin(z), -np.cos(z)]).transpose(1,2,0)
            dx = np.stack([np.cos(z) * dz, np.sin(z) * dz]).transpose(1,2,0)
            ddx = np.stack([-np.sin(z) * dz**2 + np.cos(z) * ddz, np.cos(z) * dz**2 + np.sin(z) * ddz]).transpose(1,2,0

            return x, dx, ddx
```

2.2 Hyperparameter Optimization

```
In [ ]: class Autoencoder(nn.Module):
            def __init__(self, input_dim: int, encoder_sizes: list[int], decoder_sizes: list[int]):
                super(Autoencoder, self).__init__()

                self.sindy = sindy

                self.encoder = nn.ModuleList()
                self.decoder = nn.ModuleList()

                encoder_transforms = [input_dim] + encoder_sizes + [sindy.library.dim]
                decoder_transforms = [sindy.library.dim] + decoder_sizes + [input_dim]

                # Encoder
                for i in range(len(encoder_transforms) - 2):
                    self.encoder.append(nn.Linear(encoder_transforms[i], encoder_transforms[i + 1], bias=False))
                    self.encoder.append(nn.Sigmoid())

                self.encoder.append(nn.Linear(encoder_transforms[-2], encoder_transforms[-1], bias=False))

                # Decoder
                for i in range(len(decoder_transforms) - 2):
                    self.decoder.append(nn.Linear(decoder_transforms[i], decoder_transforms[i + 1], bias=False))
                    self.decoder.append(nn.Sigmoid())
```

```
        self.decoder.append(nn.Linear(decoder_transforms[-2], decoder_transforms[-1], bias=False))

        # Xavier initialization and set bias to zero
        for layer in self.encoder:
            if isinstance(layer, nn.Linear):
                torch.nn.init.xavier_uniform_(layer.weight)

        for layer in self.decoder:
            if isinstance(layer, nn.Linear):
                torch.nn.init.xavier_uniform_(layer.weight)

    def encode(self, x):
        for layer in self.encoder:
            x = layer(x)
        return x

    def decode(self, x, dx=None, ddx=None):
        for layer in self.decoder:
            x = layer(x)

        return x

    def forward(self, x):
        # Encode the input
        z = self.encode(x)
        x_hat = self.decode(z)

        return z, x_hat
```

In [ ]:
```
def train_autoencoder(loss_history, autoencoder, optimizer, x_train, x_val, epochs, batch_size=32, verbose=True
    loss_fn = nn.MSELoss()

    loss_history['train_autoencoder'] = []
    loss_history['val_autoencoder'] = []

    pbar = tqdm(range(epochs), disable=not verbose)

    for epoch in pbar:
        # Training
        autoencoder.train()

        for i in range(0, x_train.shape[0], batch_size):
            # Backpropagation
            optimizer.zero_grad()

            x_batch = x_train[i : i + batch_size]

            _, x_hat_batch = autoencoder.forward(x_batch)

            autoencoder_loss = loss_fn(x_hat_batch, x_batch)

            autoencoder_loss.backward()
            optimizer.step()

            loss_history['train_autoencoder'].append([epoch + i / x_train.shape[0], autoencoder_loss.item() / x_

        # Validation
        autoencoder.eval()
        with torch.no_grad():
            _, x_hat_val = autoencoder.forward(x_val)

            autoencoder_loss = loss_fn(x_hat_val, x_val)

            loss_history['val_autoencoder'].append([epoch, autoencoder_loss.item() / x_val.shape[0]])

        if verbose:
            pbar.set_description(f"Train Autoencoder: {autoencoder_loss.item():.3e} | Val Autoencoder: {loss_his
```

In [ ]:
```
# Create the data
x_optimization_train, _, _, z_optimization_train, _, _, t_optimization_train = create_pendulum_data(
    z0_min=-np.pi,
    z0_max=np.pi,
    dz0_min=-2.1,
    dz0_max=2.1,
    coefficients=[target_coefficients[term] for term in terms_np],
    terms=[terms_np[term] for term in terms_np],
    T=T,
    dt=DT,
    N=100,
    embedding=embed_cartesian,
)

x_optimization_val, _, _, z_optimization_val, _, _, t_optimization_val = create_pendulum_data(
    z0_min=-np.pi,
    z0_max=np.pi,
    dz0_min=-2.1,
    dz0_max=2.1,
    coefficients=[target_coefficients[term] for term in terms_np],
    terms=[terms_np[term] for term in terms_np],
    T=T,
    dt=DT,
    N=20,
    embedding=embed_cartesian,
)

# Create tensors
x_optimization_train = torch.tensor(x_optimization_train).float().view(-1, 2)

# Shuffle the training data
idx = torch.randperm(x_optimization_train.shape[0])
x_optimization_train = x_optimization_train[idx]

x_optimization_val = torch.tensor(x_optimization_val).float().view(-1, 2)

print(f"{x_optimization_train.shape = }")
print(f"{x_optimization_val.shape = }")
```

```
x_optimization_train.shape = torch.Size([5000, 2])
x_optimization_val.shape = torch.Size([1000, 2])
```

Test a variety of layer sizes

In [ ]:
```
# Define different hyperparameters
encoder_sizes_list = [[], [2], [2, 2], [2, 2, 2], [4], [4, 4], [4, 4, 4], [16], [16, 16], [16, 16, 16], [32]*5,
```

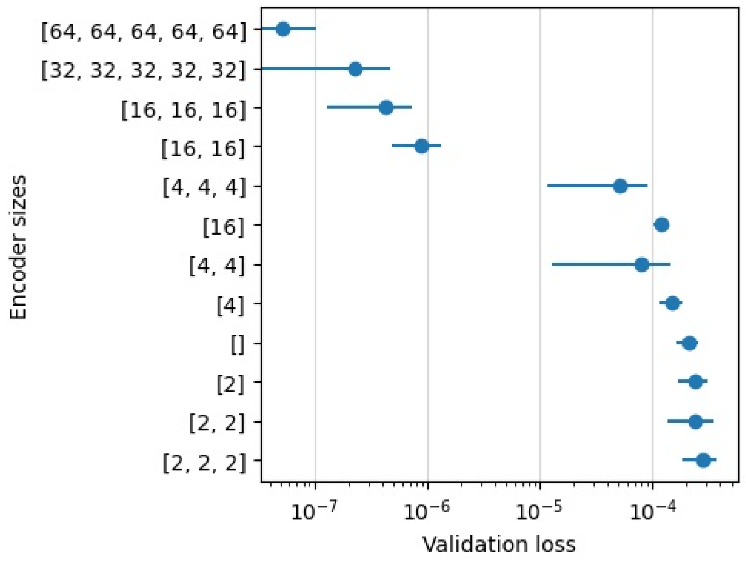
In [ ]:
```
if OPTIMIZE_AUTOENCODER_HYPERPARAMETERS:
    results = [[] for _ in encoder_sizes_list]

    for i in tqdm(range(N_REPEAT)):
        for j, encoder_size in enumerate(encoder_sizes_list):
            # Create the autoencoder
            autoencoder = Autoencoder(
                input_dim=2,
                encoder_sizes=encoder_size,
                decoder_sizes=list(reversed(encoder_size))
            ).to(device)

            # Create the optimizer
            optimizer = Adam(autoencoder.parameters(), lr=1e-3)

            # Train the autoencoder
            loss_history = {}

            train_autoencoder(loss_history, autoencoder, optimizer, x_optimization_train.to(device), x_optimiza

            # Store the results
            results[j].append(loss_history)

    mean_losses = [np.mean([np.array(loss_history['val_autoencoder'])[-1, 1] for loss_history in results[i]]) fo
    std_losses = [np.std([np.array(loss_history['val_autoencoder'])[-1, 1] for loss_history in results[i]]) for

    # Create a dataframe with the results
    df = pd.DataFrame({'encoder_sizes': encoder_sizes_list, 'mean_losses': mean_losses, 'std_losses': std_losses

    # Sort the dataframe by the minimum upper bound
    df = df.sort_values(by='upper_bound')

    # Store the dataframe
    df.to_csv('autoencoder_hyperparameters.csv')

else:
    # Load the dataframe
    df = pd.read_csv('autoencoder_hyperparameters.csv', index_col=0)
    df['encoder_sizes'] = df['encoder_sizes'].apply(lambda x: eval(x))

df
```

Out[ ]:

	encoder_sizes	mean_losses	std_losses	upper_bound
11	[64, 64, 64, 64, 64]	5.293863e-08	5.211123e-08	1.050499e-07
10	[32, 32, 32, 32, 32]	2.277067e-07	2.459541e-07	4.736608e-07
9	[16, 16, 16]	4.321564e-07	3.036445e-07	7.358009e-07
8	[16, 16]	8.957387e-07	4.146153e-07	1.310354e-06
6	[4, 4, 4]	5.098624e-05	3.927191e-05	9.025815e-05
7	[16]	1.204230e-04	1.992787e-05	1.403508e-04
5	[4, 4]	7.827353e-05	6.541364e-05	1.436872e-04
4	[4]	1.496088e-04	3.336963e-05	1.829784e-04
0	[]	2.084083e-04	4.449524e-05	2.529036e-04
1	[2]	2.350184e-04	6.871960e-05	3.037380e-04
2	[2, 2]	2.383337e-04	1.048691e-04	3.432028e-04
3	[2, 2, 2]	2.781500e-04	9.539238e-05	3.735424e-04

In [ ]:
```python
# Plot the mean and std for each configuration
fig, ax = plt.subplots(figsize=(4, 4))

ax.errorbar(df['mean_losses'][::-1], np.arange(len(df)), xerr=df['std_losses'][::-1], fmt='o')
ax.set_xscale('log')
ax.grid(axis='x', alpha=0.5)
ax.set_yticks(np.arange(len(df)))
ax.set_yticklabels(df['encoder_sizes'][::-1].apply(lambda x: str(x)))
ax.set_xlabel('Validation loss')
ax.set_ylabel('Encoder sizes')
```

Out[ ]:
```
Text(0, 0.5, 'Encoder sizes')
```



2.3 Propagation of Time Derivatives

Derivative Layers

In [ ]:
```python
# Differentiable layers
class SigmoidDerivatives(nn.Module):
    def __init__(self):
        super(SigmoidDerivatives, self).__init__()

    def forward(self, x: torch.Tensor, dx: torch.Tensor | None = None, ddx: torch.Tensor | None = None) -> tupl
        z = torch.sigmoid(x)

        if dx is not None:
            sigmoid_derivative = z * (1 - z)
            dz = sigmoid_derivative * dx
        else:
            dz = None

        if ddx is not None:
            sigmoid_derivative_2 = sigmoid_derivative * (1 - 2 * z)
            ddz = sigmoid_derivative_2 * dx**2 + sigmoid_derivative * ddx
        else:
            ddz = None

        return z, dz, ddz

class LinearDerivatives(nn.Linear):
    def __init__(self, *args, **kwargs):
        super(LinearDerivatives, self).__init__(*args, **kwargs)

    def forward(self, x: torch.Tensor, dx: torch.Tensor | None = None, ddx: torch.Tensor | None = None) -> tupl
        z = F.linear(x, self.weight, self.bias)

        if dx is not None:
            dz = F.linear(dx, self.weight)
        else:
            dz = None

        if ddx is not None:
            ddz = F.linear(ddx, self.weight)
        else:
            ddz = None

        return z, dz, ddz
```

2.4 Implementation

SINDy-Autoencoder

In [ ]:
```python
class SINDyAutoencoder(nn.Module):
    def __init__(self, sindy: SINDy, input_dim: int, encoder_sizes: list[int], decoder_sizes: list[int]):
        super(SINDyAutoencoder, self).__init__()

        self.sindy = sindy

        self.encoder = nn.ModuleList()
        self.decoder = nn.ModuleList()

        encoder_transforms = [input_dim] + encoder_sizes + [sindy.library.dim]
        decoder_transforms = [sindy.library.dim] + decoder_sizes + [input_dim]

        # Encoder
        for i in range(len(encoder_transforms) - 2):
            self.encoder.append(LinearDerivatives(encoder_transforms[i], encoder_transforms[i + 1], bias=False)
            self.encoder.append(SigmoidDerivatives())

        self.encoder.append(LinearDerivatives(encoder_transforms[-2], encoder_transforms[-1], bias=False))

        # Decoder
        for i in range(len(decoder_transforms) - 2):
            self.decoder.append(LinearDerivatives(decoder_transforms[i], decoder_transforms[i + 1], bias=False)
            self.decoder.append(SigmoidDerivatives())

        self.decoder.append(LinearDerivatives(decoder_transforms[-2], decoder_transforms[-1], bias=False))

        # Xavier initialization and set bias to zero
        for layer in self.encoder:
            if isinstance(layer, LinearDerivatives):
                torch.nn.init.xavier_uniform_(layer.weight)

        for layer in self.decoder:
            if isinstance(layer, LinearDerivatives):
                torch.nn.init.xavier_uniform_(layer.weight)

    def encode(self, x, dx=None, ddx=None):
        for layer in self.encoder:
            x, dx, ddx = layer(x, dx, ddx)

        return x, dx, ddx

    def decode(self, x, dx=None, ddx=None):
        for layer in self.decoder:
            x, dx, ddx = layer(x, dx, ddx)

        return x, dx, ddx

    def to(self, device):
        super(SINDyAutoencoder, self).to(device)
        self.device = device
        self.encoder = self.encoder.to(device)
        self.decoder = self.decoder.to(device)
        self.sindy = self.sindy.to(device)
        return self

    def forward(self, x, dx, ddx):
        # Encode the input
        z, dz, ddz_lhs = self.encode(x, dx, ddx)

        # Compute the SINDy coefficients
        ddz_rhs = self.sindy(z, dz)

        # Decode the rhs
        # x_hat, _, _ = self.decode(z)
        x_hat, _, ddx_hat_rhs = self.decode(z, dz, ddz_rhs)

        return x_hat, ddx_hat_rhs, ddz_lhs, ddz_rhs
```

In [ ]:
```
T_dev = 100
```

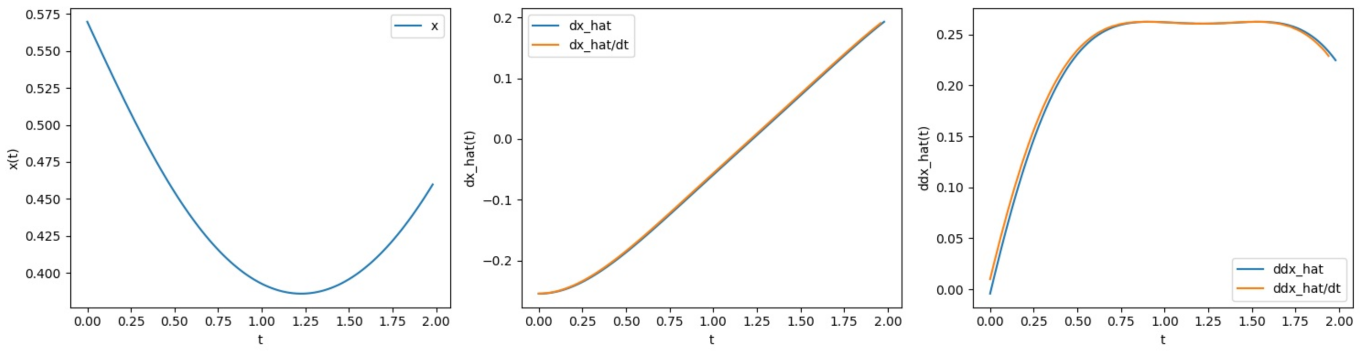
In [ ]:
```
x_dev, dx_dev, ddx_dev, z_dev, dz_dev, ddz_dev, t_dev = create_pendulum_data(
    z0_min=-np.pi,
    z0_max=np.pi,
    dz0_min=-2.1,
    dz0_max=2.1,
    coefficients=[target_coefficients[term] for term in terms_np],
    terms=[terms_np[term] for term in terms_np],
    T=T_dev,
    dt=DT,
    N=1,
    embedding=embed_cartesian,
)
```

In [ ]:
```
# Create tensors
x_dev = torch.tensor(x_dev).float().view(-1, 2)
dx_dev = torch.tensor(dx_dev).float().view(-1, 2)
ddx_dev = torch.tensor(ddx_dev).float().view(-1, 2)

z_dev = torch.tensor(z_dev).float().view(-1, 1)
dz_dev = torch.tensor(dz_dev).float().view(-1, 1)
ddz_dev = torch.tensor(ddz_dev).float().view(-1, 1)


print(f"{x_dev.shape = }")
print(f"{dx_dev.shape = }")
print(f"{ddx_dev.shape = }")
```

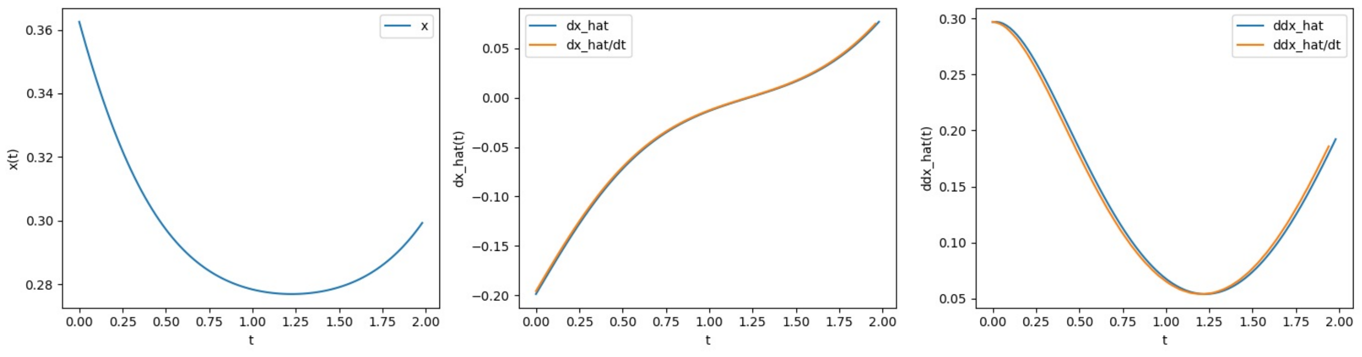
```
x_dev.shape = torch.Size([100, 2])
dx_dev.shape = torch.Size([100, 2])
ddx_dev.shape = torch.Size([100, 2])
```

Verify Derivative Layers

In [ ]:
```
linear_derivative = LinearDerivatives(2, 2)

x_hat, dx_hat, ddx_hat = linear_derivative(x_dev[:T_dev], dx_dev[:T_dev], ddx_dev[:T_dev])
x_hat_diff = np.diff(x_hat[:, 0].detach().cpu().numpy()) / np.diff(t_dev[:T_dev])
x_hat_diff_diff = np.diff(x_hat_diff) / np.diff(t_dev[:T_dev-1])


# Plot the results
fig, ax = plt.subplots(1, 3, figsize=(15, 4))

ax[0].plot(t_dev[:T_dev], x_hat[:T_dev, 0].detach().cpu().numpy(), label='x')
ax[0].set_xlabel('t')
ax[0].set_ylabel('x(t)')
ax[0].legend()

ax[1].plot(t_dev[:T_dev], dx_hat[:, 0].detach().cpu().numpy(), label='dx_hat')
ax[1].plot(t_dev[:T_dev-1], x_hat_diff, label='dx_hat/dt')
ax[1].set_xlabel('t')
ax[1].set_ylabel('dx_hat(t)')
ax[1].legend()


ax[2].plot(t_dev[:T_dev], ddx_hat[:, 0].detach().cpu().numpy(), label='ddx_hat')
ax[2].plot(t_dev[:T_dev-2], x_hat_diff_diff, label='ddx_hat/dt')
ax[2].set_xlabel('t')
ax[2].set_ylabel('ddx_hat(t)')
ax[2].legend()

fig.tight_layout()
```



In [ ]:
```
sigmoid_derivative = SigmoidDerivatives()

x_hat, dx_hat, ddx_hat = sigmoid_derivative(x_dev[:T_dev], dx_dev[:T_dev], ddx_dev[:T_dev])
x_hat_diff = np.diff(x_hat[:, 0].detach().cpu().numpy()) / np.diff(t_dev[:T_dev])
x_hat_diff_diff = np.diff(x_hat_diff) / np.diff(t_dev[:T_dev-1])


# Plot the results
fig, ax = plt.subplots(1, 3, figsize=(15, 4))

ax[0].plot(t_dev[:T_dev], x_hat[:T_dev, 0].detach().cpu().numpy(), label='x')
ax[0].set_xlabel('t')
ax[0].set_ylabel('x(t)')
ax[0].legend()

ax[1].plot(t_dev[:T_dev], dx_hat[:, 0].detach().cpu().numpy(), label='dx_hat')
ax[1].plot(t_dev[:T_dev-1], x_hat_diff, label='dx_hat/dt')
ax[1].set_xlabel('t')
ax[1].set_ylabel('dx_hat(t)')
ax[1].legend()


ax[2].plot(t_dev[:T_dev], ddx_hat[:, 0].detach().cpu().numpy(), label='ddx_hat')
ax[2].plot(t_dev[:T_dev-2], x_hat_diff_diff, label='ddx_hat/dt')
ax[2].set_xlabel('t')
ax[2].set_ylabel('ddx_hat(t)')
ax[2].legend()

fig.tight_layout()
```



In [ ]:
```
sindy_autoencoder = SINDyAutoencoder(sindy, input_dim=2, encoder_sizes=[32] * 0, decoder_sizes=[32] * 0).to(dev

x_hat, ddx_hat_rhs, ddz_lhs, ddz_rhs = sindy_autoencoder.forward(x_dev[:5].to(device), dx_dev[:5].to(device), d

print(f"{x_hat.shape = }")
print(f"{ddx_hat_rhs.shape = }")
print(f"{ddz_lhs.shape = }")
print(f"{ddz_rhs.shape = }")
```

```
x_hat.shape = torch.Size([5, 2])
ddx_hat_rhs.shape = torch.Size([5, 2])
ddz_lhs.shape = torch.Size([5, 1])
ddz_rhs.shape = torch.Size([5, 1])
```

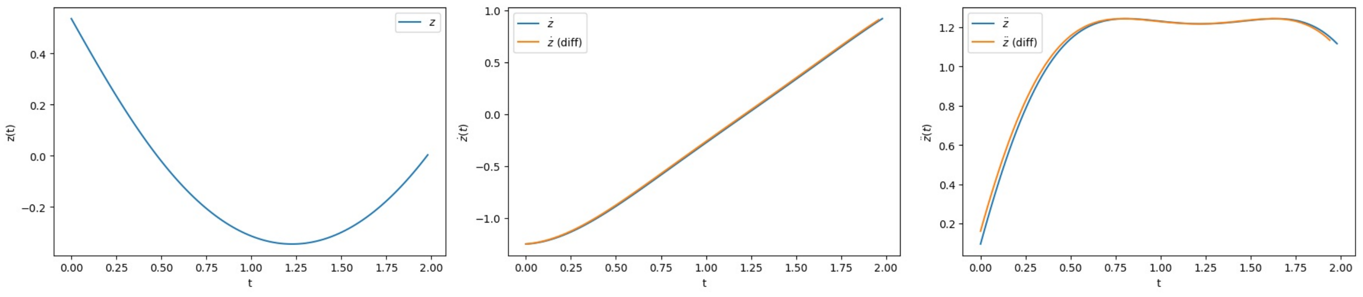
In [ ]:
```
z, dz, ddz = sindy_autoencoder.encode(x_dev[:T_dev].to(device), dx_dev[:T_dev].to(device), ddx_dev[:T_dev].to(d

print(f"{z.shape = }")
print(f"{dz.shape = }")
print(f"{ddz.shape = }")
```

```
z.shape = torch.Size([100, 1])
dz.shape = torch.Size([100, 1])
ddz.shape = torch.Size([100, 1])
```

In [ ]:
```
z_diff = np.diff(z.detach().cpu().numpy()[:, 0]) / np.diff(t_dev[:T_dev])
```

```python
z_diff_diff = np.diff(z_diff) / np.diff(t_dev[:T_dev - 1])
```

In [ ]:
```python
# Plot the autoencoder output
fig, ax = plt.subplots(1, 3, figsize=(18, 4))

ax[0].plot(t_dev[:T_dev], z.detach().cpu().numpy()[:, 0], label='$z$')
ax[0].set_xlabel('t')
ax[0].set_ylabel('z(t)')
ax[0].legend()

ax[1].plot(t_dev[:T_dev], dz.detach().cpu().numpy()[:, 0], label='$\dot z$')
ax[1].plot(t_dev[:T_dev - 1], z_diff, label='$\dot z$ (diff)')
ax[1].set_xlabel('t')
ax[1].set_ylabel('$\dot z(t)$')
ax[1].legend()

ax[2].plot(t_dev[:T_dev], ddz.detach().cpu().numpy()[:, 0], label='$\ddot z$')
ax[2].plot(t_dev[:T_dev - 2], z_diff_diff, label='$\ddot z$ (diff)')
ax[2].set_xlabel('t')
ax[2].set_ylabel('$\ddot z(t)$')
ax[2].legend()

fig.tight_layout()
```



Evaluation Methods

In [ ]:
```python
def compute_FVU(x, x_hat):
    return torch.sum((x - x_hat)**2) / torch.sum((x - torch.mean(x))**2)
```

2.6 & 2.7 Training and Evaluation

In [ ]:
```python
def train_sindy_autoencoder(loss_history, sindy_autoencoder: SINDyAutoencoder, optimizer, x_train, dx_train, dd
    loss_fn = nn.MSELoss()

    loss_history['train_x'] = []
    loss_history['train_l1'] = []
    loss_history['train_ddx'] = []
    loss_history['train_ddz'] = []
    loss_history['val_x'] = []
    loss_history['val_l1'] = []
    loss_history['val_ddx'] = []
    loss_history['val_ddz'] = []
    loss_history['active_terms'] = []
    loss_history['coefficients'] = []

    pbar = tqdm(range(epochs), disable=not verbose)

    for epoch in pbar:
        # Training
        sindy_autoencoder.sindy.train()

        epoch_train_reconstruction_loss = 0
        epoch_train_ddx_loss = 0
        epoch_train_ddz_loss = 0
        epoch_train_l1_loss = 0

        for i in range(0, z_train.shape[0], batch_size):
            # Backpropagation
            optimizer.zero_grad()

            x_batch = x_train[i : i + batch_size]
            dx_batch = dx_train[i : i + batch_size]
            ddx_batch = ddx_train[i : i + batch_size]

            x_hat, ddx_hat_rhs, ddz_lhs, ddz_rhs = sindy_autoencoder.forward(x_batch, dx_batch, ddx_batch)

            reconstruction_loss = loss_fn(x_hat, x_batch)
            ddz_loss = loss_fn(ddz_lhs, ddz_rhs)
            ddx_loss = loss_fn(ddx_hat_rhs, ddx_batch)
            if epoch >= refinement_after_epochs:
                l1_loss = torch.Tensor([0]).to(device)
            else:
                l1_loss = torch.norm(sindy_autoencoder.sindy.coef * sindy_autoencoder.sindy.coef_mask, p=1)

            loss = (reconstruction_loss + ddx_weight * ddx_loss + ddz_weight * ddz_loss + l1_weight * l1_loss *

            loss.backward()
            optimizer.step()

            epoch_train_reconstruction_loss += reconstruction_loss.item()
            epoch_train_ddx_loss += ddx_loss.item()
            epoch_train_ddz_loss += ddz_loss.item()
            epoch_train_l1_loss += l1_loss.item()

            loss_history['train_x'].append([epoch + i / z_train.shape[0], reconstruction_loss.item()])
            loss_history['train_ddx'].append([epoch + i / z_train.shape[0], ddx_loss.item()])
            loss_history['train_ddz'].append([epoch + i / z_train.shape[0], ddz_loss.item()])
            loss_history['train_l1'].append([epoch + i / z_train.shape[0], l1_loss.item()])

        # Average the losses
        epoch_train_reconstruction_loss /= z_train.shape[0]
        epoch_train_ddx_loss /= z_train.shape[0]
        epoch_train_ddz_loss /= z_train.shape[0]
        epoch_train_l1_loss /= z_train.shape[0]

        # Thresholding
        sindy_autoencoder.sindy.coef_mask.data = sindy_autoencoder.sindy.thresholder(sindy_autoencoder.sindy.co
        sindy_autoencoder.sindy.coef.data = sindy_autoencoder.sindy.coef.data * sindy_autoencoder.sindy.coef_ma
        loss_history['active_terms'].append([epoch + i / z_train.shape[0], torch.sum(sindy_autoencoder.sindy.co

        # Store the coefficients
        loss_history['coefficients'].append([epoch + i / z_train.shape[0], sindy_autoencoder.sindy.coef.detach(

        # Validation
        epoch_val_reconstruction_loss = 0
        epoch_val_ddx_loss = 0
        epoch_val_ddz_loss = 0
        epoch_val_l1_loss = 0

        sindy_autoencoder.sindy.eval()
        with torch.no_grad():
            x_hat, ddx_hat_rhs, ddz_lhs, ddz_rhs = sindy_autoencoder.forward(x_val, dx_val, ddx_val)

            reconstruction_loss = loss_fn(x_hat, x_val)
            ddz_loss = loss_fn(ddz_lhs, ddz_rhs)
            ddx_loss = loss_fn(ddx_hat_rhs, ddx_val)

            if epoch >= refinement_after_epochs:
                l1_loss = torch.Tensor([0]).to(device)
            else:
                l1_loss = torch.norm(sindy_autoencoder.sindy.coef * sindy_autoencoder.sindy.coef_mask, p=1)

            loss = (reconstruction_loss + ddx_weight * ddx_loss + ddz_weight * ddz_loss + l1_weight * l1_loss *

            epoch_val_reconstruction_loss += reconstruction_loss.item()
            epoch_val_ddx_loss += ddx_loss.item()
            epoch_val_ddz_loss += ddz_loss.item()
            epoch_val_l1_loss += l1_loss.item()

            loss_history['val_x'].append([epoch, reconstruction_loss.item()])
            loss_history['val_ddx'].append([epoch, ddx_loss.item()])
            loss_history['val_ddz'].append([epoch, ddz_loss.item()])
            loss_history['val_l1'].append([epoch, l1_loss.item()])

        # Average the losses
        epoch_val_reconstruction_loss /= z_val.shape[0]
        epoch_val_ddx_loss /= z_val.shape[0]
        epoch_val_ddz_loss /= z_val.shape[0]
        epoch_val_l1_loss /= z_val.shape[0]

        if verbose:
            coefs = sindy_autoencoder.sindy.coef[0].detach().cpu().numpy()
            coef_mask = sindy_autoencoder.sindy.coef_mask[0].detach().cpu().numpy()
            terms = sindy_autoencoder.sindy.library.terms
            equation_string = " ".join([f"{'+ ' if coef >= 0 else '- '}{np.abs(coef):.3f} {term}" for coef, ter
            pbar.set_description(f"T x: {epoch_train_reconstruction_loss:.3e} | T ddx: {epoch_train_ddx_loss:.3

        # If the entire mask is zero, stop training
        if torch.sum(sindy_autoencoder.sindy.coef_mask) == 0:
            break
```

Train on Cartesian Data

In [ ]:
```
def get_data_cartesian():
    x_train, dx_train, ddx_train, z_train, dz_train, ddz_train, t_train = create_pendulum_data(
        z0_min=-np.pi,
        z0_max=np.pi,
        dz0_min=-2.1,
        dz0_max=2.1,
        coefficients=[target_coefficients[term] for term in terms_np],
        terms=[terms_np[term] for term in terms_np],
        T=T,
        dt=DT,
        N=100,
        embedding=embed_cartesian,
    )

    x_val, dx_val, ddx_val, z_val, dz_val, ddz_val, t_val = create_pendulum_data(
        z0_min=-np.pi,
        z0_max=np.pi,
        dz0_min=-2.1,
        dz0_max=2.1,
        coefficients=[target_coefficients[term] for term in terms_np],
        terms=[terms_np[term] for term in terms_np],
        T=T,
        dt=DT,
        N=20,
        embedding=embed_cartesian,
    )

    # Create tensors
    x_train = torch.tensor(x_train).float().view(-1, 2)
    dx_train = torch.tensor(dx_train).float().view(-1, 2)
    ddx_train = torch.tensor(ddx_train).float().view(-1, 2)

    z_train = torch.tensor(z_train).float().view(-1, 1)
    dz_train = torch.tensor(dz_train).float().view(-1, 1)
    ddz_train = torch.tensor(ddz_train).float().view(-1, 1)

    # Shuffle the training data
    idx = torch.randperm(z_train.shape[0])
    x_train = x_train[idx]
    dx_train = dx_train[idx]
    ddx_train = ddx_train[idx]

    z_train = z_train[idx]
    dz_train = dz_train[idx]
    ddz_train = ddz_train[idx]

    x_val = torch.tensor(x_val).float().view(-1, 2)
    dx_val = torch.tensor(dx_val).float().view(-1, 2)
    ddx_val = torch.tensor(ddx_val).float().view(-1, 2)

    z_val = torch.tensor(z_val).float().view(-1, 1)
    dz_val = torch.tensor(dz_val).float().view(-1, 1)
    ddz_val = torch.tensor(ddz_val).float().view(-1, 1)

    return x_train, dx_train, ddx_train, z_train, dz_train, ddz_train, t_train, x_val, dx_val, ddx_val, z_val,
```

In [ ]:
```
x_train, dx_train, ddx_train, z_train, dz_train, ddz_train, t_train, x_val, dx_val, ddx_val, z_val, dz_val, ddz_
```

In [ ]:
```
# Test set
x_test, dx_test, ddx_test, z_test, dz_test, ddz_test, t_test = create_pendulum_data(
    z0_min=-np.pi,
    z0_max=np.pi,
    dz0_min=-2.1,
    dz0_max=2.1,
    coefficients=[target_coefficients[term] for term in terms_np],
    terms=[terms_np[term] for term in terms_np],
    T=T * 5,
    dt=DT,
    N=100,
    embedding=embed_cartesian,
)
```

In [ ]:
```
# Create tensors
x_test = torch.tensor(x_test).float().view(-1, 2)
dx_test = torch.tensor(dx_test).float().view(-1, 2)
ddx_test = torch.tensor(ddx_test).float().view(-1, 2)

z_test = torch.tensor(z_test).float().view(-1, 1)
dz_test = torch.tensor(dz_test).float().view(-1, 1)
```

```
ddz_test = torch.tensor(ddz_test).float().view(-1, 1)

print(f"{x_test.shape = }")
print(f"{dx_test.shape = }")
print(f"{ddx_test.shape = }")
print(f"{z_test.shape = }")
print(f"{dz_test.shape = }")
print(f"{ddz_test.shape = }")
```

```
x_test.shape = torch.Size([25000, 2])
dx_test.shape = torch.Size([25000, 2])
ddx_test.shape = torch.Size([25000, 2])
z_test.shape = torch.Size([25000, 1])
dz_test.shape = torch.Size([25000, 1])
ddz_test.shape = torch.Size([25000, 1])
```

PTAT

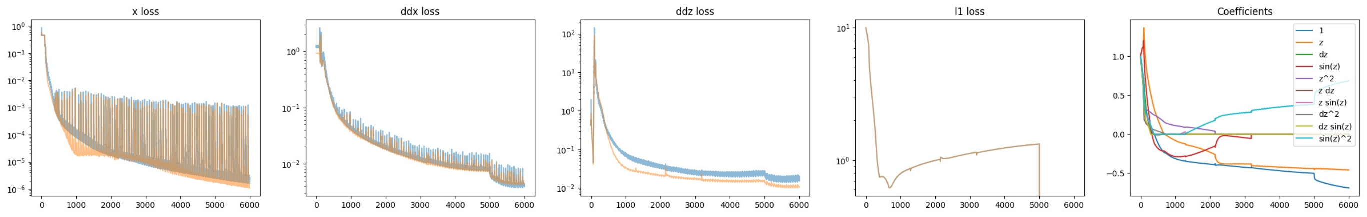
In [ ]:
```
lib = Library(['z', 'dz'], 1, terms_torch)
thresholder = PatientTrendAwareThresholder(lib, threshold_a=0.1, threshold_b=0.002, patience=1000)
sindy = SINDy(lib, thresholder, init="ones").to(device)

sindy_autoencoder = SINDyAutoencoder(sindy, input_dim=2, encoder_sizes=[32] * 5, decoder_sizes=[32] * 5).to(dev

optimizer = Adam(sindy_autoencoder.parameters(), lr=1e-3)
```

In [ ]:
```
batch_size = 1000
loss_history = {}

train_sindy_autoencoder(loss_history, sindy_autoencoder, optimizer,
                        x_train.to(device), dx_train.to(device), ddx_train.to(device),
                        x_val.to(device), dx_val.to(device), ddx_val.to(device),
                        epochs=6000, refinement_after_epochs=5000,
                        l1_weight=1e-5 / batch_size, ddx_weight=5e-4, ddz_weight=5e-5,
                        batch_size=batch_size, verbose=True)
```

```
T x: 2.173e-09 | T ddx: 4.099e-06 | T ddz: 1.729e-05 | T L1: 0.000e+00 | V x: 1.088e-09 | V ddx: 4.388e-06 | V d
dz: 1.057e-05 | V L1: 0.000e+00 | T: 3 (- 0.689 1 - 0.458 z + 0.685 sin(z)^2): 100%|██████████| 6000/6000 [02:48
<00:00, 35.60it/s]
```

In [ ]:
```
fig, ax = plt.subplots(1, 5, figsize=(30, 4))

for i, loss_key in enumerate(['x', 'ddx', 'ddz', 'l1']):
    ax[i].plot(*np.array(loss_history[f'train_{loss_key}']).T, label=f'Train {loss_key}', color='tab:blue', alpl
    ax[i].plot(*np.array(loss_history[f'val_{loss_key}']).T, label=f'Validation {loss_key}', color='tab:orange'
    # ax[i].set_xscale('log')
    ax[i].set_yscale('log')
    ax[i].set_title(f'{loss_key} loss')

coef_epochs = np.array([c[0] for c in loss_history['coefficients']])
coef_values = np.array([c[1] for c in loss_history['coefficients']])

for coefficient, coef_name in enumerate(sindy_autoencoder.sindy.library.terms.keys()):
    ax[-1].plot(coef_epochs, coef_values[:, 0, coefficient], label=coef_name)

# ax[-1].set_xscale('log')
ax[-1].set_title('Coefficients')
ax[-1].legend(loc='upper right')
```

Out[ ]:
```
<matplotlib.legend.Legend at 0x7f5078f693d0>
```



In [ ]:
```
# Print the final equation
coefs = sindy_autoencoder.sindy.coef[0].detach().cpu().numpy()
coef_mask = sindy_autoencoder.sindy.coef_mask[0].detach().cpu().numpy()
terms_pred = sindy_autoencoder.sindy.library.terms

equation_string = " ".join([f"{'+ ' if coef >= 0 else '- '}{np.abs(coef):.3f} {term}" for coef, term, active in
print(equation_string)
```

```
- 0.689 1 - 0.458 z + 0.685 sin(z)^2
```

In [ ]:
```
results = {
    'fvu_x': [],
    'fvu_ddx': [],
    'fvu_ddz': [],
    'coefficients': [],
    'resimulation_z': [],
    'resimulation_x': [],
}

for i in range(N_REPEAT):
    x_train, dx_train, ddx_train, z_train, dz_train, ddz_train, t_train, x_val, dx_val, ddx_val, z_val, dz_val,

    lib = Library(['z', 'dz'], 1, terms_torch)
    thresholder = PatientTrendAwareThresholder(lib, threshold_a=0.1, threshold_b=0.002, patience=1000)
    sindy = SINDy(lib, thresholder, init="ones").to(device)

    sindy_autoencoder = SINDyAutoencoder(sindy, input_dim=2, encoder_sizes=[32] * 5, decoder_sizes=[32] * 5).to

    optimizer = Adam(sindy_autoencoder.parameters(), lr=1e-3)
    batch_size = 1000
    loss_history = {}

    train_sindy_autoencoder(loss_history, sindy_autoencoder, optimizer,
                            x_train.to(device), dx_train.to(device), ddx_train.to(device),
                            x_val.to(device), dx_val.to(device), ddx_val.to(device),
                            epochs=6000, refinement_after_epochs=5000,
                            l1_weight=1e-5 / batch_size, ddx_weight=5e-4, ddz_weight=5e-5,
                            batch_size=batch_size, verbose=True)

    # Compute the FVU on the test set
    sindy_autoencoder.sindy.eval()

    with torch.no_grad():
        x_hat, ddx_hat_rhs, ddz_lhs, ddz_rhs = sindy_autoencoder.forward(x_test.to(device), dx_test.to(device),

        fvu_x = compute_FVU(x_test.to(device), x_hat)
        fvu_ddx = compute_FVU(ddx_test.to(device), ddx_hat_rhs)
        fvu_ddz = compute_FVU(ddz_lhs, ddz_rhs)

        results['fvu_x'].append(fvu_x.item())
        results['fvu_ddx'].append(fvu_ddx.item())
        results['fvu_ddz'].append(fvu_ddz.item())

        coefficients = sindy_autoencoder.sindy.coef.detach().cpu().numpy().copy()
        results['coefficients'].append(coefficients)

        # Encode the first x of the test set to get the initial conditions
        z0, dz0, _ = sindy_autoencoder.encode(
            x_test.reshape(100, T * 5, 2)[:, 0].to(device),
            dx_test.reshape(100, T * 5, 2)[:, 0].to(device))
        z0 = z0.detach().cpu().numpy().copy()[:, 0]
        dz0 = dz0.detach().cpu().numpy().copy()[:, 0]

        # Resmuluate the system
        t, z, dz = simulate_pendulum(z0, dz0, coefficients[0], [terms_np[term] for term in terms_np], T * 5, DT
        ddz = pendulum_rhs(z, dz, coefficients[0], [terms_np[term] for term in terms_np])
        x, dx, ddx = sindy_autoencoder.decode(
            torch.tensor(z).float().view(-1, 1).to(device),
            torch.tensor(dz).float().view(-1, 1).to(device),
            torch.tensor(ddz).float().view(-1, 1).to(device)
        )
        x = x.detach().cpu().numpy().reshape(100, T * 5, 2)
        dx = dx.detach().cpu().numpy().reshape(100, T * 5, 2)
        ddx = ddx.detach().cpu().numpy().reshape(100, T * 5, 2)

        # Save the resimulated x
        results['resimulation_z'].append(z)
        results['resimulation_x'].append(x)
```

```
T x: 3.795e-09 | T ddx: 2.103e-05 | T ddz: 9.226e-05 | T L1: 0.000e+00 | V x: 4.643e-09 | V ddx: 2.386e-05 | V d
dz: 1.007e-04 | V L1: 0.000e+00 | T: 1 (- 0.477 z): 100%|██████████| 6000/6000 [02:47<00:00, 35.81it/s]
T x: 6.224e-09 | T ddx: 2.284e-06 | T ddz: 1.726e-05 | T L1: 0.000e+00 | V x: 4.160e-07 | V ddx: 3.499e-06 | V d
dz: 5.386e-05 | V L1: 0.000e+00 | T: 3 (- 0.083 1 - 0.119 z - 0.783 sin(z)): 100%|██████████| 6000/6000 [02:45<0
0:00, 36.16it/s]
T x: 1.383e-09 | T ddx: 1.109e-06 | T ddz: 6.940e-06 | T L1: 0.000e+00 | V x: 2.010e-09 | V ddx: 1.210e-06 | V d
dz: 1.264e-05 | V L1: 0.000e+00 | T: 1 (- 1.007 sin(z)): 100%|██████████| 6000/6000 [02:52<00:00, 34.86it/s]
T x: 3.247e-09 | T ddx: 2.171e-06 | T ddz: 1.344e-05 | T L1: 0.000e+00 | V x: 1.562e-09 | V ddx: 1.019e-06 | V d
dz: 2.812e-06 | V L1: 0.000e+00 | T: 1 (- 0.983 sin(z)): 100%|██████████| 6000/6000 [02:53<00:00, 34.49it/s]
T x: 2.268e-09 | T ddx: 1.800e-05 | T ddz: 4.277e-05 | T L1: 0.000e+00 | V x: 7.791e-10 | V ddx: 2.300e-05 | V d
dz: 4.165e-05 | V L1: 0.000e+00 | T: 1 (- 0.512 z): 100%|██████████| 6000/6000 [02:54<00:00, 34.33it/s]
T x: 5.728e-10 | T ddx: 3.583e-06 | T ddz: 6.987e-06 | T L1: 0.000e+00 | V x: 2.604e-09 | V ddx: 4.192e-06 | V d
dz: 2.745e-05 | V L1: 0.000e+00 | T: 1 (- 0.813 sin(z)): 100%|██████████| 6000/6000 [02:54<00:00, 34.29it/s]
T x: 9.150e-09 | T ddx: 1.187e-05 | T ddz: 2.434e-05 | T L1: 0.000e+00 | V x: 3.840e-09 | V ddx: 1.587e-05 | V d
dz: 1.913e-05 | V L1: 0.000e+00 | T: 3 (- 0.201 1 - 0.064 z - 0.643 sin(z)): 100%|██████████| 6000/6000 [02:54<0
0:00, 34.44it/s]
T x: 2.567e-08 | T ddx: 6.728e-06 | T ddz: 5.908e-05 | T L1: 0.000e+00 | V x: 3.252e-08 | V ddx: 5.274e-06 | V d
dz: 5.684e-06 | V L1: 0.000e+00 | T: 3 (- 0.543 1 + 0.548 z sin(z) + 0.226 sin(z)^2): 100%|██████████| 6000/6000
[02:52<00:00, 34.71it/s]
T x: 7.491e-10 | T ddx: 2.812e-06 | T ddz: 1.080e-05 | T L1: 0.000e+00 | V x: 1.055e-05 | V ddx: 2.246e-05 | V d
dz: 5.703e-04 | V L1: 0.000e+00 | T: 4 (- 0.925 1 + 0.216 z^2 + 0.371 z sin(z) - 0.116 dz^2): 100%|██████████| 6
000/6000 [02:52<00:00, 34.79it/s]
T x: 7.203e-09 | T ddx: 9.110e-06 | T ddz: 2.084e-05 | T L1: 0.000e+00 | V x: 1.762e-09 | V ddx: 1.044e-05 | V d
dz: 8.802e-06 | V L1: 0.000e+00 | T: 3 (- 0.090 1 + 0.219 z - 0.958 sin(z)): 100%|██████████| 6000/6000 [02:53<0
0:00, 34.54it/s]
```

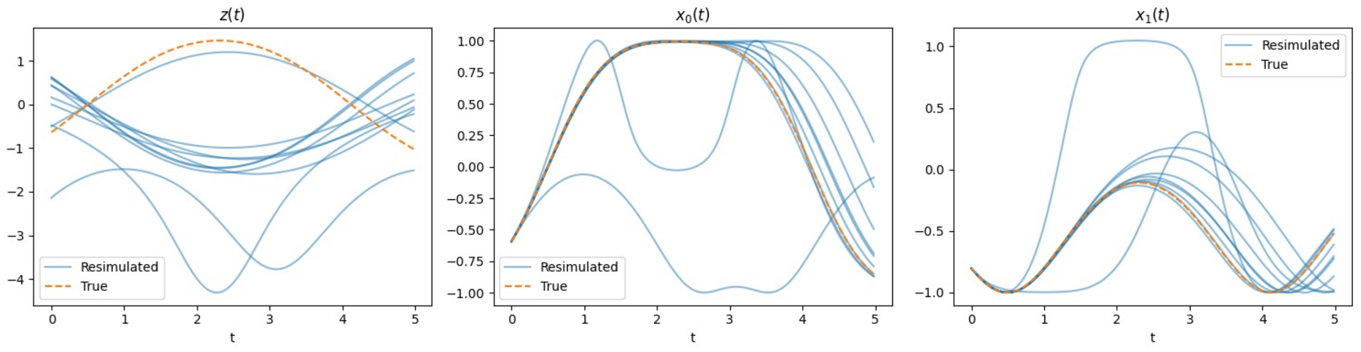
In [ ]:
```python
# Compute the mean and std of the FVU
results['fvu_x'] = np.array(results['fvu_x'])
results['fvu_ddx'] = np.array(results['fvu_ddx'])
results['fvu_ddz'] = np.array(results['fvu_ddz'])

print(f"FVU_x = {results['fvu_x'].mean():.4f} ± {results['fvu_x'].std():.4f}")
print(f"FVU_ddx = {results['fvu_ddx'].mean():.4f} ± {results['fvu_ddx'].std():.4f}")
print(f"FVU_ddz = {results['fvu_ddz'].mean():.4f} ± {results['fvu_ddz'].std():.4f}")
```

```
FVU_x = 0.0007 ± 0.0017
FVU_ddx = 0.0086 ± 0.0070
FVU_ddz = 0.0804 ± 0.0550
```

In [ ]:
```python
# Calculate the MSE between ground truth and resimulation at each timestep
resimulation_x = np.array(results['resimulation_x'])
resimulation_z = np.array(results['resimulation_z'])

print(f"{resimulation_x.shape = }")
print(f"{resimulation_z.shape = }")

resimulation_mse_x_median = np.median((resimulation_x - x_test.reshape(100, T * 5, 2).cpu().numpy())**2, axis=(
resimulation_mse_x_quantiles = np.quantile((resimulation_x - x_test.reshape(100, T * 5, 2).cpu().numpy())**2, [(

resimulation_mse_z_median = np.median((resimulation_z - z_test.reshape(100, T * 5).cpu().numpy())**2, axis=(0, 
resimulation_mse_z_quantiles = np.quantile((resimulation_z - z_test.reshape(100, T * 5).cpu().numpy())**2, [0.1
```

```
resimulation_x.shape = (10, 100, 250, 2)
resimulation_z.shape = (10, 100, 250)
```

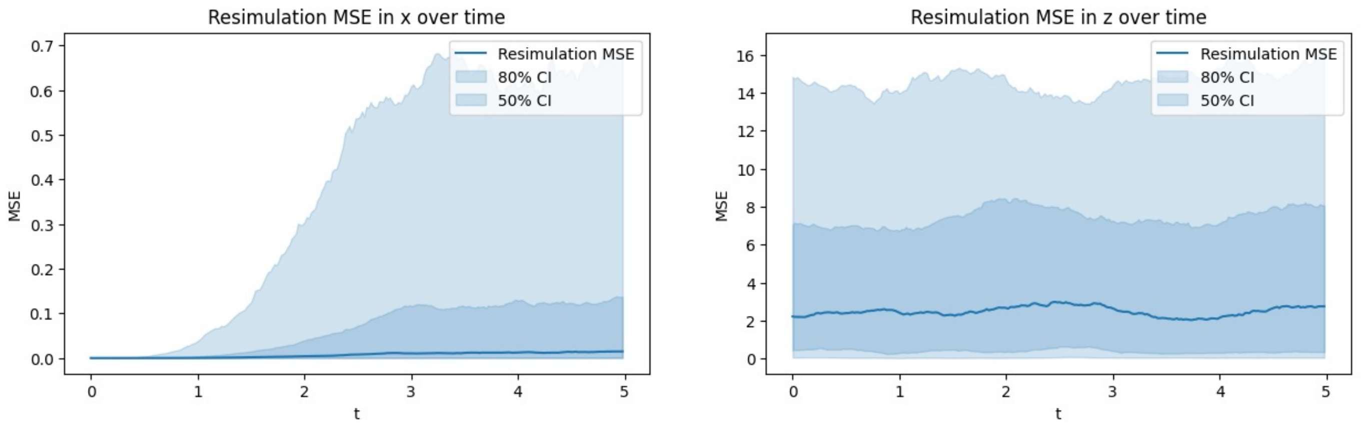
In [ ]:
```python
# Show a resimulated trajectory
fig, axes= plt.subplots(1, 3, figsize=(15, 4))

# x
for dim, ax in enumerate(axes[1:]):
    for i in range(len(results['resimulation_x'])):
        ax.plot(t_test, results['resimulation_x'][i][0, :, dim], label='Resimulated' if i == 0 else None, alpha=
    ax.plot(t_test, x_test.reshape(100, T * 5, 2)[0, :, dim], label='True', color='tab:orange', linestyle='--')
    ax.set_xlabel('t')
    ax.set_title(f'$x_{dim}(t)$')
    ax.legend()

# z
for i in range(len(results['resimulation_x'])):
    axes[0].plot(t_test, results['resimulation_z'][i][0], label='Resimulated' if i == 0 else None, alpha=0.5, c
axes[0].plot(t_test, z_test.reshape(100, T * 5, 1)[0, :, 0], label='True', color='tab:orange', linestyle='--')
axes[0].set_xlabel('t')
axes[0].set_title('$z(t)$')
axes[0].legend()

fig.tight_layout()
```



Note that the some learned and resimulated z(t) trajectories are mirrored due to the symmetry of the system.

```
In [ ]: # Show the resimulation error over time
        fig, axes = plt.subplots(1, 2, figsize=(15, 4))

        axes[0].plot(t_test, resimulation_mse_x_median, label='Resimulation MSE', color='C0')
        axes[0].fill_between(t_test, resimulation_mse_x_quantiles[0], resimulation_mse_x_quantiles[-1], alpha=0.2, labe
        axes[0].fill_between(t_test, resimulation_mse_x_quantiles[1], resimulation_mse_x_quantiles[-2], alpha=0.2, labe
        axes[0].set_xlabel('t')
        axes[0].set_ylabel('MSE')
        axes[0].set_title('Resimulation MSE in x over time')
        axes[0].legend()

        axes[1].plot(t_test, resimulation_mse_z_median, label='Resimulation MSE', color='C0')
        axes[1].fill_between(t_test, resimulation_mse_z_quantiles[0], resimulation_mse_z_quantiles[-1], alpha=0.2, labe
        axes[1].fill_between(t_test, resimulation_mse_z_quantiles[1], resimulation_mse_z_quantiles[-2], alpha=0.2, labe
        axes[1].set_xlabel('t')
        axes[1].set_ylabel('MSE')
        axes[1].set_title('Resimulation MSE in z over time')
        axes[1].legend()
```

```
Out[ ]: <matplotlib.legend.Legend at 0x7f50760e0150>
```



The Resimulation MSE in z over time does not account for the inherent symmetry of the system. Ideally, you should account for this by flipping the sign of the resimulated trajectories to match the GT sign.

ST

```
In [ ]: x_train, dx_train, ddx_train, z_train, dz_train, ddz_train, t_train, x_val, dx_val, ddx_val, z_val, dz_val, ddz_
```

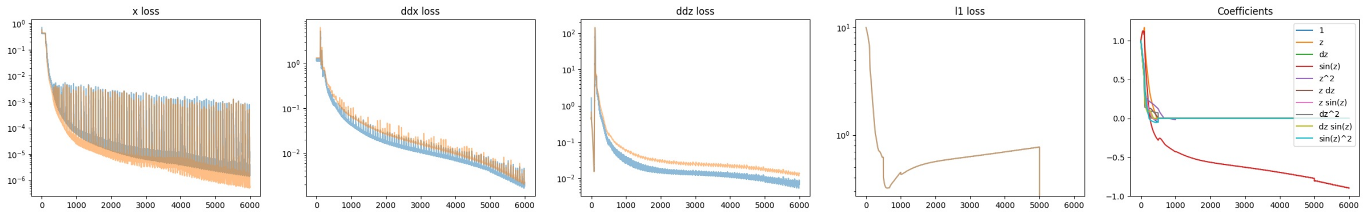
```
In [ ]: lib = Library(['z', 'dz'], 1, terms_torch)
        thresholder = SequentialThresholder(lib, threshold=0.1, interval=500)
        sindy = SINDy(lib, thresholder, init="ones").to(device)

        sindy_autoencoder = SINDyAutoencoder(sindy, input_dim=2, encoder_sizes=[32] * 5, decoder_sizes=[32] * 5).to(dev

        optimizer = Adam(sindy_autoencoder.parameters(), lr=1e-3)
```

```
In [ ]: batch_size = 1000
        loss_history = {}

        train_sindy_autoencoder(loss_history, sindy_autoencoder, optimizer,
                                x_train.to(device), dx_train.to(device), ddx_train.to(device),
                                x_val.to(device), dx_val.to(device), ddx_val.to(device),
                                epochs=6000, refinement_after_epochs=5000,
                                l1_weight=1e-5 / batch_size, ddx_weight=5e-4, ddz_weight=5e-5,
                                batch_size=batch_size, verbose=True)
```

```
T x: 2.562e-09 | T ddx: 1.973e-06 | T ddz: 7.587e-06 | T L1: 0.000e+00 | V x: 5.853e-10 | V ddx: 2.229e-06 | V d
dz: 1.344e-05 | V L1: 0.000e+00 | T: 1 (- 0.898 sin(z)): 100%|██████████| 6000/6000 [02:52<00:00, 34.70it/s]
```

In [ ]:
```python
fig, ax = plt.subplots(1, 5, figsize=(30, 4))

for i, loss_key in enumerate(['x', 'ddx', 'ddz', 'l1']):
    ax[i].plot(*np.array(loss_history[f'train_{loss_key}']).T, label=f'Train {loss_key}', color='tab:blue', alpl
    ax[i].plot(*np.array(loss_history[f'val_{loss_key}']).T, label=f'Validation {loss_key}', color='tab:orange'
    # ax[i].set_xscale('log')
    ax[i].set_yscale('log')
    ax[i].set_title(f'{loss_key} loss')

coef_epochs = np.array([c[0] for c in loss_history['coefficients']])
coef_values = np.array([c[1] for c in loss_history['coefficients']])

for coefficient, coef_name in enumerate(sindy_autoencoder.sindy.library.terms.keys()):
    ax[-1].plot(coef_epochs, coef_values[:, 0, coefficient], label=coef_name)

# ax[-1].set_xscale('log')
ax[-1].set_title('Coefficients')
ax[-1].legend(loc='upper right')
```

Out[ ]:
```
<matplotlib.legend.Legend at 0x7f50833093d0>
```



In [ ]:
```python
# Print the final equation
coefs = sindy_autoencoder.sindy.coef[0].detach().cpu().numpy()
coef_mask = sindy_autoencoder.sindy.coef_mask[0].detach().cpu().numpy()
terms_pred = sindy_autoencoder.sindy.library.terms

equation_string = " ".join([f"{'+ ' if coef >= 0 else '- '}{np.abs(coef):.3f} {term}" for coef, term, active in
print(equation_string)
```

```
- 0.898 sin(z)
```

In [ ]:
```python
results = {
    'fvu_x': [],
    'fvu_ddx': [],
    'fvu_ddz': [],
    'coefficients': [],
    'resimulation_z': [],
    'resimulation_x': [],
}

for i in range(N_REPEAT):
    x_train, dx_train, ddx_train, z_train, dz_train, ddz_train, t_train, x_val, dx_val, ddx_val, z_val, dz_val,

    lib = Library(['z', 'dz'], 1, terms_torch)
    thresholder = SequentialThresholder(lib, threshold=0.1, interval=500)
    sindy = SINDy(lib, thresholder, init="ones").to(device)

    sindy_autoencoder = SINDyAutoencoder(sindy, input_dim=2, encoder_sizes=[32] * 5, decoder_sizes=[32] * 5).to

    optimizer = Adam(sindy_autoencoder.parameters(), lr=1e-3)
    batch_size = 1000
    loss_history = {}

    train_sindy_autoencoder(loss_history, sindy_autoencoder, optimizer,
                            x_train.to(device), dx_train.to(device), ddx_train.to(device),
                            x_val.to(device), dx_val.to(device), ddx_val.to(device),
                            epochs=6000, refinement_after_epochs=5000,
                            l1_weight=1e-5 / batch_size, ddx_weight=5e-4, ddz_weight=5e-5,
                            batch_size=batch_size, verbose=True)

    # Compute the FVU on the test set
    sindy_autoencoder.sindy.eval()

    with torch.no_grad():
        x_hat, ddx_hat_rhs, ddz_lhs, ddz_rhs = sindy_autoencoder.forward(x_test.to(device), dx_test.to(device),

        fvu_x = compute_FVU(x_test.to(device), x_hat)
        fvu_ddx = compute_FVU(ddx_test.to(device), ddx_hat_rhs)
        fvu_ddz = compute_FVU(ddz_lhs, ddz_rhs)

        results['fvu_x'].append(fvu_x.item())
        results['fvu_ddx'].append(fvu_ddx.item())
        results['fvu_ddz'].append(fvu_ddz.item())

        coefficients = sindy_autoencoder.sindy.coef.detach().cpu().numpy().copy()
        results['coefficients'].append(coefficients)

        # Encode the first x of the test set to get the initial conditions
        z0, dz0, _ = sindy_autoencoder.encode(
            x_test.reshape(100, T * 5, 2)[:, 0].to(device),
            dx_test.reshape(100, T * 5, 2)[:, 0].to(device))
        z0 = z0.detach().cpu().numpy().copy()[:, 0]
        dz0 = dz0.detach().cpu().numpy().copy()[:, 0]

        # Resmuluate the system
        t, z, dz = simulate_pendulum(z0, dz0, coefficients[0], [terms_np[term] for term in terms_np], T * 5, DT
        ddz = pendulum_rhs(z, dz, coefficients[0], [terms_np[term] for term in terms_np])
        x, dx, ddx = sindy_autoencoder.decode(
            torch.tensor(z).float().view(-1, 1).to(device),
            torch.tensor(dz).float().view(-1, 1).to(device),
            torch.tensor(ddz).float().view(-1, 1).to(device)
        )
        x = x.detach().cpu().numpy().reshape(100, T * 5, 2)
        dx = dx.detach().cpu().numpy().reshape(100, T * 5, 2)
        ddx = ddx.detach().cpu().numpy().reshape(100, T * 5, 2)

        # Save the resimulated x
        results['resimulation_z'].append(z)
        results['resimulation_x'].append(x)
```

```
T x: 7.602e-09 | T ddx: 5.108e-06 | T ddz: 1.121e-05 | T L1: 0.000e+00 | V x: 9.397e-09 | V ddx: 5.138e-06 | V d
dz: 6.509e-06 | V L1: 0.000e+00 | T: 1 (- 0.752 sin(z)): 100%|██████████| 6000/6000 [02:54<00:00, 34.35it/s]
T x: 5.232e-09 | T ddx: 4.842e-06 | T ddz: 2.453e-05 | T L1: 0.000e+00 | V x: 1.847e-09 | V ddx: 3.406e-06 | V d
dz: 1.006e-05 | V L1: 0.000e+00 | T: 1 (- 0.790 sin(z)): 100%|██████████| 6000/6000 [02:55<00:00, 34.11it/s]
T x: 5.556e-09 | T ddx: 4.930e-05 | T ddz: 9.251e-05 | T L1: 0.000e+00 | V x: 3.061e-10 | V ddx: 6.932e-05 | V d
dz: 5.076e-05 | V L1: 0.000e+00 | T: 3 (+ 0.288 z + 0.291 z^2 - 0.554 dz^2): 100%|██████████| 6000/6000 [02:55<0
0:00, 34.25it/s]
T x: 1.625e-09 | T ddx: 1.411e-06 | T ddz: 1.536e-05 | T L1: 0.000e+00 | V x: 5.188e-09 | V ddx: 3.386e-06 | V d
dz: 1.565e-05 | V L1: 0.000e+00 | T: 1 (- 1.042 sin(z)): 100%|██████████| 6000/6000 [02:55<00:00, 34.23it/s]
T x: 1.440e-08 | T ddx: 3.555e-06 | T ddz: 1.590e-05 | T L1: 0.000e+00 | V x: 7.116e-07 | V ddx: 3.381e-06 | V d
dz: 5.534e-05 | V L1: 0.000e+00 | T: 3 (- 0.419 1 - 0.828 sin(z) + 0.128 z^2): 100%|██████████| 6000/6000 [02:55
<00:00, 34.21it/s]
T x: 6.997e-09 | T ddx: 3.755e-06 | T ddz: 4.314e-05 | T L1: 0.000e+00 | V x: 2.598e-09 | V ddx: 3.590e-06 | V d
dz: 1.018e-05 | V L1: 0.000e+00 | T: 3 (- 0.434 1 - 0.634 sin(z) + 0.496 sin(z)^2): 100%|██████████| 6000/6000 [
02:54<00:00, 34.39it/s]
T x: 7.293e-09 | T ddx: 2.434e-06 | T ddz: 3.961e-05 | T L1: 0.000e+00 | V x: 3.351e-09 | V ddx: 2.513e-06 | V d
dz: 7.463e-06 | V L1: 0.000e+00 | T: 1 (- 0.938 sin(z)): 100%|██████████| 6000/6000 [02:55<00:00, 34.14it/s]
T x: 6.487e-08 | T ddx: 9.483e-06 | T ddz: 6.306e-05 | T L1: 0.000e+00 | V x: 8.941e-08 | V ddx: 1.443e-05 | V d
dz: 2.305e-05 | V L1: 0.000e+00 | T: 1 (- 0.735 sin(z)): 100%|██████████| 6000/6000 [02:54<00:00, 34.39it/s]
T x: 1.481e-09 | T ddx: 7.427e-07 | T ddz: 6.455e-06 | T L1: 0.000e+00 | V x: 5.748e-10 | V ddx: 3.969e-07 | V d
dz: 6.927e-07 | V L1: 0.000e+00 | T: 1 (- 1.019 sin(z)): 100%|██████████| 6000/6000 [02:57<00:00, 33.71it/s]
T x: 5.061e-08 | T ddx: 1.220e-06 | T ddz: 1.321e-05 | T L1: 0.000e+00 | V x: 5.900e-08 | V ddx: 2.149e-06 | V d
dz: 8.694e-06 | V L1: 0.000e+00 | T: 1 (- 0.967 sin(z)): 100%|██████████| 6000/6000 [02:54<00:00, 34.37it/s]
```

```
In [ ]: # Compute the mean and std of the FVU
        results['fvu_x'] = np.array(results['fvu_x'])
        results['fvu_ddx'] = np.array(results['fvu_ddx'])
        results['fvu_ddz'] = np.array(results['fvu_ddz'])

        print(f"FVU_x = {results['fvu_x'].mean():.4f} ± {results['fvu_x'].std():.4f}")
        print(f"FVU_ddx = {results['fvu_ddx'].mean():.4f} ± {results['fvu_ddx'].std():.4f}")
        print(f"FVU_ddz = {results['fvu_ddz'].mean():.4f} ± {results['fvu_ddz'].std():.4f}")
```

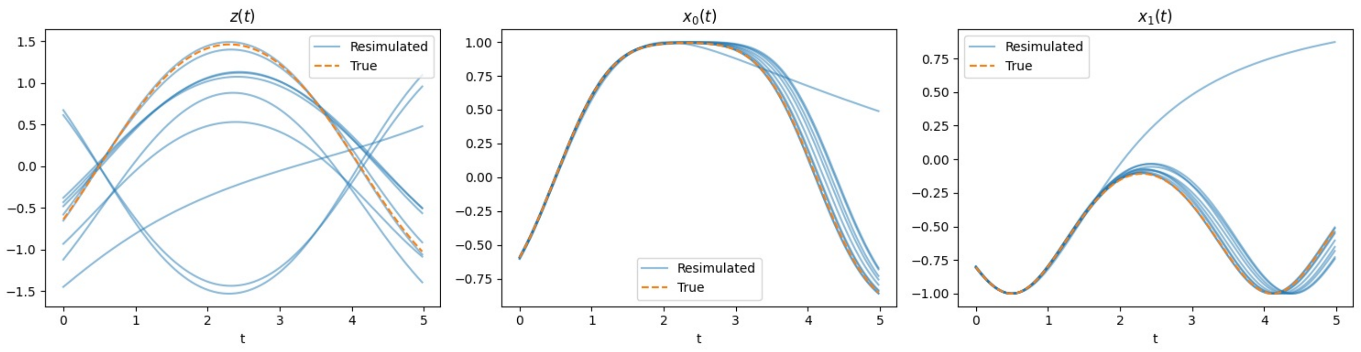
```
FVU_x = 0.0007 ± 0.0006
FVU_ddx = 0.0088 ± 0.0165
FVU_ddz = 0.0929 ± 0.1370
```

```
In [ ]: # Calculate the MSE between ground truth and resimulation at each timestep
        resimulation_x = np.array(results['resimulation_x'])
        resimulation_z = np.array(results['resimulation_z'])

        print(f"{resimulation_x.shape = }")
        print(f"{resimulation_z.shape = }")

        resimulation_mse_x_median = np.median((resimulation_x - x_test.reshape(100, T * 5, 2).cpu().numpy())**2, axis=(
        resimulation_mse_x_quantiles = np.quantile((resimulation_x - x_test.reshape(100, T * 5, 2).cpu().numpy())**2, [(
        resimulation_mse_z_median = np.median((resimulation_z - z_test.reshape(100, T * 5).cpu().numpy())**2, axis=(0, 
        resimulation_mse_z_quantiles = np.quantile((resimulation_z - z_test.reshape(100, T * 5).cpu().numpy())**2, [0.1
```

```
resimulation_x.shape = (10, 100, 250, 2)
resimulation_z.shape = (10, 100, 250)
```

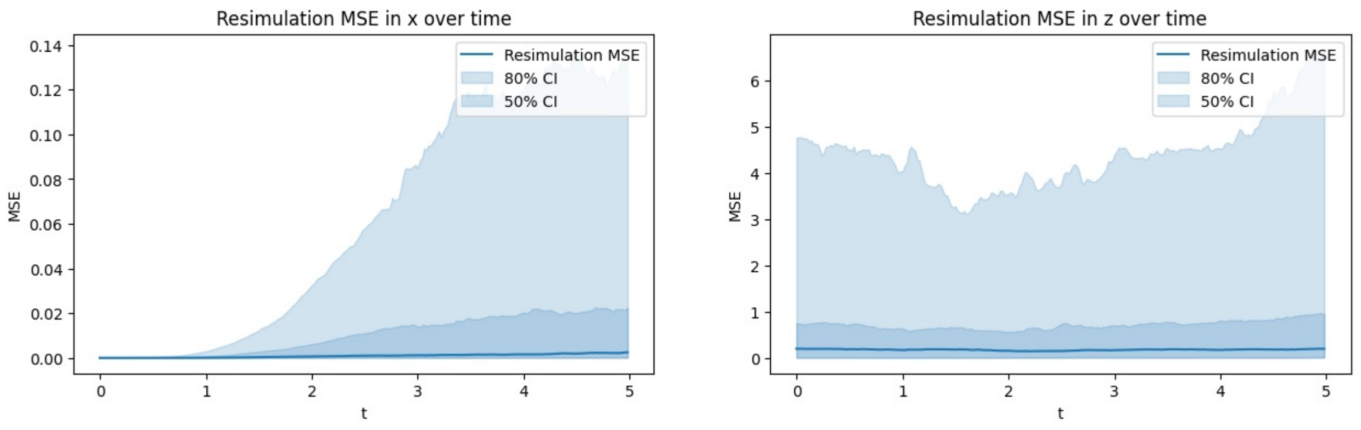
```
In [ ]: # Show a resimulated trajectory
        fig, axes= plt.subplots(1, 3, figsize=(15, 4))
```

```
# x
for dim, ax in enumerate(axes[1:]):
    for i in range(len(results['resimulation_x'])):
        ax.plot(t_test, results['resimulation_x'][i][0, :, dim], label='Resimulated' if i == 0 else None, alpha
    ax.plot(t_test, x_test.reshape(100, T * 5, 2)[0, :, dim], label='True', color='tab:orange', linestyle='--')
    ax.set_xlabel('t')
    ax.set_title(f'$x_{dim}(t)$')
    ax.legend()

# z
for i in range(len(results['resimulation_x'])):
    axes[0].plot(t_test, results['resimulation_z'][i][0], label='Resimulated' if i == 0 else None, alpha=0.5, c
axes[0].plot(t_test, z_test.reshape(100, T * 5, 1)[0, :, 0], label='True', color='tab:orange', linestyle='--')
axes[0].set_xlabel('t')
axes[0].set_title('$z(t)$')
axes[0].legend()

fig.tight_layout()
```



```
In [ ]: # Show the resimulation error over time
        fig, axes = plt.subplots(1, 2, figsize=(15, 4))

        axes[0].plot(t_test, resimulation_mse_x_median, label='Resimulation MSE', color='C0')
        axes[0].fill_between(t_test, resimulation_mse_x_quantiles[0], resimulation_mse_x_quantiles[-1], alpha=0.2, label
        axes[0].fill_between(t_test, resimulation_mse_x_quantiles[1], resimulation_mse_x_quantiles[-2], alpha=0.2, label
        axes[0].set_xlabel('t')
        axes[0].set_ylabel('MSE')
        axes[0].set_title('Resimulation MSE in x over time')
        axes[0].legend()

        axes[1].plot(t_test, resimulation_mse_z_median, label='Resimulation MSE', color='C0')
        axes[1].fill_between(t_test, resimulation_mse_z_quantiles[0], resimulation_mse_z_quantiles[-1], alpha=0.2, label
        axes[1].fill_between(t_test, resimulation_mse_z_quantiles[1], resimulation_mse_z_quantiles[-2], alpha=0.2, label
        axes[1].set_xlabel('t')
        axes[1].set_ylabel('MSE')
        axes[1].set_title('Resimulation MSE in z over time')
        axes[1].legend()
```

```
Out[ ]: <matplotlib.legend.Legend at 0x7f507b205290>
```



3 Bonus: SINDy-Autoencoder on Videos

```
In [ ]: T = 100
```

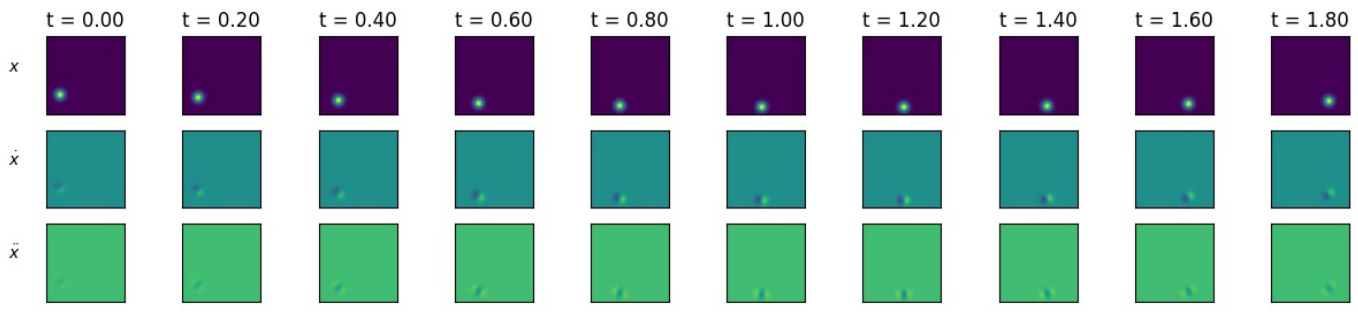
```
In [ ]: def embed_grid(z, dz, ddz, t, res=50, sigma=0.1):
            """
            Artificially embed the point mass intoa video of the tip of the pendulum
            """
            x = np.stack([np.sin(z), -np.cos(z)]).transpose(1,2,0)  # Shape (N, T, 2)
```

```
    # Create grid of shape (N, T, res, res), i.e. a list of videos of the tip of the pendulum
    # Model the tip of the pendulum as a gaussian with mean x and std sigma
    grid = np.zeros((x.shape[0], x.shape[1], res, res))
    dgrid = np.zeros((x.shape[0], x.shape[1], res, res))
    ddgrid = np.zeros((x.shape[0], x.shape[1], res, res))

    image_linspace = np.linspace(-1.2, 1.2, res)

    grid = np.exp(-((image_linspace[None, None, :, None] - x[:, :, 0, None, None])**2 + (image_linspace[None, No

    # Numerically compute the time derivative of the grid
    dgrid[:, 1:, :, :] = (grid[:, 1:, :, :] - grid[:, :-1, :, :]) / (t[None, 1:, None, None] - t[None, :-1, None

    # Numerically compute the time derivative of the grid
    ddgrid[:, 1:-1, :, :] = (grid[:, 2:, :, :] - 2 * grid[:, 1:-1, :, :] + grid[:, :-2, :, :]) / (t[None, 1:-1,

    # Remove the first and last frame
    grid = grid[:, 1:-1, :, :]
    dgrid = dgrid[:, 1:-1, :, :]
    ddgrid = ddgrid[:, 1:-1, :, :]

    return grid, dgrid, ddgrid
```

In [ ]:
```
# Simulate
x, dx, ddx, z, dz, ddz, t = create_pendulum_data(
    z0_min=-np.pi,
    z0_max=np.pi,
    dz0_min=-2.1,
    dz0_max=2.1,
    coefficients=[target_coefficients[term] for term in terms_np],
    terms=[terms_np[term] for term in terms_np],
    T=T * 10,
    dt=DT,
    N=2,
    embedding=embed_grid
)

print(f"{t.shape = }")
print(f"{x.shape = }")
print(f"{dx.shape = }")
print(f"{ddx.shape = }")
print(f"{z.shape = }")
print(f"{dz.shape = }")
print(f"{ddz.shape = }")
```

```
t.shape = (1000,)
x.shape = (2, 998, 50, 50)
dx.shape = (2, 998, 50, 50)
ddx.shape = (2, 998, 50, 50)
z.shape = (2, 1000)
dz.shape = (2, 1000)
ddz.shape = (2, 1000)
```

In [ ]:
```
# Show the grid
fig, ax = plt.subplots(3, 10, figsize=(15, 3))

xmin, xmax = x.min(), x.max()
dxmin, dxmax = dx.min(), dx.max()
ddxmin, ddxmax = ddx.min(), ddx.max()

for d, (data, lim) in enumerate(zip([x, dx, ddx], [(xmin, xmax), (dxmin, dxmax), (ddxmin, ddxmax)])):
    for i in range(10):
        ax[d, i].imshow(data[0, i * 10, :, :].T, origin="lower", vmin=lim[0], vmax=lim[1])
        ax[d, i].set_xticks([])
        ax[d, i].set_yticks([])

        if i == 0:
            ax[d, i].set_ylabel(["$x$", "$\dot x$", "$\ddot x$"][d], rotation=0, labelpad=20)

        if d == 0:
            ax[d, i].set_title(f"t = {t[i * 10]:.2f}")
```



```
In [ ]: def get_data_video(T = 50):
            x_train, dx_train, ddx_train, z_train, dz_train, ddz_train, t_train = create_pendulum_data(
                z0_min=-np.pi,
                z0_max=np.pi,
                dz0_min=-2.1,
                dz0_max=2.1,
                coefficients=[target_coefficients[term] for term in terms_np],
                terms=[terms_np[term] for term in terms_np],
                T=T,
                dt=DT,
                N=100,
                embedding=embed_grid,
            )

            x_val, dx_val, ddx_val, z_val, dz_val, ddz_val, t_val = create_pendulum_data(
                z0_min=-np.pi,
                z0_max=np.pi,
                dz0_min=-2.1,
                dz0_max=2.1,
                coefficients=[target_coefficients[term] for term in terms_np],
                terms=[terms_np[term] for term in terms_np],
                T=T,
                dt=DT,
                N=20,
                embedding=embed_grid,
            )

            # Create tensors
            x_train = torch.tensor(x_train).float().view(-1, 2500)
            dx_train = torch.tensor(dx_train).float().view(-1, 2500)
            ddx_train = torch.tensor(ddx_train).float().view(-1, 2500)

            z_train = torch.tensor(z_train).float().view(-1, 1)
            dz_train = torch.tensor(dz_train).float().view(-1, 1)
            ddz_train = torch.tensor(ddz_train).float().view(-1, 1)

            # Shuffle the training data
            idx = torch.randperm(x_train.shape[0])
            x_train = x_train[idx]
            dx_train = dx_train[idx]
            ddx_train = ddx_train[idx]

            z_train = z_train[idx]
            dz_train = dz_train[idx]
            ddz_train = ddz_train[idx]

            x_val = torch.tensor(x_val).float().view(-1, 2500)
            dx_val = torch.tensor(dx_val).float().view(-1, 2500)
            ddx_val = torch.tensor(ddx_val).float().view(-1, 2500)

            z_val = torch.tensor(z_val).float().view(-1, 1)
            dz_val = torch.tensor(dz_val).float().view(-1, 1)
            ddz_val = torch.tensor(ddz_val).float().view(-1, 1)

            return x_train, dx_train, ddx_train, z_train, dz_train, ddz_train, t_train, x_val, dx_val, ddx_val, z_val,
```

```
In [ ]: # Test set
        x_test, dx_test, ddx_test, z_test, dz_test, ddz_test, t_test = create_pendulum_data(
            z0_min=-np.pi,
            z0_max=np.pi,
            dz0_min=-2.1,
            dz0_max=2.1,
            coefficients=[target_coefficients[term] for term in terms_np],
            terms=[terms_np[term] for term in terms_np],
            T=T * 5,
            dt=DT,
            N=100,
            embedding=embed_grid,
        )

        test_timesteps = T * 5 - 2
```

```
In [ ]: # Create tensors
        x_test = torch.tensor(x_test).float().view(-1, 2500)
        dx_test = torch.tensor(dx_test).float().view(-1, 2500)
        ddx_test = torch.tensor(ddx_test).float().view(-1, 2500)

        z_test = torch.tensor(z_test).float().view(-1, 1)
        dz_test = torch.tensor(dz_test).float().view(-1, 1)
        ddz_test = torch.tensor(ddz_test).float().view(-1, 1)

        print(f"{x_test.shape = }")
        print(f"{dx_test.shape = }")
        print(f"{ddx_test.shape = }")
        print(f"{z_test.shape = }")
        print(f"{dz_test.shape = }")
        print(f"{ddz_test.shape = }")
```

```
x_test.shape = torch.Size([49800, 2500])
dx_test.shape = torch.Size([49800, 2500])
ddx_test.shape = torch.Size([49800, 2500])
z_test.shape = torch.Size([50000, 1])
dz_test.shape = torch.Size([50000, 1])
ddz_test.shape = torch.Size([50000, 1])
```

PTAT

In [ ]:
```
x_train, dx_train, ddx_train, z_train, dz_train, ddz_train, t_train, x_val, dx_val, ddx_val, z_val, dz_val, ddz_
```

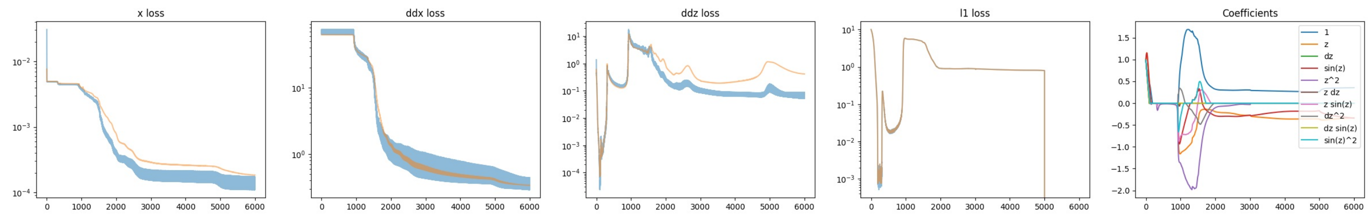
In [ ]:
```
lib = Library(['z', 'dz'], 1, terms_torch)
thresholder = PatientTrendAwareThresholder(lib, threshold_a=0.1, threshold_b=0.002, patience=1000)
sindy = SINDy(lib, thresholder, init="ones").to(device)

sindy_autoencoder = SINDyAutoencoder(sindy, input_dim=2500, encoder_sizes=[128, 64, 32], decoder_sizes=[32, 64,

optimizer = Adam(sindy_autoencoder.parameters(), lr=1e-3)
```

In [ ]:
```
batch_size = 1000
loss_history = {}

train_sindy_autoencoder(loss_history, sindy_autoencoder, optimizer,
                        x_train.to(device), dx_train.to(device), ddx_train.to(device),
                        x_val.to(device), dx_val.to(device), ddx_val.to(device),
                        epochs=6000, refinement_after_epochs=5000,
                        l1_weight=1e-5 / batch_size, ddx_weight=5e-4, ddz_weight=5e-5,
                        batch_size=batch_size, verbose=True)
```

```
  0%|          | 0/6000 [00:00<?, ?it/s]T x: 1.431e-07 | T ddx: 3.679e-04 | T ddz: 7.083e-05 | T L1: 0.000e+00 |
V x: 9.208e-08 | V ddx: 1.696e-04 | V ddz: 2.070e-04 | V L1: 0.000e+00 | T: 3 (+ 0.354 1 - 0.338 z - 0.342 sin(z
)): 100%|██████████| 6000/6000 [04:37<00:00, 21.63it/s]
```

In [ ]:
```
fig, ax = plt.subplots(1, 5, figsize=(30, 4))

for i, loss_key in enumerate(['x', 'ddx', 'ddz', 'l1']):
    ax[i].plot(*np.array(loss_history[f'train_{loss_key}']).T, label=f'Train {loss_key}', color='tab:blue', alpl
    ax[i].plot(*np.array(loss_history[f'val_{loss_key}']).T, label=f'Validation {loss_key}', color='tab:orange'
    # ax[i].set_xscale('log')
    ax[i].set_yscale('log')
    ax[i].set_title(f'{loss_key} loss')

coef_epochs = np.array([c[0] for c in loss_history['coefficients']])
coef_values = np.array([c[1] for c in loss_history['coefficients']])

for coefficient, coef_name in enumerate(sindy_autoencoder.sindy.library.terms.keys()):
    ax[-1].plot(coef_epochs, coef_values[:, 0, coefficient], label=coef_name)

# ax[-1].set_xscale('log')
ax[-1].set_title('Coefficients')
ax[-1].legend(loc='upper right')
```

Out[ ]:
```
<matplotlib.legend.Legend at 0x7f5088a16d50>
```



In [ ]:
```
# Print the final equation
coefs = sindy_autoencoder.sindy.coef[0].detach().cpu().numpy()
coef_mask = sindy_autoencoder.sindy.coef_mask[0].detach().cpu().numpy()
terms_pred = sindy_autoencoder.sindy.library.terms

equation_string = " ".join([f"{'+ ' if coef >= 0 else '- '}{np.abs(coef):.3f} {term}" for coef, term, active in
print(equation_string)
```

```
+ 0.354 1 - 0.338 z - 0.342 sin(z)
```

In [ ]:
```
results = {
    'fvu_x': [],
    'fvu_ddx': [],
```

```python
    'fvu_ddz': [],
    'coefficients': [],
    'resimulation_z': [],
    'resimulation_x': [],
}

for i in range(N_REPEAT):
    x_train, dx_train, ddx_train, z_train, dz_train, ddz_train, t_train, x_val, dx_val, ddx_val, z_val, dz_val,

    lib = Library(['z', 'dz'], 1, terms_torch)
    thresholder = PatientTrendAwareThresholder(lib, threshold_a=0.1, threshold_b=0.002, patience=1000)
    sindy = SINDy(lib, thresholder, init="ones").to(device)

    sindy_autoencoder = SINDyAutoencoder(sindy, input_dim=2500, encoder_sizes=[128, 64, 32], decoder_sizes=[32,

    optimizer = Adam(sindy_autoencoder.parameters(), lr=1e-3)
    batch_size = 1000
    loss_history = {}

    train_sindy_autoencoder(loss_history, sindy_autoencoder, optimizer,
                            x_train.to(device), dx_train.to(device), ddx_train.to(device),
                            x_val.to(device), dx_val.to(device), ddx_val.to(device),
                            epochs=6000, refinement_after_epochs=5000,
                            l1_weight=1e-5 / batch_size, ddx_weight=5e-4, ddz_weight=5e-5,
                            batch_size=batch_size, verbose=True)

    # Compute the FVU on the test set
    sindy_autoencoder.sindy.eval()

    with torch.no_grad():
        x_hat, ddx_hat_rhs, ddz_lhs, ddz_rhs = sindy_autoencoder.forward(x_test.to(device), dx_test.to(device),

        fvu_x = compute_FVU(x_test.to(device), x_hat)
        fvu_ddx = compute_FVU(ddx_test.to(device), ddx_hat_rhs)
        fvu_ddz = compute_FVU(ddz_lhs, ddz_rhs)

        results['fvu_x'].append(fvu_x.item())
        results['fvu_ddx'].append(fvu_ddx.item())
        results['fvu_ddz'].append(fvu_ddz.item())

        coefficients = sindy_autoencoder.sindy.coef.detach().cpu().numpy().copy()
        results['coefficients'].append(coefficients)

        # Encode the first x of the test set to get the initial conditions
        z0, dz0, _ = sindy_autoencoder.encode(
            x_test.reshape(100, test_timesteps, 2500)[:, 0].to(device),
            dx_test.reshape(100, test_timesteps, 2500)[:, 0].to(device))
        z0 = z0.detach().cpu().numpy().copy()[:, 0]
        dz0 = dz0.detach().cpu().numpy().copy()[:, 0]

        # Resmuluate the system
        t, z, dz = simulate_pendulum(z0, dz0, coefficients[0], [terms_np[term] for term in terms_np], T * 5, DT
        ddz = pendulum_rhs(z, dz, coefficients[0], [terms_np[term] for term in terms_np])
        x, dx, ddx = sindy_autoencoder.decode(
            torch.tensor(z).float().view(-1, 1).to(device),
            torch.tensor(dz).float().view(-1, 1).to(device),
            torch.tensor(ddz).float().view(-1, 1).to(device)
        )
        x = x.detach().cpu().numpy().reshape(100, T * 5, 50, 50)
        dx = dx.detach().cpu().numpy().reshape(100, T * 5, 50, 50)
        ddx = ddx.detach().cpu().numpy().reshape(100, T * 5, 50, 50)

        # Remove the first and last elements to match the test set which is numerically differentiated
        z = z[:, 1:-1]
        dz = dz[:, 1:-1]
        ddz = ddz[:, 1:-1]
        x = x[:, 1:-1]
        dx = dx[:, 1:-1]
        ddx = ddx[:, 1:-1]

        # Save the resimulated x
        results['resimulation_z'].append(z)
        results['resimulation_x'].append(x)
```

```
T x: 6.651e-08 | T ddx: 3.210e-04 | T ddz: 4.323e-05 | T L1: 0.000e+00 | V x: 2.749e-08 | V ddx: 2.340e-04 | V d
dz: 2.879e-05 | V L1: 0.000e+00 | T: 3 (+ 0.375 1 - 0.344 z - 0.397 sin(z)): 100%|██████████| 6000/6000 [04:34<0
0:00, 21.88it/s]
T x: 2.118e-06 | T ddx: 2.273e-02 | T ddz: 3.868e-03 | T L1: 0.000e+00 | V x: 1.015e-06 | V ddx: 1.096e-02 | V d
dz: 1.608e-03 | V L1: 0.000e+00 | T: 4 (+ 0.507 1 - 1.641 z - 0.629 z^2 - 0.802 z sin(z)): 100%|██████████| 6000
/6000 [04:23<00:00, 22.80it/s]
T x: 2.541e-06 | T ddx: 3.169e-02 | T ddz: 7.611e-04 | T L1: 0.000e+00 | V x: 1.196e-06 | V ddx: 1.362e-02 | V d
dz: 3.946e-04 | V L1: 0.000e+00 | T: 2 (+ 1.155 1 - 2.067 z^2): 100%|██████████| 6000/6000 [04:19<00:00, 23.13it
/s]
/home/psaegert/miniconda3/envs/ignns/lib/python3.11/site-packages/scipy/integrate/_odepack_py.py:248: ODEintWarn
ing: Excess work done on this call (perhaps wrong Dfun type). Run with full_output = 1 to get quantitative infor
mation.
  warnings.warn(warning_msg, ODEintWarning)
T x: 3.011e-08 | T ddx: 3.625e-04 | T ddz: 5.221e-05 | T L1: 0.000e+00 | V x: 2.078e-09 | V ddx: 6.370e-05 | V d
dz: 9.946e-06 | V L1: 0.000e+00 | T: 2 (+ 0.421 1 - 0.583 z): 100%|██████████| 6000/6000 [04:20<00:00, 23.05it/s
]
T x: 4.573e-08 | T ddx: 5.970e-04 | T ddz: 5.308e-05 | T L1: 0.000e+00 | V x: 1.513e-07 | V ddx: 2.568e-04 | V d
dz: 8.051e-05 | V L1: 0.000e+00 | T: 1 (- 0.679 sin(z)): 100%|██████████| 6000/6000 [04:20<00:00, 22.99it/s]
T x: 2.627e-06 | T ddx: 2.933e-02 | T ddz: 1.989e-03 | T L1: 0.000e+00 | V x: 1.391e-06 | V ddx: 1.035e-02 | V d
dz: 6.260e-04 | V L1: 0.000e+00 | T: 4 (- 1.264 z - 0.808 sin(z) - 0.678 z^2 - 0.542 z sin(z)): 100%|██████████|
6000/6000 [04:19<00:00, 23.10it/s]
T x: 1.067e-08 | T ddx: 7.562e-05 | T ddz: 2.706e-05 | T L1: 0.000e+00 | V x: 4.105e-09 | V ddx: 2.254e-05 | V d
dz: 1.291e-05 | V L1: 0.000e+00 | T: 1 (- 0.710 sin(z)): 100%|██████████| 6000/6000 [04:20<00:00, 23.01it/s]
T x: 1.219e-07 | T ddx: 3.926e-04 | T ddz: 6.483e-05 | T L1: 0.000e+00 | V x: 1.863e-07 | V ddx: 3.523e-04 | V d
dz: 8.819e-05 | V L1: 0.000e+00 | T: 2 (+ 0.201 1 - 0.567 z): 100%|██████████| 6000/6000 [04:19<00:00, 23.10it/s
]
T x: 3.345e-07 | T ddx: 8.450e-04 | T ddz: 1.150e-04 | T L1: 0.000e+00 | V x: 6.255e-08 | V ddx: 3.517e-04 | V d
dz: 7.575e-05 | V L1: 0.000e+00 | T: 1 (- 0.366 z): 100%|██████████| 6000/6000 [04:20<00:00, 23.04it/s]
T x: 1.320e-08 | T ddx: 1.397e-04 | T ddz: 4.228e-05 | T L1: 0.000e+00 | V x: 2.594e-08 | V ddx: 6.163e-05 | V d
dz: 1.364e-04 | V L1: 0.000e+00 | T: 2 (+ 0.224 1 - 0.563 z): 100%|██████████| 6000/6000 [04:20<00:00, 23.02it/s
]
```

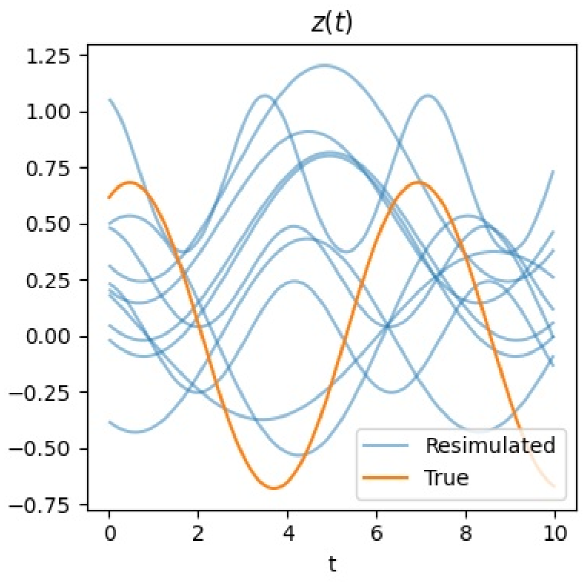
In [ ]:
```python
# Compute the mean and std of the FVU
results['fvu_x'] = np.array(results['fvu_x'])
results['fvu_ddx'] = np.array(results['fvu_ddx'])
results['fvu_ddz'] = np.array(results['fvu_ddz'])

print(f"FVU_x = {results['fvu_x'].mean():.4f} ± {results['fvu_x'].std():.4f}")
print(f"FVU_ddx = {results['fvu_ddx'].mean():.4f} ± {results['fvu_ddx'].std():.4f}")
print(f"FVU_ddz = {results['fvu_ddz'].mean():.4f} ± {results['fvu_ddz'].std():.4f}")
```

```
FVU_x = 0.1568 ± 0.2169
FVU_ddx = 0.4646 ± 0.7282
FVU_ddz = 0.4422 ± 0.3264
```

In [ ]:
```python
# Calculate the MSE between ground truth and resimulation at each timestep
resimulation_x = np.array(results['resimulation_x'])
resimulation_z = np.array(results['resimulation_z'])

print(f"{resimulation_x.shape = }")
print(f"{resimulation_z.shape = }")

resimulation_mse_x_median = np.median((resimulation_x - x_test.reshape(100, test_timesteps, 50, 50).cpu().numpy
resimulation_mse_x_quantiles = np.quantile((resimulation_x - x_test.reshape(100, test_timesteps, 50, 50).cpu().
resimulation_mse_z_median = np.median((resimulation_z - z_test.reshape(100, T * 5).cpu().numpy()[:, 1:-1])**2, 
resimulation_mse_z_quantiles = np.quantile((resimulation_z - z_test.reshape(100, T * 5).cpu().numpy()[:, 1:-1])
```

```
resimulation_x.shape = (10, 100, 498, 50, 50)
resimulation_z.shape = (10, 100, 498)
```

In [ ]:
```python
# Show a resimulated trajectory
fig, ax = plt.subplots(1, 1, figsize=(4, 4))

# z
for i in range(len(results['resimulation_x'])):
    ax.plot(t_test[1:-1], results['resimulation_z'][i][0], label='Resimulated' if i == 0 else None, alpha=0.5, 
ax.plot(t_test, z_test.reshape(100, T * 5, 1)[0, :, 0], label='True', color='tab:orange')
ax.set_xlabel('t')
ax.set_title('$z(t)$')
ax.legend()

fig.tight_layout()
```

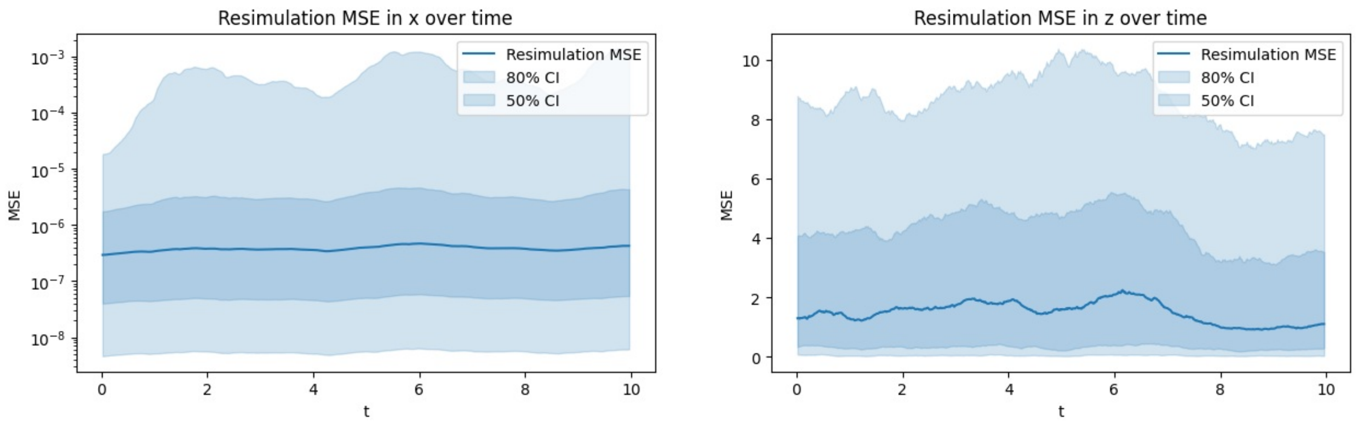


```
In [ ]: resimulation_mse_x_median.shape
```

```
Out[ ]: (498,)
```

```
In [ ]: # Show the resimulation error over time
        fig, axes = plt.subplots(1, 2, figsize=(15, 4))

        axes[0].plot(t[1:-1], resimulation_mse_x_median, label='Resimulation MSE', color='C0')
        axes[0].fill_between(t[1:-1], resimulation_mse_x_quantiles[0], resimulation_mse_x_quantiles[-1], alpha=0.2, labe
        axes[0].fill_between(t[1:-1], resimulation_mse_x_quantiles[1], resimulation_mse_x_quantiles[-2], alpha=0.2, labe
        axes[0].set_xlabel('t')
        axes[0].set_ylabel('MSE')
        axes[0].set_title('Resimulation MSE in x over time')
        axes[0].legend()
        axes[0].set_yscale('log')

        axes[1].plot(t[1:-1], resimulation_mse_z_median, label='Resimulation MSE', color='C0')
        axes[1].fill_between(t[1:-1], resimulation_mse_z_quantiles[0], resimulation_mse_z_quantiles[-1], alpha=0.2, labe
        axes[1].fill_between(t[1:-1], resimulation_mse_z_quantiles[1], resimulation_mse_z_quantiles[-2], alpha=0.2, labe
        axes[1].set_xlabel('t')
        axes[1].set_ylabel('MSE')
        axes[1].set_title('Resimulation MSE in z over time')
        axes[1].legend()
```

```
Out[ ]: <matplotlib.legend.Legend at 0x7f5072612590>
```



ST

```
In [ ]: x_train, dx_train, ddx_train, z_train, dz_train, ddz_train, t_train, x_val, dx_val, ddx_val, z_val, dz_val, ddz_
```

```
In [ ]: lib = Library(['z', 'dz'], 1, terms_torch)
        thresholder = SequentialThresholder(lib, threshold=0.1, interval=500)
        sindy = SINDy(lib, thresholder, init="ones").to(device)

        sindy_autoencoder = SINDyAutoencoder(sindy, input_dim=2500, encoder_sizes=[128, 64, 32], decoder_sizes=[32, 64,

        optimizer = Adam(sindy_autoencoder.parameters(), lr=1e-3)
```

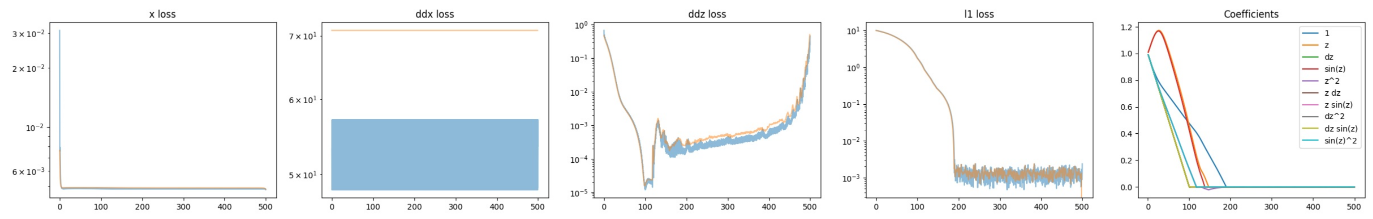
```
In [ ]: batch_size = 1000
        loss_history = {}
```

```
train_sindy_autoencoder(loss_history, sindy_autoencoder, optimizer,
                        x_train.to(device), dx_train.to(device), ddx_train.to(device),
                        x_val.to(device), dx_val.to(device), ddx_val.to(device),
                        epochs=6000, refinement_after_epochs=5000,
                        l1_weight=1e-5 / batch_size, ddx_weight=5e-4, ddz_weight=5e-5,
                        batch_size=batch_size, verbose=True)
```

```
  0%|          | 0/6000 [00:00<?, ?it/s]T x: 4.927e-06 | T ddx: 5.338e-02 | T ddz: 3.239e-04 | T L1: 1.423e-06 |
V x: 2.417e-06 | V ddx: 3.545e-02 | V ddz: 2.546e-04 | V L1: 0.000e+00 | T: 0 ():   8%|█         | 499/6000 [00:
21<03:57, 23.20it/s]
```

In [ ]:
```
fig, ax = plt.subplots(1, 5, figsize=(30, 4))

for i, loss_key in enumerate(['x', 'ddx', 'ddz', 'l1']):
    ax[i].plot(*np.array(loss_history[f'train_{loss_key}']).T, label=f'Train {loss_key}', color='tab:blue', alpl
    ax[i].plot(*np.array(loss_history[f'val_{loss_key}']).T, label=f'Validation {loss_key}', color='tab:orange'
    # ax[i].set_xscale('log')
    ax[i].set_yscale('log')
    ax[i].set_title(f'{loss_key} loss')

coef_epochs = np.array([c[0] for c in loss_history['coefficients']])
coef_values = np.array([c[1] for c in loss_history['coefficients']])

for coefficient, coef_name in enumerate(sindy_autoencoder.sindy.library.terms.keys()):
    ax[-1].plot(coef_epochs, coef_values[:, 0, coefficient], label=coef_name)

# ax[-1].set_xscale('log')
ax[-1].set_title('Coefficients')
ax[-1].legend(loc='upper right')
```

Out[ ]:
```
<matplotlib.legend.Legend at 0x7f9c76b70e90>
```



In [ ]:
```
# Print the final equation
coefs = sindy_autoencoder.sindy.coef[0].detach().cpu().numpy()
coef_mask = sindy_autoencoder.sindy.coef_mask[0].detach().cpu().numpy()
terms_pred = sindy_autoencoder.sindy.library.terms

equation_string = " ".join([f"{'+ ' if coef >= 0 else '- '}{np.abs(coef):.3f} {term}" for coef, term, active in
print(equation_string)
```

In [ ]:
```
results = {
    'fvu_x': [],
    'fvu_ddx': [],
    'fvu_ddz': [],
    'coefficients': [],
    'resimulation_z': [],
    'resimulation_x': [],
}

for i in range(N_REPEAT):
    x_train, dx_train, ddx_train, z_train, dz_train, ddz_train, t_train, x_val, dx_val, ddx_val, z_val, dz_val,

    lib = Library(['z', 'dz'], 1, terms_torch)
    thresholder = SequentialThresholder(lib, threshold=0.1, interval=500)
    sindy = SINDy(lib, thresholder, init="ones").to(device)

    sindy_autoencoder = SINDyAutoencoder(sindy, input_dim=2500, encoder_sizes=[128, 64, 32], decoder_sizes=[32,

    optimizer = Adam(sindy_autoencoder.parameters(), lr=1e-3)
    batch_size = 1000
    loss_history = {}

    train_sindy_autoencoder(loss_history, sindy_autoencoder, optimizer,
                            x_train.to(device), dx_train.to(device), ddx_train.to(device),
                            x_val.to(device), dx_val.to(device), ddx_val.to(device),
                            epochs=6000, refinement_after_epochs=5000,
                            l1_weight=1e-5 / batch_size, ddx_weight=5e-4, ddz_weight=5e-5,
                            batch_size=batch_size, verbose=True)

    # Compute the FVU on the test set
    sindy_autoencoder.sindy.eval()

    with torch.no_grad():
        x_hat, ddx_hat_rhs, ddz_lhs, ddz_rhs = sindy_autoencoder.forward(x_test.to(device), dx_test.to(device),

        fvu_x = compute_FVU(x_test.to(device), x_hat)
        fvu_ddx = compute_FVU(ddx_test.to(device), ddx_hat_rhs)
        fvu_ddz = compute_FVU(ddz_lhs, ddz_rhs)

        results['fvu_x'].append(fvu_x.item())
        results['fvu_ddx'].append(fvu_ddx.item())
        results['fvu_ddz'].append(fvu_ddz.item())

        coefficients = sindy_autoencoder.sindy.coef.detach().cpu().numpy().copy()
        results['coefficients'].append(coefficients)

        # Encode the first x of the test set to get the initial conditions
        z0, dz0, _ = sindy_autoencoder.encode(
            x_test.reshape(100, test_timesteps, 2500)[:, 0].to(device),
            dx_test.reshape(100, test_timesteps, 2500)[:, 0].to(device))
        z0 = z0.detach().cpu().numpy().copy()[:, 0]
        dz0 = dz0.detach().cpu().numpy().copy()[:, 0]

        # Resmuluate the system
        t, z, dz = simulate_pendulum(z0, dz0, coefficients[0], [terms_np[term] for term in terms_np], T * 5, DT
        ddz = pendulum_rhs(z, dz, coefficients[0], [terms_np[term] for term in terms_np])
        x, dx, ddx = sindy_autoencoder.decode(
            torch.tensor(z).float().view(-1, 1).to(device),
            torch.tensor(dz).float().view(-1, 1).to(device),
            torch.tensor(ddz).float().view(-1, 1).to(device)
        )
        x = x.detach().cpu().numpy().reshape(100, T * 5, 50, 50)
        dx = dx.detach().cpu().numpy().reshape(100, T * 5, 50, 50)
        ddx = ddx.detach().cpu().numpy().reshape(100, T * 5, 50, 50)

        # Remove the first and last z to match the grid
        z = z[:, 1:-1]
        dz = dz[:, 1:-1]
        ddz = ddz[:, 1:-1]
        x = x[:, 1:-1]
        dx = dx[:, 1:-1]
        ddx = ddx[:, 1:-1]

        # Save the resimulated x
        results['resimulation_z'].append(z)
        results['resimulation_x'].append(x)
```

```
T x: 4.445e-06 | T ddx: 6.054e-02 | T ddz: 3.942e-04 | T L1: 3.317e-06 | V x: 2.269e-06 | V ddx: 2.810e-02 | V d
dz: 1.863e-04 | V L1: 0.000e+00 | T: 0 ():   8%|▊         | 499/6000 [00:21<03:57, 23.12it/s]
T x: 4.465e-06 | T ddx: 6.562e-02 | T ddz: 2.657e-04 | T L1: 2.105e-06 | V x: 2.335e-06 | V ddx: 3.398e-02 | V d
dz: 1.194e-04 | V L1: 0.000e+00 | T: 0 ():   8%|▊         | 499/6000 [00:20<03:44, 24.46it/s]
T x: 4.462e-06 | T ddx: 5.435e-02 | T ddz: 2.857e-04 | T L1: 2.756e-05 | V x: 2.234e-06 | V ddx: 4.096e-02 | V d
dz: 2.111e-04 | V L1: 0.000e+00 | T: 0 ():   8%|▊         | 499/6000 [00:20<03:45, 24.35it/s]
T x: 4.518e-06 | T ddx: 6.451e-02 | T ddz: 2.668e-04 | T L1: 3.480e-05 | V x: 2.285e-06 | V ddx: 3.625e-02 | V d
dz: 1.414e-04 | V L1: 0.000e+00 | T: 0 ():   8%|▊         | 499/6000 [00:20<03:44, 24.45it/s]
T x: 4.510e-06 | T ddx: 7.896e-02 | T ddz: 2.437e-04 | T L1: 1.242e-06 | V x: 2.162e-06 | V ddx: 4.250e-02 | V d
dz: 1.395e-04 | V L1: 0.000e+00 | T: 0 ():   8%|▊         | 499/6000 [00:21<03:52, 23.61it/s]
T x: 4.595e-06 | T ddx: 6.030e-02 | T ddz: 4.550e-04 | T L1: 1.223e-04 | V x: 2.301e-06 | V ddx: 2.016e-02 | V d
dz: 1.699e-04 | V L1: 0.000e+00 | T: 0 ():   8%|▊         | 499/6000 [00:21<03:51, 23.75it/s]
T x: 4.516e-06 | T ddx: 6.189e-02 | T ddz: 1.946e-04 | T L1: 1.869e-06 | V x: 2.258e-06 | V ddx: 3.389e-02 | V d
dz: 1.155e-04 | V L1: 0.000e+00 | T: 0 ():   8%|▊         | 499/6000 [00:20<03:49, 23.93it/s]
T x: 3.906e-08 | T ddx: 1.418e-04 | T ddz: 3.670e-05 | T L1: 0.000e+00 | V x: 1.164e-08 | V ddx: 7.576e-05 | V d
dz: 2.887e-05 | V L1: 0.000e+00 | T: 1 (- 0.685 sin(z)): 100%|██████████| 6000/6000 [04:11<00:00, 23.83it/s]
T x: 1.566e-08 | T ddx: 1.861e-04 | T ddz: 4.113e-05 | T L1: 0.000e+00 | V x: 5.849e-08 | V ddx: 9.863e-05 | V d
dz: 7.304e-05 | V L1: 0.000e+00 | T: 2 (+ 0.236 1 - 0.591 z): 100%|██████████| 6000/6000 [04:18<00:00, 23.18it/s
]
T x: 4.939e-06 | T ddx: 7.479e-02 | T ddz: 6.947e-06 | T L1: 1.170e-06 | V x: 2.467e-06 | V ddx: 1.969e-02 | V d
dz: 3.983e-06 | V L1: 0.000e+00 | T: 0 ():   8%|▊         | 499/6000 [00:21<03:53, 23.56it/s]
```

In [ ]:
```
# Compute the mean and std of the FVU
results['fvu_x'] = np.array(results['fvu_x'])
results['fvu_ddx'] = np.array(results['fvu_ddx'])
results['fvu_ddz'] = np.array(results['fvu_ddz'])

print(f"FVU_x = {results['fvu_x'].mean():.4f} ± {results['fvu_x'].std():.4f}")
print(f"FVU_ddx = {results['fvu_ddx'].mean():.4f} ± {results['fvu_ddx'].std():.4f}")
print(f"FVU_ddz = {results['fvu_ddz'].mean():.4f} ± {results['fvu_ddz'].std():.4f}")
```

```
FVU_x = 0.7063 ± 0.3397
FVU_ddx = 0.8019 ± 0.3960
FVU_ddz = 0.8758 ± 0.2489
```

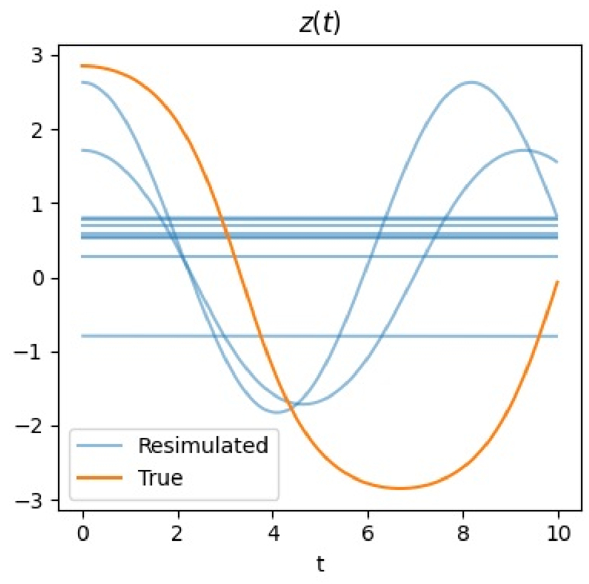
In [ ]:
```
# Calculate the MSE between ground truth and resimulation at each timestep
resimulation_x = np.array(results['resimulation_x'])
resimulation_z = np.array(results['resimulation_z'])
```

```
print(f"{resimulation_x.shape = }")
print(f"{resimulation_z.shape = }")

resimulation_mse_x_median = np.median((resimulation_x - x_test.reshape(100, test_timesteps, 50, 50).cpu().numpy
resimulation_mse_x_quantiles = np.quantile((resimulation_x - x_test.reshape(100, test_timesteps, 50, 50).cpu().
resimulation_mse_z_median = np.median((resimulation_z - z_test.reshape(100, T * 5).cpu().numpy()[:, 1:-1])**2,
resimulation_mse_z_quantiles = np.quantile((resimulation_z - z_test.reshape(100, T * 5).cpu().numpy()[:, 1:-1])
```

```
resimulation_x.shape = (10, 100, 498, 50, 50)
resimulation_z.shape = (10, 100, 498)
```

In [ ]:

```
# Show a resimulated trajectory
fig, ax = plt.subplots(1, 1, figsize=(4, 4))

# z
for i in range(len(results['resimulation_x'])):
    ax.plot(t_test[1:-1], results['resimulation_z'][i][0], label='Resimulated' if i == 0 else None, alpha=0.5,
ax.plot(t_test, z_test.reshape(100, T * 5, 1)[0, :, 0], label='True', color='tab:orange')
ax.set_xlabel('t')
ax.set_title('$z(t)$')
ax.legend()

fig.tight_layout()
```

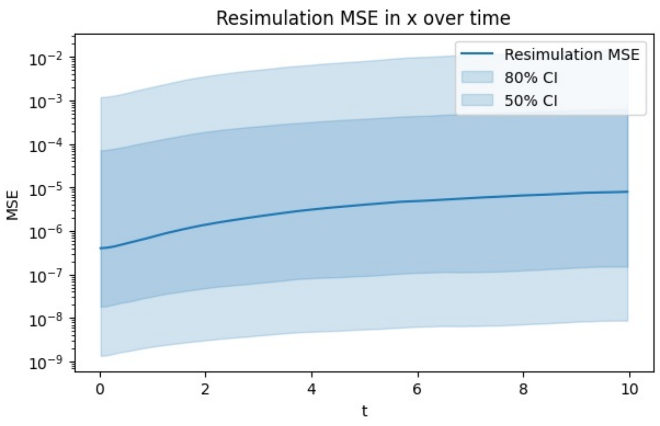


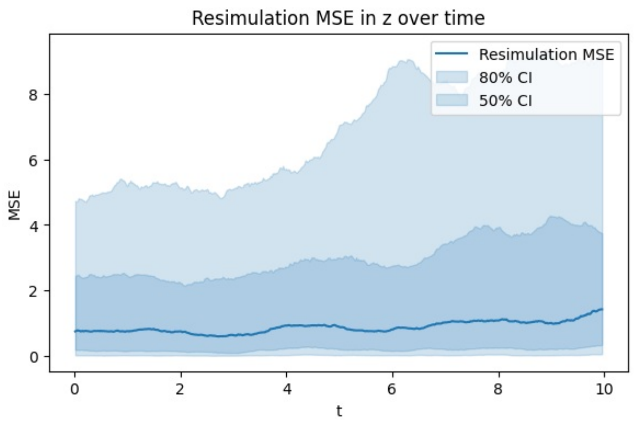
In [ ]:

```
# Show the resimulation error over time
fig, axes = plt.subplots(1, 2, figsize=(15, 4))

axes[0].plot(t_test[1:-1], resimulation_mse_x_median, label='Resimulation MSE', color='C0')
axes[0].fill_between(t_test[1:-1], resimulation_mse_x_quantiles[0], resimulation_mse_x_quantiles[-1], alpha=0.2
axes[0].fill_between(t_test[1:-1], resimulation_mse_x_quantiles[1], resimulation_mse_x_quantiles[-2], alpha=0.2
axes[0].set_xlabel('t')
axes[0].set_ylabel('MSE')
axes[0].set_title('Resimulation MSE in x over time')
axes[0].legend()
axes[0].set_yscale('log')

axes[1].plot(t_test[1:-1], resimulation_mse_z_median, label='Resimulation MSE', color='C0')
axes[1].fill_between(t_test[1:-1], resimulation_mse_z_quantiles[0], resimulation_mse_z_quantiles[-1], alpha=0.2
axes[1].fill_between(t_test[1:-1], resimulation_mse_z_quantiles[1], resimulation_mse_z_quantiles[-2], alpha=0.2
axes[1].set_xlabel('t')
axes[1].set_ylabel('MSE')
axes[1].set_title('Resimulation MSE in z over time')
axes[1].legend()
```

Out[ ]:

```
<matplotlib.legend.Legend at 0x7f9c76d48250>
```


Resimulation MSE in x over time
Resimulation MSE
80% CI
50% CI
MSE
t


Resimulation MSE in z over time
Resimulation MSE
80% CI
50% CI
MSE
t

```
In [ ]:
```