

## A THE ASSOCIATED LIE OPERATOR OF THE KOOPMAN OPERATOR

Besides the linear system in Eq. (9)) in the main text, we can also consider the generator operator of such a Koopman operator, which is referred to as the Lie operator because it is the Lie derivative of  $\mathbf{g}(\cdot)$  along the vector field  $\gamma(\cdot)$  Koopman (1931); Abraham et al. (2012); Chicone & Latushkin (1999)

$$\mathcal{L}_t \mathbf{g} = \lim_{t+\varepsilon \rightarrow t} \frac{\mathcal{K}_t^{t+\varepsilon} \mathbf{g}(\gamma_t) - \mathbf{g}(\gamma_t)}{t + \varepsilon - t}. \quad (21)$$

The generator operator also defines a linear system of  $\mathbf{g}(\gamma_t)$  because

$$\partial_t \mathbf{g}(\gamma_t) = \lim_{\varepsilon \rightarrow 0} \frac{\mathbf{g}(\gamma_{t+\varepsilon}) - \mathbf{g}(\gamma_t)}{\varepsilon} = \lim_{t+\varepsilon \rightarrow t} \frac{\mathcal{K}_t^{t+\varepsilon} \mathbf{g}(\gamma_t) - \mathbf{g}(\gamma_t)}{\varepsilon} = \mathcal{L}_t \mathbf{g}(\gamma_t). \quad (22)$$

Although our work primarily focuses on the Koopman operator, one can also model the Lie operator in application.

## B MATHEMATICAL DETAILS OF IMPLEMENTED PDES

In our experiments, we consider the 1-dimensional Bateman–Burgers equation Benton & Platzman (1972) and the 2-dimensional Navier-Stokes equation Wang (1991). Below, we introduce their mathematical definitions.

**Bateman–Burgers equation.** The 1-dimensional Bateman–Burgers equation is

$$\partial_t \gamma(x_t) + \partial_x \left( \frac{\gamma^2(x_t)}{2} \right) = \nu \partial_{xx} \gamma(x_t), \quad x_t \in (0, 1) \times (0, 1], \quad x_t \in (0, 1) \times (0, 1], \quad (23)$$

$$\gamma(x_0) = \gamma_I, \quad x_0 \in (0, 1) \times \{0\}, \quad (24)$$

where  $\gamma_I$  is a periodic initial condition  $\gamma_I \in L^2_{\text{periodic}}[(0, 1); \mathbb{R}]$ , parameter  $\nu \in (0, \infty)$  is the viscosity coefficient, which is set as  $\nu = 100$  in our experiments. The data set of Eqs. (23-24) is provided by Li et al. (2020a). Please see Li et al. (2020a) for details of data generation.

**Navier-Stokes equation.** Mathematically, the incompressible 2-dimensional Navier-Stokes equation in a vorticity form is defined as

$$\partial_t \gamma(x_t) + \chi(x_t) \nabla \gamma(x_t) = \nu \Delta \gamma(x_t) + \psi(x_t), \quad x_t \in (0, 1)^2 \times (0, \infty), \quad (25)$$

$$\nabla \chi(x_t) = 0, \quad x_t \in (0, 1)^2 \times (0, \infty), \quad (26)$$

$$\gamma(x_0) = \gamma_I, \quad x_0 \in (0, 1) \times \{0\}, \quad (27)$$

where  $\gamma(\cdot)$  measures the vorticity,  $\chi(\cdot)$  defines the velocity,  $\psi(\cdot)$  denotes a time-independent forcing term. The viscosity coefficient is set as  $\nu \in \{10^{-3}, 10^{-4}\}$ . Similar to the situation of Bateman–Burgers equation, the data of Eqs. (25-27) is provided by Li et al. (2020a). Please see Li et al. (2020a) for details.

## C ABLATION EXPERIMENT RESULTS

Our objective in the ablation experiment is to compare between the performance of KNO models with ( $\beta > 0$ ) and without ( $\beta = 0$ ) the reconstruction term of loss function in Eq. (20). When  $\beta > 0$ , the prediction is undertaken by a learned Koopman operator (a linear layer of  $o \times o$ ) associated with a convolutional layer in **Parts 2-5** while encoder (**Part 1**) and decoder (**Part 6**) contribute to reconstruction. In such a case, the performance is mainly contributed by the Koopman operator because the convolutional layer only serves as the complement of high-frequency information (as suggested by Li et al. (2020a), a pure convolutional network only achieves poor performance in PDE solving). Once  $\beta = 0$ , encoder and decoder become to be trained for prediction as well, making the whole network a unified predictor (similar to FNO Li et al. (2020a)). The Koopman operator is validated as important if the prediction performance mainly realized by it ( $\beta > 0$ ) is same as or better than the performance achieved by the whole network as a unified predictor ( $\beta = 0$ ). The ablation experiment is implemented as a 1-second prediction task on the data of 1-dimensional Bateman–Burgers equation with  $2^8$  grids. As shown in **Table 2**, KNO models generally perform better when  $\beta > 0$ , suggesting the significance of the Koopman operator.

Operator size $o$	Mode number $f$	Iteration number $r$	$\alpha$	$\beta$	MSE
8	64	10	5.0	0.5	$2.43 \times 10^{-5}$
8	64	10	5.0	0	$2.67 \times 10^{-5}$
16	10	16	5.0	0.5	$1.05 \times 10^{-5}$
16	10	16	5.0	0	$1.10 \times 10^{-5}$
32	10	16	5.0	0.5	$5.37 \times 10^{-6}$
32	10	16	5.0	0	$6.04 \times 10^{-6}$
32	64	16	5.0	0.5	$5.47 \times 10^{-6}$
32	64	16	5.0	0	$5.78 \times 10^{-6}$
128	10	16	5.0	0.5	$3.53 \times 10^{-6}$
128	10	16	5.0	0	$4.34 \times 10^{-6}$

Table 4: Results of ablation experiment.

## D CODE

The basic implementation of the Koopman neural operator is demonstrated here. The full version of our code will be released once the double-blind review finishes.

```

import torch
import numpy as np
import torch.nn as nn
import torch.nn.functional as F

torch.manual_seed(0)

class encoder(nn.Module):
    def __init__(self, t_len, op_size):
        super(encoder, self).__init__()
        self.layer = nn.Linear(t_len, op_size)
    def forward(self, x):
        x = self.layer(x)
        x = torch.tanh(x)
        return x

class decoder(nn.Module):
    def __init__(self, t_len, op_size):
        super(decoder, self).__init__()
        self.layer = nn.Linear(op_size, t_len)
    def forward(self, x):
        x = torch.tanh(x)
        x = self.layer(x)
        return x

class Koopman_Operator1D(nn.Module):
    def __init__(self, op_size, modes_x = 16):
        super(Koopman_Operator1D, self).__init__()
        self.op_size = op_size
        self.scale = (1 / (op_size * op_size))
        self.modes_x = modes_x
        self.koopman_matrix = nn.Parameter(self.scale * torch.rand(
            (op_size, op_size, self.modes_x, dtype=torch.cfloat)))
    # Complex multiplication
    def time_marching(self, input, weights):
        # (batch, t, x), (t, t+1, x) -> (batch, t+1, x)
        return torch.einsum("btx, tfx->bfx", input, weights)
    def forward(self, x):

```

```

        batchsize = x.shape[0]
        # Fourier Transform
        x_ft = torch.fft.rfft(x)
        # Koopman Operator Time Marching
        out_ft = torch.zeros(x_ft.shape, dtype=torch.cfloat,
                             device = x.device)
        out_ft[:, :, :self.modes_x] = self.time_marching(x_ft[:,
                                                         :, :self.modes_x], self.koopman_matrix)
        # Inverse Fourier Transform
        x = torch.fft.irfft(out_ft, n=x.size(-1))
        return x

class KNOld(nn.Module):
    def __init__(self, op_size, modes_x = 16, decompose = 4, t_len
                 = 1):
        super(KNOld, self).__init__()
        # Parameter
        self.op_size = op_size
        self.decompose = decompose
        # Layer Structure
        self.enc = encoder(t_len, op_size)
        self.dec = decoder(t_len, op_size)
        self.koopman_layer = Koopman.Operator1D(self.op_size,
                                                modes_x = modes_x)
        self.w0 = nn.Conv1d(op_size, op_size, 1)
    def forward(self, x):
        # Reconstruct
        x_reconstruct = self.enc(x)
        x_reconstruct = torch.tanh(x_reconstruct)
        x_reconstruct = self.dec(x_reconstruct)
        # Predict
        x = self.enc(x) # Encoder
        x = x.permute(0, 2, 1)
        x_w = x
        for i in range(self.decompose):
            x1 = self.koopman_layer(x) # Koopman Operator
            x = torch.tanh(x + x1)
            #  $x = x + x1$ 
        x = self.w0(x_w) + x
        x = x.permute(0, 2, 1)
        x = self.dec(x) # Decoder
        return x, x_reconstruct

class Koopman.Operator2D(nn.Module):
    def __init__(self, op_size, modes):
        super(Koopman.Operator2D, self).__init__()
        self.op_size = op_size
        self.scale = (1 / (op_size * op_size))
        self.modes_x = modes
        self.modes_y = modes
        self.koopman_matrix = nn.Parameter(self.scale * torch.rand(
            op_size, op_size, self.modes_x, self.modes_y, dtype=
            torch.cfloat))

    # Complex multiplication
    def time_marching(self, input, weights):
        #  $(batch, t, x, y), (t, t+1, x, y) \rightarrow (batch, t+1, x, y)$ 
        return torch.einsum("btxy, txy->btxy", input, weights)

```

```

def forward(self, x):
    batchsize = x.shape[0]
    # Fourier Transform
    x_ft = torch.fft.rfft2(x)
    # Koopman Operator Time Marching
    out_ft = torch.zeros(x_ft.shape, dtype=torch.cfloat,
                        device = x.device)
    out_ft[:, :, :self.modes_x, :self.modes_y] = self.
        time_marching(x_ft[:, :, :self.modes_x, :self.modes_y]
                    ], self.koopman_matrix)
    out_ft[:, :, -self.modes_x:, :self.modes_y] = self.
        time_marching(x_ft[:, :, -self.modes_x:, :self.modes_y]
                    ], self.koopman_matrix)
    # Inverse Fourier Transform
    x = torch.fft.irfft2(out_ft, s=(x.size(-2), x.size(-1)))
    return x

class KNO2d(nn.Module):
    def __init__(self, op_size, modes = 10, decompose = 6, t_len =
        10):
        super(KNO2d, self).__init__()
        # Parameter
        self.op_size = op_size
        self.decompose = decompose
        self.modes = modes
        # Layer Structure
        self.enc = encoder(t_len, op_size)
        self.dec = decoder(t_len, op_size)
        self.koopman_layer = Koopman.Operator2D(self.op_size, self
            .modes)
        self.w0 = nn.Conv2d(op_size, op_size, 1)
    def forward(self, x):
        # Reconstruct
        x_reconstruct = self.enc(x)
        x_reconstruct = torch.tanh(x_reconstruct)
        x_reconstruct = self.dec(x_reconstruct)
        # Predict
        x = self.enc(x) # Encoder
        x = x.permute(0, 3, 1, 2)
        x_w = x
        for i in range(self.decompose):
            x1 = self.koopman_layer(x) # Koopman Operator
            x = torch.tanh(x + x1)
        x = self.w0(x_w) + x
        x = x.permute(0, 2, 3, 1)
        x = self.dec(x) # Decoder
        return x, x_reconstruct

```