ENGR 520 Homework 3

Anthony Su

May 13, 2025

Collaboration statement: I did not collaborate with others on this assignment. I utilized large language models for code debugging.

1

This section discusses modeling the Lotka-Volterra (L-V) Predator-Prey system using sparse identification of nonlinear dynamics (SINDy). The L-V system is defined as:

$$\dot{x}_1 = \alpha x_1 - \beta x_1 x_2$$
$$\dot{x}_2 = \delta x_1 x_2 - \gamma x_2$$

A SINDy model is found by solving the following linear system:

$$\dot{\mathbf{X}} = \mathbf{\Theta}(\mathbf{X})\mathbf{\Xi}$$

where $\mathbf{X}$ are the data, $\mathbf{\Theta}$ are the library of candidate functions, and $\mathbf{\Xi}$ are the coefficients of $\mathbf{\Theta}$. In this report, $\mathbf{\Theta}$ are polynomial functions and the solution is found via sequentially-thresholded least-squares (STLS).

1.a

Figure 1 shows the phase portrait of this model as it is simulated on the interval $t = [0, 10]$.

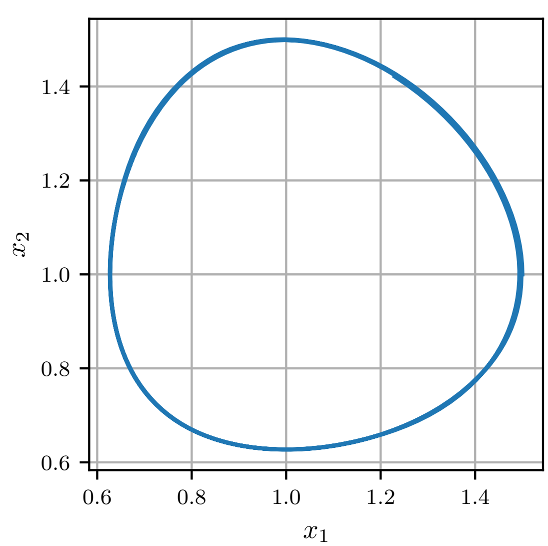


Figure 1: Lotka-Volterra Model Phase Portrait

1.b

Table 1 shows the results of SINDy when using a 2nd-order polynomial library for $\mathbf{\Theta}$.

Table 1: SINDy using STLS optimizer and 2nd-order polynomial library

Threshold	Model
0.20	$\dot{x}_1 = 2.001x_1 - 2.001x_1x_2$ $\dot{x}_2 = -2.002x_2 + 2.002x_1x_2$
0.19	$\dot{x}_1 = 2.001x_1 - 2.001x_1x_2$ $\dot{x}_2 = -2.002x_2 + 2.002x_1x_2$
0.18	$\dot{x}_1 = 2.001x_1 - 2.001x_1x_2$ $\dot{x}_2 = -2.002x_2 + 2.002x_1x_2$
0.17	$\dot{x}_1 = 2.001x_1 - 2.001x_1x_2$ $\dot{x}_2 = -2.002x_2 + 2.002x_1x_2$
0.16	$\dot{x}_1 = 2.001x_1 - 2.001x_1x_2$ $\dot{x}_2 = 0.408 - 0.408x_1 - 2.450x_2 + 0.197x_1^2 + 1.998x_1x_2 + 0.216x_2^2$
0.15	$\dot{x}_1 = -0.381 + 2.374x_1 + 0.427x_2 - 0.182x_1^2 - 1.996x_1x_2 - 0.206x_2^2$ $\dot{x}_2 = 0.408 - 0.408x_1 - 2.450x_2 + 0.197x_1^2 + 1.998x_1x_2 + 0.216x_2^2$

1.c

For thresholds above 0.16, the discovered Ξ matches well with the ground truth. For this system's sparse dynamics, a higher threshold performs well. For lower thresholds, some extra terms erroneously are included. As the threshold is modulated, the model's sparsity pattern changes in discrete and sudden jumps.

1.d

Table 2 shows the SINDy models of the L-V system when using a 3rd-order polynomial library for $\mathbf{\Theta}$.

Table 2: SINDy L-V Models

Threshold	Model
0.20	$\dot{x}_1 = 2.001x_1 - 2.001x_1x_2$ $\dot{x}_2 = -2.002x_2 + 2.002x_1x_2$
0.19	$\dot{x}_1 = 2.001x_1 - 2.001x_1x_2$ $\dot{x}_2 = -2.002x_2 + 2.002x_1x_2$
0.18	$\dot{x}_1 = 2.001x_1 - 2.001x_1x_2$ $\dot{x}_2 = -2.002x_2 + 2.002x_1x_2$
0.17	$\dot{x}_1 = 2.001x_1 - 2.001x_1x_2$ $\dot{x}_2 = -2.002x_2 + 2.002x_1x_2$
0.16	$\dot{x}_1 = 2.001x_1 - 2.001x_1x_2$ $\dot{x}_2 = 0.408 - 0.408x_1 - 2.450x_2 + 0.197x_1^2 + 1.998x_1x_2 + 0.216x_2^2$
0.15	$\dot{x}_1 = -0.381 + 2.374x_1 + 0.427x_2 - 0.182x_1^2 - 1.996x_1x_2 - 0.206x_2^2$ $\dot{x}_2 = 0.408 - 0.408x_1 - 2.450x_2 + 0.197x_1^2 + 1.998x_1x_2 + 0.216x_2^2$

The results exactly match those from the 2nd-order polynomial solution. Thus, it was able to converge to the correct solution at higher thresholds. This implies that the system has no significant cubic-like behavior for the SINDy model to fit to.

1.e

Figure 2 shows the trajectory of this model as it is simulated on the interval $t = [0, 10]$ with a small amount of noise added.

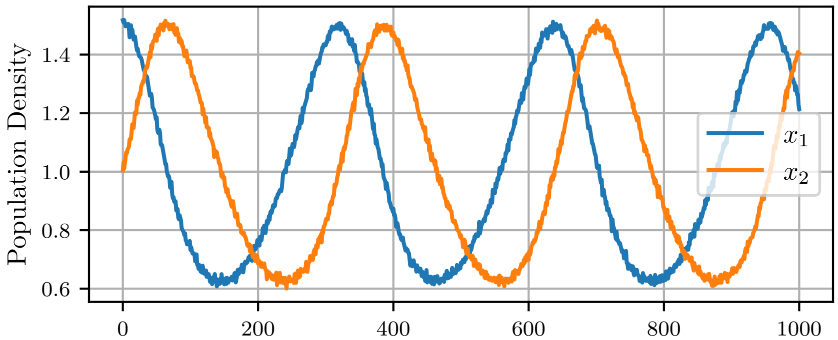


Figure 2: Trajectory with $\sigma = 0.01$ noise

Table 3 shows the SINDy models of the L-V system from noisy data when using a 3rd-order polynomial library for $\mathbf{\Theta}$.

Table 3: SINDy L-V Models from noisy data

Threshold	Model
0.20	$\dot{x}_1 = 1.996x_1 - 2.003x_1x_2$ $\dot{x}_2 = -2.013x_2 + 2.015x_1x_2$
0.19	$\dot{x}_1 = 1.996x_1 - 2.003x_1x_2$ $\dot{x}_2 = -2.013x_2 + 2.015x_1x_2$
0.18	$\dot{x}_1 = 0.106 + 1.939x_1 - 0.136x_2 - 0.001x_1^2 - 1.959x_1x_2 + 0.044x_2^2$ $\dot{x}_2 = -2.013x_2 + 2.015x_1x_2$
0.17	$\dot{x}_1 = 0.106 + 1.939x_1 - 0.136x_2 - 0.001x_1^2 - 1.959x_1x_2 + 0.044x_2^2$ $\dot{x}_2 = -2.013x_2 + 2.015x_1x_2$
0.16	$\dot{x}_1 = 0.106 + 1.939x_1 - 0.136x_2 - 0.001x_1^2 - 1.959x_1x_2 + 0.044x_2^2$ $\dot{x}_2 = -2.013x_2 + 2.015x_1x_2$
0.15	$\dot{x}_1 = 0.106 + 1.939x_1 - 0.136x_2 - 0.001x_1^2 - 1.959x_1x_2 + 0.044x_2^2$ $\dot{x}_2 = -2.013x_2 + 2.015x_1x_2$

The noise increased the threshold required to obtain the correct model. It seems that SINDy is prone to overfitting in the presence of noise.

2

This section discusses modeling the Lorenz system using SINDy. The Lorenz system is defined as:

$$\begin{aligned}\dot{x} &= \sigma(y - x)\\ \dot{y} &= x(\rho - z) - y\\ \dot{z} &= xy - \beta z\end{aligned}$$

where $\sigma = 10$, $\rho = 28$, and $\beta = 8/3$.

2.a

Table 4 shows the SINDy models of the Lorenz system when using a 2nd-order polynomial library for $\mathbf{\Theta}$ and a SLTS threshold of 0.2.

Table 4: SINDy Lorenz models

t_{end}	Model
0.5	$\dot{x} = -10.003x + 10.003y$ $\dot{y} = 0.2471 + 28.081x + -1.021y + -1.000xz$ $\dot{z} = -55.5971 + 36.681x + -20.192y + 0.462z + -1.891x^2 + 1.803xy + -1.312xz + 0.498yz$
1.0	$\dot{x} = -9.992x + 9.998y$ $\dot{y} = 0.2521 + 28.289x + -1.095y + -1.005xz$ $\dot{z} = -2.662z + 0.999xy$
1.5	$\dot{x} = -10.006x + 10.007y$ $\dot{y} = 9.2411 + 3.013x + 11.052y + -0.732z + 1.443x^2 + -0.684xy + -0.346yz$ $\dot{z} = -2.665z + 1.000xy$
2.0	$\dot{x} = -10.005x + 10.005y$ $\dot{y} = 27.782x + -0.960y + -0.992xz$ $\dot{z} = -2.665z + 0.999xy$

2.b

Data equivalent to that in Table 4 was generated for several data sampling rates. These models were compared to the true equations of motion. Table 5 shows which models had the same polynomial terms as the true equations. In other words, Table 5 shows which models have the same sparsity matrix as the true model.

Table 5: Sampling rates where SINDy recovers the true sparsity matrix

		Δt			
		0.0001	0.001	0.01	0.1
t_{end}	0.5				
	0.5				
	1.0				
	2.0	✓	✓		
	4.0	✓	✓	✓	
	9.0	✓	✓	✓	

Reducing sampling rate reduces the accuracy of the SINDy model.

2.c

Data equivalent to that in Table 4 was generated for many magnitudes of noise added to the training data. These models were compared to the true equations of motion. Figure 3 shows the probability that models have the same sparsity matrix as the true model subject to various magnitudes of noise in the training data.

Note that the sampling rate in this study is the original $\Delta t = 0.0001$.

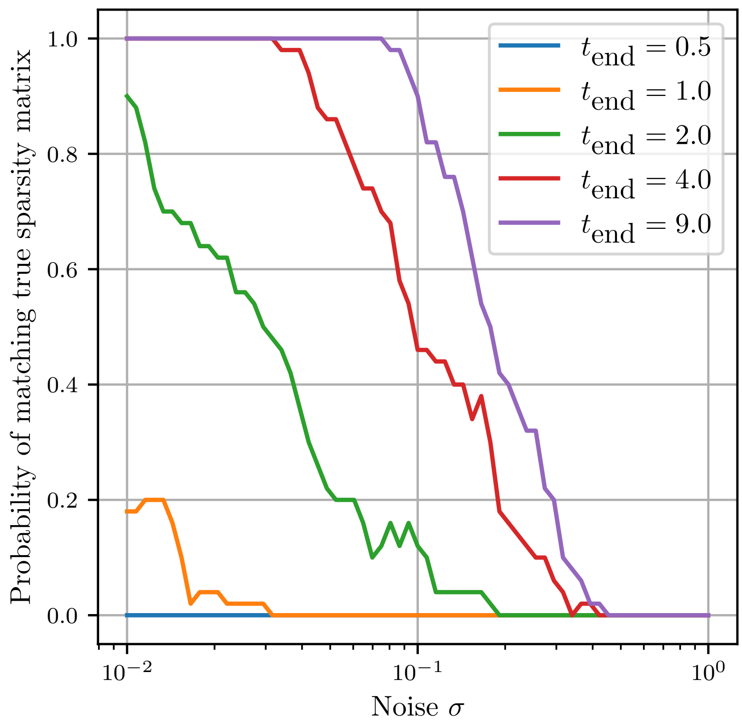


Figure 3: SINDy performance vs. noise magnitude of various dataset lengths

Increasing the data noise reduces the accuracy of the SINDy model.

3

3.b

Functions for forward difference numerical differentiation and central difference numerical differentiation are shown in Listings 1 and 2.

```python
def forward_difference(x: ArrayLike, t: ArrayLike=np.arange(len(x))) -> NDArray:
    """Forward difference numerical differentiation

    Args:
        x (ArrayLike): independent variable
        t (ArrayLike, optional): dependent variable. Defaults to np.arange(len(x)).

    Returns:
        NDArray: derivative of independent variable
    """
    dxdt = np.empty_like(x)
    dxdt[:-1] = (x[1:] - x[:-1]) / (t[1:] - t[:-1]).reshape(-1, 1)
    dxdt[-1] = dxdt[-2]
    return dxdt
```

Listing 1: Forward difference function

```python
def central_difference(x: ArrayLike, t: ArrayLike=np.arange(len(x))) -> NDArray:
    """Central difference numerical differentiation

    Args:
        x (ArrayLike): independent variable
        t (ArrayLike, optional): dependent variable. Defaults to np.arange(len(x)).

    Returns:
        NDArray: derivative of independent variable
    """
    dxdt = np.empty_like(x)
    dxdt[1:-1] = (x[2:] - x[:-2]) / (t[2:] - t[:-2]).reshape(-1, 1)
    dxdt[0] = dxdt[1]
    dxdt[-1] = dxdt[-2]
    return dxdt
```

Listing 2: Central difference function

3.c

Table 6: SINDy models using various differentiation schemes

t_{end}	Model
Forward difference	$\dot{x} = -9.992x + 9.998y$ $\dot{y} = 0.2521 + 28.289x + -1.095y + -1.005xz$ $\dot{z} = -2.662z + 0.999xy$
Central difference	$\dot{x} = -9.985x + 9.985y$ $\dot{y} = 27.592x + -0.916y + -0.987xz$ $\dot{z} = -2.660z + 0.997xy$
Smooth Difference	$\dot{x} = -9.971x + 9.971y$ $\dot{y} = 27.420x + -0.878y + -0.983xz$ $\dot{z} = -2.655z + 0.995xy$

Using central difference and smooth difference schemes, SINDy is able to converge to the correct model from the data. However, the model constants/coefficients are not as precise due to the noise.

3.d

For each of 65 noise magnitudes between 0.01 and 1, I generated 50 instantiations of noisy data. I fitted a SINDy model to each noisy dataset. The success rate of these models in matching the sparsity matrix of the true system are shown in Figure 4.

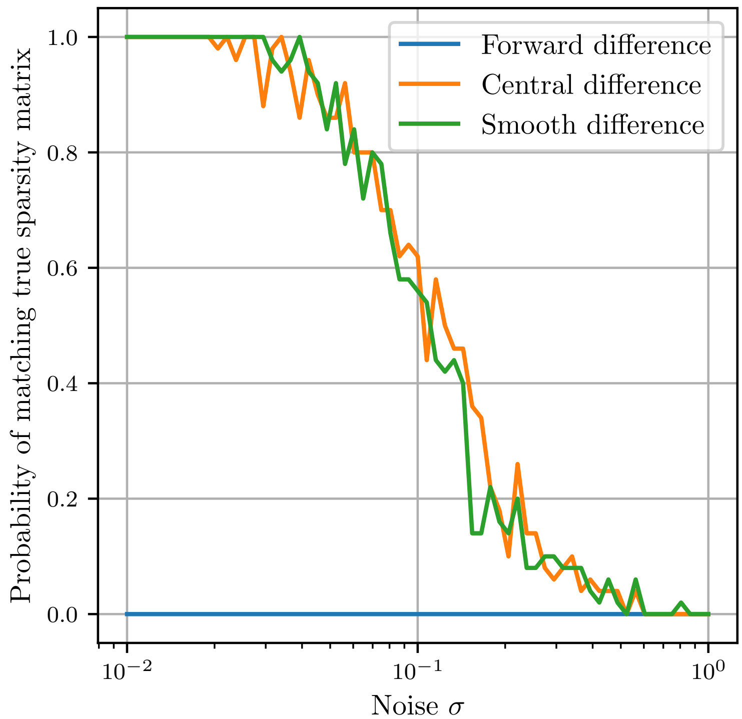


Figure 4: SINDy performance vs. noise magnitude of various differentiation schemes

Using smooth difference and central difference schemes, SINDy is able to converge to the correct model if the noise is small. However, it is never able to do so using the forward difference scheme.

4

4.a

The sparsest discovered SINDy model which supports the data is:

$$u_t = -0.985u_x - 0.010u_{xxx} + -0.089uu_x \tag{1}$$

This model's solution is visualized in Figure 5.

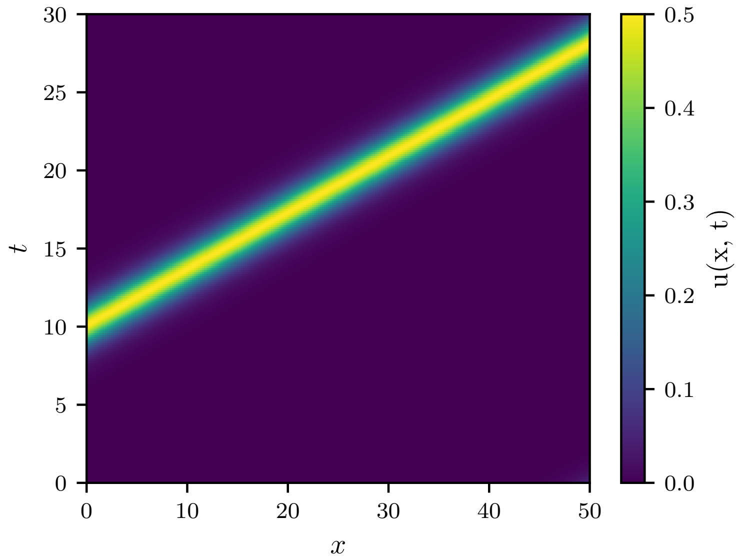


Figure 5: SINDy model of Kortweg-De Vries system ($c = 1$)

This is the incorrect sparsity matrix for this system.

4.b

The sparsest discovered SINDy model which supports the concatenated data is:

$$u_t = -1.183u_{xxx} + -6.192uu_x$$

This model's solution is visualized in Figure 6.

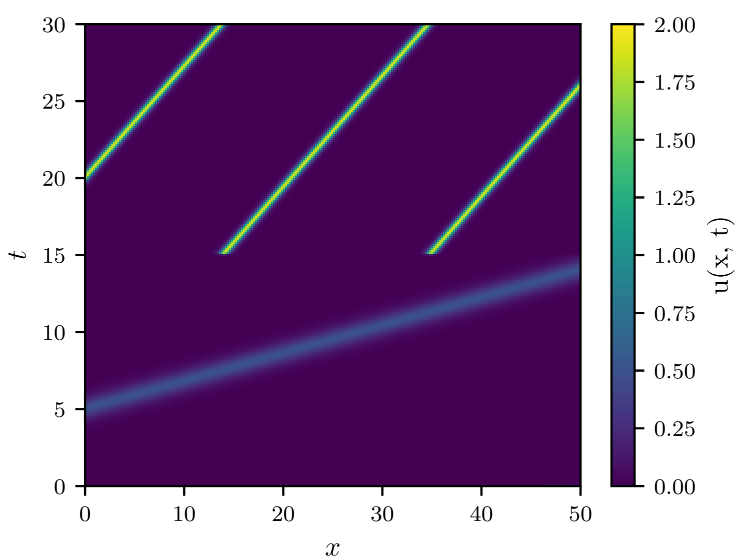


Figure 6: SINDy model of Kortweg-De Vries system from concatenated data

This is the correct sparsity matrix for this system.

4.c

When training using only the first dataset, some features of the system were not expressed in the data. Thus, the sparse optimizer converged to a simpler system which was capable of matching the data.

In contrast, more features of the dataset and the underlying system were expressed in the concatenated dataset. Thus, the sparse optimizer converged to a more complex system which was capable of matching the data.

Code Apendix

Exercise 3-1

In [ ]:
```
import pysindy as ps
import numpy as np
from numpy.typing import ArrayLike, NDArray
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp
```

In [2]:
```
plt.rcParams.update({
    "text.usetex": True,
    "font.family": "serif",
    "font.serif": "Computer Modern Roman",
    "font.size": 10,
    "xtick.labelsize": 8,
    "ytick.labelsize": 8,
})
```

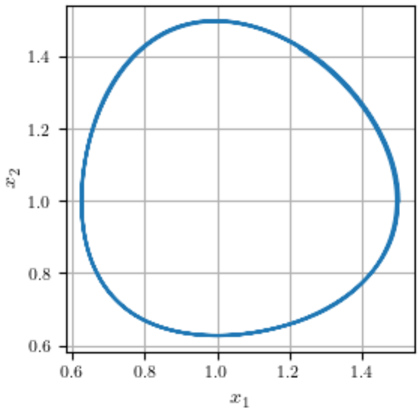
3.1.1 Numerical Simulation

In [ ]:
```
def lv_dynamics(
    t: float,
    x: ArrayLike,
    alpha: float=2,
    beta: float=2,
    delta: float=2,
    gamma: float=2,
) -> NDArray:
    """Dynamics of the Lotka-Volterra predator-prey system

    Args:
        x (ArrayLike): N x 2 array of [prey, predator] population
        alpha (float, optional): prey growth rate. Defaults to 2.
        beta (float, optional): prey death rate due to predators. Defaults to 2.
        delta (float, optional): predator growth rate due to prey. Defaults to 2.
        gamma (float, optional): predator death rate. Defaults to 2.

    Returns:
        NDArray: N x 2 array of time derivatives of x1 and x2
    """
    return np.array([alpha * x[0] - beta * x[0] * x[1], delta * x[0] * x[1] - gamma * x[1]])
```

In [4]:
```
# Compute trajectory
x0 = np.array([1.5, 1])
dt = 0.01
t = np.arange(0, 10+dt, dt)
solution = solve_ivp(lv_dynamics, (t[0], t[-1]), x0, t_eval=t)
x = solution.y.T
```

In [5]:
```
# Compute time derivative
dxdt_true = np.array([lv_dynamics(0, _x) for _x in x])
dxdt_estimate = np.array([np.gradient(_x, t) for _x in x.T])
```

In [ ]:
```
# Plot phase portrait
plt.figure(figsize=(3, 3))
plt.plot(x[:, 0], x[:, 1])
plt.xlabel("$x_1$")
plt.ylabel("$x_2$")
# plt.title("Lotka-Volterra System Phase Portrait")
plt.grid(True)
plt.savefig("p1fig1.pdf", bbox_inches="tight")
plt.show()
```



3.1.2 SINDy with 2nd-Order Polynomials

In [7]:
```
thresholds = np.array([0.2, 0.19, 0.18, 0.17, 0.16, 0.15])
feature_library = ps.feature_library.PolynomialLibrary(degree=2)
for threshold in thresholds:
    model = ps.SINDy(optimizer=ps.STLSQ(threshold=threshold), feature_names=["x1", "x2"])
    model.fit(x, t=t)
    print(f"threshold: {threshold}")
    model.print()
    print("")
```

```
threshold: 0.2
(x1)' = 2.001 x1 + -2.001 x1 x2
(x2)' = -2.002 x2 + 2.002 x1 x2

threshold: 0.19
(x1)' = 2.001 x1 + -2.001 x1 x2
(x2)' = -2.002 x2 + 2.002 x1 x2

threshold: 0.18
(x1)' = 2.001 x1 + -2.001 x1 x2
(x2)' = -2.002 x2 + 2.002 x1 x2

threshold: 0.17
(x1)' = 2.001 x1 + -2.001 x1 x2
(x2)' = -2.002 x2 + 2.002 x1 x2

threshold: 0.16
(x1)' = 2.001 x1 + -2.001 x1 x2
(x2)' = 0.408 1 + -0.408 x1 + -2.450 x2 + 0.197 x1^2 + 1.998 x1 x2 + 0.216 x2^2

threshold: 0.15
(x1)' = -0.381 1 + 2.374 x1 + 0.427 x2 + -0.182 x1^2 + -1.996 x1 x2 + -0.206 x2^2
(x2)' = 0.408 1 + -0.408 x1 + -2.450 x2 + 0.197 x1^2 + 1.998 x1 x2 + 0.216 x2^2
```

3.1.4 SINDy with 3rd-Order Polynomials

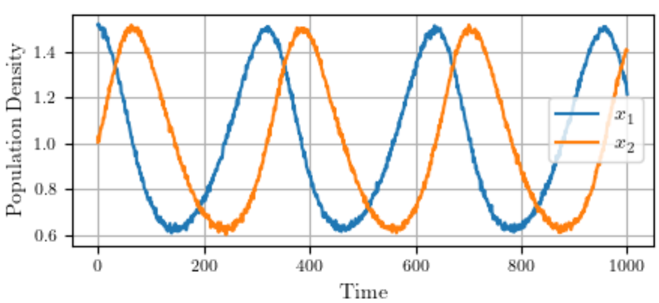
In [8]:
```
thresholds = np.array([0.2, 0.19, 0.18, 0.17, 0.16, 0.15])
feature_library = ps.feature_library.PolynomialLibrary(degree=3)
for threshold in thresholds:
    model = ps.SINDy(optimizer=ps.STLSQ(threshold=threshold), feature_names=["x1", "x2"])
    model.fit(x, t=t)
    print(f"threshold: {threshold}")
    model.print()
    print("")
```

```
threshold: 0.2
(x1)' = 2.001 x1 + -2.001 x1 x2
(x2)' = -2.002 x2 + 2.002 x1 x2

threshold: 0.19
(x1)' = 2.001 x1 + -2.001 x1 x2
(x2)' = -2.002 x2 + 2.002 x1 x2

threshold: 0.18
(x1)' = 2.001 x1 + -2.001 x1 x2
(x2)' = -2.002 x2 + 2.002 x1 x2

threshold: 0.17
(x1)' = 2.001 x1 + -2.001 x1 x2
(x2)' = -2.002 x2 + 2.002 x1 x2

threshold: 0.16
(x1)' = 2.001 x1 + -2.001 x1 x2
(x2)' = 0.408 1 + -0.408 x1 + -2.450 x2 + 0.197 x1^2 + 1.998 x1 x2 + 0.216 x2^2

threshold: 0.15
(x1)' = -0.381 1 + 2.374 x1 + 0.427 x2 + -0.182 x1^2 + -1.996 x1 x2 + -0.206 x2^2
(x2)' = 0.408 1 + -0.408 x1 + -2.450 x2 + 0.197 x1^2 + 1.998 x1 x2 + 0.216 x2^2
```

3.1.5 SINDy with Noise

In [9]:
```python
np.random.seed(0)
x_noisy = x + 0.01 * np.random.randn(*x.shape)
dxdt_noisy_estimate = np.array([np.gradient(_x, t) for _x in x_noisy.T]).T
```

In [ ]:
```python
plt.figure(figsize=(5, 2))
plt.plot(x_noisy)
plt.xlabel("Time")
plt.ylabel("Population Density")
# plt.title("Lotka-Volterra Noisy Trajectory")
plt.legend(("$x_1$", "$x_2$"), loc="right")
plt.grid(True)
plt.savefig("p1fig2.pdf", bbox_inches="tight")
plt.show()
```



In [11]:
```python
for threshold in thresholds:
    model = ps.SINDy(optimizer=ps.STLSQ(threshold=threshold), feature_names=["x1", "x2"])
    model.fit(x_noisy, t=t)
    print(f"threshold: {threshold}")
    model.print()
    print("")
```

```
threshold: 0.2
(x1)' = 1.996 x1 + -2.003 x1 x2
(x2)' = -2.013 x2 + 2.015 x1 x2

threshold: 0.19
(x1)' = 1.996 x1 + -2.003 x1 x2
(x2)' = -2.013 x2 + 2.015 x1 x2

threshold: 0.18
(x1)' = 0.106 1 + 1.939 x1 + -0.136 x2 + -0.001 x1^2 + -1.959 x1 x2 + 0.044 x2^2
(x2)' = -2.013 x2 + 2.015 x1 x2

threshold: 0.17
(x1)' = 0.106 1 + 1.939 x1 + -0.136 x2 + -0.001 x1^2 + -1.959 x1 x2 + 0.044 x2^2
(x2)' = -2.013 x2 + 2.015 x1 x2

threshold: 0.16
(x1)' = 0.106 1 + 1.939 x1 + -0.136 x2 + -0.001 x1^2 + -1.959 x1 x2 + 0.044 x2^2
(x2)' = -2.013 x2 + 2.015 x1 x2

threshold: 0.15
(x1)' = 0.106 1 + 1.939 x1 + -0.136 x2 + -0.001 x1^2 + -1.959 x1 x2 + 0.044 x2^2
(x2)' = -2.013 x2 + 2.015 x1 x2
```

Exercise 3-2

In [1]:
```
import pysindy as ps
import numpy as np
from numpy.typing import ArrayLike, NDArray
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp
```

In [2]:
```
plt.rcParams.update({
    "text.usetex": True,
    "font.family": "serif",
    "font.serif": "Computer Modern Roman",
    "font.size": 10,
    "xtick.labelsize": 8,
    "ytick.labelsize": 8,
})
```

Numerical Simulation

In [3]:
```
def lorenz_dynamics(
    t: float,
    x: ArrayLike,
    sigma: float=10,
    rho: float=28,
    beta: float=8/3,
) -> NDArray:
    """Dynamics of the Lorenz system

    Args:
        x (ArrayLike): N x 3 array of [x, y, z]
        sigma (float, optional): Model parameter. Defaults to 10.
        rho (float, optional): Model parameter. Defaults to 28.
        beta (float, optional): Model parameter. Defaults to 8/3.

    Returns:
        NDArray: Nx3 array of time derivatives of x1 and x2
    """
    xdot = np.empty_like(x)
    xdot[0] = sigma * (x[1] - x[0])
    xdot[1] = x[0] * (rho - x[2]) - x[1]
    xdot[2] = x[0] * x[1] - beta * x[2]
    return xdot
```

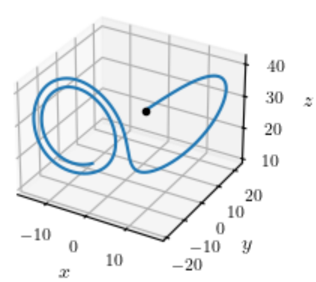
In [4]:
```
# Compute trajectory
x0 = np.array([0, 1, 20])
dt = 0.0001
t = np.arange(0, 10+dt, dt)
solution = solve_ivp(lorenz_dynamics, (t[0], t[-1]), x0, t_eval=t)
x = solution.y.T[int(1/dt):]  # truncate first t=1 of data
t = t[int(1/dt):]
```

3.2.1 SINDy Data Length Study

In [5]:
```
t_ends = [0.5, 1, 1.5, 2]
feature_library = ps.feature_library.PolynomialLibrary(degree=2)
for t_end in t_ends:
    idx_end = int(t_end / dt)
    model = ps.SINDy(optimizer=ps.STLSQ(threshold=0.2), feature_names=["x", "y", "z"])
    model.fit(x[:idx_end], t=t[:idx_end])
    print(f"simulation time: {t_end}")
    model.print()
    print("")
```

```
simulation time: 0.5
(x)' = -10.003 x + 10.003 y
(y)' = 0.247 1 + 28.081 x + -1.021 y + -1.000 x z
(z)' = -55.597 1 + 36.681 x + -20.192 y + 0.462 z + -1.891 x^2 + 1.803 x y + -1.312 x z + 0.498 y z

simulation time: 1
(x)' = -9.992 x + 9.998 y
(y)' = 0.252 1 + 28.289 x + -1.095 y + -1.005 x z
(z)' = -2.662 z + 0.999 x y

simulation time: 1.5
(x)' = -10.006 x + 10.007 y
(y)' = 9.241 1 + 3.013 x + 11.052 y + -0.732 z + 1.443 x^2 + -0.684 x y + -0.346 y z
(z)' = -2.665 z + 1.000 x y

simulation time: 2
(x)' = -10.005 x + 10.005 y
(y)' = 27.782 x + -0.960 y + -0.992 x z
(z)' = -2.665 z + 0.999 x y
```

In [6]:
```python
# Plot phase portrait up to t=2
t_end = 2
idx_end = int(2 / dt)

fig = plt.figure(figsize=(3, 3))
ax = fig.add_subplot(projection="3d")
ax.plot(x[:idx_end, 0], x[:idx_end, 1], x[:idx_end, 2])
ax.plot(x[idx_end, 0], x[idx_end, 1], x[idx_end, 2], ".k")
ax.set_xlabel("$x$")
ax.set_ylabel("$y$")
ax.set_zlabel("$z$")
# ax.set_title("Lorenz System Phase Portrait")
ax.grid(True)
ax.set_box_aspect(None, zoom=0.75)  # fix z-axis label off canvas
fig.savefig("p2fig1.pdf", bbox_inches="tight")
plt.show()
```



In [7]:
```python
# Store the coefficient matrix from the last model
Phi_mask = model.coefficients().astype(bool)
```

3.2.2 SINDy Sampling Rate Study

In [8]:
```python
dt_base = 0.0001
dts = [0.0001, 0.001, 0.01, 0.1]
t_ends = [0.5, 1, 2, 4, 9]
feature_library = ps.feature_library.PolynomialLibrary(degree=2)
for dt in dts:
    skip = int(dt / dt_base)
    for t_end in t_ends:
        idx_end = int(t_end / dt_base)
        model = ps.SINDy(optimizer=ps.STLSQ(threshold=0.2), feature_names=["x", "y", "z"])
        model.fit(x[:idx_end:skip], t[:idx_end:skip])
        print(f"time step: {dt}")
        print(f"simulation time: {t_end}")
        if np.all(model.coefficients().astype(bool)== Phi_mask):
            print("🟩 SPARSITY PASS\n")
        else:
```

```
            print("🟥 SPARSITY FAIL\n")
```

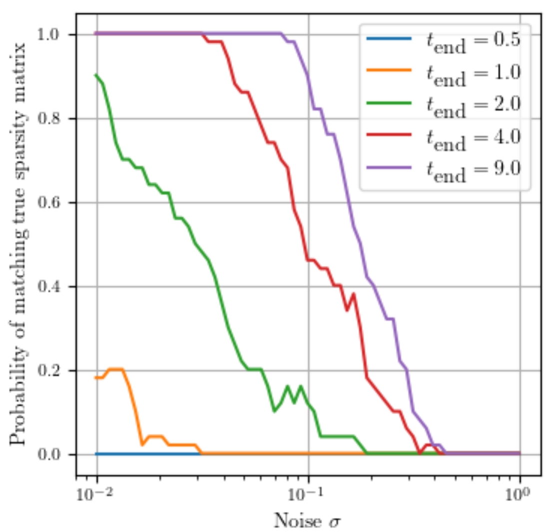
```
time step: 0.0001
simulation time: 0.5
🟥 SPARSITY FAIL

time step: 0.0001
simulation time: 1
🟥 SPARSITY FAIL

time step: 0.0001
simulation time: 2
🟩 SPARSITY PASS

time step: 0.0001
simulation time: 4
🟩 SPARSITY PASS

time step: 0.0001
simulation time: 9
🟩 SPARSITY PASS

time step: 0.001
simulation time: 0.5
🟥 SPARSITY FAIL

time step: 0.001
simulation time: 1
🟥 SPARSITY FAIL

time step: 0.001
simulation time: 2
🟩 SPARSITY PASS

time step: 0.001
simulation time: 4
🟩 SPARSITY PASS

time step: 0.001
simulation time: 9
🟩 SPARSITY PASS

time step: 0.01
simulation time: 0.5
🟥 SPARSITY FAIL

time step: 0.01
simulation time: 1
🟥 SPARSITY FAIL

time step: 0.01
simulation time: 2
🟥 SPARSITY FAIL

time step: 0.01
simulation time: 4
🟩 SPARSITY PASS

time step: 0.01
simulation time: 9
🟩 SPARSITY PASS

time step: 0.1
simulation time: 0.5
🟥 SPARSITY FAIL

time step: 0.1
simulation time: 1
🟥 SPARSITY FAIL

time step: 0.1
simulation time: 2
🟥 SPARSITY FAIL

time step: 0.1
simulation time: 4
🟥 SPARSITY FAIL
```

```
time step: 0.1
simulation time: 9
🟥 SPARSITY FAIL
```

3.1.3 SINDy Noise Study

In [9]:
```python
dt = dt_base
noise_seeds = list(range(50))
noise_stds = np.logspace(-2, 0, 65)
t_ends = [0.5, 1, 2, 4, 9]
probability_matrix = np.empty((len(noise_stds), len(t_ends)))
feature_library = ps.feature_library.PolynomialLibrary(degree=2)
for i, noise_std in enumerate(noise_stds):
    for j, t_end in enumerate(t_ends):
        success_mask = []
        for noise_seed in noise_seeds:
            np.random.seed(noise_seed)
            idx_end = int(t_end / dt_base)
            noise = noise_std * np.random.randn(*x[:idx_end].shape)
            model = ps.SINDy(optimizer=ps.STLSQ(threshold=0.2), feature_names=["x", "y", "z"])
            model.fit(x[:idx_end] + noise, t[:idx_end])
            if np.all(model.coefficients().astype(bool)== Phi_mask):
                success_mask.append(True)
            else:
                success_mask.append(False)
        probability_matrix[i, j] = sum(success_mask) / len(success_mask)
```

In [10]:
```python
fig = plt.figure(figsize=(4, 4))
ax = fig.add_subplot()
ax.set_xscale("log")
for t_end, t_end_probabilities in zip(t_ends, probability_matrix.T):
    label = r"$t_{\textrm{end}}=" + f"{t_end:2.1f}$"
    ax.plot(noise_stds, t_end_probabilities, label=label)
ax.set_xlabel("Noise $\sigma$")
ax.set_ylabel("Probability of matching true sparsity matrix")
ax.legend()
ax.grid(True)
fig.savefig("p2fig2.pdf", bbox_inches="tight")
plt.show()
```



Exercise 3-3

In [1]:
```
import pysindy as ps
import numpy as np
from numpy.typing import ArrayLike, NDArray
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp
```

In [2]:
```
plt.rcParams.update({
    "text.usetex": True,
    "font.family": "serif",
    "font.serif": "Computer Modern Roman",
    "font.size": 10,
    "xtick.labelsize": 8,
    "ytick.labelsize": 8,
})
```

In [3]:
```
np.random.seed(0)
```

3.3.1 Numerical Simulation

In [4]:
```
def lorenz_dynamics(
    t: float,
    x: ArrayLike,
    sigma: float=10,
    rho: float=28,
    beta: float=8/3,
) -> NDArray:
    """Dynamics of the Lorenz system

    Args:
        x (ArrayLike): N x 3 array of [x, y, z]
        sigma (float, optional): Model parameter. Defaults to 10.
        rho (float, optional): Model parameter. Defaults to 28.
        beta (float, optional): Model parameter. Defaults to 8/3.

    Returns:
        NDArray: Nx3 array of time derivatives of x1 and x2
    """
    xdot = np.empty_like(x)
    xdot[0] = sigma * (x[1] - x[0])
    xdot[1] = x[0] * (rho - x[2]) - x[1]
    xdot[2] = x[0] * x[1] - beta * x[2]
    return xdot
```

In [5]:
```
# Compute trajectory
x0 = np.array([0, 1, 20])
dt = 0.01
t = np.arange(0, 6+dt, dt)
solution = solve_ivp(lorenz_dynamics, (t[0], t[-1]), x0, t_eval=t)
x = solution.y.T[int(1/dt):]  # truncate first t=1 of data
t = t[int(1/dt):]
```

3.3.2 Estimate Derivative

In [6]:
```
def forward_difference(x: ArrayLike, t: ArrayLike=np.arange(len(x))) -> NDArray:
    """Forward difference numerical differentiation

    Args:
        x (ArrayLike): independent variable
        t (ArrayLike, optional): dependent variable. Defaults to np.arange(len(x)).

    Returns:
        NDArray: derivative of independent variable
    """
    dxdt = np.empty_like(x)
    dxdt[:-1] = (x[1:] - x[:-1]) / (t[1:] - t[:-1]).reshape(-1, 1)
    dxdt[-1] = dxdt[-2]
```

```
    return dxdt
```

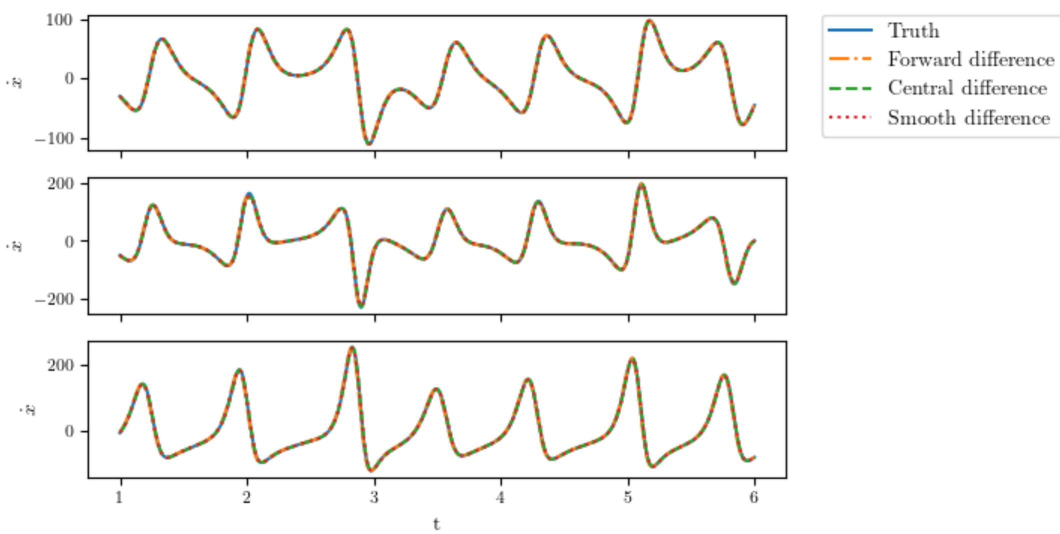
In [7]:
```
def central_difference(x: ArrayLike, t: ArrayLike=np.arange(len(x))) -> NDArray:
    """Central difference numerical differentiation

    Args:
        x (ArrayLike): independent variable
        t (ArrayLike, optional): dependent variable. Defaults to np.arange(len(x)).

    Returns:
        NDArray: derivative of independent variable
    """
    dxdt = np.empty_like(x)
    dxdt[1:-1] = (x[2:] - x[:-2]) / (t[2:] - t[:-2]).reshape(-1, 1)
    dxdt[0] = dxdt[1]
    dxdt[-1] = dxdt[-2]
    return dxdt
```

In [8]:
```
true_difference = lambda x, t: np.array([lorenz_dynamics(_t, _x) for _t, _x in zip(t, x)])
```

In [9]:
```
smooth_difference = ps.differentiation.SmoothedFiniteDifference()
```

In [10]:
```
noise_std = 0.001
x_noisy = x + noise_std * np.random.randn(*x.shape)

dxdt_estimates = []
for differentiator in [true_difference, forward_difference, central_difference, smooth_difference]:
    dxdt_estimates.append(differentiator(x_noisy, t))
```

In [11]:
```
estimator_names = ["Truth", "Forward difference", "Central difference", "Smooth difference"]
linestyles = ["-", "-.", "--", ":"]
fig, axs = plt.subplots(3, 1, figsize=(6, 4), sharex=True)
for i, (dxdt_estimate, label) in enumerate(zip(dxdt_estimates, estimator_names)):
    for j, ax in enumerate(axs):
        ax.plot(t, dxdt_estimate[:, j], linestyles[i], label=label)
_ = [ax.set_ylabel("$\dot{x}$") for ax in axs]
axs[2].set_xlabel("t")
axs[0].legend(bbox_to_anchor=(1.05, 1), loc='upper left', borderaxespad=0.)
plt.show()
```



3.3.c SINDy from numerical derivatives

In [12]:
```
feature_library = ps.feature_library.PolynomialLibrary(degree=2)
model = ps.SINDy(optimizer=ps.STLSQ(threshold=0.2), feature_names=["x", "y", "z"])
for dxdt, name in zip(dxdt_estimates, estimator_names):
    model.fit(x=x_noisy, t=t, x_dot=dxdt)
    print(f"gradient method: {name}")
    model.print()
    print("")
```

```
    if name == "Truth":
        Phi_mask = model.coefficients().astype(bool)
```

```
gradient method: Truth
(x)' = -10.000 x + 10.000 y
(y)' = 28.000 x + -1.000 y + -1.000 x z
(z)' = -2.667 z + 1.000 x y

gradient method: Forward difference
(x)' = -10.145 x + 9.986 y
(y)' = 30.156 x + -2.129 y + -1.037 x z
(z)' = 10.600 1 + -3.110 z + 0.997 x y

gradient method: Central difference
(x)' = -9.985 x + 9.985 y
(y)' = 27.592 x + -0.916 y + -0.987 x z
(z)' = -2.660 z + 0.997 x y

gradient method: Smooth difference
(x)' = -9.971 x + 9.971 y
(y)' = 27.420 x + -0.878 y + -0.983 x z
(z)' = -2.655 z + 0.995 x y
```

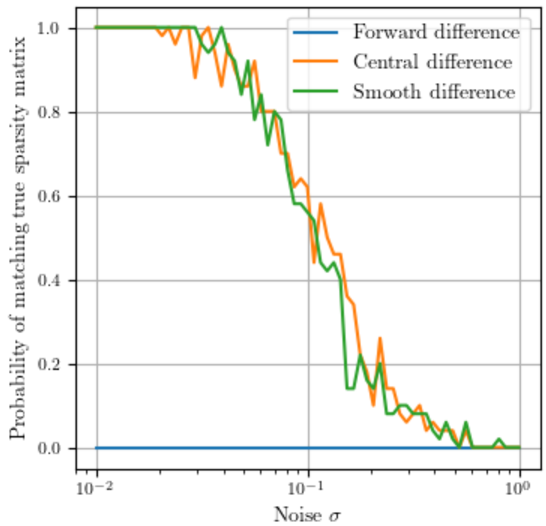
3.3.3 SINDy Noise Study

In [13]:

```
feature_library = ps.feature_library.PolynomialLibrary(degree=2)
model = ps.SINDy(optimizer=ps.STLSQ(threshold=0.2), feature_names=["x", "y", "z"])
differentiators = [true_difference, forward_difference, central_difference, smooth_difference]
noise_stds = np.logspace(-2, 0, 65)
probability_matrix = np.empty((len(noise_stds), len(differentiators)))
for i, noise_std in enumerate(noise_stds):
    for j, (differentiator, name) in enumerate(zip(differentiators, estimator_names)):
        success_mask = []
        for seed in list(range(50)):
            x_noisy = x + noise_std * np.random.randn(*x.shape)
            dxdt = differentiator(x_noisy, t)
            model.fit(x=x_noisy, t=t, x_dot=dxdt)
            if np.all(model.coefficients().astype(bool)== Phi_mask):
                success_mask.append(True)
            else:
                success_mask.append(False)
        probability_matrix[i, j] = sum(success_mask) / len(success_mask)
```

In [15]:

```
fig = plt.figure(figsize=(4, 4))
ax = fig.add_subplot()
ax.set_xscale("log")
for name, t_end_probabilities in zip(estimator_names[1:], probability_matrix.T[1:]):
    label = f"{name}"
    ax.plot(noise_stds, t_end_probabilities, label=label)
ax.set_xlabel("Noise $\sigma$")
ax.set_ylabel("Probability of matching true sparsity matrix")
ax.legend()
ax.grid(True)
fig.savefig("p3fig1.pdf", bbox_inches="tight")
plt.show()
```



Forward difference
Central difference
Smooth difference
Probability of matching true sparsity matrix
1.0
0.8
0.6
0.4
0.2
0.0
10^{-2}
10^{-1}
10^{0}
Noise σ

Exercise 3-4

In [79]:
```python
import pysindy as ps
import numpy as np
from numpy.typing import ArrayLike, NDArray
from scipy.fftpack import diff as psdiff
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp
```

In [80]:
```python
plt.rcParams.update({
    "text.usetex": True,
    "font.family": "serif",
    "font.serif": "Computer Modern Roman",
    "font.size": 10,
    "xtick.labelsize": 8,
    "ytick.labelsize": 8,
})
```

3.4.1 KdV Solution

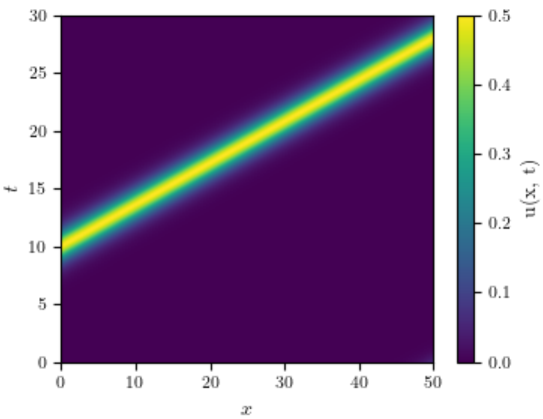
In [81]:
```python
do_load = False

if do_load:
    try:
        KdV_sol = np.load("KdV_solution.npz")
        X1 = KdV_sol["X1"]
        X2 = KdV_sol["X2"]
    except FileNotFoundError:
        KdV_sol = None
else:
    KdV_sol = None
```

In [82]:
```python
def kdv_exact(x, c, t=0):
    """
    Profile of the exact solution to the KdV for a single soliton.

    From https://scipy-cookbook.readthedocs.io/items/KdV.html
    """
    u = 0.5 * c * np.cosh(0.5 * np.sqrt(c) * (x - c * t))**(-2)
    return u

def kdv(t, u, L):
    """
    Differential equations for the KdV equation.

    From From https://scipy-cookbook.readthedocs.io/items/KdV.html
    """
    ux = psdiff(u, period=L)
    uxxx = psdiff(u, period=L, order=3)
    dudt = -6 * u * ux - uxxx
    return dudt
```

In [ ]:
```python
# Spatial domain
L = 50
N = 256
dx = L / (N - 1.0)
x = np.linspace(0, (1 - 1.0 / N) * L, N)

# Time domain
dt = 0.1
t = np.arange(0, 30 + dt, dt)

# Initial condition
c = 1
u0 = kdv_exact(x - L / 3, c)
```

In [84]:
```
if not KdV_sol:
    sol1 = solve_ivp(kdv, (t[0], t[-1]), u0, args=(L,), t_eval=t, dense_output=True, method='Radau')
    X1 = sol1.y.reshape(len(x), len(t))
```

In [85]:
```
plt.figure(figsize=(4, 3))
plt.imshow(X1, extent=[0, L, t[0], t[-1]], aspect='auto', origin='lower', cmap='viridis')
plt.colorbar(label="u(x, t)")
plt.xlabel("$x$")
plt.ylabel("$t$")
# plt.title('KdV Solution (c=1)')
plt.savefig("p4fig1.pdf", bbox_inches='tight')
plt.show()
```



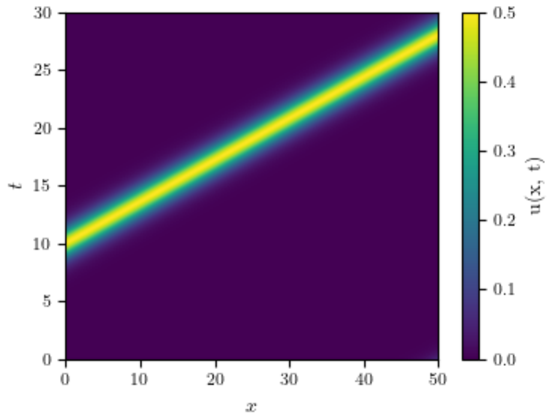
In [86]:
```
lib_funcs = [
    lambda u: u,
    lambda u: u**2,
]
func_names = [
    lambda u: "u",
    lambda u: "u^2",
]
pde_lib = ps.PDELibrary(
    library_functions=lib_funcs,
    function_names=func_names,
    derivative_order=4,
    spatial_grid=x,
    include_bias=True,
)

model = ps.SINDy(
    feature_library=pde_lib,
    optimizer=ps.STLSQ(threshold=0.01),
    feature_names=["u"]
)
model.fit(X1.reshape(len(x), len(t), 1), t=dt)
model.print()
```

```
(u)' = -0.985 u_1 + -0.010 u_111 + -0.089 uu_1
```

In [87]:
```
def kdv_model_1(t, u, L):
    """
    Differential equations for the KdV equation.
    """
    ux = psdiff(u, period=L)
    uxxx = psdiff(u, period=L, order=3)
    dudt = -0.985 * ux - 0.010 * uxxx - 0.089 * u * ux
    return dudt

test_sol1 = solve_ivp(kdv_model_1, (t[0], t[-1]), u0, args=(L,), t_eval=t, dense_output=True, method='Radau
X1_model = sol1.y.reshape(len(x), len(t))
```

In [88]:
```
plt.figure(figsize=(4, 3))
plt.imshow(X1_model, extent=[0, L, t[0], t[-1]], aspect='auto', origin='lower', cmap='viridis')
plt.colorbar(label="u(x, t)")
plt.xlabel("$x$")
plt.ylabel("$t$")
```

```
# plt.title('KdV SINDy Model')
plt.savefig("p4afig2.pdf", bbox_inches='tight')
plt.show()
```



3.4.2

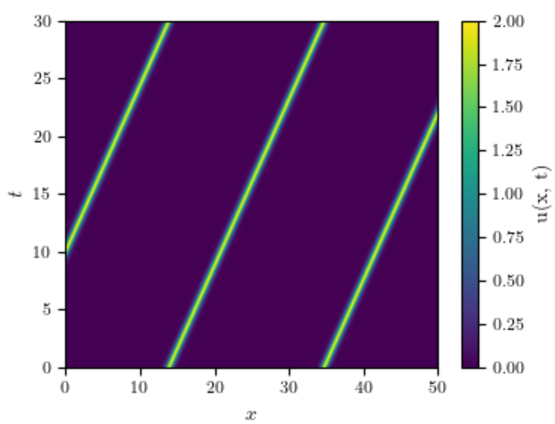
In [89]:
```
c = 4
u0 = kdv_exact(x - L / 3, c)  # Single soliton, offset for periodicity
```

In [90]:
```
if not KdV_sol:
    sol2 = solve_ivp(kdv, (t[0], t[-1]), u0, args=(L,), t_eval=t, dense_output=True, method='Radau')
    X2 = sol2.y.reshape(len(x), len(t))
```

In [91]:
```
X = np.concatenate((X1, X2), axis=0)
```

In [92]:
```
if not KdV_sol:
    np.save("KdV_solution.npz", {"X1": X1, "X2": X2}, allow_pickle=True)
```

In [93]:
```
plt.figure(figsize=(4, 3))
plt.imshow(X2, extent=[0, L, t[0], t[-1]], aspect='auto', origin='lower', cmap='viridis')
plt.colorbar(label="u(x, t)")
plt.xlabel("$x$")
plt.ylabel("$t$")
# plt.title('KdV Solution (c=4)')
plt.savefig("p4fig3.pdf", bbox_inches='tight')
plt.show()
```



In [94]:
```
lib_funcs = [
    lambda u: u,
    lambda u: u**2,
]
func_names = [
    lambda u: "u",
    lambda u: "u^2",
]
pde_lib = ps.PDELibrary(
    library_functions=lib_funcs,
```

```python
    function_names=func_names,
    derivative_order=4,
    spatial_grid=x,
    include_bias=True,
)

model = ps.SINDy(
    feature_library=pde_lib,
    optimizer=ps.STLSQ(threshold=1),
    feature_names=["u"]
)
model.fit(X.reshape(len(x), 2*len(t), 1), t=dt)
model.print()
```

```
(u)' = -3.782 uu_1 + 1.596 u^2u_1
```

In [95]:
```python
def kdv_model_2(t, u, L):
    """
    Differential equations for the KdV equation.
    """
    ux = psdiff(u, period=L)
    uxxx = psdiff(u, period=L, order=3)
    dudt = - 1.183 * uxxx -6.192 * u * ux
    return dudt

test_sol1 = solve_ivp(kdv, (t[0], t[-1]), u0, args=(L,), t_eval=t, dense_output=True, method='Radau')
X2_model = sol2.y.reshape(len(x), len(t))
X_model = np.concatenate((X1_model, X2_model), axis=0)
```

In [96]:
```python
plt.figure(figsize=(4, 3))
plt.imshow(X_model, extent=[0, L, t[0], t[-1]], aspect='auto', origin='lower', cmap='viridis')
plt.colorbar(label="u(x, t)")
plt.xlabel("$x$")
plt.ylabel("$t$")
# plt.title('KdV SINDy model using concatenateed data')
plt.savefig("p4fig2.pdf", bbox_inches='tight')
plt.show()
```