# Brunton and Kutz Problem 6.1 part d: Lorenz System Prediction

Source Filename: /main.py

Rico A. R. Picone

This is the solution for Brunton and Kutz (2022), exercise 6.1, part d regarding the Lorenz equations. Only the $\rho = 28$ case is considered. First, import the necessary libraries:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import integrate
from mpl_toolkits.mplot3d import Axes3D
import keras
from keras.models import Sequential
from keras.layers import Dense, Input, Activation
from keras import optimizers
```

Set script options:

```
regenerate_data = True  # Regenerate the training data
retrain = True  # Retrain the model
```

Define the Lorenz equations:

```
def lorenz(x_, t, sigma=10, beta=8/3, rho=28):
    """
    Lorenz equations dynamics (dx/dt, dy/dt, dz/dt)
    """
    x, y, z = x_
    dx = sigma * (y - x)
    dy = x * (rho - z) - y
    dz = x * y - beta * z
    return [dx, dy, dz]
```

Define a function to generate the training data by numerically solving the Lorenz equations for a given initial condition:

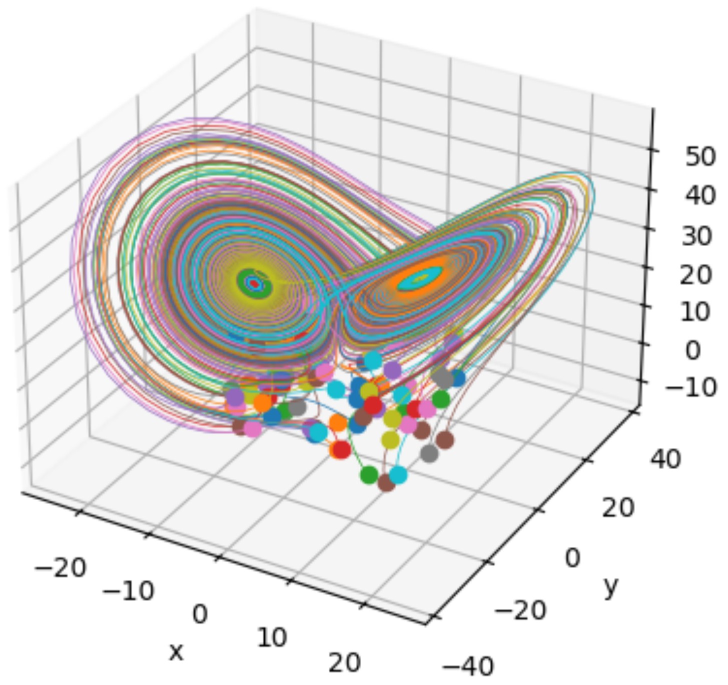
```
def generate_data(n_samples, n_timesteps, dt, sigma=10, beta=8/3, rho=28,
seed_offset=0):
    """
    Generate training data for the Lorenz equations
    """
    t = np.linspace(0, (n_timesteps-1)*dt, n_timesteps)  # Time array
    x = np.zeros((n_samples, n_timesteps, 3))  # Array to store the data
    for i in range(n_samples):
        np.random.seed(i+seed_offset)  # For reproducibility
        x0 = np.random.uniform(-15, 15, 3)  # Random initial condition
        x[i] = integrate.odeint(
            lorenz,  # Dynamics to integrate
            x0,  # Initial condition
            t,  # Time array
            args=(sigma, beta, rho)  # Parameters for the Lorenz equations
        )

    return x
```

Generate the training data:

```
n_samples = 100  # Number of samples
n_t = 1000  # Number of time steps
dt = 0.01  # Time step
rhos_train = [10, 28, 40]  # Values of rho for training data
n_rhos = len(rhos_train)
if regenerate_data:
    data = np.zeros((n_rhos, n_samples, n_t, 3))
    for i, rho in enumerate(rhos_train):
        data[i] = generate_data(n_samples, n_t, dt, rho=rho)
    np.save('training-data.npy', data)
else:
    data = np.load('training-data.npy')
```

Plot the integrated trajectories of the Lorenz variables for rho = 28:

```
rhoi = 1  # Index of rho value
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
for i in range(n_samples):
    ax.plot(
        data[rhoi, i, :, 0],
        data[rhoi, i, :, 1],
        data[rhoi, i, :, 2],
        lw=0.5
    )
    ax.plot(
        data[rhoi, i, 0, 0], data[rhoi, i, 0, 1], data[rhoi, i, 0, 2],
        lw=0.5, marker='o', color=ax.lines[-1].get_color()
    )
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')
plt.draw()
```



Transform the data into a format suitable for training a neural network. The input to the network will be the states of the Lorenz variables at time $t$ and the output will be the states at time $t + 1$. The samples are concatenated along the first axis:

```
X = np.zeros(((n_t-1)*n_samples*n_rhos, 3))
Y = np.zeros(((n_t-1)*n_samples*n_rhos, 3))
for j in range(n_rhos):
    for i in range(n_samples):
        X[(j*n_samples+i)*(n_t-1):(j*n_samples+i+1)*(n_t-1)] = \
            data[j, i, :-1]
        Y[(j*n_samples+i)*(n_t-1):(j*n_samples+i+1)*(n_t-1)] = \
            data[j, i, 1:]
```

Define the neural network architecture:

```
def build_model():
    """
    Build the feedforward neural network model
    """
    model = Sequential()
    model.add(Input(shape=(3,)))
    model.add(Dense(10))
    model.add(Activation('relu'))
    model.add(Dense(10))
    model.add(Activation('relu'))
    model.add(Dense(10))
    model.add(Activation('relu'))
    model.add(Dense(3))
    return model
```

Compile the model:

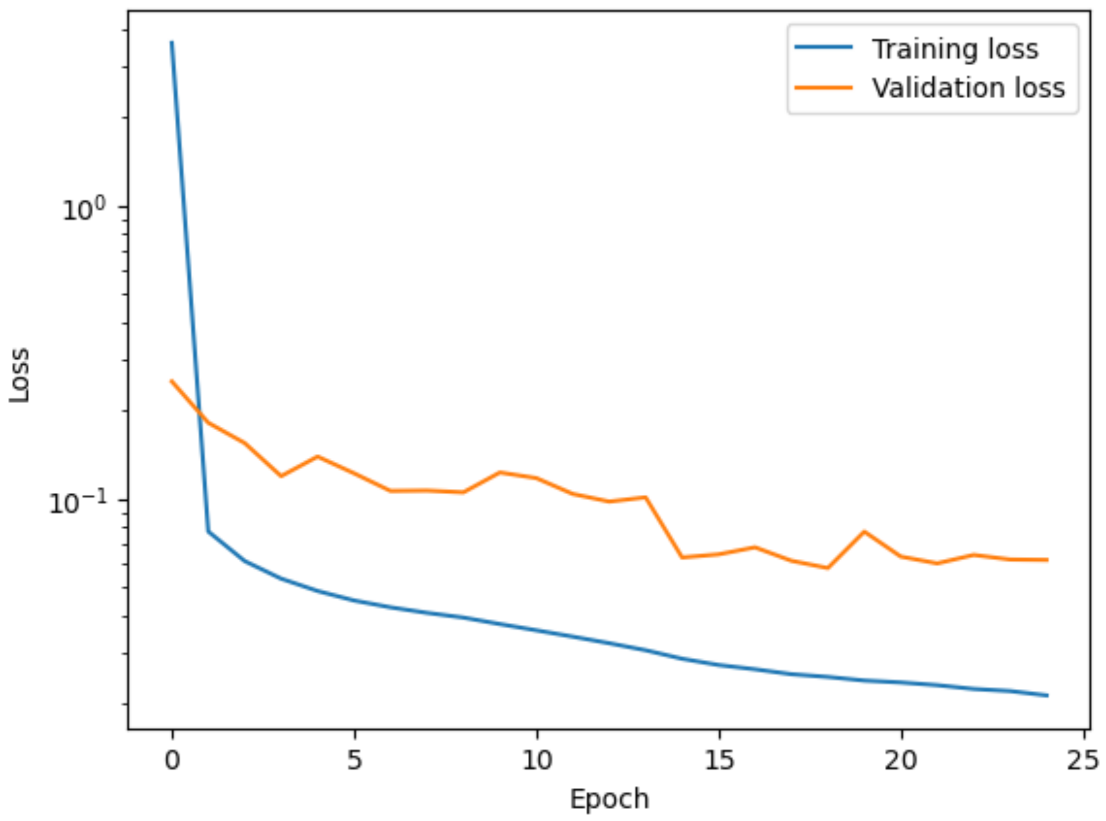
```
model = build_model()
model.compile(
    optimizer=optimizers.Adam(learning_rate=0.001),
    loss='mean_squared_error',  # Loss function
    metrics=['mean_absolute_error'],  # Metrics to monitor
)
```

Train the model:

```
if retrain:
    history = model.fit(
        X,  # Input data
        Y,  # Target data
        epochs=25,  # Number of epochs
        batch_size=32,  # Batch size
        validation_split=0.2,  # Validation split
        shuffle=True,  # Shuffle the data
    )
    model.save('model.keras')
    history = True
else:
    model = keras.models.load_model('model.keras')
    history = False
```

Plot the training and validation loss versus the epoch:

```
if history:
    fig, ax = plt.subplots()
    ax.set_yscale('log')
    ax.plot(model.history.history['loss'], label='Training loss')
    ax.plot(model.history.history['val_loss'], label='Validation loss')
    ax.set_xlabel('Epoch')
    ax.set_ylabel('Loss')
    ax.legend()
    plt.draw()
```



Generate new test trajectories using the trained model:

```
n_test_samples = 20  # Number of test samples
rhos_test = [17, 35]  # Values of rho for test data
n_test_rhos = len(rhos_test)
if regenerate_data:
    data_test = np.zeros((n_test_rhos, n_test_samples, n_t, 3))
    for i, rho in enumerate(rhos_test):
        data_test[i] = generate_data(n_test_samples, n_t, dt, rho=rho,
seed_offset=2*n_samples*i)
    np.save('test-data.npy', data_test)
else:
    data_test = np.load('test-data.npy')
```

Transform the data into a format suitable for the neural network:

```
X_test = np.zeros(((n_t-1)*n_test_samples*n_test_rhos, 3))
Y_test = np.zeros(((n_t-1)*n_test_samples*n_test_rhos, 3))
for j in range(n_test_rhos):
    for i in range(n_test_samples):
        X_test[(j*n_test_samples+i)*(n_t-1):(j*n_test_samples+i+1)*(n_t-1)] =
\
            data_test[j, i, :-1]
        Y_test[(j*n_test_samples+i)*(n_t-1):(j*n_test_samples+i+1)*(n_t-1)] =
\
            data_test[j, i, 1:]
```

Predict the next state using the trained model:

```
Y_pred = model.predict(X_test)
```

```
   1/1249 ──────────────────────────────────── 35s 28ms/step
 153/1249 ──────────────────────────────────── 0s 330us/step
 284/1249 ──────────────────────────────────── 0s 391us/step
 442/1249 ──────────────────────────────────── 0s 365us/step
 611/1249 ──────────────────────────────────── 0s 346us/step
 786/1249 ──────────────────────────────────── 0s 333us/step
 956/1249 ──────────────────────────────────── 0s 326us/step
1134/1249 ──────────────────────────────────── 0s 319us/step
1249/1249 ──────────────────────────────────── 0s 331us/step
1249/1249 ──────────────────────────────────── 0s 332us/step
```

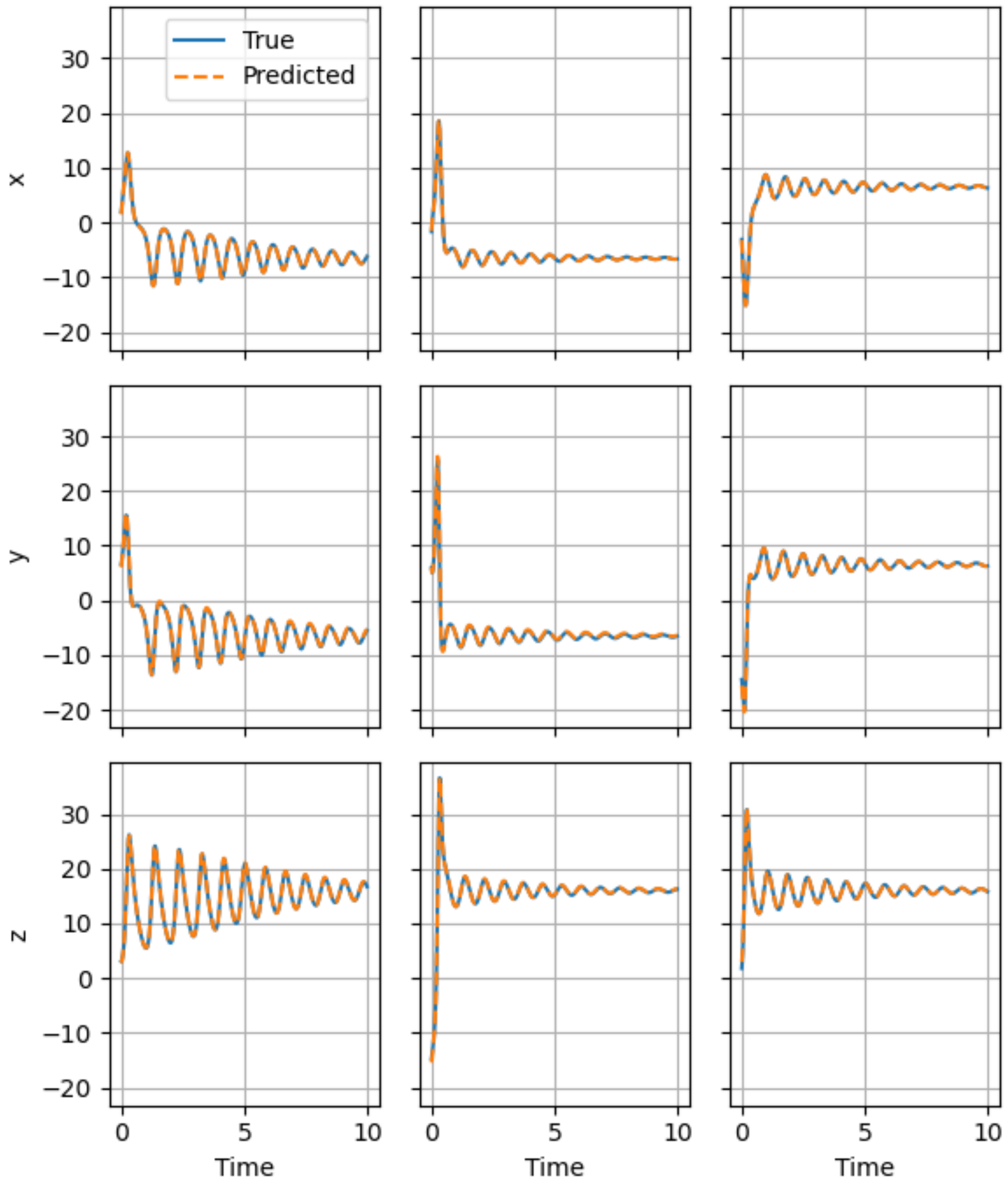
Compute the mean absolute error (MAE) between the predicted and true states:

```
mae = np.mean(np.abs(Y_test - Y_pred))
print(f'Mean absolute error (MAE) for test trajectories: {mae}')
```

```
Mean absolute error (MAE) for test trajectories: 0.07700586249013075
```

Plot the x, y, and z coordinates of the true and predicted trajectories for 3 test samples:

```
t = np.linspace(0, (n_t-1)*dt, n_t)  # Time array
labels = ['x', 'y', 'z']
fig, axs = plt.subplots(3, 3, figsize=(6, 7), sharex=True, sharey=True)
for i in range(3):
    for j in range(3):
        axs[j, i].plot(
            t[:-1], Y_test[i*(n_t-1):(i+1)*(n_t-1), j], label='True'
        )
        axs[j, i].plot(
            t[:-1], Y_pred[i*(n_t-1):(i+1)*(n_t-1), j], label='Predicted',
            linestyle='--'
        )
        axs[j, i].grid()
        if i == 0:
                axs[j, i].set_ylabel(labels[j])
                axs[0, i].legend()
    axs[2, i].set_xlabel('Time')
plt.tight_layout()
plt.show()
```



We achieve excellent agreement between the true (numerically integrated) and predicted trajectories, even for lobe transitions. The test values of $\rho$ are 17 and 35, different than the training values of 10, 28, and 40. This demonstrates the ability of the neural network to generalize to new values of $\rho$.

# Brunton and Kutz Problem 7.5: Rossler SINDy Modeling

Source Filename: /main.py

Rico A. R. Picone

This is the solution for Brunton and Kutz (2022), exercise 7.5 regarding the SINDy identification of the Rossler system. First, import the necessary libraries:

```python
import numpy as np
from scipy.special import comb
import matplotlib.pyplot as plt
from scipy import integrate
from mpl_toolkits.mplot3d import Axes3D
```

We use two different approaches to solve the exercise:

1. Using the sequential thresholded least-squares (STLS) algorithm provided in the book.
2. Using the `pySINDy` library.

In both cases, we will use the same data generated from the Rossler system. In the second case, we will explore the effect of varying the sparsity threshhold, the number of samples, and the trajectory length.

Define the Rossler system:

```python
def rossler(x_, t, a=0.2, b=0.2, c=14):
    """
    Rossler system dynamics (dx/dt, dy/dt, dz/dt)
    """
    x, y, z = x_
    dx = -y - z
    dy = x + a*y
    dz = b + z*(x - c)
    return [dx, dy, dz]
```

Define a function to generate the training data by numerically solving the Rossler system for random initial conditions:

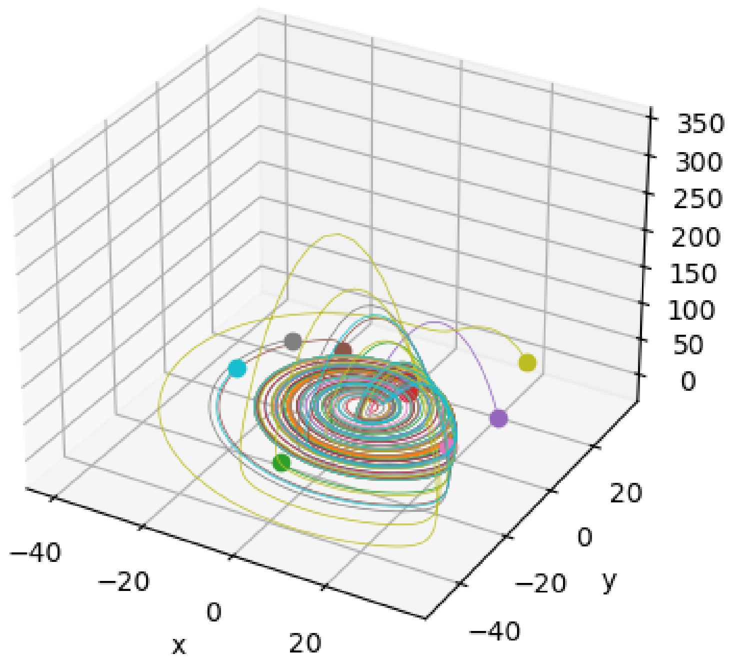
```python
def generate_data(n_samples, n_timesteps, dt, a=0.2, b=0.2, c=14, seed_offset=0):
    """
    Generate training data for the Rossler system
    """
    t = np.linspace(0, (n_timesteps-1)*dt, n_timesteps)  # Time array
    x = np.zeros((n_samples, n_timesteps, 3))  # Array to store the data
    for i in range(n_samples):
        np.random.seed(i + seed_offset)  # For reproducibility
        x0 = np.random.uniform(-30, 30, 3)  # Random initial condition
        x[i] = integrate.odeint(
            rossler,  # Dynamics to integrate
            x0,  # Initial condition
            t,  # Time array
            args=(a, b, c)  # Parameters for the Lorenz equations
        )
    return x
```

Generate the training data:

```
n_samples = 10  # Number of samples
n_t = 5_000  # Number of time steps
dt = 0.01  # Time step
a = 0.2
b = 0.2
c = 14
time = np.linspace(0, (n_t-1)*dt, n_t)  # Time array
data = generate_data(n_samples, n_t, dt, a=a, b=b, c=c)
```

Plot the integrated trajectories of the Rossler variables:

```
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
for i in range(n_samples):
    ax.plot(data[i, :, 0], data[i, :, 1], data[i, :, 2], lw=0.5)
    ax.plot(
        data[i, 0, 0], data[i, 0, 1], data[i, 0, 2],
        lw=0.5, marker='o', color=ax.lines[-1].get_color()
    )
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')
plt.show()
```



Compute the $\dot{x}$ matrix using the dynamics of the Rossler system:

```
dx_data = np.zeros((n_samples, n_t, 3))
for i in range(n_samples):
    for j in range(n_t):
        dx_data[i, j] = rossler(data[i, j], time[j])
```

Concatenate the data samples to create the input matrix $X$ and the output matrix $\dot{X}$:

```
X = np.concatenate(data, axis=0)
dX = np.concatenate(dx_data, axis=0)
```

## Using the Sequential Thresholded Least-Squares (STLS) Algorithm

In this section, we use the sequential thresholded least-squares (STLS) algorithm to identify the terms of the Rossler system. Begin by defining the library of functions $\Theta(X)$ that are candidates to be included in the identified dynamics:

```
def Theta(X):
    """A library of second-degree polynomial functions for the simple algorithm
    The columns are [1, x, y, z, xx, xy, xz, yy, yz, zz]
    """
    n = X.shape[1]  # Number of variables
    # m = 10  # Number of terms (columns) in the library
        # comb(n + highest_degree, highest_degree)  # Number of terms
        # See https://math.stackexchange.com/a/2928878
        # Instead of a general formula based on the highest degree,
        # we hardcode the 10 columns for simplicity
    Theta = np.zeros((X.shape[0], m))
    Theta[:, 0] = 1  # Constant term
    Theta[:, 1:4] = X  # Linear terms
    Theta[:, 4] = X[:, 0]**2  # Quadratic terms
    Theta[:, 5] = X[:, 0]*X[:, 1]
    Theta[:, 6] = X[:, 0]*X[:, 2]
    Theta[:, 7] = X[:, 1]**2
    Theta[:, 8] = X[:, 1]*X[:, 2]
    Theta[:, 9] = X[:, 2]**2
    return Theta
```

Define the function to sparsify the dynamics using the STLS algorithm:

```
def sparsifyDynamics(Theta, dXdt, threshold, n):
    """Sparsify the dynamics using the STLS algorithm

    Uses the sequential thresholded least-squares (STLS) algorithm
    from Brunton and Kutz (2022) to sparsify the dynamics.
    """
    Xi = np.linalg.lstsq(Theta,dXdt,rcond=None)[0]  # Initial guess
    for k in range(10):  # 10 iterations
        smallinds = np.abs(Xi) < threshold  # Find small coeffs
        Xi[smallinds] = 0   # Zero out small coeffs
        for ind in range(n):  # n is state dimension
            biginds = smallinds[:,ind] == 0  # Find big coeffs
            # Regress onto remaining terms to find sparse Xi
            Xi[biginds,ind] = np.linalg.lstsq(
                Theta[:, biginds], dXdt[:,ind], rcond=None
            )[0]
    return Xi
```

Sparsify the dynamics:

```
m = 10  # Number of terms in the library
threshold = 0.1  # Regularization parameter
Xi = sparsifyDynamics(Theta(X), dX, threshold, 3)
```

Print the identified terms:

```
def print_terms(Xi):
    """Print the identified terms"""
    terms = [
        '1', 'x', 'y', 'z', 'x^2', 'xy', 'xz', 'y^2', 'yz', 'z^2',
    ]
    variables = ['x', 'y', 'z']
    for i in range(Xi.shape[1]):
        print(f"d{variables[i]}/dt = ", end='')
        for j in range(Xi.shape[0]):
            if Xi[j, i] != 0:
                print(f" + ({Xi[j, i]:.2f}){terms[j]}", end='')
        print()
```

```
print_terms(Xi)
```

```
dx/dt =  + (-1.00)y + (-1.00)z
dy/dt =  + (1.00)x + (0.20)y
dz/dt =  + (0.20)1 + (-14.00)z + (1.00)xz
```

So we have properly identified the terms of the Rossler system using the sequential thresholded least-squares (STLS) algorithm. In the next section, we will use the `pySINDy` library to identify the terms, and explore the use of varying sparsity threshhold, number of samples, and trajectory lengths.

## Using the `pySINDy` Package

In this section, we use the `pySINDy` package to identify the terms of the Rossler system. We will also explore the effect of varying the sparsity threshhold, the number of samples, and the trajectory length.

First, load the `pySINDy` package:

```
import pysindy
```

Fit the SINDy model. We use the same data generated from the Rossler system, but we don't need to concatenate the samples. Above we used the dynamics of the Rossler system to compute the time derivatives. In the case of a measured system, we often don't have measurements of the time derivatives. Therefore, although we can pass the derivative computations based on the model, the `pySINDy` package will estimate the time derivatives from the state data, which is more realistic.

```
model = pysindy.SINDy(feature_names=["x", "y", "z"])
model.fit(list(data), t=dt, multiple_trajectories=True)
```

```
SINDy(differentiation_method=FiniteDifference(),
      feature_library=PolynomialLibrary(), feature_names=['x', 'y', 'z'],
      optimizer=STLSQ())
```

Print the identified dynamics:

```
print("Dynamics identified by pySINDy:")
model.print()
```

```
Dynamics identified by pySINDy:
(x)' = -1.000 y + -0.999 z
(y)' = 1.000 x + 0.200 y
(z)' = 0.200 1 + -13.931 z + 0.995 x z
```

The identified dynamics are similar to the ones obtained using the STLS algorithm.

Next, we explore the effect of varying the sparsity threshhold, the number of samples, trajectory length, and amounts of noise. Begin by defining a function to fit the SINDy model and return the identified dynamics:

```
def fit_sindy(
    data,
    dt,
    feature_names,
    threshold=0.01,
    n_samples=None,
    n_timesteps=None,
    noise_std=None,
):
    """Returns a SINDy model fitted to the data"""
    # Parse arguments
    if n_samples is None:
        n_samples = data.shape[0]
    elif n_samples > data.shape[0]:
        raise ValueError(
            "n_samples must be less than or equal to the number of" +
                "samples in the data."
        )
    if n_timesteps is None:
        n_timesteps = data.shape[1]
    elif n_timesteps > data.shape[1]:
        raise ValueError(
            "n_timesteps must be less than or equal to the number of" +
                "time steps in the data."
        )
    if noise_std is None:
        noise_std = 0
    else:
        data += np.random.normal(0, noise_std, data.shape)
    # Fit the SINDy model
    model = pysindy.SINDy(
        feature_names=feature_names,
        optimizer=pysindy.STLSQ(threshold=threshold)
    )
    multiple_trajectories = n_samples > 1
    # Choose n_samples random sample indices
    random_sample_indices = np.random.choice(
        np.arange(data.shape[0]), n_samples, replace=False
    )
    if multiple_trajectories:
        data_to_fit = list(data[random_sample_indices, :n_timesteps])
    else:
        data_to_fit = data[random_sample_indices, :n_timesteps]
    model.fit(
        data_to_fit,
        t=dt, multiple_trajectories=multiple_trajectories,
    )
    return model
```

Define arrays of sparsity thresholds, trajectory lengths, and noise levels to explore. We will use a single sample for all cases because we suspect that the number of timesteps and the number of samples have similar effects.

```
thresholds = np.linspace(0.1, .5, 11)
n_timesteps = np.flip(np.linspace(1500, 5000, 11, dtype=int))
noise_stds = np.linspace(0, 8, 3)
```

Fit the SINDy model for different parameters:

```
models = np.zeros(
    (len(thresholds), len(n_timesteps), len(noise_stds)), dtype=object
)
for i, threshold in enumerate(thresholds):
    for j, n_t in enumerate(n_timesteps):
        for k, noise_std in enumerate(noise_stds):
            models[i, j, k] = fit_sindy(
                data, dt, feature_names=["x", "y", "z"],
                threshold=threshold, n_timesteps=n_t,
                noise_std=noise_std, n_samples=1,
            )
            # print(f"\nThreshold = {threshold}, n_timesteps = {n_t}, noise_std = {noise_st
            # models[i, j, k].print()
```

```
/Users/ricopicone/anaconda3/envs/595/lib/python3.11/site-packages/pysindy/optimizers/stlsq.p
  warnings.warn(

/Users/ricopicone/anaconda3/envs/595/lib/python3.11/site-packages/pysindy/optimizers/stlsq.p
  warnings.warn(
/Users/ricopicone/anaconda3/envs/595/lib/python3.11/site-packages/pysindy/optimizers/stlsq.p
  warnings.warn(

/Users/ricopicone/anaconda3/envs/595/lib/python3.11/site-packages/pysindy/optimizers/stlsq.p
```

```
  warnings.warn(
```

Extract the coefficients of the identified dynamics:

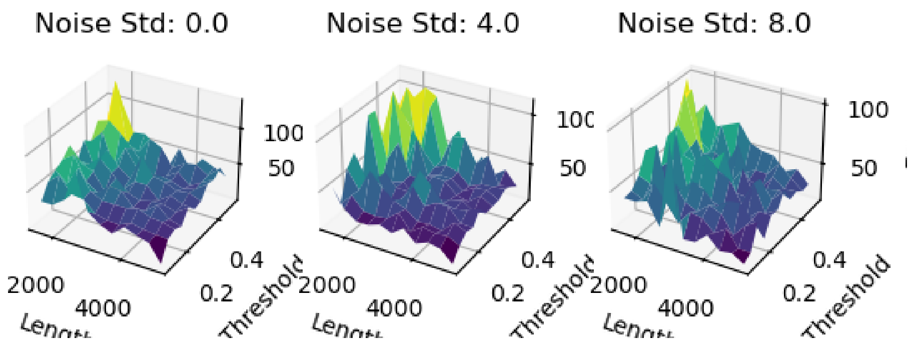
```
coefficients = np.zeros(
    (len(thresholds), len(n_timesteps), len(noise_stds), 3, 10)
)
for i in range(len(thresholds)):
    for j in range(len(n_timesteps)):
        for k in range(len(noise_stds)):
            coefficients[i, j, k] = models[i, j, k].coefficients()
```

Compute the absolute error between the identified coefficients and the true coefficients of the Rossler system:

```
true_coefficients = np.array([
    [0., 0.,  -1. ,  -1., 0., 0., 0., 0., 0., 0.],
    [0., 1., a, 0., 0., 0., 0., 0., 0. , 0.],
    [b, 0., 0. , -c, 0., 0., 1., 0., 0., 0.]
])  # True coefficients of the Rossler system
errors = np.zeros(
    (len(thresholds), len(n_timesteps), len(noise_stds), 3, 10)
)  # Array to store the errors

for i in range(len(thresholds)):
    for j in range(len(n_timesteps)):
        for k in range(len(noise_stds)):
            errors[i, j, k] = np.abs(
                coefficients[i, j, k] - true_coefficients
            )
```

Plot the errors. For the first 3 values of noise, create a 3D surf plot of the error for each coefficient as a function of the trajectory length and the sparsity threshold:

```
fig, axes = plt.subplots(1, 3, subplot_kw={"projection": "3d"})
for i in range(3):
    x, y = np.meshgrid(n_timesteps, thresholds)
    z = np.sum(np.sum(errors[:, :, i], axis=-1), axis=-1)
    ax = axes[i]
    ax.plot_surface(x, y, z, cmap='viridis')
    ax.set_xlabel('Length')
    ax.set_ylabel('Threshold')
    ax.set_zlabel('Error')
    ax.set_title(f'Noise Std: {noise_stds[i]}')
plt.show()
```



BOUNTY: The error is small for low noise, low sparsity threshold, and high trajectory length. I don't understand why the error is so small for the best set of parameters (low noise, low sparsity threshold, and high trajectory length). If I change the best set of parameters to be something else, I get a similarly small error there, but not in the previous best set of parameters. This must be a bug, but I can't find it.

# Brunton and Kutz Problem 7.8: Markov Chain Modeling and DMD

Source Filename: /main.py

Rico A. R. Picone

First, import the necessary libraries:

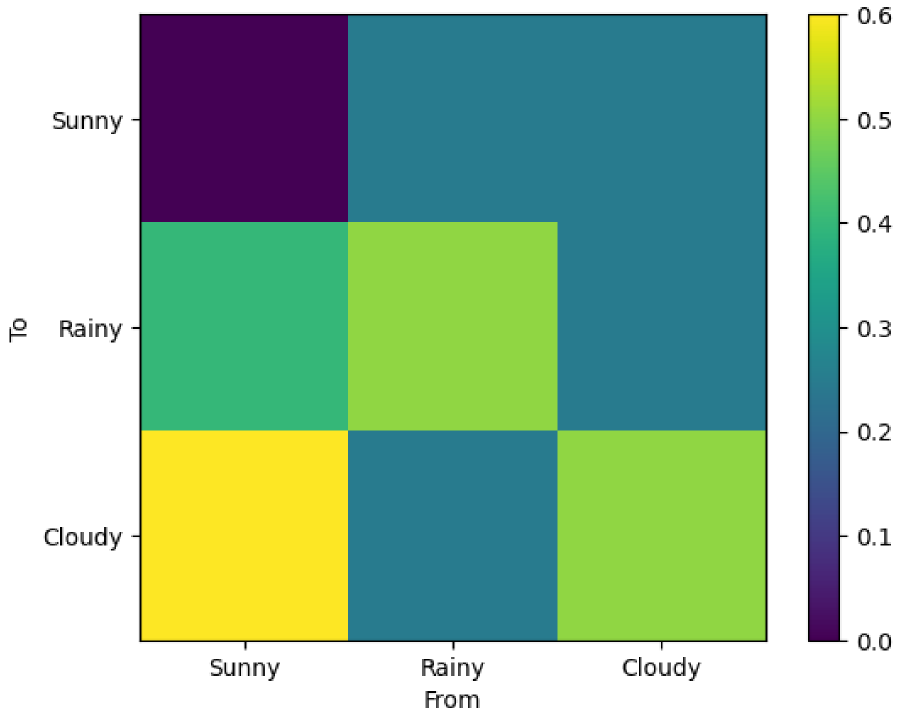
```python
import numpy as np
import matplotlib.pyplot as plt
```

Next, define the Markov chain transition matrix:

```python
P = np.array([
    [0.0, 0.25, 0.25], [0.40, 0.50, 0.25], [0.60, 0.25, 0.50]
])
```

As the footnote in the exercise states, this is the transpose of the more common Russian standard notation. This means that the $(i, j)$ element of the matrix is the probability of transitioning from state $j$ to state $i$. We can visualize the matrix using a heatmap:

```python
fig, ax = plt.subplots()
im = ax.imshow(P, cmap="viridis")
ax.set_xticks([0, 1, 2])
ax.set_yticks([0, 1, 2])
ax.set_xticklabels(["Sunny", "Rainy", "Cloudy"])
ax.set_yticklabels(["Sunny", "Rainy", "Cloudy"])
ax.set_xlabel("From")
ax.set_ylabel("To")
fig.colorbar(im)
plt.draw()
```



We see that the probability of transitioning from a sunny day to a sunny day is 0, to a rainy day is 0.4, and to a cloudy day is 0.6. Note that the columns of the matrix sum to 1 (i.e., it has to be sunny, rainy, or cloudy at any given time step).

## Long-term distribution

The long-term distribution of the Markov chain is the eigenvector of the transition matrix corresponding to the eigenvalue of 1. We can find this eigenvector using the `np.linalg.eig` function:

```
eigenvalues, eigenvectors = np.linalg.eig(P)
idx = np.argmin(np.abs(eigenvalues - 1.0))  # Index of e-val close to 1
v = np.real(eigenvectors[:, idx])  # Corresponding eigenvector
v = v / np.sum(v)  # Normalize eigenvector
print(f"Long-term distribution: {v}")
print(f"Interpretation:\nSunny probability:{v[0]:.2f}\n"+
      f"Rainy probability: {v[1]:.2f}\n"
      f"Cloudy probability: {v[2]:.2f}")
```

```
Long-term distribution: [0.2        0.37333333 0.42666667]
Interpretation:
Sunny probability:0.20
```

```
Rainy probability: 0.37
Cloudy probability: 0.43
```

## Simulation

Define an observer function that samples the state of the Markov chain:

```
def observer(x):
    """Observe the state of the Markov chain"""
    random_index = np.random.choice(range(3), p=x)
    x = np.zeros(x.shape)
    x[random_index] = 1.0
    return x
```

Simulate a random instance (i.e., one corresponding to a random initial condition) of the process, observing the state at each time step:

```
T = 5000  # Number of time steps
x = np.zeros((T, 3))  # State at each time step
np.random.seed(0)  # Seed random number generator for reproducibility
initial_nonzero_index = np.random.randint(0, x.shape[1])
x[0, initial_nonzero_index] = 1.0  # Initial condition
for t in range(0, T-1):
    x_pre_observation = P @ x[t]
    x[t+1] = observer(x_pre_observation)
print(f"First 10 states: {x[:10]}")
```

```
First 10 states: [[1. 0. 0.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 1. 0.]
 [0. 1. 0.]
 [1. 0. 0.]
 [0. 1. 0.]]
```

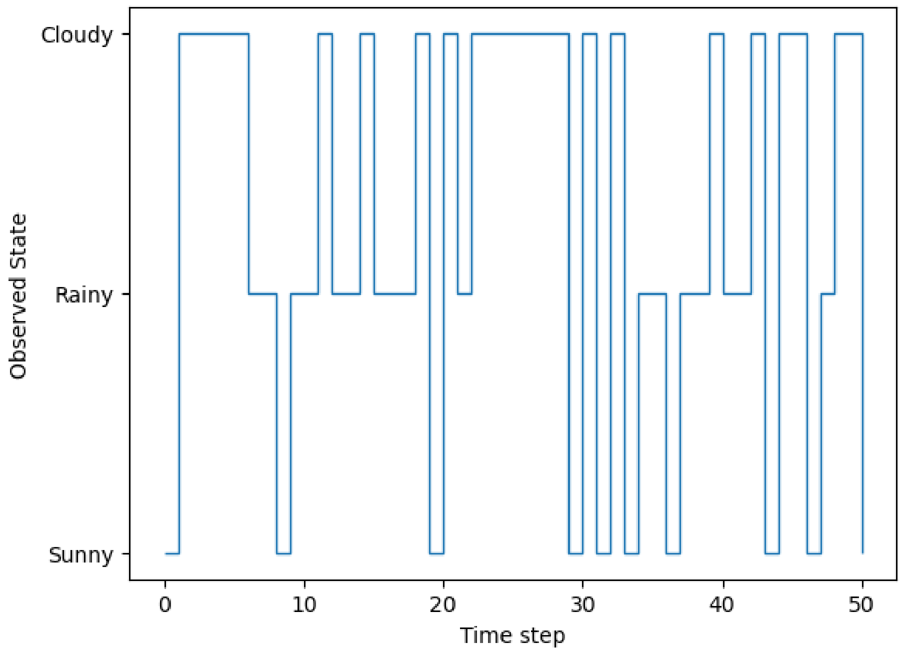
Visualize the simulation. First, convert the state to values that can be plotted:

```
x_visualize = np.argmax(x, axis=1)  # Integer representation of states
```

Plot the first 50 observed states:

```
n_plot = 50
fig, ax = plt.subplots()
ax.stairs(
    x_visualize[:n_plot],
    edges=np.arange(0, len(x_visualize[:n_plot])+1)
)
```

```
ax.set_xlabel("Time step")
ax.set_ylabel("Observed State")
ax.set_yticks([0, 1, 2])
ax.set_yticklabels(["Sunny", "Rainy", "Cloudy"])
ax.set_ylim(-0.1, 2.1)
plt.draw()
```



## Dynamic Mode Decomposition (DMD)

Define the exact DMD function from Brunton and Kutz (2022):

```
def DMD(X,Xprime,r):
    # Step 1
    U, Sigma, VT = np.linalg.svd(X, full_matrices=0)
    Ur = U[:, :r]
    Sigmar = np.diag(Sigma[:r])
    VTr = VT[:r, :]
    # Step 2
    Atilde = np.linalg.solve(Sigmar.T, (Ur.T @ Xprime @ VTr.T).T).T
    # Step 3
    Lambda, W = np.linalg.eig(Atilde)
    Lambda = np.diag(Lambda)
    # Step 4
    Phi = Xprime @ np.linalg.solve(Sigmar.T, VTr).T @ W
    alpha1 = Sigmar @ VTr[:,0]
    b = np.linalg.solve(W @ Lambda, alpha1)
    return Phi, Lambda, b
```

Construct the data matrices $X$ and $X'$:

```
X = x[:-1].T
Xprime = x[1:].T
```

Compute the DMD modes:

```
r = 3  # Number of modes
Phi, Lambda, b = DMD(X, Xprime, r)
print(f"Phi:\n{Phi}")
print(f"Lambda:\n{Lambda}")
print(f"b:\n{b}")
```

```
Phi:
[[-0.33268176 -0.01006597 -0.20196528]
 [-0.61594571  0.18226477  0.05599596]
 [-0.71409645 -0.1721988   0.14596931]]
Lambda:
[[ 1.          0.          0.        ]
 [ 0.          0.25094661  0.        ]
 [ 0.          0.         -0.25540666]]
b:
[-0.60142276 -0.82832248 -3.91938537]
```

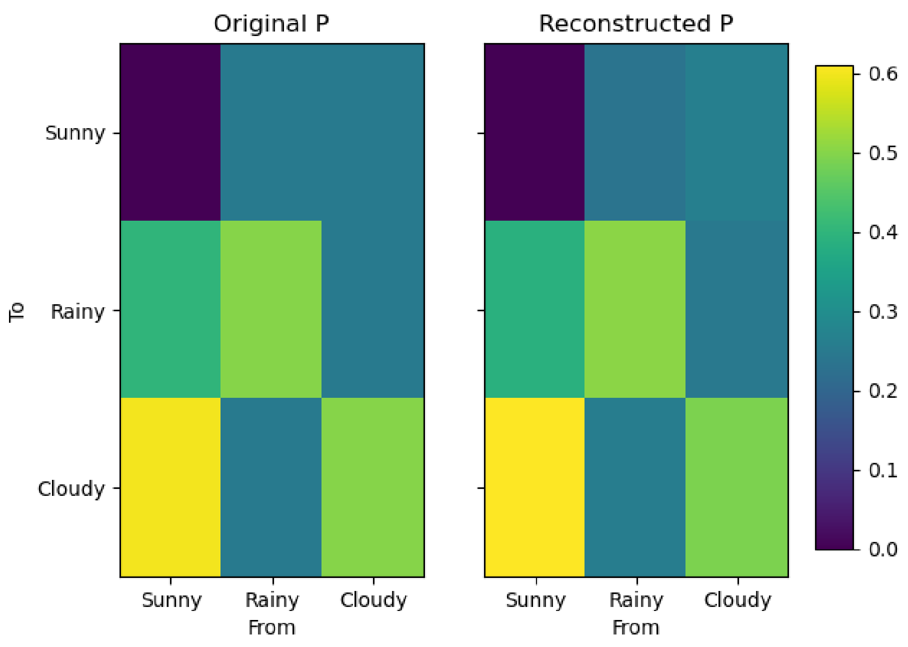
Reconstruct the P matrix from the DMD modes:

```
print(f"Phi shape: {Phi.shape}")
print(f"Lambda shape: {Lambda.shape}")
P_dmd = Phi @ Lambda @ np.linalg.inv(Phi)
print(f"P:\n{P}")
print("P_dmd:\n" +
    np.array2string(
        P_dmd, precision=2,
        floatmode="fixed", suppress_small=True
    )
)
```

```
Phi shape: (3, 3)
Lambda shape: (3, 3)
P:
[[0.   0.25 0.25]
 [0.4  0.5  0.25]
 [0.6  0.25 0.5 ]]
P_dmd:
[[0.00 0.23 0.26]
 [0.39 0.50 0.25]
 [0.61 0.26 0.49]]
```

Visually compare the original and reconstructed transition matrices:

```
zmin = min(np.min(P), np.min(P_dmd))
zmax = max(np.max(P), np.max(P_dmd))
fig, ax = plt.subplots(1, 2, sharex=True, sharey=True)
im0 = ax[0].imshow(
    P, cmap="viridis", vmin=zmin, vmax=zmax, aspect='auto'
)
ax[0].set_title("Original P")
im1 = ax[1].imshow(
    P_dmd, cmap="viridis", vmin=zmin, vmax=zmax, aspect='auto'
)
ax[1].set_title("Reconstructed P")
ax[0].set_ylabel("To")
for a in ax:
    a.set_xticks([0, 1, 2])
    a.set_yticks([0, 1, 2])
    a.set_xticklabels(["Sunny", "Rainy", "Cloudy"])
    a.set_yticklabels(["Sunny", "Rainy", "Cloudy"])
    a.set_xlabel("From")
    a.set_xlim(-0.5, 2.5)
    a.set_ylim(2.5, -0.5)
fig.subplots_adjust(right=0.85)
cbar_ax = fig.add_axes([0.88, 0.15, 0.04, 0.7])
fig.colorbar(im1, cax=cbar_ax)
plt.show()
```



We appear to achieve a good reconstruction of the transition matrix using DMD. The DMD modes can be used to predict the future state of the system.

Perhaps a numerical evaluation of the prediction error would be useful. One approach is to compute the Eucldean distance between the eigenvectors of the original and reconstructed transition matrices. We have already computed both sets of eigenvectors, so we can proceed to write a function to compare the eigenvectors of two matrices and compute the error. The function will normalize the eigenvectors and compute the Euclidean distance between them. However, the ordering of the eigenvectors may differ between the two matrices, so we will need to account for this:

```
def eigenvector_error(M1: np.ndarray, M2: np.ndarray):
    """Compute the distance between eigenvectors of two matrices

    The eigenvectors are normalized before computing the error.
    The error corresponds to the Euclidean distance between the
    eigenvectors of two matrices. The indexing and sign of the
    eigenvectors may differ between the two matrices, so the
    columns are matched. The output is the error for each,
    using the index of the M1 eigenvectors.

    Parameters
    ----------
    M1 : np.ndarray
    Matrix of eigenvectors
M2 : np.ndarray
    Matrix of eigenvectors

Returns
-------
error : np.ndarray
    Error for each eigenvector
M1_out : np.ndarray
    Normalized matrix of eigenvectors from M1
M2_out : np.ndarray
    Normalized and potentially reordered (and sign-flipped)
    matrix of eigenvectors from M2
"""
M1 = M1 / np.linalg.norm(M1, ord=2, axis=0)  # Normalize e-vecs
M2 = M2 / np.linalg.norm(M2, ord=2, axis=0)
M2_out = M2.copy()  # Copy M2 for matching
# Match e-vecs of M1 to e-vecs of M2
error = np.zeros(M1.shape[1])  # Initialize error
used_indices = []  # Indices of M2 columns already matched
for i in range(M1.shape[1]):  # Loop over columns of M1
    v = M1[:, [i]]  # Current e-vec of M1
    distances1 = np.linalg.norm(M2 - v, ord=2, axis=0)
    distances2 = np.linalg.norm(M2 + v, ord=2, axis=0)
    distances = np.vstack([distances1, distances2])
    di_min = np.unravel_index(
        np.argmin(distances, axis=None), distances.shape
    )  # Index of minimum distance
    if di_min[1] in used_indices:  # Column already matched
        raise RuntimeError(
            f"Column {di_min[1]} already matched so 2 columns" +
            "are too close for this method"
        )
    else:
        used_indices.append(di_min[1])
    if di_min[1] != i:  # Columns do not match
        # Swap columns of M2 to match e-vecs of M1
        M2_out[:, [i, di_min[1]]] = \
            M2[:, [di_min[1], i]]
    if di_min[0] == 0:  # No sign change
        error[i] = distances1[di_min[1]]
    else:  # Sign change
        M2_out[:, i] = -M2_out[:, i]
        error[i] = distances2[di_min[1]]
return error, M1, M2_out
```

Now apply the function to the eigenvectors of the original transition matrix and the DMD eigenvectors of Φ:

```
error, MP, MP_dmd = eigenvector_error(eigenvectors, Phi)
print("Eigenvector errors (percent): " +
    f"{np.array2string(error*100, precision=2, suppress_small=True)}"
)
print("Original eigenvectors (normalized):\n" +
    f"{np.array2string(MP, precision=2, suppress_small=True)}"
)
print("DMD eigenvectors (normalized, reconstructed):\n" +
    f"{np.array2string(MP_dmd, precision=2, suppress_small=True)}"
)
```

```
Eigenvector errors (percent): [0.67 2.48 4.91]
Original eigenvectors (normalized):
[[-0.33 -0.8   0.  ]
 [-0.62  0.24 -0.71]
 [-0.71  0.56  0.71]]
DMD eigenvectors (normalized, reconstructed):
[[-0.33 -0.79  0.04]
 [-0.62  0.22 -0.73]
 [-0.71  0.57  0.69]]
```

## Check to see if the columns of the reconstructed transition matrix sum to 1, as they should:

```
print("Sum of columns of P_dmd: " +
    np.array2string(np.sum(P_dmd, axis=0), precision=16)
)
```

```
Sum of columns of P_dmd: [1.0000000000000022 0.9999999999999997 0.9999999999999599]
```

These sums are remarkably close to 1, which means that the reconstructed transition matrix is a valid Markov chain transition matrix.

## Using the PyDMD Package

The PyDMD package provides a convenient implementation of DMD. We can use it to compare the results with our implementation.

```
from pydmd import DMD
dmd = DMD(svd_rank=3, exact=True)
dmd.fit(X=X, Y=Xprime)
```

```
<pydmd.dmd.DMD at 0x123722110>
```

Extract the DMD modes, eigenvalues, and mode amplitudes:

```
Phi_pydmd = dmd.modes
Lambda_pydmd = np.diag(dmd.eigs)
b_pydmd = dmd.amplitudes
```

Compare the DMD modes from PyDMD with our implementation

```
error_pydmd, _, Phi_pydmd = eigenvector_error(eigenvectors, Phi_pydmd)
print("PyDMD eigenvectors (normalized):\n" +
    np.array2string(Phi_pydmd, precision=2, suppress_small=True)
)
print("PyDMD eigenvector errors (percent): " +
    np.array2string(error_pydmd*100, precision=2, suppress_small=True)
)
```

```
PyDMD eigenvectors (normalized):
[[-0.33 -0.79  0.04]
 [-0.62  0.22 -0.73]
 [-0.71  0.57  0.69]]
PyDMD eigenvector errors (percent): [0.67 2.48 4.91]
```

The DMD modes from PyDMD are very similar to those obtained using our implementation. The error is very low, indicating that the two sets of modes are very close.