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:

```
import numpy as np
from scipy.special import comb
import matplotlib.pyplot as plt
# __import__("matplotlib").use("TkAgg")  # Use this to rotate the 3D plot
# However, it doesn't publish well.
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 regularization parameter λ, the number of samples, and the trajectory length.

Define the Rossler system:

```
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:

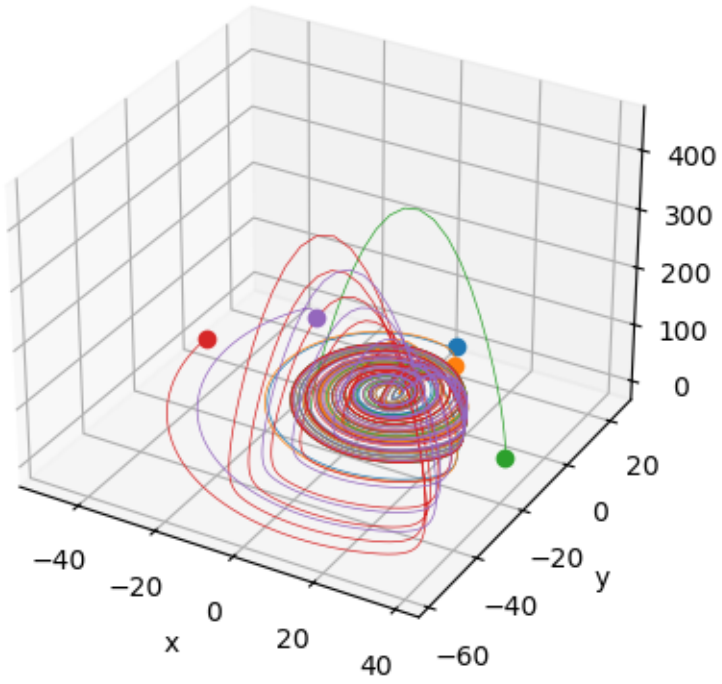
```
def generate_data(n_samples, n_timesteps, dt, a=0.2, b=0.2, c=14):
    """
    Generate training data for the Rossler system
    """
    t = np.linspace(0, (n_timesteps-1)*dt, n_timesteps)  # Time array
    x = np.zeros((n_samples, n_timesteps, 3))  # Array to store the data
    for i in range(n_samples):
        np.random.seed(i*3)  # For reproducibility
        x0 = np.random.uniform(-50, 50, 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 = 5  # Number of samples
n_t = 10000  # Number of time steps
dt = 0.01  # Time step
time = np.linspace(0, (n_t-1)*dt, n_t)  # Time array
data = generate_data(n_samples, n_t, dt)
```

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, x^2, y^2, z^2, xy, xz, yz]
    """
    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[:, 1]**2
    Theta[:, 6] = X[:, 2]**2
    Theta[:, 7] = X[:, 0]*X[:, 1]  # Cross terms
    Theta[:, 8] = X[:, 0]*X[:, 2]
    Theta[:, 9] = X[:, 1]*X[:, 2]
    return Theta
```

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

```
def sparsifyDynamics(Theta, dXdt, lamb, 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) < lamb  # 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
lamb = 0.1  # Regularization parameter
Xi = sparsifyDynamics(Theta(X), dX, lamb, 3)
```

Print the identified terms:

```
print(Xi.shape)
print(Xi)
```

```python
def print_terms(Xi):
    """Print the identified terms"""
    terms = [
        '1', 'x', 'y', 'z', 'x^2', 'y^2', 'z^2', 'xy', 'xz', 'yz'
    ]
    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)
```

```
(10, 3)
[[  0.    0.    0.2]
 [  0.    1.    0. ]
 [ -1.    0.2   0. ]
 [ -1.    0.  -14. ]
 [  0.    0.    0. ]
 [  0.    0.    0. ]
 [  0.    0.    0. ]
 [  0.    0.    0. ]
 [  0.    0.    1. ]
 [  0.    0.    0. ]]
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 regularization parameter λ, number of samples, and trajectory lengths.

Using the `pySINDy` Package