# 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
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:

```
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, seed_offset=0):
    """
```

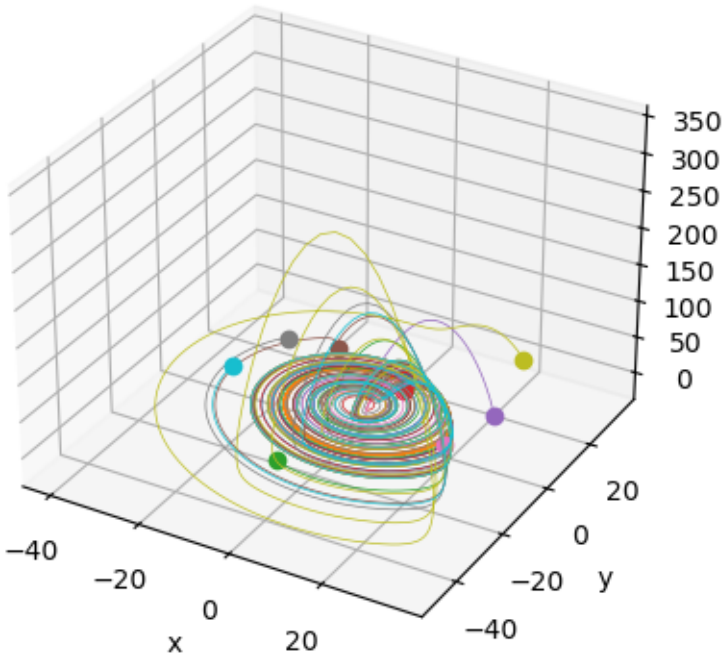
```
    Generate training data for the Rossler system
    """
    t = np.linspace(0, (n_timesteps-1)*dt, n_timesteps)  # Time array
    x = np.zeros((n_samples, n_timesteps, 3))  # Array to store the data
    for i in range(n_samples):
        np.random.seed(i + seed_offset)  # For reproducibility
        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 simple 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:

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

```
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)
```

```
(10, 3)
[[  0.    0.    0.2]
 [  0.    1.    0. ]
 [ -1.    0.2   0. ]
 [ -1.    0.  -14. ]
 [  0.    0.    0. ]
 [  0.    0.    0. ]
 [  0.    0.    1. ]
 [  0.    0.    0. ]
 [  0.    0.    0. ]
 [  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 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 = [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(0.01, 0.1, 11)
n_timesteps = np.flip(np.linspace(1500, 5000, 11, dtype=int))
noise_stds = np.linspace(0, 4, 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()
```

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()
```

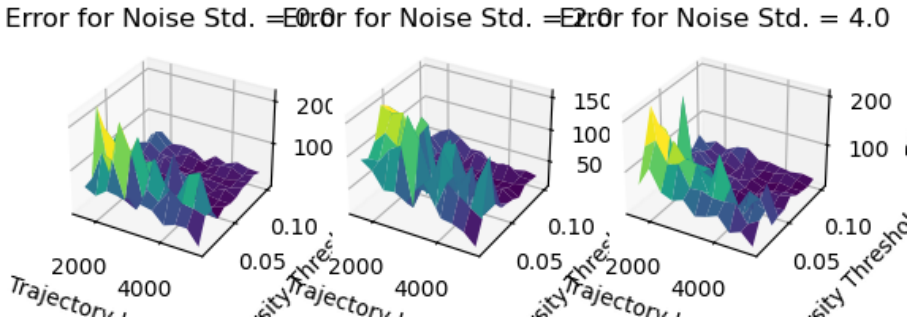
Compute the absolute error between the identified coefficients and the true coefficients of the Rossler system:

```
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('Trajectory Length')
    ax.set_ylabel('Sparsity Threshold')
    ax.set_zlabel('Error')
    ax.set_title(f'Error for Noise Std. = {noise_stds[i]}')
plt.show()
```




The error increases with the noise standard deviation and sparsity threshold (at a certain point, the model will get too sparse and zero out all coefficients). Similarly, the error decreases with the trajectory length. However, I don't quite 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.