```
import numpy as np
from scipy.integrate import solve_ivp
import matplotlib.pyplot as plt
```

Check the eigenvalues of the Jacobian matrix

```
# Define the Jacobian matrix of the system
def jacobian_matrix(x, y):
    # Elements of the Jacobian matrix
    df_dx = -0.1 * 3 * x**2  # Partial derivative of dx/dt with 
respect to x
    df_dy = 2 * 3 * y**2     # Partial derivative of dx/dt with 
respect to y
    dg_dx = -2 * 3 * x**2    # Partial derivative of dy/dt with 
respect to x
    dg_dy = -0.1 * 3 * y**2  # Partial derivative of dy/dt with 
respect to y
    # Construct the Jacobian matrix
    return np.array([[df_dx, df_dy], [dg_dx, dg_dy]])

# Initial conditions or equilibrium point (e.g., x=0.1, y=0.1 as 
given)
x0, y0 = 0.1, 0.1

# Calculate the Jacobian matrix at this point
J = jacobian_matrix(x0, y0)

# Calculate the eigenvalues of the Jacobian matrix
eigenvalues = np.linalg.eigvals(J)

# Display the eigenvalues and check if the real parts are negative
print("Eigenvalues of the Jacobian matrix:", eigenvalues)
print("Real parts of eigenvalues:", np.real(eigenvalues))

# Check stability based on the real parts of the eigenvalues
if np.all(np.real(eigenvalues) < 0):
    print("The system is stable (all eigenvalues have negative real 
parts).")
else:
    print("The system may be unstable (some eigenvalues have non-
negative real parts).")
```

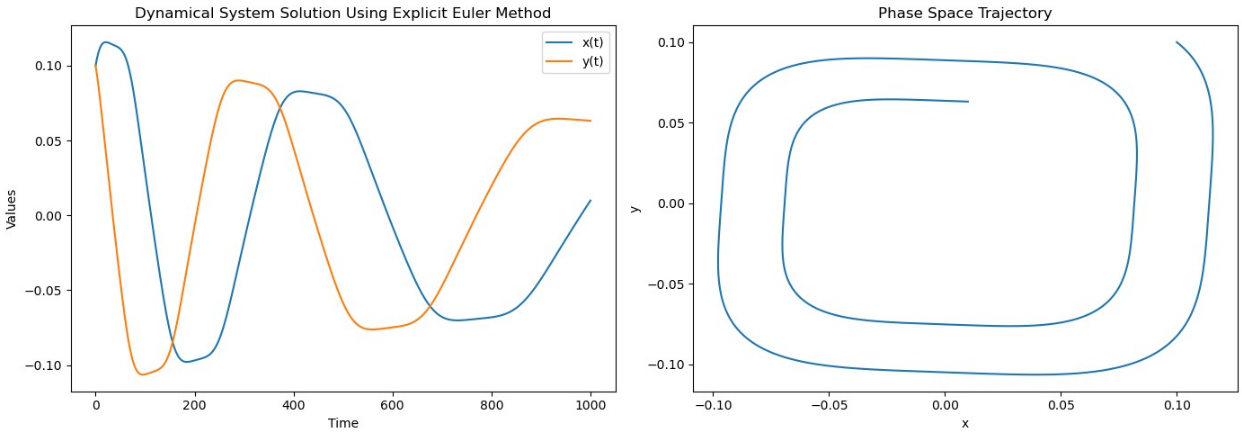
```
Eigenvalues of the Jacobian matrix: [-0.003+0.06j -0.003-0.06j]
Real parts of eigenvalues: [-0.003 -0.003]
The system is stable (all eigenvalues have negative real parts).
```

Generate Training Data by Numerically Solving the ODE Equation

1. Generate the data using an explicit scheme

```python
# Define the system of differential equations
def dynamical_system(z):
    x, y = z
    dxdt = -0.1 * x**3 + 2 * y**3
    dydt = -2 * x**3 - 0.1 * y**3
    return np.array([dxdt, dydt])

# Initial conditions
x0, y0 = 0.1, 0.1
initial_conditions = np.array([x0, y0])

# Parameters for the explicit Euler method
t_start = 0
t_end = 1000
dt = 0.00001  # Small time step for numerical stability
n_steps = int((t_end - t_start) / dt)
time_points = np.linspace(t_start, t_end, n_steps)

# Arrays to store the results of x(t) and y(t)
x_values = np.zeros(n_steps)
y_values = np.zeros(n_steps)
x_values[0], y_values[0] = initial_conditions

# Explicit Euler method loop
z = initial_conditions
for i in range(1, n_steps):
    z = z + dt * dynamical_system(z)  # Update based on current state
    x_values[i], y_values[i] = z  # Store updated values

# Plot x(t) and y(t) over time, and phase space trajectory in a single
row
plt.figure(figsize=(14, 5))

# Plot x(t) and y(t) over time
plt.subplot(1, 2, 1)
plt.plot(time_points, x_values, label='x(t)')
plt.plot(time_points, y_values, label='y(t)')
plt.xlabel('Time')
plt.ylabel('Values')
plt.legend()
plt.title('Dynamical System Solution Using Explicit Euler Method')

# Plot phase space trajectory
plt.subplot(1, 2, 2)
plt.plot(x_values, y_values)
plt.xlabel('x')
plt.ylabel('y')
plt.title('Phase Space Trajectory')

# Show the plots
```

```
plt.tight_layout()
plt.show()
```

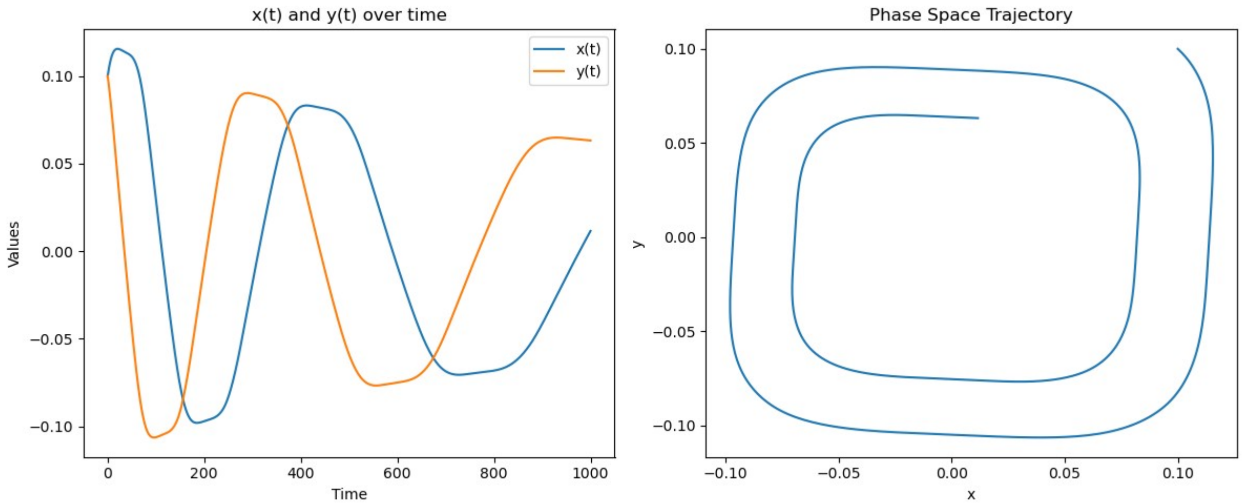


Generate Training Data by Numerically Solving the ODE Equation

1. The RK45 method, an implicit, adaptive-step Runge-Kutta method

```
# Define the dynamical system
def dynamical_system(t, z):
    x, y = z
    dxdt = -0.1 * x**3 + 2 * y**3
    dydt = -2 * x**3 - 0.1 * y**3
    return [dxdt, dydt]

# Initial conditions
x0, y0 = 0.1, 0.1
initial_conditions = [x0, y0]

# Time span for the solution
t_span = (0, 1000)  # simulate from t=0 to t=10
t_eval = np.linspace(*t_span, 1000)

# Solve the ODE
solution = solve_ivp(dynamical_system, t_span, initial_conditions,
t_eval=t_eval, method='RK45')

# Extract results
t_data = solution.t
x_data = solution.y[0]
y_data = solution.y[1]

# Plot the results
plt.figure(figsize=(12, 5))

# Plot x and y over time
plt.subplot(1, 2, 1)
plt.plot(t_data, x_data, label="x(t)")
```

```python
plt.plot(t_data, y_data, label="y(t)")
plt.xlabel("Time")
plt.ylabel("Values")
plt.legend()
plt.title("x(t) and y(t) over time")

# Plot phase space trajectory
plt.subplot(1, 2, 2)
plt.plot(x_data, y_data)
plt.xlabel("x")
plt.ylabel("y")
plt.title("Phase Space Trajectory")
plt.tight_layout()
plt.show()
```



```
import numpy as np
import pysindy as ps
from scipy.integrate import solve_ivp
import matplotlib.pyplot as plt
```

Use SINDy to Learn the Dynamical System

- 2 (a) - (c)

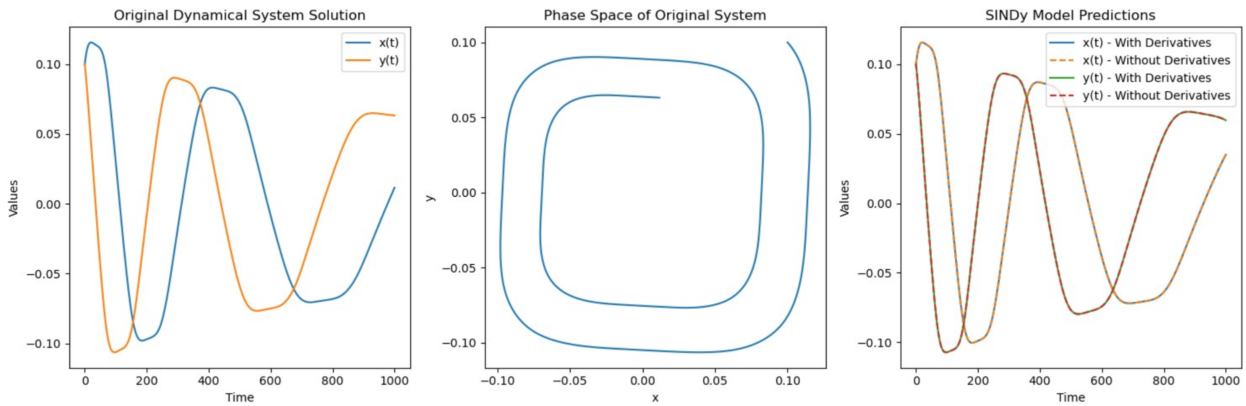
```
# Define the original dynamical system for generating training data
def dynamical_system(t, z):
    """Dynamical system with cubic terms for x and y."""
    x, y = z
    dxdt = -0.1 * x**3 + 2 * y**3
    dydt = -2 * x**3 - 0.1 * y**3
    return [dxdt, dydt]

# Generate training data using solve_ivp
x0, y0 = 0.1, 0.1  # Initial conditions
t_span = (0, 1000)  # Time span for simulation
t_eval = np.linspace(*t_span, 10000)  # Evaluation time points
solution = solve_ivp(dynamical_system, t_span, [x0, y0],
t_eval=t_eval, method='RK45')
x_data = solution.y.T  # Transpose to have time series data in rows
t_data = solution.t  # Time data

# Part 2(a): Construct a library of polynomial functions up to the 5th
order
library = ps.PolynomialLibrary(degree=5)  # Polynomial terms up to 5th
degree

# Define a SINDy optimizer with a smaller threshold to prevent over-
sparsification
optimizer = ps.STLSQ(threshold=0.0005)  # Reduced threshold for better
model fitting

# Part 2(b)(i): Fit the model using x and ẋ as data
model_with_derivatives = ps.SINDy(feature_library=library,
optimizer=optimizer)
model_with_derivatives.fit(x_data, t=t_data)  # Fitting the model
using time derivatives directly
print("Model with derivatives (x and ẋ as data):")
model_with_derivatives.print()

# Part 2(b)(ii): Fit the model using only x as data (finite difference
for ẋ)
model_with_x_only = ps.SINDy(
    feature_library=library,
    optimizer=optimizer,
    differentiation_method=ps.FiniteDifference()  # Use finite
```

```
difference to estimate ẋ from x data
)
model_with_x_only.fit(x_data, t=t_data)

# Part 2(c): Report the fitted models
print("Model without derivatives (only x as data):")
model_with_x_only.print()
```

```
Model with derivatives (x and ẋ as data):
(x0)' = -0.001 x1^2 + -0.108 x0^3 + -0.004 x0^2 x1 + 1.996 x1^3 +
0.087 x1^4
(x1)' = -0.001 x1 + -0.001 x0 x1 + 0.001 x1^2 + -1.994 x0^3 + 0.037
x0^2 x1 + 0.007 x0 x1^2 + -0.017 x0^4
Model without derivatives (only x as data):
(x0)' = -0.001 x1^2 + -0.108 x0^3 + -0.004 x0^2 x1 + 1.996 x1^3 +
0.087 x1^4
(x1)' = -0.001 x1 + -0.001 x0 x1 + 0.001 x1^2 + -1.994 x0^3 + 0.037
x0^2 x1 + 0.007 x0 x1^2 + -0.017 x0^4
```

```
# Plot the original dynamical system solution (training data)
plt.figure(figsize=(15, 5))

# Plot x(t) and y(t) over time
plt.subplot(1, 3, 1)
plt.plot(t_data, x_data[:, 0], label='x(t)')
plt.plot(t_data, x_data[:, 1], label='y(t)')
plt.xlabel('Time')
plt.ylabel('Values')
plt.legend()
plt.title('Original Dynamical System Solution')

# Phase space plot for the original system (x vs y)
plt.subplot(1, 3, 2)
plt.plot(x_data[:, 0], x_data[:, 1])
plt.xlabel('x')
plt.ylabel('y')
plt.title('Phase Space of Original System')

# Compare the models fitted by SINDy with and without derivatives
# Use the SINDy models to predict over the same time span
sindy_x_with_derivatives = model_with_derivatives.simulate(x_data[0],
t_data)
sindy_x_without_derivatives = model_with_x_only.simulate(x_data[0],
t_data)

# Plot x(t) and y(t) predictions from SINDy model with derivatives
plt.subplot(1, 3, 3)
plt.plot(t_data, sindy_x_with_derivatives[:, 0], label='x(t) - With
Derivatives')
plt.plot(t_data, sindy_x_without_derivatives[:, 0], label='x(t) -
```

```
Without Derivatives', linestyle='--')
plt.plot(t_data, sindy_x_with_derivatives[:, 1], label='y(t) - With 
Derivatives')
plt.plot(t_data, sindy_x_without_derivatives[:, 1], label='y(t) - 
Without Derivatives', linestyle='--')
plt.xlabel('Time')
plt.ylabel('Values')
plt.legend()
plt.title('SINDy Model Predictions')

plt.tight_layout()
plt.show()
```



Use SINDy to Learn the Dynamical System

- 2 (d)

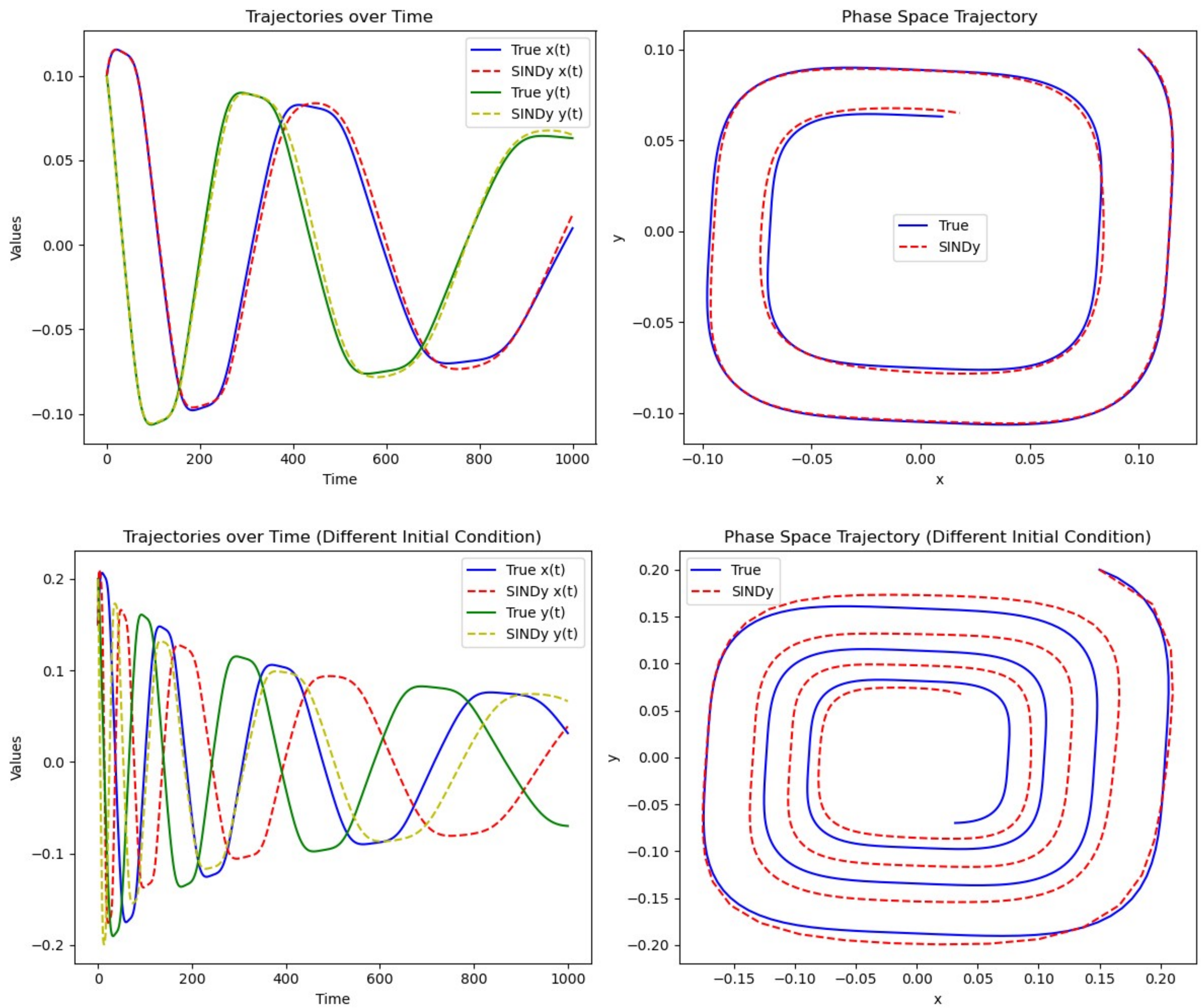
```
# Define the dynamical system (cubic damped SHO)
def cubic_damped_SHO(t, z):
    x, y = z
    dxdt = -0.1 * x**3 + 2 * y**3
    dydt = -2 * x**3 - 0.1 * y**3
    return [dxdt, dydt]

# Generate training data
dt = 0.0001  # time step
t_train = np.arange(0, 1000, dt)  # time points
t_train_span = (t_train[0], t_train[-1])  # time span for the ODE 
solver
x0_train = [0.1, 0.1]  # initial conditions
integrator_keywords = {'rtol': 1e-12, 'atol': 1e-12}  # integrator 
tolerance settings

# Solve the ODE to generate training data
x_train = solve_ivp(cubic_damped_SHO, t_train_span, x0_train, 
t_eval=t_train, **integrator_keywords).y.T

# Define a custom function library
```

```
def make_custom_functions():
    functions = [
        lambda x: 1,                # f1, constant term
        lambda x: x,                # f2(x), linear in x
        lambda y: y,                # f3(y), linear in y
        lambda x: x**2,             # f4(x^2), quadratic in x
        lambda y: y**2,             # f5(y^2), quadratic in y
        lambda x: x**4,             # f6(x^4), quartic in x
        lambda y: y**4,             # f7(y^4), quartic in y
        lambda x: x**5,             # f8(x^5), quintic in x
        lambda y: y**5,             # f9(y^5), quintic in y
        lambda x, y: x * y,         # f10(xy), interaction term
        lambda x, y: x**2 * y,      # f11(x^2 * y), mixed quadratic-
cubic term
        lambda x, y: x * y**2,      # f12(x * y^2), mixed cubic-
quadratic term
        lambda x, y: x**2 * y**2,  # f13(x^2 * y^2), mixed quadratic
terms
        lambda x, y: x**4 * y,      # f14(x^4 * y), higher-order mixed
terms
        lambda x, y: x * y**4,      # f15(x * y^4)
        lambda x, y: x**2 * y**3,  # f16(x^2 * y^3)
        lambda x, y: x**3 * y**2,  # f17(x^3 * y^2)
        lambda x, y: x**3 * y,      # f18(x^3 * y)
        lambda x, y: x * y**3       # f19(x * y^3)
    ]
    return functions

# Create the custom library
functions = make_custom_functions()
custom_lib = ps.CustomLibrary(
    library_functions=functions,
    interaction_only=True,  # consider only interaction terms
    include_bias=False  # exclude constant bias term
)

# Define the model and set the optimizer
model = ps.SINDy(
    optimizer=ps.STLSQ(threshold=0.0002),  # threshold for sparse
regression
    feature_library=custom_lib
)

# Fit the model to the training data
model.fit(x_train, t=dt)

# Print the identified model
model.print()
```

```
(x0)' = 0.003 f1(x1) + 0.003 f2(x1) + 0.002 f3(x0) + -0.009 f3(x1) +
0.002 f4(x0) + -0.009 f4(x1) + -0.126 f5(x0) + 1.449 f5(x1) + -0.126
f6(x0) + 1.449 f6(x1) + -4.091 f7(x0) + 67.481 f7(x1) + -4.091 f8(x0)
+ 67.481 f8(x1) + 0.011 f9(x0,x1) + -0.816 f10(x0,x1) + 0.064
f11(x0,x1) + 0.165 f12(x0,x1) + 25.778 f13(x0,x1) + -7.480 f14(x0,x1)
+ 92.894 f15(x0,x1) + -12.132 f16(x0,x1) + -2.347 f18(x0,x1)
(x1)' = -0.004 f1(x0) + -0.004 f2(x0) + -0.017 f3(x0) + -0.017 f4(x0)
+ 1.965 f5(x0) + 0.048 f5(x1) + 1.965 f6(x0) + 0.048 f6(x1) + -58.003
f7(x0) + -1.559 f7(x1) + -58.003 f8(x0) + -1.559 f8(x1) + -0.151
f10(x0,x1) + 1.267 f11(x0,x1) + 2.526 f12(x0,x1) + 22.918 f13(x0,x1) +
-53.447 f14(x0,x1) + 9.822 f15(x0,x1) + -122.207 f16(x0,x1) + -0.580
f17(x0,x1) + 0.109 f18(x0,x1)
```

```
# Define a function to simulate the SINDy model and compare with true
dynamics
def simulate_sindy_model(model, initial_state, t_span, t_eval):
    """Simulates the SINDy model for given initial conditions and time
span."""
    def sindy_ode(t, z):
        return model.predict(z.reshape(1, -1)).flatten()

    # Solve the ODE using the SINDy model
    sindy_solution = solve_ivp(sindy_ode, t_span, initial_state,
t_eval=t_eval)
    return sindy_solution

# Set up time points for evaluation and initial conditions
t_eval = np.linspace(0, 1000, 1000)  # time points for evaluation
initial_state = [0.1, 0.1]  # initial conditions for testing

# Solve the true system for comparison
true_solution = solve_ivp(cubic_damped_SHO, (t_eval[0], t_eval[-1]),
initial_state, t_eval=t_eval, **integrator_keywords)

# Simulate the SINDy model with the same initial condition
sindy_solution = simulate_sindy_model(model, initial_state,
(t_eval[0], t_eval[-1]), t_eval)

# Plot the trajectories over time
plt.figure(figsize=(12, 5))

# Plot x(t) and y(t) trajectories
plt.subplot(1, 2, 1)
plt.plot(t_eval, true_solution.y[0], 'b', label='True x(t)')
plt.plot(t_eval, sindy_solution.y[0], 'r--', label='SINDy x(t)')
plt.plot(t_eval, true_solution.y[1], 'g', label='True y(t)')
plt.plot(t_eval, sindy_solution.y[1], 'y--', label='SINDy y(t)')
plt.xlabel('Time')
plt.ylabel('Values')
plt.legend()
```

```
plt.title('Trajectories over Time')

# Plot phase space trajectory (x vs y)
plt.subplot(1, 2, 2)
plt.plot(true_solution.y[0], true_solution.y[1], 'b', label='True')
plt.plot(sindy_solution.y[0], sindy_solution.y[1], 'r--',
label='SINDy')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.title('Phase Space Trajectory')
plt.tight_layout()
plt.show()

# Try different initial conditions and compare results
new_initial_state = [0.15, 0.2]
true_solution_new_ic = solve_ivp(cubic_damped_SHO, (t_eval[0],
t_eval[-1]), new_initial_state, t_eval=t_eval, **integrator_keywords)
sindy_solution_new_ic = simulate_sindy_model(model, new_initial_state,
(t_eval[0], t_eval[-1]), t_eval)

# Plot for different initial conditions
plt.figure(figsize=(12, 5))

# Plot x(t) and y(t) trajectories with new initial condition
plt.subplot(1, 2, 1)
plt.plot(t_eval, true_solution_new_ic.y[0], 'b', label='True x(t)')
plt.plot(t_eval, sindy_solution_new_ic.y[0], 'r--', label='SINDy
x(t)')
plt.plot(t_eval, true_solution_new_ic.y[1], 'g', label='True y(t)')
plt.plot(t_eval, sindy_solution_new_ic.y[1], 'y--', label='SINDy
y(t)')
plt.xlabel('Time')
plt.ylabel('Values')
plt.legend()
plt.title('Trajectories over Time (Different Initial Condition)')

# Plot phase space trajectory (x vs y) with new initial condition
plt.subplot(1, 2, 2)
plt.plot(true_solution_new_ic.y[0], true_solution_new_ic.y[1], 'b',
label='True')
plt.plot(sindy_solution_new_ic.y[0], sindy_solution_new_ic.y[1],
'r--', label='SINDy')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.title('Phase Space Trajectory (Different Initial Condition)')
plt.tight_layout()
plt.show()
```



Trajectories over Time
True x(t)
SINDy x(t)
True y(t)
SINDy y(t)
Values
Time
Phase Space Trajectory
True
SINDy
y
x
Trajectories over Time (Different Initial Condition)
True x(t)
SINDy x(t)
True y(t)
SINDy y(t)
Values
Time
Phase Space Trajectory (Different Initial Condition)
True
SINDy
y
x

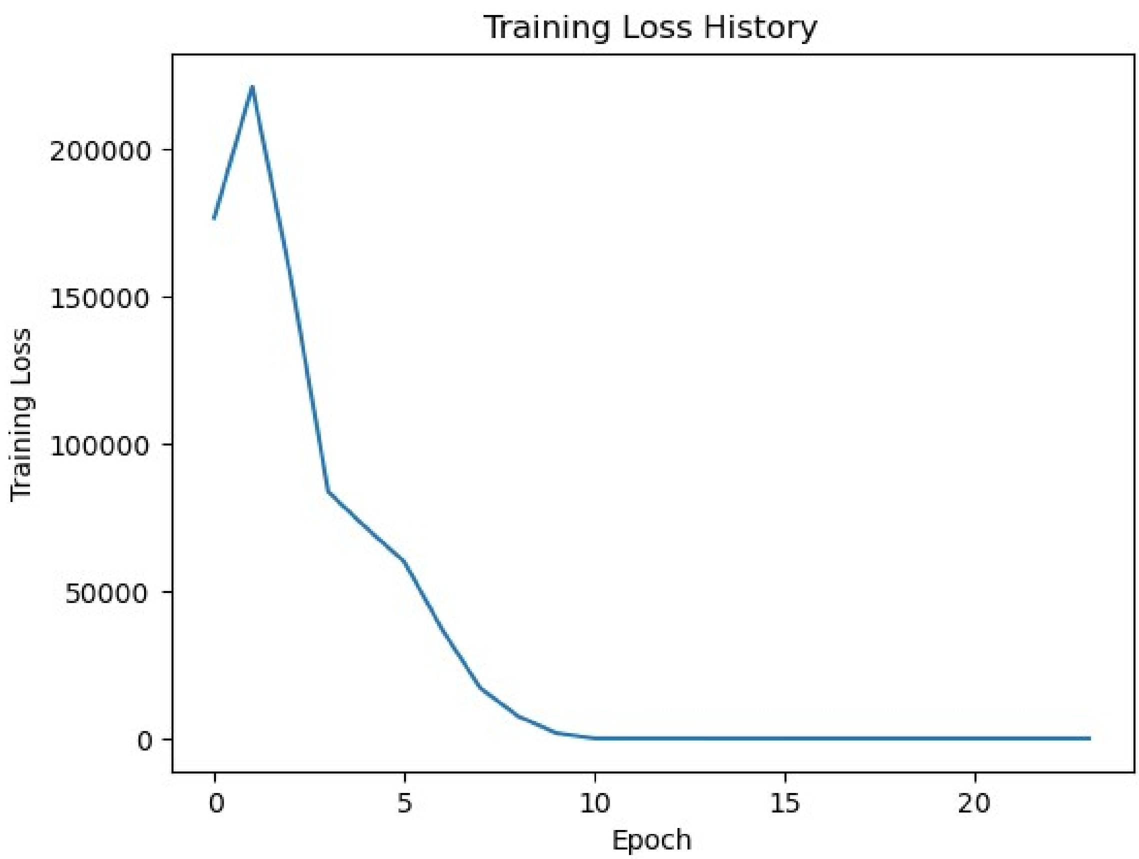
```python
import torch
import torch.nn as nn
import numpy as np
import matplotlib.pyplot as plt
from torchdiffeq import odeint
from scipy.integrate import solve_ivp

# Define the neural network for the ODE function
class ODEFunc(nn.Module):
    def __init__(self):
        super(ODEFunc, self).__init__()
        # Updated network architecture with more layers and neurons
        self.net = nn.Sequential(
            nn.Linear(2, 64),
            nn.Tanh(),
            nn.Linear(64, 64),
            nn.Tanh(),
            nn.Linear(64, 2)
        )
        self.net.apply(self.init_weights)

    def forward(self, t, y):
        return self.net(y)

    @staticmethod
    def init_weights(m):
        if isinstance(m, nn.Linear):
            nn.init.kaiming_normal_(m.weight)
            nn.init.constant_(m.bias, 0)

# Define the training function with dynamic learning rate adjustment
and gradient clipping
def train_neural_ode(func, x_data, t_data, epochs=1000, lr=5e-4,
loss_threshold=0.01):
    optimizer = torch.optim.Adam(func.parameters(), lr=lr,
weight_decay=1e-4)  # L2 regularization
    scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer,
'min', factor=0.5, patience=100, min_lr=1e-5)
    loss_fn = nn.MSELoss()
    loss_history = []

    for epoch in range(epochs):
        optimizer.zero_grad()
        y_pred = odeint(func, x_data[0], t_data, method='dopri5',
rtol=1e-4, atol=1e-4)
        loss = loss_fn(y_pred, x_data)
        loss.backward()

        # Apply gradient clipping
        torch.nn.utils.clip_grad_norm_(func.parameters(),
```

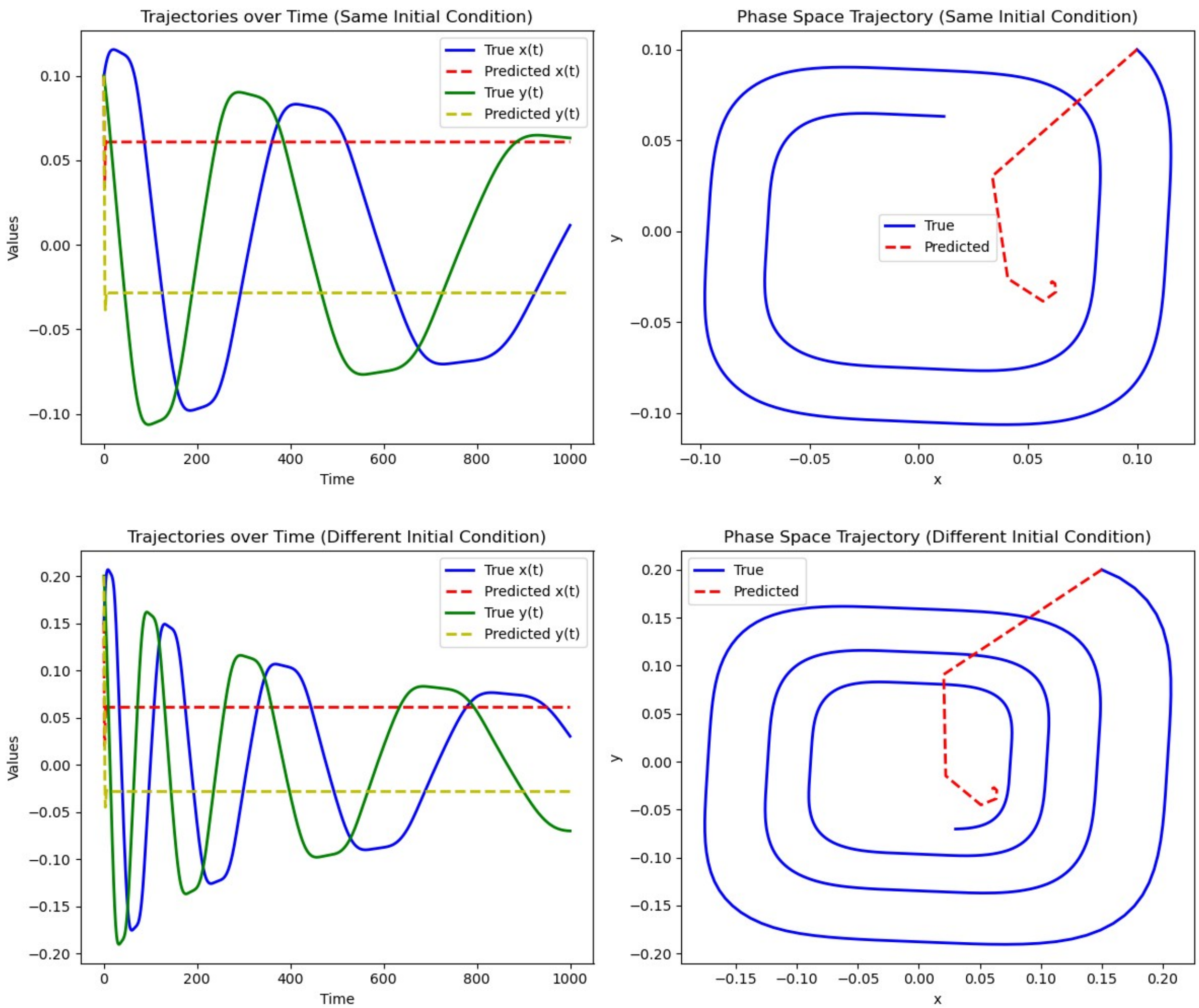
```
max_norm=1.0)

        optimizer.step()
        scheduler.step(loss)  # Adjust learning rate based on loss

        loss_history.append(loss.item())

        if loss.item() <= loss_threshold:
            print(f"Training converged at epoch {epoch+1} with loss
{loss.item()}")
            break

        # Print progress every 500 epochs
        if epoch % 10 == 0:
            print(f"Epoch {epoch+1}, Loss: {loss.item()}")

    return func, loss_history

# Generate data from the dynamical system
def dynamical_system(t, y):
    dxdt = -0.1 * y[0]**3 + 2 * y[1]**3
    dydt = -2 * y[0]**3 - 0.1 * y[1]**3
    return np.array([dxdt, dydt])

t_eval = np.linspace(0, 1000, 1000)
initial_state = np.array([0.1, 0.1])

# Use scipy to generate training data
sol = solve_ivp(dynamical_system, (0, 1000), initial_state,
t_eval=t_eval)
x_data = torch.tensor(sol.y.T, dtype=torch.float32)
t_data = torch.tensor(t_eval, dtype=torch.float32)

# Create an ODEFunc model
func = ODEFunc()

# Train the Neural ODE model
func, loss_history = train_neural_ode(func, x_data, t_data)

# Plot the training loss
plt.plot(loss_history)
plt.xlabel('Epoch')
plt.ylabel('Training Loss')
plt.title('Training Loss History')
plt.show()

# Define helper function for prediction
def predict(func, initial_state, t_eval):
    initial_state = torch.tensor(initial_state, dtype=torch.float32)
    t_data = torch.tensor(t_eval, dtype=torch.float32)
```

```
    with torch.no_grad():
        pred = odeint(func, initial_state, t_data, method='dopri5',
rtol=1e-6, atol=1e-6)
    return pred.numpy()

# Compare model prediction with true system
def plot_results(sol, pred, title):
    plt.figure(figsize=(12, 5))
    # Plot trajectories over time
    plt.subplot(1, 2, 1)
    plt.plot(t_eval, sol.y[0], 'b', label='True x(t)', linewidth=2)
    plt.plot(t_eval, pred[:, 0], 'r--', label='Predicted x(t)',
linewidth=2)
    plt.plot(t_eval, sol.y[1], 'g', label='True y(t)', linewidth=2)
    plt.plot(t_eval, pred[:, 1], 'y--', label='Predicted y(t)',
linewidth=2)
    plt.xlabel('Time')
    plt.ylabel('Values')
    plt.legend()
    plt.title(f'Trajectories over Time ({title})')

    # Plot phase space trajectory
    plt.subplot(1, 2, 2)
    plt.plot(sol.y[0], sol.y[1], 'b', label='True', linewidth=2)
    plt.plot(pred[:, 0], pred[:, 1], 'r--', label='Predicted',
linewidth=2)
    plt.xlabel('x')
    plt.ylabel('y')
    plt.legend()
    plt.title(f'Phase Space Trajectory ({title})')
    plt.tight_layout()
    plt.show()

# Predict with the same initial condition
pred_same_ic = predict(func, initial_state, t_eval)
plot_results(sol, pred_same_ic, title="Same Initial Condition")

# Predict with a different initial condition
new_initial_state = [0.15, 0.2]
sol_new_ic = solve_ivp(dynamical_system, (0, 1000), new_initial_state,
t_eval=t_eval)
pred_new_ic = predict(func, new_initial_state, t_eval)
plot_results(sol_new_ic, pred_new_ic, title="Different Initial
Condition")

# Part 3(b) Add noise and retrain
noise_level = 0.02  # Reduced noise level for stability
x_data_noisy = x_data + noise_level * torch.randn_like(x_data)

# Retrain the Neural ODE with noisy data
```

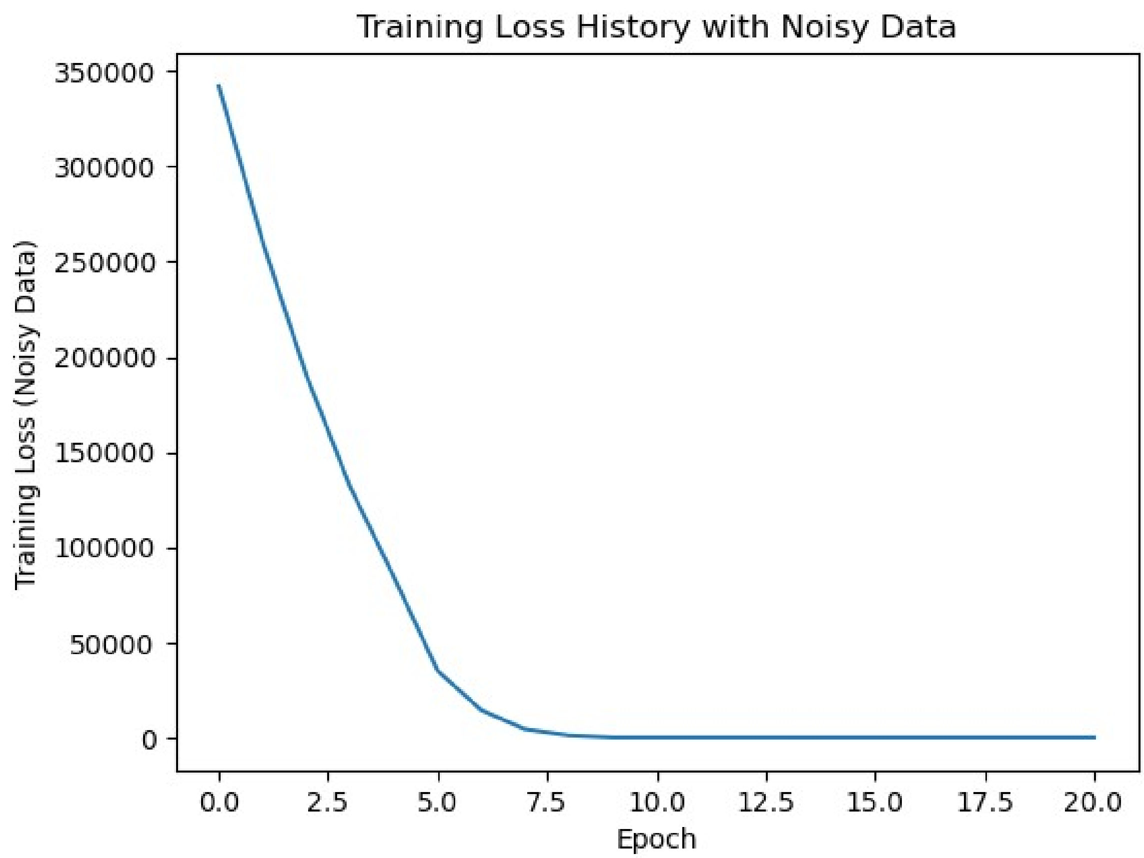
```
func_noisy = ODEFunc()
func_noisy, loss_history_noisy = train_neural_ode(func_noisy,
x_data_noisy, t_data)

# Plot noisy training loss
plt.plot(loss_history_noisy)
plt.xlabel('Epoch')
plt.ylabel('Training Loss (Noisy Data)')
plt.title('Training Loss History with Noisy Data')
plt.show()

# Predict with noisy model and a different initial condition
pred_new_ic_noisy = predict(func_noisy, new_initial_state, t_eval)
plot_results(sol_new_ic, pred_new_ic_noisy, title="Different Initial
Condition (Noisy Data)")
```

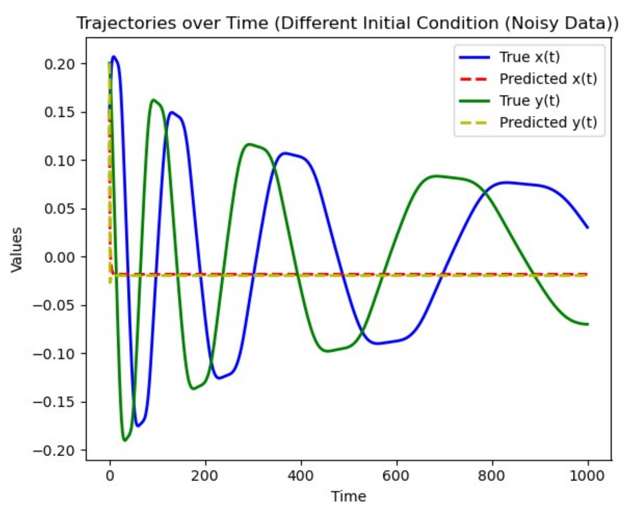
```
Epoch 1, Loss: 176523.203125
Epoch 11, Loss: 14.312626838684082
Epoch 21, Loss: 0.04516028240323067
Training converged at epoch 24 with loss 0.007955534383654594
```

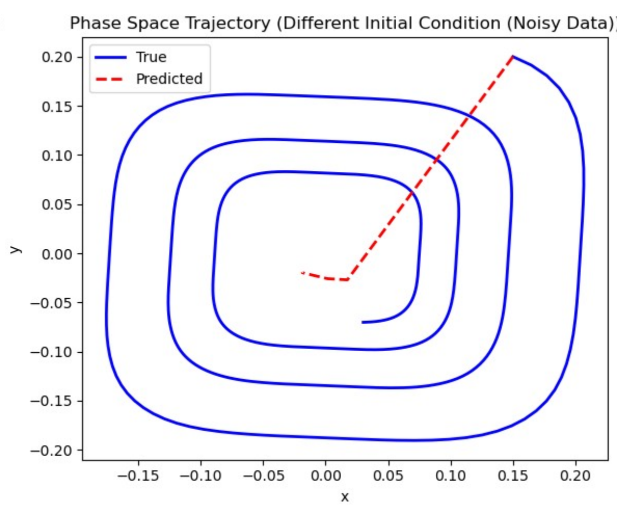





```
Epoch 1, Loss: 342125.9375
Epoch 11, Loss: 1.8877500295639038
Training converged at epoch 21 with loss 0.005802476312965155
```


Training Loss History with Noisy Data
Training Loss (Noisy Data)
Epoch
350000
300000
250000
200000
150000
100000
50000
0
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0


Trajectories over Time (Different Initial Condition (Noisy Data))
True x(t)
Predicted x(t)
True y(t)
Predicted y(t)
Values
Time


Phase Space Trajectory (Different Initial Condition (Noisy Data))
True
Predicted
y
x