Brunton and Kutz Problem 10.1: Model Predictive Control (MPC) to Control the Lorenze System Using DMDc, SINDYc, and NN Models

Source Filename: /main.py

Rico A. R. Picone

This problem is rather involved, so we will systemaically build up the machinery to solve it.

First, import the necessary libraries:

```
import numpy as np
import matplotlib.pyplot as plt
import control
import control.optimal as opt
import pydmd
import pysindy
```

Set up some flags for running the MPC simulations:

```
run_lorenz = True
run_DMDc = True
run_SINDYc = False
run_NN = False
```

Model Predictive Control

We will define an MPC simulation class that can handle nonlinear systems. The class will contain a (potentially nonlinear) system model `sys` that is either a plant or a closed-loop system. It will also require a predictor function that can predict future states of the plant over a time horizon given the current state and control input. It is best for it to be as general as possible, but we won't try to make it capable of handling all possible cases.

```
class MPCSimulation:
    """Model Predictive Control Simulation

    Simulate a system using model predictive control.
    Attributes:
        sys: System model (plant or closed-loop)
        inplist: List of input variables
        outlist: List of output variables
        predictor: Function that predicts future states given
            desired current state and control input
        T_horizon: Prediction horizon
        T_update: Update period
        n_updates: Number of updates
        n_horizon: Number of points in prediction horizon
        n_update: Number of points in update period
        xd: Desired state trajectory
        results: Simulation results
    """

    def __init__(self,
        sys,
        inplist,
        outlist,
        predictor,
        T_horizon,
        T_update,
        n_updates=10,
        n_horizon=31,
        n_update=10,
        xd=None
    ):
        self.sys = sys
        self.inplist = inplist
        self.outlist = outlist
        self.predictor = predictor
        self.T_horizon = T_horizon
        self.T_update = T_update
        self.n_updates = n_updates
        self.n_horizon = n_horizon
        self.n_update = n_update
        self.t_horizon = np.linspace(0, T_horizon, n_horizon)
        self.t_update = np.linspace(0, T_update, n_update+1)
        self.t_sim = np.linspace(0, T_update * n_updates, n_update * n_updates + 1)
        if xd is None:
            xd = np.zeros((sys.nstates, n_update * n_updates + 1))  # Regulate to zero
        self.xd = xd
        self.results = {
            "predictions": {
                "states": np.zeros(
                    (self.sys.nstates, self.n_horizon, self.n_updates)
```

```python
                ),
                "inputs": np.zeros(
                    (self.sys.ninputs, self.n_horizon, self.n_updates)
                )
            },
            "simulation": {
                "states": np.zeros(
                    (self.sys.nstates, self.n_update * self.n_updates + 1)
                ),
                "inputs": np.zeros(
                    (self.sys.ninputs, self.n_update * self.n_updates + 1)
                )
            }
        }  # Store results here

    def _predict(self, xd, t_horizon):
        return self.predictor(xd, t_horizon)

    def _simulate_update_period(self, xd, period):
        """Simulate over the update period

        Implement feedforward control.
        """
        xp, up = self._predict(xd, self.t_horizon)
        self.results["predictions"]["states"][:, :, period] = xp
        self.results["predictions"]["inputs"][:, :, period] = up
        xd = xp[:, :self.n_update+1]
        ud = up[:, :self.n_update+1]
        sim = control.input_output_response(
            self.sys, T=self.t_update, U=ud, X0=xd[:, 0]
        )
        return sim

    def simulate(self):
        for i in range(self.n_updates):
            print(f"Simulating update {i+1}/{self.n_updates}")
            j = i*self.n_update
            xd_period = self.xd[:, j:j+self.n_update+1]
            if i != 0:
                xd_period[:, 0] = \
                    self.results["simulation"]["states"][:, j]
                        # Start with last state from previous period
            sim = self._simulate_update_period(xd_period, i)

            self.results["simulation"]["states"][:, j:j+self.n_update+1] = sim.outputs
            self.results["simulation"]["inputs"][:, j:j+self.n_update+1] = sim.inputs
```

```
def plot_results(self):
    fig, ax = plt.subplots(2, 1, sharex=True)
    # Plot desired states:
    if self.xd is not None:
        for i in range(self.sys.nstates):
            ax[0].plot(
                self.t_sim,
                self.xd[i, :],
                'r-.', linewidth=1
            )
    # Plot states:
    ## Plot predicted states:
    for i in range(self.sys.nstates):
        for j in range(self.n_updates):
            ax[0].plot(
                j*self.T_update,
                self.results["predictions"]["states"][i, 0, j],
                'k.'
            )  # Initial state
            ax[0].plot(
                self.t_horizon + j*self.T_update,
                self.results["predictions"]["states"][i, :, j],
                'k--', linewidth=0.5
            )  # Predicted state
    ## Plot simulated states:
    for i in range(self.sys.nstates):
        ax[0].plot(
            self.t_sim,
            self.results["simulation"]["states"][i],
            label=f"State {i}"
        )
    ax[0].set_ylabel('State')
    ax[0].legend()
    # Plot inputs:
    ## Plot predicted inputs:
    for i in range(self.sys.ninputs):
        for j in range(self.n_updates):
            ax[1].plot(
                j*self.T_update,
                self.results["predictions"]["inputs"][i, 0, j],
                'k.'
            )
            ax[1].plot(
                self.t_horizon + j*self.T_update,
                self.results["predictions"]["inputs"][i, :, j],
                'k--', linewidth=0.5
            )
    ## Plot simulated inputs:
    for i in range(self.sys.ninputs):
        ax[1].plot(
            self.t_sim,
            self.results["simulation"]["inputs"][i],
            label=f"Input {i}"
        )
    ax[1].set_ylabel('Input')
    ax[1].set_xlabel('Time')
    ax[1].legend()
    plt.draw()
    return fig, ax
```

The `predictor()` function is quite general here. We will write three different versions, one for each of the DMDc, SINDYc, and NN Models.

The Lorenz System and Testing the MPC Simulation

We will use the Lorenz system as the plant model. Define the Lorenz system dynamics:

```
def lorenz_forced(t, x_, u, params={}):
    """
    Forced Lorenz equations dynamics (dx/dt, dy/dt, dz/dt)
    """
    sigma=10
    beta=8/3
    rho=28
    x, y, z = x_
    dx = sigma * (y - x) + u[0]
    dy = x * (rho - z) - y
    dz = x * y - beta * z
    return [dx, dy, dz]
```

Because we're using the `control` package, we can use the `input_output_response()` function to simulate the forced Lorenz system. Create a `NonlinearIOSystem` object for the forced Lorenz system:

```
lorenz_forced_sys = control.NonlinearIOSystem(
    lorenz_forced, None, inputs=["u"], states=["x", "y", "z"],
    name="lorenz_forced_sys"
)
```

We would like to predict a trajectory, state and input, over a time horizon. For all three models (and the exact model used here), this will involve predicting the future states given the desired state trajectory. The challenge is that we don't know the future inputs. There are multiple ways to approach this. The first is to assume that the future inputs are the same as the current input. This works for short update periods relative to the dynamics of the system. The second approach is to solve an optimal control problem to determine the future inputs. This will give better results but is more computationally expensive. We will write a function that can handle both cases.

```
def predict_trajectory(sys, x0, t_horizon, u0=None, cost=None, terminal_cost=None):
    if cost is None:  # Constant input
        if u0 is None:
            u0 = np.zeros_like(t_horizon)  # Zero input
        u = u0 * np.ones_like(t_horizon)  # Constant input
        x = control.input_output_response(
            sys, T=t_horizon, U=u, x0=x0
        ).states
    else:  # Solve optimal control problem
        ocp = opt.OptimalControlProblem(
            sys, t_horizon, cost, terminal_cost=terminal_cost
        )
        res = ocp.compute_trajectory(x0, print_summary=False)
        u = res.inputs
        x = res.states
    return x, u
```

This isn't specific enough to be used as a predictor function, but it can be used to write a predictor function. We will write a predictor function that uses the Lorenz system model to predict future states. We will use the optimal control approach to determine the future inputs.

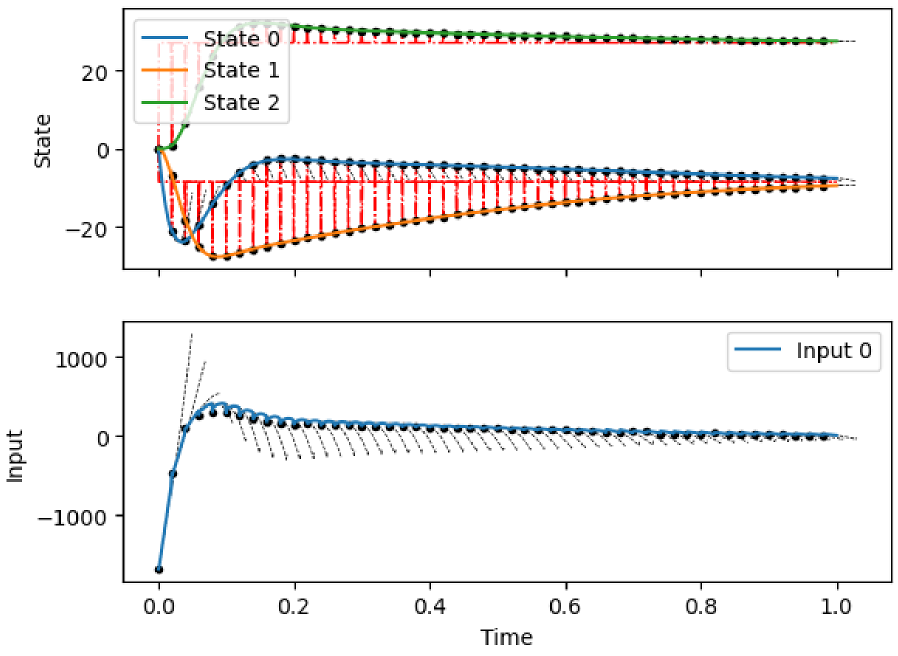
```
def lorenz_predictor(xd, t_horizon, sys):
    """Predictor for Lorenz system using optimal control"""
    Q = np.eye(sys.nstates)
    R = 0.01 * np.eye(sys.ninputs)
    cost = control.optimal.quadratic_cost(sys, Q, R, x0=xd[:, -1])
    terminal_cost = control.optimal.quadratic_cost(
        sys, 5*Q, 0*R, x0=xd[:, -1]
    )  # Penalize terminal state more
    x, u = predict_trajectory(
        sys, xd[:, 0], t_horizon, cost=cost, terminal_cost=terminal_cost
    )
    return x, u
```

We can now test the MPC simulation class using the Lorenz system model to predict future inputs. We expect the results to be good because we're using the exact model.

```
dt_lorenz = 1e-3  # Time step
T_horizon = dt_lorenz * 50
```

```
T_update = dt_lorenz * 20
n_horizon = int(np.floor(T_horizon/dt_lorenz)) + 1
n_update = int(np.floor(T_update/dt_lorenz)) + 1
n_updates = 50
xeq = np.array([-np.sqrt(72), -np.sqrt(72), 27])
print(f"xeq: {xeq}")
command = np.outer(xeq, np.ones(n_update * n_updates + 1))
command[:, 0] = np.array([0, 0, 0])  # Initial state
if run_lorenz:
    mpc_lorenz = MPCSimulation(
        sys=lorenz_forced_sys,
        inplist=['u'],
        outlist=['lorenz_forced_sys.x', 'lorenz_forced_sys.y', 'lorenz_forced_sys.z'],
        predictor=lambda x0, t_horizon: lorenz_predictor(x0, t_horizon, sys=lorenz_forced_sy
        T_horizon=T_horizon,
        T_update=T_update,
        n_updates=n_updates,
        n_horizon=n_horizon,
        n_update=n_update,
        xd=command
    )
    results_mpc_lorenz = mpc_lorenz.simulate()
    mpc_lorenz.plot_results()
    plt.draw()
```

```
xeq: [-8.48528137 -8.48528137 27.        ]
Simulating update 1/50

Simulating update 2/50

Simulating update 3/50

Simulating update 4/50

Simulating update 5/50

Simulating update 6/50

Simulating update 7/50

Simulating update 8/50

Simulating update 9/50

Simulating update 10/50

Simulating update 11/50

Simulating update 12/50

Simulating update 13/50

Simulating update 14/50
Simulating update 15/50
Simulating update 16/50
Simulating update 17/50
Simulating update 18/50
Simulating update 19/50
Simulating update 20/50
Simulating update 21/50
Simulating update 22/50
Simulating update 23/50
Simulating update 24/50
Simulating update 25/50
Simulating update 26/50
Simulating update 27/50
Simulating update 28/50
Simulating update 29/50
Simulating update 30/50
Simulating update 31/50
Simulating update 32/50
Simulating update 33/50
Simulating update 34/50
Simulating update 35/50
Simulating update 36/50
Simulating update 37/50
Simulating update 38/50
Simulating update 39/50
Simulating update 40/50
Simulating update 41/50
Simulating update 42/50
Simulating update 43/50
Simulating update 44/50
Simulating update 45/50
Simulating update 46/50

Simulating update 47/50

Simulating update 48/50

Simulating update 49/50

Simulating update 50/50
```



The results are good. Note that if a global optimal control problem is solved and used as the `xd` commanded trajectory, the results will be better. Here we have used a constant commanded trajectory, which is not a feasible trajectory for the Lorenz system. The deviations of the predictor from the actual response are due to the Lorenz system being chaotic, numerical optimization limitations, and the finite time horizon over which the optimal control problem is solved.

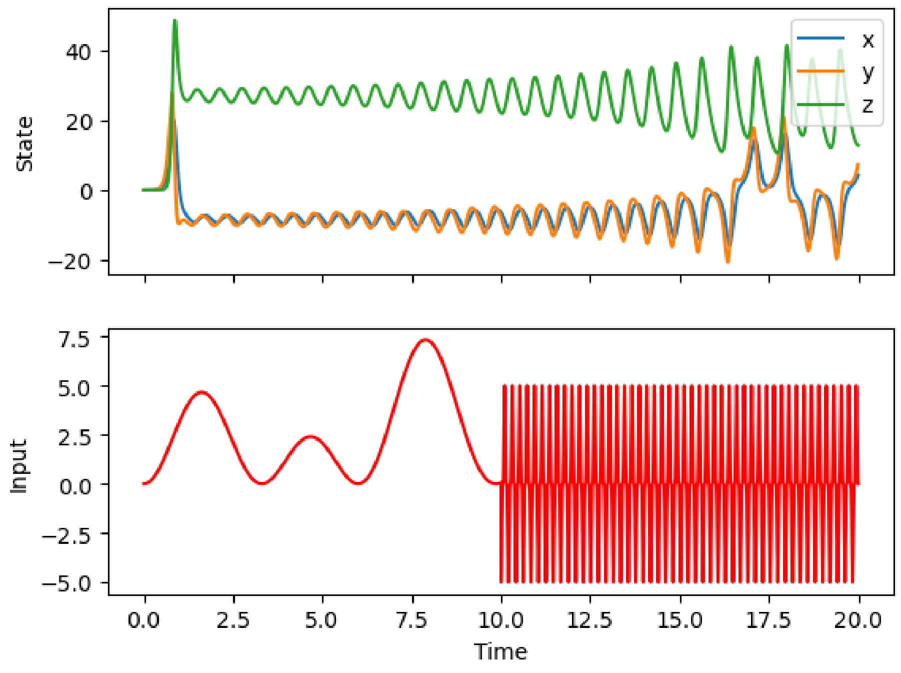
Generating Training and Testing Data

Now we generate some training and testing data for the predictors. Generate the training and testing data as follows:

```python
dt_data = 1e-3  # Time step
t_data = np.arange(0, 20, dt_data)  # Time array
n_data = len(t_data)
n_train = int(n_data/2)
u_data_train = (2*np.sin(t_data[:n_train])
                + np.sin(0.1*t_data[:n_train]))**2  # Input
u_data_test = (5*np.sin(30*t_data[n_train:])**3)
u_data = np.hstack((u_data_train, u_data_test))
x_data = control.input_output_response(
    lorenz_forced_sys, T=t_data, U=u_data
).states
```

Plot the data over time:

```
fig, ax = plt.subplots(2, 1, sharex=True)
ax[0].plot(t_data, x_data[0], label='x')
ax[0].plot(t_data, x_data[1], label='y')
ax[0].plot(t_data, x_data[2], label='z')
ax[0].set_ylabel('State')
ax[0].legend()
ax[1].plot(t_data, u_data, label='u', color='r')
ax[1].set_ylabel('Input')
ax[1].set_xlabel('Time')
plt.draw()
```



Partition the data into training and testing sets:

```
n_train = int(n_data/2)
n_test = n_data - n_train
t_train = t_data[:n_train]
```

```
t_test = t_data[n_train:]
u_train = u_data[:n_train]
u_test = u_data[n_train:]
x_train = x_data[:, :n_train]
x_test = x_data[:, n_train:]
```

Dynamic Mode Decomposition with Control (DMDc) Model

We could define the exact DMDc function from Brunton and Kutz (2022) section 7.2 and modified based on section 10.2. However, it is more convenient to use the PyDMD package to compute the DMDc model.

Compute the DMDc model:

```
dmdc = pydmd.DMDc()
dmdc.fit(X=x_train, I=u_train[:-1])
Phi_pydmd = dmdc.modes
Lambda_pydmd = np.diag(dmdc.eigs)
b_pydmd = dmdc.amplitudes
A_pydmd = Phi_pydmd @ Lambda_pydmd @ np.linalg.pinv(Phi_pydmd)
B_pydmd = dmdc.B
print(f"A_pydmd:\n{A_pydmd}")
print(f"B_pydmd:\n{B_pydmd}")
```

```
A_pydmd:
[[ 0.06789146  0.07258139 -0.24086006]
 [ 0.07258139  0.0775953  -0.25749863]
 [-0.24086006 -0.25749863  0.85450466]]
B_pydmd:
[[-0.00013028]
 [-0.00013924]
 [ 0.00046259]]
```

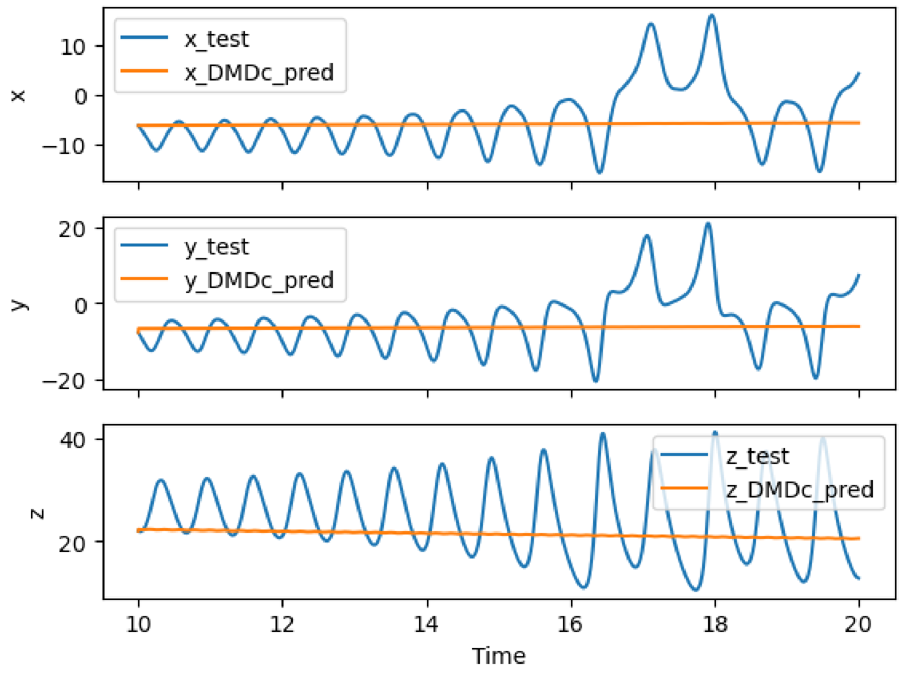
Predict the trajectory on the test data:

```
dt = t_train[1] - t_train[0]
x0 = x_test[:, 0]
sys_DMDc = control.ss(A_pydmd, B_pydmd, np.eye(A_pydmd.shape[0]), 0, dt=dt)
x_DMDc_pred = control.forced_response(sys_DMDc, T=t_test, U=u_test, X0=x0).states
```

Plot the predicted trajectory with the test data:

```
fig, ax = plt.subplots(3, 1, sharex=True)
ax[0].plot(t_test, x_test[0], label='x_test')
ax[0].plot(t_test, x_DMDc_pred[0], label='x_DMDc_pred')
ax[0].set_ylabel('x')
ax[0].legend()
ax[1].plot(t_test, x_test[1], label='y_test')
```

```
ax[1].plot(t_test, x_DMDc_pred[1], label='y_DMDc_pred')
ax[1].set_ylabel('y')
ax[1].legend()
ax[2].plot(t_test, x_test[2], label='z_test')
ax[2].plot(t_test, x_DMDc_pred[2], label='z_DMDc_pred')
ax[2].set_ylabel('z')
ax[2].set_xlabel('Time')
ax[2].legend()
plt.draw()
```



The results are so bad because the DMDc model is linear and the Lorenz system is highly nonlinear. With an MPC controller, the prediction doesn't have to be good for long, but these predictions deviate almost immediately, so we don't have high hopes for the MPC controller with DMDc.

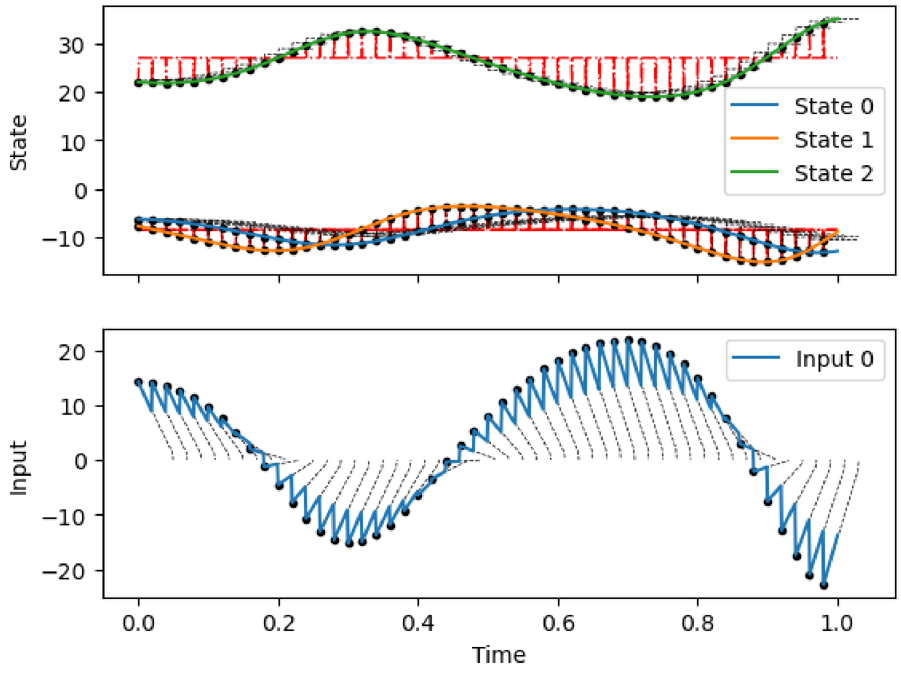
Nonetheless, define the DMDc predictor function:

```
def DMDc_predictor(xd, t_horizon, sys=None):
    """Predictor for DMDc model using optimal control"""
    Q = np.eye(sys.nstates)
    R = 0.01 * np.eye(sys.ninputs)
    cost = control.optimal.quadratic_cost(sys, Q, R, x0=xd[:, -1])
    terminal_cost = control.optimal.quadratic_cost(sys, 5*Q, 0*R, x0=xd[:, -1])
    x, u = predict_trajectory(
        sys, xd[:, 0], t_horizon, cost=cost, terminal_cost=terminal_cost
    )
    return x, u
```

Now that we have our DMCc predictor, we can try it in the MPC simulation. We could use feedback control as well, which would improve the results, but using feedforward only allows us to get a better comparison among the predictors. Again, we don't expect good results, but we can at least see how it performs.

```
T_horizon = dt * 50
T_update = dt * 20
n_horizon = int(np.floor(T_horizon/dt)) + 1  # Must match DMDc model timebase
n_update = int(np.floor(T_update/dt)) + 1
n_updates = 50
xeq = np.array([-np.sqrt(72), -np.sqrt(72), 27])
print(f"xeq: {xeq}")
command = np.outer(xeq, np.ones(n_update * n_updates + 1))
command[:, 0] = x_test[:, 0]  # Initial state
if run_DMDc:
    mpc_DMDc = MPCSimulation(
        sys=lorenz_forced_sys,
        inplist=['u'],
        outlist=['lorenz_forced_sys.x', 'lorenz_forced_sys.y', 'lorenz_forced_sys.z'],
        predictor=lambda x0, t_horizon: DMDc_predictor(x0, t_horizon, sys=sys_DMDc),
        T_horizon=T_horizon,
        T_update=T_update,
        n_updates=n_updates,
        n_horizon=n_horizon,
        n_update=n_update,
        xd=command
    )
    results_mpc_DMDc = mpc_DMDc.simulate()
    mpc_DMDc.plot_results()
    plt.show()
```

```
xeq: [-8.48528137 -8.48528137 27.        ]
Simulating update 1/50

Simulating update 2/50

Simulating update 3/50

Simulating update 4/50

Simulating update 5/50

Simulating update 6/50

Simulating update 7/50

Simulating update 8/50
Simulating update 9/50
Simulating update 10/50
Simulating update 11/50
Simulating update 12/50
Simulating update 13/50
Simulating update 14/50
Simulating update 15/50
Simulating update 16/50
Simulating update 17/50
Simulating update 18/50
Simulating update 19/50
Simulating update 20/50
Simulating update 21/50
Simulating update 22/50
Simulating update 23/50
Simulating update 24/50
Simulating update 25/50
Simulating update 26/50
Simulating update 27/50
Simulating update 28/50
Simulating update 29/50
Simulating update 30/50
Simulating update 31/50
Simulating update 32/50
Simulating update 33/50
Simulating update 34/50
Simulating update 35/50
Simulating update 36/50
Simulating update 37/50
Simulating update 38/50
Simulating update 39/50
Simulating update 40/50
Simulating update 41/50
Simulating update 42/50
Simulating update 43/50
Simulating update 44/50
Simulating update 45/50
Simulating update 46/50
Simulating update 47/50
Simulating update 48/50
Simulating update 49/50
Simulating update 50/50
```



As expected, the DMDc model performs poorly. An alternative DMDc approach for highly nonlinear systems is to use extended DMDc (EDMDc) with nonlinear measurements. This is connected to the Koopman operator theory. We will not pursue this here.

Sparse Identification of Nonlinear Dynamics (SINDy)

Define the SINDy model:

```python
sindy_model = pysindy.SINDy(feature_names=["x", "y", "z"])
sindy_model.fit(x_train.T, t=dt, multiple_trajectories=False)
print("Dynamics identified by pySINDy:")
sindy_model.print()
```

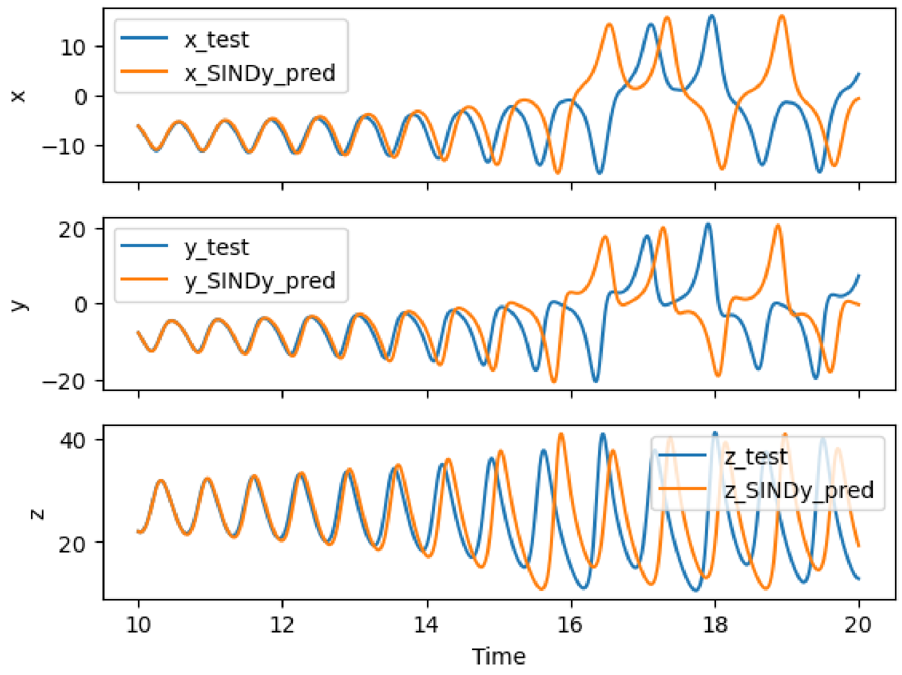
```
Dynamics identified by pySINDy:
(x)' = 2.126 1 + -9.917 x + 9.875 y
(y)' = 27.866 x + -1.008 y + -0.994 x z
(z)' = -2.662 z + 0.998 x y
```

The SINDy model `sindy_model` has the `simulate()` method that can be used to predict the evolution of the system. First, let's predict the trajectory on the test data:

```python
x_SINDy_pred = sindy_model.simulate(x_test[:,0], t_test)
```

Plot the simulated and predicted trajectory with the test data:

```python
fig, ax = plt.subplots(3, 1, sharex=True)
ax[0].plot(t_test, x_test[0], label='x_test')
ax[0].plot(t_test, x_SINDy_pred[:, 0], label='x_SINDy_pred')
ax[0].set_ylabel('x')
ax[0].legend()
ax[1].plot(t_test, x_test[1], label='y_test')
ax[1].plot(t_test, x_SINDy_pred[:, 1], label='y_SINDy_pred')
ax[1].set_ylabel('y')
ax[1].legend()
ax[2].plot(t_test, x_test[2], label='z_test')
ax[2].plot(t_test, x_SINDy_pred[:, 2], label='z_SINDy_pred')
ax[2].set_ylabel('z')
ax[2].set_xlabel('Time')
ax[2].legend()
plt.draw()
```



So the SINDy model is not perfect, but it is much better than the DMDc model. For a few seconds, the model is quite good. With more training data, the model would likely improve.

We can use the model to write a predictor function for the SINDy model. The optimal control approach requires we create a `control.NonlinearIOSystem` object from the SINDy model.

```
print(sindy_model.coefficients())
print(sindy_model.get_feature_names())
```

```
[[ 2.12617393 -9.91728749  9.87540105  0.          0.          0.
   0.          0.          0.          0.        ]
 [ 0.         27.86624432 -1.00768669  0.          0.          0.
  -0.99420703  0.          0.          0.        ]
 [ 0.          0.          0.         -2.66221585  0.          0.99787211
   0.          0.          0.          0.        ]]
['1', 'x', 'y', 'z', 'x^2', 'x y', 'x z', 'y^2', 'y z', 'z^2']
```

Define the SINDy predictor function:

```
def SINDy_predictor(xd, t_horizon, sys=None):
    """Predictor for SINDy model using optimal control"""
    Q = np.eye(sys.nstates)
    R = 0.01 * np.eye(sys.ninputs)
    cost = control.optimal.quadratic_cost(sys, Q, R, x0=xd[:, -1])
    terminal_cost = control.optimal.quadratic_cost(sys, 5*Q, 0*R, x0=xd[:, -1])
    x, u = predict_trajectory(
        sys, xd[:, 0], t_horizon, cost=cost, terminal_cost=terminal_cost
    )
    return x, u

plt.show()
```