Murray Problem 4.6: Model Predictive Control (i.e., Receding Horizon Control) of a Thrust-Vectoring Aircraft

Source Filename: /main.py

Rico A. R. Picone

In this problem, we will design a model predictive controller (MPC) for the planar vertical takeoff and landing (PVTOL) system. The MPC will use feed-forward predictions of future states to optimize the input over a finite time horizon.

Begin by importing the necessary libraries and modules as follows:

```
import numpy as np
import matplotlib.pyplot as plt
import control
import control.optimal as opt
import time
from pvtol import pvtol, pvtol_windy
```

The PVTOL system dynamics and utility functions are defined in the `pvtol` module, found here. We have loaded the dynamics as `pvtol`. Here is some basic information about the system:

```
print(f"Number of states: {pvtol.nstates}")
print(f"Number of inputs: {pvtol.ninputs}")
print(f"Number of outputs: {pvtol.noutputs}")
```

```
Number of states: 6
Number of inputs: 2
Number of outputs: 6
```

From `pvtol.py`, the state vector x and input vector u are defined as follows:

$$x = \begin{bmatrix} x & y & \theta & \dot{x} & \dot{y} & \dot{\theta} \end{bmatrix}^\top \quad u = \begin{bmatrix} F_1 & F_2 \end{bmatrix}^\top$$

Our first task is to determine the equilibrium state and input that corresponds to a hover at the origin:

```python
xeq, ueq = control.find_eqpt(
    pvtol,  # System dynamics
    x0=np.zeros((pvtol.nstates)),  # Initial guess for equilibrium state
    u0=np.zeros((pvtol.ninputs)),  # Initial guess for equilibrium input
    y0=np.zeros((pvtol.noutputs)),  # Initial guess for equilibrium output
    iy=[0, 1],  # Indices of states that should be zero at equilibrium
)
print(f'Equilibrium state: {xeq}')
print(f'Equilibrium input: {ueq}')
```

```
Equilibrium state: [0. 0. 0. 0. 0. 0.]
Equilibrium input: [ 0.  39.2]
```

Let's define a function to simulate an MPC controller for the PVTOL system, including future considerations of the disturbance input and feedback control:

```python
def simulate_mpc(
    sys,  # System dynamics
    ocp,  # Optimal control problem object
    t_end,  # Simulation end time
    dt,  # Update time step ("time period")
    x0,  # Initial state
    u0,  # Initial input
    disturbance_fun=None,  # Function to compute disturbance
    feedback=False,  # Feedback control included in sys
):
    """Simulate MPC controller for a system"""
    # Extract time parameters
    dt_ocp = ocp.timepts[1] - ocp.timepts[0]  # Time step for OCP
    nt_ocp = len(ocp.timepts)  # Number of OCP time steps
    nt = int(np.round(dt / dt_ocp))  # OCP time steps per update
    # Initialize storage
    t = np.arange(0, t_end, dt)  # Update times
    t_sim = np.linspace(0, t_end, int(nt*t_end/dt + 1))  # Simulation times
    x = np.zeros((sys.nstates, len(t)*nt+1))  # Simulated states
    u = np.zeros((sys.ninputs, len(t)*nt+1))  # Simulated inputs
    x_pred = np.zeros((sys.nstates, len(t), nt_ocp))  # Predicted states
    ninputs = sys.ninputs
    npredinputs = ninputs
    if disturbance_fun is not None:
        npredinputs += -1
    if feedback:
        npredinputs += -6
    u_pred = np.zeros((npredinputs, len(t), nt_ocp))
    # Loop over update times
    for i in range(0, len(t)):
        j = i*nt  # Index for update timesteps
```

```
        if i != 0:
            x0 = x[:, j]  # Update initial state
            u0 = u[:, j]  # Update initial input
        # Compute trajectory
        comp_traj_time = time.time()
        traj = ocp.compute_trajectory(x0, print_summary=False)
        comp_traj_time = time.time() - comp_traj_time
        print(f"Simulation time: {t[i]:.2f} s, "
              f"Trajectory computation time: {comp_traj_time:.2f} s")
        # Extract predicted state and inputs
        x_pred[:, i, :] = traj.states
        u_pred[:, i, :] = traj.inputs[:2]
        # Simulate the system over the update time/trajectory
        inputs = traj.inputs
        if feedback:  # Closed-loop command [xd, ud]
            inputs = np.vstack(
                [traj.states, inputs]
            )
        if disturbance_fun is not None:
            inputs = np.vstack(
                [inputs, disturbance_fun(t[i] + traj.time)]
            )
        sim = control.input_output_response(
            sys, traj.time[:nt+1], inputs[:, :nt+1], X0=x0
        )
        x[:, j:j+nt+1] = sim.states  # Update state
        u[:, j:j+nt+1] = sim.inputs  # Update input
    t_pred = traj.time
    return t, t_pred, t_sim, x, u, x_pred, u_pred
```

Feedforward MPC sans Wind Disturbance

Set up the optimization problem parameters:

```
x0 = xeq + np.array([0, 5, 0, 0, 0, 0])  # Initial state
xf = xeq + np.array([10, 5, 0, 0, 0, 0])  # Final state
Q = np.diag([1, 1, 10, 0, 0, 0])  # State cost matrix
R = np.diag([10, 1])  # Input cost matrix
cost = opt.quadratic_cost(pvtol, Q=Q, R=R, x0=xf, u0=ueq)  # J
Q_term = 5*Q  # Terminal cost matrix (improved results)
```

Set up the time horizon and time steps for the trajectory. The longer the time horizon, the more optimal the solution, but the longer the computation time. As we will see, the computation time will still be too long for real-time applications. In a real-time application, we would attempt to optimize the computation time. We will also set up the optimal control problem (OCP) object, including trajectory constraints and a terminal cost to improve the results:

```
T = 3  # Time horizon
nt = 31  # Number of time steps for OCP
t_ocp = np.linspace(0, T, nt)  # Time steps for OCP
A_con = np.array([[-1, -0.1], [1, -0.1], [0, -1], [0, 1]])
b_con = [0, 0, 0, 1.5 * pvtol.params['m'] * pvtol.params['g']]
traj_constraints = opt.input_poly_constraint(
    pvtol, A_con, b_con
)  # A [F1, F2] <= b - Results improved with these constraints
ocp = opt.OptimalControlProblem(
    pvtol, t_ocp, cost,
    trajectory_constraints=traj_constraints,
    terminal_cost=opt.quadratic_cost(pvtol, Q_term, None, x0=xf, u0=ueq),
)  # OCP object
```

Simulate the MPC controller for the PVTOL system:

```
t_end = 10  # Simulation end time
dt = 1  # Update time step
t, t_pred, t_sim, x, u, x_pred, u_pred = simulate_mpc(
    pvtol, ocp=ocp, t_end=t_end, dt=dt, x0=x0, u0=ueq
)
```

```
Simulation time: 0.00 s, Trajectory computation time: 3.67 s

Simulation time: 1.00 s, Trajectory computation time: 4.16 s

Simulation time: 2.00 s, Trajectory computation time: 3.39 s

Simulation time: 3.00 s, Trajectory computation time: 3.84 s

Simulation time: 4.00 s, Trajectory computation time: 4.01 s

Simulation time: 5.00 s, Trajectory computation time: 3.86 s

Simulation time: 6.00 s, Trajectory computation time: 3.69 s

Simulation time: 7.00 s, Trajectory computation time: 3.85 s

Simulation time: 8.00 s, Trajectory computation time: 3.51 s

Simulation time: 9.00 s, Trajectory computation time: 3.67 s
```

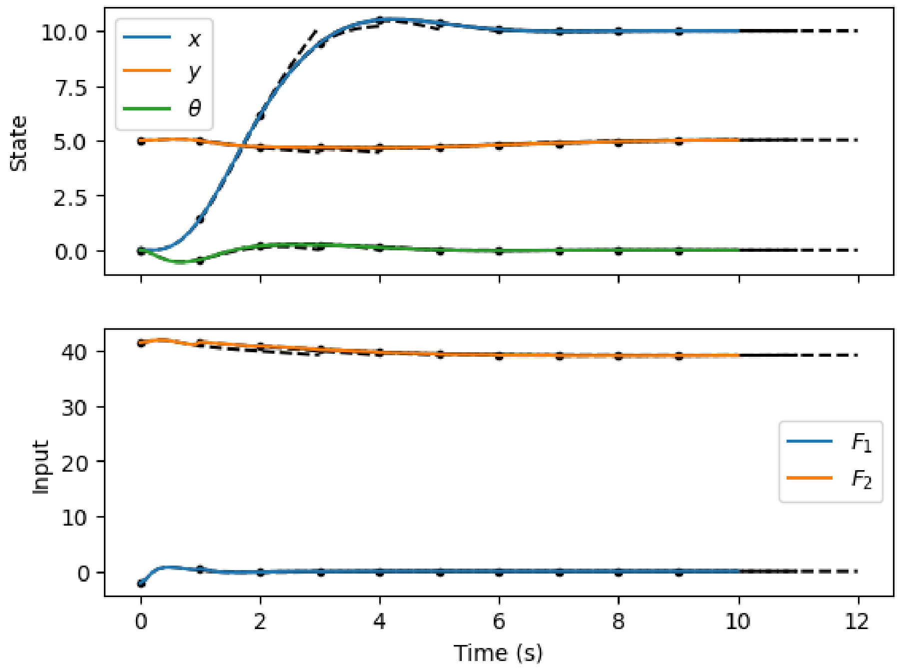
Plot the predicted and simulated states and inputs. First, define a function to plot the results:

```
def plot_results(t, t_pred, t_sim, x, u, x_pred, u_pred, feedback=False):
    if feedback:
        u = u[6:8]
    fig, ax = plt.subplots(2, 1, sharex=True)
    for i in range(0, len(t)):  # Plot predicted states
        ax[0].plot(t[i], x_pred[0, i, 0], 'k.')
```

```
        ax[0].plot(t[i], x_pred[1, i, 0], 'k.')
        ax[0].plot(t[i], x_pred[2, i, 0], 'k.')
        ax[0].plot(t_pred + t[i], x_pred[0, i, :], 'k--')
        ax[0].plot(t_pred + t[i], x_pred[1, i, :], 'k--')
        ax[0].plot(t_pred + t[i], x_pred[2, i, :], 'k--')
    ax[0].plot(t_sim, x[0], label='$x$')
    ax[0].plot(t_sim, x[1], label='$y$')
    ax[0].plot(t_sim, x[2], label='$\\theta$')
    ax[0].legend()
    ax[0].set_ylabel('State')
    for i in range(0, len(t)):  # Plot predicted inputs
        ax[1].plot(t[i], u_pred[0, i, 0], 'k.')
        ax[1].plot(t[i], u_pred[1, i, 0], 'k.')
        ax[1].plot(t_pred + t[i], u_pred[0, i, :], 'k--')
        ax[1].plot(t_pred + t[i], u_pred[1, i, :], 'k--')
    ax[1].plot(t_sim, u[0], label='$F_1$')
    ax[1].plot(t_sim, u[1], label='$F_2$')
    ax[1].legend()
    ax[1].set_xlabel('Time (s)')
    ax[1].set_ylabel('Input')
    plt.draw()
    return fig, ax
```

Plot the results:

```
fig, ax = plot_results(t, t_pred, t_sim, x, u, x_pred, u_pred)
```



The results show that the MPC performs well. Note that despite the predicted trajectory over the horizon does not match the simulated results perfectly. However, the frequent updates of the predicted trajectory allow the feedforward controller to adjust the input to track the desired trajectory. In a sense, the ”feedforward” controller is a form of feedback control because it is updated frequently by the measured state.

Feedforward MPC with Wind Disturbance

We can use the same MPC controller as before, but now we will add a wind disturbance to the system. The `pvtol` module provides the `pvtol_windy` object, which is the PVTOL system with a wind disturbance. This enters the system as a third input, d. We have planned for this in the `simulate_mpc` function by providing a `disturbance_fun` argument. Therefore, we can simulate the MPC controller with the wind disturbance as follows:

```
t, t_pred, t_sim, x, u, x_pred, u_pred = simulate_mpc(
    pvtol_windy, ocp=ocp, t_end=t_end, dt=dt, x0=x0, u0=ueq,
    disturbance_fun=np.sin
)
```

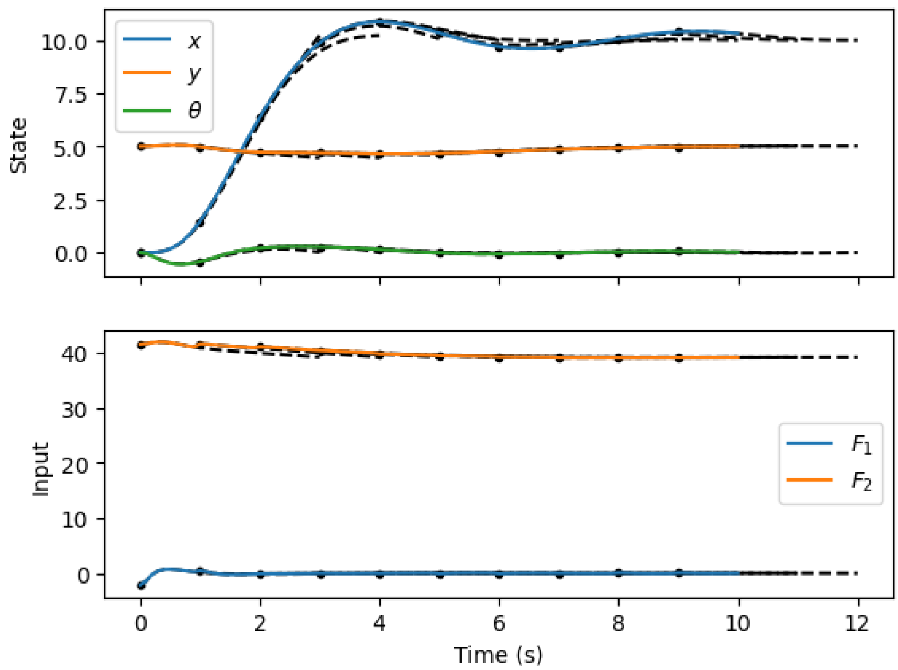
```
Simulation time: 0.00 s, Trajectory computation time: 3.71 s
```

```
Simulation time: 1.00 s, Trajectory computation time: 4.19 s
```

```
Simulation time: 2.00 s, Trajectory computation time: 4.02 s

Simulation time: 3.00 s, Trajectory computation time: 3.53 s

Simulation time: 4.00 s, Trajectory computation time: 3.68 s

Simulation time: 5.00 s, Trajectory computation time: 3.85 s

Simulation time: 6.00 s, Trajectory computation time: 3.87 s

Simulation time: 7.00 s, Trajectory computation time: 4.03 s

Simulation time: 8.00 s, Trajectory computation time: 3.87 s

Simulation time: 9.00 s, Trajectory computation time: 3.92 s
```

Plot the predicted and simulated states and inputs:

```
fig, ax = plot_results(t, t_pred, t_sim, x, u, x_pred, u_pred)
```



The results show that the MPC controller is not as effective in the presence of the wind disturbance. The predictions are not as accurate, and the controller struggles to track the desired trajectory. However, the tracking is not as bad as we might expect, due to the frequent updates.

Feedforward and Feedback MPC with Wind Disturbance

We can improve the performance of the MPC controller by adding a feedback term to the input. This feedback term can be computed by solving a linear quadratic regulator (LQR) problem as follows:

```
Q = np.diag([10, 100, 50, 0, 0, 0])  # State cost matrix
R = np.diag([1, 1])  # Input cost matrix
K, _, _ = control.lqr(pvtol.linearize(xeq, ueq), Q, R)  # Feedback gain
```

Define an output function for the feedback controller. The inputs to this function are the desired state, desired input, and current state. Consider the following function:

```
def lqr_output(t, x, u, params):
    xd, ud, x = u[0:6], u[6:8], u[8:14]  # Extract inputs
    return ud - K @ (x - xd)  # Feedforward/feedback control law
```

Define the LQR controller as an input-output system:

```
labels_ctrl = [f"xd[{i}]" for i in range(pvtol.nstates)] \
    + [f"ud[{i}]" for i in range(pvtol.ninputs)]
lqr_ctrl = control.NonlinearIOSystem(
    None,  # System dynamics
    lqr_output,  # Output function
    inputs=labels_ctrl + pvtol.state_labels,  # Inputs
    outputs=["F1", "F2"],  # Outputs
)
```

Define the closed-loop system with the LQR controller:

```
pvtol_windy_ctrl = control.interconnect(
    [pvtol_windy, lqr_ctrl],  # Systems
    inplist=labels_ctrl+["d"],  # Inputs
    outlist=pvtol.output_labels,  # Outputs
)
```

Now we can simulate the MPC controller, including the feedback term, with the wind disturbance:

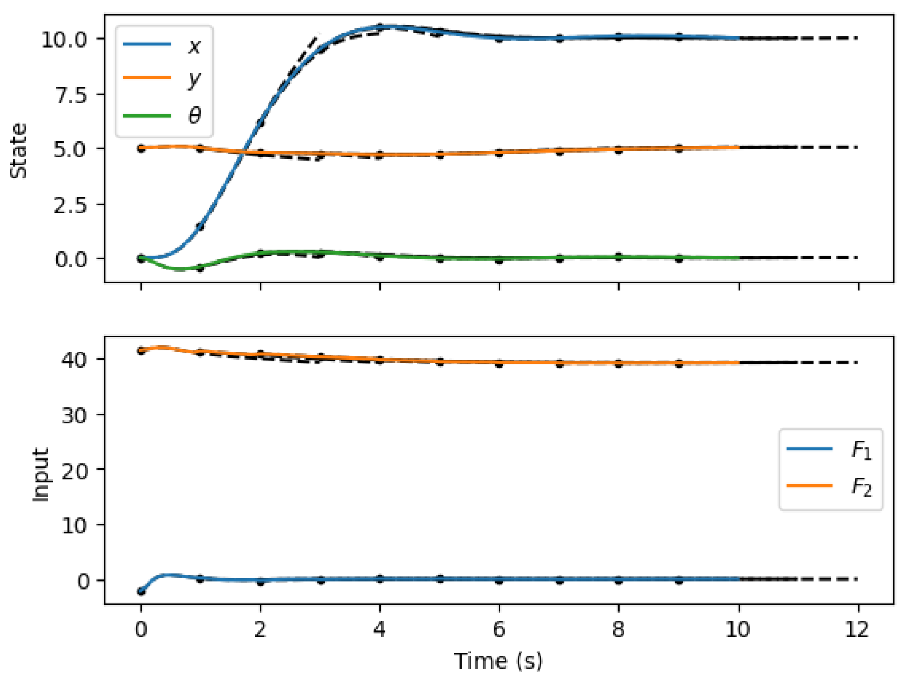
```
t, t_pred, t_sim, x, u, x_pred, u_pred = simulate_mpc(
    pvtol_windy_ctrl, ocp=ocp, t_end=t_end, dt=dt, x0=x0, u0=ueq,
    disturbance_fun=np.sin, feedback=True
)
```

```
Simulation time: 0.00 s, Trajectory computation time: 3.72 s

Simulation time: 1.00 s, Trajectory computation time: 4.20 s

Simulation time: 2.00 s, Trajectory computation time: 4.06 s
```

```
Simulation time: 3.00 s, Trajectory computation time: 3.83 s

Simulation time: 4.00 s, Trajectory computation time: 3.85 s

Simulation time: 5.00 s, Trajectory computation time: 3.83 s

Simulation time: 6.00 s, Trajectory computation time: 3.84 s

Simulation time: 7.00 s, Trajectory computation time: 3.85 s

Simulation time: 8.00 s, Trajectory computation time: 4.02 s

Simulation time: 9.00 s, Trajectory computation time: 3.86 s
```

Plot the predicted and simulated states and inputs:

```
fig, ax = plot_results(
    t, t_pred, t_sim, x, u, x_pred, u_pred, feedback=True
)
plt.show()
```



This result shows that the trajectory tracking is significantly improved with the feedback term. The controller can adjust the input more frequently (instead of only once per second) to account for the wind disturbance and track the desired trajectory more accurately.

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 pickle
import time
import control
import control.optimal as opt
import pydmd
import pysindy
import keras
from keras.models import Sequential
from keras.layers import Dense, Input, Activation
from keras import optimizers
import tensorflow as tf
```

Set up some flags for running the MPC simulations and test predictions:

```
control_lorenz = True
control_DMDc = True
control_SINDYc = True
control_NN = True

retrain = True  # Retrain the NN model

test_DMDc = True
test_SINDYc = True
test_NN = True
```

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
```

```python
        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)
                ),
                "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
```

```python
def simulate(self):
    for i in range(self.n_updates):
        print(f"Simulating update {i+1}/{self.n_updates} ... ", end="", flush=True)
        tic = time.time()
        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)
        toc = time.time()
        print(f"done in {toc-tic:.2f} s.")
        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, title="MPC Simulation"):
    fig, ax = plt.subplots(2, 1, sharex=True)
    fig.suptitle(title)
    # 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,
```

```python
                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

    def save(self, filename):
        """Save an MPC simulation object to a pickle file"""
        with open(filename, 'wb') as f:
            pickle.dump(self, f)

    @classmethod
    def load(self, filename):
        """Load an MPC simulation object from a pickle file"""
        with open(filename, 'rb') as f:
            return pickle.load(f)
```

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 approach we use is to numerically solve an optimal control problem to determine the optimal future inputs. The following function does this:

```
def predict_trajectory(xd, t_horizon, sys):
    """Predict trajectory 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
    ocp = opt.OptimalControlProblem(
        sys, t_horizon, cost, terminal_cost=terminal_cost
    )
    res = ocp.compute_trajectory(xd[:, 0], print_summary=False)
    u = res.inputs
    x = res.states
```

```python
    return x, u
```

This isn't quite specific enough to be used as a predictor function, but it can be used to write a predictor function for each case. Here is one for the Lorenz system:

```python
def lorenz_predictor(xd, t_horizon):
    """Predictor for Lorenz system using optimal control"""
    x, u = predict_trajectory(xd, t_horizon, lorenz_forced_sys)
    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.

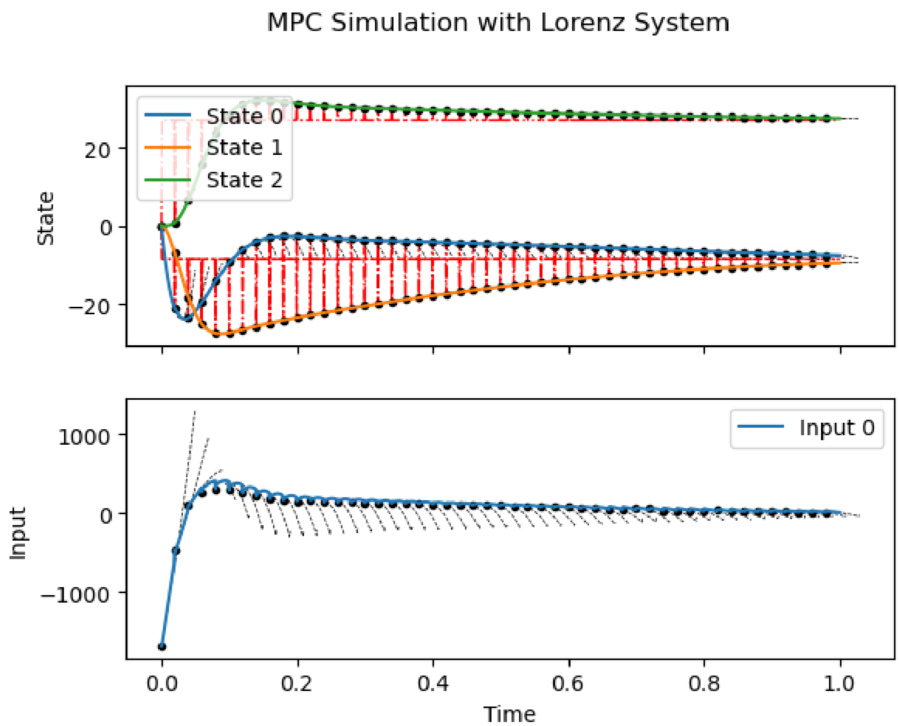
```python
dt_lorenz = 1e-3  # Time step
T_horizon = dt_lorenz * 50
T_update = dt_lorenz * 20
n_horizon = int(np.floor(T_horizon/dt_lorenz)) + 1
n_update = int(np.floor(T_update/dt_lorenz)) + 1
n_updates = 50
xeq = np.array([-np.sqrt(72), -np.sqrt(72), 27])
print(f"xeq: {xeq}")
command = np.outer(xeq, np.ones(n_update * n_updates + 1))
command[:, 0] = np.array([0, 0, 0])  # Initial state
if not control_lorenz:
    mpc_lorenz = MPCSimulation.load("mpc_lorenz.pickle")
else:
    mpc_lorenz = MPCSimulation(
        sys=lorenz_forced_sys,
        inplist=['u'],
        outlist=['lorenz_forced_sys.x', 'lorenz_forced_sys.y', 'lorenz_forced_sys.z'],
        predictor=lorenz_predictor,
        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.save("mpc_lorenz.pickle")
mpc_lorenz.plot_results("MPC Simulation with Lorenz System")
plt.draw()
```

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

done in 3.81 s.
```

```
Simulating update 2/50 ...
done in 4.33 s.
Simulating update 3/50 ...
done in 6.53 s.
Simulating update 4/50 ...
done in 4.55 s.
Simulating update 5/50 ...
done in 3.71 s.
Simulating update 6/50 ...
done in 3.52 s.
Simulating update 7/50 ...
done in 3.71 s.
Simulating update 8/50 ...
done in 3.51 s.
Simulating update 9/50 ...
done in 3.90 s.
Simulating update 10/50 ...
done in 3.82 s.
Simulating update 11/50 ...
done in 3.48 s.
Simulating update 12/50 ...
done in 5.40 s.
Simulating update 13/50 ...
done in 3.53 s.
Simulating update 14/50 ...
done in 3.32 s.
Simulating update 15/50 ...
done in 3.62 s.
Simulating update 16/50 ...
done in 3.43 s.
Simulating update 17/50 ...
done in 3.52 s.
Simulating update 18/50 ...
done in 3.42 s.
Simulating update 19/50 ...
done in 3.34 s.
Simulating update 20/50 ...
```

```
done in 3.26 s.
Simulating update 21/50 ...
done in 2.92 s.
Simulating update 22/50 ...
done in 3.22 s.
Simulating update 23/50 ...
done in 3.30 s.
Simulating update 24/50 ...
done in 3.60 s.
Simulating update 25/50 ...
done in 3.82 s.
Simulating update 26/50 ...
done in 3.43 s.
Simulating update 27/50 ...
done in 3.22 s.
Simulating update 28/50 ...
done in 3.34 s.
Simulating update 29/50 ...
done in 3.51 s.
Simulating update 30/50 ...
done in 3.02 s.
Simulating update 31/50 ...
done in 2.93 s.
Simulating update 32/50 ...
done in 2.92 s.
Simulating update 33/50 ...
done in 2.82 s.
Simulating update 34/50 ...
done in 1.93 s.
Simulating update 35/50 ...
done in 1.95 s.
Simulating update 36/50 ...
done in 3.90 s.
Simulating update 37/50 ...
done in 3.31 s.
Simulating update 38/50 ...
```

```
done in 1.92 s.
Simulating update 39/50 ...
done in 2.04 s.
Simulating update 40/50 ...
done in 1.95 s.
Simulating update 41/50 ...
done in 1.93 s.
Simulating update 42/50 ...
done in 2.72 s.
Simulating update 43/50 ...
done in 2.02 s.
Simulating update 44/50 ...
done in 2.06 s.
Simulating update 45/50 ...
done in 2.02 s.
Simulating update 46/50 ...
done in 2.04 s.
Simulating update 47/50 ...
done in 2.02 s.
Simulating update 48/50 ...
done in 2.02 s.
Simulating update 49/50 ...
done in 2.04 s.
Simulating update 50/50 ...
done in 2.03 s.
```



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

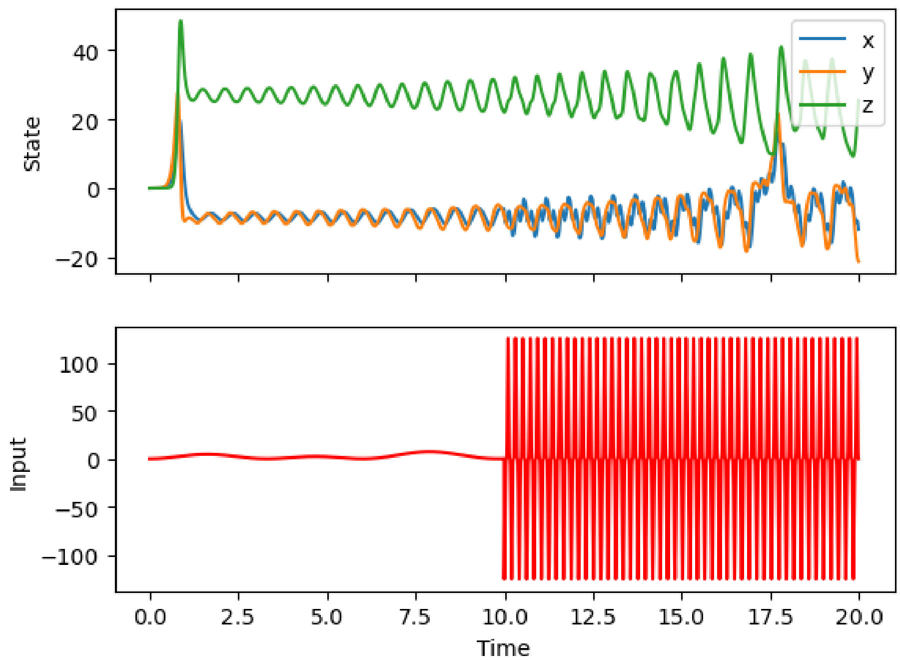
Now we generate some training and testing data for the predictors. Generate the training and testing data as follows:

```
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:
[[ 0.06790008  0.0725858  -0.24087446]
 [ 0.0725858   0.07759488 -0.25749698]
 [-0.24087446 -0.25749698  0.85449833]]
B_pydmd:
[[-0.00012709]
 [-0.00013582]
 [ 0.00045124]]
```

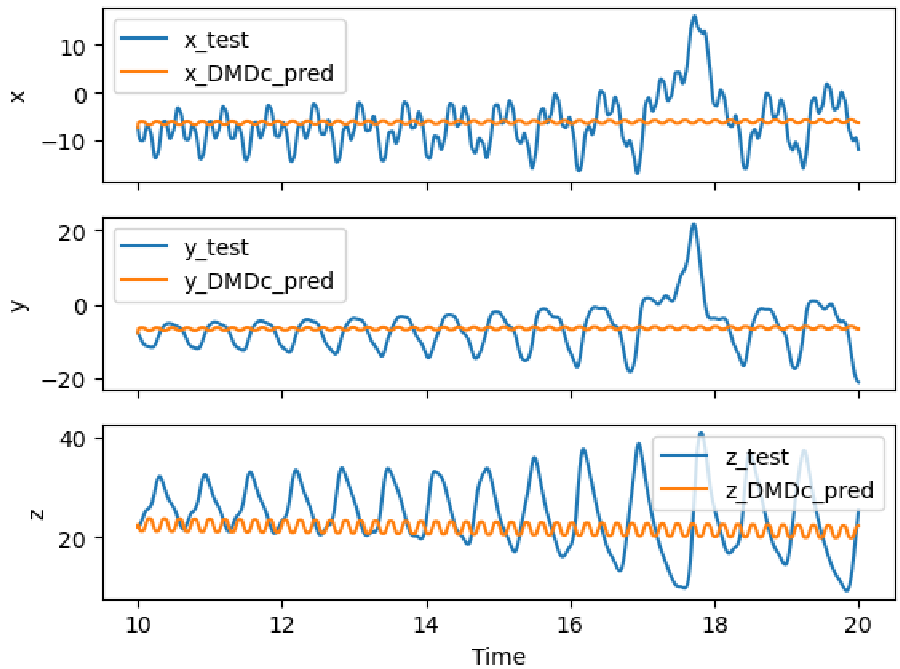
Predict the trajectory on the test data:

```
if test_DMDc:
    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:

```
if test_DMDc:
    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):
    """Predictor for DMDc model using optimal control"""
    x, u = predict_trajectory(xd, t_horizon, sys_DMDc)
    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_data * 50
T_update = dt_data * 20
n_horizon = int(np.floor(T_horizon/dt_data)) + 1  # Must match DMDc model timebase
```

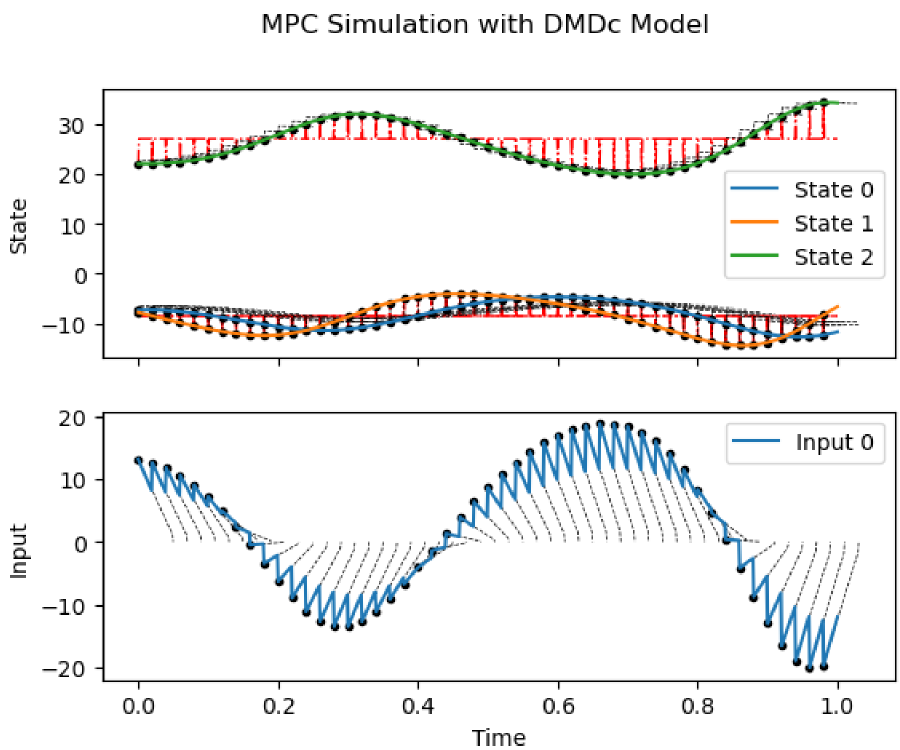
```python
n_update = int(np.floor(T_update/dt_data)) + 1
n_updates = 50
xeq = np.array([-np.sqrt(72), -np.sqrt(72), 27])
print(f"xeq: {xeq}")
command = np.outer(xeq, np.ones(n_update * n_updates + 1))
command[:, 0] = x_test[:, 0]  # Initial state
if not control_DMDc:
    mpc_DMDc = MPCSimulation.load("mpc_DMDc.pickle")
else:
    sys_DMDc = control.ss(A_pydmd, B_pydmd, np.eye(A_pydmd.shape[0]), 0, dt=dt_data)
    mpc_DMDc = MPCSimulation(
        sys=lorenz_forced_sys,
        inplist=['u'],
        outlist=['lorenz_forced_sys.x', 'lorenz_forced_sys.y', 'lorenz_forced_sys.z'],
        predictor=DMDc_predictor,
        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.save("mpc_DMDc.pickle")
mpc_DMDc.plot_results("MPC Simulation with DMDc Model")
plt.draw()
```

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

done in 2.30 s.
Simulating update 2/50 ...

done in 6.13 s.
Simulating update 3/50 ...

done in 3.27 s.
Simulating update 4/50 ...

done in 4.76 s.
Simulating update 5/50 ...

done in 3.63 s.
Simulating update 6/50 ...

done in 3.79 s.
Simulating update 7/50 ...

done in 4.59 s.
Simulating update 8/50 ...
```

```
done in 3.84 s.
Simulating update 9/50 ...

done in 0.56 s.
Simulating update 10/50 ...

done in 3.89 s.
Simulating update 11/50 ...

done in 2.36 s.
Simulating update 12/50 ...

done in 8.45 s.
Simulating update 13/50 ...

done in 3.86 s.
Simulating update 14/50 ...

done in 2.21 s.
Simulating update 15/50 ...

done in 1.72 s.
Simulating update 16/50 ...

done in 3.74 s.
Simulating update 17/50 ...

done in 2.53 s.
Simulating update 18/50 ...

done in 2.20 s.
Simulating update 19/50 ...

done in 2.67 s.
Simulating update 20/50 ...

done in 7.37 s.
Simulating update 21/50 ...

done in 4.28 s.
Simulating update 22/50 ...

done in 9.62 s.
Simulating update 23/50 ...

done in 0.66 s.
Simulating update 24/50 ...

done in 4.13 s.
Simulating update 25/50 ...

done in 5.18 s.
Simulating update 26/50 ...
```

```
done in 6.28 s.
Simulating update 27/50 ...
done in 0.98 s.
Simulating update 28/50 ...
done in 0.98 s.
Simulating update 29/50 ...
done in 1.94 s.
Simulating update 30/50 ...
done in 3.39 s.
Simulating update 31/50 ...
done in 5.20 s.
Simulating update 32/50 ...
done in 1.63 s.
Simulating update 33/50 ...
done in 7.22 s.
Simulating update 34/50 ...
done in 4.58 s.
Simulating update 35/50 ...
done in 1.46 s.
Simulating update 36/50 ...
done in 1.97 s.
Simulating update 37/50 ...
done in 2.39 s.
Simulating update 38/50 ...
done in 7.52 s.
Simulating update 39/50 ...
done in 3.62 s.
Simulating update 40/50 ...
done in 4.69 s.
Simulating update 41/50 ...
done in 3.85 s.
Simulating update 42/50 ...
done in 2.88 s.
Simulating update 43/50 ...
done in 0.59 s.
Simulating update 44/50 ...
```

```
done in 5.10 s.
Simulating update 45/50 ...

done in 2.21 s.
Simulating update 46/50 ...

done in 5.27 s.
Simulating update 47/50 ...

done in 2.05 s.
Simulating update 48/50 ...

done in 3.21 s.
Simulating update 49/50 ...

done in 4.59 s.
Simulating update 50/50 ...

done in 1.58 s.
```



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:

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

def extract_sindy_dynamics(sindy_model, eps=1e-12):
    """Extract SINDy dynamics"""
    variables = sindy_model.feature_names  # ["x", "y", "z", "u"]
    coefficients = sindy_model.coefficients()
    features = sindy_model.get_feature_names()
        # ["1", "x", "y", "z", "u", "x * y", "x * z", "x * u", "y * z", ...]
    features = [f.replace("^", "**") for f in features]
    features = [f.replace(" ", " * ") for f in features]
    def rhs(coefficients, features):
        rhs = []
        for row in range(coefficients.shape[0]):
            rhs_row = ""
            for col in range(coefficients.shape[1]):
                if np.abs(coefficients[row, col]) > eps:
                    if rhs_row:
                        rhs_row += " + "
                    rhs_row += f"{coefficients[row, col]} * {features[col]}"
            rhs.append(rhs_row)
        return rhs
    rhs_str = rhs(coefficients, features)  # Eager evaluation
    n_equations = len(rhs_str)
    def sindy_dynamics(t, x_, u_, params={}):
        states_inputs = x_.tolist() + np.atleast_1d(u_).tolist()
        variables_dict = dict(zip(variables, states_inputs))
        return [eval(rhs_str[i], variables_dict) for i in range(n_equations)]
    return sindy_dynamics
```

```
Dynamics identified by pySINDy:
(x)' = -0.208 1 + -10.020 x + 10.023 y + 1.055 u
(y)' = 27.866 x + -1.008 y + -0.994 x z
(z)' = -2.662 z + 0.998 x y
```

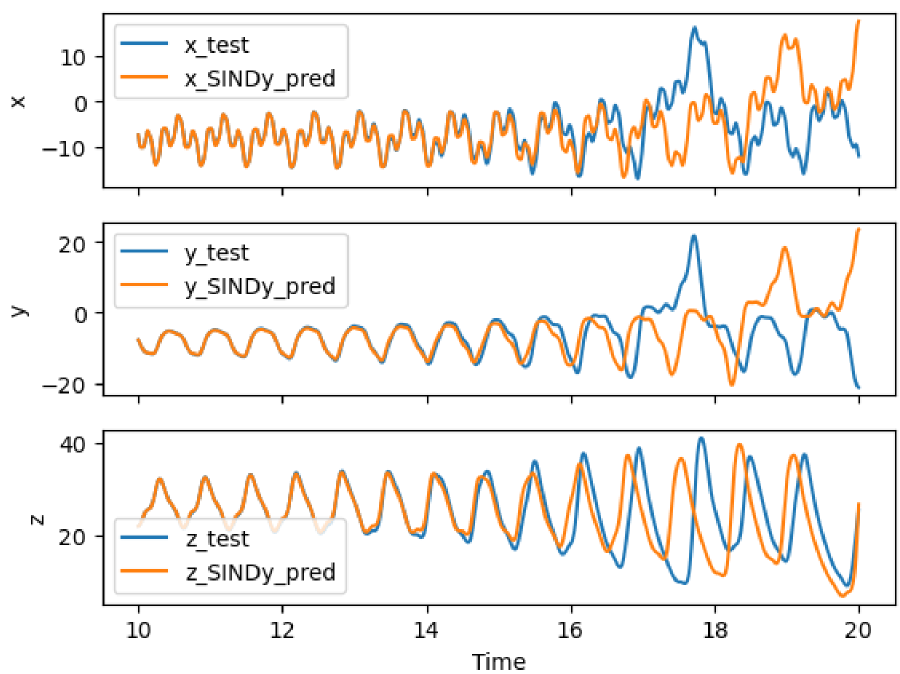
Let's predict the trajectory on the test data:

```
if test_SINDYc:
    sindy_dynamics = extract_sindy_dynamics(sindy_model)
    sindy_sys = control.NonlinearIOSystem(
        sindy_dynamics, None, inputs=["u"], states=["x", "y", "z"],
        name="sindy_sys"
```

```
    )
    x_SINDy_pred = control.input_output_response(
        sindy_sys, T=t_test, U=u_test, X0=x_test[:, 0]
    ).states
```

Plot the simulated and predicted trajectory with the test data:

```
if test_SINDYc:
    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.

```
sindy_dynamics = extract_sindy_dynamics(sindy_model)
sys_SINDy = control.NonlinearIOSystem(
    sindy_dynamics, None, inputs=["u"], states=["x", "y", "z"],
    name="sys_SINDy"
)
```

Define the SINDy predictor function:

```
def SINDy_predictor(xd, t_horizon):
    """Predictor for SINDy model using optimal control"""
    x, u = predict_trajectory(xd, t_horizon, sys_SINDy)
    return x, u
```

Now we can test the SINDy model in the MPC simulation. We expect the results to be about as good as the exact model.

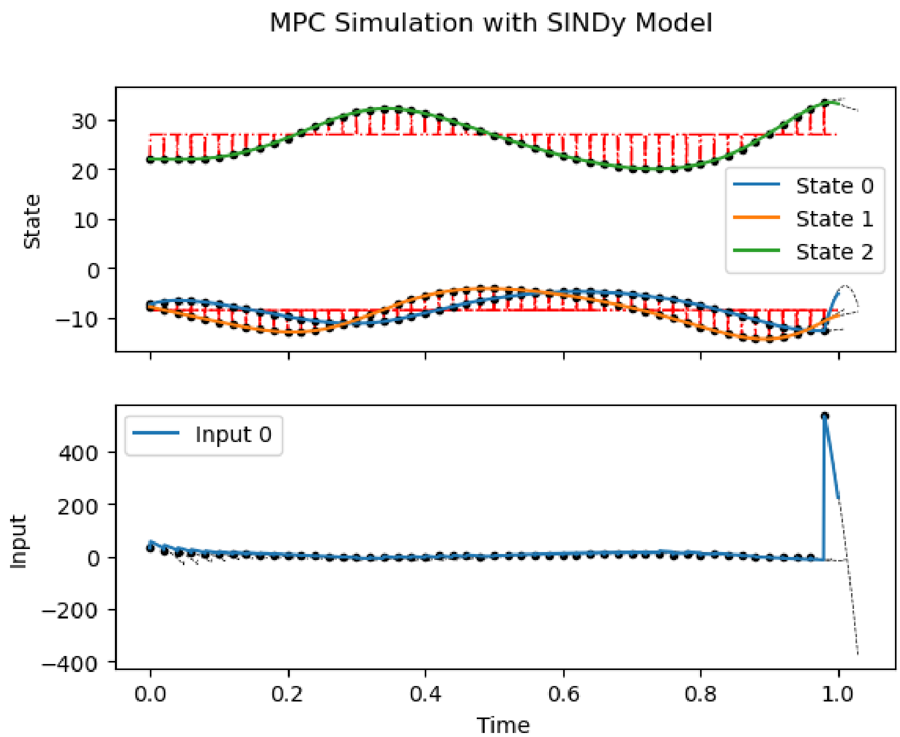
```
if not control_SINDYc:
    mpc_SINDYc = MPCSimulation.load("mpc_SINDYc.pickle")
else:
    mpc_SINDYc = MPCSimulation(
        sys=lorenz_forced_sys,
        inplist=['u'],
        outlist=['lorenz_forced_sys.x', 'lorenz_forced_sys.y', 'lorenz_forced_sys.z'],
        predictor=SINDy_predictor,
        T_horizon=T_horizon,
        T_update=T_update,
        n_updates=n_updates,
        n_horizon=n_horizon,
        n_update=n_update,
        xd=command
    )
    results_mpc_SINDYc = mpc_SINDYc.simulate()
    mpc_SINDYc.save("mpc_SINDYc.pickle")
mpc_SINDYc.plot_results("MPC Simulation with SINDy Model")
plt.draw()
```

```
Simulating update 1/50 ...

done in 6.30 s.
Simulating update 2/50 ...

done in 6.25 s.
Simulating update 3/50 ...
```

```
done in 6.23 s.
Simulating update 4/50 ...

done in 6.27 s.
Simulating update 5/50 ...

done in 6.23 s.
Simulating update 6/50 ...

done in 6.31 s.
Simulating update 7/50 ...

done in 6.24 s.
Simulating update 8/50 ...

done in 6.20 s.
Simulating update 9/50 ...

done in 6.22 s.
Simulating update 10/50 ...

done in 6.22 s.
Simulating update 11/50 ...

done in 7.79 s.
Simulating update 12/50 ...

done in 6.26 s.
Simulating update 13/50 ...

done in 6.46 s.
Simulating update 14/50 ...

done in 6.28 s.
Simulating update 15/50 ...

done in 6.29 s.
Simulating update 16/50 ...

done in 3.73 s.
Simulating update 17/50 ...

done in 3.71 s.
Simulating update 18/50 ...

done in 6.22 s.
Simulating update 19/50 ...

done in 6.20 s.
Simulating update 20/50 ...

done in 6.22 s.
Simulating update 21/50 ...
```

```
done in 6.22 s.
Simulating update 22/50 ...
done in 6.27 s.
Simulating update 23/50 ...
done in 6.26 s.
Simulating update 24/50 ...
done in 6.25 s.
Simulating update 25/50 ...
done in 1.53 s.
Simulating update 26/50 ...
done in 3.71 s.
Simulating update 27/50 ...
done in 3.74 s.
Simulating update 28/50 ...
done in 3.72 s.
Simulating update 29/50 ...
done in 3.71 s.
Simulating update 30/50 ...
done in 3.74 s.
Simulating update 31/50 ...
done in 3.71 s.
Simulating update 32/50 ...
done in 3.73 s.
Simulating update 33/50 ...
done in 3.72 s.
Simulating update 34/50 ...
done in 3.72 s.
Simulating update 35/50 ...
done in 3.73 s.
Simulating update 36/50 ...
done in 3.73 s.
Simulating update 37/50 ...
done in 3.69 s.
Simulating update 38/50 ...
done in 6.60 s.
Simulating update 39/50 ...
```

```
done in 3.73 s.
Simulating update 40/50 ...
done in 6.27 s.
Simulating update 41/50 ...
done in 3.74 s.
Simulating update 42/50 ...
done in 6.23 s.
Simulating update 43/50 ...
done in 6.30 s.
Simulating update 44/50 ...
done in 6.02 s.
Simulating update 45/50 ...
done in 3.74 s.
Simulating update 46/50 ...
done in 3.76 s.
Simulating update 47/50 ...
done in 3.73 s.
Simulating update 48/50 ...
done in 3.72 s.
Simulating update 49/50 ...
done in 3.71 s.
Simulating update 50/50 ...
done in 11.85 s.
```



The results are quite good, very similar to the results with the exact model. The SINDy model is a good choice for the MPC simulation.

Neural Network (NN) Model

We can train a NN model to predict the Lorenz system dynamics, as in Brunton and Kutz (2022) problem 6.1d.

Begin by defining the neural network architecture:

```
def build_model():
    """Build the feedforward neural network model"""
    model = Sequential()
    model.add(Input(shape=(4,)))  # 3 states + 1 input
    model.add(Dense(5))
    model.add(Activation('relu'))
    model.add(Dense(5))
    model.add(Activation('relu'))
    model.add(Dense(5))
    model.add(Activation('relu'))
    model.add(Dense(5))
    model.add(Activation('relu'))
```

```
    model.add(Dense(5))
    model.add(Activation('relu'))
    model.add(Dense(5))
    model.add(Activation('relu'))
    model.add(Dense(5))
    model.add(Activation('relu'))
    model.add(Dense(3))  # 3 states
    return model
```

Compile the model:

```
model = build_model()
model.compile(
    optimizer=optimizers.Adam(learning_rate=0.0001),
    loss='mean_squared_error',  # Loss function
    # metrics=['mean_absolute_error'],  # Metrics to monitor
)
```

Generate extra training data for the NN model, using a random input:

```
t_train_extra = np.arange(0, 5000, dt_data)  # Time array
n_train_extra = len(t_train_extra)
u_train_extra = 50 * np.random.randn(n_train_extra)
X = np.hstack([30 * np.random.randn(n_train_extra, 2), 10 * np.random.randn(n_train_extra, 1
Y = np.zeros((n_train_extra, 3))
for i in range(0, n_train_extra):
    Y[i, :] = X[i, :3] + dt_data * np.array(lorenz_forced(t_train_extra[i], X[i, :3], u_trai
# x_train_extra = control.input_output_response(
#     lorenz_forced_sys, T=t_train_extra, U=u_train_extra
# ).states
```

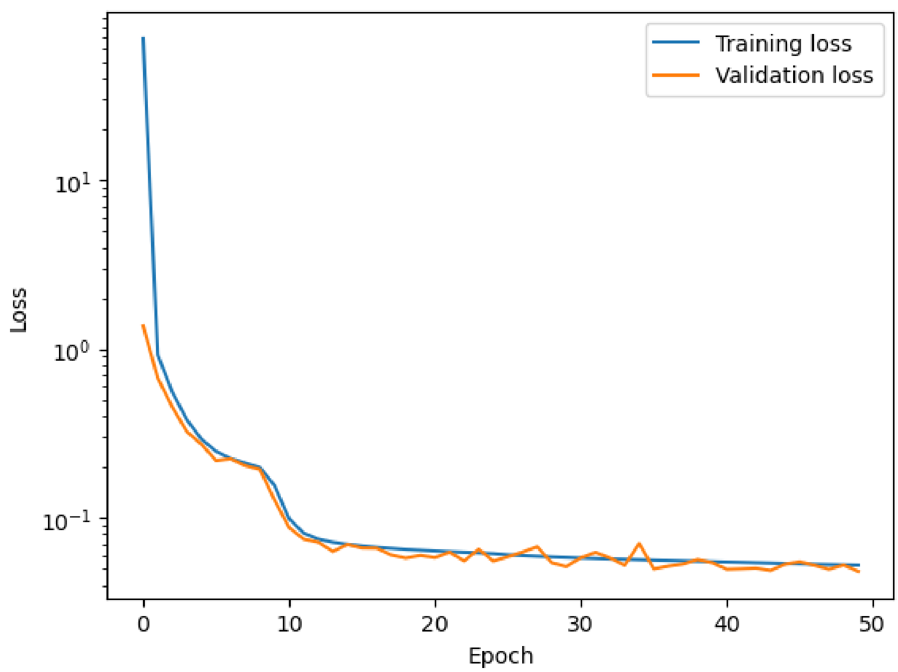
Train the model:

```
# X = np.hstack([x_train[:, :-1].T, u_train[:-1].reshape(-1, 1)])
# X_extra = np.hstack([x_train_extra[:, :-1].T, u_train_extra[:-1].reshape(-1, 1)])
# # X = np.vstack([X, X_extra])
# X = X_extra
# Y = x_train[:, 1:].T
# Y_extra = x_train_extra[:, 1:].T
# # Y = np.vstack([Y, Y_extra])
# Y = Y_extra
if retrain:
    history = model.fit(
        X,  # Input data
        Y,  # Target data
        epochs=50,  # Number of epochs
        # batch_size=1,  # 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()
```



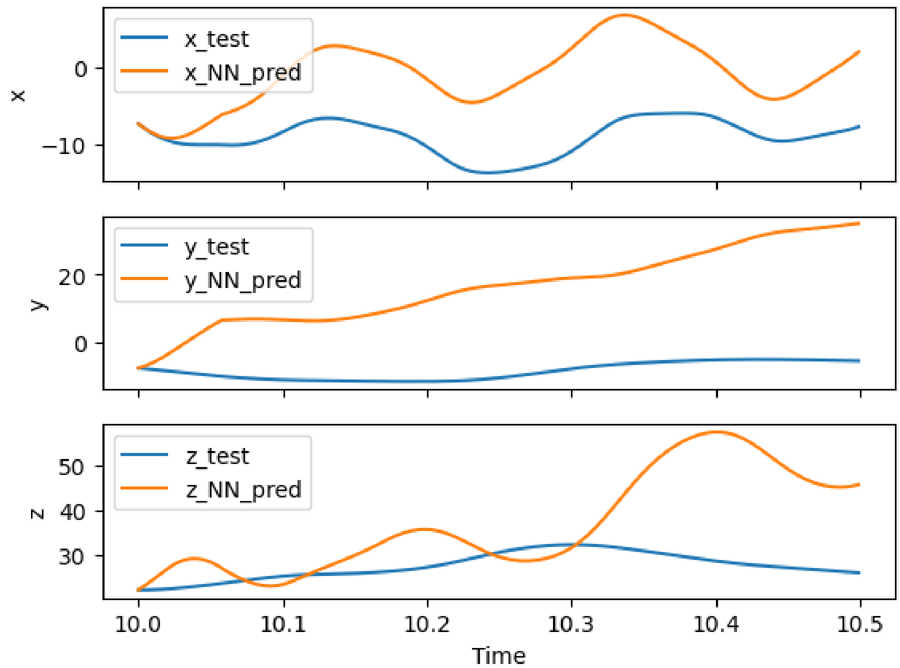
Now we can test the NN model on the test data. To predict the trajectory, we could use the `model.predict()` method, but it is slow to predict many individual points (it's better for a batch prediction, but the next state depends on the previous state, so we can't do that). So we may as well create a `control.NonlinearIOSystem` object from the NN model and use the TensorFlow function to predict the next state, which we will need anyway for the MPC simulation.

```
@tf.function  # Decorator for TensorFlow function
def NN_dynamics_tf(X):
    """Dynamics for NN model as a TensorFlow function (fast)"""
    return model(X)
def NN_dynamics(t, x_, u_, params={}):
    """NN dynamics"""
    X = np.hstack([x_, u_])[np.newaxis, :]
    return NN_dynamics_tf(X).numpy().flatten()
sys_NN = control.NonlinearIOSystem(
    NN_dynamics, None, inputs=["u"], states=["x", "y", "z"],
    dt=dt_data, name="sys_NN"
)
```

```
if test_NN:
    NN_pred = control.input_output_response(
        sys_NN, T=t_test, U=u_test, X0=x_test[:, 0]
    ).states
```

Plot the predicted trajectory with the test data:

```
if test_NN:
    maxi = 500
    fig, ax = plt.subplots(3, 1, sharex=True)
    ax[0].plot(t_test[:maxi], x_test[0, :maxi], label='x_test')
    ax[0].plot(t_test[:maxi], NN_pred[0, :maxi], label='x_NN_pred')
    ax[0].set_ylabel('x')
    ax[0].legend()
    ax[1].plot(t_test[:maxi], x_test[1, :maxi], label='y_test')
    ax[1].plot(t_test[:maxi], NN_pred[1, :maxi], label='y_NN_pred')
    ax[1].set_ylabel('y')
    ax[1].legend()
    ax[2].plot(t_test[:maxi], x_test[2, :maxi], label='z_test')
    ax[2].plot(t_test[:maxi], NN_pred[2, :maxi], label='z_NN_pred')
    ax[2].set_ylabel('z')
    ax[2].set_xlabel('Time')
    ax[2].legend()
    plt.draw()
```



The results are pretty terrible. Now create a predictor function for the NN model:

```
def NN_predictor(xd, t_horizon):
    """Predictor for NN model using optimal control"""
    x, u = predict_trajectory(xd, t_horizon, sys_NN)
    return x, u
```

Now we can test the NN model in the MPC simulation.

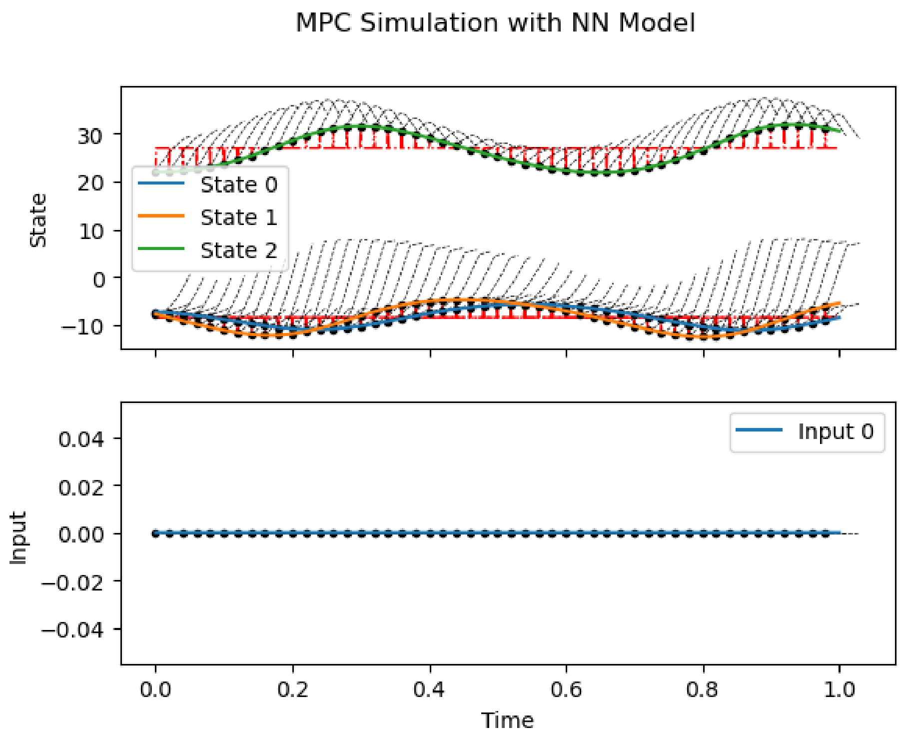
```
if not control_NN:
    mpc_NN = MPCSimulation.load("mpc_NN.pickle")
else:
    mpc_NN = MPCSimulation(
        sys=lorenz_forced_sys,
        inplist=['u'],
        outlist=['lorenz_forced_sys.x', 'lorenz_forced_sys.y', 'lorenz_forced_sys.z'],
        predictor=NN_predictor,
        T_horizon=T_horizon,
        T_update=T_update,
        n_updates=n_updates,
        n_horizon=n_horizon,
        n_update=n_update,
        xd=command
    )
```

```
    results_mpc_NN = mpc_NN.simulate()
    mpc_NN.save("mpc_NN.pickle")
mpc_NN.plot_results("MPC Simulation with NN Model")
plt.draw()
```

```
Simulating update 1/50 ...

done in 0.60 s.
Simulating update 2/50 ...

done in 0.57 s.
Simulating update 3/50 ...

done in 0.59 s.
Simulating update 4/50 ...

done in 0.57 s.
Simulating update 5/50 ...

done in 0.61 s.
Simulating update 6/50 ...

done in 0.60 s.
Simulating update 7/50 ...

done in 0.59 s.
Simulating update 8/50 ...

done in 0.60 s.
Simulating update 9/50 ...

done in 0.66 s.
Simulating update 10/50 ...

done in 0.69 s.
Simulating update 11/50 ...

done in 0.60 s.
Simulating update 12/50 ...

done in 0.59 s.
Simulating update 13/50 ...

done in 0.66 s.
Simulating update 14/50 ...

done in 0.67 s.
Simulating update 15/50 ...

done in 0.64 s.
Simulating update 16/50 ...

done in 0.56 s.
Simulating update 17/50 ...
```

```
done in 0.58 s.
Simulating update 18/50 ...
done in 0.57 s.
Simulating update 19/50 ...
done in 0.56 s.
Simulating update 20/50 ...
done in 0.54 s.
Simulating update 21/50 ...
done in 0.59 s.
Simulating update 22/50 ...
done in 0.68 s.
Simulating update 23/50 ...
done in 0.62 s.
Simulating update 24/50 ...
done in 0.63 s.
Simulating update 25/50 ...
done in 0.63 s.
Simulating update 26/50 ...
done in 0.55 s.
Simulating update 27/50 ...
done in 0.56 s.
Simulating update 28/50 ...
done in 0.56 s.
Simulating update 29/50 ...
done in 0.58 s.
Simulating update 30/50 ...
done in 0.61 s.
Simulating update 31/50 ...
done in 0.59 s.
Simulating update 32/50 ...
done in 0.60 s.
Simulating update 33/50 ...
done in 0.62 s.
Simulating update 34/50 ...
done in 0.64 s.
Simulating update 35/50 ...
```

```
done in 0.63 s.
Simulating update 36/50 ...

done in 0.64 s.
Simulating update 37/50 ...

done in 0.55 s.
Simulating update 38/50 ...

done in 0.56 s.
Simulating update 39/50 ...

done in 0.54 s.
Simulating update 40/50 ...

done in 0.59 s.
Simulating update 41/50 ...

done in 0.61 s.
Simulating update 42/50 ...

done in 0.59 s.
Simulating update 43/50 ...

done in 0.55 s.
Simulating update 44/50 ...

done in 0.56 s.
Simulating update 45/50 ...

done in 0.55 s.
Simulating update 46/50 ...

done in 0.55 s.
Simulating update 47/50 ...

done in 0.55 s.
Simulating update 48/50 ...

done in 0.55 s.
Simulating update 49/50 ...

done in 0.56 s.
Simulating update 50/50 ...

done in 0.54 s.
```


MPC Simulation with NN Model
State
State 0
State 1
State 2
30
20
10
0
−10
Input
Input 0
0.04
0.02
0.00
−0.02
−0.04
0.0
0.2
0.4
0.6
0.8
1.0
Time

```
plt.show()
```