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:

```
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:

```
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
```

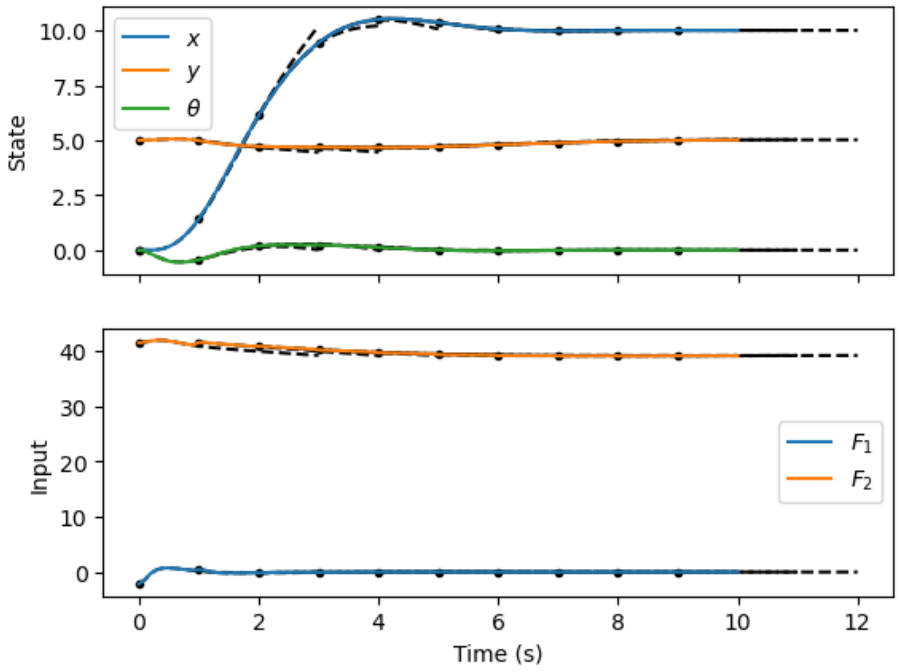
Plot the predicted and simulated states and inputs. First, define a function to plot the results:

```
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
)
```

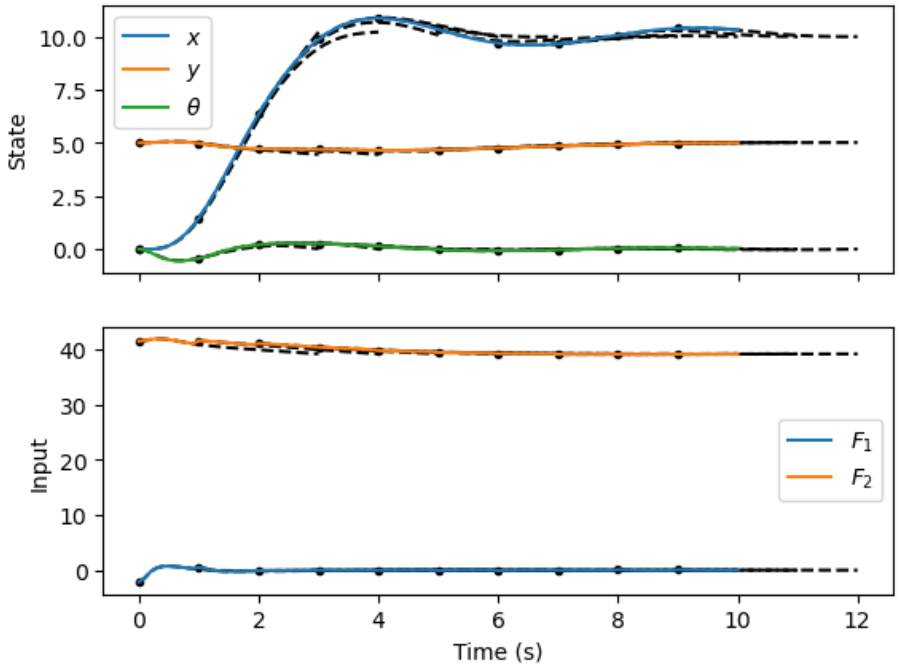
```
Simulation time: 0.00 s, Trajectory computation time: 3.71 s
```

```
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
```

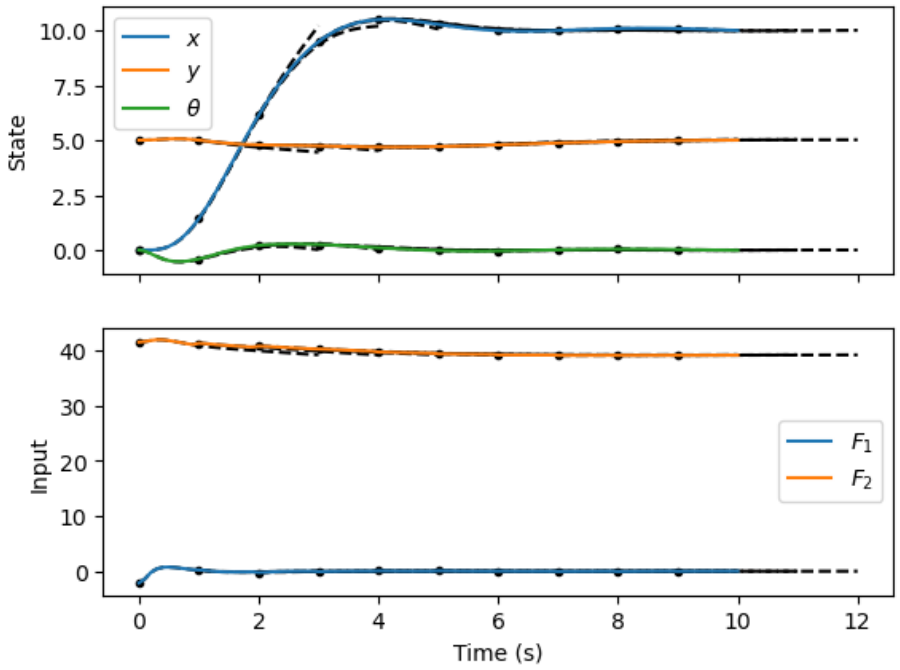
```
Simulation time: 1.00 s, Trajectory computation time: 4.20 s
```

```
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.