# Murray Problem 3.12: Trajectory Optimization of a Thrust-Vectoring Aircraft

Source Filename: /main.py

Rico A. R. Picone

In this problem, we will compute optimal trajectories for the PVTOL system using two diferent methods: numerical optimization and exploiting (near) differential flatness. The numerical optimization method allows us to compute an optimal trajectory $(x_d(t), u_d(t))$ that minimizes a cost function $J$ subject to the dynamics of the system. The numerical nature of this approach requires some care in selecting the parameters of the optimization problem, such as the initial guess, the cost function, and the constraints. The `control.optimal` module provides a function `solve_ocp()` that can be used to solve optimal control problems of this form.

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 control.flatsys as fs
import time
from pvtol import pvtol, plot_results
```

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]
```

Set up the optimization problem parameters as follows:

```
x0 = xeq  # Initial state
u0 = ueq  # Initial input
xf = x0 + np.array([1, 0, 0, 0, 0, 0])  # Final state
Q = np.diag([1, 10, 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=u0)  # J
```

The cost function `cost` is constructed with `x0=xf`, the final state, because this is the desired state where the cost should be zero.

Set up the time horizon, time steps, and initial guess for the trajectory, a straight line from `x0` to `xf`, as follows:

```
tf = 5  # Time horizon
nt = 14  # Number of time steps
t = np.linspace(0, tf, nt)  # Time steps
def init_straight(x0, xf, u0, t):
    """Straight-line trajectory with constant input"""
    nt = t.size
    nstates = x0.size
    return np.vstack([
        np.linspace(x0[i], xf[i], nt) for i in range(nstates)
    ]), np.outer(u0, np.ones_like(t))
x_init, u_init = init_straight(x0, xf, u0, t)
```

Solve the optimal control problem as follows:

```
solve_time_start = time.time()
opt1 = opt.solve_ocp(
    pvtol, t, x0, cost, log=True,
    initial_guess=(x_init, u_init),
)
```

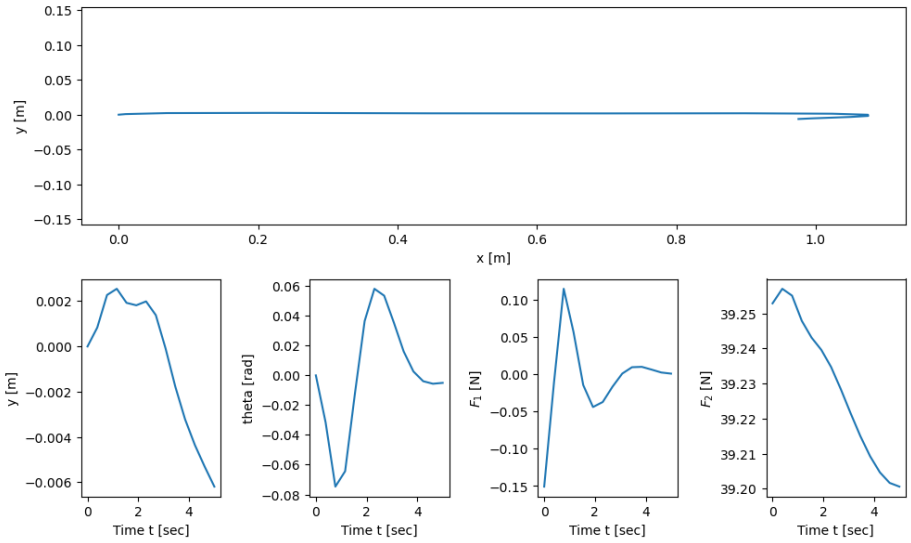
```
solve_time_end = time.time()
print(f"Optimization time: {solve_time_end - solve_time_start:.2f} s")
```

```
Summary statistics:
* Cost function calls: 1821
* Cost function process time: 0.1299670000000015
* System simulations: 0
* Final cost: 1.5053222300852327
Optimization time: 0.42 s
```

Extract the optimal trajectory and plot the results as follows:

```
print(f"Initial position: {opt1.states[0:2, 0]}")
print(f"Final position: {opt1.states[0:2, -1]}")
plot_results(opt1.time, opt1.states, opt1.inputs)
plt.draw()
```

```
Initial position: [0. 0.]
Final position: [ 0.97535942 -0.00619235]
```




We observe that the final value is fairly close to the desired final state. Some overshoot has occurred. The response is not particularly smooth, which is primarily due to the sparsity of the time steps. We could increase the number of time steps to improve the smoothness, but this would also increase the computation time.

Instead, let's try to smooth out the trajectory by using Bezier curves as the parameterization (instead of straight lines between time points).

```
nt2 = 2*nt  # More is OK due to the efficiency of Bezier curves
t2 = np.linspace(0, tf, nt2)
```

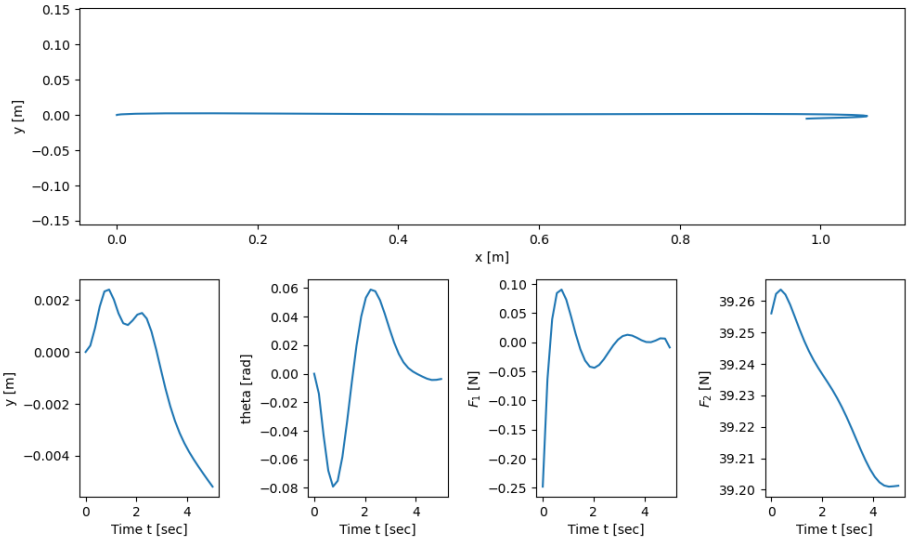
```
x_init2, u_init2 = init_straight(x0, xf, u0, t2)
basis = fs.BezierFamily(8, T=tf)
solve_time_start = time.time()
opt2 = opt.solve_ocp(
    pvtol, t2, x0, cost, log=True,
    initial_guess=(x_init2, u_init2),
    basis=basis,
)
solve_time_end = time.time()
print(f"Optimization time: {solve_time_end - solve_time_start:.2f} s")
```

```
Summary statistics:
* Cost function calls: 5367
* Cost function process time: 4.310419000000037
* System simulations: 0
* Final cost: 1.4445212102307345
Optimization time: 9.70 s
```

Extract the optimal trajectory and plot the results as follows:

```
print(f"Initial position: {opt2.states[0:2, 0]}")
print(f"Final position: {opt2.states[0:2, -1]}")
plot_results(opt2.time, opt2.states, opt2.inputs)
plt.draw()
```

```
Initial position: [0. 0.]
Final position: [ 0.98021887 -0.00520007]
```




The Bezier curve parameterization has smoothed out the trajectory significantly, but the computation time has increased. In some cases, using Bezier curves

instead of straight lines with more time steps can be more efficient.

Now create both a trajectory cost and a terminal cost that penalizes the final state and input as follows:

```
cost_traj = opt.quadratic_cost(pvtol, Q=Q/10, R=R, x0=xf, u0=u0)
cost_term = opt.quadratic_cost(pvtol, Q=Q*10, R=R, x0=xf, u0=u0)
```

The trajectory cost `cost_traj` penalizes the trajectory, while the terminal cost `cost_term` penalizes the final state and input. Solve the optimal control problem with the trajectory and terminal costs as follows:

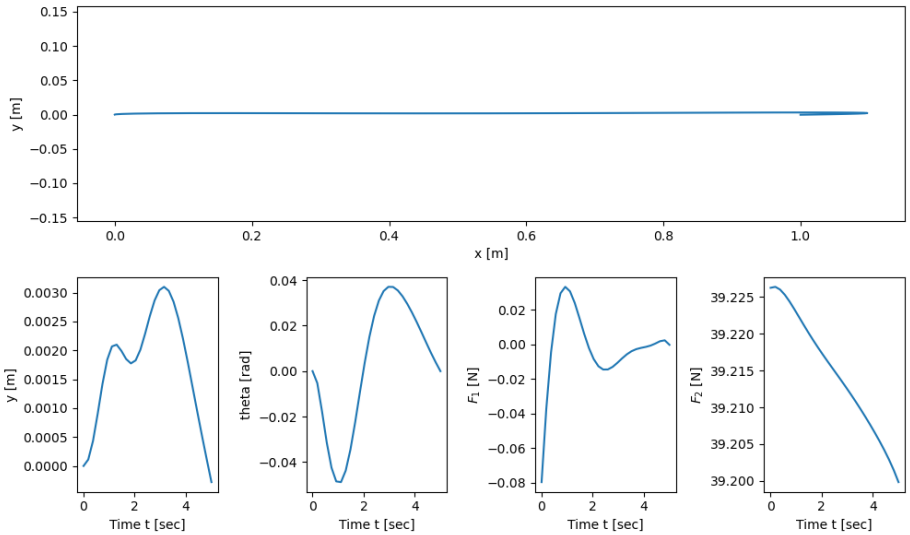
```
solve_time_start = time.time()
opt3 = opt.solve_ocp(
    pvtol, t2, x0, cost_traj, terminal_cost=cost_term,
    initial_guess=(x_init2, u_init2),
    basis=basis,  # Bezier curve basis again
)
solve_time_end = time.time()
print(f"Optimization time: {solve_time_end - solve_time_start:.2f} s")
```

```
Summary statistics:
* Cost function calls: 6293
* System simulations: 0
* Final cost: 0.18210007813464557
Optimization time: 11.28 s
```

Extract the optimal trajectory and plot the results as follows:

```
print(f"Initial position: {opt3.states[0:2, 0]}")
print(f"Final position: {opt3.states[0:2, -1]}")
plot_results(opt3.time, opt3.states, opt3.inputs)
plt.draw()
```

```
Initial position: [0. 0.]
Final position: [ 9.99795695e-01 -2.79186063e-04]
```




The terminal cost has improved the steady-state error.

Now use a terminal *constraint* (not cost) to enforce the final state to be exactly at the desired final state as follows:

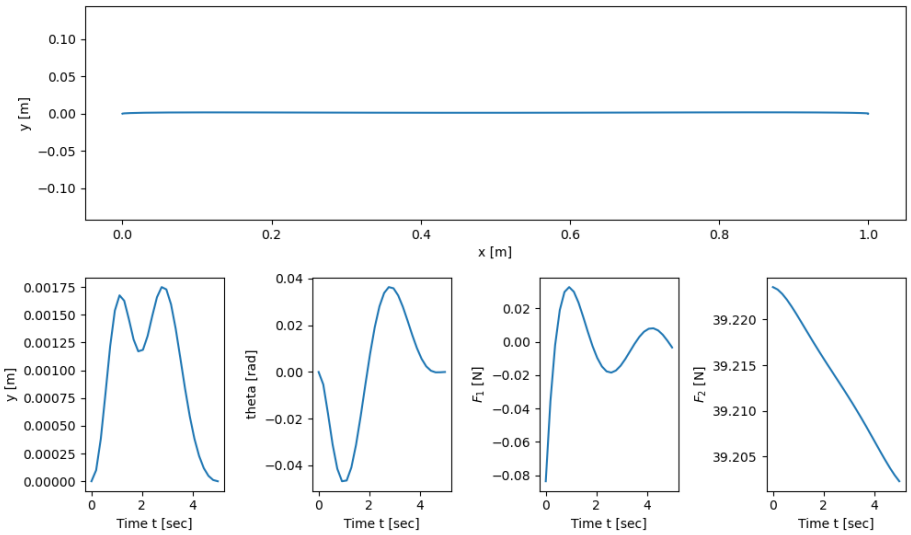
```
con_term = opt.state_range_constraint(pvtol, lb=xf, ub=xf)
solve_time_start = time.time()
opt4 = opt.solve_ocp(
    pvtol, t2, x0, cost_traj, terminal_constraints=con_term,
    initial_guess=(x_init2, u_init2),
    basis=basis,
)
solve_time_end = time.time()
print(f"Optimization time: {solve_time_end - solve_time_start:.2f} s")
```

```
Summary statistics:
* Cost function calls: 2776
* Eqconst calls: 2977
* System simulations: 0
* Final cost: 0.18477758399760782
Optimization time: 7.08 s
```

Extract the optimal trajectory and plot the results as follows:

```
print(f"Initial position: {opt4.states[0:2, 0]}")
print(f"Final position: {opt4.states[0:2, -1]}")
plot_results(opt4.time, opt4.states, opt4.inputs)
plt.draw()
```

```
Initial position: [0. 0.]
Final position: [1. 0.]
```




The terminal constraint has forced the final state to be exactly at the desired final state.

Now we will exploit the (near) differential flatness of the PVTOL system to compute an optimal trajectory. The `pvtol` object is a `control.flatsys.FlatSystem` object. The forward map of $x$ and $u$ and their time derivatives to a flat output $\overline{z}$ is defined in the `pvtol` module as `_pvtol_flat_forward()`. The inverse map from $\overline{z}$ to $x$ and $u$ and its derivatives is defined as `_pvtol_flat_reverse()`. A `FlatSystem` object can be passed to the `control.flatsys.point_to_point()` function to compute a trajectory that starts at one point and ends at another and satisfies dynamics constraints (this is the primary advantage to finding a flat output for a system). Consider the following:

```
solve_time_start = time.time()
flat_traj = fs.point_to_point(
    pvtol, timepts=t2, x0=x0, u0=ueq, xf=xf, uf=ueq
)
print(f"Flat solve time: {solve_time_end - solve_time_start:.2f} s")
```

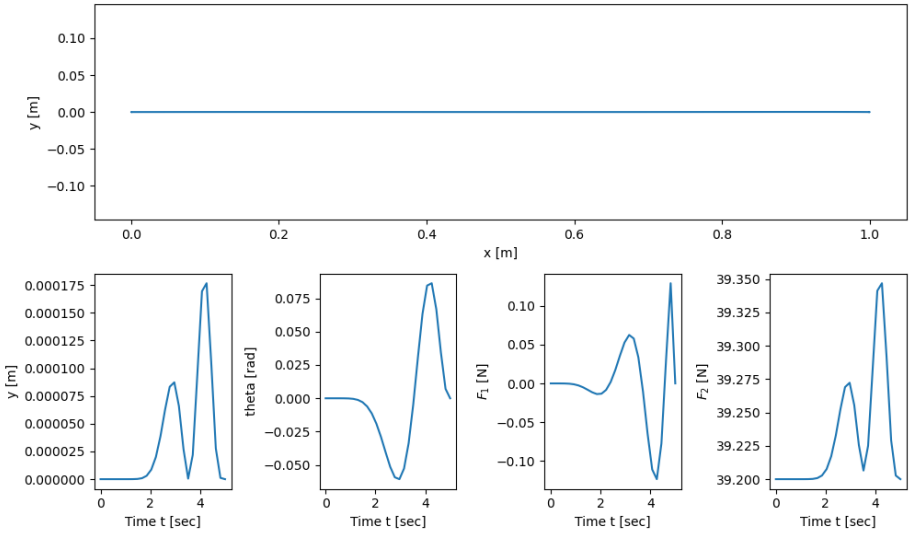
```
Flat solve time: -0.20 s
```

```
/Users/ricopicone/homepage/courses/me595-machine-learning-modern-control/_source/special-sol
  warn("System is only approximately flat (c != 0)")
```

Extract the optimal trajectory and plot the results as follows:

```
flat_traj.states, flat_traj.inputs = flat_traj.eval(t2)
print(f"Initial position: {flat_traj.states[0:2, 0]}")
print(f"Final position: {flat_traj.states[0:2, -1]}")
plot_results(t2, flat_traj.states, flat_traj.inputs)
plt.draw()
```

```
Initial position: [5.30661049e-14 0.00000000e+00]
Final position: [ 1.00000000e+00 -1.47098722e-11]
```




The flatness-based trajectory is smoother and faster to compute than the numerical optimization-based trajectories. The flatness-based trajectory ends essentially exactly at the desired final state.

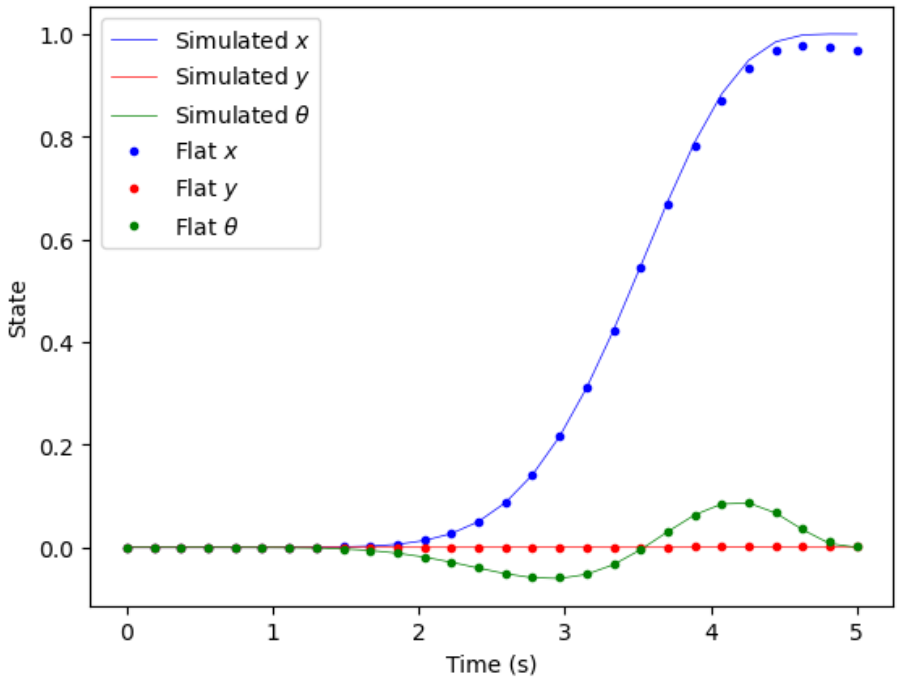
Note that the flatness-based trajectory is not necessarily optimal in the sense of minimizing a cost function. There is an option to add a cost function to the `point_to_point()` function, but I had trouble getting it to yield a good result.

Finally, let's attempt to use the flatness-based trajectory as the feedforward input to a state feedback controller.

```
_, y = control.forced_response(
    pvtol, T=t2, X0=x0, U=flat_traj.inputs, params={"c": 0}
)
```

Plot the position response of the feedforward system as follows:

```
fig, ax = plt.subplots()
ax.plot(t2, flat_traj.states[0], 'b', linewidth=0.5, label='Simulated $x$')
ax.plot(t2, flat_traj.states[1], 'r', linewidth=0.5, label='Simulated $y$')
ax.plot(t2, flat_traj.states[2], 'g', linewidth=0.5, label='Simulated $\\theta$')
ax.plot(t2, y[0], 'b.', label='Flat $x$')
ax.plot(t2, y[1], 'r.', label='Flat $y$')
ax.plot(t2, y[2], 'g.', label='Flat $\\theta$')
ax.set_xlabel('Time (s)')
ax.set_ylabel('State')
ax.legend()
plt.show()
```




The simulated position state and the predicted flat position state trajectories are quite close. With feedback control, the actual position state trajectories would likely be even closer to the predicted flat position state trajectories.