PVTOL Linear Quadratic Regulator Example

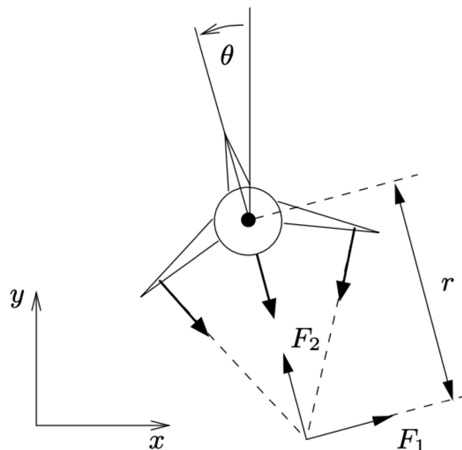
Richard M. Murray, 25 Jan 2022

This notebook contains an example of LQR control applied to the PVTOL system. It demonstrates how to construct an LQR controller and also the importance of the feedforward component of the controller. A gain scheduled design is also demonstrated.

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import control as ct
```

System description

We use the PVTOL dynamics from the textbook, which are contained in the `pvtol` module. The vehicle model is both an I/O system model and a flat system model (for the case when the viscous damping coefficient c is zero).



$$
\begin{aligned}
m\ddot{x} &= F_1 \cos\theta - F_2 \sin\theta - c\dot{x}, \\
m\ddot{y} &= F_1 \sin\theta + F_2 \cos\theta - mg - c\dot{y}, \\
J\ddot{\theta} &= rF_1.
\end{aligned}
$$

The parameter values for the PVTOL system come from the Caltech ducted fan experiment, shown in the video below (the wing forces are not included in the PVTOL model):

In [2]:
```
from IPython.display import YouTubeVideo
display(YouTubeVideo('ZFb5kFpgCm4', width=640, height=480))

from pvtol import pvtol, plot_results
print(pvtol)
```

```
<FlatSystem>: pvtol
Inputs (2): ['F1', 'F2']
Outputs (6): ['x0', 'x1', 'x2', 'x3', 'x4', 'x5']
States (6): ['x0', 'x1', 'x2', 'x3', 'x4', 'x5']

Update: <function _pvtol_update at 0x7fd8d95dcaf0>
Output: <function _pvtol_output at 0x7fd8d95dc8b0>

Forward: <function _pvtol_flat_forward at 0x7fd8c8ea9510>
Reverse: <function _pvtol_flat_reverse at 0x7fd8c8eaa710>
```

Since we will be creating a linear controller, we need a linear system model. We obtain that model by linearizing the dynamics around an equilibrium point. This can be done in python-control using the `find_eqpt` function. We fix the output of the system to be zero and find the state and inputs that hold us there.

In [3]:

```
# Find the equilibrium point corresponding to hover
xeq, ueq = ct.find_eqpt(pvtol, np.zeros(6), np.zeros(2), y0=np.zeros(6), iy=

print("xeq = ", xeq)
print("ueq = ", ueq)

# Get the linearized dynamics
linsys = pvtol.linearize(xeq, ueq)
print(linsys)
```

```
xeq =  [ 0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00 -1.766654e-
27
  0.000000e+00]
ueq =  [ 0.  39.2]
<LinearIOSystem>: sys[2]
Inputs (2): ['u[0]', 'u[1]']
Outputs (6): ['y[0]', 'y[1]', 'y[2]', 'y[3]', 'y[4]', 'y[5]']
States (6): ['x[0]', 'x[1]', 'x[2]', 'x[3]', 'x[4]', 'x[5]']

A = [[ 0.00000000e+00  0.00000000e+00  0.00000000e+00  1.00000000e+00
       0.00000000e+00  0.00000000e+00]
     [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
       1.00000000e+00  0.00000000e+00]
     [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
       0.00000000e+00  1.00000000e+00]
     [ 0.00000000e+00  0.00000000e+00 -9.80000000e+00 -1.25000000e-02
       0.00000000e+00  0.00000000e+00]
     [ 0.00000000e+00  0.00000000e+00 -4.90096852e-06  0.00000000e+00
      -1.25000000e-02  0.00000000e+00]
     [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
       0.00000000e+00  0.00000000e+00]]

B = [[0.         0.        ]
     [0.         0.        ]
     [0.         0.        ]
     [0.25       0.        ]
     [0.         0.25      ]
     [5.26315789 0.        ]]

C = [[1. 0. 0. 0. 0. 0.]
     [0. 1. 0. 0. 0. 0.]
     [0. 0. 1. 0. 0. 0.]
     [0. 0. 0. 1. 0. 0.]
     [0. 0. 0. 0. 1. 0.]
     [0. 0. 0. 0. 0. 1.]]

D = [[0. 0.]
     [0. 0.]
     [0. 0.]
     [0. 0.]
     [0. 0.]
     [0. 0.]]
```

Linear quadratic regulator (LQR) design

Now that we have a linearized model of the system, we can compute a controller using linear quadratic regulator theory. We seek to find the control law that minimizes the function

$$J(x(\cdot), u(\cdot)) = \int_0^\infty x^T(\tau) Q_x x(\tau) + u^T(\tau) Q_u u(\tau) \, d\tau$$

The weighting matrices $Q_x \in \mathbb{R}^{n \times n}$ and $Q_u \in \mathbb{R}^{m \times m}$ should be chosen based on the desired performance of the system (tradeoffs in state errors and input magnitudes). See Example 3.5 in OBC for a discussion of how to choose these weights. For now, we just choose identity weights for all states and inputs.

In [4]:
```
# Start with a diagonal weighting
Qx1 = np.diag([1, 1, 1, 1, 1, 1])
Qu1 = np.diag([1, 1])
K, X, E = ct.lqr(linsys, Qx1, Qu1)
```

To create a controller for the system, we need to create an I/O system that takes in the desired trajectory (x_d, u_d) and the current state x and generates the control law

$$u = u_\text{d} - K(x - x_\text{d})$$

The function `create_statefbk_iosystem()` does this (see documentation for details).

In [5]:
```
control, pvtol_closed = ct.create_statefbk_iosystem(pvtol, K)
print(control, "\n")
print(pvtol_closed)
```

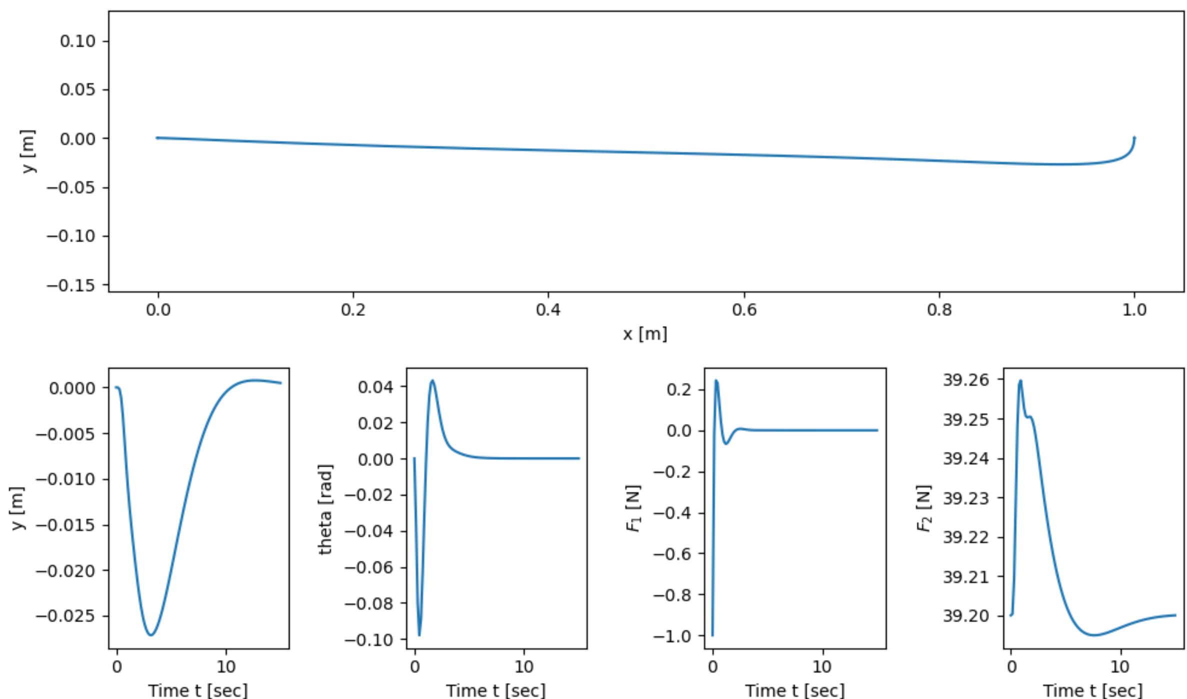
```
<LinearIOSystem>: sys[3]
Inputs (14): ['xd[0]', 'xd[1]', 'xd[2]', 'xd[3]', 'xd[4]', 'xd[5]', 'ud
[0]', 'ud[1]', 'x0', 'x1', 'x2', 'x3', 'x4', 'x5']
Outputs (2): ['F1', 'F2']
States (0): []

A = []

B = []

C = []

D = [[-1.00000000e+00 -4.18744304e-07  7.85405998e+00 -1.60495816e+00
      -5.38237361e-07  2.06849834e+00  1.00000000e+00  0.00000000e+00
       1.00000000e+00  4.18744304e-07 -7.85405998e+00  1.60495816e+00
       5.38237361e-07 -2.06849834e+00]
     [-4.18744302e-07  1.00000000e+00 -2.48287514e-07 -1.36974381e-06
       2.95041664e+00  3.94965562e-08  0.00000000e+00  1.00000000e+00
       4.18744302e-07 -1.00000000e+00  2.48287514e-07  1.36974381e-06
      -2.95041664e+00 -3.94965562e-08]]


<InterconnectedSystem>: F2
Inputs (8): ['xd[0]', 'xd[1]', 'xd[2]', 'xd[3]', 'xd[4]', 'xd[5]', 'ud[0]',
'ud[1]']
Outputs (8): ['x0', 'x1', 'x2', 'x3', 'x4', 'x5', 'F1', 'F2']
States (6): ['pvtol_x0', 'pvtol_x1', 'pvtol_x2', 'pvtol_x3', 'pvtol_x4', 'p
vtol_x5']
```

Closed loop system simulation

We now generate a trajectory for the system and track that trajectory.

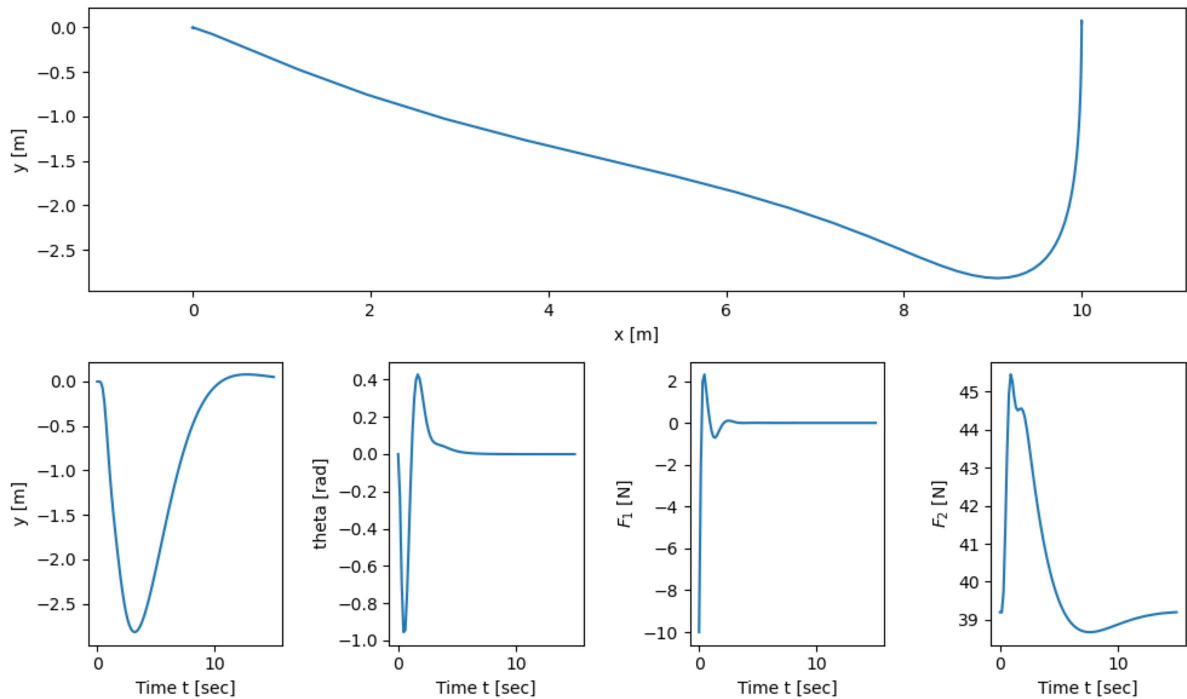
For this simple example, we take the system input to be a "step" input that moves the system 1 meter to the right. More complex trajectories (eg, using the results from HW #3) could also be used.

In [6]:
```
# Generate a step response by setting xd, ud
Tf = 15
T = np.linspace(0, Tf, 100)
xd = np.outer(np.array([1, 0, 0, 0, 0, 0]), np.ones_like(T))
ud = np.outer(ueq, np.ones_like(T))
ref = np.vstack([xd, ud])

response = ct.input_output_response(pvtol_closed, T, ref, xeq)
plot_results(response.time, response.states, response.outputs[6:])
```



The limitations of the linear controlller can be seen if we take a larger step, say 10 meters.

In [7]:
```
xd = np.outer(np.array([10, 0, 0, 0, 0, 0]), np.ones_like(T))
ref = np.vstack([xd, ud])
response = ct.input_output_response(pvtol_closed, T, ref, xeq)
plot_results(response.time, response.states, response.outputs[6:])
```



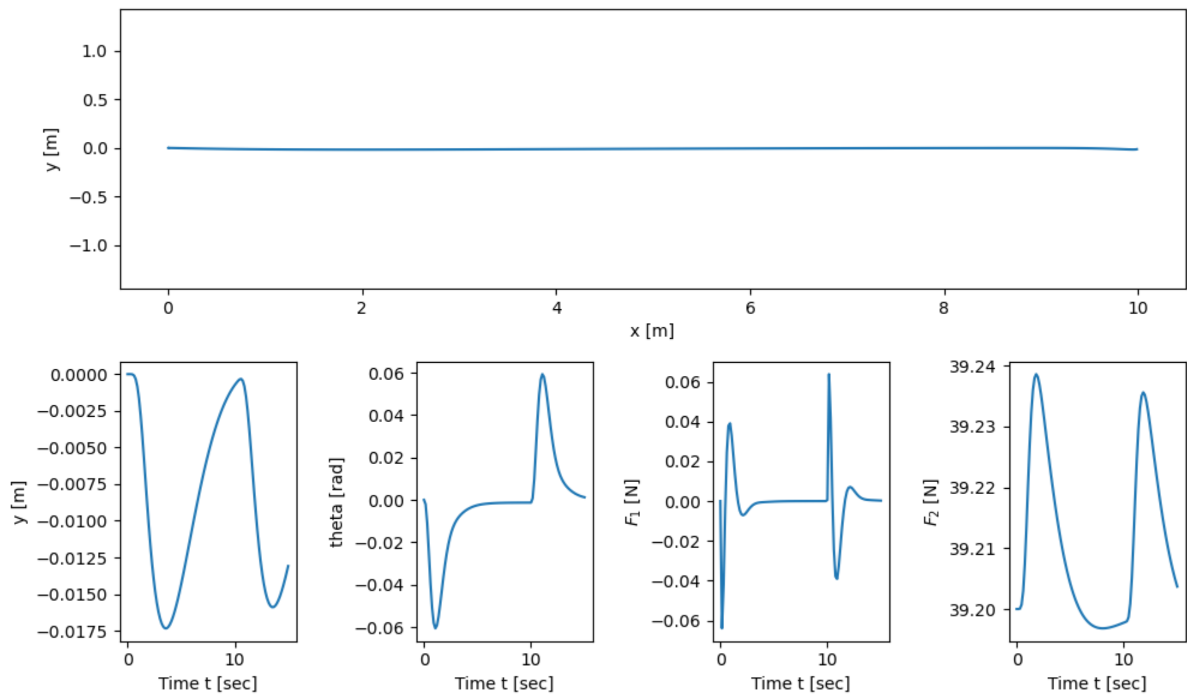
We see that the large initial error causes the vehicle to rotate to a very high role angle (almost 1 radian $\approx 60^\circ$), at which point the linear model is not very accurate and the controller errors in the y direction get very large.

One way to fix this problem is to change the gains on the controller so that we penalize the y error more and try to keep that error from building up. However, given the fact that we are trying to stabilize a point that is fairly far from our initial condition, it can be difficult to manage the tradesoffs to get good performance.

An alterntaive approach is is to stabilize the system around a trajectory that moves from the initial to final condition. As a very simple approach, we start by using a *nonfeasible* trajectory that goes from 0 to 10 in 10 seconds.

In [8]:

```
timepts = np.linspace(0, 15, 100)
xf = np.array([10, 0, 0, 0, 0, 0])
xd = np.array([xf/10 * t if t < 10 else xf for t in timepts]).T
ud = np.outer(ueq, np.ones_like(timepts))
ref = np.vstack([xd, ud])
response = ct.input_output_response(pvtol_closed, timepts, ref, xeq)
plot_results(response.time, response.states, response.outputs[6:])
```



Note that even though the trajectory was not feasible (it asked the system to move sideways while remaining pointed in the vertical ($\theta = 0$) direction, the controller has very good performance.

Gain scheduled controller design

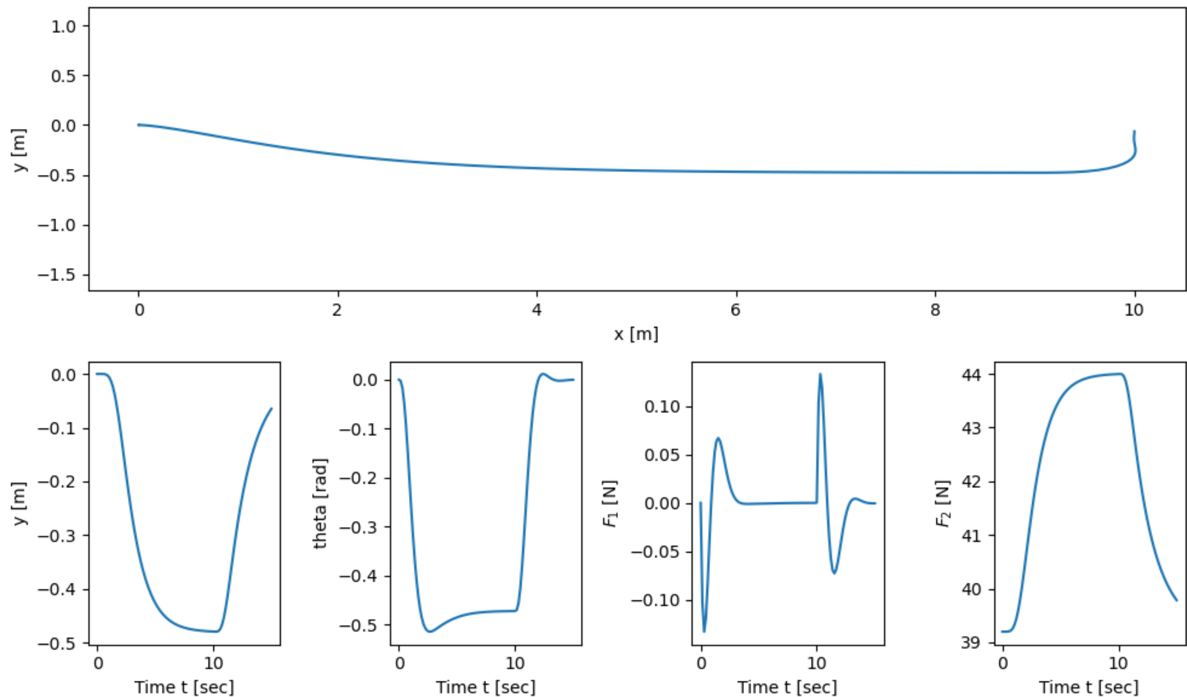
Another challenge in using linearized models is that they are only accurate near the point in which they were computed. For the PVTOL system, this can be a problem if the roll angle θ gets large, since in this case the linearization changes significantly (the forces F_1 and F_2 are no longer aligned with the horizontal and vertical axes).

One approach to solving this problem is to compute different gains at different points in the operating envelope of the system. The code below illustrates the use of gain scheduling by modifying the system drag to a very high value (so that the vehicle must roll to a large angle in order to move sideways against the high drag) and then demonstrates the difficulty in obtaining good performance while trying to track the (still infeasible) trajectory.

In [9]:

```
# Increase the viscous drag to force larger angles
linsys = pvtol.linearize(xeq, ueq, params={'c': 20})

# Change to physically motivated gains
Qx3 = np.diag([10, 100, (180/np.pi) / 5, 0, 0, 0])
Qu3 = np.diag([10, 1])

# Compute a single gain around hover
K, X, E = ct.lqr(linsys, Qx3, Qu3)
control, pvtol_closed = ct.create_statefbk_iosystem(pvtol, K)
```

```
# Simulate the response trying to track horizontal trajectory
response = ct.input_output_response(pvtol_closed, T, ref, xeq, params={'c':
plot_results(response.time, response.states, response.outputs[6:])
```



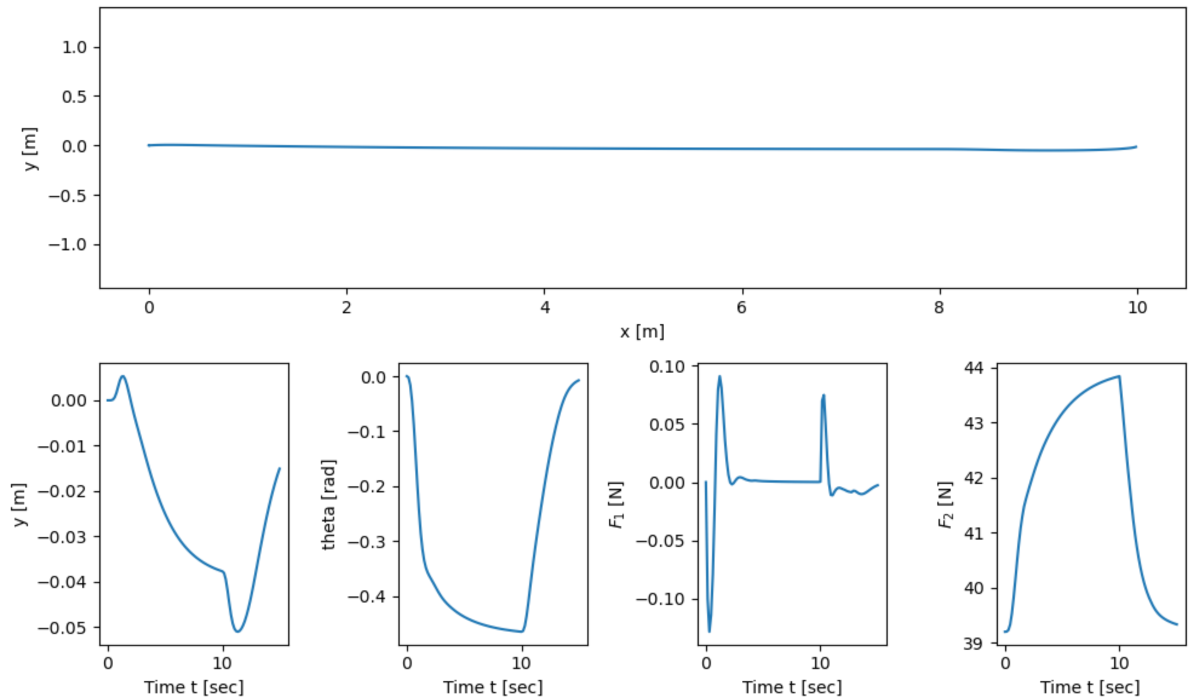
Note that the angle θ is quite large (-0.5 rad) during the initla portion of the trajectory, and at this angle (~30°) it is difficult to maintain our altitude while moving sideways. This happens in large part becuase the system model that we used was linearized about the $\theta = 0$ configuration.

This problem can be addressed by designing a gain scheduled controller in which we compute different system gains at different roll angles. We carry out those computations below, using the `create_statefbk_iosystem` function, but now passing a set of gains and points instead of just a single gain.

(Note: there is a bug in control-0.9.3 that requires gain scheduling to be done on two or more variables, so we also schedule on the horizontal velocity $\dot{x}$, even though that doesn't matter that much here.)

In [10]:
```
import itertools
import math

# Set up points around which to linearize (control-0.9.3: must be 2D or grea
angles = np.linspace(-math.pi/3, math.pi/3, 10)
speeds = np.linspace(-10, 10, 3)
points = list(itertools.product(angles, speeds))

# Compute the gains at each design point
gains = []
for point in points:
    # Compute the state that we want to linearize about
    xgs = xeq.copy()
```

```python
    xgs[2], xgs[3] = point[0], point[1]

    # Linearize the system and compute the LQR gains
    linsys = pvtol.linearize(xgs, ueq, params={'c': 20})
    K, X, E = ct.lqr(linsys, Qx3, Qu3)
    gains.append(K)

# Create a gain scheduled controller off of the current state
control, pvtol_closed = ct.create_statefbk_iosystem(
    pvtol, (gains, points), gainsched_indices=['x2', 'x3'])

# Simulate the response
response = ct.input_output_response(pvtol_closed, T, ref, xeq, params={'c':
plot_results(response.time, response.states, response.outputs[6:])
```



We see that the response is much better, with about 10X less error in the y coordinate.

In [ ]: