L8-3: PVTOL full controller stack

CDS 110/ChE 105, Winter 2024
Richard M. Murray, Manisha Kapasiawala

The purpose of this lecture is to introduce tools that can be used for frequency domain modeling and analysis of linear systems.

In [1]:
```
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt
from math import sin, cos, pi
from scipy.optimize import NonlinearConstraint
import time

try:
  import control as ct
  print("python-control", ct.__version__)
except ImportError:
  !pip install control
  import control as ct

try:
    import pvtol
except ImportError:
    !wget --no-check-certificate https://www.cds.caltech.edu/~murray/courses

from pvtol import plot_results

import control.optimal as opt
import control.flatsys as fs
```

```
python-control 0.10.0
```

System definition

Consider the PVTOL system `pvtol_noisy`, defined in `pvtol.py`:

$$\begin{aligned} m\ddot x &= F_1 \cos\theta - F_2 \sin\theta - c\dot x + D_x, \\ m\ddot y &= F_1 \sin\theta + F_2 \cos\theta - c\dot y - mg + D_y, \\ J\ddot\theta &= rF_1, \end{aligned} \qquad \vec Y = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix} + \begin{bmatrix} N_x \\ N_y \\ N_z \end{bmatrix}.$$

Assume that the input disturbances are modeled by independent, first order Markov (Ornstein-Uhlenbeck) processes with $Q_D = \mathrm{diag}(0.01, 0.01)$ and $\omega_0 = 1$ and that the noise is modeled as white noise with covariance matrix

$$Q_N = \begin{bmatrix} 2\times 10^{-4} & 0 & 1\times 10^{-5} \\ 0 & 2\times 10^{-4} & 1\times 10^{-5} \\ 1\times 10^{-5} & 1\times 10^{-5} & 1\times 10^{-4} \end{bmatrix}.$$

We will design a controller consisting of a trajectory generation module, a gain-scheduled, trajectory tracking module, and a state estimation module the moves the system from the origin to the equilibrum point point x_f, y_f = 10, 0 while satisfying the constraint $0.5 \sin(\pi x / 10) - 0.1 \leq y \leq 1$.

We start by creating the PVTOL system without noise or disturbances.

In [2]:
```
# STANDARD PVTOL DYNAMICS
def _pvtol_update(t, x, u, params):

    # Get the parameter values
    m = params.get('m', 4.)             # mass of aircraft
    J = params.get('J', 0.0475)         # inertia around pitch axis
    r = params.get('r', 0.25)           # distance to center of force
    g = params.get('g', 9.8)            # gravitational constant
    c = params.get('c', 0.05)           # damping factor (estimated)

    # Get the inputs and states
    x, y, theta, xdot, ydot, thetadot = x
    F1, F2 = u

    # Constrain the inputs
    F2 = np.clip(F2, 0, 1.5 * m * g)
    F1 = np.clip(F1, -0.1 * F2, 0.1 * F2)

    # Dynamics
    xddot = (F1 * cos(theta) - F2 * sin(theta) - c * xdot) / m
    yddot = (F1 * sin(theta) + F2 * cos(theta) - m * g - c * ydot) / m
    thddot = (r * F1) / J

    return np.array([xdot, ydot, thetadot, xddot, yddot, thddot])

# Define pvtol output function to only be x, y, and theta
def _pvtol_output(t, x, u, params):
    return x[0:3]

# Create nonlinear input-output system of nominal pvtol system
pvtol_nominal = ct.nlsys(
    _pvtol_update, _pvtol_output, name="pvtol_nominal",
    states = [f'x{i}' for i in range(6)],
    inputs = ['F1', 'F2'],
    outputs = [f'x{i}' for i in range(3)]
)

print(pvtol_nominal)
```

```
<NonlinearIOSystem>: pvtol_nominal
Inputs (2): ['F1', 'F2']
Outputs (3): ['x0', 'x1', 'x2']
States (6): ['x0', 'x1', 'x2', 'x3', 'x4', 'x5']

Update: <function _pvtol_update at 0x1454f7420>
Output: <function _pvtol_output at 0x1454f74c0>
```

Next, we create a PVTOL system with noise and disturbances. This system will use the nominal PVTOL system and add disturbances as inputs to the state dynamics and noise to the system output.

In [3]:

```
# ADD WIND, NOISE
def _noisy_update(t, x, u, params):
    # Get the inputs
    F1, F2, Dx, Dy = u[:4]
    if u.shape[0] > 4:
        Nx, Ny, Nth = u[4:]
    else:
        Nx, Ny, Nth = 0, 0, 0

    # Get the system response from the original dynamics
    xdot, ydot, thetadot, xddot, yddot, thddot = \
        _pvtol_update(t, x, [F1, F2], params)

    # Get the parameter values we need
    m = params.get('m', 4.)             # mass of aircraft
    J = params.get('J', 0.0475)         # inertia around pitch axis

    # Now add the disturbances
    xddot += Dx / m
    yddot += Dy / m

    return np.array([xdot, ydot, thetadot, xddot, yddot, thddot])

# Define pvtol_noisy output function to only be x, y, and theta
def _noisy_output(t, x, u, params):
    F1, F2, Dx, Dy, Nx, Ny, Nth = u
    return x[0:3] + np.array([Nx, Ny, Nth])

# CREATE NONLINEAR INPUT-OUTPUT SYSTEM
pvtol_noisy = ct.nlsys(
    _noisy_update, _noisy_output, name="pvtol_noisy",
    states = [f'x{i}' for i in range(6)],
    inputs = ['F1', 'F2'] + ['Dx', 'Dy'] + ['Nx', 'Ny', 'Nth'],
    outputs = ['x', 'y', 'theta'],
    params = {
        'm': 4.,                # mass of aircraft
        'J': 0.0475,            # inertia around pitch axis
        'r': 0.25,              # distance to center of force
        'g': 9.8,               # gravitational constant
        'c': 0.05,              # damping factor (estimated)
    }
)
```

```
print(pvtol_noisy)
```

```
<NonlinearIOSystem>: pvtol_noisy
Inputs (7): ['F1', 'F2', 'Dx', 'Dy', 'Nx', 'Ny', 'Nth']
Outputs (3): ['x', 'y', 'theta']
States (6): ['x0', 'x1', 'x2', 'x3', 'x4', 'x5']

Update: <function _noisy_update at 0x1454f72e0>
Output: <function _noisy_output at 0x1454f7240>
```

Note that the outputs of `pvtol_noisy` are not the full set of states, but rather the states we can measure: x, y, and θ.

Estimator

We start by designing an optimal estimator for the system. We choose the noise intensities based on knowledge of the modeling errors, disturbances, and sensor characteristics:

In [4]:
```
# Disturbance and noise intensities
Qv = np.diag([1e-2, 1e-2])
Qw = np.array([[2e-4, 0, 1e-5], [0, 2e-4, 1e-5], [1e-5, 1e-5, 1e-4]])
Qwinv = np.linalg.inv(Qw)

# Initial state covariance
P0 = np.eye(pvtol_noisy.nstates)
```

We will use a linear quadratic estimator (Kalman filter) to design an optimal estimator for the system. Recall that the `ct.lqe` function takes in a linear system as input, so we first linear our `pvtol_noisy` system around its equilibrium point.

In [5]:
```
# Find the equiblirum point corresponding to the origin
xe, ue = ct.find_eqpt(
    sys = pvtol_noisy,
    x0 = np.zeros(pvtol_noisy.nstates),
    u0 = np.zeros(pvtol_noisy.ninputs),
    y0 = [0, 0, 0],
    iu=range(2, pvtol_noisy.ninputs),
    iy=[0, 1]
)
print(f"{xe=}")
print(f"{ue=}")

# Linearize system for Kalman filter
pvtol_noisy_lin = pvtol_noisy.linearize(xe, ue)

# Extract the linearization for use in LQR design
A, B, C = pvtol_noisy_lin.A, pvtol_noisy_lin.B, pvtol_noisy_lin.C
```

```
xe=array([0., 0., 0., 0., 0., 0.])
ue=array([ 0. , 39.2,  0. ,  0. ,  0. ,  0. ,  0. ])
```

We want to define an estimator that takes in the measured states x, y, and θ, as well as applied inputs F_1 and F_2. As the estimator doesn't have any measurement of the noise/disturbances applied to the system, we will design our controller with only these inputs.

In [6]:
```
# use ct.lqe to create an L matrix, using only measured inputs F1 and F2
L, Pf, _ = ct.lqe(A, B[:,:2], C, Qv, Qw)
```

We now create our estimator.

In [7]:
```
# Create standard (optimal) estimator update function
def estimator_update(t, xhat, u, params):

    # Extract the inputs to the estimator
    y = u[0:3]                    # just grab the first three outputs
    u_cmd = u[3:5]                # get the inputs that were applied as well

    # Update the state estimate using PVTOL (non-noisy) dynamics
    return _pvtol_update(t, xhat, u_cmd, params) - L @ (C @ xhat - y)

# Create estimator
estimator = ct.nlsys(
    estimator_update, None,
    name = 'Estimator',
    states=pvtol_noisy.nstates,
    inputs= pvtol_noisy.output_labels  \
        + pvtol_noisy.input_labels[0:2],
    outputs=[f'xh{i}' for i in range(pvtol_noisy.nstates)],
)
```

In [8]:
```
print(estimator)
```

```
<NonlinearIOSystem>: Estimator
Inputs (5): ['x', 'y', 'theta', 'F1', 'F2']
Outputs (6): ['xh0', 'xh1', 'xh2', 'xh3', 'xh4', 'xh5']
States (6): ['x[0]', 'x[1]', 'x[2]', 'x[3]', 'x[4]', 'x[5]']

Update: <function estimator_update at 0x1454f79c0>
Output: None
```

Gain scheduled controller

We next design our (gain scheduled) controller for the system. Here, as in the case of the estimator, we will create the controller using the nominal PVTOL system, so that the applied inputs to the system are only F_1 and F_2. If we were to make a controller using the noisy PVTOL system, then the inputs applied via control action would include noise and disturbances, which is incorrect.

In [9]:
```
# Define the weights for the LQR problem
Qx = np.diag([100, 10, (180/np.pi) / 5, 0, 0, 0])
```

```
# Qx = np.diag([10, 100, (180/np.pi) / 5, 0, 0, 0])
Qu = np.diag([10, 1])
```

In [10]:
```
# Construct the array of gains and the gain scheduled controller
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 of angles and speeds
gains = []

# Iterate through points
for point in points:

    # Compute the state that we want to linearize about
    xgs = xe.copy()
    xgs[2], xgs[4] = point[0], point[1]

    # Linearize the system and compute the LQR gains
    linsys = pvtol_noisy.linearize(xgs, ue)
    A = linsys.A
    B = linsys.B[:,:2]
    K, X, E = ct.lqr(A, B, Qx, Qu)
    gains.append(K)

# Construct the controller
gs_ctrl, gs_clsys = ct.create_statefbk_iosystem(
    sys = pvtol_nominal,
    gain = (gains, points),
    gainsched_indices=['xh2', 'xh4'],
    estimator=estimator
)

print(gs_ctrl)
```

```
<NonlinearIOSystem>: sys[31]
Inputs (14): ['xd[0]', 'xd[1]', 'xd[2]', 'xd[3]', 'xd[4]', 'xd[5]', 'ud[0]',
'ud[1]', 'xh0', 'xh1', 'xh2', 'xh3', 'xh4', 'xh5']
Outputs (2): ['F1', 'F2']
States (0): []

Update: <function create_statefbk_iosystem.<locals>._control_update at 0x145
53fce0>
Output: <function create_statefbk_iosystem.<locals>._control_output at 0x145
53fd80>
```

Trajectory generation

Finally, we need to design the trajectory that we want to follow. The specifications that you are given are very similar to those that were used in HW #4, where you designed an

LQR controller for the PVTOL system.

The code below defines the initial conditions, final conditions, and constraints.

In [11]:
```
# Define the initial and final conditions
x_delta = np.array([10, 0, 0, 0, 0, 0])
x0, u0 = ct.find_eqpt(
    sys = pvtol_nominal,
    x0 = np.zeros(6),
    u0 = np.zeros(2),
    y0 = np.zeros(3),
    iy=[0, 1]
)
xf, uf = ct.find_eqpt(
    sys = pvtol_nominal,
    x0 = x0 + x_delta,
    u0 = u0,
    y0 = (x0 + x_delta)[:3],
    iy=[0, 1]
)

# Define the time horizon for the maneuver
Tf = 5
timepts = np.linspace(0, Tf, 100, endpoint=False)

# Create a constraint corresponding to the obstacle
ceiling = (NonlinearConstraint, lambda x, u: x[1], [-1], [1])
nicolas = (NonlinearConstraint,
           lambda x, u: x[1] - (0.5 * sin(pi * x[0] / 10) - 0.1), [0], [1])

# # Reset the nonlinear constraint to give some extra room
# nicolas = (NonlinearConstraint,
#            lambda x, u: x[1] - (0.8 * sin(pi * x[0] / 10) - 0.1), [0], [1]
```

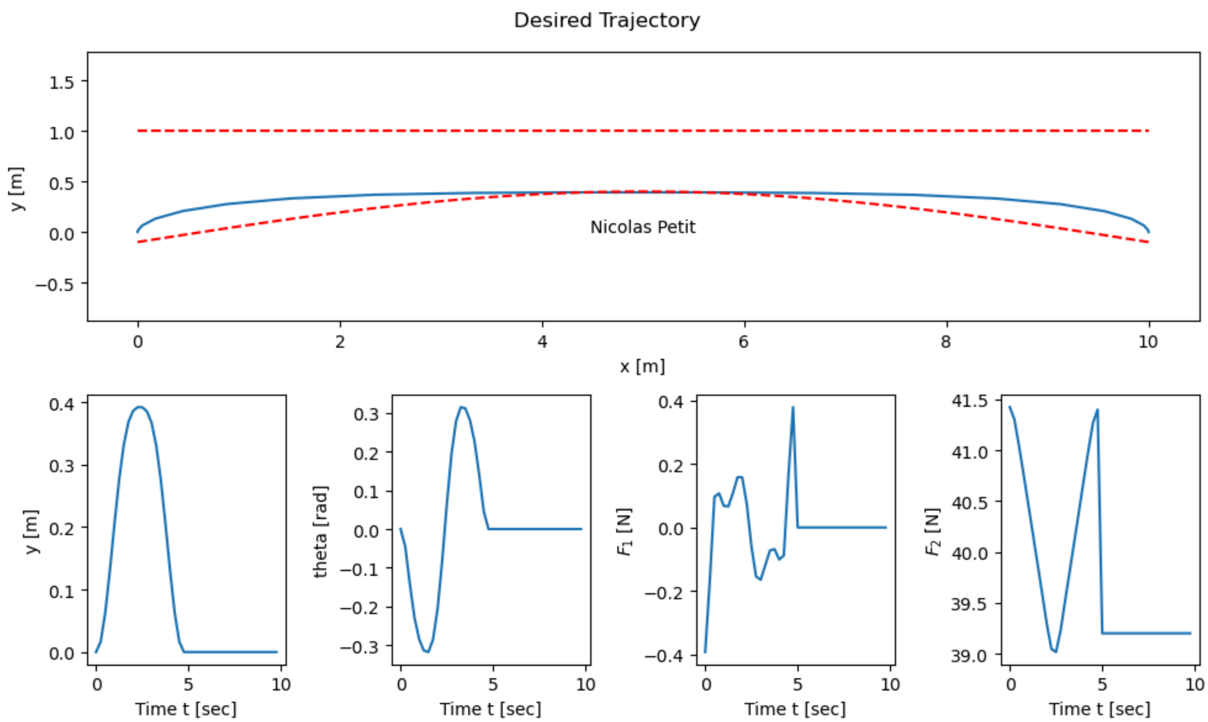
In [12]:
```
# Re-define the time horizon for the maneuver
Tf = 5
timepts = np.linspace(0, Tf, 20, endpoint=False)

# We provide a tent shape as an intial guess
xm = (x0 + xf) / 2 + np.array([0, 0.5, 0, 0, 0, 0])
tm = int(len(timepts)/2)
# straight line from start to midpoint to end with nominal input
tent = (
    np.hstack([
      np.array([x0 + (xm - x0) * t/(Tf/2) for t in timepts[0:tm]]).transpose
      np.array([xm + (xf - xm) * t/(Tf/2) for t in timepts[0:tm]]).transpose
    ]),
    u0
)

# terminal constraint
term_constraints = opt.state_range_constraint(pvtol_nominal, xf, xf)

# trajectory cost
traj_cost = opt.quadratic_cost(pvtol_nominal, None, Qu, x0=xf, u0=uf)
```

```python
# find optimal trajectory
start_time = time.process_time()
traj = opt.solve_ocp(
    sys = pvtol_nominal,
    timepts = timepts,
    initial_guess=tent,
    X0=x0,
    cost = traj_cost,
    trajectory_constraints=[ceiling, nicolas],
    terminal_constraints=term_constraints,
)
print("* Total time = %5g seconds\n" % (time.process_time() - start_time))

# Create the desired trajectory
xd, ud = traj.states, traj.inputs
```

```
Summary statistics:
* Cost function calls: 2908
* Constraint calls: 3088
* Eqconst calls: 3088
* System simulations: 0
* Final cost: 8.739277745398129
* Total time = 1.66713 seconds
```

In [13]:

```python
# Extend the trajectory to hold the final position for Tf seconds
holdpts = np.arange(Tf, Tf + Tf, timepts[1]-timepts[0])
xd = np.hstack([xd, np.outer(xf, np.ones_like(holdpts))])
ud = np.hstack([ud, np.outer(uf, np.ones_like(holdpts))])
timepts = np.hstack([timepts, holdpts])

# Plot the desired trajectory
plot_results(timepts, xd, ud)
plt.suptitle('Desired Trajectory')

# Add the constraints to the plot
plt.subplot(2, 1, 1)
plt.plot([0, 10], [1, 1], 'r--')
x_nic = np.linspace(0, 10, 50)
y_nic = 0.5 * np.sin(pi * x_nic / 10) - 0.1
plt.plot(x_nic, y_nic, 'r--')
plt.text(5, 0, 'Nicolas Petit', ha='center')
plt.tight_layout()
```



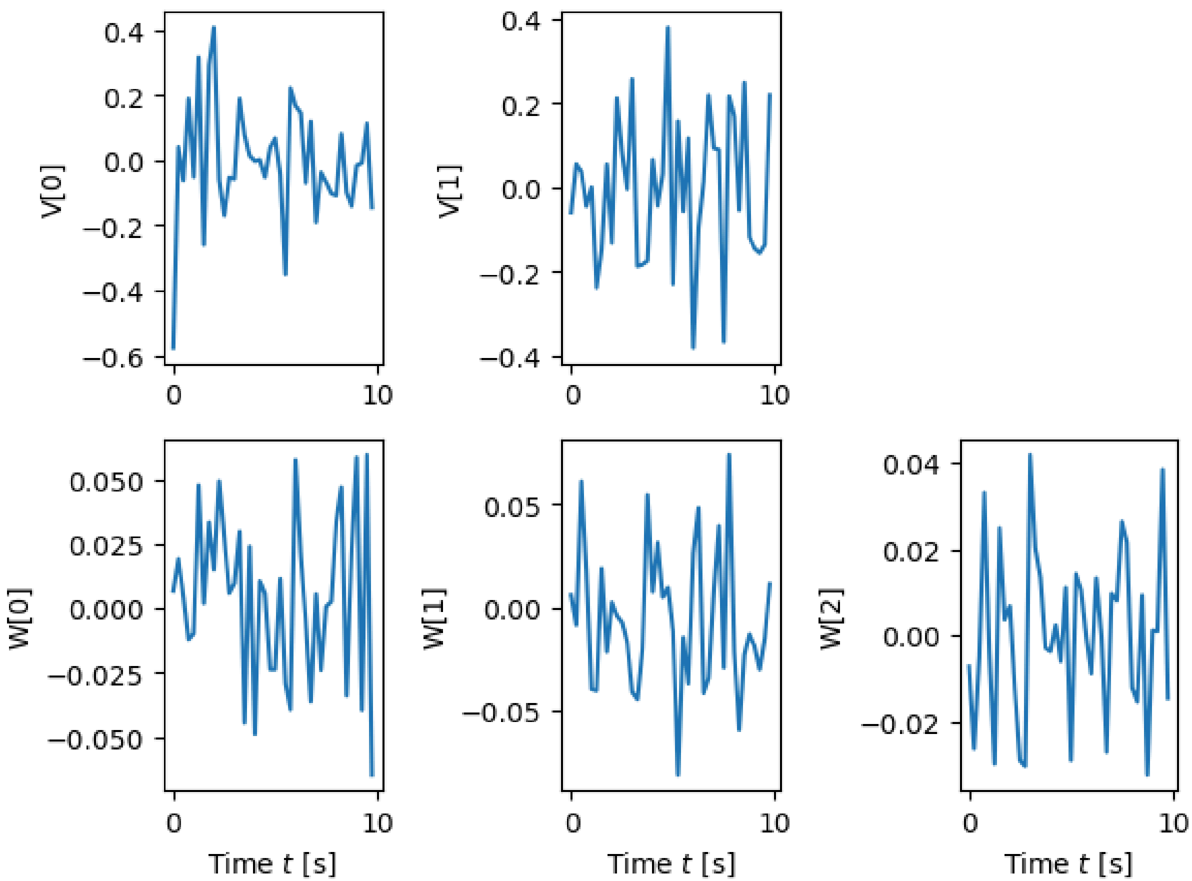
Final Control System Implementation

We now put together the final control system and simulate it. If you have named your inputs and outputs to each of the subsystems properly, the code below should connect everything up correctly. If you get errors about inputs or outputs that are not connected to anything, check the names of your inputs and outputs in the various systems above and make sure everything lines up as it should.

In [14]:
```
# Create the interconnected system
clsys = ct.interconnect(
    [pvtol_noisy, gs_ctrl, estimator],
    inputs=gs_clsys.input_labels[:8] + pvtol_noisy.input_labels[2:],
    outputs=pvtol_noisy.output_labels + pvtol_noisy.input_labels[:2]
)
print(clsys)
```

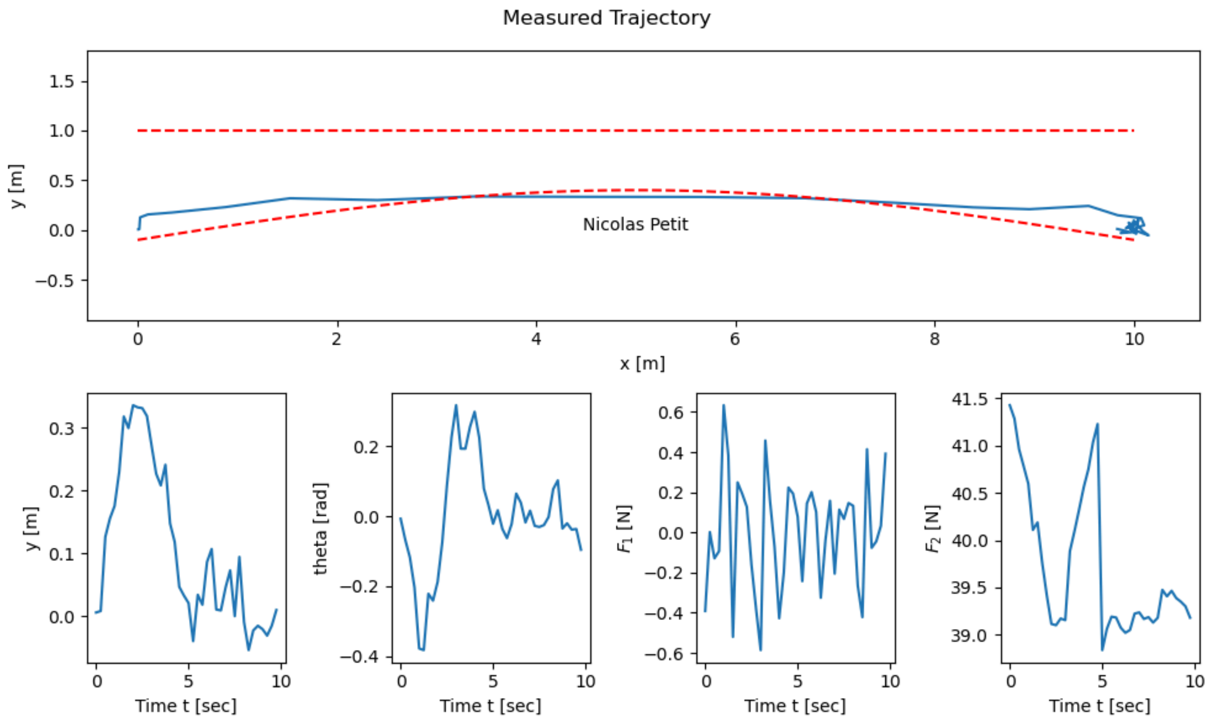
```
<InterconnectedSystem>: sys[32]
Inputs (13): ['xd[0]', 'xd[1]', 'xd[2]', 'xd[3]', 'xd[4]', 'xd[5]', 'ud[0]',
'ud[1]', 'Dx', 'Dy', 'Nx', 'Ny', 'Nth']
Outputs (5): ['x', 'y', 'theta', 'F1', 'F2']
States (12): ['pvtol_noisy_x0', 'pvtol_noisy_x1', 'pvtol_noisy_x2', 'pvtol_n
oisy_x3', 'pvtol_noisy_x4', 'pvtol_noisy_x5', 'Estimator_x[0]', 'Estimator_x
[1]', 'Estimator_x[2]', 'Estimator_x[3]', 'Estimator_x[4]', 'Estimator_x
[5]']

Update: <function InterconnectedSystem.__init__.<locals>.updfcn at 0x1457e31
a0>
Output: <function InterconnectedSystem.__init__.<locals>.outfcn at 0x145bb2a
c0>
```

In [15]:
```
# Generate disturbance and noise vectors
V = ct.white_noise(timepts, Qv)
W = ct.white_noise(timepts, Qw)
for i in range(V.shape[0]):
    plt.subplot(2, 3, i+1)
    plt.plot(timepts, V[i])
    plt.ylabel(f'V[{i}]')

for i in range(W.shape[0]):
    plt.subplot(2, 3, i+4)
    plt.plot(timepts, W[i])
    plt.ylabel(f'W[{i}]')
    plt.xlabel('Time $t$ [s]')

plt.tight_layout()
```



In [16]:
```
# Simulate the open loop system and plot the results (+ state trajectory)
resp = ct.input_output_response(
    sys = clsys,
    T = timepts,
    U = [xd, ud, V, W],
    X0 = np.zeros(12))

plot_results(resp.time, resp.outputs[0:3], resp.outputs[3:5])

# Add the constraints to the plot
plt.subplot(2, 1, 1)
plt.plot([0, 10], [1, 1], 'r--')
```

```
x_nic = np.linspace(0, 10, 50)
y_nic = 0.5 * np.sin(pi * x_nic / 10) - 0.1
plt.plot(x_nic, y_nic, 'r--')
plt.text(5, 0, 'Nicolas Petit', ha='center')
plt.suptitle("Measured Trajectory")
plt.tight_layout()
```



Comment on the design here. If you were Nicolas Petit, would you be OK with the controller that you designed? If not, try to modify the controller to make Nicolas less nervous and explain what you modified and why.

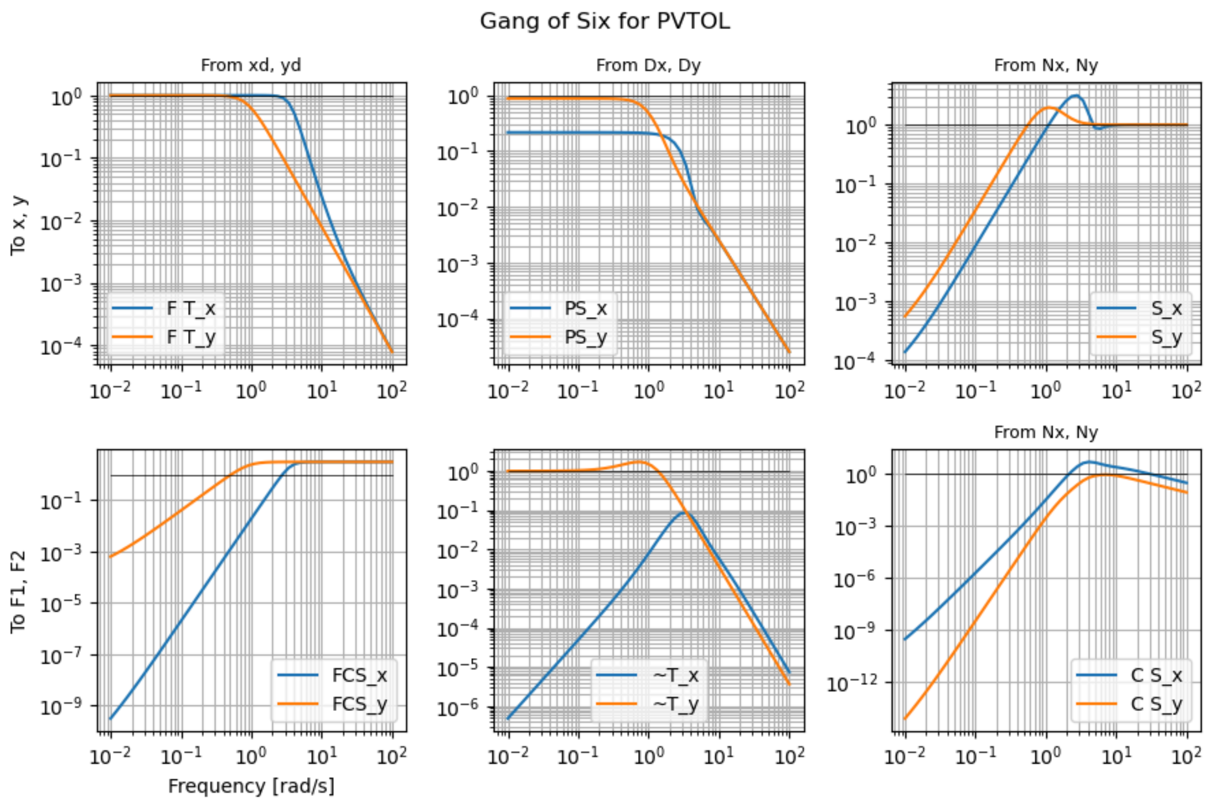
Small signal analysis

In [17]:
```
## Small signal analysis
X0 = np.hstack([x0, x0])                 # system state, estim state
U0 = np.hstack([x0, u0, np.zeros(5)])    # xd, ud, dist, noise
G = clsys.linearize(X0, U0)
print(clsys)

# Get input/output dictionaries: inp['sig'] = index for 'sig'
inp = clsys.input_index
out = clsys.output_index

fig, axs = plt.subplots(2, 3, figsize=[9, 6])
omega = np.logspace(-2, 2)

# Complementary sensitivity
G_x_xd = ct.tf(G[out['x'], inp['xd[0]']])
G_y_yd = ct.tf(G[out['y'], inp['xd[1]']])
ct.bode_plot(
    [G_x_xd, G_y_yd], omega,
```

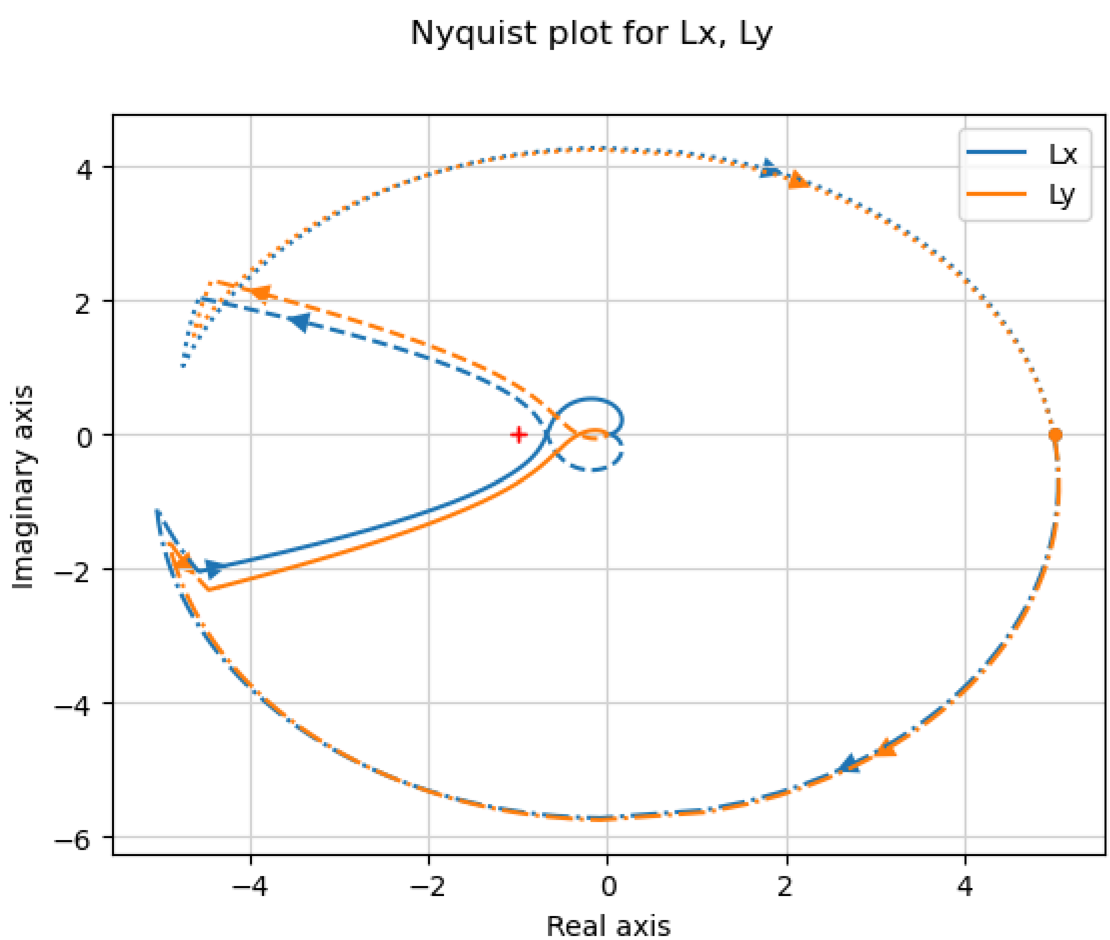
```python
    plot_phase=False, ax=np.array([[axs[0, 0]]]))
axs[0, 0].legend(['F T_x', 'F T_y'])
axs[0, 0].loglog([omega[0], omega[-1]], [1, 1], 'k', linewidth=0.5)
axs[0, 0].set_title("From xd, yd", fontsize=9)
axs[0, 0].set_ylabel("To x, y")
axs[0, 0].set_xlabel("")

# Load (or input) sensitivity
G_x_dx = ct.tf(G[out['x'], inp['Dx']])
G_y_dy = ct.tf(G[out['y'], inp['Dy']])
ct.bode_plot(
    [G_x_dx, G_y_dy], omega,
    plot_phase=False, ax=np.array([[axs[0, 1]]]))
axs[0, 1].legend(['PS_x', 'PS_y'])
axs[0, 1].loglog([omega[0], omega[-1]], [1, 1], 'k', linewidth=0.5)
axs[0, 1].set_title("From Dx, Dy", fontsize=9)
axs[0, 1].set_xlabel("")
axs[0, 1].set_ylabel("")

# Sensitivity
G_x_Nx = ct.tf(G[out['x'], inp['Nx']])
G_y_Ny = ct.tf(G[out['y'], inp['Ny']])
ct.bode_plot(
    [G_x_Nx, G_y_Ny], omega,
    plot_phase=False, ax=np.array([[axs[0, 2]]]))
axs[0, 2].legend(['S_x', 'S_y'])
axs[0, 2].set_title("From Nx, Ny", fontsize=9)
axs[0, 2].loglog([omega[0], omega[-1]], [1, 1], 'k', linewidth=0.5)
axs[0, 2].set_xlabel("")
axs[0, 2].set_ylabel("")

# Noise (or output) sensitivity
G_F1_xd = ct.tf(G[out['F1'], inp['xd[0]']])
G_F2_yd = ct.tf(G[out['F2'], inp['xd[1]']])
ct.bode_plot(
    [G_F1_xd, G_F2_yd], omega,
    plot_phase=False, ax=np.array([[axs[1, 0]]]))
axs[1, 0].legend(['FCS_x', 'FCS_y'])
axs[1, 0].loglog([omega[0], omega[-1]], [1, 1], 'k', linewidth=0.5)
axs[1, 0].set_ylabel("To F1, F2")

G_F1_dx = ct.tf(G[out['F1'], inp['Dx']])
G_F2_dy = ct.tf(G[out['F2'], inp['Dy']])
ct.bode_plot(
    [G_F1_dx, G_F2_dy], omega,
    plot_phase=False, ax=np.array([[axs[1, 1]]]))
axs[1, 1].legend(['~T_x', '~T_y'])
axs[1, 1].loglog([omega[0], omega[-1]], [1, 1], 'k', linewidth=0.5)
axs[1, 1].set_xlabel("")
axs[1, 1].set_ylabel("")

# Sensitivity
G_F1_Nx = ct.tf(G[out['F1'], inp['Nx']])
G_F1_Ny = ct.tf(G[out['F1'], inp['Ny']])
ct.bode_plot(
    [G_F1_Nx, G_F1_Ny], omega,
```

```
    plot_phase=False, ax=np.array([[axs[1, 2]]]))
axs[1, 2].legend(['C S_x', 'C S_y'])
axs[1, 2].set_title("From Nx, Ny", fontsize=9)
axs[1, 2].loglog([omega[0], omega[-1]], [1, 1], 'k', linewidth=0.5)
axs[1, 2].set_xlabel("")
axs[1, 2].set_ylabel("")

plt.suptitle("Gang of Six for PVTOL")
plt.tight_layout()
```

```
<InterconnectedSystem>: sys[32]
Inputs (13): ['xd[0]', 'xd[1]', 'xd[2]', 'xd[3]', 'xd[4]', 'xd[5]', 'ud[0]',
'ud[1]', 'Dx', 'Dy', 'Nx', 'Ny', 'Nth']
Outputs (5): ['x', 'y', 'theta', 'F1', 'F2']
States (12): ['pvtol_noisy_x0', 'pvtol_noisy_x1', 'pvtol_noisy_x2', 'pvtol_n
oisy_x3', 'pvtol_noisy_x4', 'pvtol_noisy_x5', 'Estimator_x[0]', 'Estimator_x
[1]', 'Estimator_x[2]', 'Estimator_x[3]', 'Estimator_x[4]', 'Estimator_x
[5]']

Update: <function InterconnectedSystem.__init__.<locals>.updfcn at 0x1457e31
a0>
Output: <function InterconnectedSystem.__init__.<locals>.outfcn at 0x145bb2a
c0>
```



In [18]:
```
# Solve for the loop transfer function horizontal direction
# S = 1 / (1 + L) => S + SL = 1 => L = (1 - S)/S
Lx = (1 - G_x_Nx) / G_x_Nx; Lx.name = 'Lx'
Ly = (1 - G_y_Ny) / G_y_Ny; Ly.name = 'Ly'

# Create Nyquist plot
ct.nyquist_plot([Lx, Ly], max_curve_magnitude=5, max_curve_offset=0.2);
```



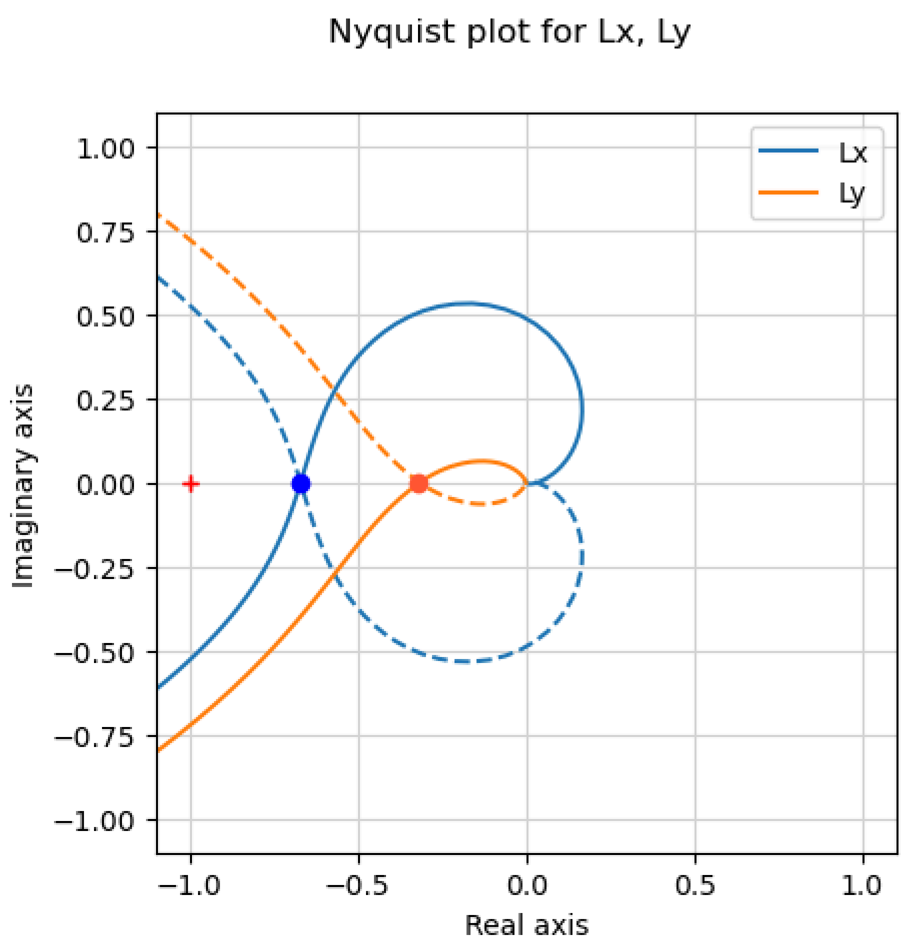
We can zoom in on the plot to see the gain, phase, and stability margins.

Gain Margins of L_x, L_y

In [19]:
```
# add customizations to Nyquist plot
ct.nyquist_plot([Lx, Ly])
lower_upper_bound = 1.1
plt.axis([-lower_upper_bound,
          lower_upper_bound,
          -lower_upper_bound,
          lower_upper_bound])
ax = plt.gca()
ax.set_aspect('equal')

# Gain margin for Lx
neg1overgm_x = -0.67 #vary this manually to find intersection with curve
plt.plot(neg1overgm_x, 0, color='b', marker='o')
gm_x = -1/neg1overgm_x

# Gain margin for Ly
neg1overgm_y = -0.32 #vary this manually to find intersection with curve
plt.plot(neg1overgm_y, 0, color='#FF5733', marker='o')
gm_y = -1/neg1overgm_y

print('Margins obtained visually:')
```

```
print('Gain margin of Lx: '+str(gm_x))
print('Gain margin of Ly: '+str(gm_y))
print('\n')

# get gain margin computationally
gm_xc, pm_xc, wpc_xc, wgc_xc = ct.margin(Lx)
gm_yc, pm_yc, wpc_yc, wgc_yc = ct.margin(Ly)

print('Margins obtained computationally:')
print('Gain margin of Lx: '+str(gm_xc))
print('Gain margin of Ly: '+str(gm_yc))

print('\n')
```

```
Margins obtained visually:
Gain margin of Lx: 1.4925373134328357
Gain margin of Ly: 3.125


Margins obtained computationally:
Gain margin of Lx: 1.4950420635127128
Gain margin of Ly: 3.1807801212975133
```

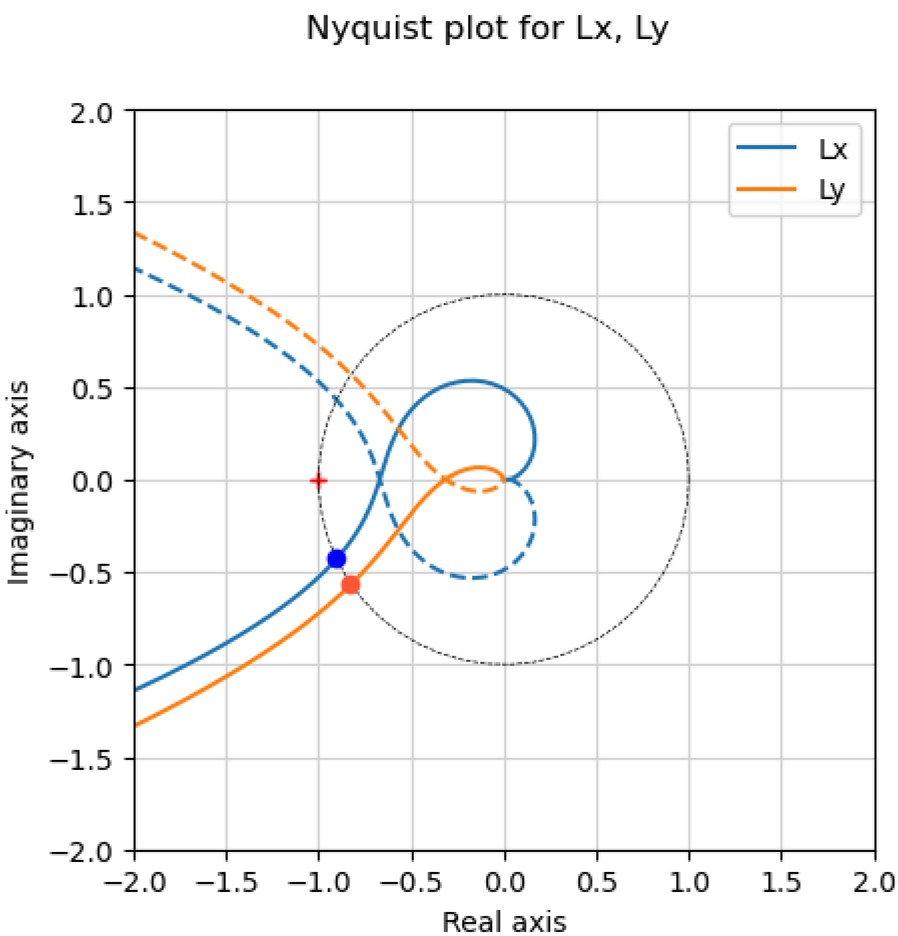


Phase Margins of L_x, L_y

In [20]:

```
# add customizations to Nyquist plot
ct.nyquist_plot([Lx, Ly], max_curve_magnitude=5, max_curve_offset=0.2)
lower_upper_bound = 2
plt.axis([-lower_upper_bound,
          lower_upper_bound,
          -lower_upper_bound,
          lower_upper_bound])
ax = plt.gca()
ax.set_aspect('equal')

# plot circle
theta = np.linspace(0, 2 * pi)
plt.plot(np.cos(theta), np.sin(theta), 'k--', linewidth=0.5)

# Phase margin of Lx:
th_pm_x = 0.14*np.pi
th_plt_x = np.pi + th_pm_x
plt.plot(np.cos(th_plt_x), np.sin(th_plt_x), color='blue', marker='o')

# Phase margin of Ly
th_pm_y = 0.19*np.pi
th_plt_y = np.pi + th_pm_y
plt.plot(np.cos(th_plt_y), np.sin(th_plt_y), color='#FF5733', marker='o')

print('Margins obtained visually:')
print('Phase margin: '+str(float(th_pm_x)))
print('Phase margin: '+str(float(th_pm_y)))
print('\n')

# get margin computationally
gm_xc, pm_xc, wpc_xc, wgc_xc = ct.margin(Lx)
gm_yc, pm_yc, wpc_yc, wgc_yc = ct.margin(Ly)

print('Margins obtained computationally:')
print('Phase margin of Lx: '+str(np.deg2rad(pm_xc)))
print('Phase margin of Ly: '+str(np.deg2rad(pm_yc)))

print('\n')
```

```
Margins obtained visually:
Phase margin: 0.4398229715025711
Phase margin: 0.5969026041820606


Margins obtained computationally:
Phase margin of Lx: 0.443697944544791
Phase margin of Ly: 0.6021311950492322
```



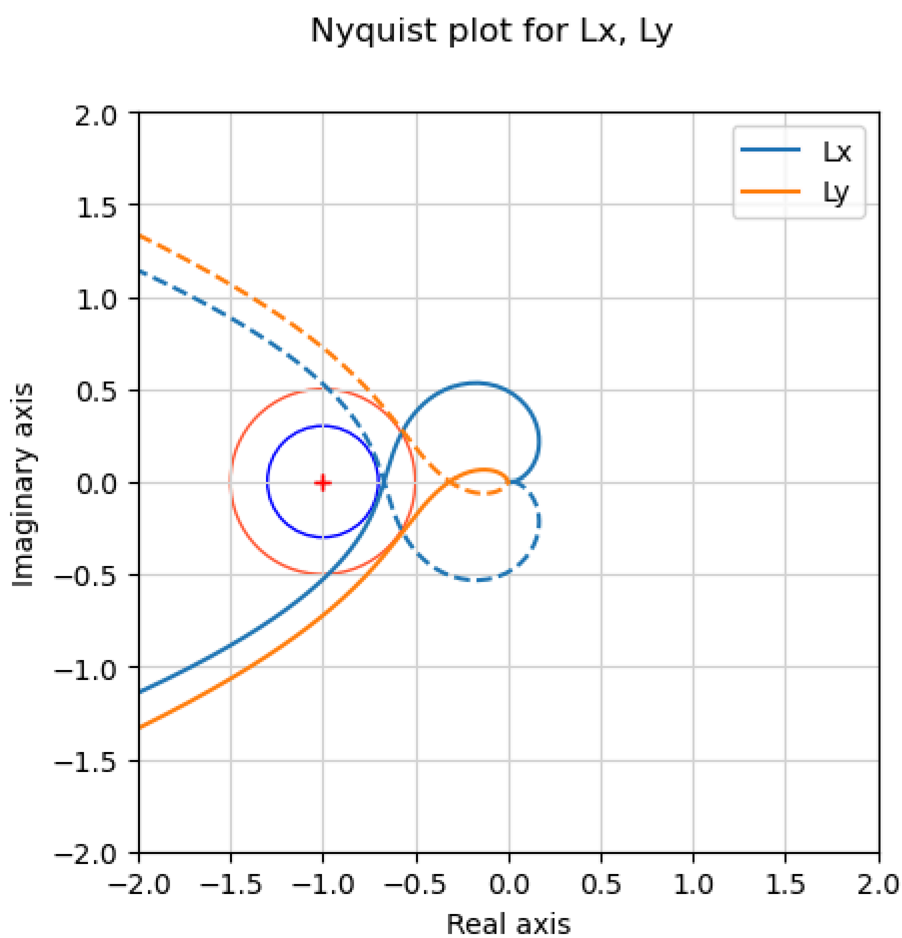
Stability Margins of L_x, L_y

In [21]:

```
# add customizations to Nyquist plot
ct.nyquist_plot([Lx, Ly], max_curve_magnitude=5, max_curve_offset=0.2)
lower_upper_bound = 2
plt.axis([-lower_upper_bound,
          lower_upper_bound,
          -lower_upper_bound,
          lower_upper_bound])
ax = plt.gca()
ax.set_aspect('equal')

# Stability margin:
sm_x = 0.3 #vary this manually to find min which intersects
sm_circle = plt.Circle((-1, 0), sm_x, color='blue', fill=False, ls='-')
ax.add_patch(sm_circle)
sm_y = 0.5 #vary this manually to find min which intersects
sm_circle = plt.Circle((-1, 0), sm_y, color='#FF5733', fill=False, ls='-')
ax.add_patch(sm_circle)

print('Margins obtained visually:')
print('Stability margin of Lx: '+str(sm_x))
print('Stability margin of Ly: '+str(sm_y))

# Compute the stability margin computationally
```

```
print('\n')
print('Margins obtained computationally:')
resp = ct.frequency_response(1 + Lx)
sm = np.min(resp.magnitude)
wsm = resp.omega[np.argmin(resp.magnitude)]
print(f"Stability margin of Lx = {sm:2.2g} (at {wsm:.2g} rad/s)")
resp = ct.frequency_response(1 + Ly)
sm = np.min(resp.magnitude)
wsm = resp.omega[np.argmin(resp.magnitude)]
print(f"Stability margin of Ly = {sm:2.2g} (at {wsm:.2g} rad/s)")
print('\n')
```

```
Margins obtained visually:
Stability margin of Lx: 0.3
Stability margin of Ly: 0.5


Margins obtained computationally:
Stability margin of Lx = 0.31 (at 2.7 rad/s)
Stability margin of Ly = 0.51 (at 1.1 rad/s)
```



We see that the frequencies at which the stability margins are found corresponds to the peak of the magnitude of the sensitivity functions for L_x and L_y.

In [22]:
```
# Confirm stability using Nyquist criterion
nyqresp_x = ct.nyquist_response(Lx)
nyqresp_y = ct.nyquist_response(Ly)

print("Nx =", nyqresp_x.count, "; Px =", np.sum(np.real(Lx.poles()) > 0))
print("Ny =", nyqresp_y.count, "; Py =", np.sum(np.real(Ly.poles()) > 0))
```

```
Nx = 0 ; Px = 0
Ny = 0 ; Py = 0
```

In [23]:
```
# Take a look at the locatoins of the poles
np.real(Ly.poles())
```

Out[23]:
```
array([-5.12875275e+00, -5.12875275e+00, -5.12874173e+00, -5.12874173e+00,
       -1.48993267e+00, -1.48993267e+00, -1.48990844e+00, -1.48990844e+00,
       -3.24406273e+00, -3.24406273e+00, -3.24407516e+00, -3.24407516e+00,
       -1.83246455e+00, -1.83246455e+00, -1.83162028e+00, -1.83162028e+00,
       -1.56114778e+00, -1.56114778e+00, -9.41479353e-01, -9.41479353e-01,
       -6.25063929e-01, -6.25063929e-01, -1.25000000e-02, -4.01830586e-15])
```

In [24]:
```
# See what happened in the contour
plt.plot(np.real(nyqresp_y.contour), np.imag(nyqresp_y.contour))
plt.axis([-1e-4, 4e-4, 0, 4e-4])
plt.title("Zoom on D-contour");
```