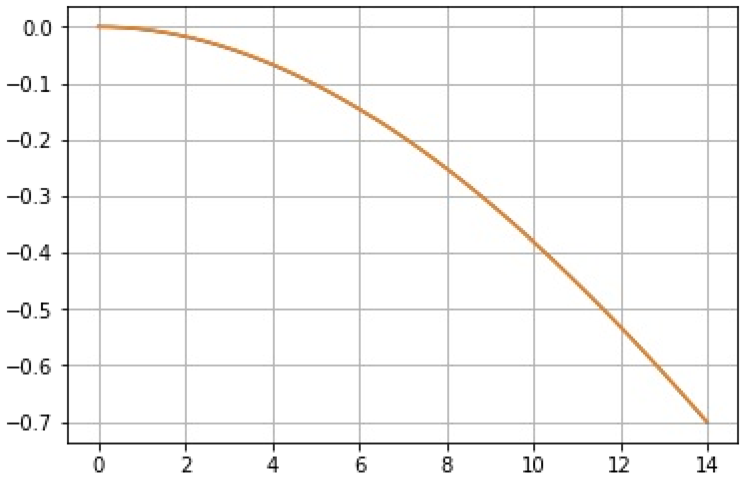
```
Jupyter QtConsole 4.3.1
Python 3.7.2 (default, Jan  3 2019, 02:55:40)
Type "copyright", "credits" or "license" for more information.

IPython 5.8.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

Imported:
numpy as np
scipy as sp
matplotlib.pyplot as plt
supsictrl.ctrl_utils as ctut
control as ct
myEnv

In [1]: from sympy import symbols, Matrix, pi
   ...: from sympy.physics.mechanics import *
   ...: import numpy as np
   ...: from scipy.optimize import leastsq
   ...: import scipy as sp
   ...: import matplotlib.pyplot as plt
   ...: from control import *
   ...: import control.matlab as mt
   ...: from supsictrl.ctrl_utils import *
   ...: import supsictrl.ctrl_repl as rp
   ...:

In [2]: ########### System and Controller choice ################
   ...: # Choose Controller
   ...: # 1 - State feedback
   ...: # 2 - LQR controller
   ...: Controller = 2
   ...:
   ...: # Choose Observer:
   ...: # 1: Reduced order observer
   ...: # 2: Full order observer
   ...: Observer = 1
   ...:
   ...: # Choose Ball
   ...: # 1: Blue Ball
   ...: # 2: Green Ball
   ...: Ball = 1
   ...:
   ...: wn = 1             # Bandwidth for State feedback controller
   ...: obs_k = 8          # Distance factor for observer poles to closed loop
poles
   ...:
   ...: ##############################################
   ...:

In [3]: # Kane's Model of the system
   ...: # Index _b: angle between Wheel center and Ball CM
```

```
   ...: # Index _w: Wheel
   ...: # Index _roll: Ball
   ...:
   ...: # Dynamic symbols
   ...: phi_b, phi_w, phi_roll = dynamicsymbols('phi_b phi_w phi_roll')
   ...: w_b, w_w, w_roll = dynamicsymbols('w_b w_w w_roll')
   ...:

In [4]: T = dynamicsymbols('T')

In [5]: # Symbols
   ...: J_w, J_b = symbols('J_w J_b')
   ...: M_w, M_b = symbols('M_w M_b')
   ...: R_w, R_b = symbols('R_w R_b')
   ...: d_w      = symbols('d_w')
   ...: g       = symbols('g')
   ...: t       = symbols('t')
   ...:

In [6]: # Mechanical system
   ...: N = ReferenceFrame('N')
   ...:
   ...: O = Point('O')
   ...: O.set_vel(N,0)
   ...:

In [7]: # Roll conditions
   ...: phi_roll = -(phi_w*R_w-phi_b*R_w)/R_b
   ...: w_roll = phi_roll.diff(t)
   ...:

In [8]: # Rotating axes
   ...: # Ball rotation
   ...: # Wheel rotation
   ...: # Ball position
   ...: N_b = N.orientnew('N_b','Axis',[phi_b,N.y])
   ...: N_w = N.orientnew('N_w','Axis',[phi_w,N.y])
   ...: N_roll = N.orientnew('N_roll','Axis',[phi_roll,N.y])
   ...:
   ...: N_w.set_ang_vel(N,w_w*N.y)
   ...: N_roll.set_ang_vel(N,w_roll*N.y)
   ...: N_b.set_ang_vel(N, w_b*N.y)
   ...:

In [9]: # Ball Center of mass
   ...: CM2 = O.locatenew('CM2',(R_w+R_b)*N_b.z)
   ...: CM2.v2pt_theory(O,N,N_b)
   ...:
Out[9]: (R_b + R_w)*w_b*N_b.x

In [10]: # Inertia
    ...: Iy = outer(N.y,N.y)
    ...: In1T = (J_w*Iy, O)
    ...: In2T = (J_b*Iy, CM2)
    ...:
```

```
In [11]: # Bodies
    ...: B_w = RigidBody('B_w', O, N_w, M_w, In1T)
    ...: B_r = RigidBody('B_r', CM2, N_roll, M_b, In2T)
    ...:

In [12]: B_w.kinetic_energy(N)
Out[12]: J_w*w_w(t)**2/2

In [13]: B_r.kinetic_energy(N)
Out[13]: J_b*(R_w*Derivative(phi_b(t), t) - R_w*Derivative(phi_w(t), t))**2/
(2*R_b**2) + M_b*(R_b + R_w)**2*w_b(t)**2/2

In [14]: # Forces
    ...: forces = [(CM2,-M_b*g*N.z), (N_w, T*N.y) ]

In [15]: # Relations between position and speed
    ...: kindiffs = [phi_w.diff(t)-w_w,phi_b.diff(t)-w_b]

In [16]: # Identification with Kane's method
    ...: KM = KanesMethod(N,q_ind=[phi_b, phi_w],u_ind=[w_b,
w_w],kd_eqs=kindiffs)
    ...: fr, frstar = KM.kanes_equations([B_r, B_w], forces)
    ...:

In [17]: fr
Out[17]:
Matrix([
[M_b*g*(R_b + R_w)*sin(phi_b(t))],
[                           T(t)]])

In [18]: frstar
Out[18]:
Matrix([
[J_b*R_w**2*Derivative(w_w(t), t)/R_b**2 - (J_b*R_w**2/R_b**2 + M_b*(R_b +
R_w)**2)*Derivative(w_b(t), t)],
[               J_b*R_w**2*Derivative(w_b(t), t)/R_b**2 - (J_b*R_w**2/R_b**2 +
J_w)*Derivative(w_w(t), t)]])

In [19]: # Linearization and Identification
    ...: # Equilibrium point
    ...: eq_pt = [0, 0, 0, 0, 0]
    ...: eq_dict = dict(zip([phi_b, phi_w,w_b, w_w, T], eq_pt))
    ...:

In [20]: # Motor and Wheel identification
    ...: def plot_Res(t, y, g):
    ...:     Y,T = mt.step(g,t)
    ...:     plt.plot(T,Y)
    ...:     plt.plot(t,y)
    ...:     plt.grid()
    ...:     plt.show()
    ...:
    ...: def residuals(p, y, t):
    ...:     [k,alpha] = p
```

```
   ...:     g = tf(k,[1,alpha,0])
   ...:     Y,T = mt.step(g,t)
   ...:     err=y-Y
   ...:     return err
   ...:
   ...: GearsRatio = 48.0/18.0
   ...:
   ...: kt = -1.62e-4   # [Nm/TqUnits]
   ...: Kt = GearsRatio*kt
   ...:
   ...: Input = 500
   ...:

In [21]: # Read the measure of the step response
   ...: x = np.loadtxt('idWheel.txt');
   ...: t = x[:,0]
   ...: y = x[:,1]
   ...:
   ...: t = t[100:]
   ...: y = y[100:]*(-1)
   ...: t = t-t[0]
   ...: yn = y/Input
   ...:
   ...: p0 = [1,1]
   ...: plsq = leastsq(residuals, p0, args=(yn, t))
   ...: K = plsq[0][0]
   ...: alpha = plsq[0][1]
   ...:

In [22]: # Wheel transfer function
   ...: G_w = tf(K,[1, alpha,0])

In [23]: G_w
Out[23]:

   -0.008959
---------------
s^2 + 0.05112 s

In [24]: # Compare simulation and real values
   ...: plot_Res(t, yn, G_w)
```




```
In [25]: J = Kt/K     # Jwheel/Kgears^2+Jmotore
    ...: D = alpha*J
    ...:
    ...: # Constants
    ...: gp = 9.81;
    ...:

In [26]: if Ball == 1:
    ...:     # Blue Ball
    ...:     Mb = 0.106712     # Mass
    ...:     Rb = 0.105/2         # Radius
    ...:     Jb = 2.3608e-04;    # Inertia
    ...:
    ...: elif Ball == 2:
    ...:     # Green Ball
    ...:     Mb = 0.13119         # Mass
    ...:     Rb = 0.09/2              # Radius
    ...:     Jb = 2.0/3*M_b*R_b**2      # Inertia
    ...:

In [27]: Rw = 0.285       # Wheel radius
    ...:
    ...: # Parameters for Model identification
    ...: R1p = Rw
    ...: R2p = Rb
    ...: M2p = Mb
    ...: J1p = J       # Identified Wheel J from motor side
    ...: d1p = 0      # not consider wheel friction in final model
    ...: J2p = Jb
    ...:
    ...: pars = [R_w, R_b, M_b, J_w, J_b, d_w, g]
    ...: par_vals =  [R1p, R2p, M2p, J1p, J2p, d1p, gp]
    ...: par_dict = dict(zip(pars, par_vals))
    ...:

In [28]: M = KM.mass_matrix_full
    ...: F = KM.forcing_full
    ...:
```

```
In [29]: M
Out[29]:
Matrix([
[1, 0,                                      0,                         0],
[0, 1,                                      0,                         0],
[0, 0, J_b*R_w**2/R_b**2 + M_b*(R_b + R_w)**2,      -J_b*R_w**2/R_b**2],
[0, 0,                     -J_b*R_w**2/R_b**2, J_b*R_w**2/R_b**2 + J_w]])

In [30]: F
Out[30]:
Matrix([
[                          w_b(t)],
[                          w_w(t)],
[M_b*g*(R_b + R_w)*sin(phi_b(t))],
[                            T(t)]])

In [31]: # Linearize the system for control design
    ...: # symbolically linearize about arbitrary equilibrium
    ...: M, linear_state_matrix, linear_input_matrix, inputs =
KM.linearize(new_method=True)

In [31]:

In [32]: # subst values at the equilibrium point and with the parameters
    ...: f_A_lin = linear_state_matrix.subs(eq_dict)
    ...: f_B_lin = linear_input_matrix.subs(eq_dict)
    ...:

In [33]: # compute A and B
    ...: Atmp = M.inv() * f_A_lin
    ...: Btmp = M.inv() * f_B_lin
    ...:

In [34]: Atmp
Out[34]:
Matrix([
[
0, 0, 1, 0],
[
0, 0, 0, 1],
[-M_b*g*(R_b + R_w)*(-J_b*R_w**2/R_b**2 - J_w)/(-J_b**2*R_w**4/R_b**4 + (-
J_b*R_w**2/R_b**2 - J_w)*(-J_b*R_w**2/R_b**2 - M_b*(R_b + R_w)**2)), 0, 0, 0],
[        J_b*M_b*R_w**2*g*(R_b + R_w)/(R_b**2*(-J_b**2*R_w**4/R_b**4 + (-
J_b*R_w**2/R_b**2 - J_w)*(-J_b*R_w**2/R_b**2 - M_b*(R_b + R_w)**2))), 0, 0,
0]])

In [35]: Btmp
Out[35]:
Matrix([
[
0],
[
0],
[                        J_b*R_w**2/(R_b**2*(-J_b**2*R_w**4/R_b**4 + (-
J_b*R_w**2/R_b**2 - J_w)*(-J_b*R_w**2/R_b**2 - M_b*(R_b + R_w)**2)))],
```

```
[-(-J_b*R_w**2/R_b**2 - M_b*(R_b + R_w)**2)/(-J_b**2*R_w**4/R_b**4 + (-
J_b*R_w**2/R_b**2 - J_w)*(-J_b*R_w**2/R_b**2 - M_b*(R_b + R_w)**2))]])

In [36]: Atmp = Atmp.subs(par_dict)
    ...: Btmp = Btmp.subs(par_dict)
    ...:

In [37]: A = np.matrix(Atmp)
    ...: B = np.matrix(Btmp)
    ...:

In [38]: A = A.astype('float64')
    ...: B = Kt*B.astype('float64')
    ...: C = [[1, 0, 0, 0], [0, 1, 0, 0]]
    ...: D = [[0],[0]]
    ...:

In [39]: A
Out[39]:
matrix([[ 0.        ,  0.        ,  1.        ,  0.        ],
        [ 0.        ,  0.        ,  0.        ,  1.        ],
        [19.37531895,  0.        ,  0.        ,  0.        ],
        [ 2.44306864,  0.        ,  0.        ,  0.        ]])

In [40]: B
Out[40]:
matrix([[ 0.        ],
        [ 0.        ],
        [-0.00298719],
        [-0.00820627]])

In [41]: C
Out[41]: [[1, 0, 0, 0], [0, 1, 0, 0]]

In [42]: D
Out[42]: [[0], [0]]

In [43]: # Plant continous and discrete
    ...: # input Motor torque
    ...: # output Wheel angle and Angle between Wheel CM and Ball CM
    ...: # States: phi_b phi_w w_b w_w
    ...: bow = ss(A, B, C, D)

In [44]: bow
Out[44]:
A = [[ 0.          0.          1.          0.        ]
 [ 0.          0.          0.          1.        ]
 [19.37531895  0.          0.          0.        ]
 [ 2.44306864  0.          0.          0.        ]]

B = [[ 0.        ]
 [ 0.        ]
 [-0.00298719]
 [-0.00820627]]
```

```
C = [[1 0 0 0]
 [0 1 0 0]]

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

```
In [45]: Ts=0.01                                        # Sampling time
    ...: bowD = c2d(bow, Ts, 'zoh')        # Get discrete state space form
    ...:

In [46]: # Control system design
    ...: # State feedback or LQR
    ...:
    ...: # Closed loop poles for state feedback
    ...: xi1 = np.sqrt(2)/2
    ...: xi2 = np.sqrt(3)/2
    ...: cl_p1 = [1,2*xi1*wn,wn**2]
    ...: cl_p2 =  [1,2*xi2*wn,wn**2]
    ...: cl_poly = sp.polymul(cl_p1, cl_p2)
    ...:
    ...: cl_2poles = np.roots(cl_p1)
    ...: cl_poles = np.roots(cl_poly)
    ...:

In [47]: Ad = bowD.A
    ...: Bd = bowD.B
    ...: Cd = bowD.C
    ...: Dd = bowD.D
    ...:

In [48]: if Controller == 1:
    ...:     # Controller without integral part
    ...:     cl_polesd = sp.exp(cl_poles*Ts)    # Desired discrete poles
    ...:     k = place(Ad, Bd, cl_polesd)
    ...:
    ...: elif Controller == 2:
    ...:     # LQR Controller
    ...:     Q = np.diag([10, 1, 20, 1]);
    ...:     R = [4];
    ...:     k, S, E = rp.dlqr(Ad, Bd, Q, R)
    ...:
    ...:     # Observer design parameters
    ...:     preg = sp.log(E[0])/Ts
    ...:     w0 = max(abs(preg));       # process spectral radius
    ...:
    ...:     # Modify poles for observer
    ...:     cl_poles = w0/wn*cl_poles
    ...:     cl_2poles = w0/wn*cl_2poles
    ...:

In [49]: if Observer == 1:
    ...:     # Reduced order observer
    ...:     T=[[0,0,1,0],[0,0,0,1]]
    ...:     obs_polesc = obs_k*cl_2poles
    ...:     obs_polesd = sp.exp(obs_polesc*Ts)
```

```
   ...:     r_obs=red_obs(bowD,T, obs_polesd)
   ...:     # Put Observer and controller together (compact form)
   ...:     ctr = comp_form(bowD, r_obs, k)
   ...:
   ...: elif Observer == 2:
   ...:     # Full Observer
   ...:     obs_polesc = obs_k*cl_poles
   ...:     obs_polesd = sp.exp(obs_polesc*Ts)
   ...:     f_obs = full_obs(bowD, obs_polesd)
   ...:     # Put Observer and controller together (compact form)
   ...:     ctr = comp_form(bowD, f_obs, k)
   ...:

In [50]: # Filter for AD sensor
   ...: wnf = 10
   ...: g = tf(wnf,[1,wnf])
   ...: gz = c2d(g,Ts)
   ...:

In [51]: # Saturation
   ...: Sat = 1300

In [52]: # Other system constants
   ...: Kd = 6.1e-2                     #  Voltage to Ball position [m]
   ...: D2PHI = Kd/(Rb+Rw)    # Voltage to Ball angle phi_b [rad]
   ...:
   ...: enc_w = 4096*GearsRatio/2/np.pi   # Motor encoder resolution (reduced
to motor)
   ...:

In [53]:
```