

```

1 # To add a new cell, type '# %%'
2 # To add a new markdown cell, type '# %% [markdown]'
3 # %%
4 from IPython import get_ipython
5
6 # %%
7 ## CE 295 - Energy Systems and Control
8 # HW 2 : State Estimation in Geothermal Heat Pump Drilling
9 # Oski Bear, SID 18681868
10 # Prof. Moura
11 # Due Date is written here
12
13 # BEAR_OSKI_HW2.ipynb
14
15 import numpy as np
16 import matplotlib.pyplot as plt
17 from scipy.integrate import odeint
18 from scipy import interp
19 from scipy import signal
20 get_ipython().run_line_magic('matplotlib', 'inline')
21 from __future__ import division
22 import pandas as pd
23 import control # Read http://python-control.sourceforge.net/manual/
24
25 fs = 15 # Font Size for plots
26
27 # Drill String Parameters
28
29 J_T = 100 # Table/top rotational inertia
30 J_B = 25 # Bottom/bit rotational inertia
31 k = 2 # Spring constant
32 b = 5 # Drag coefficient
33
34 # %% [markdown]
35 # Problem 1:
36 #
37 # - A: Define & write the modeling objective. What are the controllable and uncontrollable
38 # - Objective: The modeling objective is to estimate the drill bit velocity
39 # - State Variable, `x`:
40 # -  $w_T$ , viscous drag, top
41 # -  $w_B$ , viscous drag, bottom
42 # -  $\theta_T$ 
43 # -  $\theta_B$ 
44 # - Controllable Inputs, `u`:
45 # -  $T$ , Torque
46 # - uncontrollable Inputs,  $\omega$ :
47 # -  $T_f$ ,
48 # - Measured Outputs, `y`:
49 # - Table/Top rotation,  $w_T$ 
50 # - Performance Outputs, `z`:
51 # -  $\omega_B$ 
52 # - Parameters,  $\theta$ :
53 # -  $b$ , coeff of drag
54 # -  $k$ , spring coeff (**?)
55 # -  $J_T$ 
56 # -  $J_B$ 
57 # - B: Use Newton's second law in rotational coordinates to derive the equations of motion
58 # -  $\frac{d\omega_T}{dt} = \tau(t) - b\omega_T(t) - k[\theta_T(t) - \theta_B(t)]$ 
59 # -  $\frac{d\omega_B}{dt} = -\tau(t) - b\omega_B(t) - k[\theta_T(t) - \theta_B(t)]$ 
60 # -  $\frac{d\theta_T}{dt} = \omega_T$ 
61 # -  $\frac{d\theta_B}{dt} = \omega_B$ 

```

```

62 # - C: Write all the dynamical equations into matrix state space form. What are the A, B, C
63 #
64 # $$
65 # \frac{d}{dt}
66 # \begin{bmatrix} \omega_T & \omega_B & \theta_T & \theta_B \end{bmatrix}
67 # =
68 # \begin{bmatrix} \frac{-b}{J_T} & 0 & \frac{-k}{J_T} & \frac{k}{J_T} \\ 0 & \frac{-b}{J_B} & \frac{k}{J_B} & -\frac{k}{J_B} \\ \frac{1}{J_T} & 0 & \frac{-1}{J_B} & 0 \\ \tau & \tau_f & 0 & 0 \end{bmatrix}
69 # \begin{bmatrix} \omega_T & \omega_B & \theta_T & \theta_B \end{bmatrix}
70 # +
71 # \begin{bmatrix} \frac{1}{J_T} & 0 & 0 & \frac{-1}{J_B} \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}
72 # \begin{bmatrix} \tau & \tau_f \end{bmatrix}
73 # $$
74 #
75 # $C = \omega_T = \begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix}$
76 #
77 #
78 # %% [markdown]
79 # Problem 2:
80 #
81 # A:
82 # - $O = \begin{bmatrix} C \\ CA \\ CA^2 \\ CA^3 \end{bmatrix}$
83 #
84 #
85 #
86 # \end{bmatrix}$
87 #
88 # - Because rank(O) = 3 < than the n states, not all states are observable.
89 #
90 # B:
91 # $\frac{d}{dt}$
92 # \begin{bmatrix} \omega_T & \omega_B & \theta_T & \theta_B \end{bmatrix}
93 # =
94 # \begin{bmatrix} \frac{-b}{J_T} & 0 & \frac{k}{J_T} & -\frac{k}{J_T} \\ 0 & \frac{-b}{J_B} & \frac{k}{J_B} & -\frac{k}{J_B} \\ 1 & -1 & 0 & 0 \end{bmatrix}
95 #
96 #
97 # \end{bmatrix}
98 # \begin{bmatrix} \omega_T & \omega_B & \theta_T & \theta_B \end{bmatrix}
99 #
100 #
101 # \end{bmatrix}
102 # +
103 # \begin{bmatrix} \frac{1}{J_T} & 0 & 0 & \frac{-1}{J_B} \\ 0 & \frac{-1}{J_B} & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}
104 #
105 #
106 # \end{bmatrix}
107 # \begin{bmatrix} \tau & \tau_f \end{bmatrix}
108 #
109 # \end{bmatrix}
110 # $
111 #
112 # - $C = \omega_T = \begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix}$
113 #
114 #
115 # C:
116 # - $O = \begin{bmatrix} C \\ CA \\ CA^2 \end{bmatrix}$
117 #
118 #
119 # \end{bmatrix}$
120 #
121 # - Now that rank(O) = 3 == n states, the system is now considered observable
122 #
123 #

```

```

124 # %%
125 ## Problem 2 - Observability Analysis
126
127 # State space matrices
128 A4 = np.matrix([[ -b/J_T, 0, -k/J_T, k/J_T],
129                 [ 0, -b/J_B, k/J_B, -k/J_B],
130                 [ 1, 0, 0, 0],
131                 [ 0, 1, 0, 0]])
132
133 B4 = np.matrix([[ 1/J_T, 0],
134                 [ 0, -1/J_B],
135                 [ 0, 0],
136                 [ 0, 0]])
137
138 C4 = np.matrix([[ 1, 0, 0, 0]])
139
140 # Compute observability Matrix for 4-state system and rank
141 O4 = control.observ(A4,C4)
142 print('Rank of Observability Matrix for four-state system')
143 print(np.linalg.matrix_rank(O4))
144
145 # New A Matrix, for 3-state system
146 A = np.matrix([[ -b/J_T, 0, -k/J_T],
147                 [ 0, -b/J_B, k/J_B],
148                 [ 1, -1, 0]
149                 ])
150
151 B = np.matrix([[ 1/J_T],
152                 [ 0],
153                 [ 0]
154                 ])
155
156 C = np.matrix([[ 0, 1, 0]])
157
158 D = np.matrix([0]) #Add empty D for Q4
159
160 # Observability Matrix for 3-state system and rank
161 O = control.observ(A,C)
162 print('Rank of Observability Matrix for three-state system')
163 print(np.linalg.matrix_rank(O))
164
165
166 # %%
167 ## Load Data
168 data=np.asarray(pd.read_csv("HW2_Data.csv",header=None))
169
170 t = data[:,0]      # t : time vector [sec]
171 y_m = data[:,1]    # y_m : measured table velocity [radians/sec]
172 Torq = data[:,2]   # Torq: table torque [N-m]
173 omega_B_true = data[:,3] # \omega_B : true rotational speed of bit [radians/sec]
174
175 # Plot Data
176 plt.figure(num=1, figsize=(8, 9), dpi=80, facecolor='w', edgecolor='k')
177
178 plt.subplot(2,1,1)
179 plt.plot(t, Torq)
180 plt.ylabel('Torque [N-m]')
181 plt.xlabel('Time [sec]')
182 plt.title('Torque vs Time')
183 # Plot table torque
184
185 plt.subplot(2,1,2)

```

```

186 plt.plot(t, y_m, color='g')
187 plt.ylabel('Velocity [rads/sec]')
188 plt.xlabel('Time [sec]')
189 plt.title('Measured Table Velocity vs Time')
190 plt.tight_layout()
191 # Plot measured table velocity
192
193 plt.show()
194
195 # %% [markdown]
196 # Problem 4:
197 #
198 # A: Re(Eigenvalues of A): -0.08322949,-0.08338525,-0.08338525
199 #
200 # $
201 # \dot{\hat{x}} = A\hat{x}(t) + Bu(t) + L[y(t) - \hat{y}(t)] \ \
202 # \hat{y} = C\dot{\hat{x}}(t) + Du(t) \ \
203 # $
204 # simplify $\dot{\hat{x}}$
205 #
206 # 1. Distribute L, substitute in full form of $\hat{y}(t)$:
207 # $
208 # \dot{\hat{x}} = A\hat{x}(t) + Bu(t) + Ly(t) - L\hat{y}(t) \ \
209 # \dot{\hat{x}} = A\hat{x}(t) + Bu(t) + Ly(t) - L[C\dot{\hat{x}}(t) + Du(t)]
210 # $
211 #
212 # 2. Distribute L again:
213 # $
214 # \dot{\hat{x}} = A\hat{x}(t) - LC\dot{\hat{x}}(t) + Bu(t) - LDu(t) + Ly(t)
215 # $
216 #
217 # B:
218 # - Re(Eigenvalues of A): [-0.08322949,-0.08338525,-0.08338525]
219 # - Re(Selected Eigenvalues): $\lambda_i = $ [-0.4993769, -0.50031153,-0.50031153]
220 # - derived from the original eigenvalues * 6, in line with the "general rule of thumb"
221 #
222 # C:
223 # - using the equations derived in 4.A:
224 # $$
225 # \begin{bmatrix} \dot{\hat{x}} \end{bmatrix}
226 # = \begin{bmatrix} A-LC \end{bmatrix} \begin{bmatrix} \hat{x} \end{bmatrix} +
227 # \begin{bmatrix} B-LD, L \end{bmatrix} \begin{bmatrix} u \end{bmatrix} \ y \end{bmatrix}
228 # $$
229 #
230 # $$
231 # \begin{bmatrix} \hat{y} \end{bmatrix} = \begin{bmatrix} C \end{bmatrix} \begin{bmatrix} \hat{x} \end{bmatrix}
232 # $$
233 #
234 # D: see plot below
235
236 # %%
237 ## Problem 4 - Luenberger Observer
238
239 # Eigenvalues of open-loop system
240 print('Eigenvalues of open-loop system:')
241 lam_A, evec = np.linalg.eig(A)
242 print(lam_A)
243
244 # Desired poles of estimation error system
245 # They should have negative real parts
246 # Complex conjugate pairs
247 lam_luen = lam_A * 6

```

```

248
249 # Compute observer gain (See Remark 3.1 in Notes. Use "acker" command)
250 L = control.acker(A.T,C.T,lam_luen).T
251
252 # State-space Matrices for Luenberger Observer
253 A_lobs = (A - L*C)
254 B_lobs = np.hstack((B - L*D, L))#TODO HELP
255 C_lobs = C
256 D_lobs = np.matrix([[0,0]])
257
258 sys_lobs = signal.lti(A_lobs,B_lobs,C_lobs,D_lobs)
259
260 # Inputs to observer
261 u = np.array([Torq, y_m]).T
262
263 # Initial Conditions
264 x_hat0 = [0,0,0]
265
266 # Simulate Response
267 tsim, y, x_hat = signal.lsim(sys_lobs, U=u, T=t, X0=x_hat0)
268
269 # Parse states
270 theta_hat = x_hat[:,2]
271 omega_T_hat = x_hat[:,0]
272 omega_B_hat = x_hat[:,1]
273
274 #Add RMS
275 luen_est_err = omega_B_true-omega_B_hat
276 RMSE = np.sqrt(np.mean(np.power(omega_B_true-omega_B_hat,2)))
277 print('Luenberger RMSE: ' + str(RMSE) + ' rad/s')
278
279 # Plot Results
280 plt.figure(num=1, figsize=(8, 9), dpi=80, facecolor='w', edgecolor='k')
281
282 plt.subplot(2,1,1)
283 # Plot true and estimated bit velocity
284 plt.plot(t,omega_B_true, 'C0',label='Bit Velocity')
285 plt.plot(t,omega_B_hat, 'C1', label='Est. Bit Velocity')
286
287 plt.xlabel('Velocity [rad/sec]')
288 plt.ylabel('Time [sec]')
289 plt.title('True vs Estimated Bit Velocity (Luenberger Observer)')
290 plt.legend()
291 plt.subplot(2,1,2)
292 # Plot error between true and estimated bit velocity
293 plt.plot(t,luen_est_err, 'C2')
294
295 plt.xlabel('Velocity [rad/sec]')
296 plt.ylabel('Time [sec]')
297 plt.title('True vs Estimated Error rate')
298
299 plt.show()
300
301 # %% [markdown]
302 # Problem 5:
303 #
304 # - A: See Ch2.4.45-48
305 # - B: Using the identity matrix as a starting point, I simply tuned by testing different
306 # - C: See plots below
307 # - D: `Re(Selected Luenberger Eigenvalues):  $\lambda_i = -0.4993769, -0.50031153, -0.50031153$ ` vs `R
308 #
309

```

```

310 # %%
311 ## Problem 5 - Kalman Filter
312 # Noise Covariances
313 W = .0005 * np.identity(3) # Should be 3x3 because of the # of x states
314 N = .02
315 Sig0 = np.identity(3)
316
317 # Initial Condition
318 x_hat0 = [0,0,0]
319 states0 = np.r_[x_hat0, np.squeeze(np.asarray(Sig0.reshape(9,1)))]
320
321 # Ordinary Differential Equation for Kalman Filter
322 def ode_kf(z,it):
323
324     # Parse States
325     x_hat = np.matrix(z[:3]).T
326     Sig = np.matrix((z[3:]).reshape(3,3))
327
328     # Interpolate input signal data
329     iTorq = interp(it, t, Torq)
330     iy_m = interp(it, t, y_m)
331
332     # Compute Kalman Gain
333     L = Sig * C.T * (1/N)
334
335     # Kalman Filter
336     x_hat_dot = A * x_hat + B * iTorq + L * (iy_m - (C * x_hat))
337
338     # Riccati Equation
339     Sig_dot = Sig * A.T + A * Sig + W - Sig * C.T * (1/N) * C * Sig
340
341     # Concatenate LHS
342     z_dot = np.r_[x_hat_dot, Sig_dot.reshape(9,1)]
343
344     return(np.squeeze(np.asarray(z_dot)))
345
346
347 # Integrate Kalman Filter ODEs
348 z = odeint(ode_kf, states0, t)
349
350 # Parse States
351 theta_hat = z[:,2]
352 omega_T_hat = z[:,0]
353 omega_B_hat = z[:,1]
354 Sig33 = z[:,11] # Parse out the (3,3) element of Sigma only!
355
356 omega_B_tilde = omega_B_true - omega_B_hat
357 omega_B_hat_upperbound = omega_B_hat + np.sqrt(Sig33)
358 omega_B_hat_lowerbound = omega_B_hat - np.sqrt(Sig33)
359
360 RMSE = np.sqrt(np.mean(np.power(omega_B_tilde,2)))
361 print('Kalman Filter RMSE: ' + str(RMSE) + ' rad/s')
362
363
364 # Plot Results
365 plt.figure(num=3, figsize=(8, 9), dpi=80, facecolor='w', edgecolor='k')
366
367 plt.subplot(2,1,1)
368 # Plot true and estimated bit velocity
369 plt.plot(t,omega_B_true,'C0', label='True Bit Velocity')
370 plt.plot(t,omega_B_hat, 'C1', label='Est. Bit Velocity')
371 plt.plot(t,omega_B_hat_upperbound, 'C3--', label='Upper STD bound')

```

```

372 plt.plot(t, omega_B_hat_lowerbound, 'C3--', label='Lower STD bound')
373
374 plt.title('True vs Estimated Bit Velocity')
375 plt.xlabel('Time [sec]')
376 plt.ylabel('Bit Velocity [rads/sec]')
377 plt.legend()
378 # Plot estimated bit velocity plus/minus one sigma
379
380 plt.subplot(2,1,2)
381 # Plot error between true and estimated bit velocity
382 plt.plot(t, omega_B_tilde, 'C2')
383 plt.title('True vs Estimated Error (Kalman Filter)')
384 plt.xlabel('Time [sec]')
385 plt.ylabel('Bit Velocity [rads/sec]')
386
387 plt.show()
388
389 # %% [markdown]
390 # Problem 6:
391 #
392 # A: Use the original 3-equation ODE system, but replace the  $k\theta$  term to reflect the
393 #  $\frac{d\theta}{dt} = \omega_T - \omega_B$ 
394 #  $\frac{dw_T}{dt} = \tau(t) - b\omega_T(t) - [k_1\theta(t) + k_2\theta^3(t)]$ 
395 #  $\frac{dw_B}{dt} = -\tau(t) - b\omega_B(t) - [k_1\theta(t) + k_2\theta^3(t)]$ 
396 #  $\frac{d\omega_T}{dt} = \omega_T$ 
397 #  $\frac{d\omega_B}{dt} = \omega_B$ 
398 #
399 # Create F(t) and H(t) matrices:
400 #  $F(t) = \begin{bmatrix} 0 & 1 & -1 \\ \frac{-k_1}{J_T} - \frac{3k_2}{J_B}\theta^2 & \frac{-b}{J_T} & 0 \\ \frac{k_1}{J_T} + \frac{3k_2}{J_B}\theta^2 & 0 & \frac{-b}{J_B} \end{bmatrix}$ 
401 #
402 #  $H(t) = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}$ 
403 #
404 # %%
405 ## Problem 6 - Extended Kalman Filter
406 #
407 # New nonlinear spring parameters
408 k1 = 2
409 k2 = 0.25
410
411 # Noise Covariances
412 W = 0.005 * np.identity(3) # You design this one.
413 N = 0.02
414 Sig0 = np.identity(3)
415
416 # Initial Condition
417 x_hat0 = [0, 0, 0]
418 states0 = np.r_[x_hat0, np.squeeze(np.asarray(Sig0.reshape(9,1)))]
419
420 # Ordinary Differential Equation for Kalman Filter
421 def ode_ekf(z, it):
422
423     # Parse States
424     theta_hat = z[0]
425     omega_T_hat = z[1]
426     omega_B_hat = z[2]
427     Sig = np.matrix((z[3:]).reshape(3,3))
428
429     # Interpolate input signal data

```

```

434     iTorq = interp(it, t, Torq)
435     iy_m = interp(it, t, y_m)
436
437     # Compute Jacobians
438     F = np.matrix([[0,1,-1],
439                    [(-k1/J_T)-(3*k2/J_T)*theta_hat**2,-b/J_T,0],
440                    [(k1/J_B)+(3*k2/J_B)*theta_hat**2,0,-b/J_B]
441                    ])
442     H = np.matrix([[0,1,0]])
443
444     # Compute Kalman Gain
445     L = (Sig * H.T * (1/N))
446
447     # Compute EKF system matrices
448     y_hat = omega_T_hat
449
450     theta_hat_dot = (omega_T_hat
451                     - omega_B_hat
452                     + L[0] * (iy_m-y_hat))
453
454     omega_T_hat_dot = omega_T_hat - omega_B_hat + L[0] * (iy_m-y_hat)
455     omega_T_hat_dot = ((iTorq/J_T)
456                       - (b*omega_T_hat/J_T)
457                       - (k1*theta_hat+k2*theta_hat**3)/J_T
458                       + L[1] * (iy_m-y_hat))
459     omega_B_hat_dot = -(b*omega_B_hat/J_B)
460                       + (k1*theta_hat+k2*theta_hat**3)/J_B
461                       + L[2] * (iy_m-y_hat))
462
463     # Riccati Equation
464     Sig_dot = ((Sig*F.T) + (F*Sig) + W - Sig * H.T * (1/N) * H * Sig)
465
466     # Concatenate LHS
467     z_dot = np.r_[theta_hat_dot, omega_T_hat_dot, omega_B_hat_dot, Sig_dot.reshape(9,1)]
468
469     return(np.squeeze(np.asarray(z_dot)))
470
471
472 # Integrate Extended Kalman Filter ODEs
473 z = odeint(ode_ekf, states0, t)
474
475 # Parse States
476 theta_hat = z[:,0]
477 omega_T_hat = z[:,1]
478 omega_B_hat = z[:,2]
479 Sig33 = z[:,-1]
480
481 omega_B_tilde = omega_B_true - omega_B_hat
482 omega_B_hat_upperbound = omega_B_hat + np.sqrt(Sig33)
483 omega_B_hat_lowerbound = omega_B_hat - np.sqrt(Sig33)
484
485 RMSE = np.sqrt(np.mean(np.power(omega_B_tilde,2)))
486 print('Extended Kalman Filter RMSE: ' + str(RMSE) + ' rad/s')
487
488
489 # Plot Results
490 plt.figure(num=3, figsize=(8, 9), dpi=80, facecolor='w', edgecolor='k')
491
492 plt.subplot(2,1,1)
493 # Plot true and estimated bit velocity
494 plt.plot(t,omega_B_true, 'C0', label='True Bit Velocity')
495 plt.plot(t,omega_B_hat, 'C1', label='Est. Bit Velocity')

```



```
496 # Plot estimated bit velocity plus/minus one sigma
497 plt.plot(t, omega_B_hat_upperbound, 'C3--', label='Upper STD Bound')
498 plt.plot(t, omega_B_hat_lowerbound, 'C3--', label='Lower STD Bound')
499
500 plt.title('True vs Estimated Bit Velocity (EKF)')
501 plt.xlabel('Time [sec]')
502 plt.ylabel('Bit Velocity [rads/sec]')
503 plt.legend('')
504
505 plt.subplot(2,1,2)
506 # Plot error between true and estimated bit velocity
507 plt.plot(t, omega_B_tilde, 'C2')
508
509 plt.title('True vs Estimated Error (EKF)')
510 plt.xlabel('Time [sec]')
511 plt.ylabel('Bit Velocity [rads/sec]')
512
513 plt.show()
514
515
516 # %%
```