HW 2: State Estimation in Oil & Gas Well Drilling

Franklin Zhao
SID: 3033030808

16 Feb. 2018

Problem 1: Dynamic System Modeling

(a)

Modeling objective: The objective is to formulate a mathmetical model that estimates the drill bit velocity given the table torque $T(t)$.
Controllable input: Table torque $\mathbf{T(t)}$
Uncontrollable input: Frictional torque $\mathbf{T_f(t)}$
Measured output: Table velocity $\omega_{\mathbf{T}}(\mathbf{t})$
Performance output: Bit velocity $\omega_{\mathbf{B}}(\mathbf{t})$

(b)

For the top/table:

$$J_T\ddot{\theta}_T(t) = -k(\theta_T(t) - \theta_B(t)) - b\omega_T(t) + T(t) \tag{1}$$

For the bottom/bit:

$$J_B\ddot{\theta}_B(t) = k(\theta_T(t) - \theta_B(t)) - b\omega_B(t) - T_f(t) \tag{2}$$

(c)

$$\frac{\partial}{\partial t}\begin{bmatrix} \theta_T \\ \dot{\theta}_T \\ \theta_B \\ \dot{\theta}_B \end{bmatrix} = \underbrace{\begin{bmatrix} 0 & 1 & 0 & 0 \\ -\frac{k}{J_T} & -\frac{b}{J_T} & \frac{k}{J_T} & 0 \\ 0 & 0 & 0 & 1 \\ \frac{k}{J_B} & 0 & -\frac{k}{J_B} & -\frac{b}{J_B} \end{bmatrix}}_{\mathbf{A}} \begin{bmatrix} \theta_T \\ \dot{\theta}_T \\ \theta_B \\ \dot{\theta}_B \end{bmatrix} + \underbrace{\begin{bmatrix} 0 \\ \frac{1}{J_T} \\ 0 \\ 0 \end{bmatrix}}_{\mathbf{B}} T + \begin{bmatrix} 0 \\ 0 \\ 0 \\ -\frac{1}{J_B} \end{bmatrix} T_f \tag{3}$$

$$\dot{\theta}_T = \underbrace{\begin{bmatrix} 0 & 1 & 0 & 0 \end{bmatrix}}_{\mathbf{C}} \begin{bmatrix} \theta_T \\ \dot{\theta}_T \\ \theta_B \\ \dot{\theta}_B \end{bmatrix} \tag{4}$$

Problem 2: Observability Analysis

(a)

```python
# State space matrices
A4 = np.matrix([[0, 1, 0, 0], [-k/J_T, -b/J_T, k/J_T, 0], [0, 0, 0, 1], [k/J_B, 0, -k/J_B, -b/J_B]])
B4 = np.matrix([[0], [1/J_T], [0], [0]])
C4 = np.matrix([0, 1, 0, 0])

# Compute observability Matrix for 4-state system and rank
O4 = np.vstack([C4, C4 @ A4, C4 @ A4 @ A4, C4 @ A4 @ A4 @ A4])
print('Rank of Observability Matrix for four-state system:', np.linalg.matrix_rank(O4))
```

```
Rank of Observability Matrix for four-state system: 3
```

Since $rank(\mathcal{O}) = 3 < 4$, (A, C) is not observable.

(b)

$$\frac{\partial}{\partial t} \begin{bmatrix} \theta \\ \dot{\theta}_T \\ \dot{\theta}_B \end{bmatrix} = \underbrace{\begin{bmatrix} 0 & 1 & -1 \\ -\frac{k}{J_t} & -\frac{b}{J_t} & 0 \\ \frac{k}{J_B} & 0 & -\frac{b}{J_B} \end{bmatrix}}_{\mathbf{A}} \begin{bmatrix} \theta \\ \dot{\theta}_T \\ \dot{\theta}_B \end{bmatrix} + \underbrace{\begin{bmatrix} 0 \\ \frac{1}{J_T} \\ 0 \end{bmatrix}}_{\mathbf{B}} T + \begin{bmatrix} 0 \\ 0 \\ -\frac{1}{J_B} \end{bmatrix} T_f \tag{5}$$

$$\dot{\theta}_T = \underbrace{\begin{bmatrix} 0 & 1 & 0 \end{bmatrix}}_{\mathbf{C}} \begin{bmatrix} \theta \\ \dot{\theta}_T \\ \dot{\theta}_B \end{bmatrix} \tag{6}$$

(c)

```python
# New A Matrix, for 3-state system
A = np.matrix([[0, 1, -1], [-k/J_T, -b/J_T, 0], [k/J_B, 0, -b/J_B]])
B = np.matrix([[0], [1/J_T], [0]])
C = np.matrix([0, 1, 0])
```

```

# Observability Matrix for 3-state system and rank
O = np.vstack([C, C @ A, C @ A @ A])
print('Rank of Observability Matrix for three-state system:', np.linalg.
    matrix_rank(O))
```

```
Rank of Observability Matrix for three-state system: 3
```

Since $rank(\mathcal{O}) = 3$, (A, C) is observable.

Problem 3: Measurement Data

```
# Load Data
data=np.asarray(pd.read_csv("HW2_Data.csv",header=None))

t = data[:,0]        # t    : time vector [sec]
y_m = data[:,1]      # y_m : measured table velocity [radians/sec]
Torq = data[:,2]     # Torq: table torque [N-m]
omega_B_true = data[:,3]      # \omega_B : true rotational speed of bit [radians
    /sec]

# Plot Data
plt.figure(num=1, figsize=(8, 9), dpi=150, facecolor='w', edgecolor='k')

plt.subplot(2,1,1)
plt.plot(t, Torq)
plt.xlabel('Time')
plt.ylabel('Table torque')
# Plot table torque

plt.subplot(2,1,2)
plt.plot(t, y_m)
plt.xlabel('Time')
plt.ylabel('Measured table velocity')
# Plot measured table velocity

plt.show()
```

The plots of table torque and measured table velocity versus time are shown in Figure 1.

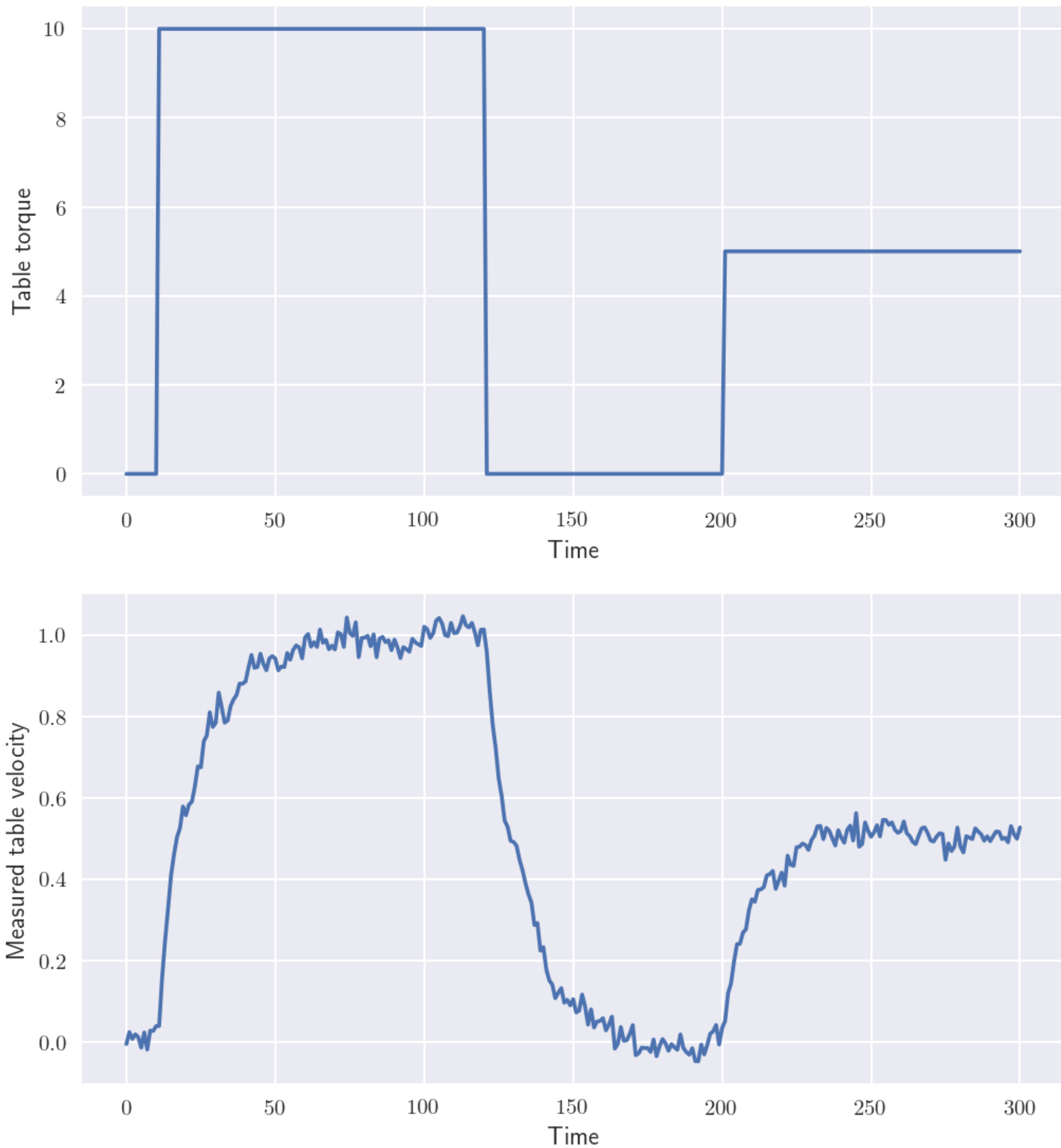

Fig. 1: Table torque and measured table velocity versus time

Problem 4: Luenberger Observer

(a)

Denote $\begin{bmatrix} \hat{\theta} \\ \hat{\dot{\theta}}_T \\ \hat{\dot{\theta}}_B \end{bmatrix}$ by $\hat{x}(t)$, $\hat{\dot{\theta}}_T$ by $\hat{y}(t)$, $T(t)$ by $u(t)$, and the Luenberger observer equations will be:

$$\dot{\hat{x}}(t) = A\hat{x}(t) + Bu(t) + L[y(t) - \hat{y}(t)], \quad \hat{x}(0) = \hat{x}_0 \tag{7}$$

$$\hat{y}(t) = C\hat{x}(t) \tag{8}$$

```python
# Eigenvalues of open-loop system
lam_A = np.linalg.eig(A)[0]
print('Eigenvalues of open-loop system:', lam_A)
```

```
Eigenvalues of open-loop system:
[-0.08338525+0.29860789j -0.08338525-0.29860789j -0.08322949+0.j        ]
```

(b)

The eigenvalues I chose were $2.5\lambda(A)$. The choice was justified in the RMSE error to be computed and converge speed to be shown in part (d). Basically I chose $\{5, 1, 2, 3, 2.5\}$ and did the validation. It turned out that $2.5\lambda(A)$ was the best choice.

```python
# Desired poles of estimation error system
#   They should have negative real parts
#   Complex conjugate pairs
lam_luen = 2.5 * lam_A

# Compute observer gain
L = control.acker(A.T, C.T, lam_luen).T
print('L:\n', L)
```

```
L:
[[-25.125]
[  0.375]
[ -0.675]]
```

(c)

Write Equation (7)(8) in LTI form, and we obtain A_{lobs}, B_{lobs}, C_{lobs}:

$$\dot{\hat{x}}(t) = \underbrace{(A - LC)}_{A_{lobs}}\hat{x}(t) + \underbrace{\begin{bmatrix} B & L \end{bmatrix}}_{B_{lobs}}\begin{bmatrix} u(t) \\ y(t) \end{bmatrix} \tag{9}$$

$$\hat{y}(t) = C_{lobs}\hat{x}(t) \tag{10}$$

(d)

The eigenvalues I chose were $2.5\lambda(A)$:

`array([-0.20846314+0.74651974j, -0.20846314-0.74651974j, -0.20807373+0.j])`

```
# State-space Matrices for Luenberger Observer
A_lobs = A - L @ C
B_lobs = np.hstack([B, L])
C_lobs = C
D_lobs = np.matrix([0, 0])

sys_lobs = signal.lti(A_lobs, B_lobs, C_lobs, D_lobs)

# Inputs to observer
u = np.vstack([Torq, y_m]).T

# Initial Conditions
x_hat0 = [0, 0.2, 0.2]

# Simulate Response
tsim, y, x_hat = signal.lsim(sys_lobs, U=u, T=t, X0=x_hat0)

# Parse states
theta_hat = x_hat[:,0]
omega_T_hat = x_hat[:,1]
omega_B_hat = x_hat[:,2]

# Compute RMSE
est_error = omega_B_true - omega_B_hat
RMSE = np.sqrt(np.mean(est_error ** 2))
print('Luenberger Observer RMSE:', RMSE)

# Plot Results
```

```python
plt.figure(num=1, figsize=(8, 9), dpi=150, facecolor='w', edgecolor='k')

plt.subplot(2,1,1)
# Plot true and estimated bit velocity
plt.plot(t, omega_B_hat, label=r'Estimated velocity $\hat{\omega}_B$')
plt.plot(t, omega_B_true, label=r'True velocity $\omega_B$')
plt.legend()
plt.xlabel('Bit velocity')
plt.ylabel('Time')

plt.subplot(2,1,2)
# Plot error between true and estimated bit velocity
plt.plot(t, omega_B_true - omega_B_hat)
plt.xlabel('Estimation error')
plt.ylabel('Time')
```

```
Luenberger Observer RMSE: 0.0991557694635
```

The result of Luenberger observer is shown in Figure 2

Fig. 2: Luenberger observer result

Problem 5: Kalman Filter (KF) Design

(a)

$$\dot{\hat{x}}(t) = A\hat{x}(t) + Bu(t) + L(t)(y(t) - \hat{y}(t)) \tag{11}$$

$$\hat{y}(t) = C\hat{x}(t) \tag{12}$$

$$L(t) = \Sigma(t)C^T N^{-1} \tag{13}$$

where $\Sigma(t)$ is the solution of the following differential equation:

$$\dot{\Sigma}(t) = \Sigma(t)A^T + A\Sigma(t) + W - \Sigma(t)C^T N^{-1} C\Sigma(t), \quad \Sigma(0) = \Sigma_0 \tag{14}$$

where W and N are covariance matrices of $w(t)$ and $n(t)$.

(b)

```python
# Noise Covariances
W = np.matrix([[0.0006, 0.005, 0.001], [0.02, 0.003, 0.004], [0.004, 0.001,
    0.003]]) #You design this one.
N = 0.02
Sig0 = np.identity(3)

# Initial Condition
x_hat0 = [0, 0, 0]
states0 = np.r_[x_hat0, np.squeeze(np.asarray(Sig0.reshape(9,1)))]

# Ordinary Differential Equation for Kalman Filter
def ode_kf(z,it):

    # Parse States
    x_hat = np.matrix(z[:3]).T
    Sig = np.matrix((z[3:]).reshape(3,3))

    # Interpolate input signal data
    iTorq = interp(it, t, Torq)
    iy_m = interp(it, t, y_m)

    # Compute Kalman Gain
    L = Sig @ C.T / N

    # Kalman Filter
```

```
    x_hat_dot = A @ x_hat + B * iTorq + L @ (iy_m - C @ x_hat)

    # Riccati Equation
    Sig_dot = Sig @ A.T + A @ Sig + W - Sig @ C.T / N @ C @ Sig

    # Concatenate LHS
    z_dot = np.r_[x_hat_dot, Sig_dot.reshape(9,1)]

    return(np.squeeze(np.asarray(z_dot)))


# Integrate Kalman Filter ODEs
z = odeint(ode_kf, states0, t)

# Parse States
theta_hat = z[:,0]
omega_T_hat = z[:, 1]
omega_B_hat = z[:, 2]
Sig33 = z[:, -1]     # Parse out the (3,3) element of Sigma only!

omega_B_tilde = omega_B_true - omega_B_hat
omega_B_hat_upperbound = omega_B_hat + np.sqrt(Sig33)
omega_B_hat_lowerbound = omega_B_hat - np.sqrt(Sig33)

RMSE = np.sqrt(np.mean(np.power(omega_B_tilde,2)))
print('Kalman Filter RMSE: ' + str(RMSE) + ' rad/s')
```

```
Kalman Filter RMSE: 0.0560275348234 rad/s
```

The value of W I tuned is listed as follow. Basically I determined the magnitude first. Then based on the plots in part (c), I tuned the diagonal elements, and finally tuned other elements. It turned out that the RMSE did not change too much, but Σ_{33} changed dramatically.

```
W = np.matrix([[0.0006, 0.005, 0.001],
               [0.02,   0.003, 0.004],
               [0.004,  0.001, 0.003]])
```

(c)

```
# Plot Results
plt.figure(num=3, figsize=(8, 9), dpi=150, facecolor='w', edgecolor='k')

plt.subplot(2,1,1)
```

```
#   Plot true and estimated bit velocity
#   Plot estimated bit velocity plus/minus one sigma
plt.plot(t, omega_B_hat, 'r', label=r'Estimated velocity $\hat{\omega}_B$')
plt.plot(t, omega_B_true, 'g', label=r'True velocity $\omega_B$')
plt.plot(t, omega_B_hat_upperbound, 'm--', label=r'Upper bound $\hat{\omega}_B+\sqrt{\Sigma_{33}}$')
plt.plot(t, omega_B_hat_lowerbound, 'b--', label=r'Lower bound $\hat{\omega}_B-\sqrt{\Sigma_{33}}$')
plt.legend()
plt.xlabel('Bit velocity')
plt.ylabel('Time')

plt.subplot(2,1,2)
#   Plot error between true and estimated bit velocity


plt.subplot(2,1,2)
# Plot error between true and estimated bit velocity
plt.plot(t, omega_B_tilde)
plt.xlabel('Estimation error')
plt.ylabel('Time')

plt.show()
```

The result of Kalman Filter is shown in Figure 3.

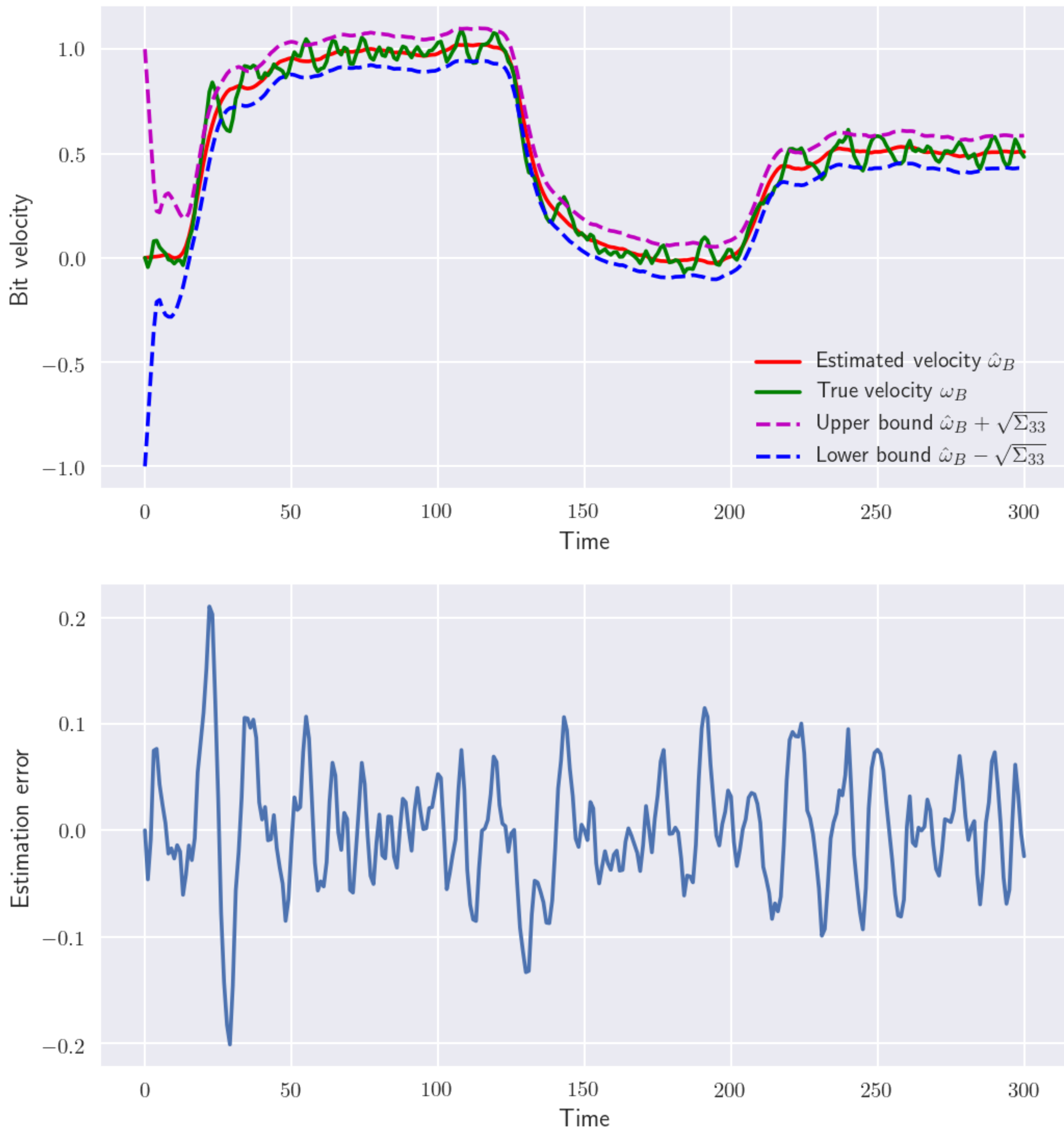

Fig. 3: Kalman filter result

(d)

```
# Compute eig(A-L(300)C)
Sig = np.matrix((z[-1, 3:]).reshape(3,3))
L = Sig @ C.T / N
```

```
np.linalg.eig(A - L @ C)[0]
```

```
array([-0.08707987+0.2849352j, -0.08707987-0.2849352j, -0.34475403+0.j])
```

The 1st and 2nd poles are very close to the eigenvalues I chose in the Luenberger observer, but the 3rd one is a bit more different.

Problem 6: Extended Kalman Filter (EKF) Design

Replace Hooke's law with nonlinear spring torque relationship, and the EKF can be obtained:

$$\underbrace{\frac{\partial}{\partial t}\begin{bmatrix}\theta \\ \dot{\theta}_T \\ \dot{\theta}_B\end{bmatrix}}_{\hat{x}(t)} = \underbrace{\begin{bmatrix}\hat{\theta}_T - \hat{\theta}_B \\ \frac{-k_1\theta - k_2\theta^3 - b\dot{\theta}_T + T(t)}{J_T} \\ \frac{k_1\theta + k_2\theta^3 - b\dot{\theta}_B}{J_B}\end{bmatrix}}_{f(x(t),u(t))} + w(t), \quad x(0) = x_0 \tag{15}$$

$$\dot{\theta}_T = C\underbrace{\begin{bmatrix}\theta \\ \dot{\theta}_T \\ \dot{\theta}_B\end{bmatrix}}_{h(x(t),u(t))} + n(t) \tag{16}$$

Hence, $F(t)$ and $H(t)$ can be derived:

$$F(t) = \frac{\partial f}{\partial x}(\hat{x}(t), u(t)) = \begin{bmatrix} 0 & 1 & -1 \\ \frac{-k_1 - 3k_2\hat{\theta}^2}{J_T} & \frac{-b}{J_T} & 0 \\ \frac{k_1 + 3k_2\hat{\theta}^2}{J_B} & 0 & \frac{-b}{J_B}\end{bmatrix} \tag{17}$$

$$H(t) = \frac{\partial h}{\partial x}(\hat{x}(t), u(t)) = C = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix} \tag{18}$$

```
# New nonlinear spring parameters
k1 = 2
k2 = 0.25

# Noise Covariances
W = np.matrix([[0.0006, 0.005, 0.001], [0.02, 0.003, 0.004], [0.004, 0.001,
    0.003]]) #You design this one.
N = 0.02
Sig0 = np.identity(3)

# Initial Condition
```

```python
x_hat0 = [0, 0, 0]
states0 = np.r_[x_hat0, np.squeeze(np.asarray(Sig0.reshape(9,1)))]

# Ordinary Differential Equation for Kalman Filter
def ode_ekf(z,it):

    # Parse States
    theta_hat = z[0]
    omega_T_hat = z[1]
    omega_B_hat = z[2]
    Sig = np.matrix((z[3:]).reshape(3,3))

    # Interpolate input signal data
    iTorq = interp(it, t, Torq)
    iy_m = interp(it, t, y_m)

    # Compute Jacobians
    F = np.matrix([[0, -1, -1], [(-k1-3*k2*theta_hat**2)/J_T, -b/J_T, 0], [(k1
    +3*k2*theta_hat**2)/J_T, 0, -b/J_B]])# YOU DERIVE THESE
    H = C# YOU DERIVE THESE

    # Compute Kalman Gain
    L = Sig @ H.T / N

    # Compute EKF system matrices
    y_hat = omega_T_hat

    f = np.matrix([[omega_T_hat - omega_B_hat], [(-k1 * theta_hat - k2 *
    theta_hat ** 2 - b * omega_T_hat + iTorq) / J_T], [(k1 * theta_hat + k2 *
    theta_hat ** 2 - b * omega_B_hat) / J_B]])
    x_dot = f + L * (iy_m - omega_T_hat)

    theta_hat_dot = x_dot[0]
    omega_T_hat_dot = x_dot[1]
    omega_B_hat_dot = x_dot[2]

    # Riccati Equation
    Sig_dot = Sig @ F.T + F @ Sig + W - Sig @ H.T / N @ H @ Sig

    # Concatenate LHS
    z_dot = np.r_[theta_hat_dot, omega_T_hat_dot, omega_B_hat_dot, Sig_dot.
    reshape(9,1)]

```

```
    return(np.squeeze(np.asarray(z_dot)))

# Integrate Extended Kalman Filter ODEs
z = odeint(ode_ekf, states0, t)

# Parse States
theta_hat = z[:,0]
omega_T_hat = z[:, 1]
omega_B_hat = z[:, 2]
Sig33 = z[:, -1]

omega_B_tilde = omega_B_true - omega_B_hat
omega_B_hat_upperbound = omega_B_hat + np.sqrt(Sig33)
omega_B_hat_lowerbound = omega_B_hat - np.sqrt(Sig33)

RMSE = np.sqrt(np.mean(np.power(omega_B_tilde,2)))
print('Extended Kalman Filter RMSE: ' + str(RMSE) + ' rad/s')


# Plot Results
plt.figure(num=2, figsize=(8, 9), dpi=150, facecolor='w', edgecolor='k')

plt.subplot(2,1,1)
#   Plot true and estimated bit velocity
#   Plot estimated bit velocity plus/minus one sigma
plt.plot(t, omega_B_hat, 'r', label=r'Estimated velocity $\hat{\omega}_B$')
plt.plot(t, omega_B_true, 'g', label=r'True velocity $\omega_B$')
plt.plot(t, omega_B_hat_upperbound, 'm--', label=r'Upper bound $\hat{\omega}_B+\sqrt{\Sigma_{33}}$')
plt.plot(t, omega_B_hat_lowerbound, 'b--', label=r'Lower bound $\hat{\omega}_B-\sqrt{\Sigma_{33}}$')
plt.legend()
plt.xlabel('Time')
plt.ylabel('Bit velocity')

plt.subplot(2,1,2)
#   Plot error between true and estimated bit velocity
plt.plot(t, omega_B_tilde)
plt.xlabel('Time')
plt.ylabel('Estimation error')

plt.show()
```

```
Extended Kalman Filter RMSE: 0.0458628285791 rad/s
```

The result of EKF is shown in Figure 4.

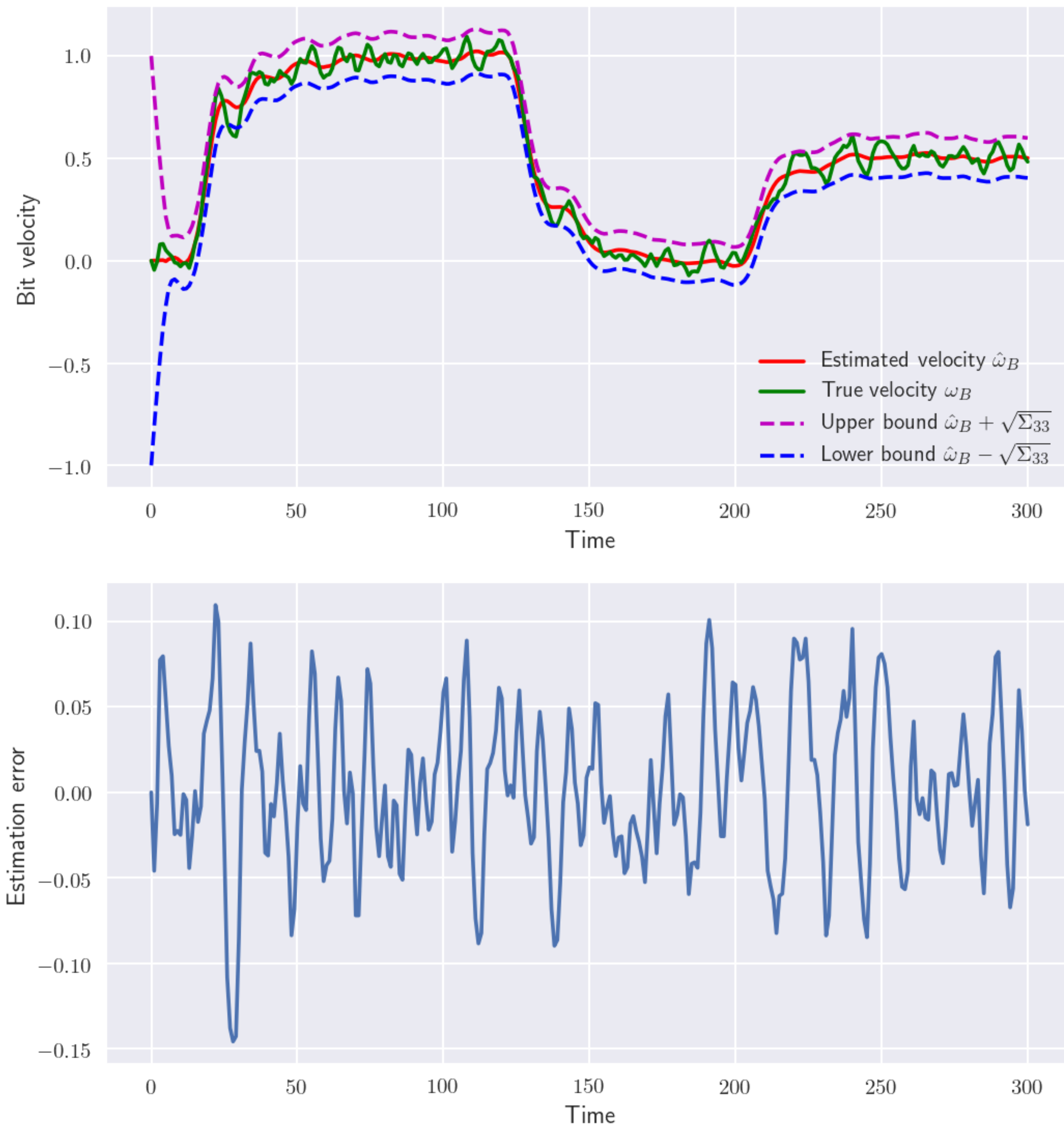

Fig. 4: EKF result