Estimation Theory & Kalman Filtering - Final Report

Michael Mason and Teal Hobson-Lowther

May 4, 2015

Abstract

We outline a simple model for a balancing robot, with the goal of estimating its angle and angular velocity so it can be controlled to balance successfully. Once the model is established, an Unscented Kalman Filter (UKF) is implemented. It is used to estimate the internal states of the balancing robot under a variety of inputs. Certain model parameters that are unknown are estimated. The estimator performs well in all scenarios and is able to estimate the unknown parameters accurately.

1 Introduction

An attempt will be made to estimate the state of a balancing robot, similar to a Segway. We model the system as a double weighted pendulum, with a basic representation shown in Figure 1. We have a motor which applies a torque u to the base. Measurements are obtained from a gyroscope and an accelerometer, which is placed a distance l_a from the base.

The gyroscope measures the angular velocity, $\dot{\theta}$, of the system and the accelerometer measures the acceleration in the x-direction of the device. Both the gyroscope and the accelerometer readings are noisy and contain time-varying bias terms, making the Kalman Filter an obvious choice for a state estimation method.

$$\ddot{\theta} = \frac{\left(g\left(m_1+m_2\right) - m_1 l\cos\theta\left(\dot{\theta}\right)^2\right)\sin\theta + \cos\left(\theta\right)u}{l\left(m_1+m_2-m_1\left(\cos\theta\right)^2\right)} \tag{1}$$

$$a = \frac{l-l_a}{l}\left(\frac{\cos\theta - m_1 l\cos\theta\left(\dot{\theta}\right)^2\sin\theta}{\left(m_1+m_2-m_1\left(\cos\theta\right)^2\right)}u + \frac{m_1+m_2}{\left(m_1+m_2-m_1\left(\cos\theta\right)^2\right)}g\sin\theta\right) \tag{2}$$

The system will be estimated using a state estimation vector at time k given as $\hat{x}_k = \begin{vmatrix} \theta_k & \dot{\theta}_k & b_k^g & b_k^a \end{vmatrix}$. In the cases of uncertain length, we augment the state to include l_k in the fifth element. We establish an output vector $y_k = \begin{vmatrix} y_k^g & y_k^a \end{vmatrix}$, where y_k^g and y_k^a represent the gyroscope and accelerometer readings, respectively. Our input at time k is simply u_k, which is assumed to be directly available via measurement. The discrete-time state and output equations are given in (3) and (4), respectively. We can solve for both the rate of change of $\dot{\theta}$ with respect to time and the acceleration in terms of our state variables via equations (1) & (2). Once these estimates are obtained, we would theoretically feed these states into a controller, which would allow the robot to be self-balancing. However, this is beyond the scope of this project.

The system parameters are given in Table 1. Note that initially l is a constant, but in later stages we implement a model that can estimate a time-varying l.

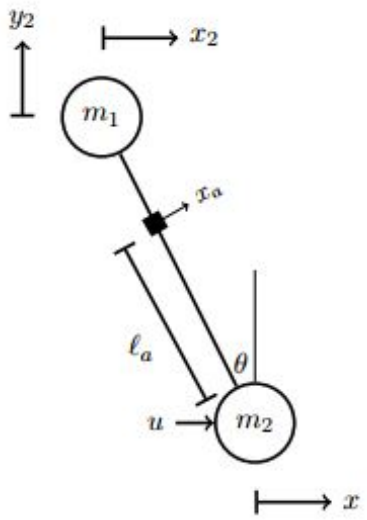


Figure 1: Symbolic model of system. Source: [1]

Table 1: System parameters

l	.38	m
l_a	.2	m
m_1	.6	kg
m_2	1.9	kg

$$x_{k+1} = f(x_k, u_k) + w_k \tag{3}$$

$$\begin{aligned} y_k^g &= \dot{\theta}(kT_s) + v_k^g + n_k^g \\ y_k^a &= a(kT_s) + v_k^a + n_k^a \end{aligned} \tag{4}$$

Section 2 contains the mathematical descriptions of the Unscented Kalman Filter and some motivation behind the method (Section 2.1), and the validation of this method on test data with time-varying parameters is demonstrated (Section 2.2).

Section 3 shows the results of applying our Unscented Kalman Filter to two separate data-sets, generated by a third party, with some unknown parameters to be estimated. The first data-set was generated using a constant length, and time-varying gyroscope and accelerometer biases (Section 3.1). The second data-set was generated using a time-varying length, gyroscope bias, and accelerometer bias (Section 3.2).

Section 4 contains the conclusions we have drawn from the implementation of the Unscented Kalman Filter, and illustrates the important results of Section 3.

2 Methods

We tackle the problem of state-estimation using a simulation method called the Unscented Kalman Filter (UKF). Simulation methods can lead to more robust estimators. They are applied when dealing with non-linear system dynamics. With simulation methods, some number of sample states are generated from the distribution representing the current state estimate and evolved through time using system dynamics. The mean and covariance of the evolved samples characterize the Gaussian distribution at the next time-step. The UKF chooses its (fewer) sample points intelligently, which makes it more computationally tractable when compared to other simulation methods, such as Monte-Carlo. The sample points of the UKF are chosen such that the mean and covariance of the samples is exactly the same as that of the current estimate.

2.1 Unscented Kalman Filter

We use an Euler approximation to discretize the system during the time update, which can be seen in Equation (6). Equations (5) and (6) describe the time and measurement updates for the Unscented Kalman Filter, respectively.

Unscented Kalman Filter Measurement-Update Equations:

$$
\begin{aligned}
P_k^{(-)} &= MM^T \\
E &= \sqrt{n}\,\big|\; I \quad -I \;\big| \\
\tilde{X} &= ME \\
\hat{X}_k^{(-)} &= \big|\; \hat{x}_k^{(-)} \quad \dots \quad \hat{x}_k^{(-)} \;\big| \\
X_k &= \hat{X}_k^{(-)} + \tilde{X} \\
X_k &= \big|\; x_k^{(1)} \quad \dots \quad x_k^{(2n)} \;\big| \\
y^{(i)} &= g(x_k^{(i)}, u_k) \\
\bar{y} &= \tfrac{1}{2n}\textstyle\sum_{i=1}^{2n} y^{(i)} \\
P_{xy} &= \tfrac{1}{2n}\textstyle\sum_{i=1}^{2n} (x_k^{(i)} - \hat{x}_k^{(-)})(y^{(i)} - \bar{y})^T \\
P_y &= \tfrac{1}{2n}\textstyle\sum_{i=1}^{2n} (y^{(i)} - \bar{y})(y^{(i)} - \bar{y})^T + R \\
K &= P_{xy} P_y^T \\
x_k^{(+)} &= \hat{x}_k^{(-)} + K(y - \bar{y}) \\
P_k^{(+)} &= P_k^{(-)} - K P_{xy}^T
\end{aligned}
\tag{5}
$$

Unscented Kalman Filter Time-Update Equations:

$$
\begin{aligned}
P_k^{(+)} &= MM^T \\
E &= \sqrt{n}\,\big|\; I \quad -I \;\big| \\
\tilde{X} &= ME \\
\hat{X}_k^{(+)} &= \big|\; \hat{x}_k^{(+)} \quad \dots \quad \hat{x}_k^{(+)} \;\big| \\
X_k &= \hat{X}_k^{(+)} + \tilde{X} \\
X_k &= \big|\; x_k^{(1)} \quad \dots \quad x_k^{(2n)} \;\big| \\
f(x_k^{(i)}, u_k) &= \begin{vmatrix} \theta_k^{(i)} + \dot{\theta}_k^{(i)} T_s \\ \dot{\theta}_k^{(i)} + \ddot{\theta}(\theta_k^{(i)}, \dot{\theta}_k^{(i)}, u_k) T_s \end{vmatrix} \\
x_{k+1}^{(i)} &= \begin{vmatrix} f(x_k^{(i)}, u_k) \\ b_k^g \\ b_k^a \\ l \end{vmatrix}
\end{aligned}
\tag{6}
$$

2.2 Validation

Before we attempt to work with the unknown data-sets provided by Dr. Vincent, we endeavor to verify our estimator by applying it to several data-sets that we generated. Figure 5 shows our estimates of an uncontrolled system, which is wildly oscillating due to an approximately resonant input. This makes the length, if unknown, easy to deduce due to the large role that it plays in the dynamics. In Figure 5a, the length is fixed and known. In Figure 5b, the length is fixed but unknown. We make a poor initial guess to see if the estimator can converge to the true length. In both Figures 5a & 5b, the biases and states are determined accurately. The covariance matrices used to provide this result are found in (7) and (8), respectively.

Covariance Matrices for Constant Length Model:

$$
P_0 = \begin{vmatrix} .01 & 0 & 0 & 0 \\ 0 & .01 & 0 & 0 \\ 0 & 0 & .01 & 0 \\ 0 & 0 & 0 & .01 \end{vmatrix}
$$

$$
Q = \begin{vmatrix} 1\text{E}-7 & 0 & 0 & 0 \\ 0 & 1\text{E}-7 & 0 & 0 \\ 0 & 0 & .0001T_s & 0 \\ 0 & 0 & 0 & .0001T_s \end{vmatrix}
$$

$$
R = \begin{vmatrix} .0001 & 0 \\ 0 & .0001 \end{vmatrix} \tag{7}
$$

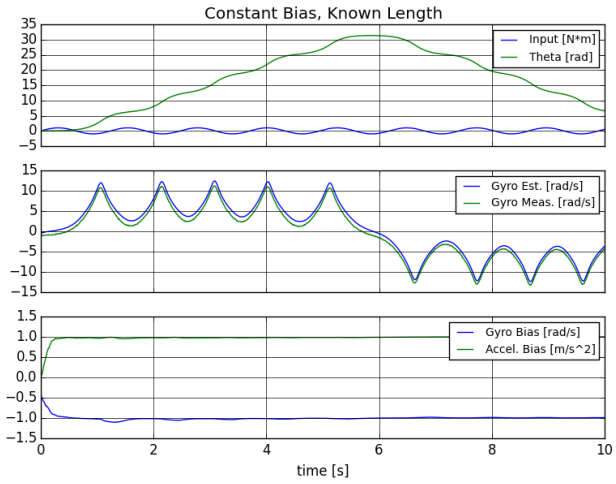
Covariance Matrices for Uncertain Length Model:

$$
P_0 = \begin{vmatrix} .01 & 0 & 0 & 0 & 0 \\ 0 & .01 & 0 & 0 & 0 \\ 0 & 0 & .01 & 0 & 0 \\ 0 & 0 & 0 & .01 & 0 \\ 0 & 0 & 0 & 0 & .01 \end{vmatrix}
$$

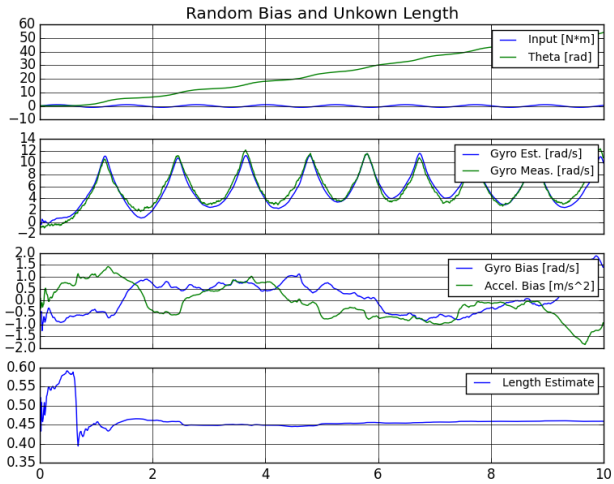
$$
Q = \begin{vmatrix} 1\text{E}-7 & 0 & 0 & 0 & 0 \\ 0 & 1\text{E}-7 & 0 & 0 & 0 \\ 0 & 0 & .0001T_s & 0 & 0 \\ 0 & 0 & 0 & .0001T_s & 0 \\ 0 & 0 & 0 & 0 & 0 \end{vmatrix}
$$

$$
R = \begin{vmatrix} .0001 & 0 \\ 0 & .0001 \end{vmatrix} \tag{8}
$$

The data-sets provided to us for analysis represent a controlled balancing robot. As such, there are very small fluctuations in the angle and angular speed. To create a data-set with similarly fluctuating conditions, a small



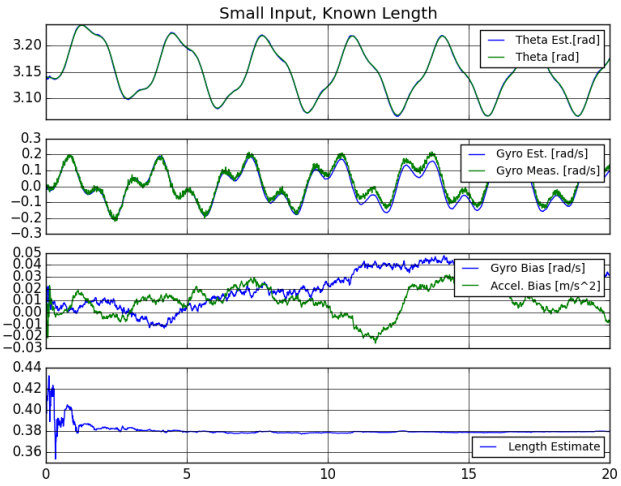
(a) The length of the pendulum is fixed and known to be 0.38 m. The biases are fixed at -1 N*m for the gyro and +1 $\frac{m}{s^2}$ for the accelerometer. We accurately estimate both of the biases in this case



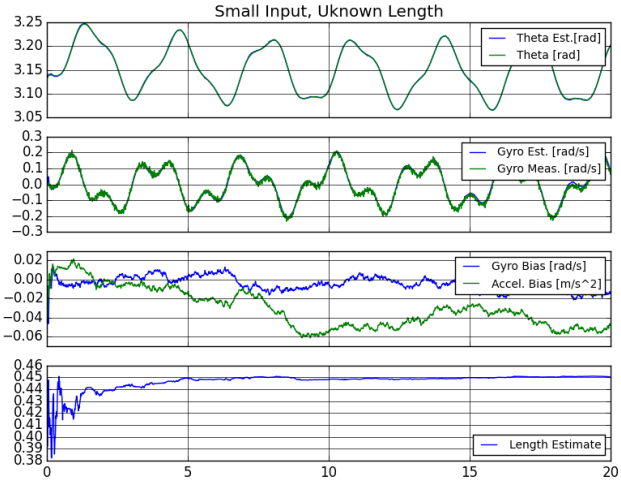
(b) The true length of the pendulum is 0.45m, but the estimator guesses an initial length of 0.38m. Note that the length estimate converges to the true value, despite a poor initial guess

Figure 5: Validating data on pendulum with an approximately resonant input

input was applied to the pendulum when $\theta = \pi$ (pendulum oriented straight down). The results of this experiment are plotted in Figures 6a & 6b. Both estimates are nearly identical to the ground truth and the length estimates converge accurately.



(a) The true length, l, is known to be 0.38m. The length estimate converges to this quickly



(b) The true length, l, is 0.45m, but we guess 0.38m initially. The length estimate converges to 0.45m regardless of this poor initial guess

Figure 6: Data is simulated using small perturbations in u, to ensure our estimator can track scenarios similar to those of "dataset1" and "dataset2"

Thus emboldened by several successful estimations with known ground-truth, we move on to uncharted territory: the analysis of the provided data-sets.

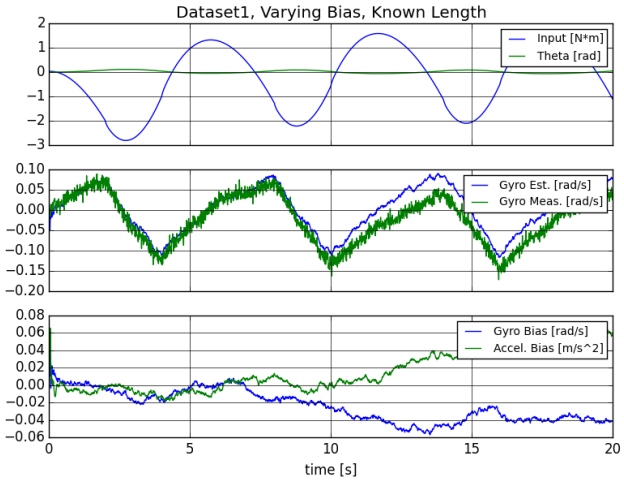
3 Results

3.1 Data-set 1

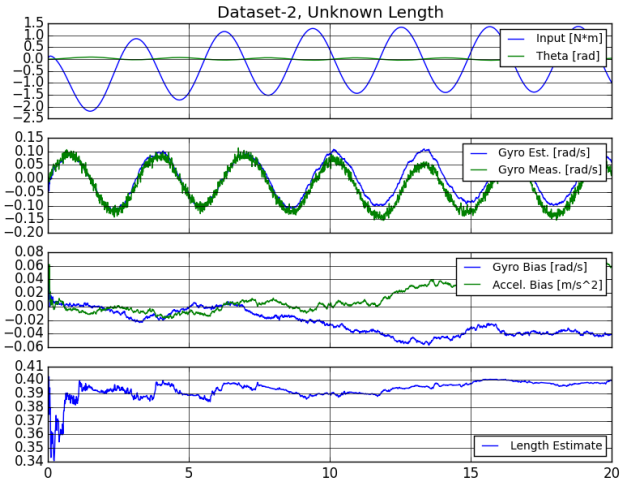
We began by applying our estimator to "dataset1". The length of the pendulum is known to be 0.38 m. The results of this estimation can be seen in Figure 7a. The states behave as we might expect controlled states to, in that both the angle and angular velocity stay relatively small. Generally, the input has the opposite sign of the angle, which makes intuitive sense from a control standpoint. A good controller would create an artificial equilibrium point about $\theta = 0$. The covariance matrices used for this simulation are given in (7).

3.2 Data-set 2

The second data-set has an additional challenge: it was generated with an unknown length. This length is modeled as an unknown constant. The results of our estimation are summarized in Figure 7b. The states follow a reasonable trajectory, similar to those of the first data-set. The length estimate also converges to a value of 0.4m. However, the convergence is not as convincing as in the validation trials. The covariance matrices used for this simulation are found in (8).



(a) "Dataset1" state estimation. θ stays relatively close to zero, and the gyroscope reading never grows larger than $0.1\frac{rad}{sec}$. The biases appear to be random walks with time-dependent variance



(b) "Dataset2" state estimation. θ stays relatively close to zero, as does the gyroscope measurement. The length is estimated to converge to approximately 0.4m

Figure 7: Estimator applied to the given data-sets. The gyroscope and accelerometer biases appear to be the same in both (a) and (b), which conforms to our a-priori knowledge of the data-sets

4 Conclusions

Using the UKF, we were able to verifiably and consistently estimate the states of the balancing robot in a variety of scenarios. The length estimation for the second data-set took most of the twenty seconds to converge to a value of ≈0.4m. One possible explanation for this long and therefore less convincing convergence is that other parameters, such as the placement of the accelerometer, could have also been altered. To determine this, we could augment our state for other physical parameters, such as l_a, and see if any estimates indicate that a parameter is different from its nominal value. If this augmentation does not produce satisfactory results, we would be forced to assume that the length is time-varying. Accounting for this possibility would require an alteration of our model parameters, namely the fifth diagonal entry in our Q matrix, given in (8). Generally speaking, the UKF seemed to be a good choice for both accuracy and speed of estimation. By re-implementing the UKF in python, we shortened the run-time to be just over one second. This time is particularly noteworthy due to its stark contrast to the

original MATLAB implementation's run-time of over 20 minutes. The obvious next step would to be to use our state estimates as the input for a controller and create a simulation of our own self-balancing robot.

References

[1] T. Vincent. EENG 519. Project Outline, Topic: "Project Details" Faculty of Electrical Engineering, Colorado School of Mines, Golden, Colorado, May 3, 2015.

[2] T. Vincent. EENG 519. Lecture 18, Topic: "Extended Kalman Filtering and Moving Horizon Estimation" Faculty of Electrical Engineering, Colorado School of Mines, Golden, Colorado, April 15, 2015.

[3] T. Vincent. EENG 519. Lecture 19, Topic: "Simulation Methods" Faculty of Electrical Engineering, Colorado School of Mines, Golden, Colorado, April 20, 2015.

Appendices

A UKF Implementation - Python

```
n = 5
E =  np.concatenate((math.sqrt(n)*np.eye(n),-math.sqrt(n)*np.eye(n)),axis = 1)
xHat = np.zeros((n,len(u)))
yBar = np.zeros((2,len(u)))
y = np.zeros((2,n*2))
P = .01 * np.eye(n)
Q = np.zeros((n,n))
Q[0,0] = .0000001
Q[1,1] = .0000001
Q[2,2] = .000001
Q[3,3] = .000001
R = .0001 * np.eye(2)
m1 = .6
m2 = 1.9
la = .2
Ts = .01
xHat[4,0] = .4
def f(state,u,steps = 10):
        th = state[0]
        thDot = state[1]
        l = state[4]
        thDotDot = ((9.81*(m1+m2)-m1*l*math.cos(th)*thDot**2)*math.sin(th)+math.cos(th)*u)/ \
    (l*(m1+m2-m1*math.cos(th)**2))
        return np.array([th+thDot*Ts,thDot+thDotDot*Ts,state[2],state[3],state[4]])
def g(state,u):
        th = state[0]
        thDot = state[1]
        l = state[4]
        a = (l-la)/l*((math.cos(th)-m1*l*math.cos(th)*thDot**2*math.sin(th))*u/(m1+m2-m1*math.cos(th)**2)+ \
        (m1+m2)*9.81*math.sin(th)/(m1+m2-m1*math.cos(th)**2))
        return np.array([thDot+state[2],a+state[3]])
for k in xrange(len(u)-1):
        #measurement update
        U, s, V = np.linalg.svd(P, full_matrices=True)
        M = np.dot(U,np.diag(np.sqrt(s)))

        xTilde = np.dot(M,E)
        xTilde += xHat[:,k:k+1]

        for i in xrange(2*n):
                y[:,i] = g(xTilde[:,i],u[k])

        yBar[:,k] = np.mean(y,1)

        PXY = 1.0/(2.0*n)*np.dot(xTilde - xHat[:,k:k+1],np.transpose(y - yBar[:,k:k+1]))
        PY = 1.0/(2.0*n)*np.dot(y - yBar[:,k:k+1],np.transpose(y-yBar[:,k:k+1]))+R
        K = np.dot(PXY,np.linalg.inv(PY))

        yMeas = np.reshape(np.array([gyro[k],accel[k]]),(2,1))

        xHat[:,k:k+1] = xHat[:,k:k+1] + np.dot(K,yMeas-yBar[:,k:k+1])
        P = P - np.dot(K,np.transpose(PXY))

        #time update
        U, s, V = np.linalg.svd(P, full_matrices=True)
        M = np.dot(U,np.diag(np.sqrt(s)))

        xTilde = np.dot(M,E)
        xTilde += xHat[:,k:k+1]

        for i in xrange(2*n):
                xTilde[:,i] = f(xTilde[:,i],u[k])
        xHat[:,k+1] = np.mean(xTilde,1)
        P = 1.0/(2.0*n)*np.dot(xTilde - xHat[:,k+1:k+2],np.transpose(xTilde - xHat[:,k+1:k+2])) + Q
```