State estimation of a Quadrotor

Ramprasad Rajagopalan
MS in Mechanical Engineering
Colorado School of Mines
Golden, USA
rrajagopalan1@mymail.mines.edu

***Abstract*—The project deals with the state estimation of a X-configuration quadrotor using Extended Kalman Filter (EKF) and Unscented Kalman Filter (UKF) based on Gyroscopes and Global Positioning System (GPS) measurements. The quadrotor system was modelled in Simulink and the filters were programmed in MATLAB. The sensors were added with measurement noise and process noise. The filter model was tested for two different operating conditions and the robustness of each was evaluated. The results shows that the performance of EKF in estimating the staes of the quadrotor is comparatively better than UKF for the given filter model.**

***Index Terms*—Extended Kalman filter, Unscented Kalman filter, Quadrotor, state estimation**

I. Introduction

The system chosen is an X configuration quadrotor with the co-ordinates frames and labels given in Fig.1. A quadrotor has 4 rotors controlled by a flight controller and mounted on 4 arms. 2 rotors on diagonally opposite side rotates in clockwise direction while 2 other rotors rotate in anticlockwise direction. Fig.1 shows the body frame x - axis x_B and y - axis y_B and inertial frame x - axis $x_{Inertial}$ and y - axis $y_{Inertial}$. The forward direction of the quadrotor is x_B and roll about x_B is given by ϕ. The pitch θ is rotation about y_B and yaw ψ is rotation about z_B. Each rotating propeller produces an upward force given by,

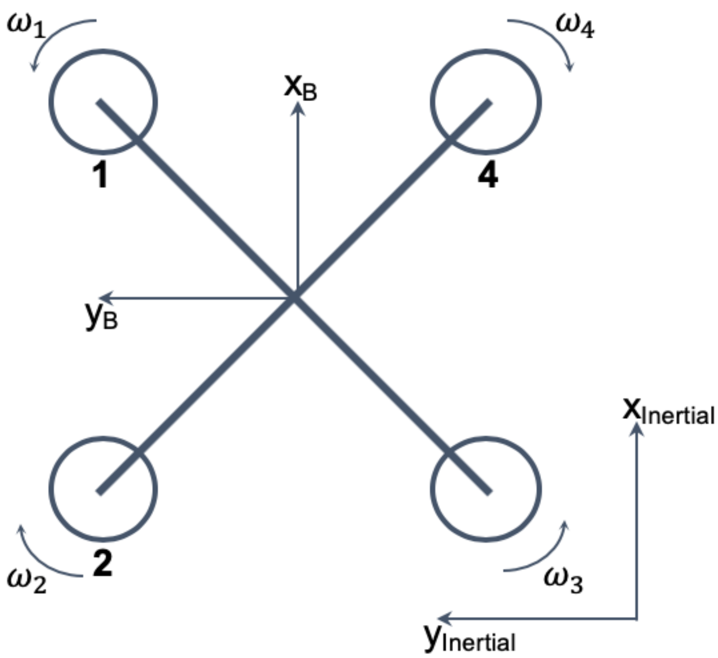

Fig. 1. Quadrotor co-ordinate frame

$$F_i = k_f {\omega_i}^2 \tag{1}$$

where F_i is the thrust generated by each propeller, k_f is the thrust factor and ω_i is the anuglar velocity of each propeller. Apart from upward thrust, a rotating propeller also produces an opposing moment given by,

$$M_i = k_m {\omega_i}^2 \tag{2}$$

where M_i is the moment generated by each propeller about z_B and k_m is the drag factor. The dynamic equations of the system is derived as follows. The total thrust from the 4 rotating propellers is given by,

$$F_{total} = k_f \left({\omega_1}^2 + {\omega_2}^2 + {\omega_3}^2 + {\omega_4}^2\right) \tag{3}$$

The linear acceleration equations are obtained as,

$$\begin{bmatrix} \ddot{x} \\ \ddot{y} \\ \ddot{z} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ -g \end{bmatrix} + \frac{R}{mass} \begin{bmatrix} 0 \\ 0 \\ F_{total} \end{bmatrix} \tag{4}$$

where $R = R_z(\psi)R_y(\theta)R_x(\phi)$ is the rotation matrix from world frame to body frame obtained from corresponding rotation along the body axes. The moments along the respective axes are given by,

$$\tau_x = \frac{Lk_f}{\sqrt{2}}\left({\omega_1}^2 + {\omega_2}^2 - {\omega_3}^2 - {\omega_4}^2\right) \tag{5}$$

$$\tau_y = \frac{Lk_f}{\sqrt{2}}\left(-{\omega_1}^2 + {\omega_2}^2 + {\omega_3}^2 - {\omega_4}^2\right) \tag{6}$$

$$\tau_z = k_m\left({\omega_1}^2 - {\omega_2}^2 + {\omega_3}^2 - {\omega_4}^2\right) \tag{7}$$

The body frame angular accelerations are given by the Euler's equation for rigid body dynamics,

$$\begin{bmatrix} \ddot{\phi} \\ \ddot{\theta} \\ \ddot{\psi} \end{bmatrix} = I^{-1}\left(\begin{bmatrix} \tau_x \\ \tau_y \\ \tau_z \end{bmatrix} - \omega \times I\omega\right) \tag{8}$$

where $I = \begin{bmatrix} I_{xx} & 0 & 0 \\ 0 & I_{yy} & 0 \\ 0 & 0 & I_{zz} \end{bmatrix}$ and $\omega = \begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix}$ in the body frame. Therefore, the equations of motion from the above relations are written as follow,

$$\ddot{x} = \frac{F_{total}(cos(\phi)sin(\theta)cos(\psi) + sin(\phi)sin(\psi))}{mass} \tag{9}$$

$$\ddot{y} = \frac{F_{total}(cos(\phi)sin(\theta)sin(\psi) - sin(\phi)cos(\psi))}{mass} \quad (10)$$

$$\ddot{z} = \frac{F_{total}(cos(\phi)cos(\theta))}{mass} - 9.81 \quad (11)$$

$$\ddot{\phi} = \frac{\tau_x + (I_{yy} - I_{zz})\dot{\theta}\dot{\psi}}{I_{xx}} \quad (12)$$

$$\ddot{\theta} = \frac{\tau_y + (I_{zz} - I_{xx})\dot{\phi}\dot{\psi}}{I_{yy}} \quad (13)$$

$$\ddot{\psi} = \frac{\tau_z + (I_{xx} - I_{yy})\dot{\theta}\dot{\phi}}{I_{zz}} \quad (14)$$

Based on the above relations, one can derive the continuous states of the quadrotor given an input angular velocity of the propeller or the thrust and moments about corresponsding axes. The state of the quadrotor i.e., position and orientation can be obtained through these relation. Although the former can be measured using a laser tracking or GPS in an environment, the latter cannot be physically measured but rather needs to be estimated. It is given that the laser tracking or GPS measurement includes noise and it is not the absolute measurement.

The objective of this study is to estimate position of the quadrotor by filtering the noise in GPS measurement and also to estimate the orientation value which cannot be directly measured. The quadrotor is modelled in Simulink and simulated for 2.5s with time step, T_s, 0.1s.

II. Methodology

A. State-space model of the system

The continuous time state-space model of the quadrotor is obtained by by defining the state vector of the system,X, as follows,

$$X = \begin{bmatrix} x & y & z & \phi & \theta & \psi & \dot{x} & \dot{y} & \dot{z} & \dot{\phi} & \dot{\theta} & \dot{\psi} \end{bmatrix}$$

$$u = \begin{bmatrix} F_{total} & \tau_x & \tau_y & \tau_z \end{bmatrix}$$

where u is the input vector. Then, the continuous time state-space model can be written as,

$$\dot{X} = f_x(X, u) + w$$

where $f_x(X, u)$ is the non-linear function matrix mapping the first order derivative states to the system states and w is the process noise or disturbance in input. The non-linear function matrix, $f_x(X, u)$, can obtained from Eqn.9-14.

The process noise or input disturbance, $w \sim \mathcal{N}(\bar{x}, \sigma_w)$, is assumed to be a Gaussian distribution with mean, $\bar{x} = 0$ and variance, $\sigma_w = 1e-6$. The system variables that are assumed as given in Table.I.

TABLE I
SYSTEM VARIABLES

Parameter	Value
Mass	0.25 kg
I_{xx}	0.03 kgm^2
I_{yy}	0.03 kgm^2
I_{zz}	0.03 kgm^2
L	0.1 m
k_f	0.1
k_m	0.1

B. Sensor model

The sensors chosen for measurement are GPS and Gyroscope. The GPS measures the position of the quadrotor in the world frame. The GPS sensor is modelled as follows,

$$y_k{}^{\text{GPS}} = r(kT_s) + n_k{}^{\text{GPS}}$$

where $y_k{}^{\text{GPS}}$ is the GPS measurement, $r(kT_s)$ is the actual position of quadrotor in world frame and $n_k{}^{\text{GPS}}$ is the iid measurement noise. $n_k{}^{\text{GPS}}$ is assumed to be Gaussian distribution i.e., $n_k{}^{\text{GPS}} \sim \mathcal{N}(0, \sigma_{GPS})$. The variance σ_{GPS} is assumed to be 0.5. The Gyroscope measures the angular velocities in body frame and it is modelled as follows,

$$y_k{}^{\text{Gyro}} = \dot{\Omega}(kT_s) + n_k{}^{\text{Gyro}}$$

where $y_k{}^{\text{Gyro}}$ is the gyroscope measurement, $\dot{\Omega}(kT_s)$ is the actual angular velocity of quadrotor in body frame and $n_k{}^{\text{Gyro}}$ is the iid measurement noise. $n_k{}^{\text{Gyro}}$ is assumed to be Gaussian distribution i.e., $n_k{}^{\text{Gyro}} \sim \mathcal{N}(0, \sigma_{Gyro})$. The variance σ_{Gyro} is assumed to be 0.1.

C. Extended Kalman Filter model

The EKF was chosen to filter the sensor measurements and estimate the states of the quadrotor. As the GPS measurement values had noise, the noise in the measurement is needed to be filtered out. The orientation of the quadcopter cannot be directly measured, so it is the unobservable state in our system. This is estimated by EKF based on the gyroscope measurements which in turn has noise that needs to be filtered out. The EKF model is given as follows,

$$\dot{X} = f_x(X, u) + w$$

$$Y = HX + v$$

where Y is the sensor measurement given by $\begin{bmatrix} x_{GPS} & y_{GPS} & z_{GPS} & \dot{\phi}_{gyro} & \dot{\theta}_{gyro} & \dot{\psi}_{gyro} \end{bmatrix}$,

$$H = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

v is the measurement noise with distribution $v \sim \mathcal{N}\left(\begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} \sigma_{GPS} & 0 \\ 0 & \sigma_{gyro} \end{bmatrix} \right)$ The update equations

of the EKF are given as follows,
TIME UPDATE:

$$\Phi_k = \nabla_x f(\hat{x}_k{}^{(+)}, u_k) = \frac{\partial f_x(\hat{x}_k{}^{(+)}, u_k)}{\partial x} T_s + I$$

$$\hat{x}_{k+1}{}^{(-)} = f(\hat{x}_k{}^{(+)}, u_k)$$

$$P_{k+1}{}^{(-)} = \Phi_k P_k{}^{(+)} \Phi_k{}^{\mathrm{T}} + Q$$

MEASUREMENT UPDATE:

$$K_k = P_k{}^{(-)} H (H P_k{}^{(-)} H^{\mathrm{T}} + R)^{-1}$$

$$\hat{x}_k{}^{(+)} = \hat{x}_k{}^{(-)} + K_k (Y_k - H g_x(\hat{x}_k{}^{(-)}, u_k))$$

$$P_k{}^{(+)} = (I - K_k H) P_k{}^{(-)}$$

where, $Q = \sigma_w * eye(12)$ and $R \sim \mathbb{R}^{6\mathrm{X}6}$. The initial estimate $X_0{}^{(+)}$ was assumed to be zeros. The assumption made for initial covariance, $P_k{}^{(+)} = eye(12)$. The implementation of EKF in MATLAB is given in Appendix.A.

D. Unscented Kalman Filter

The UKF was implemented to compare the performance of EKF in estimating the states of the quadrotor. The UKF model is straight forward in approach and steps to estimate the states using the sensor measurements and updates are as follows,

TIME UPDATE:

$$x^{(\mathrm{i})} = \hat{x}^{(+)} + \widetilde{x}^{(\mathrm{i})}$$

$$x_{k+1}{}^{(\mathrm{i})} = f(x^{(\mathrm{i})}, u_k)$$

$$\hat{x}_{k+1}{}^{(-)} = \frac{1}{2n} \sum_{i=1}^{2n} x_{k+1}{}^{(\mathrm{i})}$$

$$P_{k+1}{}^{(-)} = \frac{1}{2n} \sum_{i=1}^{2n} (x_{k+1}{}^{(\mathrm{i})} - \hat{x}_{k+1}{}^{(-)})(x_{k+1}{}^{(\mathrm{i})} - \hat{x}_{k+1}{}^{(-)})^{\mathrm{T}} + Q$$

MEASUREMENT UPDATE:

$$\bar{Y} = \frac{1}{2n} \sum_{i=1}^{2n} Y^{(\mathrm{i})}$$

$$P_{xy} = \frac{1}{2n} \sum_{i=1}^{2n} (x_k{}^{(\mathrm{i})} - \hat{x}_k{}^{(-)})(Y^{(\mathrm{i})} - \bar{Y})^{\mathrm{T}}$$

$$P_y = \frac{1}{2n} \sum_{i=1}^{2n} (Y^{(\mathrm{i})} - \bar{Y})(Y^{(\mathrm{i})} - \bar{Y})^{\mathrm{T}} + R$$

$$K = P_{xy} P_y{}^{-1}$$

$$x_k{}^{(+)} = \hat{x}_k{}^{(-)} + K(Y_k - \bar{Y})$$

$$P_k{}^{(+)} = P_k{}^{(-)} - K P_{xy}{}^{\mathrm{T}}$$

The filter model was implemented using the same initial condition and measurement noise model as EKF. The implementation of UKF in MATLAB is given in Appendix.B.

III. RESULTS

The Simulink quadrotor model was simulated to make it pitch forward and move in the forward direction while climbing up. In order to make the quadrotor move in such a fashion, the following values for angular velocity of the propeller were given as input,

$$\omega_1 = 2.8 rad/s \quad \omega_4 = 2.8 rad/s$$
$$\omega_2 = 3.2 rad/s \quad \omega_3 = 3.2 rad/s$$

The plots obtained in Fig.2 and Fig.3 shows the comparison between the z estimate obtained from EKF and UKF implementation. The plots obtained in Fig.4 and Fig.5 shows the comparison between the x estimate obtained from EKF and UKF implementation. The plots obtained in Fig.6 and Fig.7 shows the comparison between the y estimate obtained from EKF and UKF implementation.

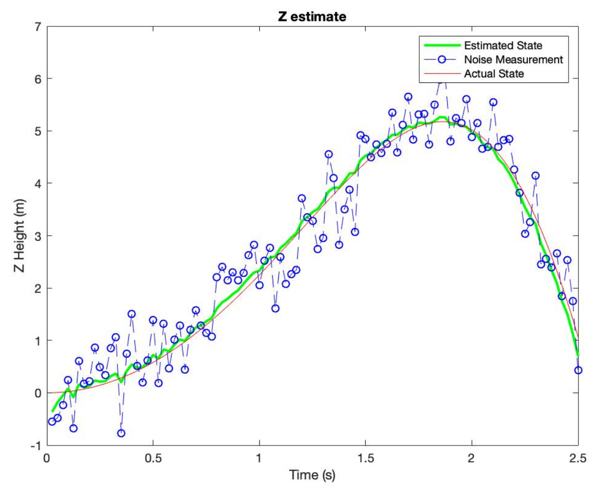

Fig. 2. z estimate using EKF

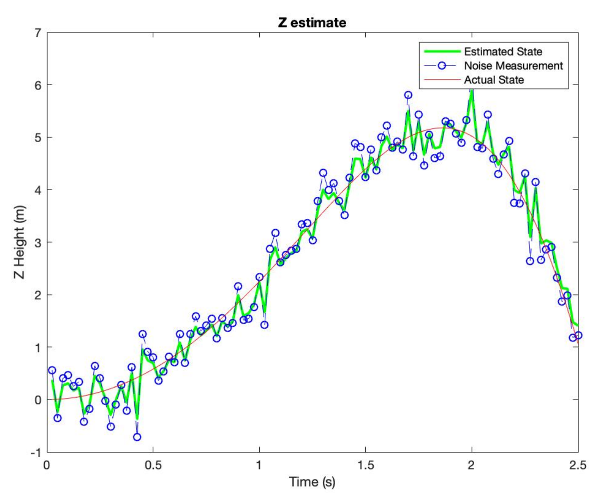

Fig. 3. z estimate using UKF

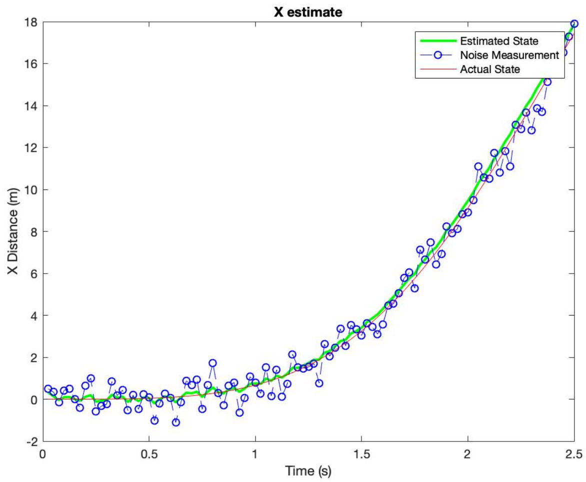

Fig. 4. x estimate using EKF

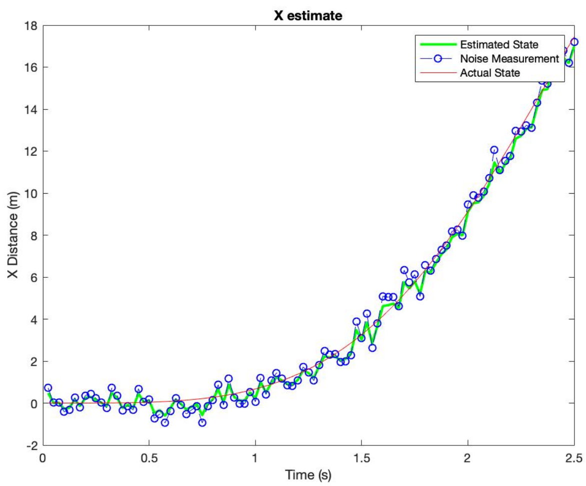

Fig. 5. x estimate using UKF

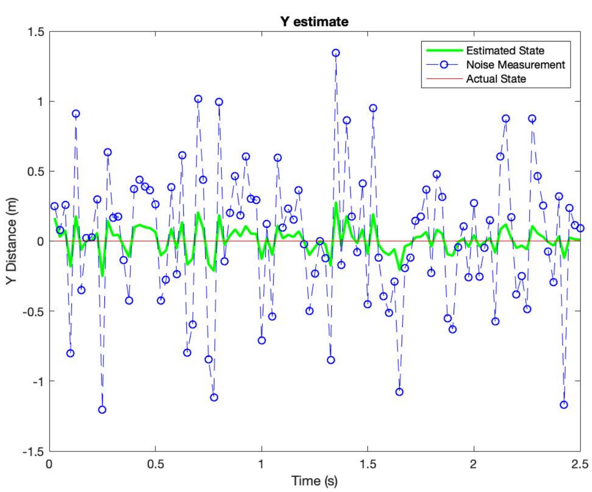

Fig. 6. y estimate using EKF

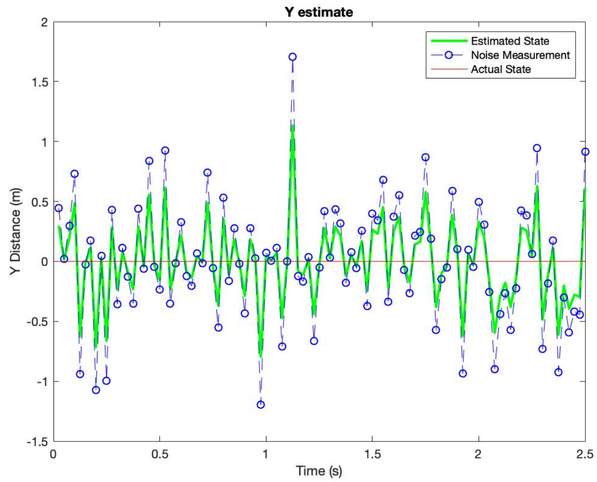

Fig. 7. y estimate using UKF

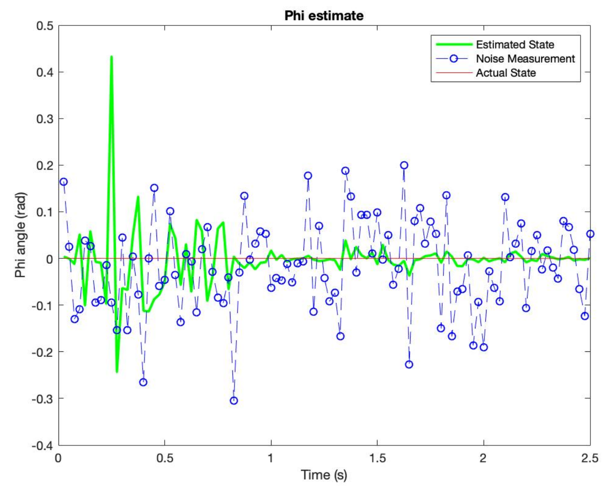

Fig. 8. ϕ estimate using EKF

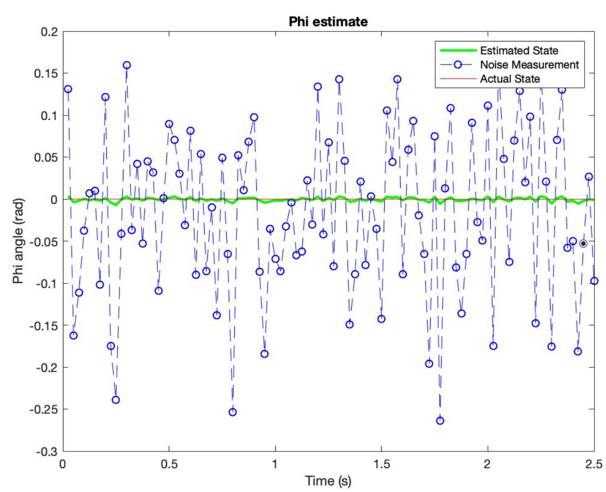

Fig. 9. ϕ estimate using UKF

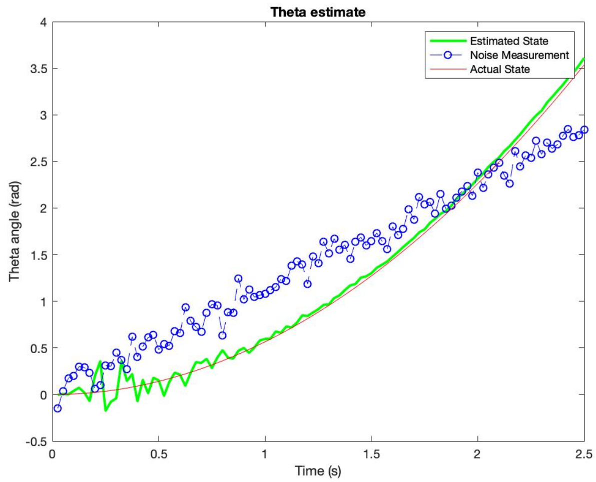

Fig. 10. θ estimate using EKF

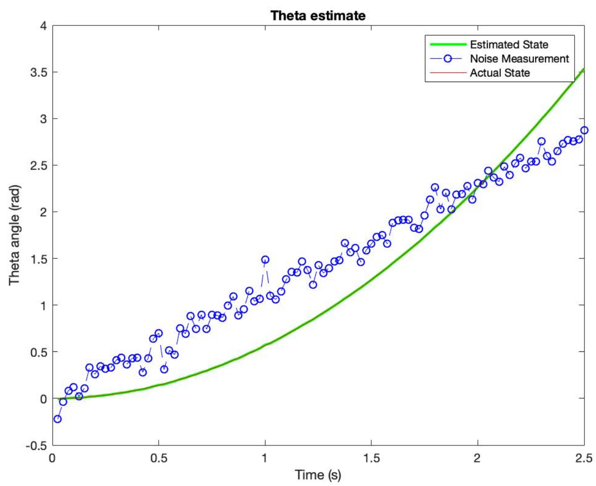

Fig. 11. θ estimate using UKF

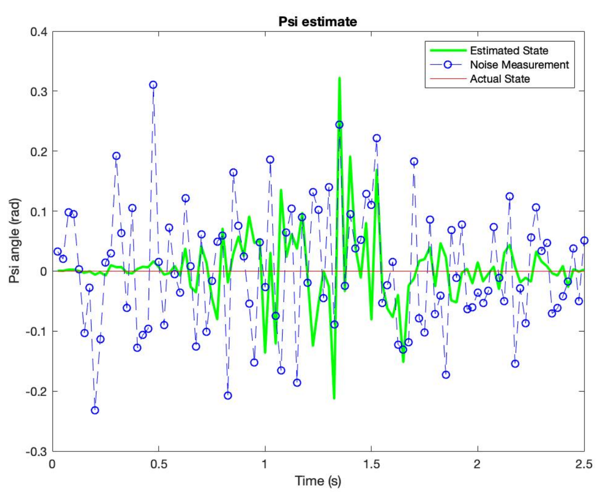

Fig. 12. ψ estimate using EKF

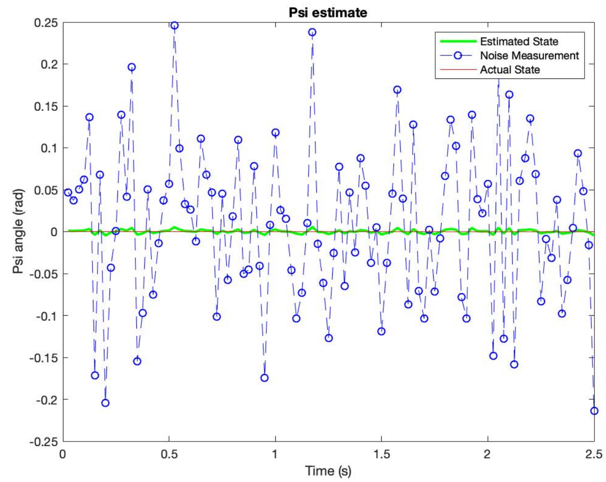

Fig. 13. ψ estimate using UKF

The plots obtained in Fig.8 and Fig.9 shows the comparison between the ϕ estimate obtained from EKF and UKF implementation. The plots obtained in Fig.10 and Fig.11 shows the comparison between the θ estimate obtained from EKF and UKF implementation. The plots obtained in Fig.12 and Fig.13 shows the comparison between the ψ estimate obtained from EKF and UKF implementation.

As from the above figures, we can see that EKF performs better in estimating the state of the quadrotor. it filters out the noise better than UKF. Although EKF performs better in estimating the position states of the quadrotor, UKF stands out while predicting the orientation states of the quadrotor. This is an interesting observation as more non linearity occurs in measuring the orientation angles. This serves to prove that UKF are better and easier to apply for more complex non-linear systems.

IV. Conclusion

The quadrotor was modelled in Simulink and filters viz., EKF and UKF were implemented in MATLAB. The study compared the performance of both filters in estimating the quadrotor's postion and orientation. EKF performed better at predicting the position with high noise levels. UKF was robust to sensitivities in measurement and performed better in estimating the orientation of the quadrotor. By adding, more sensor information viz., accelerometers, the states can be better estimated. The sensor model can be improved by including drift and bias in measurement to get a more accurate performance of both filter.

References

[1] T.Vincent, "Estimation theory and Kalman filtering".

[2] Kurak, Sevkuthan, and Migdat Hodzic. "Control and Estimation of a Quadcopter Dynamical Model." Periodicals of Engineering and Natural Sciences (PEN) 6.1 (2018): 63-75.

Appendix A
Extended Kalman Filter

```matlab
%% Extended Kalman Filter
x0 = 0.0;
y0 = 0.0;
z0 = 0.0;
xDot0 = 0.0;
yDot0 = 0.0;
zDot0 = 0.0;
phi0 = 0.0;
theta0 = 0.0;
psi0 = 0.0;
phiDot0 = 0.0;
thetaDot0 = 0.0;
psiDot0 = 0.0;
Ts = 0.025;
totT = 2.5;
sigmaGPS = 0.5;
sigmaGY = 0.1;
e = 1e-6;
x_k0 = [x0;y0;z0;phi0;theta0;psi0;...
    xDot0;yDot0;zDot0;phiDot0;thetaDot0;psiDot0];
count = 1;
t = Ts:Ts:totT;
P_k1 = eye(12);
Q = diag(e*ones(1,12));
R = eye(6);
R(1,1) = sigmaGPS*R(1,1);
R(2,2) = sigmaGPS*R(2,2);
R(3,3) = sigmaGPS*R(3,3);
R(4,4) = sigmaGY*R(4,4);
R(5,5) = sigmaGY*R(5,5);
R(6,6) = sigmaGY*R(6,6);
H = [1 0 0 0 0 0 0 0 0 0 0 0;...
    0 1 0 0 0 0 0 0 0 0 0 0;...
    0 0 1 0 0 0 0 0 0 0 0 0;...
    0 0 0 0 0 0 0 0 0 1 0 0;...
    0 0 0 0 0 0 0 0 0 0 1 0;...
    0 0 0 0 0 0 0 0 0 0 0 1];
simout = sim('quadcopter_control','StartTime',num2str(0),'StopTime',...
    num2str(Ts),'OutputOption','SpecifiedOutputTimes','OutputTimes',...
    num2str(Ts));
input = simout.Input.Data(end,:)';
del_x_kp1(:,count) = simout.simout.Data(end,:)';
yMeasurement(:,count) = simout.sensorMeasure.Data(end,:)';
yMeasurement(:,count) = yMeasurement(:,count) + [sigmaGPS*randn(3,1);...
    sigmaGY*randn(3,1)];
%
deltaF = zeros(12); % Derivative of the non-linear function matrix
%
F = zeros(12,1); % Non-linear function matrix
%
deltaF(1,7) = 1.0; % xDot
deltaF(2,8) = 1.0; % yDot
deltaF(3,9) = 1.0; % zDot
deltaF(4,10) = 1.0; % phiDot
deltaF(5,11) = 1.0; % thetaDot
deltaF(6,12) = 1.0; % psiDot
%
deltaF(7,4) = (input(1)/mass)*((-sin(phi0)*sin(theta0)*cos(psi0))+(cos(phi0)*sin(psi0)));
deltaF(7,5) = (input(1)/mass)*(cos(phi0)*cos(theta0)*cos(psi0));
deltaF(7,6) = (input(1)/mass)*((-cos(phi0)*sin(theta0)*sin(psi0))+(sin(phi0)*cos(psi0)));
%
deltaF(8,4) = (input(1)/mass)*((-sin(phi0)*sin(theta0)*sin(psi0))-(cos(phi0)*cos(psi0)));
deltaF(8,5) = (input(1)/mass)*(cos(phi0)*cos(theta0)*sin(psi0));
deltaF(8,6) = (input(1)/mass)*((cos(phi0)*sin(theta0)*cos(psi0))+(sin(phi0)*sin(psi0)));
%
deltaF(9,4) = (input(1)/mass)*(-sin(phi0)*cos(theta0));
deltaF(9,5) = (input(1)/mass)*(-cos(phi0)*sin(theta0));
%
deltaF(10,11) = ((iY-iZ)/iX)*psiDot0;
deltaF(10,12) = ((iY-iZ)/iX)*thetaDot0;
%
deltaF(11,10) = ((iZ-iX)/iY)*psiDot0;
deltaF(11,12) = ((iZ-iX)/iY)*phiDot0;
%
deltaF(12,10) = ((iX-iY)/iZ)*thetaDot0;
deltaF(12,11) = ((iX-iY)/iZ)*phiDot0;
%
phi_k = eye(12) + deltaF*Ts;
% EKF
% Time update
xCap_kMinus = phi_k*(del_x_kp1(:,count));
P_kMinus = phi_k*P_k1*phi_k' + Q;
% Measurement update
K_k = (P_kMinus*H')*inv(H*P_kMinus*H'+R);
%
xCap_k = xCap_kMinus + K_k*(yMeasurement - H*xCap_kMinus);
%
```

```
P_k = (eye(12) - K_k*H)*P_kMinus;
%
x_k = xCap_k;
P_k1 = P_k;
%
stateEstimates(:,count) = x_k(1:6);
%
x_k0 = del_x_kp1(:,count);
%
t = Ts:Ts:totT;
%
for k = 1:(length(t)-1)
    %
    x0 = x_k0(1) + e*(-1^count);
    y0 = x_k0(2) + (e*(-1^(count+1)));
    z0 = x_k0(3) + (e*(-1^count));
    xDot0 = x_k0(7) + (e*(-1^(count+1)));
    yDot0 = x_k0(8) + (e*(-1^count));
    zDot0 = x_k0(9) + (e*(-1^(count+1)));
    phi0 = x_k0(4) + (e*(-1^count));
    theta0 = x_k0(5) + (e*(-1^(count+1)));
    psi0 = x_k0(6) + (e*(-1^count));
    phiDot0 = x_k0(10) + (e*(-1^(count+1)));
    thetaDot0 = x_k0(11) + (e*(-1^count));
    psiDot0 = x_k0(12) + (e*(-1^(count+1)));
    %
    simout = sim('quadcopter_control','StartTime',num2str(0),'StopTime',...
        num2str(Ts),'OutputOption','SpecifiedOutputTimes','OutputTimes',...
        num2str(Ts));
    input = simout.Input.Data(end,:)';
    %
    del_x_kp1(:,count+1) = simout.simout.Data(end,:)';
    yMeasurement(:,count+1) = simout.sensorMeasure.Data(end,:)';
    yMeasurement(:,count+1) = yMeasurement(:,count+1) + [sigmaGPS*randn(3,1);...
        sigmaGY*randn(3,1)];
    %
    x = x_k(1);
    y = x_k(2);
    z = x_k(3);
    xDot = x_k(7);
    yDot = x_k(8);
    zDot = x_k(9);
    phi = x_k(4);
    theta = x_k(5);
    psi = x_k(6);
    phiDot = x_k(10);
    thetaDot = x_k(11);
    psiDot = x_k(12);
    % EKF
    deltaF = zeros(12); % Derivative of the non-linear function matrix
    F = zeros(12);
    %
    deltaF(1,7) = 1.0; % xDot
    deltaF(2,8) = 1.0; % yDot
    deltaF(3,9) = 1.0; % zDot
    deltaF(4,10) = 1.0; % phiDot
    deltaF(5,11) = 1.0; % thetaDot
    deltaF(6,12) = 1.0; % psiDot
    %
    deltaF(7,4) = (input(1)/mass)*((-sin(phi)*sin(theta)*cos(psi))+(cos(phi)*sin(psi)));
    deltaF(7,5) = (input(1)/mass)*(cos(phi)*cos(theta)*cos(psi));
    deltaF(7,6) = (input(1)/mass)*((-cos(phi)*sin(theta)*sin(psi))+(sin(phi)*cos(psi)));
    %
    deltaF(8,4) = (input(1)/mass)*((-sin(phi)*sin(theta)*sin(psi))-(cos(phi)*cos(psi)));
    deltaF(8,5) = (input(1)/mass)*(cos(phi)*cos(theta)*sin(psi));
    deltaF(8,6) = (input(1)/mass)*((cos(phi)*sin(theta)*cos(psi))+(sin(phi)*sin(psi)));
    %
    deltaF(9,4) = (input(1)/mass)*(-sin(phi)*cos(theta));
    deltaF(9,5) = (input(1)/mass)*(-cos(phi)*sin(theta));
    %
    deltaF(10,11) = ((iY-iZ)/iX)*psiDot;
    deltaF(10,12) = ((iY-iZ)/iX)*thetaDot;
    %
    deltaF(11,10) = ((iZ-iX)/iY)*psiDot;
    deltaF(11,12) = ((iZ-iX)/iY)*phiDot;
    %
    deltaF(12,10) = ((iX-iY)/iZ)*thetaDot;
    deltaF(12,11) = ((iX-iY)/iZ)*phiDot;
    %
    phi_k = eye(12) + deltaF*Ts;
    % EKF
    % Time update
    xCap_kMinus = phi_k*(del_x_kp1(:,count+1));
    P_kMinus = phi_k*P_k1*phi_k' + Q;
```

```
    % Measurement update
    K_k = (P_kMinus*H')*inv(H*P_kMinus*H'+R);
    %
    xCap_k = xCap_kMinus + K_k*(yMeasurement(:,count+1) - H*xCap_kMinus);
    %
    P_k = (eye(12) - K_k*H)*P_kMinus;
    %
    x_k = xCap_k;
    P_k1 = P_k;
    %
    count = count + 1;
    stateEstimates(:,count) = x_k(1:6);
    %
    x_k0 = del_x_kp1(:,count);
end
%% Plot the results
%
figure
p = plot(t,stateEstimates(3,:),'g',t,yMeasurement(3,:),'b--o',t,del_x_kp1(3,:),'r');
p(1).LineWidth = 2;
title('Z estimate')
xlabel('Time (s)')
ylabel('Z Height (m)')
legend('Estimated State','Noise Measurement','Actual State')
%
figure
p = plot(t,stateEstimates(1,:),'g',t,yMeasurement(1,:),'b--o',t,del_x_kp1(1,:),'r');
p(1).LineWidth = 2;
title('X estimate')
xlabel('Time (s)')
ylabel('X Distance (m)')
legend('Estimated State','Noise Measurement','Actual State')
%
figure
p = plot(t,stateEstimates(2,:),'g',t,yMeasurement(2,:),'b--o',t,del_x_kp1(2,:),'r');
p(1).LineWidth = 2;
title('Y estimate')
xlabel('Time (s)')
ylabel('Y Distance (m)')
legend('Estimated State','Noise Measurement','Actual State')
%
figure
p = plot(t,stateEstimates(4,:),'g',t,yMeasurement(4,:),'b--o',t,del_x_kp1(4,:),'r');
p(1).LineWidth = 2;
title('Phi estimate')
xlabel('Time (s)')
ylabel('Phi angle (rad)')
legend('Estimated State','Noise Measurement','Actual State')
%
figure
p = plot(t,stateEstimates(5,:),'g',t,yMeasurement(5,:),'b--o',t,del_x_kp1(5,:),'r');
p(1).LineWidth = 2;
title('Theta estimate')
xlabel('Time (s)')
ylabel('Theta angle (rad)')
legend('Estimated State','Noise Measurement','Actual State')
%
figure
p = plot(t,stateEstimates(6,:),'g',t,yMeasurement(6,:),'b--o',t,del_x_kp1(6,:),'r');
p(1).LineWidth = 2;
title('Psi estimate')
xlabel('Time (s)')
ylabel('Psi angle (rad)')
legend('Estimated State','Noise Measurement','Actual State')
%
```

Appendix B
Unscented Kalman Filter

```
%% Unscented Kalman Filter
%
Ts = 0.025;
totT = 2.5;
sigmaGPS = 0.25;
sigmaGY = 0.1;
e = 1e-6;
%
x0 = 0.0;
y0 = 0.0;
z0 = 0.0;
xDot0 = 0.0;
yDot0 = 0.0;
zDot0 = 0.0;
phi0 = 0.0;
theta0 = 0.0;
psi0 = 0.0;
phiDot0 = 0.0;
thetaDot0 = 0.0;
psiDot0 = 0.0;
%
count = 1;
%
```

```
t = Ts:Ts:totT;
%
P_k1 = eye(12);
%
Q = diag(e*ones(1,12));
%
R = eye(6);
R(1,1) = sigmaGPS*R(1,1);
R(2,2) = sigmaGPS*R(2,2);
R(3,3) = sigmaGPS*R(3,3);
R(4,4) = sigmaGY*R(4,4);
R(5,5) = sigmaGY*R(5,5);
R(6,6) = sigmaGY*R(6,6);
%
stateEstimates = zeros(6,24);
%
x_k0 = [x0;y0;z0;phi0;theta0;psi0;...
    xDot0;yDot0;zDot0;phiDot0;thetaDot0;psiDot0];
%
n = length(x_k0);
%
for k = 1:length(t)
    %
    x0 = x_k0(1)+e*(-1^(count+1));
    y0 = x_k0(2)+e*(-1^count);
    z0 = x_k0(3)+e*(-1^(count+1));
    xDot0 = x_k0(7)+e*(-1^count);
    yDot0 = x_k0(8)+e*(-1^(count+1));
    zDot0 = x_k0(9)+e*(-1^count);
    phi0 = x_k0(4)+e*(-1^(count+1));
    theta0 = x_k0(5)+e*(-1^count);
    psi0 = x_k0(6)+e*(-1^(count+1));
    phiDot0 = x_k0(10)+e*(-1^count);
    thetaDot0 = x_k0(11)+e*(-1^(count+1));
    psiDot0 = x_k0(12)+e*(-1^count);
    %
    simout = sim('quadcopter_control','StartTime',num2str(0),'StopTime',...
        num2str(Ts),'OutputOption','SpecifiedOutputTimes','OutputTimes',...
        num2str(Ts));
    %
    input = simout.Input.Data(end,:)';
    %
    del_x_kp1(:,count) = simout.simout.Data(end,:)';
    yMeasurement(:,count) = simout.sensorMeasure.Data(end,:)';
    yMeasurement(:,count) = yMeasurement(:,count) + [sigmaGPS*randn(3,1);...
        sigmaGY*randn(3,1)];
    % Time update
    M = chol(P_k1,'upper');
    %
    xBar_i_p = sqrt(n)*[M -M];
    %
    xI_p = x_k0 + xBar_i_p;
    %
    for i = 1:(2*n)
        %
        x0 = xI_p(1,i);
        y0 = xI_p(2,i);
        z0 = xI_p(3,i);
        phi0 = xI_p(4,i);
        theta0 = xI_p(5,i);
        psi0 = xI_p(6,i);
        xDot0 = xI_p(7,i);
        yDot0 = xI_p(8,i);
        zDot0 = xI_p(9,i);
        phiDot0 = xI_p(10,i);
        thetaDot0 = xI_p(11,i);
        psiDot0 = xI_p(12,i);
        %
        simout_Temp = sim('quadcopter_control','StartTime',num2str(0),'StopTime',...
            num2str(Ts),'OutputOption','SpecifiedOutputTimes','OutputTimes',...
            num2str(Ts));
        %
        x_ki(:,i) = simout_Temp.simout.Data(end,:)';
    end
    %
    xCap_Minus = (1/(2*n))*sum(x_ki,2);
    %
    PMinus = zeros(12);
    %
    for i = 1:(2*n)
        PMinus = PMinus + (x_ki(:,i)-xCap_Minus)*(x_ki(:,i)-xCap_Minus)' + Q;
    end
    %
    PMinus = (1/(2*n)).*PMinus;
    %
    % Measurement update
    M = chol(PMinus,'upper');
    %
    xBar_i = sqrt(n)*[M -M];
    %
    xI_p = xCap_Minus + xBar_i;
    %
    for i = 1:(2*n)
        %
        x0 = xI_p(1,i);
```

```
            y0 = xI_p(2,i);
            z0 = xI_p(3,i);
            phi0 = xI_p(4,i);
            theta0 = xI_p(5,i);
            psi0 = xI_p(6,i);
            xDot0 = xI_p(7,i);
            yDot0 = xI_p(8,i);
            zDot0 = xI_p(9,i);
            phiDot0 = xI_p(10,i);
            thetaDot0 = xI_p(11,i);
            psiDot0 = xI_p(12,i);
            %
            simout_Temp = sim('quadcopter_control','StartTime','num2str(0),'StopTime',...
                num2str(Ts),'OutputOption','SpecifiedOutputTimes','OutputTimes',...
                num2str(Ts));
            %
            yMeasure(:,i) = simout_Temp.sensorMeasure.Data(end,:)';
        end
        %
        yBar = (1/(2*n))*sum(yMeasure,2);
        %
        Pxy = zeros(12,6);
        Py = zeros(6);
        %
        for i = 1:(2*n)
            Pxy = Pxy + (xI_p(:,i)-xCap_Minus)*(yMeasure(:,i)-yBar)';
            Py = Py + (yMeasure(:,i)-yBar)*(yMeasure(:,i)-yBar)' + R;
        end
        %
        Pxy = (1/(2*n)).*Pxy;
        Py = (1/(2*n)).*Py;
        %
        K = Pxy*inv(Py);
        %
        xCap = xCap_Minus + K*(yMeasurement(:,count) - yBar);
        %
        P_k = PMinus - K*Pxy';
        %
        stateEstimates(:,count) = xCap(1:6);
        %
        x_k0 = del_x_kp1(:,count);
        %
        count = count + 1;
end
%
%% Plot the results
%
figure
p = plot(t,stateEstimates(3,:),'g',t,yMeasurement(3,:),'b--o',t,del_x_kp1(3,:),'r');
p(1).LineWidth = 2;
title('Z estimate')
xlabel('Time (s)')
ylabel('Z Height (m)')
legend('Estimated State','Noise Measurement','Actual State')
%
figure
p = plot(t,stateEstimates(1,:),'g',t,yMeasurement(1,:),'b--o',t,del_x_kp1(1,:),'r');
p(1).LineWidth = 2;
title('X estimate')
xlabel('Time (s)')
ylabel('X Distance (m)')
legend('Estimated State','Noise Measurement','Actual State')
%
figure
p = plot(t,stateEstimates(2,:),'g',t,yMeasurement(2,:),'b--o',t,del_x_kp1(2,:),'r');
p(1).LineWidth = 2;
title('Y estimate')
xlabel('Time (s)')
ylabel('Y Distance (m)')
legend('Estimated State','Noise Measurement','Actual State')
%
figure
p = plot(t,stateEstimates(4,:),'g',t,yMeasurement(4,:),'b--o',t,del_x_kp1(4,:),'r');
p(1).LineWidth = 2;
title('Phi estimate')
xlabel('Time (s)')
ylabel('Phi angle (rad)')
legend('Estimated State','Noise Measurement','Actual State')
%
figure
p = plot(t,stateEstimates(5,:),'g',t,yMeasurement(5,:),'b--o',t,del_x_kp1(5,:),'r');
p(1).LineWidth = 2;
title('Theta estimate')
xlabel('Time (s)')
ylabel('Theta angle (rad)')
legend('Estimated State','Noise Measurement','Actual State')
%
figure
p = plot(t,stateEstimates(6,:),'g',t,
```

```
    yMeasurement(6,:),'b--o',t,del_x_kp1
    (6,:),'r');
p(1).LineWidth = 2;
title('Psi estimate')
xlabel('Time (s)')
ylabel('Psi angle (rad)')
legend('Estimated State','Noise
    Measurement','Actual State')
%
```