```
% quadsim_estimates.m
%
% Developed for JHU EP 525.461, UAV Systems & Control
% Adapted from design project in "Small Unmanned Aircraft: Theory and
% Practice", RWBeard & TWMcClain, Princeton Univ. Press, 2012
%
function out = quadsim_estimates(uu,P)

    % Extract variables from input vector uu
    %   uu = [meas(1:18); time(1)];
    k=(1:18);               meas=uu(k);   % Sensor Measurements
    k=k(end)+(1);           time=uu(k);   % Simulation time, s

    % Extract mesurements
    k=1;
    pn_gps = meas(k); k=k+1; % GPS North Measurement, m
    pe_gps = meas(k); k=k+1; % GPS East Measurement, m
    alt_gps= meas(k); k=k+1; % GPS Altitude Measurement, m
    Vn_gps = meas(k); k=k+1; % GPS North Speed Measurement, m/s
    Ve_gps = meas(k); k=k+1; % GPS East Speed Measurement, m/s
    Vd_gps = meas(k); k=k+1; % GPS Downward Speed Measurement, m/s
    p_gyro = meas(k); k=k+1; % Gyro Body Rate Meas. about x, rad/s
    q_gyro = meas(k); k=k+1; % Gyro Body Rate Meas. about y, rad/s
    r_gyro = meas(k); k=k+1; % Gyro Body Rate Meas. about z, rad/s
    ax_accel = meas(k); k=k+1; % Accelerometer Meas along x, m/s/s
    ay_accel = meas(k); k=k+1; % Accelerometer Meas along y, m/s/s
    az_accel = meas(k); k=k+1; % Accelerometer Meas along z, m/s/s
    static_press = meas(k); k=k+1; % Static Pressure Meas., N/m^2
    diff_press = meas(k); k=k+1; % Differential Pressure Meas., N/m^2
    psi_mag = meas(k); k=k+1; % Yaw Meas. from Magnetometer, rad
    future_use = meas(k); k=k+1;
    future_use = meas(k); k=k+1;
    future_use = meas(k); k=k+1;

    diff_press = 0;

    % Filter raw measurements
    persistent lpf_static_press ...
               lpf_diff_press ...
               lpf_p_gyro ...
               lpf_q_gyro ...
               lpf_r_gyro ...
               lpf_psi_mag
    if(time==0)
        % Filter initializations
        lpf_static_press = static_press;
        lpf_diff_press = diff_press;
        lpf_p_gyro = p_gyro;
        lpf_q_gyro = q_gyro;
        lpf_r_gyro = r_gyro;
        lpf_psi_mag = psi_mag;
    end
```

```
    lpf_static_press =
 LPF(static_press,lpf_static_press,P.tau_static_press,P.Ts);
    lpf_diff_press   = LPF(diff_press,lpf_diff_press,P.tau_diff_press,P.Ts);
    lpf_p_gyro = LPF(p_gyro,lpf_p_gyro,P.tau_gyro,P.Ts);
    lpf_q_gyro = LPF(q_gyro,lpf_q_gyro,P.tau_gyro,P.Ts);
    lpf_r_gyro = LPF(r_gyro,lpf_r_gyro,P.tau_gyro,P.Ts);
    lpf_psi_mag = LPF(psi_mag,lpf_psi_mag,P.tau_mag,P.Ts);

    P0 = 101325;
    R = 8.31432;
    M = 0.0289644;
    T = 5/9*(P.air_temp_F-32)+273.15;
    P_launch = P0*exp(((-M*P.gravity)/(R*T))*P.h0_ASL);
    h_baro = ((-R*T)/(M*P.gravity))*log(lpf_static_press/P_launch);

    Va_pitot = 0;

    Q_att = diag([P.sigma_noise_gyro; P.sigma_noise_gyro].^2);
    R_att = (10^4)*diag([P.sigma_noise_accel; P.sigma_noise_accel;
 P.sigma_noise_accel].^2);
    sigma_ekfInitUncertainty = [5*pi/180; 5*pi/180];

    persistent xhat_att P_att
    if(time==0)
        xhat_att=[0; 0]; % States: [phi; theta]
        P_att=diag(sigma_ekfInitUncertainty.^2);
    end

    phi = xhat_att(1);
    theta = xhat_att(2);

    N=10;
    for i=1:N
        f_att = [1 sin(phi)*tan(theta) cos(phi)*tan(theta); ...
            0 cos(phi) -sin(phi)]*[p_gyro; q_gyro; r_gyro];

        A_att = [q_gyro*cos(phi)*tan(theta)-r_gyro*sin(phi)*tan(theta)
 (q_gyro*sin(phi)+r_gyro*cos(phi))*(1+tan(theta)^2); ...
            -q_gyro*sin(phi)-r_gyro*cos(phi) 0];

        xhat_att = xhat_att+(P.Ts/N)*f_att;

        P_att = P_att + (P.Ts/N)*(A_att*P_att + P_att*A_att' + Q_att);

        P_att = real(.5*P_att + .5*P_att');
    end

    y_att = [ax_accel; ay_accel; az_accel]; % Vector of actual measurements

    h_att = [q_gyro*Va_pitot*sin(theta)+P.gravity*sin(theta); ...
        r_gyro*Va_pitot*cos(theta)-p_gyro*Va_pitot*sin(theta)-
P.gravity*cos(theta)*sin(phi); ...
```

```
        -q_gyro*Va_pitot*cos(theta)-P.gravity*cos(theta)*cos(phi)]; %
 Mathematical model of measurements based on xhat

    C_att = [0 q_gyro*Va_pitot*cos(theta)+P.gravity*cos(theta); ...
        -P.gravity*cos(phi)*cos(theta) -r_gyro*Va_pitot*sin(theta)-
p_gyro*Va_pitot*cos(theta)+P.gravity*sin(phi)*sin(theta); ...
        P.gravity*sin(phi)*cos(theta)
 q_gyro*Va_pitot*sin(theta)+P.gravity*cos(phi)*sin(theta)]; % Linearization
 (Jacobian) of h(x,...) wrt x

    L_att = (P_att*C_att')/(C_att*P_att*C_att' + R_att); % Kalman Gain matrix

    I_att = eye(length(xhat_att));
    P_att = (I_att - L_att*C_att)*P_att;
    xhat_att = xhat_att + L_att*(y_att - h_att);
    xhat_att = mod(xhat_att+pi,2*pi)-pi;
    phi_hat_unc   = sqrt(P_att(1,1));
    theta_hat_unc = sqrt(P_att(2,2));

    Q_gps = diag([.1 .1 .1 .1 .1 .1].^2);
    R_gps = diag([2 2 2 .1 .1 .1].^2);
    sigma_ekfInitUncertainty = [P.sigma_eta_gps_north P.sigma_eta_gps_east -
P.sigma_eta_gps_alt ...
        P.sigma_noise_gps_speed P.sigma_noise_gps_speed
 P.sigma_noise_gps_speed];

    persistent xhat_gps P_gps prev_pn_gps prev_pe_gps
    if(time==0)
        xhat_gps=[pn_gps; pe_gps; -alt_gps; Vn_gps; Ve_gps; Vd_gps];
        P_gps=diag(sigma_ekfInitUncertainty.^2);
        prev_pn_gps = pn_gps;
        prev_pe_gps = pe_gps;
    end

    p_n = xhat_gps(1);
    p_e = xhat_gps(2);
    p_d = xhat_gps(3);
    v_n = xhat_gps(4);
    v_e = xhat_gps(5);
    v_d = xhat_gps(6);

    N=10;
    for i=1:N
        R_ned2b = eulerToRotationMatrix(phi,theta,psi_mag);
        f_gps = [v_n; v_e; v_d; ((R_ned2b'*[ax_accel; ay_accel;
 az_accel])+[0;0;P.gravity])];
        A_gps = [0 0 0 1 0 0; 0 0 0 0 1 0; 0 0 0 0 0 1; zeros(3, 6)];
        xhat_gps = xhat_gps+(P.Ts/N)*f_gps;
        P_gps = P_gps + (P.Ts/N)*(A_gps*P_gps + P_gps*A_gps' + Q_gps);
        P_gps = real(.5*P_gps + .5*P_gps');
    end

    if prev_pn_gps ~= pn_gps || prev_pe_gps ~= pe_gps
        meas_available = 1;
```

```
    else
        meas_available = 0;
    end

    if meas_available
        y_gps = [pn_gps; pe_gps; -alt_gps; Vn_gps; Ve_gps; Vd_gps]; % Vector
of actual measurements
        h_gps = [p_n; p_e; p_d; v_n; v_e; v_d]; % Mathematical model of
measurements based on xhat
        C_gps = eye(length(xhat_gps)); % Linearization (Jacobian) of h(x,...)
wrt x
        L_gps = (P_gps*C_gps')/(C_gps*P_gps*C_gps' + R_gps); % Kalman Gain
matrix
        I_gps = eye(length(xhat_gps));
        P_gps = (I_gps - L_gps*C_gps)*P_gps; % Covariance matrix updated with
measurement information
        xhat_gps = xhat_gps + L_gps*(y_gps - h_gps); % States updated with
measurement information
        pn_hat_unc   = sqrt(P_gps(1,1)); % EKF-predicted uncertainty in pn
estimate, rad
        pe_hat_unc = sqrt(P_gps(2,2)); % EKF-predicted uncertainty in pe
estimate, rad
        pd_hat_unc = sqrt(P_gps(3,3)); % EKF-predicted uncertainty in pd
estimate, rad
        vn_hat_unc = sqrt(P_gps(4,4)); % EKF-predicted uncertainty in vn
estimate, m/s
        ve_hat_unc = sqrt(P_gps(5,5)); % EKF-predicted uncertainty in ve
estimate, m/s
        vd_hat_unc = sqrt(P_gps(6,6)); % EKF-predicted uncertainty in vd
estimate, m/s
    end

    prev_pn_gps = pn_gps;
    prev_pe_gps = pe_gps;

    pn_hat=xhat_gps(1);
    pe_hat=xhat_gps(2);
    h_hat=h_baro;
    Va_hat=Va_pitot;
    phi_hat=xhat_att(1);
    theta_hat=xhat_att(2);
    psi_hat=lpf_psi_mag;
    p_hat=lpf_p_gyro;
    q_hat=lpf_q_gyro;
    r_hat=lpf_r_gyro;
    Vn_hat=xhat_gps(4);
    Ve_hat=xhat_gps(5);
    Vd_hat=xhat_gps(6);
    wn_hat=0;
    we_hat=0;

    % Compile output vector
    out = [...
            pn_hat;...    % 1
```

```
            pe_hat;...     % 2
            h_hat;...      % 3
            Va_hat;...     % 4
            phi_hat;...    % 5
            theta_hat;... % 6
            psi_hat;...    % 7
            p_hat;...      % 8
            q_hat;...      % 9
            r_hat;...      % 10
            Vn_hat;...     % 11
            Ve_hat;...     % 12
            Vd_hat;...     % 13
            wn_hat;...     % 14
            we_hat;...     % 15
            phi_hat_unc;...    % 16
            theta_hat_unc;... % 17
            0; % future use
            0; % future use
            0; % future use
            0; % future use
            0; % future use
            0; % future use
        ];

end

function y = LPF(u,yPrev,tau,Ts)
%
%  Y(s)       a           1
% ------ = ------- = -----------,  tau: Filter time contsant, s
%  U(s)     s + a     tau*s + 1         ( tau = 1/a )
%

alpha_LPF = exp(-Ts/tau);
y = alpha_LPF*yPrev + (1 - alpha_LPF)*u;

end
```

```
Not enough input arguments.

Error in quadsim_estimates (line 11)
    k=(1:18);              meas=uu(k);   % Sensor Measurements
```

Published with MATLAB® R2023a