ADVANCED CONTROL SYSTEM IMPLEMENTATION ON A QUADCOPTER

Mahdi Ghanei, Chris Korabik, Kyle Williams, Eugene Zaverukha

December 2019

Abstract

Quadcopters have enjoyed a growing popularity in control theory due to their non-linear behavior and challenges associated with autonomy. There has also been commercial growth due to affordability and ease of casual usage. According to a research report by Goldman Sachs, the drone market will increase to $100B by 2020, with defense being the largest market [1]. However, drones have a variety of applications besides military. Quadcopters can perform functions that would typically be impossible to execute by humans or otherwise more difficult with manually controlled aircraft, making drones an ideal in the fields of environment mapping, load transportation and aerology [2].

This paper focuses on the implementation of a stabilizing controller with position reference tracking for a quadcopter employing a commercially available open-source flight controller. The goal of the project was to design a Linear Quadratic Regulator (LQR) controller to stabilize a quadcopter given the initial states. The drone should then hover in the air and reject any external disturbances.

1 Problem Statement and Objective

A quadcopter is a helicopter employing four rapidly spinning motors to generate thrust for flight. Each motor produces a torque which is cancelled out by two motors spinning in the opposite direction. The dynamics of quadcopters are highly nonlinear due to them having 6 degrees of freedom. Coupled with only 4 actuators to control each state, these systems are severely challenging to implement controls on.

The primary objective of this project was to design and implement an LQR controller on a quadcopter. Ideally the controller would be robust to a variety of initial conditions and capable of rejecting disturbances on the system. There were several steps involved in successfully designing and implementing this controller: modelling the 6 degree of freedom nonlinear dynamics, linearizing the model, collecting data on the system states for parameter estimation, and creating the control algorithm to ultimately implement on the system.

There were some secondary objectives suggested for this project as well. These included either combining the stabilizing controller with trajectory tracking, or tracking the acceleration profile to return the drone to its initial position upon disturbance. However, these secondary objectives could not be reached in the allotted time period for this project.

2 System Model Development

The equation below summarizes drone dynamics, where $\vec{F}$ is the total force, $\vec{\tau}$ is the total torque, $\vec{a}$ and $\vec{v}$ are linear acceleration and velocity, $\vec{\alpha}$ and $\vec{\omega}$ are angular acceleration and velocity, $\vec{I_3}$ corresponds to the moments of inertia, and m is mass.

$$\begin{bmatrix} \vec{F} \\ \vec{\tau} \end{bmatrix} = \begin{bmatrix} m1_3 & 0 \\ 0 & I_3 \end{bmatrix} * \begin{bmatrix} \vec{a} \\ \vec{\alpha} \end{bmatrix} + \begin{bmatrix} \omega * mv \\ \omega * I_3 w \end{bmatrix} \tag{1}$$

Figure 1 shows the torques and forces imposed on the drone via each rotor. These are proportional to the power of each rotor or the square of the angular velocity (w_i^2), thus can be written as:

$$fi = c_t w_i^2 \quad and \quad \tau_i = \pm c_q w_i^2 \tag{2}$$

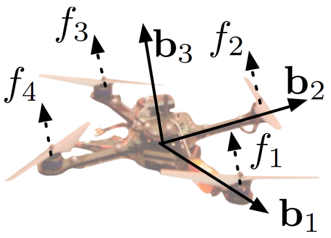

Figure 1: Individual forces and torques imposed on the drone by each rotor [3].

Where c_t and c_q are the rotor thrust and torque constants; τ_i and f_i are the individual forces and torques. With d being the distance from the rotors to the center, the total torques and forces of the quadrotor can also be represented as the following:

$$\begin{bmatrix} f \\ \tau_{b1} \\ \tau_{b2} \\ \tau_{b3} \end{bmatrix} = \begin{bmatrix} c_t & c_t & c_t & c_t \\ 0 & dc_t & 0 & -dc_t \\ -dc_t & 0 & dc_t & 0 \\ -c_q & c_q & -c_q & c_q \end{bmatrix} \begin{bmatrix} w_1^2 \\ w_2^2 \\ w_3^2 \\ w_4^2 \end{bmatrix} \tag{3}$$

Assuming small angles, the model was linearized and obtained using a standard form $\dot{x} = Ax + Bu$ where the state vector x is represented in 12 dimensions containing position and velocity in six degrees of freedom, and input u is the speed of each propeller. The states and the input can be written as:

$$\begin{aligned} x &= \begin{bmatrix} x & y & z & \dot{x} & \dot{y} & \dot{z} & \theta & \phi & \gamma & \dot{\theta} & \dot{\phi} & \dot{\gamma} \end{bmatrix}^T \\ u &= \begin{bmatrix} f & \tau_{b1} & \tau_{b2} & \tau_{b3} \end{bmatrix} \end{aligned} \tag{4}$$

Linearizing the system about its equilibrium points: $u = [mg, 0, 0, 0]$ and $x = [\bar{x}, \bar{y}, \bar{z}, 0 \cdots 0]$, the linearized A and B matrices can be found as shown:

$$A = \begin{bmatrix} 0&0&0&1&0&0&0&0&0&0&0&0 \\ 0&0&0&0&1&0&0&0&0&0&0&0 \\ 0&0&0&0&0&1&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&g&0&0&0&0 \\ 0&0&0&0&0&0&-g&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0&1&0&0 \\ 0&0&0&0&0&0&0&0&0&0&1&0 \\ 0&0&0&0&0&0&0&0&0&0&0&1 \\ 0&0&0&0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0&0&0&0 \end{bmatrix} \quad B = \begin{bmatrix} 0&0&0&0 \\ 0&0&0&0 \\ 0&0&0&0 \\ 0&0&0&0 \\ 0&0&0&0 \\ 1&0&0&0 \\ 0&0&0&0 \\ 0&0&0&0 \\ 0&0&0&0 \\ 0&\frac{1}{I_{xx}}&0&0 \\ 0&0&\frac{1}{I_{yy}}&0 \\ 0&0&0&\frac{1}{I_{zz}} \end{bmatrix} \tag{5}$$

3 Control System Design

An LQR controller was designed and its performance was evaluated in a MATLAB simulation. Q and R matrices were selected as design parameters for the state variables and the control input. To minimize the cost function J, Q and R were chosen to be positive definite, real, and symmetric. Eliminating the constraint on position, the initial choice for Q and R matrices are as follows:

$$\begin{aligned} Q = diag\left(\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 0.5 & 0.5 & 0.5 \end{bmatrix}\right) \\ R = diag\left(\begin{bmatrix} 5 & 5 & 5 & 5 \end{bmatrix}\right) \end{aligned} \tag{6}$$

The values for R were chosen to be relatively large to penalize large actuator effort, as it caused the quadcopter to jitter. Using MATLAB function $lqr(A, B, Q, R)$, the corresponding gain matrix K was determined. Initial guesses for the values in the Q and R matrices were made, and the controller was tested on the drone. Results from the trials were evaluated, and the Q and R were adjusted accordingly. This process was repeated several times until the controller was tuned successfully. Lastly, a framework for parameter estimation and a complete simulation of the nonlinear system along with a Kalman filter implementation was developed in Simulink and MATLAB respectively. Figure 2 demonstrates the simulation environment in MATLAB.

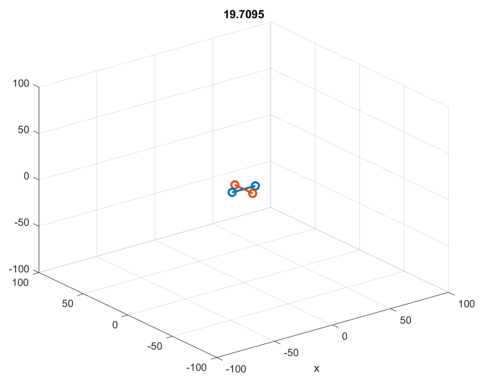

Figure 2: Simulation of the drone stabilizing itself from a random initial condition.

4 Robustness Analysis

A simulation of the drone and the LQR controller was developed to handle various control schemes such as velocity control, random initial conditions, and offset in the COM. Figure 3 depicts the results for a start from a random initial condition. As shown, the controller manages to recover the drone with zero steady-state error.

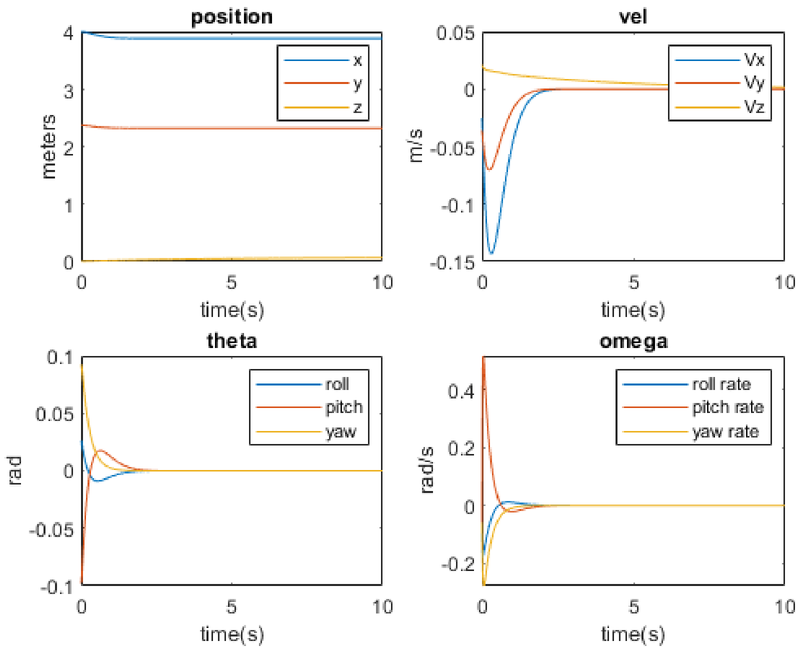

Figure 3: Simulation of the evolution of the states using the LQR controller.

The simulation shown above assumes a perfect linearized model of the nonlinear plant and perfect state measurement. The controller performs well under these conditions. The physical plant is subject to error in velocity estimation, offset bias in roll-pitch-yaw sensors, imperfect moment of inertia and aerodynamic coefficient estimation, an offset center of mass, and additional high order nonlinear aerodynamics. Simulations were performed with sensor bias, estimation error, and an offset COM. Each results in constant drift velocity. For brevity, only the COM simulation is included in Figure 4. This closely paralleled the performance of the physical plant. In contrast, the controller is relatively robust to linear changes in plant dynamics, such as imperfect moment of inertia estimation, as feedback can overcome these challenges.

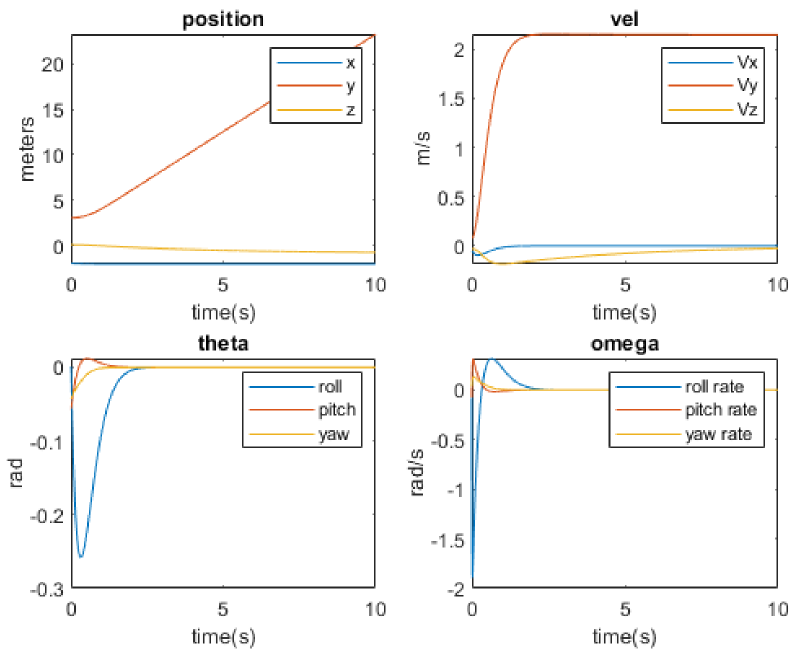

Figure 4: Simulation of the evolution of the states with an offset center of mass.

5 State Estimation and Kalman Filter

Implementing the optimal controller on the physical plant posed some challenges as the mathematical model does not represent the system 100% and the sensors used to derive the states provide noisy measurements. Kalman filtering is a common method used to account for the uncertainty in the model and noisy sensors. Assuming a sufficient number of independent random factors affect the physical system, the uncertainty in the mathematical model of the system can be represented as an Additive White Gaussian Noise (AWGN). This assumption allows for using a Kalman filter to estimate the states.

The discrete time state-space model of the system with noise can be written as:

$$
\begin{aligned}
x(k+1) &= Fx(k) + Gu(k) + w(k) \\
y(k+1) &= H(k+1)x(k+1) + v(k+1)
\end{aligned} \tag{7}
$$

Where $w(k)$ is the uncertainty in the model represented as an AWGN ($w(k) = \mathcal{N}(0,\, Q)$) and v is the measurement noise ($v(k+1) = \mathcal{N}(0,\, R)$). F, G, and H are the discrete time state-space transition matrices.

The Kalman filter attempts to predict the states by propagating them through the dynamics along with their associated uncertainty. The filter then corrects for the error in its prediction based on the new sensor measurement. Given an initial "guess" of the states ($\hat{x}_0^+$) and their uncertainty (P_0^+), the Linear Kalman Filter (LKF) algorithm is as follows:

1. KF Time Update ("Prediction Step") for time step k+1:

$$
\begin{aligned}
\hat{x}_{k+1}^- &= F\hat{x}_k^+ + Gu_k \\
P_{k+1}^- &= FP_k^+F^T + Q \\
K_{k+1} &= P_{k+1}^- H_{k+1}^T [H_{k+1} P_{k+1}^- H_{k+1}^T + R]^{-1}
\end{aligned} \tag{8}
$$

2. KF Measurement Update ("Correction Step") at time step k+1:

$$
\begin{aligned}
\hat{x}_{k+1}^+ &= \hat{x}_{k+1}^- + K_{k+1}(y_{k+1} - H_{k+1}\hat{x}_{k+1}^-) \\
P_{k+1}^+ &= (I - K_{k+1}H_{k+1})P_{k+1}^-
\end{aligned} \tag{9}
$$

Where $\hat{x}$ is estimated the state, P is the estimated covariance of the state, and K is the Kalman filter gain. The minus superscript denotes a prediction (priori) and the plus represents an estimation (posteriori) of the states and their covariance.

A Kalman filter was developed and tested to estimate the states of the drone for a given input of $u = [1.5 * mg, 0.3, 0, 0]$. Figure 5 and 6 demonstrate the estimated states given simulated noisy measurements. As shown, the output of the LKF provides a more accurate estimate of the states, and hence it ought to improve the performance of the LQR controller.

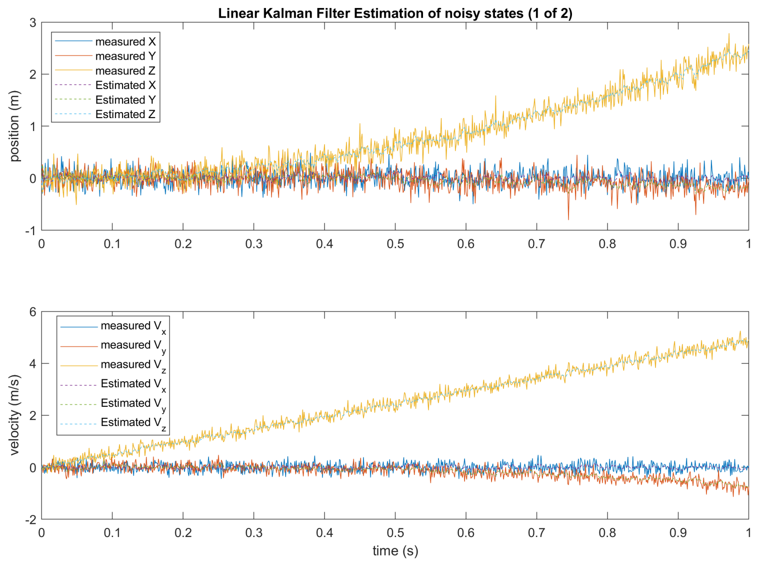

Figure 5: Linear Kalman filter estimation of the noisy position and velocities.

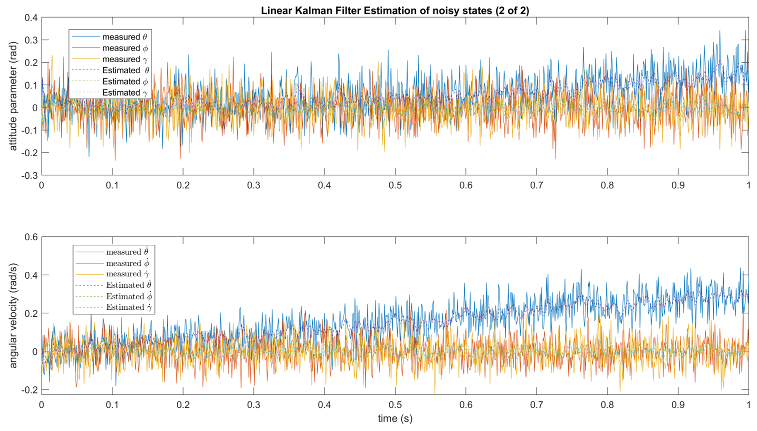

Figure 6: Linear Kalman filter estimation of the noisy attitude parameters and angular velocities.

6 Implementation on Hardware

The control algorithm was implemented on the drone using a Pixracer microcontroller board. This board in particular is a popular choice for small racing quadcopters because of its small size and built-in sensors. The board comes with two separate gyroscopes, two separate accelerometers, a magnetometer, and a barometer. Because of this convenience, the focus of the project was directed mainly toward the control design and software application.

In order to access sensor readings and actuator output commands, the PX4 Developer Environment was set up. PX4 is an open-source autopilot system that is made for use on inexpensive autonomous and remote-controlled aircraft. The use of the developer environment made building and flashing code to the Pixracer a simple task. For preliminary tests on collecting sensor data and actuating the rotors, a simulator called jMAVSim was used.

7 Discussion

There were several challenges that emerged during the controller implementation. The attitude control consistently worked to stabilize the quadcopter; however, there was no stable position control from just using the LQR controller. Since there was no direct access to position data, velocity and acceleration controls were used in separate attempts to mitigate changes in position. The velocity controller successfully bounded the position of the drone, but the velocity was never driven to zero. Similarly, the acceleration controller limited the drone's velocity, but it also could not prevent the drone from moving.

The pitch angle was consistently slanted toward the heavier side of the drone; this was likely due to error in the estimation of the system's moment of inertia and center of mass. An integral term was applied in an attempt to remove the steady-state error of the pitch angle, but it was unsuccessful. On such a small drone, any errors in parameter estimation can cause significant change in the system's behavior. The drone also experienced external forces from the tether cable and changing centers of mass from moving parts.

8 Conclusion

The quadcopter achieved fair success in the attitude control, but its performance was limited for velocity and acceleration controls. Throughout the course of the project, it became evident that an accurate model was needed to represent the physical system. There is little room for uncertainty when estimating system parameters, as moments of inertia and motor constants drastically impact the quadcopter's behavior. In addition, external reference sensors were needed to accurately measure altitude and velocity for position and velocity controls. Future implementation can focus on introducing trimming, which is a commonly used technique for adjusting aerodynamic forces on all manner of aircraft. Trimming can account for an offset center of mass to prevent drifting, as well as maintain constant altitude.

Were this project to be repeated from the beginning, the group would have made a slightly different approach. Although the Pixracer's on-board sensors were convenient, the PX4 firmware library proved to be difficult to maneuver at such a low level. While it would require some work assembling the hardware, it may be simpler to implement the control algorithm on another microcontroller from scratch. More time would be also be spent making a more accurate system model.

References

[1] Drones: Reporting for Work. (n.d.). Retrieved from
`https://www.goldmansachs.com/insights/technology-driving-innovation/drones/.`

[2] Hassanalian, M., Abdelkefi, A. (2017). Classifications, applications, and design challenges of drones: A review. Progress in Aerospace Sciences, 91, 99–131. doi: 10.1016/j.paerosci.2017.04.003

[3] Carnegie Mellon Lecture: Quadrotor Modeling and Control
`http://www.cs.cmu.edu/afs/cs.cmu.edu/academic/class/16311/www/s15/syllabus/ppp/Lec08-Control`

Appendix

Constructive Feedback from Past Labs

- Think carefully about what plots we want to show to represent our data – there are many to choose from
- Reaching thoughtful conclusions in a more concise manner. State what's important.
- Make sure to address any possible errors or inaccuracies involved with your simulations and results
- Label anything on graphs that is relevant to the discussion. For example, labelling the residual amplitude of the crane swing in the first couple labs gave a lot more meaning to the results we presented.
- Re-stating the values from the plots. Being able to infer values from the plot is not sufficient
- Adding a diagram corresponding to the dynamics equations
- Repeating the important results in the conclusion
- Carefully consider the graphs and data you present, don't present unnecessary data, and label and speak to the data you do present
- Be sure to highlight objective and discuss results
- Equations and figures should be clean and professional

MATLAB Code

Nonlinear Dynamics simulation and LQR:

```
function cmu_sim
close all; clear; clc;

tspan = [0 30];

%x0 = .1*[1 1 1 1 1 -200 1 1 1 -5 50 50]';
%x0 = .1*[0 0 0 0 0 0 1 1 1 1 1 1]';
while 1
    %close all;
    x0 = .5*(1 - 2*rand(1, 12));
    x0(1:2) = 50*[1 1];
    %x0 = 15*[1 1 0 0 0 0 0 0 0 0 0 0]
    %x0 = [0 0 0 0 0 0 0 0 0 .1 .1 .1]';
    %x0 = [0 0 0 0 0 0 0 0 0 0 0 0]';

    [t,x] = ode45(@(t,x) dynamics(t,x), tspan, x0);

    figure; animate(t, x)
    figure; plot_res(t, x)
    close all
end
end

function x_dot = dynamics(t, x)
%x = x1 x2 x3 v1 v2 v3 t1 t2 t3 w1 w2 w3
%xs and vs are in inertial frame
%ts (thetas) are roll pitch yaw in inertial frame
%w (ang. velocity) is in the body frame



t
%constants
m = 1;%mass
g = 9.81;%gravity
I = eye(3);%inertia matrix
```

```
x_dot = zeros(12, 1);

%controller command
%u = pos_control(t, x, [ 10;  10;  10]);
u = new_control(x);

%intermediate force computations
[thrust, tau1, tau2, tau3] = thrust_and_torques(u);
f = body_to_earth(thrust, x(7), x(8), x(9));
f = f - [0;0;m*g];

%linear state space, F = ma
x_dot(1:3) = x(4:6);
x_dot(4:6) = f/m;

%intermediate angular velocity computations
R = R_omega_to_earth(x(7), x(8));

%nonlinear, I w_dot = -w cross (I w) + tau
x_dot(7:9) = R*x(10:12);
x_dot(10:12) = I^-1*([tau1; tau2; tau3] - cross(x(10:12), I*x(10:12)));

end

function u = new_control(x)
x = x - [0 0 0 0 0 0 0 0 0 0 0 0].';
%constants
m = 1;%mass
g = 9.81;%gravity

persistent K
if isempty(K)
   K = lqr_design()
end

%T = [F; tau1; tau2; tau3]
T = -K*x;
T(1) = T(1)+m*g;
```

```
kt = 1;%thrust constant
d = 1;%distance b/w propellers
kd = 1;%drag constant

k = [ kt   kt    kt    kt;
      0    d*kt  0    -d*kt;
     -d*kt 0     d*kt  0;
     -kd   kd   -kd    kd;];

u = inv(k)*[T];


end

function plot_res(t, x)
figure;
subplot(2, 2, 1)
plot(t,x(:, 1:3))
title("position")
legend({'1', '2', '3'})

subplot(2, 2, 2)
plot(t,x(:, 4:6))
title("vel")
legend({'1', '2', '3'})

subplot(2, 2, 3)
plot(t,x(:, 7:9))
title("theta")
legend({'1', '2', '3'})

subplot(2, 2, 4)
plot(t,x(:, 10:12))
title("omega")
legend({'1', '2', '3'})
end
```

```
function animate(t, x)
L = 20;
[r, ~] = size(x);

body_pts = [-L/2  L/2  0   0;
             0    0   -L/2 L/2;
             0    0    0   0;];
%figure
%while 1
    for i = 1:10:r


        pts = Reb(x(i,7),x(i,8),x(i,9))*body_pts;

        plot3(x(i,1) + [pts(1, 1), pts(1, 2)], x(i,2) + [pts(2, 1), pts(2, 2)], x(i,3) + [pts(3
        hold on
        plot3(x(i,1) + [pts(1, 3), pts(1, 4)], x(i,2) + [pts(2, 3), pts(2, 4)], x(i,3) + [pts(3

        title(t(i))
        %plot3(20*sin(2*pi*t/20), -20*sin(2*pi*t/40), 5*sin(2*pi*t/20), 'k-')

        n = 100;
        look = [[-n n]+x(i,1) [-n n]+x(i,2) [-n n]+x(i,3)];
        look = [[-n n] [-n n] [-n n]];
        axis(look)

        %view(00,90)

        grid on
        xlabel("x")
        ylabel("y")
        zlabel("z")
        pause(.001)

        delete(gca)

    end
%end
```

```
end

function [f, tau1, tau2, tau3] = thrust_and_torques(cmd)
%cmd = [w w w w]'; input speed squared
kt = 1;%thrust constant
d = 1;%distance b/w propellers
kd = 1;%drag constant

k = [ kt   kt    kt    kt;
      0    d*kt  0    -d*kt;
     -d*kt 0     d*kt  0;
     -kd   kd   -kd    kd;];

T = k*cmd;

f    = T(1);
tau1 = T(2);
tau2 = T(3);
tau3 = T(4);
end

function f = body_to_earth(thrust, roll, pitch, yaw)
R = Reb(roll, pitch, yaw);
thrust = [0;
          0;
          thrust];
f = R*thrust;
end

function R = Reb(roll, pitch, yaw)
%x
R1 = [1 0        0;
      0 c(roll) -s(roll);
      0 s(roll)  c(roll)];
%y
R2 = [ c(pitch) 0 s(pitch);
       0        1 0;
      -s(pitch) 0 c(pitch)];
```

```
%z
R3 = [c(yaw) -s(yaw) 0;
      s(yaw)  c(yaw) 0;
      0       0      1];

R = R3*R2*R1;
end

function R = R_omega_to_earth(roll, pitch)

R = [1 s(roll)*tan(pitch) c(roll)*tan(pitch);
     0 c(roll) -s(roll);
     0 s(roll)/c(pitch) c(roll)/c(pitch)];

end

function out = c(t)
out = cos(t);
end

function out = s(t)
out = sin(t);
end
```

Kalman Filter:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%% Kalman Filter %%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
clear all;
close all;
clc
g= -9.8;
m=1;
Ix=1;
Iy=1;
Iz=1;

A = [ zeros(1,3) 1  zeros(1,8);
```

```
       zeros(1,4) 1 zeros(1,7);
         zeros(1,5) 1 zeros(1,6);
        zeros(1,7) g zeros(1,4);
         zeros(1,6) -g  zeros(1,5);
         zeros(1,12);
         zeros(1,9)  1  0  0;
         zeros(1,10) 1 0;
         zeros(1,11) 1;
         zeros(3,12)];


B =[ 0     0     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     0     0     0;
     0     1     0     0;
     0     0     1     0;
     0     0     0     1];

C = eye(12);%;[zeros(1,12)];
D = zeros(12,4);

dt = 0.001; % sensor update rate
maxTstep = 1000;

F = expm(A*dt);
Ah = [A B;
      zeros(4,16)];
Ah_exp = expm(Ah*dt);
G = Ah_exp(1:12,13:16); % Upper right
H = C;
M = D;
```

```
tvec = linspace(0, floor(maxTstep*dt), maxTstep+1);
uvec=[ones(1,maxTstep+1);
      zeros(1,maxTstep+1);
      zeros(1,maxTstep+1);
      zeros(1,maxTstep+1)];
% u = sin(10*t).*uvec;
u = uvec;



%% Kalman Filter
Xp(:,1) = zeros(12,1);
Pp(:,:,1) = eye(12);
Rtrue = 0.05*diag([.5*ones(1,6) .1*ones(1,6)]);
Qtrue = diag(2.25*[.5*ones(1,6) .1*ones(1,6)]);
Q = Qtrue;

% Simulate noisy measurements
run('eqn.m');
y = Y_noisy;

% Estimate the states given the noisy measurements

% dynamics prediction / correction steps (@ time step k >= 1)
for k = 1:maxTstep

    % dynamics prediction
    Xm(:,k+1) = F* Xp(:,k) + G*u(:,k);
    Pm(:,:,k+1) = F * Pp(:,:,k) * F' +  dt*eye(12)* Q;
    K(:,:,k+1) = Pm(:,:,k+1)*(H')*inv(H*Pm(:,:,k+1)*H' + 1*Rtrue);

    Xm(7:9,k+1) = wrapToPi(Xm(7:9,k+1));

    % correction
    Xp(:,k+1) = Xm(:,k+1) + K(:,:,k+1) * (y(:,k+1) - H*Xm(:,k+1));
    Pp(:,:,k+1) = (eye(12) - K(:,:,k+1)*H) * Pm(:,:,k+1);
    Xp(7:9,k+1) = wrapToPi(Xp(7:9,k+1));
```

```
end



%% Simulating noisy measurements and states

% maxTstep=1000;
% dt=0.001;
% Rtrue = dt*diag([.5*ones(1,6) .1*ones(1,6)]);
% Qtrue = diag(2.25*[.5*ones(1,6) .1*ones(1,6)]);

Sv_A = chol(Qtrue, 'lower');
Sv_B = chol(Rtrue, 'lower');

Xnoisy = zeros(12,maxTstep +1);
Y_noisy = zeros(12,maxTstep +1);
x_nonlin_last = Xnoisy(:,1) ;

%
for k = 1:maxTstep
    [t,x_nonlin] = ode45(@(t,x) dynamics(t,x,Sv_A * randn(12,1)),[dt*(k-1), dt*k], x_nonlin_la
    x_nonlin_last =  x_nonlin(end,:);

    Xnoisy(:,k+1)= x_nonlin_last;
end


for k = 1:maxTstep+1
    v = Sv_B * randn(12,1);
    Y_noisy(:,k) = Xnoisy(:,k)+ v;
end


% Plotting
figure;
tvec = linspace(0, floor(maxTstep*dt), maxTstep+1);
plot(tvec, Xnoisy(1:3,:))
title('simulated noisy states(Xnoisy)')
```

```
legend('x','y','z')

figure;
plot(tvec, Y_noisy(1:3,:))
title('simulated noisy measurements (ynoisy)')
legend('x','y','z')

save('noisy_meas','Y_noisy')

% Just using plain ode45
figure
[tode,xode] = ode45(@(t,x) dynamics(t,x,zeros(12,1)),[0, 1], zeros(12,1));
xode = xode';
plot(tode, xode(1:3,:))
title('simulated states without noise using ode45')
legend('x','y','z')
```

C++ Code

```
void MulticopterAttitudeControl::override_control(float *cmds) {
    //static declarations
    static int start_flag = 0;

    static int odom_fd;
    static int accel_fd;
    static struct vehicle_odometry_s odom;
    static struct sensor_combined_s accel;

    const static float g = -10.02;
    const static int size = 5;
    static int ndx;
    static float z_acc_avg[size];

    // start up code, only runs once
    if (start_flag == 0) {
        start_flag = 1;

        /* subscribe to sensor_combined topic */
        odom_fd  = orb_subscribe(ORB_ID(vehicle_odometry));
        accel_fd = orb_subscribe(ORB_ID(sensor_combined));

        /* limit the update rate to 1000 Hz */
        orb_set_interval( odom_fd, 1);
        orb_set_interval(accel_fd, 1);

        //initialize lowpass filter on acceleration in z
        for (int i = 0; i < size; i++) {
            z_acc_avg[i] = { 0 };
        }
        ndx = -1;
    }

    /* copy sensors raw data into local buffer */
    orb_copy(ORB_ID(vehicle_odometry), odom_fd, &odom);
    orb_copy(ORB_ID(sensor_combined), accel_fd, &accel);

    //Angles
    matrix::Eulerf euler_q = Eulerf(Quatf(odom.q));
    float roll  = euler_q.phi();
    float pitch = euler_q.theta();
    float yaw   = euler_q.psi();

    //Angular Velocities
    float rollspeed  = odom.rollspeed;
    float pitchspeed = odom.pitchspeed;
    float yawspeed   = odom.yawspeed;

    //Velocities transformed into body frame
    float Vxb =  cos(yaw)*odom.vx + sin(yaw)*odom.vy;
    float Vyb = -sin(yaw)*odom.vx + cos(yaw)*odom.vy;

    //moving average of acceleration in z
    //increment position in circular array
    ndx += 1;
    ndx %= size;
    //compare accelration to gravity
    z_acc_avg[ndx] = g - accel.accelerometer_m_s2[2];
```

```
    //sum last n measurments
    float z_acc_down = 0;
    for (int i = 0; i < size; i++) {
        z_acc_down += z_acc_avg[i];
    }
    //take average and flip sign
    z_acc_down /= size;
    z_acc_down *=-1;

    //command role pitch yaw torques and thrust
    //mixer code generate propeller speed based
    //on geometry and aerodynamic properties
    //scaling term based on propeller properties
    cmds[0] = (-0.1673f*Vyb -0.9744f*roll  -0.1673f*rollspeed) *0.0136f;
    cmds[1] = ( 0.1673f*Vxb -0.9938f*pitch -0.1790f*pitchspeed)*0.0136f;
    cmds[2] =                              (-0.1f*yawspeed)  *0.0136f;
    cmds[3] = .38f + 0.2f*(z_acc_down);
}
```