Drones in Wind

Nico Alba, Isabel Anderson, Emilio Gordon, Michael Gray

July, 2017

1 Goal

The "Drones in Wind" simulation simulates the flight of a quadcopter drone. The drone in the simulation will be modeled after the AscTec Pelican, a research-drone by the German company Ascending Technologies. The simulated drone as of the point of writing this paper is equipped with sensors to measure the velocity in the horizontal and vertical axis, the vertical position and the pitch angle. The goal is to create a controller that linearizes about a trajectory given by the third-party program OptimTraj. The trajectory comes from a cost function that minimizes the time integral of error from a desired position.

2 Model

The motion of the drone is governed by the ordinary differential equations with the state defined as

$$x = \begin{bmatrix} x \\ \dot{x} \\ z \\ \dot{z} \\ \theta \end{bmatrix} \tag{1}$$

such that x, z, correspond to horizontal and vertical position, respectively, and θ corresponds to the angle of the quad from upright. The inputs for the system are defined as

$$u = \begin{bmatrix} \omega \\ f \end{bmatrix} \tag{2}$$

such that ω corresponds to angular pitch rate and f corresponds to thrust. Taking the Jacobian lends the following A and B matrices shown by Equation 3 and Equation 4

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{f\cos\theta}{m} \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & \frac{-f\sin\theta}{m} \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{3}$$

$$B = \begin{bmatrix} 0 & 0 \\ 0 & \frac{\sin\theta}{m} \\ 0 & 0 \\ 0 & \frac{\cos\theta}{m} \\ 1 & 0 \end{bmatrix} \tag{4}$$

Substituting the equilibrium force, $f = mg$, results in the following matrices shown in Equation 5 and Equation 6.

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & g\cos\theta \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & -g\sin\theta \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{5}$$

$$B = \begin{bmatrix} 0 & 0 \\ 0 & \frac{\sin\theta}{m} \\ 0 & 0 \\ 0 & \frac{\cos\theta}{m} \\ 1 & 0 \end{bmatrix} \tag{6}$$

3 Controller and Optimization

3.1 Controller

The controller is designed to use in the loop control with LQR to follow a given trajectory. First, the init function creates a symbolic description of the A and B matrices, in which the angle θ is the only variable. These symbolic descriptions are then turned into MATLAB functions and saved in the data struct. The Q and R weight matrices are then initialized. The init function proceeds to load in the trajectory data and also adds it to the data struct. Finally, some parameters used to determine when the quad has reached its destination are initialized.

Logic is used at the start of the control loop to determine if there is still trajectory data available for the current timestep. If there is not, the controller will linearize about the last available trajectory point. This case does not happen in current simulations because the trajectory is planned for the entire duration of the simulation.

After determining that the simulation is still within the regime in which a trajectory is available, the controller will find the respective trajectory point to linearize about for the current timestep. The angle for this trajectory point is passed into the MATLAB functions to determine the relevant A and B matrices, which are then used in conjunction with the already initialized weights to create a gain matrix using LQR.

Finally, the input $u = -Kx$ is applied in which u is a column of two inputs, the first being thrust and the second being pitch rate. Note here that the thrust is augmented by the equilibrium condition for hover, $f = mg$. K is the previously specified gain matrix, and x is the difference between the current state and the trajectory state at the current time.

3.2 OptimTraj

OptimTraj (Citation needed) was used for trajectory planning. OptiTraj is a MATLAB library designed by a Cornell PHD student, Matthew P. Kelly, to solve for continuous-time single-phase trajectory optimization problems. For OptimTraj to solve the optimal trajectory, you must specify various parameters of your problem, including:

- Dynamics
- Objective function
- Bounds
- Initial trajectory guesses

Function Dynamics

We begin by setting up the function dynamics, `problem.func.dynamics`, as explained in the **Model** section above:

```
syms xdot x zdot z theta w thrust real
EOMs= [xdot;
       (thrust/mass)*sin(theta);
       zdot;
       (thrust/mass)*cos(theta)-gravity;
       w];

numf = matlabFunction(EOMs,'vars',[x xdot z zdot theta w thrust]);
problem.func.dynamics= @(t,x,u) numf( x(1,:),x(2,:),x(3,:),x(4,:),x(5,:),
                                     u(1,:), u(2,:)) );
```

Problem Bounds

The next step is to set up the `problem.func.dynamics`. For the function dynamics, a mass-normalized thrust was chosen to keep the model simple, shown in **Table 1** below.

Table 1 Problem Bounds

Parameter	Value	Description
t_i	0 s	Initial time
t_f	5 s	Final time
$\underline{F}/m$	0 m/s^2	Min thrust
$\overline{F}/m$	15 m/s^2	Max thrust
$\overline{\omega}$	5 rad/s	Max pitch rate

Cost Function

You can optimize for a certain aspect using a cost function. The cost functions for different things are shown below.

```
% Input:
problem.func.pathObj = @(t,x,u)( sum(u.^2,1) );

% Thrust:
problem.func.pathObj = @(t,x,u)( sum(u(2,:).^2,1) );

% Pitch Rate:
problem.func.pathObj = @(t,x,u)( sum(u(1,:).^2,1) );

% time
problem.func.pathObj = @(t,x,u)( sum((x(1,:)-x_f).^2,1)
                                +sum((x(3,:)-z_f).^2,1))
```

Initialize Guess

Lastly, before running the solution, the solver must be initialized. This is done with the mandatory `problem.guess` struct. Where x_i and x_f are the desired initial and final states, respectively. Additionally, u_i and u_f are the desired initial and final input, respectively.

$$\texttt{problem.guess.time} = [t_i, t_f]$$
$$\texttt{problem.guess.state} = [x_i, x_f]$$
$$\texttt{problem.guess.control} = [u_i, u_f]$$

Options

You can add options using `problem.options`, which include amount of details change accuracy settings, and select different solution methods. Different methods include trapezoid, Hermite Simpson, Chebyshev, and the Runge Kutta method.

Solution

Finally, to solve, the command `soln = optimTraj(problem)` is run. You can then unpack the simulation, save the data, and plot out

4 Analysis

5 Future Work