Drones in Wind

Nico Alba, Isabel Anderson, Emilio Gordon, Michael Gray, Will Reyes

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 and z correspond to horizontal and vertical position, respectively, while 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. The equations of motion for the system are

$$\begin{aligned}
\dot{x} &= \dot{x} \\
\ddot{x} &= \frac{f}{m}\sin\theta \\
\dot{z} &= \dot{z} \\
\ddot{z} &= \frac{f}{m}\cos\theta - g \\
\dot{\theta} &= \omega.
\end{aligned}$$

Taking the Jacobian lends the following A and B matrices

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

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

Which completes the linearized system.

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. 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 use the last available trajectory point for the state. The pitchrate and thrust will be set to $\omega = 0$ and $f = mg$, respectively. The reason that we cannot simply use the last available trajectory point for ω and f is that the trajectory frequently does not end with the equilibrium input.

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 $u = -Kx + n$ is applied in which u and n are the input to be applied and the trajectory input at the current time, respectively. 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 [1] 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

3.2.1 Function Dynamics

The simulations begins 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,:)) );
```

3.2.2 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

3.2.3 Cost Function

Using a cost function, an optimized trajectory around any aspect of the simulation can be found. The cost functions available for different variables 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))
```

3.2.4 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.

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

3.2.5 Options

Additional options are available using `problem.options`, which include details to change accuracy settings and select different solution methods. Different methods include trapezoid, Hermite Simpson, Chebyshev, and the Runge Kutta method.

3.2.6 Solution

Finally, running the command `soln = optimTraj(problem)` is used to solve. Following this, it is possible to unpack the simulation, save the data, and plot out.

4 Analysis

A method that computes quadcopter trajectories satisfying the minimal principle with respect to time-optimality has been developed. The method is based on the two-dimensional

quadcopter model with an optimal trajectory created using OptimTraj. By defining the dynamics, bounds and cost function the algorithm was able to create an optimal path for which the drone was to follow. Testing this method, the simulated drone accomplished moving from a position (0,0) to (1,1) in a time optimal manner.

5 Future Work

With the completion of a 2D optimized drone path, plans to enhance the simulation are underway. Several additions will be added to the current simulation in order to complete this task. Those are as follows

- Implement 3D rigid body dynamics
- Implement additional aerodynamic forces such as drag and ground effect

With these immediate goals in mind, an accurate simulation can be created to further test the overarching research initiative.

References

[1] MatthewPeterKelly *OptimTraj* GitHub
`https://github.com/MatthewPeterKelly/OptimTraj`