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. 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 Optimization and Controller

3.1 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

The optimal trajectory can now be solved for and subsequently "unpacked". By unpacking the solution, the state and input over time can be tracked and analyzed. One thing to note, however, is that the time-steps of solution matrices do **not** line up with the simulation that will run the controller with the optimal trajectory. This can be mitigated in the OptimTraj solver options, but will still lead to errors, which will be discussed further below.

3.1.1 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,:)) );
```

3.1.2 Problem Bounds

The next step is to set up the `problem.bounds`. For the function dynamics, a mass-normalized thrust was chosen to keep the model simple, shown in **Table 1** below. These parameters were chosen so that results of the simulation can be compared to D'Andrea's paper, *Performance benchmarking of quadrotor systems using time-optimal control*. Citation

Table 1 Problem Bounds

Parameter	Value	Description
t_i	0 s	Initial time
t_f	2 s	Final time
$\underline{F}/m$	0 m/s^2	Min thrust
$\overline{F}/m$	20 m/s^2	Max thrust
$\overline{\omega}$	10 rad/s	Max pitch rate

3.1.3 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))

% time objective function for boundary points
problem.func.bndObj = @(t0,x0,tF,xF) (tF-t0)
```

3.1.4 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.

$$\texttt{problem.guess.time} = [t_i, t_f]$$

$$\texttt{problem.guess.state} = [x_i, x_f]$$

$$\texttt{problem.guess.control} = [u_i, u_f]$$

3.1.5 Options

Additional options are available using `problem.options`, which include details to change accuracy settings and select different solution methods, like trapezoid, Chebyshev, etc.

3.1.6 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.

```
t = linspace(soln.grid.time(1),soln.grid.time(end),soln.grid.time(end)*
    TimeDensity);
x = soln.interp.state(t);
u = soln.interp.control(t);
save('traj.mat','t','x','u');
```

The reason the time grid-spacing is the end-time multiplied by `TimeDesnity` is to keep the grid-spacing of the simulation and optimal trajectory path the same. This will be elaborated on a further in Section 4, **Running Simulation**

3.2 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.

```
% Create functions
data.funcA = matlabFunction(A);
data.funcB = matlabFunction(B);

% Trajectory
load('traj.mat')
```

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.

```
% Create functions
if data.index <= length(data.T)
    data.x_eq = [data.X(data.index);
                 data.Xdot(data.index);
                 data.Z(data.index);
                 data.Zdot(data.index);
                 data.Theta(data.index)];

    data.index = data.index + 1;
end
```

Where `data.index` is initialized at 1. Recall, however, that this leads to an error. Fortunately, the states of each index are close enough to the quadcopter's actual state that it will still converge and follow a similar path to the time-optimal trajectory.

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 time-step. 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.

```
theta = data.x_eq(5,1);
A = data.funcA(theta);
B = data.funcB(theta);
```

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.

4 Running Simulation

To run the simulation, a function named `automation.m` was used to set up the problem bounds, plan the optimal trajectory, and analyze the results. Below is shown the script to do so.

```
% Constants to be applied
clear;

TimeDensity = 400;
xFinal = 5;
zFinal = 5;
runTime = 2;

gravity = 9.81;
mass = 1;

maxPitchRate = 10;
maxThrust = 20;
minThrust = 1;

save('runOptions.mat');

%Run the three scripts
planOptimalTrajectory()
DesignProblemSummer2017('Controller','datafile','data.mat')
AnalysisOfOptimalTrajectory()
```

`TimeDensity` refers to the grid-spacing in the optimal trajectory solution as well as the time-step for `DesignProblemSummer2017`. The other options set problem bounds and save them into `runOptions.mat` to be used by the DesignProblem, controller, and the Optimal trajectory planner.

`AnalysisOfOptimalTrajectory()` plots the simulation against the optimal trajectory calculated by optimTraj. **Solid lines** are to show the optimal trajectory, and **dashed lines** show the simulated quad-rotor path.

5 Analysis of Optimal Trajectory

OptimTraj was used to find the time-optimal trajectory for 5 different paths, all starting at the origin. Namely, the 5 paths ended at (1,1), (2,2), (3,3), (4,4), and (5,5). This is so that we can see how the model actually changes behaviors over these different paths. Ultimately, the results will be compared to **one** of D'Andrea's results Citation Needed, which took the path from (0,0) to (5,5). All plots will be shown alongside D'Andrea's for comparison.

5.1 OptimTraj Analysis

```
load('traj.mat')
t = t;
x = x(1,:);
xd= x(2,:);
z = x(3,:);
zd= x(4,:);
th= x(5,:);
w = u(1,:);
f = u(2,:);
```

The values for all 5 instances were unpacked and subsequently plotted over time.

5.2 Simulation Analysis

```
load('data1.mat')
t = processdata.t;
x = processdata.x;
xd = processdata.xdot;
z = processdata.z;
zd = processdata.zdot;
th = processdata.theta;
f = controllerdata.actuators.thrust;
w = controllerdata.actuators.pitchrate;
```

Similarly, the simulations were also unpacked and plotted over time.

5.3 D'Andrea Results

Before comparing results to D'Andrea's of time optimal trajectory results, the differences in simulations must be addressed. To start, D'Andrea's control input is bang-bang in thrust and bang-singular in pitch rate.Citation Needed. Additionally, those inputs are non-zero values at both the start and end. Whereas our controller is continuous(???) and the inputs both begin at zero values. The plot for his results are shown in figure X below.

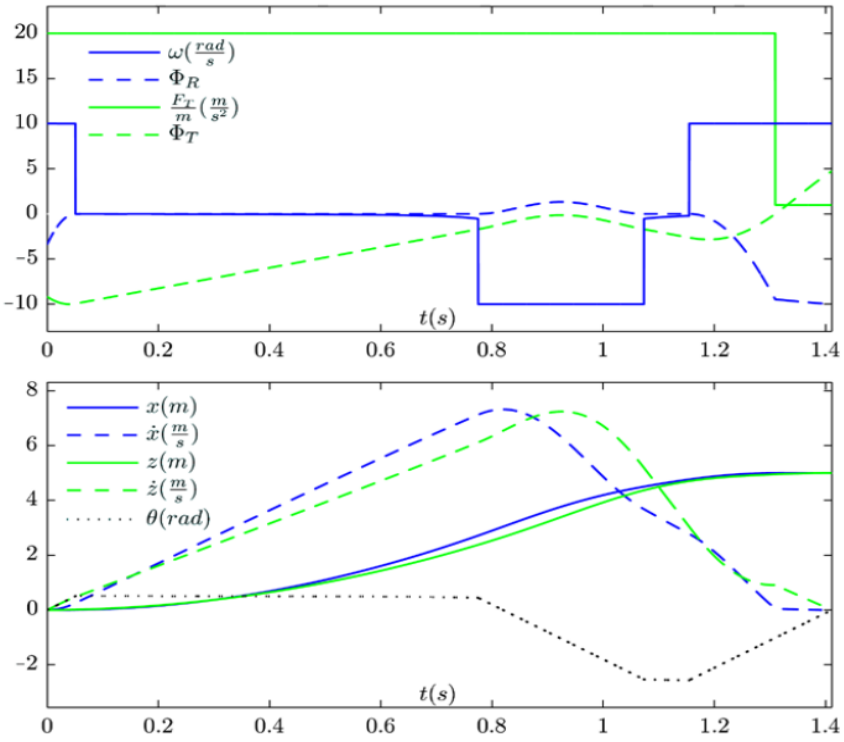

Fig. 7 Input and state trajectories of an example maneuver with $x_T = 5$ m and $z_T = 5$ m. The scaled switching functions are also drawn in the plot of the control inputs

Figure 1: Screenshot?

Qualitatively, the inputs can be broken down into sections, as is described by the switching functions Let's try and get another source about this topic. The Thrust begins at $20N$, then drops to 0 as the quad-copter nears it's final state, $x = z = 5$. The pitch rate begins at $10rad/s$, drops to 0, then to $-10rad/s$ to stabilize, and finally jumps to $10rad/s$.

In the following subsections we will show how the behavior of our controller compares and how it changes as the final state goes from $x = z = 1$ to $x = z = 5$.

6 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`