

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

$$x = \begin{bmatrix} x \\ \dot{x} \\ z \\ \dot{z} \\ \theta \end{bmatrix} \quad (1)$$

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

$$u = \begin{bmatrix} \omega \\ f \end{bmatrix} \quad (2)$$

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

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

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

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

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

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

3 Optimization and Controller

3.1 OptimTraj

OptimTraj [2] 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 solver matrices do **not** line up with the simulation time steps provided that run the controller. This is accounted for by modifying the simulation parameters.

3.1.1 Function Dynamics

The simulations begins by setting up the function dynamics, `problem.func.dynamics`, as explained in the **Model** section above:

```

1 syms xdot x zdot z theta w thrust real
2 EOMs= [xdot;
3       (thrust/mass)*sin(theta);
4       zdot;
5       (thrust/mass)*cos(theta)-gravity;
6       w];
7
8 numf = matlabFunction(EOMs,'vars',[x xdot z zdot theta w thrust]);
9 problem.func.dynamics= @(t,x,u) numf( x(1,:),x(2,:),x(3,:),x(4,:),x(5,:),
10                                     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 results from D'Andrea's paper [4].

Parameter	Value	Description
t_i	0 s	Initial time
t_f	2 s	Final time
\underline{F}/m	1 m/s ²	Min thrust
\overline{F}/m	20 m/s ²	Max thrust
$\overline{\omega}$	10 rad/s	Max pitch rate

Table 1: Problem Bounds

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.

```
1 % Input:
2 problem.func.pathObj = @(t,x,u)( sum(u.^2,1) );
3
4 % Thrust:
5 problem.func.pathObj = @(t,x,u)( sum(u(2,:).^2,1) );
6
7 % Pitch Rate:
8 problem.func.pathObj = @(t,x,u)( sum(u(1,:).^2,1) );
9
10 % Time
11 problem.func.pathObj = @(t,x,u)( sum((x(1,:)-x_f).^2,1)
12                                     +sum((x(3,:)-z_f).^2,1))
13
14 % Time objective function for boundary points
15 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.

```
problem.guess.time = [t_i, t_f]
problem.guess.state = [x_i, x_f]
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.

```
1 t = linspace(soln.grid.time(1),soln.grid.time(end),soln.grid.time(end)*
   TimeDensity);
2 x = soln.interp.state(t);
3 u = soln.interp.control(t);
4 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.

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.

```
1 % Create functions
2 data.funcA = matlabFunction(A);
3 data.funcB = matlabFunction(B);
4
5 % Trajectory
6 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.

```
1 % Create functions
2 if data.index <= length(data.T)
3     data.x_eq = [data.X(data.index);
4                 data.Xdot(data.index);
5                 data.Z(data.index);
6                 data.Zdot(data.index);
7                 data.Theta(data.index)];
8
9     data.index = data.index + 1;
10 end
```

Where `data.index` is initialized at 1.

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.

```

1 theta = data.x_eq(5,1);
2 A = data.funcA(theta);
3 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 for hover, the thrust *must* augmented by the equilibrium condition, $f = mg$. 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.

```

1 % Constants to be applied
2 clear;
3
4 TimeDensity = 400;
5 xFinal = 5;
6 zFinal = 5;
7 runTime = 2;
8
9 gravity = 9.81;
10 mass = 1;
11
12 maxPitchRate = 10;
13 maxThrust = 20;
14 minThrust = 1;
15
16 save('runOptions.mat');
17
18 %Run the three scripts
19 planOptimalTrajectory()
20 DesignProblemSummer2017('Controller','datafile','data.mat')
21 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

In this section, the trajectory for $x = 5, z = 5$ was analyzed. First the Optimal trajectory was calculated and saved, then the simulation was run and data stored, and finally both results were plotted against each other. To reiterate, **Solid lines** are to show the optimal trajectory, and **dashed lines** show the simulated quad-rotor path.

Ultimately, the results will be compared to **one** of D’Andrea’s results [4].

5.1 Simulation Analysis

```
1 load traj
2 topt = t;
3 xopt = x;
4 uopt = u;
5
6 load data
7 tsim = processdata.t;
8 xsim = [processdata.x; processdata.xdot; processdata.z; processdata.zdot;
9         processdata.theta];
10 usim = [controllerdata.actuators.pitchrate; ...
11         controllerdata.actuators.thrust];
```

The preceding lines of code are meant to show how easily the data can be unpacked and plotted against each other. Below, Figure 1 shows the optimal and actual state of the quadrotor, and Figure 2 shows the optimal and actual inputs of the quadrotor.

Reminder, **Solid lines** show the calculated optimal trajectory. **Dashed lines** show the simulated quad-rotor path.

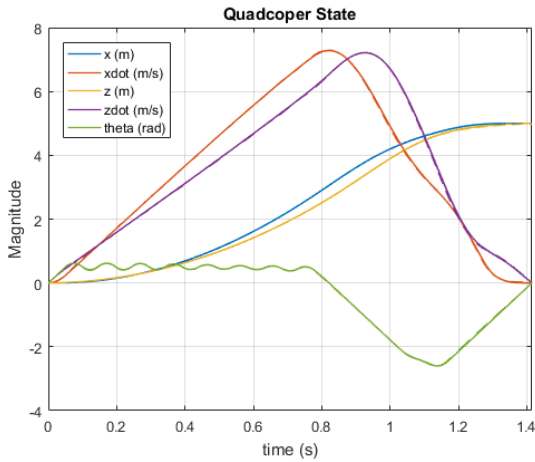


Figure 1: Quadcopter state over time.

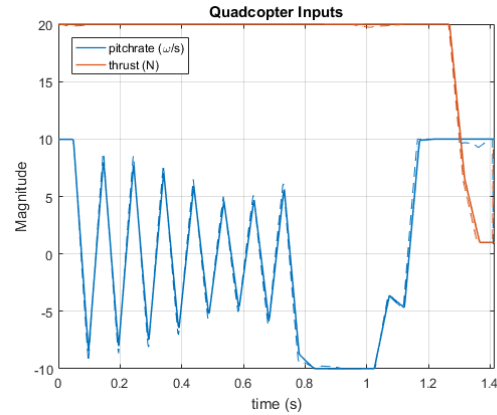


Figure 2: Quadcopter Inputs over time.

5.2 Comparison with D’Andrea Results

We can compare the simulated trajectory with that of *Performance Benchmarking of Quadrotor Systems Using Time-Optimal Control* [4]. The trajectory to be compared to from D’Andrea’s paper is shown in Figure 3

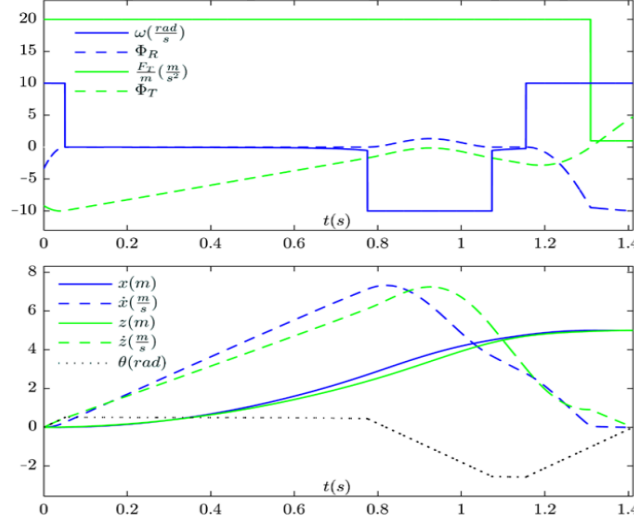


Fig. 7 Input and state trajectories of an example maneuver with $x_T = 5$ m and $z_T = 5$ m. The scaled switching functions are also drawn in the plot of the control inputs

Figure 3: Screenshot of D’Andrea Results

Though both results are nearly identical, the biggest difference in performance is in that of the inputs. D’Andrea input for thrust is bang-bang, meaning there is a switch discontinuity AND the input only takes the minimum and maximum values [3]. D’Andrea’s pitch rate is bang-singular, meaning the control input is always at full positive or negative saturation, except during singular arcs [4].

In our model, the inputs were free to vary as continuously as possible between the bounds set in table 1. Despite this, both models arrived at the destination $x = z = 5$ following the same time-optimal path and input behavior.

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

Controlling of an Single Drone [1] provides an outline for how to linearize a 3D quadcopter using rigid body dynamics. A key difference to note is that in the dynamics discussed in [1], each of the four quadrotor blades can produce its own thrust, which will then control the pitch, roll, and yaw of the rigid body. Additionally, the paper gives a brief description of the mechanism on applying aerodynamic forces, namely, drag.

References

- [1] Emile Biever *Controlling of an Single Drone*. TU/E Eindhoven: Department of Mechanical Engineering, 2015.
- [2] Matthew Peter Kelly *OptimTraj* GitHub
<https://github.com/MatthewPeterKelly/OptimTraj>
- [3] Nguyen Tan Tien *Introduction to Control Theory Including Optimal Control*. C.11 Bang-bang Control:53-58, 2002
<https://goo.gl/pj9oyA>
- [4] Raffaello D'Andrea *Performance Benchmarking of Quadrotor Systems Using Time-Optimal Control*. Auton Robot, 33:69-88, 2012.