Drones in Wind

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

August 29, 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 time optimal trajectory given by the third-party program OptimTraj.

2 Model

2.1 State

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

$$x = \begin{bmatrix} x \\ y \\ z \end{bmatrix}, \dot{x} = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \end{bmatrix}, \theta = \begin{bmatrix} \phi \\ \theta \\ \psi \end{bmatrix}, \dot{\theta} = \begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} \tag{1}$$

such that x and z correspond to horizontal and vertical position, respectively, while ϕ, θ, ψ correspond to the roll, pitch, and yaw angles respectively. However, the angular velocity vector $\omega \neq \dot{\theta}$.

2.2 Kinematics

The angular velocity vector, ω is not the same as $\dot{\theta}$, which is just the time derivative of the angles [5]. To convert the angular velocities into the angular velocity vector, we use the following relation:

$$\omega = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\phi) & \cos(\theta)\sin(\phi) \\ 0 & -\sin(\phi) & \cos(\theta)\cos(\phi) \end{bmatrix} \dot{\theta} \tag{2}$$

The body and inertial frame are related by the rotation matrix R, which goes from the body to inertial frame using the ZYX Euler angle convention.

$$R = \begin{bmatrix} \cos(\psi)\cos(\theta) & \cos(\psi)\sin(\phi)\sin(\theta) - \cos(\phi)\sin(\psi) & \sin(\phi)\sin(\psi) + \cos(\phi)\cos(psi)\sin(\theta) \\ \cos(\theta)\sin(\psi) & \cos(\phi)\cos(\psi) + \sin(\phi)\sin(\psi)\sin(\theta) & \cos(\phi)\sin(\psi)\sin(\theta) - \cos(\psi)\sin(\phi) \\ -\sin(\theta) & \cos(\theta)\sin(\phi) & \cos(\phi)\cos(\theta) \end{bmatrix} \tag{3}$$

2.3 Forces

The force of each motor is directly proportional to the angular velocity of that motor by:

$$T = k\omega^2 \tag{4}$$

Where k is a function of different constants unique to the quadrotor and derived in QDSC [5]. The thrust in the body frame is simply

$$T_B = k \begin{bmatrix} 0 \\ 0 \\ \omega_1^2 + \omega_2^2 + \omega_3^2 + \omega_4^2 \end{bmatrix} \tag{5}$$

Later on we will discuss future steps and how those constants will allow us to control the power supplied to the motors. But for now, we will simply use the force of each motor in our simulation as:

$$input(i) = f_i = \omega_i^2$$

The drag force is calculated by

$$\overrightarrow{F_D} = \frac{1}{2}\rho C_D A v^2$$

For our simulation, we will assume $\overrightarrow{F_D}$ is $[0, 0, 0]^T$, because we do not know enough about the quadcopter specs to model it accurately, though we will still include it in the equations of motions.

2.4 Torques

Each rotor contributes some torque about the body axis. The roll (6) and pitch (7) torque are respectively

$$\tau_\phi = Lk(f_3 - f_1) \tag{6}$$

$$\tau_\theta = Lk(f_4 - f_2) \tag{7}$$

and the yaw torque is defined as

$$\tau_\psi = b(f_1 - f_2 + f_3 - f_4) \tag{8}$$

Where L is the distance from the inertial center of the quadrotor to the ith motor. k and b are constants defined in [5]. The total torque *(without drag)* is $\tau = [\tau_{phi}, \tau_{theta}, \tau_{psi}]^T$

2.5 Equations of Motion

To start, the velocity and accelerations are simply:

$$\dot{x} = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \end{bmatrix}, \ddot{x} = \begin{bmatrix} 0 \\ 0 \\ -g \end{bmatrix} + \frac{RT_B + F_D}{m} \tag{9}$$

The angular velocities in the inertial frame are:

$$\omega = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\phi) & \cos(\theta)\sin(\phi) \\ 0 & -\sin(\phi) & \cos(\theta)\cos(\phi) \end{bmatrix}^{-1} \dot{\theta} \tag{10}$$

Finally, the angular accelerations are defined in equation

$$\dot{\omega} = I^{-1}(\tau - \omega \times (I\omega)) \tag{11}$$

If I, the Inertia Matrix, is axis-symmetric in the form

$$I = \begin{bmatrix} I_{xx} & 0 & 0 \\ 0 & I_{yy} & 0 \\ 0 & 0 & I_{zz} \end{bmatrix},$$

then

$$\dot{\omega} = \begin{bmatrix} \tau_\phi I_{xx}^{-1} \\ \tau_\theta I_{yy}^{-1} \\ \tau_\psi I_{zz}^{-1} \end{bmatrix} - \begin{bmatrix} \frac{I_{yy}-I_{zz}}{I_x x}\omega_y\omega_z \\ \frac{I_{zz}-I_{xx}}{I_y y}\omega_x\omega_z \\ \frac{I_{xx}-I_{yy}}{I_z z}\omega_x\omega_y \end{bmatrix}$$

The matlab code for the forces and torques is simply:

```
load('runOptions.mat')
syms x xdot y ydot z zdot phi phidot theta thetadot psi psidot
        f1 f2 f3 f4 real
state_sym = [x; xdot; y; ydot; z; zdot;
        phi; phidot; theta; thetadot; psi; psidot];
inputs_sym = [f1 f2 f3 f4];

k = 1;
b = 1;
Tb = [0;0;f1+f2+f3+f4];

tau_phi = k*(f3-f1)*dim_1;
tau_theta = k*(f4-f2)*dim_2;
tau_psi = b*(f1-f2+f3-f4);
Tau_b = [tau_phi; tau_theta; tau_psi];
```

The moment and rotation matrices are then:

```
I = [Ix 0 0; 0 Iy 0; 0 0 Iz];
Rw = [1 0 -sin(theta);
          0 cos(phi) cos(theta)*sin(phi);
          0 -sin(phi) cos(theta)*cos(phi)];

Rz = [cos(psi) -sin(psi) 0;
      sin(psi) cos(psi) 0;
      0 0 1];
Ry = [cos(theta) 0 sin(theta);
      0 1 0;
      -sin(theta) 0 cos(theta)];
Rx = [1 0 0;
      0 cos(phi) -sin(phi);
      0 sin(phi) cos(phi)];
R= Rz*Ry*Rx;
```

Lastly, the equations of motion are easily expressed as:

```
equationsOfMotion = [xdot;ydot;zdot;
                     ([0;0;-gravity]+R*Tb/mass);
                     (inv(Rw)*[phidot;thetadot;psidot]);
                      inv(I)*Tau_b-cross(omega_b,I*omega_b)]
```

2.6 Linearizion

To linearize, we do what we've always done.

```
load('runOptions')
syms x xdot y ydot z zdot phi phidot theta thetadot psi psidot f1 f2 f3
    f4 real
% Symbolic description of A matrix
A = jacobian(equationsOfMotion,state_sym);
% Symbolic description of B matrix
B = jacobian(equationsOfMotion,inputs_sym);

% Create functions
data.funcA = matlabFunction(A,'Vars',[xdot ydot zdot phi theta psi phidot
     thetadot psidot f1 f2 f3 f4]);
data.funcB = matlabFunction(B,'Vars',[phi theta psi]);
```

There is no need to substitute in equilibrium conditions because we will be linearizing about a trajectory.

3 Optimization and Controller

3.1 OptimTraj

OptimTraj [2] was used for trajectory planning. OptimTraj 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:

```
syms xdot x zdot z theta w thrust real
EOMs= [xdot;ydot;zdot;
      ([0;0;-gravity]+R*Tb/mass);
      (inv(Rw)*[phidot;thetadot;psidot]);
       inv(I)*Tau_b-cross(omega_b,I*omega_b)];

numf = matlabFunction(simplify(equationsOfMotion),...
                      'vars',[x y z xdot ydot zdot...
                              phi theta psi phidot thetadot psidot...
                              f1 f2 f3 f4]);
problem.func.dynamics = @(t,x,u)( numf(x(1,:), x(2,:), x(3,:),...
                                       x(4,:), x(5,:), x(6,:),...
                                       x(7,:), x(8,:), x(9,:),...
                                       x(10,:), x(11,:), x(12,:),...
                                       u(1,:), u(2,:), u(3,:), u(4,:) ));
```

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].

The new problem bounds are derived from the D'Andrea paper but have been adapted to the current model. Namely, the thrust values now reflect the per-rotor thrust instead of the total. They are shown in table 1 below:

Parameter	Value	Description
t_i	0 s	Initial time
t_f	4 s	Final time
$\underline{F_i}/m$	$\frac{1}{4}$ m/s^2	Min thrust
$\overline{F_i}/m$	5 m/s^2	Max thrust

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 time optimal cost function is shown below.

```
1
2 % Time objective function for boundary points
3 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 it.

```
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 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 and the thrust f are the only variables. 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.

```
% Reference trajectory

% Reference trajectory
ind = data.index;
if data.index <= length(data.trajT)
    trajX = data.trajX(ind);
    trajY = data.trajY(ind);
    trajZ = data.trajZ(ind);

    trajXdot = data.trajXdot(ind);
    trajYdot = data.trajYdot(ind);
    trajZdot = data.trajZdot(ind);

    trajPhi = data.trajPhi(ind);
    trajTheta = data.trajTheta(ind);
    trajPsi = data.trajPsi(ind);

    trajPhidot = data.trajPhidot(ind);
    trajThetadot = data.trajThetadot(ind);
    trajPsidot = data.trajPsidot(ind);

    trajF1 = data.trajF1(ind);
    trajF2 = data.trajF2(ind);
    trajF3 = data.trajF3(ind);
    trajF4 = data.trajF4(ind);

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

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 and thrust 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.

```
A = data.funcA(trajXdot, trajYdot, trajZdot, trajPhi,trajTheta, trajPsi,
               trajPhidot, trajThetadot,trajPsidot,
               trajF1, trajF2, trajF3, trajF4);
B = data.funcB(trajPhi,trajTheta,trajPsi);
data.K = lqr(A,B,data.Q,data.R);
```

Finally, the input $u = -Kx$ is applied in which 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. Shown below is the script to do so.

```
% Constants to be applied
clear;

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

gravity = 9.81;
mass = 1;

maxThrust = 5;
minThrust = 0.25;

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.

5 Analysis of Optimal Trajectory

I will **lazily** compare different plots of the simulation and optimal trajectory.

5.1 Simulation Analysis

```
load traj
topt = t;
xopt = x;
uopt = u;

load data
tsim = processdata.t;
xsim = [processdata.x; processdata.y; processdata.z;
        processdata.xdot; processdata.ydot; processdata.zdot;
        processdata.phi; processdata.theta;processdata.psi;
        processdata.phidot; processdata.thetadot; processdata.psidot];

usim = [controllerdata.actuators.f1;
        controllerdata.actuators.f2;
        controllerdata.actuators.f3;
        controllerdata.actuators.f4];
```

The preceding lines of code are meant to show how easily the data can be unpacked and plotted against each other. The below figures show the trajectory and simulated states and inputs of the quadrotor for varying goal positions.

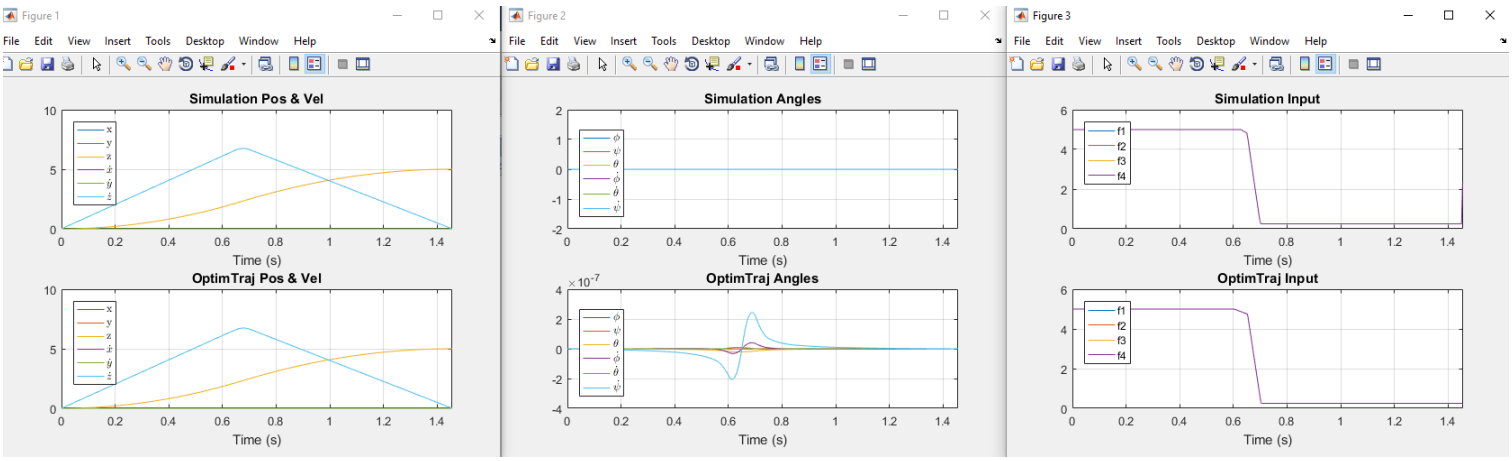

Figure 1: x=0, y=0, z=5

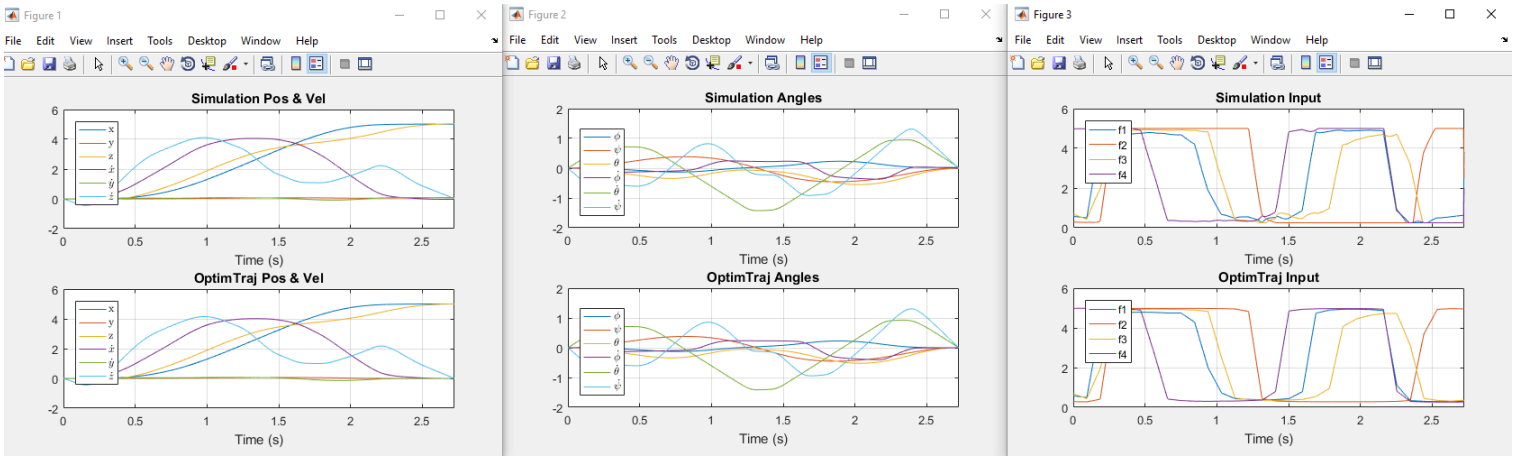

Figure 2: x=5, y=0, z=5

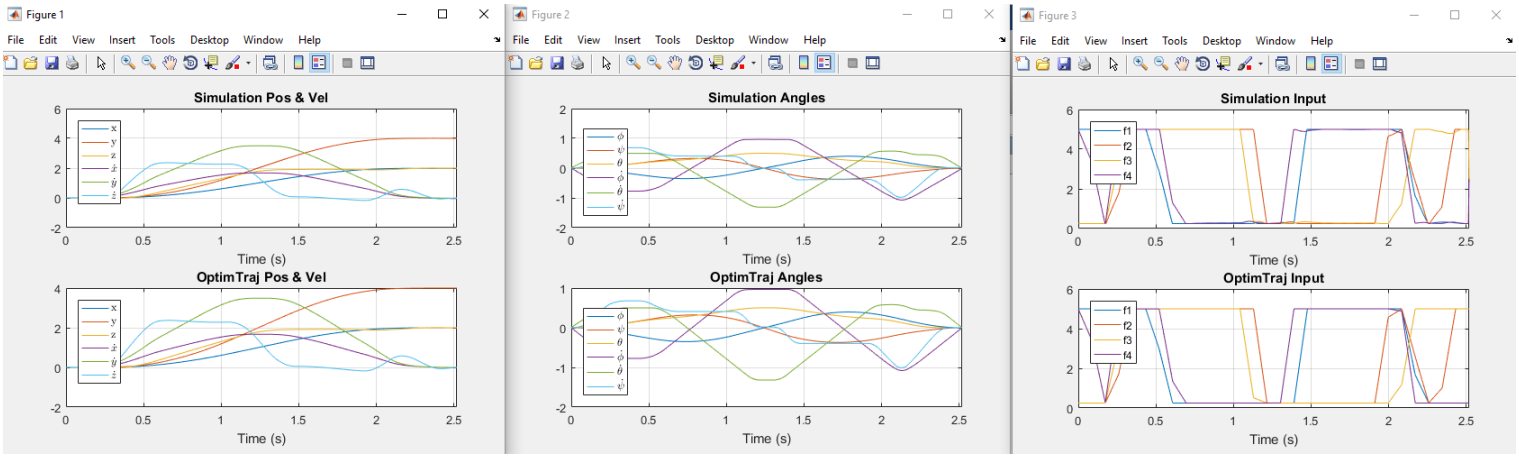

Figure 3: x=2, y=4, z=2

6 Future Work

- ~~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.

Quadcopter Dynamics, Simulation, and Control [5] provides the outline as to how we managed to redo the dynamics model. It also gives us step-by-step processes on creating a relationship between voltage drop across a motor and the thrust produced by the respective

propeller. Modelling our control inputs as voltage drops instead of thrusts is a more accurate description of what we will have access to on an actual vehicle, and may serve as preperation for writing flight code.

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] Rafaello D'Andrea *Performance Benchmarking of Quadrotor Systems Using Time-Optimal Control*. Auton Robot, 33:69-88, 2012.

[5] *Quadcopter Dynamics, Simulation, and Control*.