Quadcopter Dynamics, Simulation, and Control

Andrew Gibiansky

November 23, 2012

Introduction

A helicopter is a flying vehicle which uses rapidly spinning rotors to push air downwards, thus creating a thrust force keeping the helicopter aloft. Conventional helicopters have two rotors. These can be arranged as two coplanar rotors both providing upwards thrust, but spinning in opposite directions (in order to balance the torques exerted upon the body of the helicopter). The two rotors can also be arranged with one main rotor providing thrust and a smaller side rotor oriented laterally and counteracting the torque produced by the main rotor. However, these configurations require complicated machinery to control the direction of motion; a swashplate is used to change the angle of attack on the main rotors. In order to produce a torque the angle of attack is modulated by the location of each rotor in each stroke, such that more thrust is produced on one side of the rotor plane than the other. The complicated design of the rotor and swashplate mechanism presents some problems, increasing construction costs and design complexity.

A quadrotor helicopter (quadcopter) is a helicopter which has four equally spaced rotors, usually arranged at the corners of a square body. With four independent rotors, the need for a swashplate mechanism is alleviated. The swashplate mechanism was needed to allow the helicopter to utilize more degrees of freedom, but the same level of control can be obtained by adding two more rotors.

The development of quadcopters has stalled until very recently, because controlling four independent rotors has proven to be incredibly difficult and impossible without electronic assistance. The decreasing cost of modern microprocessors has made electronic and even completely autonomous control of quadcopters feasible for commercial, military, and even hobbyist purposes.

Quadcopter control is a fundamentally difficult and interesting problem. With six degrees of freedom (three translational and three rotational) and only four independent inputs (rotor speeds), quadcopters are severely underactuated. In order to achieve six degrees of freedom, rotational and translational motion are coupled. The resulting dynamics are highly nonlinear, especially after accounting for the complicated aerodynamic effects. Finally, unlike ground vehicles, helicopters have very little friction to prevent their motion, so they must provide their own damping in order to stop moving and remain stable. Together, these factors create a very interesting control problem. We will present a very simplified model of quadcopter dynamics and design controllers for our dynamics to follow a designated trajectory. We will then test our controllers with a numerical simulation of a quadcopter in flight.

Quadcopter Dynamics

We will start deriving quadcopter dynamics by introducing the two frames in which will operate. The inertial frame is defined by the ground, with gravity pointing in the negative z direction. The body frame is defined by the orientation of the quadcopter, with the rotor axes pointing in the positive z direction and the arms pointing in the x and y directions (Fig. 1).

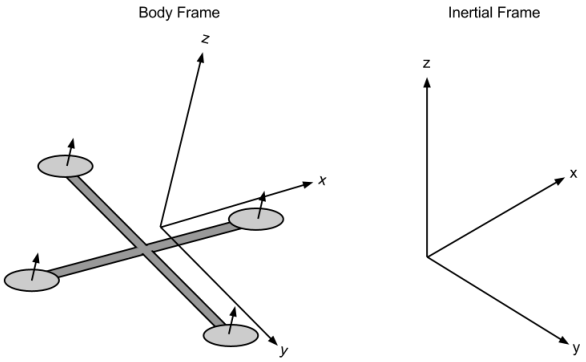

Figure 1: Quadcopter Body Frame and Inertial Frame

Kinematics

Before delving into the physics of quadcopter motion, let us formalize the kinematics in the body and inertial frames. We define the position and velocity of the quadcopter in the inertial frame as $x = (x, y, z)^T$ and $\dot{x} = (\dot{x}, \dot{y}, \dot{z})^T$, respectively. Similarly, we define the roll, pitch, and yaw angles in the body frame as $\theta = (\phi, \theta, \psi)^T$, with corresponding angular velocities equal to $\dot{\theta} = (\dot{\phi}, \dot{\theta}, \dot{\psi})^T$. However, note that the angular velocity vector $\omega \neq \dot{\theta}$. The angular velocity is a vector pointing along the axis of rotation, while $\dot{\theta}$ is just the time derivative of yaw, pitch, and roll. In order to convert these angular velocities into the angular velocity vector, we can use the following relation:

$$\omega = \begin{bmatrix} 1 & 0 & -s_\theta \\ 0 & c_\phi & c_\theta s_\phi \\ 0 & -s_\phi & c_\theta c_\phi \end{bmatrix} \dot{\theta}$$

where ω is the angular velocity vector in the body frame [1].

We can relate the body and inertial frame by a rotation matrix R which goes from the body frame to the inertial frame. This matrix is derived by using the ZYZ Euler angle conventions and successively "undoing" the yaw, pitch, and roll.

$$R = \begin{bmatrix} c_\phi c_\psi - c_\theta s_\phi s_\psi & -c_\psi s_\phi - c_\phi c_\theta s_\psi & s_\theta s_\psi \\ c_\theta c_\psi s_\phi + c_\phi s_\psi & c_\phi c_\theta c_\psi - s_\phi s_\psi & -c_\psi s_\theta \\ s_\phi s_\theta & c_\phi s_\theta & c_\theta \end{bmatrix}$$

For a given vector $\vec{v}$ in the body frame, the corresponding vector is given by $R\vec{v}$ in the inertial frame.

Physics

In order to properly model the dynamics of the system, we need an understanding of the physical properties that govern it. We will begin with a description of the motors being used for our quadcopter, and then use energy considerations to derive the forces and thrusts that these motors produce on the entire quadcopter. All motors on the quadcopter are identical, so we can analyze a single one without loss of generality. Note that adjacent propellers, however, are oriented opposite each other; if a propeller is spinning "clockwise", then the two adjacent ones will be spinning "counter-clockwise", so that torques are balanced if all propellers are spinning at the same rate.

Motors

Brushless motors are used for all quadcopter applications. For our electric motors, the torque produced is given by

$$\tau = K_t(I - I_0)$$

where τ is the motor torque, I is the input current, I_0 is the current when there is no load on the motor, and K_t is the torque proportionality constant [3]. The voltage across the motor is the sum of the back-EMF and some resistive loss:

$$V = IR_m + K_v\omega$$

where V is the voltage drop across the motor, R_m is the motor resistance, ω is the angular velocity of the motor, and K_v is a proportionality constant (indicating back-EMF generated per RPM). We can use this description of our motor to calculate the power it consumes. The power is

$$P = IV = \frac{(\tau + K_t I_0)(K_t I_0 R_m + \tau R_m + K_t K_v \omega)}{{K_t}^2}$$

For the purposes of our simple model, we will assume a negligible motor resistance. Then, the power becomes proportional to the angular velocity:

$$P \approx \frac{(\tau + K_t I_0)K_v\omega}{K_t}$$

Further simplifying our model, we assume that $K_t I_0 \ll \tau$. This is not altogether unreasonable, since I_0 is the current when there is no load, and is thus rather small. In practice, this approximation holds well enough. Thus, we obtain our final, simplified equation for power:

$$P \approx \frac{K_v}{K_t}\tau\omega.$$

Forces

The power is used to keep the quadcopter aloft. By conservation of energy, we know that the energy the motor expends in a given time period is equal to the force generated on the propeller times the distance that the air it displaces moves ($P \cdot \mathrm{d}t = F \cdot \mathrm{d}x$). Equivalently, the *power* is equal to the thrust times the air velocity ($P = F\frac{\mathrm{d}x}{\mathrm{d}t}$) [2].

$$P = Tv_h$$

We assume vehicle speeds are low, so v_h is the air velocity when hovering. We also assume that the free stream velocity, v_∞, is zero (the air in the surrounding environment is stationary relative to the quadcopter). Momentum theory gives us the equation for hover velocity as a function of thrust,

$$v_h = \sqrt{\frac{T}{2\rho A}}$$

where ρ is the density of the surrounding air and A is the area swept out by the rotor. Using our simplified equation for power, we can then write

$$P = \frac{K_v}{K_t}\tau\omega = \frac{K_v K_\tau}{K_t}T\omega = \frac{T^{\frac{3}{2}}}{\sqrt{2\rho A}}.$$

Note that in the general case, $\tau = \vec{r} \times \vec{F}$; in this case, the torque is proportional to the thrust T by some constant ratio K_τ determined by the blade configuration and parameters [2]. Solving for the thrust magnitude T, we obtain that thrust is proportional to the square of angular velocity of the motor:

$$T = \left(\frac{K_v K_\tau \sqrt{2\rho A}}{K_t}\omega\right)^2 = k\omega^2$$

where k is some appropriately dimensioned constant. Summing over all the motors, we find that the total thrust on the quadcopter (in the body frame) is given by

$$T_B = \sum_{i=1}^{4} T_i = k \begin{bmatrix} 0 \\ 0 \\ \sum {\omega_i}^2 \end{bmatrix}.$$

In addition to the thrust force, we will model friction as a force proportional to the linear velocity in each direction. This is a highly simplified view of fluid friction, but will be sufficient for our modeling and simulation. Our global drag forces will be modeled by an additional force term

$$F_D = \begin{bmatrix} -k_d \dot{x} \\ -k_d \dot{y} \\ -k_d \dot{z} \end{bmatrix}$$

If additional precision is desired, the constant k_d can be separated into three separate friction constants, one for each direction of motion. If we were to do this, we would want to model friction in the body frame rather than the inertial frame.

Torques

Now that we have computed the forces on the quadcopter, we would also like to compute the torques. Each rotor contributes some torque about the body z axis. This torque is the torque required to keep the propeller spinning and providing thrust; it creates the instantaneous angular acceleration and overcomes the frictional drag forces. The drag equation from fluid dynamics gives us the frictional *force*:

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

where ρ is the surrounding fluid density, A is the reference area (propeller cross-section, *not* area swept out by the propeller), and C_D is a dimensionless constant. This, while only accurate in some in some cases, is good enough for our purposes. This implies that the torque due to drag is given by

$$\tau_D = \frac{1}{2} R \rho C_D A v^2 = \frac{1}{2} R \rho C_D A (\omega R)^2 = b\omega^2$$

where ω is the angular velocity of the propeller, R is the radius of the propeller, and b is some appropriately dimensioned constant. Note that we've assumed that all the force is applied at the tip of the propeller, which is certainly inaccurate; however, the only result that matters for our purposes is that the drag torque is proportional to the square of the angular velocity. We can then write the complete torque about the z axis for the ith motor:

$$\tau_z = b\omega^2 + I_M \dot{\omega}$$

where I_M is the moment of inertia about the motor z axis, $\dot{\omega}$ is the angular acceleration of the propeller, and b is our drag coefficient. Note that in steady state flight (i.e. not takeoff or landing), $\dot{\omega} \approx 0$, since most of the time the propellers will be maintaining a constant (or almost constant) thrust and won't be accelerating. Thus, we ignore this term, simplifying the entire expression to

$$\tau_z = (-1)^{i+1} b {\omega_i}^2.$$

where the $(-1)^{i+1}$ term is positive for the ith propeller if the propeller is spinning clockwise and negative if it is spinning counterclockwise. The total torque about the z axis is given by the sum of all the torques from each propeller:

$$\tau_\psi = b \left({\omega_1}^2 - {\omega_2}^2 + {\omega_3}^2 - {\omega_4}^2 \right)$$

The roll and pitch torques are derived from standard mechanics. We can arbitrarily choose the $i = 1$ and $i = 3$ motors to be on the roll axis, so

$$\tau_\phi = \sum r \times T = L(k{\omega_1}^2 - k{\omega_3}^2) = Lk({\omega_1}^2 - {\omega_3}^2)$$

Correspondingly, the pitch torque is given by a similar expression

$$\tau_\theta = Lk({\omega_2}^2 - {\omega_4}^2)$$

where L is the distance from the center of the quadcopter to any of the propellers. All together, we find that the torques in the body frame are

$$\tau_B = \begin{bmatrix} Lk({\omega_1}^2 - {\omega_3}^2) \\ Lk({\omega_2}^2 - {\omega_4}^2) \\ b\left({\omega_1}^2 - {\omega_2}^2 + {\omega_3}^2 - {\omega_4}^2\right) \end{bmatrix}$$

The model we've derived so far is highly simplified. We ignore a multitude of advanced effects that contribute to the highly nonlinear dynamics of a quadcopter. We ignore rotational drag forces (our rotational velocities are relatively low), blade flapping (deformation of propeller blades due to high velocities and flexible materials), surrounding fluid velocities (wind), etc. With that said, we now have all the parts necessary to write out the dynamics of our quadcopter.

Equations of Motion

In the inertial frame, the acceleration of the quadcopter is due to thrust, gravity, and linear friction. We can obtain the thrust vector in the inertial frame by using our rotation matrix R to map the thrust vector from the body frame to the inertial frame. Thus, the linear motion can be summarized as

$$m\ddot{x} = \begin{bmatrix} 0 \\ 0 \\ -mg \end{bmatrix} + RT_B + F_D$$

where $\vec{x}$ is the position of the quadcopter, g is the acceleration due to gravity, F_D is the drag force, and T_B is the thrust vector in the body frame.

While it is convenient to have the linear equations of motion in the inertial frame, the rotational equations of motion are useful to us in the body frame, so that we can express rotations about the center of the quadcopter instead of about our inertial center. We derive the rotational equations of motion from Euler's equations for rigid body dynamics. Expressed in vector form, Euler's equations are written as

$$I\dot{\omega} + \omega \times (I\omega) = \tau$$

where ω is the angular velocity vector, I is the inertia matrix, and τ is a vector of external torques. We can rewrite this as

$$\dot{\omega} = \begin{bmatrix} \dot{\omega}_x \\ \dot{\omega}_y \\ \dot{\omega}_z \end{bmatrix} = I^{-1}\left(\tau - \omega \times (I\omega)\right).$$

We can model our quadcopter as two thin uniform rods crossed at the origin with a point mass (motor) at the end of each. With this in mind, it's clear that the symmetries result in a diagonal inertia matrix of the form

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

Therefore, we obtain our final result for the body frame rotational equations of motion

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

Simulation

Now that we have complete equations of motion describing the dynamics of the system, we can create a simulation environment in which to test and view the results of various inputs and controllers. Although more advanced methods are available, we can quickly write a simulator which utilizes Euler's method for solving differential equations to evolve the system state. In MATLAB, this simulator would be written as follows.

```
% Simulation times, in seconds.
start_time = 0;
end_time = 10;
dt = 0.005;
times = start_time:dt:end_time;

% Number of points in the simulation.
N = numel(times);

% Initial simulation state.
x = [0; 0; 10];
xdot = zeros(3, 1);
theta = zeros(3, 1);

% Simulate some disturbance in the angular velocity.
% The magnitude of the deviation is in radians / second.
deviation = 100;
thetadot = deg2rad(2 * deviation * rand(3, 1) - deviation);

% Step through the simulation, updating the state.
for t = times
    % Take input from our controller.
    i = input(t);

    omega = thetadot2omega(thetadot, theta);

    % Compute linear and angular accelerations.
    a = acceleration(i, theta, xdot, m, g, k, kd);
    omegadot = angular_acceleration(i, omega, I, L, b, k);

    omega = omega + dt * omegadot;
    thetadot = omega2thetadot(omega, theta);
    theta = theta + dt * thetadot;
    xdot = xdot + dt * a;
    x = x + dt * xdot;
end
```

We would then need functions to compute all of the physical forces and torques.

```
% Compute thrust given current inputs and thrust coefficient.
function T = thrust(inputs, k)
    % Inputs are values for \omega_i^2
    T = [0; 0; k * sum(inputs)];
end

% Compute torques, given current inputs, length, drag coefficient, and thrust coefficient.
function tau = torques(inputs, L, b, k)
    % Inputs are values for \omega_i^2
    tau = [
        L * k * (inputs(1) - inputs(3))
        L * k * (inputs(2) - inputs(4))
        b * (inputs(1) - inputs(2) + inputs(3) - inputs(4))
    ];
end

function a = acceleration(inputs, angles, xdot, m, g, k, kd)
    gravity = [0; 0; -g];
    R = rotation(angles);
    T = R * thrust(inputs, k);
    Fd = -kd * xdot;
    a = gravity + 1 / m * T + Fd;
end

function omegadot = angular_acceleration(inputs, omega, I, L, b, k)
    tau = torques(inputs, L, b, k);
    omegaddot = inv(I) * (tau - cross(omega, I * omega));
end
```

We would also need values for all of our physical constants, a function to compute the rotation matrix R, and functions to convert from an angular velocity vector ω to the derivatives of roll, pitch, and yaw and vice-versa. These are not shown. We can then draw the quadcopter in a

three-dimensional visualization which is updated as the simulation is running.

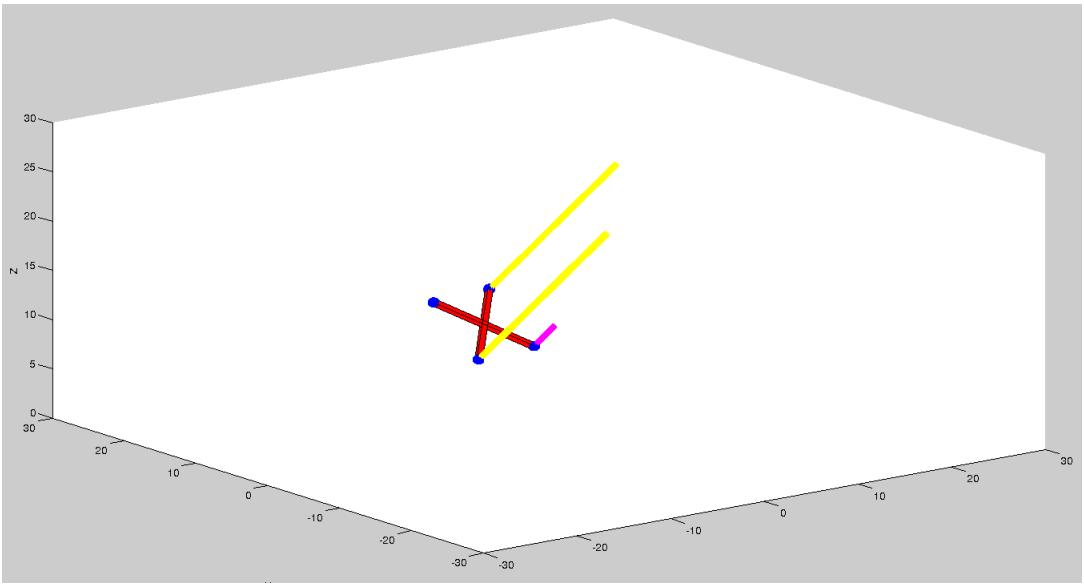

Figure 2: Quadcopter Simulation. Bars above each propeller represent, roughly, relative thrust magnitudes.

Control

The purpose of deriving a mathematical model of a quadcopter is to assist in developing controllers for physical quadcopters. The inputs to our system consist of the angular velocities of each rotor, since all we can control is the voltages across the motors. Note that in our simplified model, we only use the square of the angular velocities, ${\omega_i}^2$, and never the angular velocity itself, ω_i. For notational simplicity, let us introduce the inputs $\gamma_i = {\omega_i}^2$. Since we can set ω_i, we can clearly set γ_i as well. With this, we can write our system as a first order differential equation in state space. Let x_1 be the position in space of the quadcopter, x_2 be the quadcopter linear velocity, x_3 be the roll, pitch, and yaw angles, and x_4 be the angular velocity vector. (Note that all of these are 3-vectors.) With these being our state, we can write the state space equations for the evolution of our state.

$$
\begin{aligned}
\dot{x_1} &= x_2 \\
\dot{x_2} &= \begin{bmatrix} 0 \\ 0 \\ -g \end{bmatrix} + \frac{1}{m} R T_B + \frac{1}{m} F_D \\
\dot{x_3} &= \begin{bmatrix} 1 & 0 & -s_\theta \\ 0 & c_\phi & c_\theta s_\phi \\ 0 & -s_\phi & c_\theta c_\phi \end{bmatrix}^{-1} x_4 \\
\dot{x_4} &= \begin{bmatrix} {\tau_\phi I_{xx}}^{-1} \\ {\tau_\theta I_{yy}}^{-1} \\ {\tau_\psi I_{zz}}^{-1} \end{bmatrix} - \begin{bmatrix} \frac{I_{yy} - I_{zz}}{I_{xx}} \omega_y \omega_z \\ \frac{I_{zz} - I_{xx}}{I_{yy}} \omega_x \omega_z \\ \frac{I_{xx} - I_{yy}}{I_{zz}} \omega_x \omega_y \end{bmatrix}
\end{aligned}
$$

Note that our inputs are not used in these equations directly. However, as we will see, we can choose values for τ and T, and then solve for values of γ_i.

PD Control

In order to control the quadcopter, we will use a PD control, with a component proportional to the error between our desired trajectory and the observed trajectory, and a component proportional to the derivative of the error. Our quadcopter will only have a gyro, so we will only be able to use the angle derivatives $\dot{\phi}$, $\dot{\theta}$, and $\dot{\psi}$ in our controller; these measured values will give us the derivative of our error, and their integral will provide us with the actual error. We would like to stabilize the helicopter in a horizontal position, so our desired velocities and angles will all be zero. Torques are related to our angular velocities by $\tau = I\ddot{\theta}$, so we would like to set the torques proportional to the output of our controller, with $\tau = Iu(t)$. Thus,

$$\begin{bmatrix} \tau_\phi \\ \tau_\theta \\ \tau_\psi \end{bmatrix} = \begin{bmatrix} -I_{xx}\left(K_d\dot{\phi} + K_p\int_0^T \dot{\phi}\,\mathrm{d}t\right) \\ -I_{yy}\left(K_d\dot{\theta} + K_p\int_0^T \dot{\theta}\,\mathrm{d}t\right) \\ -I_{zz}\left(K_d\dot{\psi} + K_p\int_0^T \dot{\psi}\,\mathrm{d}t\right) \end{bmatrix}$$

We have previously derived the relationship between torque and our inputs, so we know that

$$\tau_B = \begin{bmatrix} Lk(\gamma_1 - \gamma_3) \\ Lk(\gamma_2 - \gamma_4) \\ b\left(\gamma_1 - \gamma_2 + \gamma_3 - \gamma_4\right) \end{bmatrix} = \begin{bmatrix} -I_{xx}\left(K_d\dot{\phi} + K_p\int_0^T \dot{\phi}\,\mathrm{d}t\right) \\ -I_{yy}\left(K_d\dot{\theta} + K_p\int_0^T \dot{\theta}\,\mathrm{d}t\right) \\ -I_{zz}\left(K_d\dot{\psi} + K_p\int_0^T \dot{\psi}\,\mathrm{d}t\right) \end{bmatrix}$$

This gives us a set of three equations with four unknowns. We can constrain this by enforcing the constraint that our inputs must keep the quadcopter aloft:

$$T = mg.$$

Note that this equation ignores the fact that the thrust will not be pointed directly up. This will limit the applicability of our controller, but should not cause major problems for small deviations from stability. If we had a way of determining the current angle accurately, we could compensate. If our gyro is precise enough, we can integrate the values obtained from the gyro to get the angles θ and ϕ. In this case, we can calculate the thrust necessary to keep the quadcopter aloft by projecting the thrust mg onto the inertial z axis. We find that

$$T_{\text{proj}} = mg\cos\theta\cos\phi$$

Therefore, with a precise angle measurement, we can instead enforce the requirement that the thrust be equal to

$$T = \frac{mg}{\cos\theta\cos\phi}$$

in which case the component of the thrust pointing along the positive z axis will be equal to mg. We know that the thrust is proportional to a weighted sum of the inputs:

$$T = \frac{mg}{\cos\theta\cos\phi} = k\sum\gamma_i \implies \sum\gamma_i = \frac{mg}{k\cos\theta\cos\phi}$$

With this extra constraint, we have a set of four linear equations with four unknowns γ_i. We can then solve for each γ_i, and obtain the following input values:

$$\begin{aligned}
\gamma_1 &= \frac{mg}{4k\cos\theta\cos\phi} - \frac{2be_\phi I_{xx} + e_\psi I_{zz} kL}{4bkL} \\
\gamma_2 &= \frac{mg}{4k\cos\theta\cos\phi} + \frac{e_\psi I_{zz}}{4b} - \frac{e_\theta I_{yy}}{2kL} \\
\gamma_3 &= \frac{mg}{4k\cos\theta\cos\phi} - \frac{-2be_\phi I_{xx} + e_\psi I_{zz} kL}{4bkL} \\
\gamma_4 &= \frac{mg}{4k\cos\theta\cos\phi} + \frac{e_\psi I_{zz}}{4b} + \frac{e_\theta I_{yy}}{2kL}
\end{aligned}$$

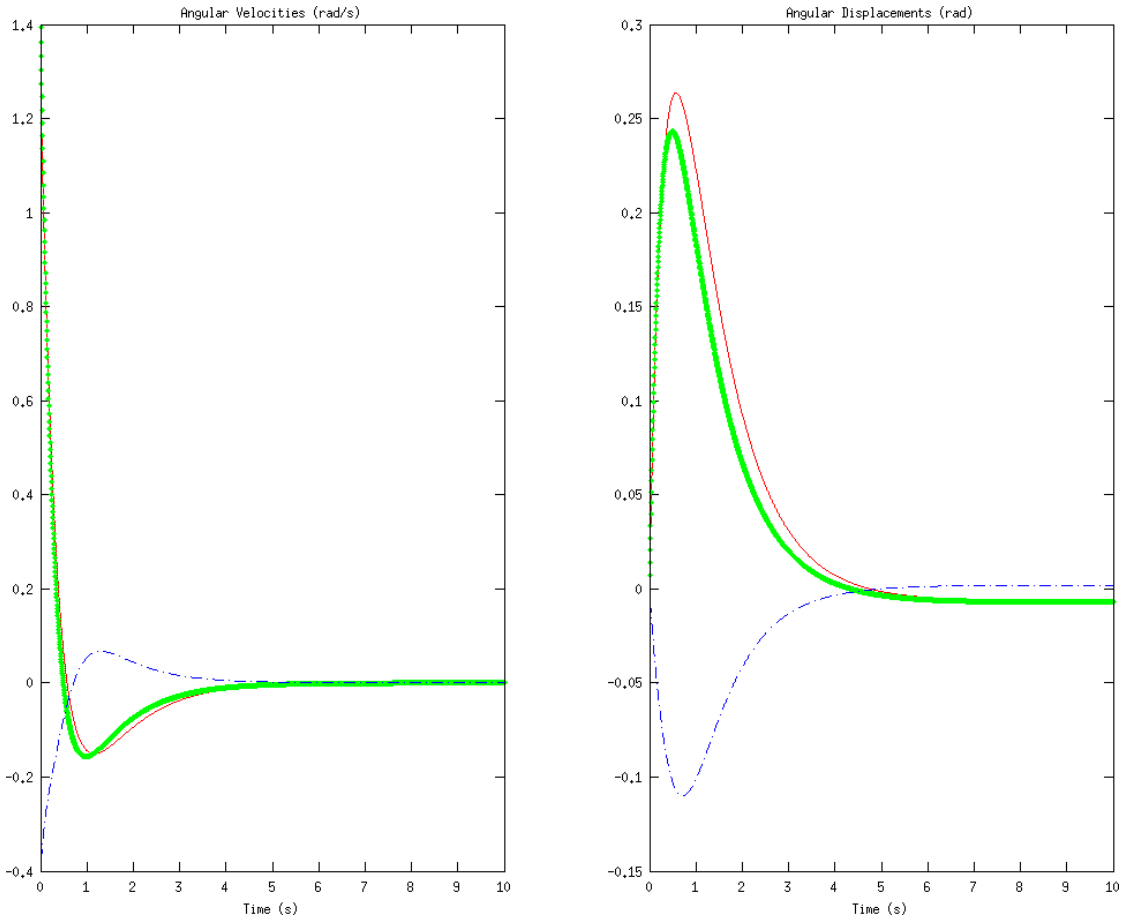

Figure 3: Left: Angular velocities. Right: angular displacements. ϕ, θ, ψ are coded as red, green, and blue.

This is a complete specification for our PD controller. We can simulate this controller using our simulation environment. The controller drives the angular velocities and angles to zero. However, note that the angles are not completely driven to zero. The average steady state error (error after 10 seconds of simulation) is approximately 0.3°. This is a common problem with using PD controllers for mechanical systems, and can be partially alleviated with a PID controller, as we will discuss in the next section.

In addition, note that since we are only controlling angular velocities, our positions and linear velocities do not converge to zero. However, the z position will remain constant, because we have constrained the total vertical thrust to be such that it keeps the quadcopter perfectly aloft, without ascending or descending. However, this is really nothing more than a curiosity. With the limited sensing that we have available to us, there is nothing we can do to control the linear position and velocity of the quadcopter. While in theory we could compute the linear velocities and positions from the angular velocities, in practice the values will be so noisy as to be completely useless. Thus, we will restrict ourselves to just stabilizing the quadcopter angle and angular velocity. (Traditionally, navigation is done by a human, and stabilization is there simply to make control easier for the human operator.)

We have implemented this PD control for use in our simulation. The controller is implemented as a function which is given some state (corresponding to controller state, not system state) and the sensor inputs, and must compute the inputs γ_i and the updated state. The code for a PD control follows.

```
% Compute system inputs and updated state.
% Note that input = [γ1, ..., γ4]
function [input, state] = pd_controller(state, thetadot)
    % Controller gains, tuned by hand and intuition.
    Kd = 4;
    Kp = 3;

    % Initialize the integral if necessary.
    if ~isfield(state, 'integral')
        params.integral = zeros(3, 1);
    end

    % Compute total thrust
    total = state.m * state.g / state.k / (cos(state.integral(1)) * cos(state.integral(2)));

    % Compute errors
    e = Kd * thetadot + Kp * params.integral;

    % Solve for the inputs, γi
    input = error2inputs(params, accels, total);

    % Update the state
    params.integral = params.integral + params.dt .* thetadot;
end
```

PID Control

PD controllers are nice in their simplicity and ease of implementation, but they are often inadequate for controlling mechanical systems. Especially in the presence of noise and disturbances, PD controllers will often lead to steady state error. A PID control is a PD control with another term added, which is proportional to the integral of the process variable. Adding an integral term causes any remaining steady-state error to build up and enact a change, so a PID controller should be able to track our trajectory (and stabilize the quadcopter) with a significantly smaller steady-state error. The equations remain identical to the ones presented in the PD case, but with an additional term in the error:

$$e_\phi = K_d\dot{\phi} + K_p\int_0^T \dot{\phi}\,\mathrm{d}t + K_i\int_0^T\int_0^T \dot{\phi}\,\mathrm{d}t\,\mathrm{d}t$$
$$e_\theta = K_d\dot{\theta} + K_p\int_0^T \dot{\theta}\,\mathrm{d}t + K_i\int_0^T\int_0^T \dot{\theta}\,\mathrm{d}t\,\mathrm{d}t$$
$$e_\psi = K_d\dot{\psi} + K_p\int_0^T \dot{\psi}\,\mathrm{d}t + K_i\int_0^T\int_0^T \dot{\psi}\,\mathrm{d}t\,\mathrm{d}t$$

However, PID controls come with their own shortcomings. One problem that commonly occurs with a PID control is known as integral windup.

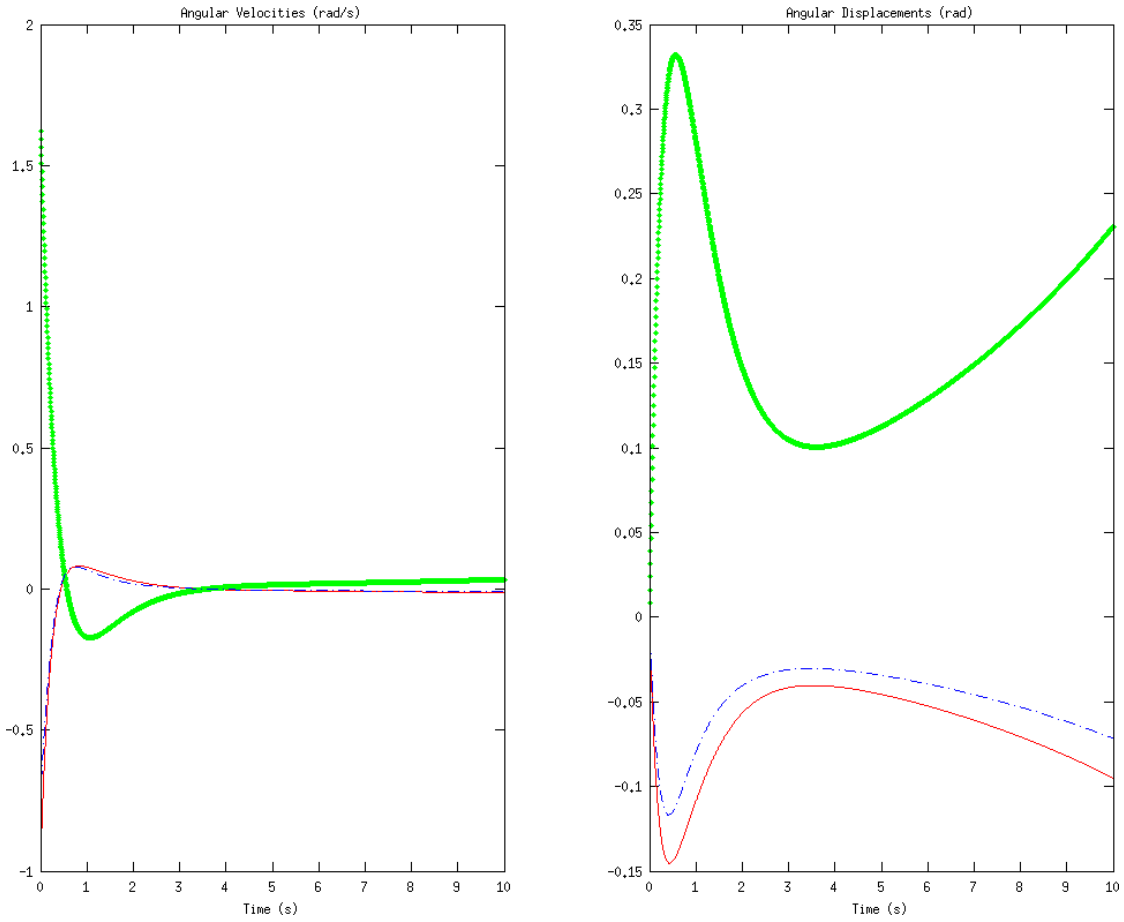

Figure 4: In some cases, integral wind-up can cause lengthy oscillations instead of settling. In other cases, wind-up may actually cause the system to become unstable, instead of taking longer to reach a steady state.

If there is a large disturbance in the process variable, this large disturbance is integrated over time, becoming a still larger control signal (due to the integral term). However, even once the system stabilizes, the integral is still large, thus causing the controller to overshoot its target. It may then begin a series of dieing down oscillations, become unstable, or simply take an incredibly long time to reach a steady state (Figure. 4). In order to avoid this, we disable the integral function until we reach something close to the steady state. Once we are in a controllable region near the desired steady state, we turn on the integral function, which pushes the system towards a low steady-state error.

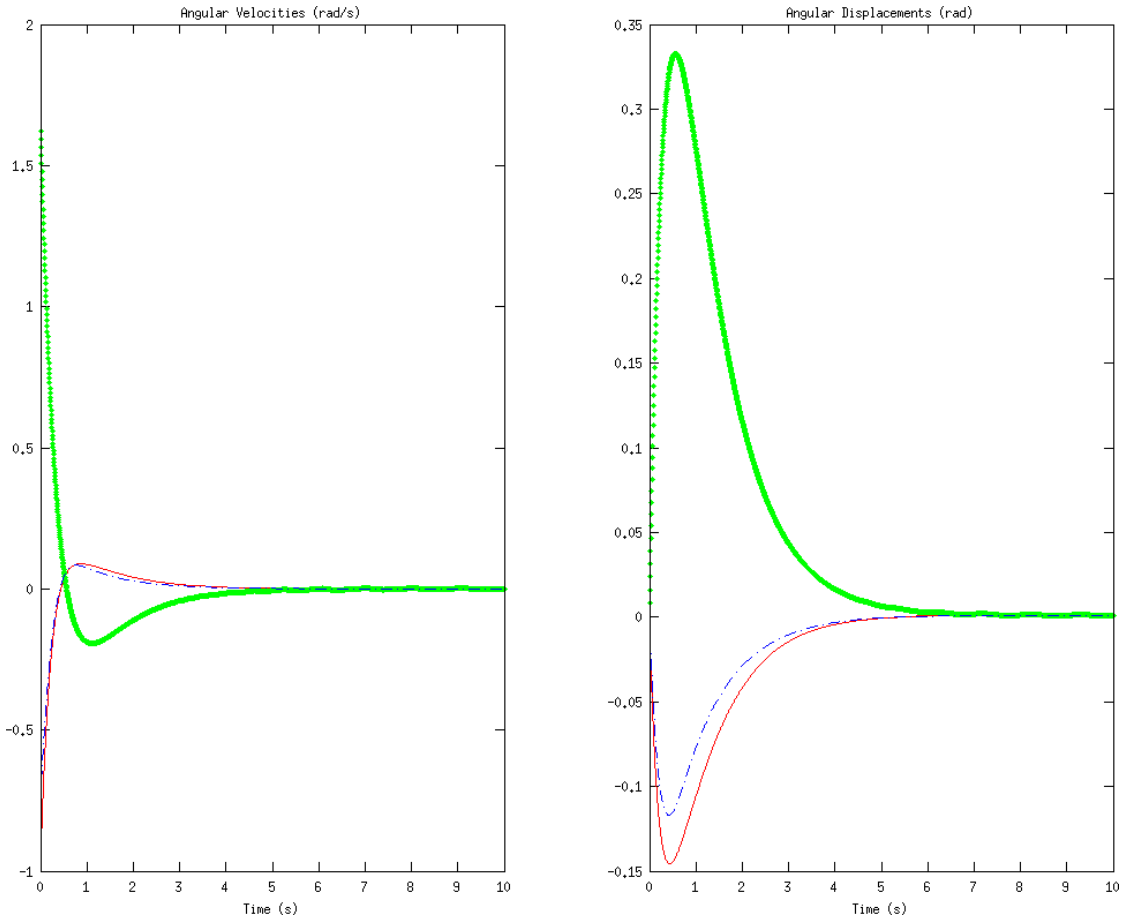

Figure 5: With a properly implemented PID, we achieve an error of approximately 0.06° after 10 seconds.

We have implemented this PID control for use in simulation, in the same way as with the PD controller shown earlier. Note that there is an additional parameter to tune in a PID. The disturbances used for all the test cases are identical, shown to compare the controllers.

```
% Compute system inputs and updated state.
% Note that input = [γ1, ..., γ4]
function [input, state] = pid_controller(state, thetadot)
    % Controller gains, tuned by hand and intuition.
    Kd = 4;
    Kp = 3;
    Ki = 5.5;

    % Initialize the integral if necessary.
    if ~isfield(state, 'integral')
        params.integral = zeros(3, 1);
        params.integral2 = zeros(3, 1);
    end

    % Prevent wind-up
    if max(abs(params.integral2)) > 0.01
        params.integral2(:) = 0;
    end

    % Compute total thrust
    total = state.m * state.g / state.k / (cos(state.integral(1)) * cos(state.integral(2)));

    % Compute errors
    e = Kd * thetadot + Kp * params.integral - Ki * params.integral2;

    % Solve for the inputs, γi
    input = error2inputs(params, accels, total);

    % Update the state
    params.integral = params.integral + params.dt .* thetadot;
    params.integral2 = params.integral2 + params.dt .* params.integral;
end
```

Automatic PID Tuning

Although PID control has the potential to perform very well, it turns out that the quality of the controller is highly dependent on the gain parameters. Tuning the parameters by hand may be quite difficult, as the ratios of the parameters is as important as the magnitudes of the parameters themselves; often, tuning parameters requires detailed knowledge of the system and an understanding of the conditions in which the PID control will be used. The parameters we chose previously were tuned by hand for good performance, simply by running simulations with many possibly disturbances and parameter values, and choosing something that worked reasonably well. This method is clearly suboptimal, not only because it can be very difficult and labor-intensive (and sometimes more or less impossible) but also because the resulting gains are not in any way guaranteed to be optimal or even close to optimal.

Ideally, we would be able to use an algorithm to analyze a system and output the "optimal" PID gains, for some reasonable definition of optimal. This problem has been studied in depth, and many methods have been proposed. Many of these methods require detailed knowledge of the system being modeled, and some rely on properties of the system, such as stability or linearity. The method we will use for choosing our PID parameters is a method known as *extremum seeking*.

Extremum seeking works exactly as the name implies. We define the "optimal" set of parameters as some vector $\vec{\theta} = (K_p, K_i, K_d)$ which minimizes some cost function $J(\vec{\theta})$. In our case, we would like to define a cost function that penalizes high error and error over extended durations of time. One candidate cost function is given by

$$J(\vec{\theta}) = \frac{1}{t_f - t_o} \int_{t_0}^{t_f} e(t, \vec{\theta})^2 \, \mathrm{d}t$$

where $e(t, \vec{\theta})$ is the error in following some reference trajectory with some initial disturbance using a set of parameters $\vec{\theta}$. Suppose we were able to somehow compute the gradient of this cost function, $\nabla J(\vec{\theta})$. In that case, we could iteratively improve our parameter vector by defining a parameter update rule

$$\vec{\theta}(k+1) = \vec{\theta}(k) - \alpha \nabla J(\vec{\theta})$$

where $\vec{\theta}(k)$ is the parameter vector after k iterations and α is some step size which dictates how much we adjust our parameter vector at each step of the iteration. As $k \to \infty$, the cost function $J(\vec{\theta})$ will approach a local minimum in the space of PID parameters.

The question remains as to how we can estimate $\nabla J(\vec{\theta})$. By definition,

$$\nabla J(\vec{\theta}) = \left(\frac{\partial}{\partial K_p} J(\vec{\theta}), \frac{\partial}{\partial K_i} J(\vec{\theta}), \frac{\partial}{\partial K_d} J(\vec{\theta}) \right).$$

We know how to compute $J(\vec{\theta})$. Using this, we can approximate the derivative with respect to any of the gains numerically, simply by computing

$$\frac{\partial}{\partial K} J(\vec{\theta}) \approx \frac{J(\vec{\theta} + \delta \cdot \hat{u}_K) - J(\vec{\theta} - \delta \cdot \hat{u}_K)}{2\delta}$$

where $\hat{u}_K$ is the unit vector in the K direction. As $\delta \to 0$, this approximation becomes better. Using this approximation, we can minimize our cost function and achieve locally optimal PID parameters. We can start with randomly initialized positive weights, disturb the system in some set manner, evaluate $J(\vec{\theta})$ by simulating the system for different PID parameters, and then compute the gradient. Then, using the method of gradient descent, we can iteratively oprtimize our gains until we have some form of convergence.

The gradient descent method does, however, have several problems. First of all, although it finds a local minimum, that minimum is only guaranteed to be a *local* minimum - there may be other minima which are better global minima. In order to avoid choosing suboptimal local minima in the cost function, we repeat our optimization several times, and

choose the best result. We initialize our PID parameters randomly, so each time we run the optimization we will get a different result. In addition, instead of choosing disturbance and then optimizing the response to that disturbance, we use several random disturbances at each iteration and use the average response to compute costs and gradients. This ensures that our parameters are general and not optimized for a specific disturbance. In addition, we vary the step size and the number of disturbances to try per iteration, in order to increase the sensitivity of our results as our iteration continues. We stop iterations when we detect a steady state, which we do by computing a linear regression on the most recent costs and iterating until the slope is statistically indistinguishable from zero using a 99% confidence interval.

Using our quadcopter simulation, we can define a function that computes the cost for a given set of PID parameters.

```
function J = cost(theta)
    % Create a controller using the given gains.
    control = controller('pid', theta(1), theta(2), theta(3));

    % Perform a simulation.
    data = simulate(control);

    % Compute the integral, 1/(tf - t0) * int_t0^tf e(t)^2 dt
    t0 = 0;
    tf = 1;
    J = 1/(tf - t0) * sum(data.theta(data.t >= t0 & data.t <= tf) .^ 2) * data.dt;
end
```

We can use this function to approximate a derivative with respect to a gain:

```
% Compute derivative with respect to first parameter.
delta = 0.01;
var = [1, 0, 0];
derivative = (cost(theta + delta * var) - cost(theta - delta * var)) / (2 * delta);
```

We can then use our gradient descent method (with all modifications described above) to minimize the cost function and obtain a good set of PID parameters. We can verify that our tuning method is working by visualizing the cost function versus the iteration number, and seeing that the cost function is indeed going down and stabilizing at a local minimum (Figure. 6).

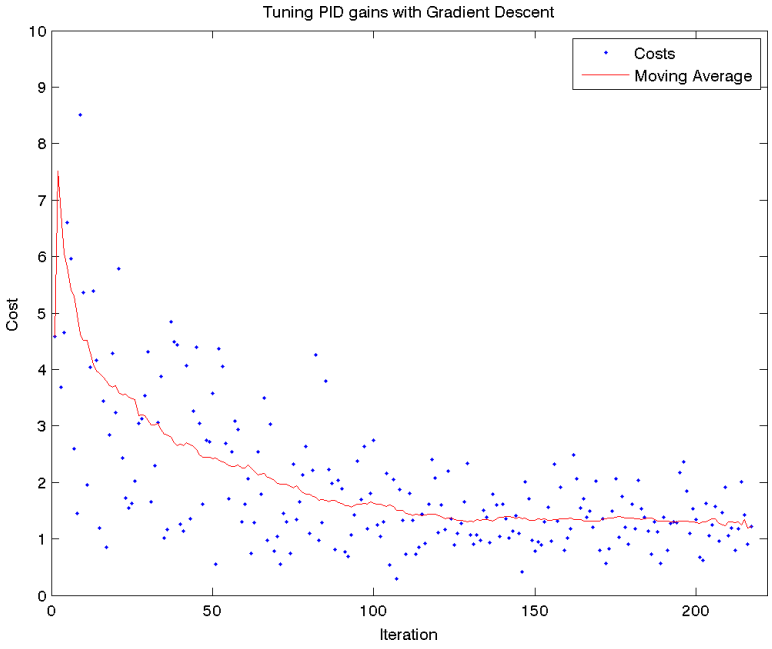

Figure 6: Cost function plotted as a function of iteration number, along with moving average. Tuning stops when the slope of the moving average becomes statistically indistinguishable from zero with a 99% confidence interval.

We can compare the manually-chosen PID parameters with those designed by the algorithm.

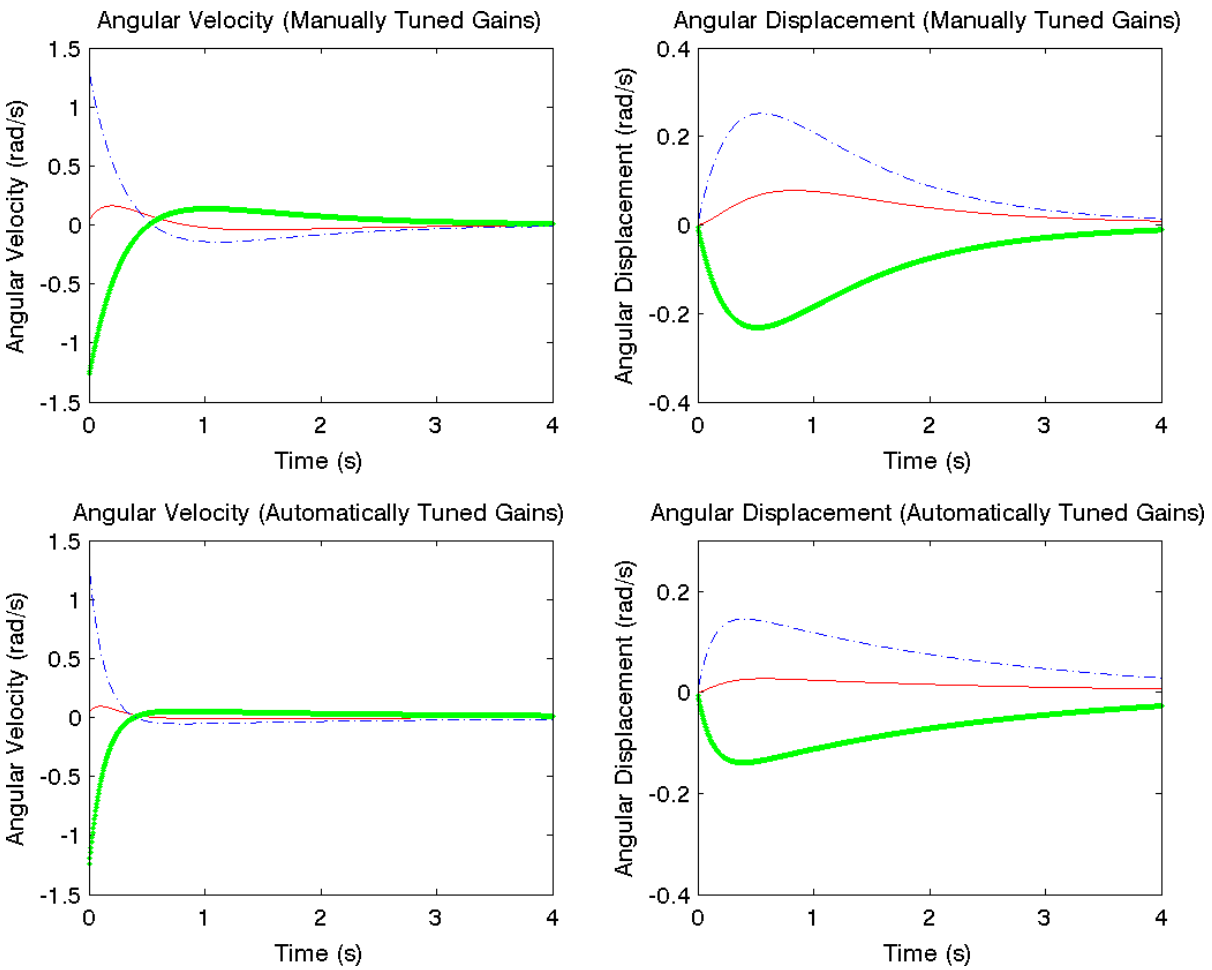

Figure 7: Top: Angular velocities and angular displacements, using manually tuned PID controller. Bottom: Angular velocities and angular displacements, using automatically tuned PID controller.

The automatically-chosen PID parameters do significantly better overall. They have significantly smaller swings in value, overshoot significantly less, and converge faster. However, the error in the angular displacement takes longer to converge to zero with the automatically tuned parameters than with the manually turned parameters, although the initial convergence is much better when the parameters are chosen via gradient descent. This is due to the fact that our cost function emphasizes squared error, and thus gives priority to minimizing overall error magnitude rather than long-term convergence. We could easily modify our cost function to give higher priority to long-term error, in which case the automatically-tuned parameters are likely to do better.

Conclusion

We derived equations of motion for a quadcopter, starting with the voltage-torque relation for the brushless motors and working through the quadcopter kinematics and dynamics. We ignored aerodynamical effects such as blade-flapping and non-zero free stream velocity, but included air friction as a linear drag force in all directions. We used the equations of motion to create a simulator in which to test and visualize quadcopter control mechanisms.

We began with a simple PD controller. Although the PD controller worked, it left a significant steady-state error. In order to decrease the steady-state error, we added an integral term in order to create a PID controller. We tested the PID controller (with minor modifications to prevent integral wind-up) and found that it was better at preventing steady-state error than the PD controller when presented with the same disturbances and using the same proportional and derivative gains. We also found that tuning the PID controller was difficult, and would often lead to an unstable system for unknown reasons. In order to avoid the difficulty of PID tuning and find the optimal set of parameters, we used a gradient-descent based extremum seeking method in order to numerically estimate gradients of a cost function in PID-parameter space and iteratively choose a set of parameters to minimize the cost function. We found that the resulting controller was significantly better than the one using manually turned parameters.

References

[1] Luukkonen, Teppo. *"Modelling and control of quadcopter"*. Aalto University, School of Science. August 22, 2011.

[2] Hoffman, Huang, et. al. *"Quadrotor Helicopter Flight Dynamics and Control: Theory and Experiment"*. American Institute of Aeronautics and Astronautics. August 23, 2007.

[3] Rademacher, Wayne. *"Brushless Motors"*. Twin City Radio Controllers. January 2008.

[4] Killingsworth, Nick; Krstic, Miroslav. *"PID Tuning Using Extremum Seeking: Online, Model-Free Performance Optimization"*. IEEE Control Systems Magazine. February 2006.

[5] Choi, Krstic, Ariyur, and Lee. *"Extremum Seeking control for discrete-time systems"*. IEEE Transactions on Automatic Control, vol. 47, pp. 318-323. February 2002.

[6] Huang, Hoffman, Waslander, and Tomlin. *"Aerodynamics and Control of Autonomous Quadrotor Helicopters in Aggressive Maneuvering"*. Robotics and Automation. May 2009.

[7] Thomas S. Alderete. *"Simulator Aero Model Implementation"*.

[8] Bouabdallah, Murrieri, and Siegwart. *"Design and Control of an Indoor Micro Quadrotor"*. Robotics and Automation. May 2004.

Appendix

Simulation

```
% Perform a simulation of a quadcopter, from t = 0 through t = 10.
% As an argument, take a controller function. This function must accept
% a struct containing the physical parameters of the system and the current
% gyro readings. The controller may use the strust to store persistent state, and
% return this state as a second output value. If no controller is given,
% a simulation is run with some pre-determined inputs.
% The output of this function is a data struct with the following fields, recorded
% at each time-step during the simulation:
%
% data =
%
%         x: [3xN double]
%     theta: [3xN double]
%       vel: [3xN double]
%    angvel: [3xN double]
%         t: [1xN double]
%     input: [4xN double]
%        dt: 0.0050
function result = simulate(controller, tstart, tend, dt)
    % Physical constants.
    g = 9.81;
    m = 0.5;
    L = 0.25;
    k = 3e-6;
    b = 1e-7;
    I = diag([5e-3, 5e-3, 10e-3]);
    kd = 0.25;

    % Simulation times, in seconds.
    if nargin < 4
        tstart = 0;
        tend = 4;
        dt = 0.005;
    end
    ts = tstart:dt:tend;

    % Number of points in the simulation.
    N = numel(ts);

    % Output values, recorded as the simulation runs.
    xout = zeros(3, N);
    xdotout = zeros(3, N);
    thetaout = zeros(3, N);
    thetadotout = zeros(3, N);
    inputout = zeros(4, N);

    % Struct given to the controller. Controller may store its persistent state in it.
    controller_params = struct('dt', dt, 'I', I, 'k', k, 'L', L, 'b', b, 'm', m, 'g', g);

    % Initial system state.
    x = [0; 0; 10];
    xdot = zeros(3, 1);
    theta = zeros(3, 1);

    % If we are running without a controller, do not disturb the system.
    if nargin == 0
        thetadot = zeros(3, 1);
    else
        % With a control, give a random deviation in the angular velocity.
        % Deviation is in degrees/sec.
        deviation = 300;
        thetadot = deg2rad(2 * deviation * rand(3, 1) - deviation);
    end

    ind = 0;
    for t = ts
        ind = ind + 1;

        % Get input from built-in input or controller.
        if nargin == 0
            i = input(t);
        else
            [i, controller_params] = controller(controller_params, thetadot);
        end

        % Compute forces, torques, and accelerations.
        omega = thetadot2omega(thetadot, theta);
        a = acceleration(i, theta, xdot, m, g, k, kd);
        omegadot = angular_acceleration(i, omega, I, L, b, k);

        % Advance system state.
        omega = omega + dt * omegadot;
```

```
        thetadot = omega2thetadot(omega, theta);
        theta = theta + dt * thetadot;
        xdot = xdot + dt * a;
        x = x + dt * xdot;

        % Store simulation state for output.
        xout(:, ind) = x;
        xdotout(:, ind) = xdot;
        thetaout(:, ind) = theta;
        thetadotout(:, ind) = thetadot;
        inputout(:, ind) = i;
    end

    % Put all simulation variables into an output struct.
    result = struct('x', xout, 'theta', thetaout, 'vel', xdotout, ...
                    'angvel', thetadotout, 't', ts, 'dt', dt, 'input', inputout);
end

% Arbitrary test input.
function in = input(t)
    in = zeros(4, 1);
    in(:) = 700;
    in(1) = in(1) + 150;
    in(3) = in(3) + 150;
    in = in .^ 2;
end

% Compute thrust given current inputs and thrust coefficient.
function T = thrust(inputs, k)
    T = [0; 0; k * sum(inputs)];
end

% Compute torques, given current inputs, length, drag coefficient, and thrust coefficient.
function tau = torques(inputs, L, b, k)
    tau = [
        L * k * (inputs(1) - inputs(3))
        L * k * (inputs(2) - inputs(4))
        b * (inputs(1) - inputs(2) + inputs(3) - inputs(4))
    ];
end

% Compute acceleration in inertial reference frame
% Parameters:
%   g: gravity acceleration
%   m: mass of quadcopter
%   k: thrust coefficient
%   kd: global drag coefficient
function a = acceleration(inputs, angles, vels, m, g, k, kd)
    gravity = [0; 0; -g];
    R = rotation(angles);
    T = R * thrust(inputs, k);
    Fd = -kd * vels;
    a = gravity + 1 / m * T + Fd;
end

% Compute angular acceleration in body frame
% Parameters:
%   I: inertia matrix
function omegad = angular_acceleration(inputs, omega, I, L, b, k)
    tau = torques(inputs, L, b, k);
    omegad = inv(I) * (tau - cross(omega, I * omega));
end

% Convert derivatives of roll, pitch, yaw to omega.
function omega = thetadot2omega(thetadot, angles)
    phi = angles(1);
    theta = angles(2);
    psi = angles(3);
    W = [
        1, 0, -sin(theta)
        0, cos(phi), cos(theta)*sin(phi)
        0, -sin(phi), cos(theta)*cos(phi)
    ];
    omega = W * thetadot;
end

% Convert omega to roll, pitch, yaw derivatives
function thetadot = omega2thetadot(omega, angles)
    phi = angles(1);
    theta = angles(2);
    psi = angles(3);
    W = [
        1, 0, -sin(theta)
        0, cos(phi), cos(theta)*sin(phi)
        0, -sin(phi), cos(theta)*cos(phi)
    ];
    thetadot = inv(W) * omega;
end
```

Controller

```
% Create a controller based on it's name, using a look-up table.
function c = controller(name, Kd, Kp, Ki)
    % Use manually tuned parameters, unless arguments provide the parameters.
    if nargin == 1
        Kd = 4;
        Kp = 3;
        Ki = 5.5;
    elseif nargin == 2 || nargin > 4
        error('Incorrect number of parameters.');
    end

    if strcmpi(name, 'pd')
        c = @(state, thetadot) pd_controller(state, thetadot, Kd, Kp);
    elseif strcmpi(name, 'pid')
        c = @(state, thetadot) pid_controller(state, thetadot, Kd, Kp, Ki);
    else
        error(sprintf('Unknown controller type "%s"', name));
    end
end

% Implement a PD controller. See simulate(controller).
function [input, state] = pd_controller(state, thetadot, Kd, Kp)
    % Initialize integral to zero when it doesn't exist.
    if ~isfield(state, 'integral')
        state.integral = zeros(3, 1);
    end

    % Compute total thrust.
    total = state.m * state.g / state.k / ...
        (cos(state.integral(1)) * cos(state.integral(2)));

    % Compute PD error and inputs.
    err = Kd * thetadot + Kp * state.integral;
    input = err2inputs(state, err, total);

    % Update controller state.
    state.integral = state.integral + state.dt .* thetadot;
end

% Implement a PID controller. See simulate(controller).
function [input, state] = pid_controller(state, thetadot, Kd, Kp, Ki)
    % Initialize integrals to zero when it doesn't exist.
    if ~isfield(state, 'integral')
        state.integral = zeros(3, 1);
        state.integral2 = zeros(3, 1);
    end

    % Prevent wind-up
    if max(abs(state.integral2)) > 0.01
        state.integral2(:) = 0;
    end

    % Compute total thrust.
    total = state.m * state.g / state.k / ...
          (cos(state.integral(1)) * cos(state.integral(2)));

    % Compute error and inputs.
    err = Kd * thetadot + Kp * state.integral - Ki * state.integral2;
    input = err2inputs(state, err, total);

    % Update controller state.
    state.integral = state.integral + state.dt .* thetadot;
    state.integral2 = state.integral2 + state.dt .* state.integral;
end

% Given desired torques, desired total thrust, and physical parameters,
% solve for required system inputs.
function inputs = err2inputs(state, err, total)
    e1 = err(1);
    e2 = err(2);
    e3 = err(3);
    Ix = state.I(1, 1);
    Iy = state.I(2, 2);
    Iz = state.I(3, 3);
    k = state.k;
    L = state.L;
    b = state.b;

    inputs = zeros(4, 1);
    inputs(1) = total/4 -(2 * b * e1 * Ix + e3 * Iz * k * L)/(4 * b * k * L);
    inputs(2) = total/4 + e3 * Iz/(4 * b) - (e2 * Iy)/(2 * k * L);
    inputs(3) = total/4 -(-2 * b * e1 * Ix + e3 * Iz * k * L)/(4 * b * k * L);
    inputs(4) = total/4 + e3 * Iz/(4 * b) + (e2 * Iy)/(2 * k * L);
end
```

Automatic PID Tuning

```
% Compute an optimal set of PID parameters
% by initializing the parameters randomly,
% minimizing a cost function using a numerical gradient
% estimate, repeating this several times, and
% choosing the best result.
function theta = tune();
    % How many times should we repeat to try to get better results?
    attempts = 10;

    % Keep track of minimum cost so far, and best parameter set.
    min_cost = 1e10;
    best_theta = -1;

    for i = 1:attempts
        % Compute next set of parameters and their costs.
        [theta, costs] = minimize;

        % If this is the best we've seen so far, store it.
        if costs(end) < min_cost
            min_cost = costs(end);
            best_theta = theta;
        end
    end

    % Return best parameters.
    theta = best_theta;
end

% Minimize a cost function by estimating the gradient
% numerically and using modified gradient descent to
% choose the best set of parameters.
function [theta, costs] = minimize()
    % Initialize weights randomly.
    theta = 1 * rand(1, 3);

    % Use a small step size α.
    alpha = 0.03;

    % Maximum number of iterations to use.
    % In general, we should not reach this,
    % as we should get to a steady-state earlier.
    max_iterations = 500;

    % Each time we compute cost and gradient, we are actually
    % computing them using different disturbances. In order to make our
    % control parameters general, we take the average of several costs and gradients
    % in order to obtain our estimates for a given parameter set. This variable
    % indicates how many different measurements we average. As we iterate longer,
    % we may want to increase this in order to make our gradients more precise.
    average_length = 3;

    for iteration = 1:max_iterations
        disp(sprintf('Iteration %d...', iteration));

        % Compute costs (with averaging) for the current parameters.
        costs(iteration) = mean_value(@cost, theta, average_length);

        % Check if we can stop. We stop if we have reached a steady-state.
        % In order to decide whether a steady-state has been reached, we
        % look at the previous fifty costs. We fit a line to the graph of
        % costs vs iterations, and if the slope of that line is statisticially
        % insignificant (the 99% confidence interval includes zero), we
        % say that we have reached a steady state, and stop iterating.
        num_costs = 50;
        if iteration > num_costs + 5
            % Previous fifty costs.
            recent_costs = costs(end - num_costs + 1:end);

            % Compute linear regression, with a bias term b(1) and a slope b(2).
            % Also, compute 99% confidence intervals for the bias and slope.
            [b, int] = regress(recent_costs', [ones(num_costs, 1) (1:num_costs)'], 0.99);

            % Find the boundaries of the slope confidence interval.
            % If zero is in-between them, our slope is negligible, and
            % further training is unnecessary. Stop iterating.
            larger = max(int(2, :));
            smaller = min(int(2, :));
            if 0 < larger && 0 > smaller
                break;
            end
        end

        % Change step size and averaging to adjust for training duration.
        % After longer training times, we may want to decrease step size and
        % increase the number of averaged samples, so that our algorithm may
```

```
        % be more sensitive to small changes and avoid overshooting the minimum.
        if iteration > 100
            alpha = 0.001;
            average_length = 8;
        elseif iteration > 200
            average_length = 15;
            alpha = 0.0005;
        end

        % Compute gradient for our parameters (with averaging).
        grad = mean_value(@gradient, theta, average_length);

        % Adjust parameters using step size and gradient.
        theta = theta - alpha * grad;
    end
end

% Given a function that has some random component and
% may return different values for the same input, as well
% as an input for that function, compute N values for
% that function and return their average.
function value = mean_value(func, input, n)
    % Compute first one out of loop, to determine size(value).
    value = func(input);

    for i = 2:n
        value = value + func(input);
    end
    value = value / n;
end

% Numerically estimate the gradient of the cost function
% at a particular point in PID parameter space.
function grad = gradient(theta)
    % Use a very small displacement to estimate the limit.
    delta = 0.001;

    % Store random seed, so that all simulations are using the same disturbance.
    % Although different gradients may use different disturbances, we want the different
    % components of the gradient to be computed using the same simulation.
    s = rng;

    for i = 1:length(theta)
        var = zeros(size(theta));
        var(i) = 1;

        % Restore the random seed for each cost computation.
        % This way, the simulation that's done is the same every time.
        rng(s); left_cost = cost(theta + delta * var);
        rng(s); right_cost = cost(theta - delta * var);

        % Compute gradient with respect to ith parameter.
        grad(i) = (left_cost - right_cost) / (2 * delta);
    end
end

% Compute the cost function for a given parameter set.
% The cost function is defined as:
%   J(θ) = 1/(t_f - t_0) ∫_{t_0}^{t_f} e(t, θ)^2 dt
% where e(t, θ) is the error at time t.
function J = cost(theta)
    % Create a controller using the given gains.
    control = controller('pid', theta(1), theta(2), theta(3));

    % Perform a simulation. Only simulate the first second, and
    % use a relatively large time-step. We do many simulations for
    % each iteration of the tuning, so we need each simulation to be quite fast.
    data = simulate(control, 0, 1, 0.05);

    % Compute the integral, 1/(t_f - t_0) ∫_{t_0}^{t_f} e(t)^2 dt
    errors = sqrt(sum(data.theta .^ 2));
    J = sum(errors .^ 2) * data.dt;
end
```

Visualization

```
% Visualize the quadcopter simulation as an animation of a 3D quadcopter.
function h = visualize_test(data)
    % Create a figure with three parts. One part is for a 3D visualization,
    % and the other two are for running graphs of angular velocity and displacement.
    figure; plots = [subplot(3, 2, 1:4), subplot(3, 2, 5), subplot(3, 2, 6)];
    subplot(plots(1));
    pause;

    % Create the quadcopter object. Returns a handle to
    % the quadcopter itself as well as the thrust-display cylinders.
    [t thrusts] = quadcopter;

    % Set axis scale and labels.
    axis([-10 30 -20 20 5 15]);
    zlabel('Height');
    title('Quadcopter Flight Simulation');

    % Animate the quadcopter with data from the simulation.
    animate(data, t, thrusts, plots);
end

% Animate a quadcopter in flight, using data from the simulation.
function animate(data, model, thrusts, plots)
    % Show frames from the animation. However, in the interest of speed,
    % skip some frames to make the animation more visually appealing.
    for t = 1:2:length(data.t)
        % The first, main part, is for the 3D visualization.
        subplot(plots(1));

        % Compute translation to correct linear position coordinates.
        dx = data.x(:, t);
        move = makehgtform('translate', dx);

        % Compute rotation to correct angles. Then, turn this rotation
        % into a 4x4 matrix represting this affine transformation.
        angles = data.theta(:, t);
        rotate = rotation(angles);
        rotate = [rotate zeros(3, 1); zeros(1, 3) 1];

        % Move the quadcopter to the right place, after putting it in the correct orientation.
        set(model,'Matrix', move * rotate);

        % Compute scaling for the thrust cylinders. The lengths should represent relative
        % strength of the thrust at each propeller, and this is just a heuristic that seems
        % to give a good visual indication of thrusts.
        scales = exp(data.input(:, t) / min(abs(data.input(:, t))) + 5) - exp(6) +  1.5;
        for i = 1:4
            % Scale each cylinder. For negative scales, we need to flip the cylinder
            % using a rotation, because makehgtform does not understand negative scaling.
            s = scales(i);
            if s < 0
                scalez = makehgtform('yrotate', pi)  * makehgtform('scale', [1, 1, abs(s)]);
            elseif s > 0
                scalez = makehgtform('scale', [1, 1, s]);
            end

            % Scale the cylinder as appropriate, then move it to
            % be at the same place as the quadcopter propeller.
            set(thrusts(i), 'Matrix', move * rotate * scalez);
        end

        % Update the drawing.
        drawnow;

        % Use the bottom two parts for angular velocity and displacement.
        subplot(plots(2));
        multiplot(data, data.angvel, t);
        xlabel('Time (s)');
        ylabel('Angular Velocity (rad/s)');
        title('Angular Velocity');

        subplot(plots(3));
        multiplot(data, data.theta, t);
        xlabel('Time (s)');
        ylabel('Angular Displacement (rad)');
        title('Angular Displacement');
    end
end

% Plot three components of a vector in RGB.
function multiplot(data, values, ind)
    % Select the parts of the data to plot.
    times = data.t(:, 1:ind);
    values = values(:, 1:ind);
```

```
    % Plot in RGB, with different markers for different components.
    plot(times, values(1, :), 'r-', times, values(2, :), 'g.', times, values(3, :), 'b-.');

    % Set axes to remain constant throughout plotting.
    xmin = min(data.t);
    xmax = max(data.t);
    ymin = 1.1 * min(min(values));
    ymax = 1.1 * max(max(values));
    axis([xmin xmax ymin ymax]);
end

% Draw a quadcopter. Return a handle to the quadcopter object
% and an array of handles to the thrust display cylinders.
% These will be transformed during the animation to display
% relative thrust forces.
function [h thrusts] = quadcopter()
    % Draw arms.
    h(1) = prism(-5, -0.25, -0.25, 10, 0.5, 0.5);
    h(2) = prism(-0.25, -5, -0.25, 0.5, 10, 0.5);

    % Draw bulbs representing propellers at the end of each arm.
    [x y z] = sphere;
    x = 0.5 * x;
    y = 0.5 * y;
    z = 0.5 * z;
    h(3) = surf(x - 5, y, z, 'EdgeColor', 'none', 'FaceColor', 'b');
    h(4) = surf(x + 5, y, z, 'EdgeColor', 'none', 'FaceColor', 'b');
    h(5) = surf(x, y - 5, z, 'EdgeColor', 'none', 'FaceColor', 'b');
    h(6) = surf(x, y + 5, z, 'EdgeColor', 'none', 'FaceColor', 'b');

    % Draw thrust cylinders.
    [x y z] = cylinder(0.1, 7);
    thrusts(1) = surf(x, y + 5, z, 'EdgeColor', 'none', 'FaceColor', 'm');
    thrusts(2) = surf(x + 5, y, z, 'EdgeColor', 'none', 'FaceColor', 'y');
    thrusts(3) = surf(x, y - 5, z, 'EdgeColor', 'none', 'FaceColor', 'm');
    thrusts(4) = surf(x - 5, y, z, 'EdgeColor', 'none', 'FaceColor', 'y');

    % Create handles for each of the thrust cylinders.
    for i = 1:4
        x = hgtransform;
        set(thrusts(i), 'Parent', x);
        thrusts(i) = x;
    end

    % Conjoin all quadcopter parts into one object.
    t = hgtransform;
    set(h, 'Parent', t);
    h = t;
end

% Draw a 3D prism at (x, y, z) with width w,
% length l, and height h. Return a handle to
% the prism object.
function h = prism(x, y, z, w, l, h)
    [X Y Z] = prism_faces(x, y, z, w, l, h);

    faces(1, :) = [4 2 1 3];
    faces(2, :) = [4 2 1 3] + 4;
    faces(3, :) = [4 2 6 8];
    faces(4, :) = [4 2 6 8] - 1;
    faces(5, :) = [1 2 6 5];
    faces(6, :) = [1 2 6 5] + 2;

    for i = 1:size(faces, 1)
        h(i) = fill3(X(faces(i, :)), Y(faces(i, :)), Z(faces(i, :)), 'r'); hold on;
    end

    % Conjoin all prism faces into one object.
    t = hgtransform;
    set(h, 'Parent', t);
    h = t;
end

% Compute the points on the edge of a prism at
% location (x, y, z) with width w, length l, and height h.
function [X Y Z] = prism_faces(x, y, z, w, l, h)
    X = [x x x x x+w x+w x+w x+w];
    Y = [y y y+l y+l y y y+l y+l];
    Z = [z z+h z z+h z z+h z z+h];
end
```

Rotation Matrix

```
% Compute rotation matrix for a set of angles.
function R = rotation(angles)
    phi = angles(3);
    theta = angles(2);
    psi = angles(1);

    R = zeros(3);
    R(:, 1) = [
        cos(phi) * cos(theta)
        cos(theta) * sin(phi)
        - sin(theta)
    ];
    R(:, 2) = [
        cos(phi) * sin(theta) * sin(psi) - cos(psi) * sin(phi)
        cos(phi) * cos(psi) + sin(phi) * sin(theta) * sin(psi)
        cos(theta) * sin(psi)
    ];
    R(:, 3) = [
        sin(phi) * sin(psi) + cos(phi) * cos(psi) * sin(theta)
        cos(psi) * sin(phi) * sin(theta) - cos(phi) * sin(psi)
        cos(theta) * cos(psi)
    ];
end
```