GUDrones: Quadcopter Dynamics and Control: A Practical Approach

Stefan Vuckovic

March 3, 2025

1 Introduction

This report compiles my research for GUDrones, and should hopefully outline the general principles of Quadcopter Dynamics in Relation to Control, to all previous education levels (assuming an engineering background :))

Report Strucutre

This document contains explanations at different levels of complexity. Throughout the report, you'll find:

- **Core concepts** - The main text explains essential information
- **Beginner notes** - Simplified explanations for society members who haven't covered principle ideas in University yet
- **Application insights** - Practical implementation
- **Advanced topics** - More complex material for those interested in deeper understanding

2 Quadcopter Basics

2.1 Frame Configuration

A standard quadcopter uses an "X" configuration with four motors:

- Front-left: Motor 1 (counterclockwise rotation)

- Front-right: Motor 2 (clockwise rotation)
- Rear-right: Motor 3 (counterclockwise rotation)
- Rear-left: Motor 4 (clockwise rotation)

This alternating rotation pattern helps balance the reactive torques and enables yaw control.

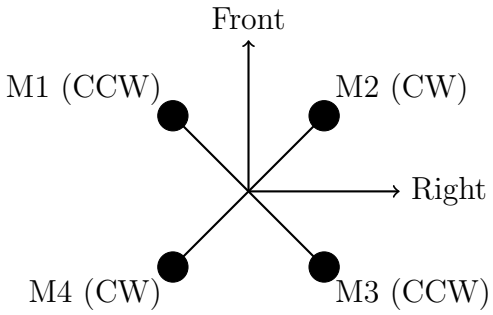

Figure 1: Standard X-configuration quadcopter (top view)

Beginner Note: Why Alternating Rotation?

When a propeller spins, it creates a reaction force that would make the drone itself spin in the opposite direction (like a helicopter without a tail rotor). By having half the propellers spin clockwise and half counter-clockwise, these forces cancel out, keeping the drone stable. This design is what makes quadcopters popular and much simpler than helicopters.

2.2 Coordinate Systems

Two main coordinate frames are used:

- Earth frame: Fixed reference with z-axis pointing upward
- Body frame: Attached to the quadcopter with x-axis pointing forward

The orientation of the quadcopter is defined by three angles:

- Roll (ϕ): Rotation around the x-axis
- Pitch (θ): Rotation around the y-axis
- Yaw (ψ): Rotation around the z-axis

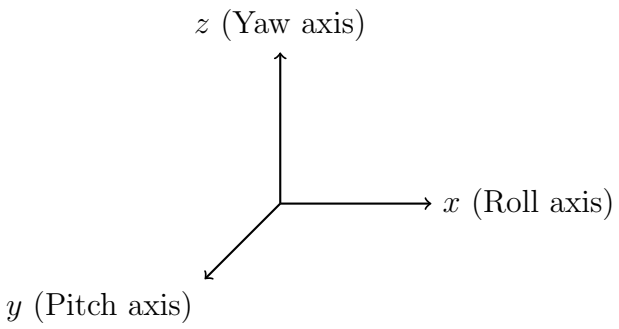

Figure 2: Body frame coordinate system

Beginner Note: Understanding Drone Movement

Think of these angles like airplane controls:

- **Roll**: Tilting left and right
- **Pitch**: Tilting forward and backward
- **Yaw**: Rotating left and right

3 Basic Physics

3.1 Forces and Moments

Each motor produces thrust proportional to the square of its angular velocity:

$$T_i = k_T \omega_i^2 \tag{1}$$

Where:

- T_i is the thrust from motor i
- k_T is the thrust coefficient
- ω_i is the angular velocity of motor i

Each motor also produces a torque:

$$\tau_i = k_M \omega_i^2 \tag{2}$$

Where k_M is the torque coefficient.

Application Insight: Determining Thrust Coefficients

The thrust coefficient k_T can be determined experimentally by mounting a motor-propeller combination on a load cell and measuring thrust at different rotation speeds. For consumer drones, k_T typically ranges from 1×10^{-6} to 1×10^{-5} N/(rad/s)^2 depending on propeller size and design. The torque coefficient k_M is usually about 1-2% of the thrust coefficient.

3.2 Control Principles

Quadcopter movement is controlled by varying the speeds of the four motors:

- Throttle (altitude): Increase/decrease all motor speeds equally
- Roll: Increase right motors, decrease left motors (or vice versa)
- Pitch: Increase front motors, decrease rear motors (or vice versa)
- Yaw: Increase clockwise motors, decrease counterclockwise motors (or vice versa)

Beginner Note: Basic Flight Controls

A helpful way to understand quadcopter controls is to remember:

- To move in any direction, the drone tilts that way (by creating more thrust on the opposite side)
- To turn (yaw), the drone increases the power of motors spinning in one direction while decreasing the opposite ones
- To maintain level flight while moving, the controller constantly adjusts all motors to keep the desired angle

4 Simplified Dynamic Model

For practical controller implementation, we can use a simplified model that treats roll, pitch, and yaw as decoupled systems.

4.1 Linear Approximation

For small angles (near hover), we can approximate:

$$\begin{aligned} \ddot{\phi} &= \frac{\tau_\phi}{I_{xx}} \\ \ddot{\theta} &= \frac{\tau_\theta}{I_{yy}} \\ \ddot{\psi} &= \frac{\tau_\psi}{I_{zz}} \end{aligned} \tag{3}$$

Where:

- I_{xx}, I_{yy}, I_{zz} are moments of inertia
- τ_ϕ, τ_θ, τ_ψ are the torques around each axis

Advanced Note: Simplification Limitations

This linear approximation is only valid for small angles (roughly ± 15). For aggressive maneuvers, a full nonlinear model incorporating gyroscopic effects, full rotational dynamics, and cross-coupling would be needed. However, for stable hovering and gentle flight, this simplified model works remarkably well and significantly reduces computational requirements.

4.2 Motor Mixing

The motor speeds are determined from the desired thrust and torques using a mixing matrix:

$$\begin{bmatrix} \omega_1^2 \\ \omega_2^2 \\ \omega_3^2 \\ \omega_4^2 \end{bmatrix} = \begin{bmatrix} 1 & -1 & -1 & -1 \\ 1 & 1 & -1 & 1 \\ 1 & 1 & 1 & -1 \\ 1 & -1 & 1 & 1 \end{bmatrix} \begin{bmatrix} T_{cmd} \\ \tau_{\phi_{cmd}} \\ \tau_{\theta_{cmd}} \\ \tau_{\psi_{cmd}} \end{bmatrix} \tag{4}$$

Application Insight: Understanding the Mixing Matrix

Let's break down what each column in the mixing matrix does:

- Column 1 (Throttle): All values are positive 1, so increasing throttle increases all motors equally
- Column 2 (Roll): Right motors get +1, left motors get -1, creating a rolling moment
- Column 3 (Pitch): Rear motors get +1, front motors get -1, creating a pitching moment
- Column 4 (Yaw): CCW motors get -1, CW motors get +1, creating a yawing moment

This matrix approach makes the code clean and efficient. In a real implementation, we would normalize these values to the available motor range.

5 PID Control Implementation

5.1 PID Controller Basics

For each control axis, we implement a PID controller:

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)d\tau + K_d \frac{d}{dt}e(t) \tag{5}$$

Where:

- $e(t)$ is the error (setpoint - measured value)
- K_p, K_i, K_d are the proportional, integral, and derivative gains

Beginner Note: Understanding PID Control

A PID controller works like this:

- **P** (Proportional): Pushes with a force proportional to how far you are from the target. The further off, the harder it pushes.
- **I** (Integral): Gradually increases force if you're not reaching the target over time. Like adding a little extra push each second you're still off target.
- **D** (Derivative): Applies a dampening force based on how quickly you're approaching the target. Helps prevent overshooting by slowing down the approach.

For a quadcopter, think of it like balancing a ball on a plate - P tries to center the ball, I gradually tilts more if the ball isn't centering, and D prevents the ball from rolling too quickly across the plate.

5.2 Code Implementation

Listing 1: Basic PID Controller Implementation

```
typedef struct {
    float Kp;           // Proportional gain
    float Ki;           // Integral gain
    float Kd;           // Derivative gain
    float previous_error;
    float integral;
    float dt;           // Sample time
    float output_limit;
} PIDController;

float updatePID(PIDController* pid, float setpoint, float
   measured_value) {
    float error = setpoint - measured_value;

    // Update integral term
    pid->integral += error * pid->dt;

    // Calculate derivative term
    float derivative = (error - pid->previous_error) / pid->
       dt;

    // Calculate PID output
    float output = pid->Kp * error +
```

```
                    pid->Ki * pid->integral +
                    pid->Kd * derivative;

    // Apply output limiting
    if (output > pid->output_limit)
        output = pid->output_limit;
    else if (output < -pid->output_limit)
        output = -pid->output_limit;

    // Store error for next iteration
    pid->previous_error = error;

    return output;
}
```

Application Insight: Optimizing PID Implementation

This PID implementation can be optimized in several ways:

- For microcontrollers with limited floating-point performance, consider fixed-point arithmetic
- The derivative calculation shown is susceptible to noise. Consider applying a low-pass filter to the derivative term
- For processors with deterministic timing, you could pre-calculate the gain-time products ($K_i \cdot dt$ and K_d/dt)
- Add a "derivative on measurement" option that only applies the derivative term to the measurement, not the setpoint changes

5.3 Control Loop Implementation

The main control loop for the quadcopter:

Listing 2: Main Control Loop

```
void controlLoop() {
    // Read sensor data
    readSensors();

    // Calculate current orientation
    updateAttitudeEstimate();

    // Run PID controllers for each axis
```

```
    float roll_output = updatePID(&roll_pid, roll_setpoint,
        current_roll);
    float pitch_output = updatePID(&pitch_pid, pitch_setpoint
        , current_pitch);
    float yaw_output = updatePID(&yaw_pid, yaw_setpoint,
        current_yaw);
    float altitude_output = updatePID(&alt_pid,
        altitude_setpoint, current_altitude);

    // Mix outputs to motor commands
    float motor1 = altitude_output - roll_output -
        pitch_output - yaw_output;
    float motor2 = altitude_output + roll_output -
        pitch_output + yaw_output;
    float motor3 = altitude_output + roll_output +
        pitch_output - yaw_output;
    float motor4 = altitude_output - roll_output +
        pitch_output + yaw_output;

    // Apply motor commands
    setMotorSpeeds(motor1, motor2, motor3, motor4);
}
```

Application Insight: Control Loop Timing

The control loop timing is critical for stable flight. For a typical quadcopter:

- The attitude control loop (roll, pitch, yaw) should run at least at 100-500 Hz
- The motor update rate should match the ESC update capability (typically 400-500 Hz)
- For optimal performance, use a real-time operating system (RTOS) or bare-metal programming with hardware timers to ensure consistent timing
- If possible, separate the sensor fusion (which can be CPU intensive) into its own task that feeds the faster control loop

6 Sensor Fusion

6.1 Complementary Filter

A simple complementary filter can combine gyroscope and accelerometer data for attitude estimation:

$$\begin{aligned} \phi_{est} &= \alpha(\phi_{prev} + \omega_x \Delta t) + (1 - \alpha)\phi_{acc} \\ \theta_{est} &= \alpha(\theta_{prev} + \omega_y \Delta t) + (1 - \alpha)\theta_{acc} \end{aligned} \tag{6}$$

Where:

- α is typically 0.95-0.98
- ω_x, ω_y are gyroscope readings
- ϕ_{acc}, θ_{acc} are accelerometer-derived angles

Listing 3: Complementary Filter Implementation

```
void updateAttitudeEstimate() {
    // Convert accelerometer readings to angles
    float accel_roll = atan2(accel_y, accel_z);
    float accel_pitch = atan2(-accel_x, sqrt(accel_y*accel_y
        + accel_z*accel_z));

    // Apply complementary filter
    float alpha = 0.96;  // Filter coefficient
    roll = alpha * (roll + gyro_x * dt) + (1-alpha) *
        accel_roll;
    pitch = alpha * (pitch + gyro_y * dt) + (1-alpha) *
        accel_pitch;

    // Simple yaw integration (no magnetometer correction)
    yaw += gyro_z * dt;
}
```

Beginner Note: Why Sensor Fusion?

Each sensor has strengths and weaknesses:

- **Gyroscopes** measure rotation speed very precisely in the short term, but drift over time
- **Accelerometers** measure gravity direction (which gives absolute orientation) but are noisy and affected by acceleration

Sensor fusion combines them: gyros provide smooth short-term data, while accelerometers correct long-term drift. It's like using a compass to occasionally correct your direction when following detailed turn-by-turn instructions.

Application Insight: Why α

The choice of $\alpha = 0.96$ represents a time constant of approximately: $\tau = \frac{dt}{1-\alpha}$
With a typical $dt = 0.01s$ (100Hz), this gives $\tau = 0.25s$, meaning the accelerometer influence takes about 1 second to reach 98% effect. This provides a good balance:

- High enough to reject short-term accelerometer noise
- Low enough to correct gyro drift within a reasonable time
- For racing drones, increase to 0.98-0.99
- For photography drones, decrease to 0.92-0.95

7 Practical PID Tuning

7.1 Step-by-Step Tuning Process

1. Start with all gains at zero
2. Increase K_p until the quadcopter responds quickly but may oscillate
3. Increase K_d to dampen oscillations
4. Add small amounts of K_i to eliminate steady-state errors
5. Test and refine

Application Insight: Practical Tuning Tips

When tuning your quadcopter:

- Secure the drone to a test rig that allows rotation but prevents it from flying away
- Start with roll/pitch tuning before moving to yaw and altitude
- Make small adjustments (10-20% at a time)
- Keep a log of each change and its effect
- Listen for high-frequency motor noise which indicates too much K_d
- Watch for slow oscillations which indicate too much K_p
- If possible, log flight data and analyze the step responses

7.2 Recommended Starting Values

Controller	K_p	K_i	K_d
Roll/Pitch	4.0	0.02	0.2
Yaw	2.0	0.01	0.1
Altitude	1.0	0.1	0.5

Application Insight: Why These PID Values?

These recommended values are based on a normalized control system where:

- Roll/Pitch: $K_p = 4.0$ gives a natural frequency of about 2 Hz, which is responsive yet stable. $K_d = 0.2$ provides critical damping, and $K_i = 0.02$ corrects for small imbalances.
- Yaw: Lower values because yaw control requires less precision and has higher inertia.
- Altitude: Higher integral gain because altitude control often fights constant forces (gravity and battery discharge), and position error is more tolerable.

These values assume angles in radians and a control loop running at 100-500 Hz. Scale proportionally if using degrees (multiply by $\pi/180$).

8 Practical Considerations

8.1 Anti-Windup

To prevent integral windup, limit the integrated error:

Listing 4: Anti-Windup Implementation

```
// Update integral with anti-windup
pid->integral += error * pid->dt;

// Limit integral term
if (pid->integral > pid->integral_limit)
    pid->integral = pid->integral_limit;
else if (pid->integral < -pid->integral_limit)
    pid->integral = -pid->integral_limit;
```

Beginner Note: What is Integral Windup?

Imagine pushing against a wall with a spring. If the wall doesn't move, you keep pushing harder and harder (the integral term growing). Then suddenly the wall disappears - you'd fly forward uncontrollably!
This happens in drones if motors can't provide enough power to reach the target angle (like in strong wind). When conditions improve, the accumulated integral term causes severe overshooting. Anti-windup prevents this by capping how much "pushing force" the integral can accumulate.

8.2 Motor Constraints

Always enforce minimum and maximum motor speeds:

Listing 5: Motor Constraints Implementation

```
// Apply motor constraints
motor1 = constrain(motor1, MIN_THROTTLE, MAX_THROTTLE);
motor2 = constrain(motor2, MIN_THROTTLE, MAX_THROTTLE);
motor3 = constrain(motor3, MIN_THROTTLE, MAX_THROTTLE);
motor4 = constrain(motor4, MIN_THROTTLE, MAX_THROTTLE);
```

Application Insight: Setting $\mathrm{MIN}_THROTTLE and MAX_THROTTLE$

Choosing appropriate motor constraints is critical:

- **MIN_THROTTLE** should be set just above the motor start-up threshold (typically 5-10% of maximum). This prevents motors from stopping mid-flight, which would cause loss of control.
- **MAX_THROTTLE** is usually set to 85-95% of the absolute maximum. This reserves some headroom for stabilization even at full throttle.
- For racing drones, MIN_THROTTLE can be higher (15-20%) to maintain responsive control during aggressive maneuvers.
- For photography drones, keeping a tighter operating range (15-85%) can improve flight smoothness.

8.3 Battery Voltage Compensation

As battery voltage decreases, motor power decreases. Compensate by adjusting throttle:

Listing 6: Battery Compensation

```
// Compensate for battery voltage drop
float compensation = NOMINAL_VOLTAGE / current_voltage;
motor1 *= compensation;
motor2 *= compensation;
motor3 *= compensation;
motor4 *= compensation;
```

Application Insight: Why Square Compensation is Better

The code above uses linear compensation, but motor thrust is actually proportional to voltage squared. A more accurate implementation would be:

```
float compensation = (NOMINAL_VOLTAGE * NOMINAL_VOLTAGE)
    /
                     (current_voltage * current_voltage);
```

However, this can overcompensate at very low battery levels. A practical approach is:

```
float compensation = pow(NOMINAL_VOLTAGE /
   current_voltage, 1.5);
// Limit maximum compensation
if(compensation > 1.5f) compensation = 1.5f;
```

This prevents excessive compensation when the battery is nearly depleted, which could mask the low battery condition.

9 Safety Features

9.1 Failsafe Mechanism

Implement various failsafe mechanisms:

Listing 7: Failsafe Implementation

```
void checkFailsafe() {
    // Check for radio signal loss
    if (lastRadioUpdateTime > RADIO_TIMEOUT) {
        // Enter failsafe mode
        setFailsafeMode();
    }
```

```
7
8       // Check for low battery
9       if (batteryVoltage < LOW_VOLTAGE_THRESHOLD) {
10          // Land immediately
11          setLandingMode();
12      }
13
14      // Check for excessive tilt
15      if (fabs(roll) > MAX_SAFE_ANGLE || fabs(pitch) >
            MAX_SAFE_ANGLE) {
16          // Disarm motors
17          disarmMotors();
18      }
19  }
```

Application Insight: Selecting Failsafe Parameters

When configuring failsafe parameters:

- **RADIO_TIMEOUT**: Typically 500ms-1s. Too short causes false triggers; too long delays emergency response.
- **LOW_VOLTAGE_THRESHOLD**: For a 3S LiPo (11.1V nominal), set around 10.5V for normal drones, 10.2V for racing drones.
- **MAX_SAFE_ANGLE**: For photography drones, set to 45-60 degrees. For racing drones, set higher (70-80 degrees) or disable for acrobatic flying.
- Consider progressive failsafe actions: warning LEDs → altitude reduction → return-to-home → emergency landing → disarm

10 Flight Testing

10.1 Pre-Flight Checklist

2. Check propellers and motors
3. Check battery voltage
4. Calibrate sensors on level ground
5. Start with low throttle and small movements

10.2 Testing Procedure

1. Test hover stability first
2. Gradually test each axis (roll, pitch, yaw)
3. Make small PID adjustments between flights
4. Record and analyze flight data for improvements

11 Advanced Topics

This section introduces more complex concepts for potential future development

11.1 Model Predictive Control

While PID control is effective for many applications, Model Predictive Control (MPC) offers advantages for trajectory following and constraint handling.

MPC works by solving an optimization problem at each time step:

$$\min_{u_{0:N-1}} \sum_{k=0}^{N-1} \left[(x_k - x_{ref})^T Q (x_k - x_{ref}) + u_k^T R u_k \right] \tag{7}$$

Subject to system dynamics and constraints on inputs and states.

Advanced Note: MPC Benefits and Challenges

MPC offers several advantages over PID:

- Explicit handling of constraints (like maximum tilt angles or motor limits)
- Optimizing multiple steps ahead for smoother trajectory following
- Ability to incorporate the full nonlinear dynamics model

However, MPC has challenges:

- Significantly higher computational requirements
- Need for accurate system models
- More complex tuning process

For example: in consumer drones, MPC is typically only implemented on companion computers (like Raspberry Pi) rather than directly on the flight controller.

11.2 Extended Kalman Filter

The complementary filter works well for basic attitude estimation, but the Extended Kalman Filter (EKF) provides improved state estimation for complex maneuvers and sensor fusion.

The EKF uses a predict-update cycle:

$$\begin{aligned} \mathbf{x}_{k|k-1} &= f(\mathbf{x}_{k-1|k-1}, \mathbf{u}_{k-1}) \\ \mathbf{P}_{k|k-1} &= \mathbf{F}_{k-1}\mathbf{P}_{k-1|k-1}\mathbf{F}_{k-1}^T + \mathbf{Q}_{k-1} \\ \mathbf{K}_k &= \mathbf{P}_{k|k-1}\mathbf{H}_k^T(\mathbf{H}_k\mathbf{P}_{k|k-1}\mathbf{H}_k^T + \mathbf{R}_k)^{-1} \\ \mathbf{x}_{k|k} &= \mathbf{x}_{k|k-1} + \mathbf{K}_k(\mathbf{z}_k - h(\mathbf{x}_{k|k-1})) \\ \mathbf{P}_{k|k} &= (\mathbf{I} - \mathbf{K}_k\mathbf{H}_k)\mathbf{P}_{k|k-1} \end{aligned} \tag{8}$$

Where $\mathbf{x}$ is the state vector, $\mathbf{P}$ is the covariance matrix, $\mathbf{K}$ is the Kalman gain, and $\mathbf{z}$ contains sensor measurements.

Advanced Note: EKF Implementation

When implementing an EKF:

- The state vector typically includes position, velocity, orientation, and sensor biases
- The process model (f) incorporates the full quadcopter dynamics
- The measurement model (h) maps states to expected sensor readings
- The Jacobians $\mathbf{F}$ and $\mathbf{H}$ are the linearized process and measurement models
- Tuning primarily involves adjusting the process noise ($\mathbf{Q}$) and measurement noise ($\mathbf{R}$) covariances

Many open-source flight stacks like PX4 and ArduPilot use EKF-based estimation and provide tuned implementations.

11.3 Trajectory Generation

For autonomous flight, smooth trajectory generation is essential. Minimum snap trajectories minimize the 4th derivative of position (snap):

$$\min_{p(t)} \int_{t_0}^{t_f} \left\| \frac{d^4 p(t)}{dt^4} \right\|^2 dt \tag{9}$$

This approach creates trajectories that align with quadcopter dynamics, since motor commands are proportional to the second derivative of attitude, which relates to the fourth derivative of position.

Application Insight: Polynomial Trajectory Implementation

For simple applications, 7th-order polynomials provide sufficient smoothness:

$$p(t) = \sum_{i=0}^{7} c_i t^i \tag{10}$$

The coefficients are determined by boundary conditions:

- Position, velocity, acceleration, and jerk at start and end points
- For waypoint navigation, ensure continuity of these derivatives across segments
- Pre-calculate trajectories to reduce computational load during flight
- Use time scaling to adjust speed while maintaining the trajectory shape

11.4 Disturbance Rejection

Wind and other external forces can significantly affect flight performance. Advanced controllers incorporate disturbance observers to estimate and compensate for these effects.

Advanced Note: Disturbance Observer Design

A basic disturbance observer estimates external forces using:

$$\hat{d} = m(\ddot{x}_{measured} - \ddot{x}_{expected}) \tag{11}$$

Where $\ddot{x}_{expected}$ is calculated from the control inputs and dynamic model, and $\ddot{x}_{measured}$ comes from IMU data.
This estimate can be filtered and fed back as a compensation term in the controller:

$$u = u_{nominal} - \hat{d} \tag{12}$$

For practical implementation, a low-pass filtered version is often used to reduce noise sensitivity.

11.5 Fault-Tolerant Control

Consumer drones typically lack redundancy in actuators and sensors. However, fault-tolerant control methods can maintain limited control even after component failures.

Application Insight: Motor Failure Recovery

If one motor fails on a quadcopter, it's still possible to maintain limited control by:

- Immediately increasing power to all remaining motors
- Switching to a modified control allocation matrix that excludes the failed motor
- Accepting the loss of one control dimension (full 3D control is no longer possible)
- Using rapid rotation or oscillation to create virtual control in the lost dimension
- Executing a controlled descent that minimizes crash impact

Research has shown quadcopters can maintain controlled flight with just three motors, though with reduced maneuverability.

11.6 Advanced Power Management

Battery life is a key constraint for quadcopters. Advanced energy-aware control can optimize flight time.

Advanced Note: Energy-Optimal Trajectories

Energy consumption in a quadcopter is approximately:

$$P = P_{hover} \cdot (1 + \frac{v_h^2}{v_{tip}^2} + \frac{v_v^2}{v_{ind}^2}) \tag{13}$$

Where P_{hover} is hover power, v_h and v_v are horizontal and vertical velocities, v_{tip} is propeller tip speed, and v_{ind} is induced velocity.
This model suggests that:

- Hovering consumes more power than slow forward flight
- Rapid vertical ascent is especially power-intensive
- Optimizing trajectories for energy can increase flight time by 10-20%
- Flying at the speed that minimizes power-per-distance maximizes range

12 Conclusion

This report covers the basics of quadcopter dynamics and practical control implementation.
The progression from basic PID control to advanced techniques represents the evolution of drone technology over the past decade. While just PID control massively popular due to its simplicity and robustness, the advanced approaches discussed in the final section offer pathways to improved performance for specialized applications.
For most applications, the simple PID approach with proper tuning and safety features will provide excellent performance.

The following references aren't all directly used in this report, but are generally the cited sources for the research I did on this report.

13 References

1. Bouabdallah, S., "Design and control of quadrotors with application to autonomous flying," PhD Thesis, EPFL, 2007.

2. Mellinger, D., and Kumar, V., "Minimum snap trajectory generation and control for quadrotors," IEEE International Conference on Robotics and Automation, 2011.

3. Mahony, R., Kumar, V., and Corke, P., "Multirotor Aerial Vehicles: Modeling, Estimation, and Control of Quadrotor," IEEE Robotics Automation Magazine, 2012.

4. Åström, K.J., and Hägglund, T., "PID Controllers: Theory, Design, and Tuning," 2nd Edition, 1995.

5. Lee, T., Leok, M., and McClamroch, N.H., "Geometric tracking control of a quadrotor UAV on SE(3)," 49th IEEE Conference on Decision and Control (CDC), 2010.

6. Mueller, M.W., and D'Andrea, R., "Stability and control of a quadrocopter despite the complete loss of one, two, or three propellers," IEEE International Conference on Robotics and Automation (ICRA), 2014.

7. Hoffmann, G.M., Huang, H., Waslander, S.L., and Tomlin, C.J., "Quadrotor helicopter flight dynamics and control: Theory and experiment," AIAA Guidance, Navigation and Control Conference and Exhibit, 2007.