AN25010 – Quadcopter Stability and Control

July 2025

Battlespace Simulations.
1229 Oak Valley Drive
Ann Arbor, Michigan 48108
Phone: 210-857-9672

If you have questions or comments, please contact us at support@bssim.com.

Overview

The MACE API exposes both aerodynamic and flight control characteristics for quadcopters that make it easy to tune their flight characteristics. This app note will show how to access the aerodynamic and control system interfaces for quadcopters.

Accessing the Aero Model and Control System

Quadcopter aero and control system properties are exposed via the `IControlSystemQuadcopter` interface, in turn exposed on the `IPhyscialEntityAero` interface as `IPhysicalEntityAero.ControlSystem`. The following code can be used to reliably test whether a physical entity is a quadcopter with the correct interfaces available for use

```
// entity is an instance of IPhysicalEntity. IPhysicalEntityAero is exposed
via the Aero property on
// IPhysicalEntity, but the instance can also be cast to IPhysicalEntityAero
to access

if(entity.Aero?.ControlSystem != null && entity.Aero.ControlSystem is
IControlSystemQuadcopter)
{
    // do something with the aero or control system
}
```

Quadcopter PID Control

PID controllers are fundamental to quadcopter stability and control because quadcopters are inherently unstable systems. Without constant, precise adjustments, they would immediately tumble out of the sky. A quadcopter's flight controller uses PID loops to maintain a desired orientation (roll, pitch, yaw) and altitude, translating pilot commands into motor speed adjustments. For instance, if you want the quadcopter to hover level, the PID controller ensures it stays precisely at a 0-degree roll and pitch angle. If you command it to tilt forward, the PID controller works to achieve and maintain that specific tilt angle.

At the heart of a PID controller is the "error" – the difference between the setpoint (the desired state, e.g., desired roll angle from the pilot's stick input or a pre-programmed hover) and the process variable (the actual state, measured by sensors like gyroscopes and accelerometers).

PID Gains

In the context of PID controllers, "gain" refers to a multiplication factor that determines the strength or aggressiveness of each of the proportional, integral, and derivative terms in influencing the controller's output.

Think of it like a volume knob for dialing in each part of the correction:

- Proportional Gain (Kp): This gain dictates how much the controller reacts to the current error. A higher Kp means a larger immediate corrective action for a given error. If your quadcopter is tilted 5 degrees, and you have a high Kp, the controller will quickly apply a strong force to bring it back to level. Too high, and it will overcorrect, leading to oscillations.

- Integral Gain (Ki): This gain determines how strongly the controller reacts to accumulated past errors. It's like how persistent the controller is in eliminating any lingering offset or drift. A higher Ki means the controller will work harder and faster to zero out any steady-state error. Too high, and it can cause sluggishness and overshooting as it continues to correct based on old errors even after the system is close to the setpoint.
- Derivative Gain (Kd): This gain influences how much the controller reacts to the rate of change of the error. It's a "damping" factor. A higher Kd means the controller will apply a stronger counter-force when the error is changing rapidly, helping to prevent overshoot and dampen oscillations. It smooths out the response. Too high, and it can make the system overly sensitive to noise in the sensor readings, leading to jerky movements or vibrations.

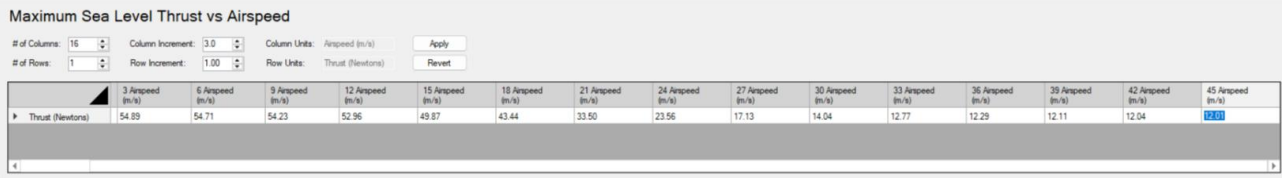
For each axis of rotation (roll, pitch, yaw), and often for altitude, there's a separate PID loop. The output of these PID controllers are signals that tell the individual motors how much faster or slower to spin. By carefully adjusting the speeds of the four propellers, the quadcopter generates the necessary forces and torques to stabilize itself and follow commands.

In essence, PID controllers are the "brains" of a quadcopter's flight controller, constantly monitoring its orientation and position, calculating errors, and making tiny, rapid adjustments to the motor speeds to keep it stable, responsive, and obedient to the pilot's commands or pre-programmed flight paths. As in the real world, MACE uses Proportional-Integral-Derivative (PID) control for both constructive and virtual flight models for quadcopters. The `IControlSystemQuadcopter` interface exposes the PID controller used for each control axis (pitch, roll, and yaw) for both autonomous flight (i.e. constructive flight model) as well as flyable flight models with both stability augmentation activated and deactivated. Each axis returns an instance of `IPIDController`, which in turn exposes Proportional (Kp), Integral (Ki), and Derivative (Kd) gains used for each axis (see Summary of interfaces).

Additionally, the body axis angular rates for roll, pitch, and yaw rates in radians per second for the quadcopter are also exposed via `IControlSystemQuadcopter`. The body axis rates are crucial for how the quadcopter feels to a pilot, particularly during FPV flight. When a pilot inputs a command to roll, they are essentially requesting a certain roll rate. The flight controller, through its PID loops, works to achieve and maintain that commanded rate. Tuning the PID gains directly affects how quickly and smoothly the quadcopter reaches and holds those desired rates. In essence, body axis rates provide the immediate, dynamic information about the quadcopter's rotational motion, allowing the PID controller to make rapid, proportional, and predictive adjustments to maintain stable and controlled flight.

Propellor Components and Dynamic Thrust Curves

All propellor components are exposed as a list collection on `IControlSystemQuadcopter`. Each propellor component implements the `IPropellorComponent` interface, which in turn exposes a look up table of thrust values as horizontal airspeed varies (i.e. a dynamic thrust curve) as well as the moment arm of each propellor. The thrust curves themselves are user modifiable in the Mission Object Configuration Tool (MOCT).

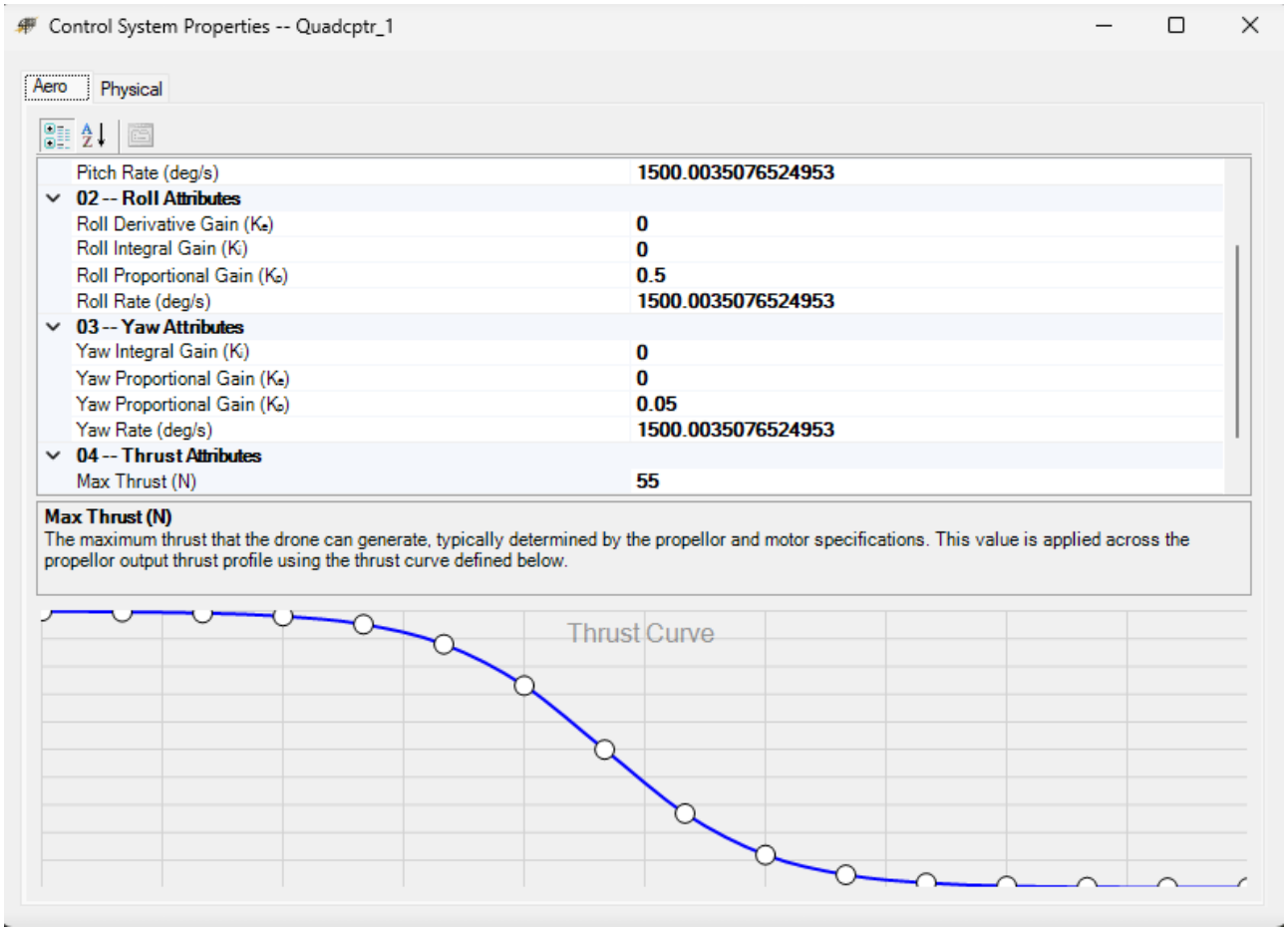


Dynamic thrust curves are important for quadcopter perfomance as they directly impact a quadcopter's maximum horizontal speed, its acceleration capabilities, and its power consumption during various flight maneuvers (e.g., fast forward flight, turns). Understanding how thrust changes with airspeed helps in optimizing flight paths for maximum endurance or range. While PID loops primarily manage attitude (roll, pitch, yaw) and altitude, the underlying thrust characteristics at different airspeeds are essential for the higher-level navigation and trajectory controllers that command the quadcopter's velocity.

In essence, a thrust curve at a given forward airspeed provides a more complete picture of a quadcopter's propulsion system performance, accounting for the complex aerodynamic interactions that occur when the vehicle is in motion, beyond just hovering or vertical ascent. They can greatly impact the handling characteristics of a quadcopter while under FPV control.

Using the CurveEditor UI component

If you're exposing the thrust curves in your plugin, as of MACE 2025R1 the MACE API provides a convenient curve editor component you can use in your user interface. It is in defined in the `SignalGeneratorUI`. You can point the toolbox in visual studio to the `SignalGeneratorUI.dll` to add the curve editor to the designer toolbox. See the documentation for the curve editor component in the MACE API documentation for more information.



Summary of Interfaces

Each interface discussed in this application note is shown in complete form below:

```
public interface IControlSystemQuadcopter : IControlSystem
{
    /// <summary>
    /// Roll axis PID controller while under virtual control
    /// with stability augmentation system on
    /// </summary>
    /// <returns></returns>
    IPIDController StabilityAugmentationOnRollPIDController { get; }

    /// <summary>
    /// Pitch axis PID controller while under virtual control
    /// with stability augmentation system on
    /// </summary>
    /// <returns></returns>
    IPIDController StabilityAugmentationOnPitchPIDController { get; }

    /// <summary>
    /// Yaw axis PID controller while under virtual control
    /// with stability augmentation system on
    /// </summary>
    /// <returns></returns>
    IPIDController StabilityAugmentationOnYawPIDController { get; }

    /// <summary>
    /// Roll axis PID controller while under virtual control
    /// with stability augmentation system off
    /// </summary>
    /// <returns></returns>
    IPIDController StabilityAugmentationOffRollPIDController { get; }

    /// <summary>
    /// Pitch axis PID controller while under virtual control
    /// with stability augmentation system off
    /// </summary>
    /// <returns></returns>
    IPIDController StabilityAugmentationOffPitchPIDController { get; }

    /// <summary>
    /// Yaw axis PID controller while under virtual control
    /// with stability augmentation system off
    /// </summary>
    /// <returns></returns>
    IPIDController StabilityAugmentationOffYawPIDController { get; }

    /// <summary>
    /// Roll axis PID controller when under autonomous control
    /// </summary>
    /// <returns></returns>
    IPIDController AutonomousRollPIDController { get; }

    /// <summary>
    /// Pitch axis PID controller when under autonomous control
    /// </summary>
```

```
        /// <returns></returns>
        IPIDController AutonomousPitchPIDController { get; }

        /// <summary>
        /// Yaw axis PID controller when under autonomous control
        /// </summary>
        /// <returns></returns>
        IPIDController AutonomousYawPIDController { get; }

        /// <summary>
        /// Returns true if stability augmentation is enabled
        /// </summary>
        /// <returns></returns>
        bool StabilityAugmentationEnabled { get; set; }

        /// <summary>
        /// A list of all propellor components on the quadcopter
        /// </summary>
        /// <returns></returns>
        IList<IPropellorComponent> PropellorComponents { get; }

        /// <summary>
        /// Roll rate in radians per second
        /// </summary>
        /// <returns></returns>
        double RollRate_radiansPerSecond { get; set; }

        /// <summary>
        /// Pitch rate in radians per second
        /// </summary>
        /// <returns></returns>
        double PitchRate_radiansPerSecond { get; set; }

        /// <summary>
        /// Yaw rate in radians per second
        /// </summary>
        /// <returns></returns>
        double YawRate_radiansPerSecond { get; set; }
    }

    /// <summary>
    /// PID controller Interface
    /// </summary>
    public interface IPIDController
    {
        /// <summary>
        /// Proportional Gain (Kp)
        /// </summary>
        double ProportionalGain { get; set; }

        /// <summary>
        /// Integral Gain (Ki)
        /// </summary>
```

```
        double IntegralGain { get; set; }

        /// <summary>
        /// Derivative Gain (Kd)
        /// </summary>
        double DerivativeGain { get; set; }
    }

    /// <summary>
    /// Propellor Component interface
    /// </summary>
    public interface IPropellorComponent
    {
        /// <summary>
        /// Max sea level thrust in newtons
        /// </summary>
        /// <returns></returns>
        double[] MaxSeaLevelThrust_newtons { get; set; }

        /// <summary>
        /// Body moment arm for the propellor component, in newton meters
        /// </summary>
        /// <returns></returns>
        Vector MomentBody_newtonMeters { get; set; }
    }
```