# Building A Controller Project

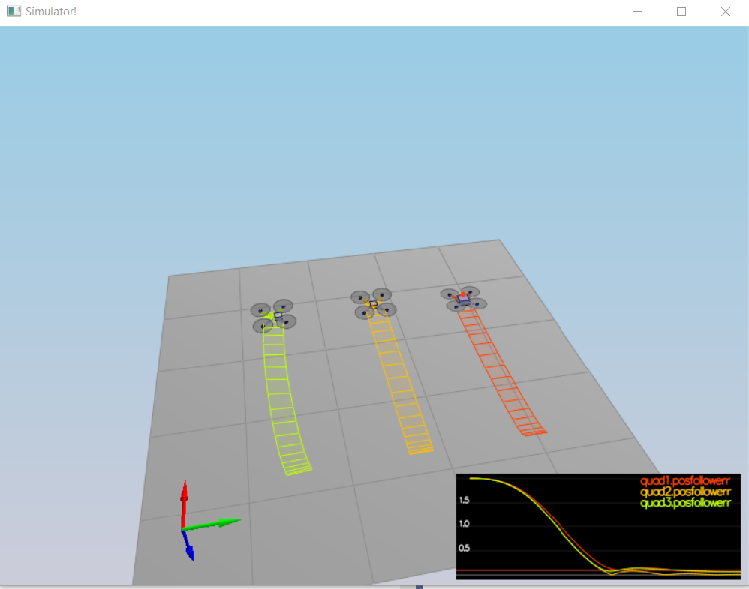
Dhar Rawal [dharrawal@gmail.com](mailto:dharrawal@gmail.com) 281-995-6423 11/22/2020

## Udacity – Flying Car and Autonomous Flight Engineer Program

## Introduction

The challenge here is to build a controller for a simulated drone such that the proper amounts of thrust are generated for each of the four propellers to fly a given path. This controller is implemented in C++ because of its speed. Control is time-sensitive given the drone dynamics and C++ provides much better repeatability/control of execution speed.

Image of the drones in the Control simulator



## Starter Code

The starter code consists of a simulator C++ project. The important files are – QuadControl.cpp/h, QuadDynamics.cpp/h and QuadControlParams.txt.

### QuadDynamics.cpp/h

These files implement a class called QuadDynamics that simulates the physics of the drone and how it would fly when different amounts of thrust are applied. Note here that the controller update is applied every 2 milliseconds via the ONBOARD\_TS parameter.

### QuadControl.cpp/h

This is where I worked to implement the control methods:

|  |  |
| --- | --- |
| **Methods** | **Description** |
| GenerateMotorCommands | Converts total thrust and moment to individual thrusts to be applied at each propeller |
| BodyRateControl | Innermost controller that controls p, q, r – angular velocities in drone coordinates. Converts commanded p, q, r into required angular accelerations/moments in drone coordinates |
| RollPitchControl | Converts x, y, z acceleration commands in world coordinates into the required pitch (p) and roll (q) angular velocities in drone coordinates. This is then fed to the BodyRateControl |
| AltitudeControl | Converts commanded z position and velocity into required acceleration/thrust in world coordinates. This is then fed into RollPitchControl |
| LateralPositionControl | Converts commanded z and y positions and velocities into required corresponding accelerations for feeding into RollPitchControl |
| YawControl | Converts commanded yaw angle in world coordinates into required angular velocity (r) in drone coordinates. This is then fed to the body rate controller. |
| RunControl | This method calls the individual control methods from outer to inner as follows:   1. AltitudeControl 2. LateralPositionControl 3. RollPitchControl 4. YawControl 5. BodyRateControl   The commanded thrust for altitude is constrained to leave some reserve thrust for controlling the x, y and yaw. |

### QuadControlParams.txt

This is where we can adjust the below control tuning parameters on the fly.

* kpPosXY
* kpPosZ
* KiPosZ
* kpVelXY
* kpVelZ
* kpBank
* kpYaw
* kpPQR

## BodyRateControl Implementation

This is straightforward.

V3F pqr\_error = pqrCmd - pqr;

V3F ubar\_pqr = kpPQR \* pqr\_error;

momentCmd.x = ubar\_pqr.x \* Ixx;

momentCmd.y = ubar\_pqr.y \* Iyy;

momentCmd.z = ubar\_pqr.z \* Izz;

## RollPitchControl Implementation

Note that we cannot reverse direction of propellers, so no negative thrust control is possible. Note that commanded thrust is upward and therefore acceleration (c\_cmd) is negative. Note also that x and y target angles in world coordinates are constrained by maxTiltAngle. Beyond this angle, thrust would be ineffective and the drone would become unstable.

pqrCmd[2] = 0.0f; // just to be 100% sure yaw rate is 0

if (collThrustCmd <= 0.0f)

{

pqrCmd[0] = 0.0f;

pqrCmd[1] = 0.0f;

}

else

{

float c\_cmd = -collThrustCmd / mass;

float b\_x\_a = R(0, 2);

float b\_y\_a = R(1, 2);

float b\_x\_c\_target = accelCmd.x / c\_cmd;

b\_x\_c\_target = CONSTRAIN(b\_x\_c\_target, -maxTiltAngle, maxTiltAngle);

float b\_y\_c\_target = accelCmd.y / c\_cmd;

b\_y\_c\_target = CONSTRAIN(b\_y\_c\_target, -maxTiltAngle, maxTiltAngle);

float b\_x\_c\_dot = kpBank \* (b\_x\_c\_target - b\_x\_a);

float b\_y\_c\_dot = kpBank \* (b\_y\_c\_target - b\_y\_a);

pqrCmd[0] = (R(1, 0) \* b\_x\_c\_dot - R(0, 0) \* b\_y\_c\_dot) / R(2, 2);

pqrCmd[1] = (R(1, 1) \* b\_x\_c\_dot - R(0, 1) \* b\_y\_c\_dot) / R(2, 2);

}

## AltitudeControl Implementation

Note here how the z velocity is contrained by maxAscentRate. Also, we use an integral controller to account for errors in drone mass estimate that would introduce a cumulative deviation in the z path. Note that downward acceleration is positive. Note how the drone thrust is constrained based on the min and max thrusts possible.

float errPosZ = posZCmd - posZ;

velZCmd += kpPosZ \* errPosZ;

velZCmd = CONSTRAIN(velZCmd, -maxAscentRate, maxDescentRate);

float errVelZ = velZCmd - velZ;

integratedAltitudeError += errPosZ \* dt;

float u1\_bar = KiPosZ\*integratedAltitudeError + kpVelZ \* errVelZ + accelZCmd;

float c\_cmd = (grav\_acc - u1\_bar) / R(2, 2); // grav\_acc is positive direction

thrust = mass \* c\_cmd; // multiply by mass since we are returning force.

thrust = CONSTRAIN(thrust, 4\*minMotorThrust, 4\*maxMotorThrust); // multiply by 4 since we are returning total thrust

## LateralPositionControl Implementation

Note here how the position and velocity gains are applied serially to constrain commanded speeds and accelerations.

V3F posErr = posCmd - pos;

velCmd += kpPosXY \* posErr;

velCmd.x = CONSTRAIN(velCmd.x, -maxSpeedXY, maxSpeedXY);

velCmd.y = CONSTRAIN(velCmd.y, -maxSpeedXY, maxSpeedXY);

V3F velErr = velCmd - vel;

accelCmd += kpVelXY \* velErr;

accelCmd.x= CONSTRAIN(accelCmd.x, -maxAccelXY, maxAccelXY);

accelCmd.y = CONSTRAIN(accelCmd.y, -maxAccelXY, maxAccelXY);

## YawControl Implementation

Note here how yaw is constrained between 0 and 2pi and the logic to take the shortest rotation path (clock-wise or counter-clock-wise) to get to the desired yaw.

yawCmd = fmodf(yawCmd, 2\*PI); // to ensure you are ranging from 0 to 2pi

yaw = fmodf(yaw, 2\*PI); // to ensure you are ranging from 0 to 2pi

float yawErr = yawCmd - yaw;

// we want to take the shortest rotation path to get there

if (yawErr > PI)

yawErr = yawErr - 2\*PI;

else if (yawErr < -PI)

yawErr = yawErr + 2\*PI;

yawRateCmd = kpYaw \* yawErr;

## GenerateMotorCommands Implementation

Note the calculation of distance from propeller to drone center along the world coordinate axis based on the distance L from the propeller to the drone.

Note that momentCmd.z is negative to account for a bug in the calculations. See <https://knowledge.udacity.com/questions/251050>

Note also how the rear right and rear left propellers are flipped from the course exercises in python.

float perpDist = L \* ONEOVERSQRT2;

// this assumes Kf = 1

float c\_bar = collThrustCmd; // total force

float p\_bar = momentCmd.x / perpDist; // twisting force in x

float q\_bar = momentCmd.y / perpDist; // twisting force in y

// why is below -momentCmd.z? See https://knowledge.udacity.com/questions/251050

float r\_bar = -momentCmd.z / kappa; // twisting force in z

cmd.desiredThrustsN[0] = 0.25f\*(c\_bar + p\_bar + q\_bar + r\_bar); // front left

cmd.desiredThrustsN[1] = 0.25f\*(c\_bar - p\_bar + q\_bar - r\_bar); // front right

cmd.desiredThrustsN[2] = 0.25f\*(c\_bar + p\_bar - q\_bar - r\_bar); // rear left (F3 and F4 are flipped from course)

cmd.desiredThrustsN[3] = 0.25f\*(c\_bar - p\_bar - q\_bar + r\_bar); // rear right

## Tuning

Once the controller code was in place, the real challenge began! Tuning is hard work! It took me 3 days to get it right.

TIPS:

* Start by setting all gains to zero.
* Increase gains by very small amounts slowly starting with the innermost controller.
* Stabilize each scenario in order. After you stabilize a scenario, go back and check the previous scenarios to make sure they have not regressed.
* Pay attention to the ratios of Kp/Kd and try to preserve these ratios as you change gains.
* Note that for the fourth scenarios, you have to focus on KpBank and KpPQR to conteract the effect of unbalanced mass for a drone. IMPLEMENTING A KiPosXY INTEGRAL GAIN DID NOT HELP HERE.
* Reread all the project documentation and knowledge base and review the tuning tips there.

### FINAL TUNING PARAMETERS

# Position control gains

kpPosXY = 2.4

kpPosZ = 3

# Integral control gains

KiPosZ = 24

# Velocity control gains

kpVelXY = 10.0

kpVelZ = 14.1

# Angle control gains

kpBank = 8

kpYaw = 1.7

# Angle rate gains

kpPQR = 80, 80, 5.1