

YAMEB

Daniel J. Gonzalez presents: Yet Another Mechanical Engineering Blog

Home	Projects	My Resume	
------	----------	-----------	--

27 May 2013

My First Quadrotor: Control Overview and Filtering Sensor Feedback

So I'm about to sit down and write the flight controller for my Quadrotor (My First Quadrotor? MFQ? MFQuad? MFCopter?), but rather than simply document that "I wrote the code and it worked", like I have been doing, I am going to try and be more explicit with my thought processes, in order to become more aware of my coding style, as well as to receive criticism in my style.

So, a quad flight controller. What's that?

In its most basic form, a Quadrotor Flight Controller (QFC for now, because Three Letter Acronyms (TLAs) are Really Freaking Cool (RFQ)) does the following:

- Reads data from the Gyroscope and Accelerometer
- Filters it into a usable form
- Generate commands the four motors based on the sensor values and user input in order to keep the quadrotor in a certain state, which is usually balancing pitch and roll to keep a stable hovering platform.
- Drives motors with these commands

Additionally, a QFC should be able to wirelessly receive commands from an external source in order to be able to move around. This can be as simple as 4 servo commands from a handheld receiver, or as complicated as X,Y,Z,Pitch,Roll,Yaw, and their derivatives. To keep the amount of data being transferred wirelessly down, however, I will keep this down to pitch, roll, yaw, and altitude.

Finally, for robustness, the QFC should be able to keep the quadrotor stable in the case of a broken connection mid-flight. Should bytes be dropped or a sudden change in communication occur, an onboard "Lost Connection" timeout will trigger, and the quadrotor will orient itself to a more stable orientation (The Pitch and Roll commands will reset to 0, or a landing sequence will initiate).

Okay.

Blog Archive

► 2014 (4)

▼ 2013 (15)

► Aug (2)

► Jul (3)

► Jun (3)

▼ May (3)

My First Quadrotor: Control Overview and Filtering...

My First Quadrotor: Parts Selection and Build

I'm Back!!! Here's What I've Been Up To:

► Jan (4)

► 2012 (31)

Categories

Arduino (4)

CNC (1)

code bits (1)

Cruscooter (6)

DeltaBot (2)

EtekChopper (9)

EVs (19)

MASLab (2)

MelonChopper (5)

MIT Classes (4)

New Toys (1)

potpourri (4)

Quadrotors (4)


Robotics (16)

TinyArm (5)

Turtlebot (2)

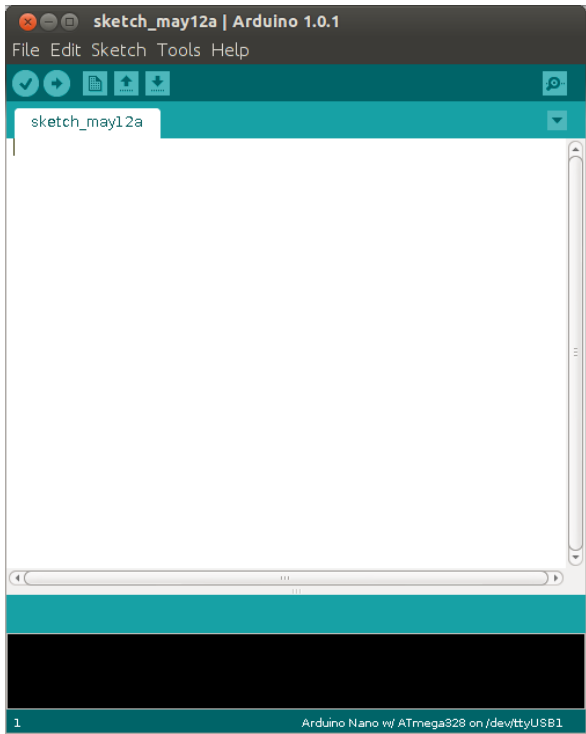
Work (4)

About Me



1 of 5

18/06/14 01:21



cracks knuckles
Let's do this.

So first of all our QFC needs a way to read sensor data. The gorgeous MultiWii board Shane was kind enough to give me is essentially an Arduino Nano 328 with a built-in 3-axis accelerometer, 3-axis gyroscope, magnetometer, barometer, and a bunch of servo-style breakout pins. The sensors all communicate with the Atmega 328 using the i2c protocol. While I have dealt with i2c communication at my job, every device is different, and requires slightly different requests/commands.

Shane gave me some example MultiWii code to be able to start talking with the sensors over i2c.

```
#include <Wire.h>

// I2C addresses for each sensor:
#define GYR_ADR 104
#define ACC_ADR 64
#define BMP085_ADDRESS 119
#define MAG_ADR 30

// sensor scaling
#define GYR_K 0.06956 // [(deg/s)/LSB]
#define ACC_K 0.01399 // [deg/LSB]
#define VEL_K 0.002395 // [(ft/s^2)/LSB]
#define ACC_Z_ZERO 4096 // [LSB]
#define RADTODEG 57.2957795
```

The above code contains the proper library inclusion (Wire.h) for i2c interfacing, the proper i2c addresses and conversion constants for the various MultiWii board sensors, a function to initialize each of these sensors, the i2c initialization code needed within the Arduino setup() function, and a function to read the raw sensor values from the Accelerometer and Gyroscope and convert them to usable forms (angle, acceleration, angle rate, etc).

Note: I eventually want to be able to read the Barometer data, so I found some excellent explanations and example code here (<http://www.sparkfun.com/tutorials/253>) as well as a more official source of barometric altimetry theory (which i used to great affect in 2.671). For now, we'll ignore the barometer data.

In addition to reading sensor data, we are going to need to filter it. The Second Order Complementary Filter that i'm used to requires a PI controller in order for it to work. Additionally, a PID controller will necessary for quadrotor self-balancing, so I opted to write a generic C++ PID class. For more info on PID Feedback Control,

Daniel J. Gonzalez
is studying Mechanical Engineering with a concentration in Controls, Instrumentation, and Robotics at MIT, due to graduate in 2014. This is a portfolio of my various class and personal projects.
[View my complete profile](#)

My Peers, My Idols

Banks Hunter
Bayley Wang
Ben Katz
Candace Chen
Charles Guan
David Wise
Eli Davis
Jacqueline Sly
Jeremy Bloyd-Peshkin
Jordan Jin Stone
Khalil Elbaggari
Nick Kirkby
Shane Colton
Victor Rodriguez
Willow Garage

just Google it. Wikipedia offers an excellent writeup, as well as an excellent textbook document hosted by Caltech.

Here's the generic PID Controller class I've written:

```
class PIDController{
  double kP;
  double kI;
  double kD;
  double xError;
  double xDesired;
  double xErrorPrev;
  double xErrorIntegral;
  boolean feed;
public:
  PIDController(double myKP, double myKI, double myKD, boolean isFeed){
    kP = myKP;
    kI = myKI;
    kD = myKD;
    xError = 0;
  }
};
```

When the class is instantiated, it requires the three PID constants and an option of whether the rate (derivative) is fed to the controller at each update, or if it should estimate it from past timesteps. This last option is useful if you have separate feedback for rate and position (like on this quad, which has both a filtered angle and a dedicated gyro for feedback). Before the controller is used, however, the set point for the control variable must be set via the `setDesired()` function. The `update()` function takes the current sensor feedback, as well as the (optional) derivative feedback, and spits out an actuator command.

This controller can command the motors of my quadrotor to counteract its roll or pitch angle deviating from the angle I want it at (which is usually 0), maintain a constant altitude with barometer feedback, or act as the backbone for the Second Order Complementary Filter in order to get proper pitch and roll angle feedback by combining both the Accelerometer and Gyroscope data, as I did in 2.12.

Here is the entire filter implemented in Arduino:

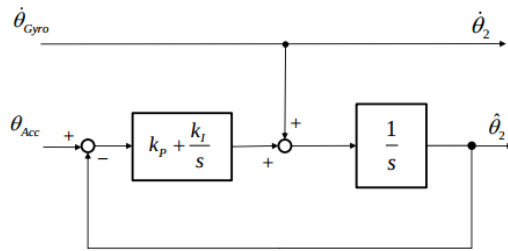
```
#define FREQ 200.0

double dt = 1.0/FREQ;
long loopStart;
double pitch = 0;
double roll = 0;
double initialPitch = 0;
double initialRoll = 0;
float wn = .045;
float filterKP = wn*2.0;
float filterKI = wn*wn*dt;

PIDController rollFilter(filterKP,filterKI,0, 0);
PIDController pitchFilter(filterKP,filterKI,0, 0);
```

Sweet! Now in my main loop I can call `IMUFilterUpdate()` immediately after updating the raw sensor values to get actual pitch and roll, provided I ensure the main loop runs at a frequency of 200 Hz. This can be done with a simple if statement that checks if the time elapsed since the previous run of the loop is greater than or equal to the period ($dt = 1/\text{Frequency} = 1/(200 \text{ Hz}) = 5\text{ms}$).

An excellent forum post about the Second Order Complementary Filter (SOCF?) can be found here (post #1286), and a writeup pdf by the author here. Also, is Shane Colton's incredibly useful whitepaper on the First-Order Complementary Filter (Not quite as robust, but super intuitive, easy to understand, and easy to implement).



The Second Order Complementary Filter (block diagram pictured above) works by first integrating the gyroscope data over time (or multiplying it by $1/s$ in Laplace space), resulting in an estimated angle. To do that, the the old angle estimate is added to the new change in angle:

```
angleNew = angleOld + (dThetadt * dt);
```

where dThetadt is the change in angle read directly from the gyroscope, in [degrees/s], and $dt = 0.005$ seconds.

However, the gyroscope drifts! Over a short amount of time, the angle estimate will have accumulated error, leading to an incorrect angle estimate (and causing the quadrotor to possibly become unstable). For short-term situations, especially when the helicopter is moving, the gyro is pretty accurate.

So this is corrected by taking advantage of the fact that the direction of gravitational acceleration can be found from the 3-axis accelerometer, regardless of the orientation, by taking the arctangent of the horizontal and vertical values of the accelerometer (Or linearizing the angle only using the horizontal component to save computation time, as I did above. The full atan version is commented out). This value is only useful if the quadrotor is not moving, however, and can be waaaaay off in situations where the vehicle is accelerating on its own accord.

So, we want the integrated gyro data when the vehicle is in motion, and the accelerometer angle estimate when the vehicle is static.

We can use both simultaneously, by providing the angleFilter PI controller the estimated angle as feedback, and the current Accelerometer estimated angle as the set point. The difference will generate a rate command to overcome the gyroscope drift, resulting in a more accurate angle that is robust against both gyro drift AND vehicle accelerations.

The values for the proportional and integrator constants, K_p and K_i respectively, can be determined if we model the PI controller as a second order system. The transfer function for the PI controller turns out in a fashion such that

$$K_p = 2 * \omega_n * \zeta;$$

and

$$K_i = \omega_n^2;$$

where ω_n is the natural frequency of the system and ζ is the damping ratio. Because we are designing this system (It isn't a physical system with its own uneditable system dynamics), we can set the damping ratio to be a perfect 1 (effectively eliminating the need for a ζ term) and the natural frequency ω_n to be the frequency at which we trust the gyroscope (or, alternatively, how often we correct the gyroscope with the Accelerometer estimate). I chose a natural frequency of 0.55Hz (period 1.81 seconds), though this may depend on your sensors, loop frequency, etc. I multiply the integral term by dt in order to take into account that our system is in discrete (digital) time.

UPDATE on ω_n : After testing some with this configuration and seeing delay between copter motion and , I've decided to trust the gyroscope even more and make ω_n 0.045, about an order of magnitude less. The resulting behavior leads to much faster response to copter motion.

A good test to see if your natural frequency is correct is to run the code and wait a while and see if the angle estimate drifts. Then, shake (accelerate) your vehicle to try and induce spikes in the angle estimate due to the accelerometer picking up the unwanted noise. If the angle remains stable for both tests, it works! :D

```
angleFilter.setDesired( AccelerometerEstimate );
angleNew = angleOld + (dThetadt + angleFilter.update(angleOld,0)) * dt;
```

The code all together, which prints out the filtered roll and pitch values of a multiWii328 board:

```
/*
Quadrotor Sensor and Filter Firmware
Daniel J. Gonzalez
dgonz@mit.edu
January 2013
*/

#include <Wire.h>
#include <math.h>

// I2C addresses for each sensor:
#define GYR_ADR 104
#define ACC_ADR 64
#define BMP085_ADDRESS 119
#define MAG_ADR 30


// sensor scaling
#define GYR_K 0.06956 // [(deg/s)/LSB]
#define ACC_K 0.01399 // [deg/LSB]
#define VEL_K 0.002395 // [(ft/s^2)/LSB]
#define ACC_Z_ZERO 4096 // [LSB]
#define RADTODEG 57.2957795

float gyro_pitch, gyro_roll, gyro_yaw; // [deg/s]
```

See you next post!

break;

at 2:52 AM

 Recommend this on Google

Labels: Arduino, Quadrotors, Robotics

No comments:

Post a Comment

Enter your comment...

Comment as:

Publish

Preview

[Newer Post](#)

[Home](#)

[Older Post](#)

Subscribe to: Post Comments (Atom)

Template images by Ollustrator. Powered by Blogger.