Semi-autonomous Quadcopter

Justin Syria

Fall 2017

Computer Science Senior Project

Table of Contents

Table of Contents:.....1
Introduction.....2
Languages Used.....2
Technologies Used.....3
Software.....3
Hardware.....4
Problems.....5
.....5
Overall Thoughts.....5
What I Would Have Done Differently.....6
Conclusion.....6

Introduction

The purpose of this project was to build and program a quadcopter capable of maintaining stability without fine input from a human. Other goals were mostly concerned with getting telemetry and other data to the operator, and allowing the operator to give the craft higher level instructions such as increase/decrease altitude, change heading and movement direction. In this paper I plan on outlining languages and technologies used, problems I ran into during development, what I would do different, and others.

Languages Used

The Arduino Due was programmed strictly in C++ as that is the only language[1] the 32-bit ARM core microcontroller accepts. Though much to my surprise, even if I could choose another language to program the Due in, I think I would stick with C++ due to a combination of it's fast runtime and capacity for supporting objects.

The Raspberry Pi, due to being a fully self-contained pc with an operating system can run programs written in any language. The flexibility of the device allows me to use what language I feel is best for the demands required of it. And for the Pi, python is going to be the best choice for me. It is fast to prototype and can do UI work with the help of libraries.

[1] : Arduino microcontrollers could actually be written in any language those compilers produce binary machine code for the target processor. However, the IDE only supports C/C++, which has the ability to upload code to the Arduino with minimal hassle, and verifies that it was uploaded correctly.
https://en.wikipedia.org/wiki/Arduino#Software_and_programming_tools

Technologies Used

This project required me to make use of technologies I had no experience with before, each being quite different from any other. For the sake of keeping this paper clean I will separate software and hardware

Software

PID Controllers are by far one of the most critical systems in the project. Proportional-Integral-Derivative Controllers are a control system that takes in some input (in this case, either the pitch/roll/heading/altitude of the craft),three values for K_p, K_i, and K_d, and a target value. These 5 values produce one output. The formula for a PID controller is as follows:

[2]
$$u(t) = K_\mathrm{p} e(t) + K_\mathrm{i} \int_0^t e(t)\,dt + K_\mathrm{d} \frac{de(t)}{dt},$$

Proportional (K_p) is based off of how far away from the target the input is. K_p, as will be the case for Integral and Derivative, is a multiplier. K_p is intended to correct for most of the error in a system. K_p sadly does not fully eliminate error due to it's output being tied to the difference between target and current values.

Integral (K_i) accounts for past error, and integrates those errors over time. The purpose here is to correct for the small error that the Proportional leaves behind.

Derivative (K_d) estimates the future trend of the error via derivatives. K_d is primarily used to both respond to large changes in the system and dampen the system to reduce overshoot. However, K_d has issues with changes in the target, causing a phenomenon called "Derivative Kick" where when the target changes, the system responds violently, due to the instantaneous change in the error.

In a perfect world with no variables, gravity, mass or inertia, only P is necessary to create an ideal control system. In most cases however, I and D are needed to create a system that behaves properly. This results in P, PI, PID, PD, and I controllers as all being viable options depending on what the controller is being used for. In the case of my quadcopter, PI or PID are the best choices due to signal noise and the inaccuracies in the quadcopter's construction.

[2]: https://en.wikipedia.org/wiki/PID_controller

Hardware

Electronic Speed Controllers (ESC's) are the interface between the brushless motors and the Arduino, on top of providing power to the motor. There is one for each motor and when they receive power they must be getting a Pulse Width Modulated(PWM) signal from the Arduino at a particular frequency, in the case of my motors. 1000µs pulse every 20ms. Should the ESC not receive that signal (which in a flight controller for a RC aircraft represents idle throttle) it will not engage the motors and power will have to be cycled.

The Arduino Due is a microcontroller that acts as the nervous system of the quadcopter, acting in place of a flight controller on a modern RC aircraft. It's job is to collect, process, and transmit signals from the various sensors. It will also be communicating with the Raspberry Pi to send telemetry information to the user and receive commands. A point of constant concern for me with this project is that the Due uses 3.3v logic, compared to most other microcontrollers/devices using 5v logic. Though any problem I'm come across so far has been a non-issue.

The Raspberry Pi 3 will serve as the interface between the user, camera, and Arduino, passing along messages across usb. While it does have wifi capabilities, it will not make use of it at this time.

The Adafruit 10-DoF (Degrees of Freedom) Inertial Measurement Unit (IMU) is what enables the system to know it's orientation and altitude. It has an onboard gyroscope, accelerometer, barometer, compass, and thermometer. It communicates with the Arduino over an Inter-Integrated Circuit (I^2C) serial bus, while I^2C technology is new to me, libraries allowed for easy communication without having to learn how exactly it works.

Problems

The largest issue I faced during this project was with the IMU. They are very sensitive to noise, and especially so to vibrations from the motors. Vibrations from the motors would cause the IMU to report changes in pitch by up to 35 degrees, when the device was physically pitching up 2 degrees at most. This problem went unnoticed until I started manually trying to tune the PID controllers. Where for some reason I could not get the quadcopter stable in a controlled environment. A notable amount of

time was wasted trying to tune PID controllers with the IMU giving erratic reports at best. I implemented an averaging system where I collect IMU data every 2ms for 10ms then average that value to help reduce noise. On the hardware side of things I also got vibration dampening material to help reduce noise, though some still exists.

PID controllers must be tuned, and for my project this has been a major road block, as without any sort of simulation software or precise telemetry I have to resort to manually tuning the controllers myself, which is time consuming and inaccurate. Manually tuning is possible but is dependent on having experience tuning them, which I do not have. With a proper testing rig I could get decent approximations to work off of, or with a simulation I could use better tuning algorithms that would get me to a good tuning much faster.

Overall Thoughts

Overall I'm enjoying working on this project. It is one that I can continue working on well past my graduation and it is allowing me to use things I've learned throughout my degree as well as letting me apply good software development practices. However, I do regret choosing a project with extensive mechanical components, as my skills and learning were not directed in that area at all, which makes for stressful times when I have to deal with mechanical/electrical engineering issues.

Drawing up object diagrams on a whiteboard was probably the single greatest thing I did to help with designing software for this project. Before any Unit tests or code was written I knew what should have gone in each object and how they would interact. Which during development saved me a lot of time on top of helping keep the number of lines of code down.

With solid object designs and unit tests I felt very comfortable with code I was running on the Arduino and Raspberry Pi. Though I wasn't able to write tests for Arduino specific libraries due to time constraints and lack of understanding how to do it properly without wasting a bunch of time. The observer pattern was a massive help in getting information between objects. I felt better about those interactions knowing that every object that needed information from another was getting it the moment the host object updated itself.

What I Would Have Done Differently

If I started this project over from scratch, and for future projects similar to this one, there are a few things I wish I would have dealt with early on. For one, avoid using any hardware that has ‘unique’ aspects to it like the Due’s 3.3v logic. While not an issue for interfacing, there are many times where I was worried I might burnout the board if I wired it to devices that normally operate on 5v logic. Another is that I would use task management software such as Trello to help track tasks and their progress. Such a system would help me track how quickly I’m progressing on a project and keep me from diverging away from goals. Not setting up a git repo for this project also wasn’t the smartest idea. Being able to track changes would have been very useful, as well as knowing that there is a consistently updated offsite backup.

Conclusion

I knew going into this project that it wasn’t going to be as easy as I thought it was. But from a purely software point of view, ignoring hardware issues, I believe that most of my objectives were obtainable. That said, I should have made the grading rubric focus more on purely software development instead of setting objectives that required both software and hardware to not only function properly, but work together as well.