# Autonomous Collision Avoidance For Small Unmanned Aerial Vehicles

Minjie Zhu
University of California, Davis
mjzhu@ucdavis.edu

Samuel Cheung
University of California, Davis
shocheung@ucdavis.edu

## Abstract

The goal of this project was to program a flight controller firmware to achieve autonomous collision avoidance (ACA) for small unmanned aerial vehicles (UAVs). The UAV we adopted was a quadcopter frame from Aerotestra[1]. The firmware was developed from the existing open-source code from ardupilot community [2]. ACA was realized by utilization of a LIDAR sensor to measure the distance between quadcopter and the closest object in its nose direction.

## 1. Introduction

A multicopter is a mechanically simple aerial vehicle whose motion is controlled by speeding or slowing multiple downward thrusting motor/propeller units. Unlike traditional helicopter or fixed-wing aircraft, it does not need to vary the rotor blade pitch angle. This simplifies the design and control of the vehicle. It is often used in applications such as aero-photography, aerial mapping. In our project we use a 4 propeller multicopter frame, i.e. a quadcopter.

The existing firmware code developed by ardupilot community supports auto-piloting feature utilizing the 3DR uBlox GPS sensor [3]. The feature is achieved by running the quadcopter in auto-mode. In this flight mode, the quadcopter runs a mission by reading the pre-written GPS coordinates and commands itself to fly towards each waypoint sequentially. However, the existing firmware does not support any object detection and avoidance schemes. For instance, the quadcopter will crash into a tree if the tree stands between two waypoints. The firmware on the Pixhawk flight controller is modified so that it can detect the object blocking its way during a mission, avoid the object, and continue to its next waypoint, shown in figure 1. Implementation of this will be explained in this report.

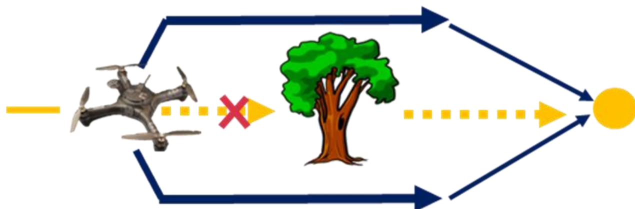

Figure 1 Idea of ACA

## 2. Hardware and Software

### 2.1 Hardware

**Quadcopter frame** We use Aerotestra quadcopter model *Hugo* for our project. The model includes the Foxtech 4016/380 Kv Brushless Motor, 15' * 5.5 carbon fiber propeller and 915 Mhz telemetry radio link. [1]

**Flight controller** The 3DR Pixhawk is used for *Hugo* as the flight controller. The flight controller is installed with our custom firmware based on ArduCopter V3.3 from the Ardupilot community. The firmware base has the essentials for the quadcopter to fly, but has been modified to achieve our goals. [4]

**Essential Flight sensors** The position of the quadcopter is obtained by the longitude and latitude from the GPS module and the altitude is obtained from the barometer. The orientation and rotation are known by the angular velocity and acceleration from the three axis gyroscope and accelerometer. The electrical compass is also needed to determine the heading. All essential sensors except GPS are onboard.

**LIDAR-Lite** The LIDAR-Lite laser-based sensor is used to do object detection and avoidance. The communication protocol it uses is I2C. It has a very narrow beam angle of 1.5 degrees and can detect objects from 0.2 meters up to 40 meters. The sensor has a resolution of 1cm.

The LIDAR-Lite can be used for various applications depending on the orientation of the sensor. If the sensor is pointing down to the ground, it can be used for altitude. In our case, the LIDAR-Lite sensor is mounted on the front of the quadcopter to detect objects in the forward direction.

In order to power the sensor, we have attached 4 AA batteries in a battery pack to supply the required 4.7 - 5.5 Volts.

### 2.2 Software

**User interface** Mission Planner is adopted to act as a ground station for control. Flight status can be monitored on the mission planner interface. [5]

**Code Modification software** Eclipse is the compiler we use to modify and build the firmware.

### 2.3 Firmware Structure

The original Firmware developed by ardupilot is a well-structured code with hierarchy. We fully understood and inherited its structure during the project. The hierarchy of the firmware structure is listed below, from highest level to lowest level:

**Scheduler** The scheduler acts as the main function that runs repeatedly. It is also a real time operating system that assigns each thread with a certain call frequency, priority and maximum run time.

**Flight mode controller** The flight mode controller determines the feature of quadcopter movement. At any time, the board must be inside one of the flight modes to ensure system integrity. Depending on the flight mode, values from remote controller channels such as pitch/roll/yaw may be applied or ignored. The

flight modes can be changed via channel 5 of the remote controller. Additional flight modes can be created for a more isolated environment and more control over the quadcopter for different applications.

**Movement controller** The movement controller includes four controllers dealing with altitude, attitude, position, and waypoints. The altitude controller uses the barometer to control the z-axis acceleration. It tries to maintain the altitude in altitude hold condition and also adjusts altitude if ascending/descending is required. The attitude controller controls self-rotation and balances the quadcopter. This is useful for balancing against wind and motor output differences. The position controller controls horizontal x-y axis acceleration. Lastly, the waypoint controller works with the GPS to determine the corresponding direction and speed in auto mode according to the preset waypoints.

**Library files** The library files interact with all the lower level communication with sensors, data processing and computations, as well as motor outputs. Examples of library files include: GPS, LIDAR, PID control, mavlink, filter, etc.

**HAL layer** The HAL (hardware abstraction layer) deals with the board I/O itself. It sets the value for digital/analog inputs and outputs.

# 3. Method

For completing our object detection goal, we implemented fixes to the LIDAR-Lite sensor, created an AI object, created various states, and made an avoidance scheme.

## 3.1 LIDAR stagnancy

The LIDAR-Lite sensor has a problem dealing with objects that are out of its range (0.2m to 40m). When the LIDAR is suddenly pointing at something out of its range, it will keep the previous value and will not update the latest value. For example, if the LIDAR is pointing at an object 5 meters away and is suddenly moved to point to something 45 meters away, the reading returned will be 5 meters. When the sensor doesn't update, we consider the LIDAR reading to be stagnant. To solve this problem, we implemented a function that filters the LIDAR sensor reading. When the previous reading is greater than a threshold (we set it to 1 meter) and it stagnates for more than one second, we assume that the sensor is pointing to something beyond 40 meters. We arbitrarily interpret the reading as infinite. Conversely, when the previous reading is less than 1 meter and it stagnates for more than one second, we assume that the sensor is pointing to something less than 0.2 meters. With the filter, we will always have readings that correspond to what the LIDAR sensor detects.

Figure 2 shows the function for filtering LIDAR stagnancy

```
#define ACA_LIDAR_STAG_THR 100      //lidar stagnates within THR, considered as blocked,
                                    // else considered infinite
#define ACA_LIDAR_STAG_BLOCKED 1            //return of lidar when in BLOCKED state
#define ACA_LIDAR_STAG_INIFINITE 9999       //return of lidar when in INIFINITE state
void
ACA_AI::update_lidar(int sonar_distance_cm)
{       lidar_curr = sonar_distance_cm;
                if (lidar_curr == lidar_prev){lidar_stag_ct++;}
                else{lidar_stag_ct = 0;}
        //Lidar return value keeps the same for 1 sec, considered stagnant
                if(lidar_stag_ct > ACA_LIDAR_STAG_CT)
                {
                        if(lidar_curr >ACA_LIDAR_STAG_THR)
                        {lidar_cm = ACA_LIDAR_STAG_INIFINITE;}
                        else
                        {lidar_cm = ACA_LIDAR_STAG_BLOCKED;}
                }
                else//Lidar return value not considered stagnant
                {lidar_cm = lidar_curr;}
        lidar_prev = lidar_curr;
}
```

Figure 2 update_lidar function

## 3.2 ACA AI object

To implement the ACA feature, we create a new C++ class called ACA_AI. All modifications to the original source code regarding ACA is implemented and processed in this object. To minimize interference with the other running schedules and keep the code structure clear, the member functions in ACA object run independently. ACA references two global variables (`sonar_distance_cm` as distance measured from LIDAR sensor, and `milliseconds` as system clock from flight controller), generates corresponding RC channel values, and updates the state iteratively. The values generated and updated are applied by the flight mode controller in auto mode accordingly. As a whole, the ACA AI object acts as an internal artificial intelligence (AI) that replaces the remote control to send appropriate commands to the quadcopter.

Figure 3 shows the file linkage relationship between ACA AI and some other essential files/objects

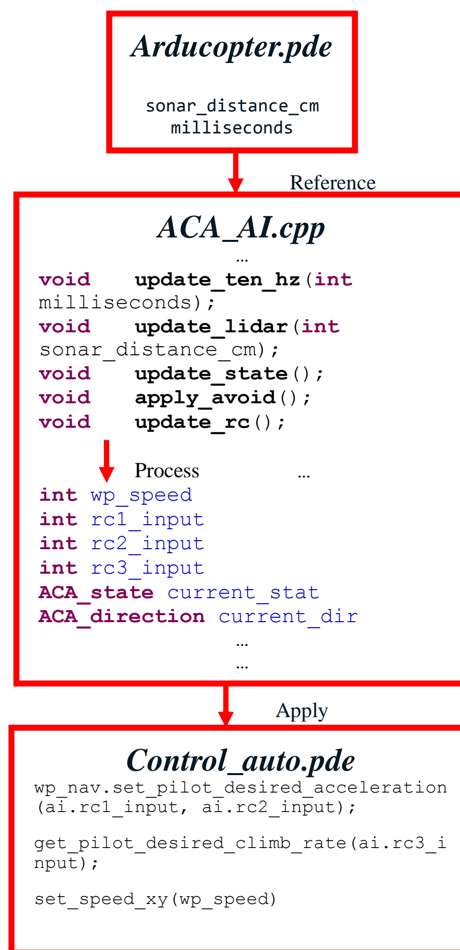


Figure 3 ACA AI object

### 3.3 State transition

ACA AI serves as a company with the normal auto mode of the quadcopter. In normal auto mode, the quadcopter will run a flight mission according to the pre-set way points. When the ACA AI is activated, it will transits among four states according to the LIDAR distance. The four states are Sleep, Slow, Halt and Avoid. ACA AI enters Sleep state and keeps the way point speed high when no objects is nearby. It switches to Slow state and reduces the way point speed to be ready for object incidence when object is sensed at a long distance. It goes into Halt state and cancels movement in auto mode when object is sensed at a short distance. After reassuring the object incidence is true, it activates Avoid state to apply an avoidance scheme.

Figure 4 illustrates the state transition diagram of ACA AI

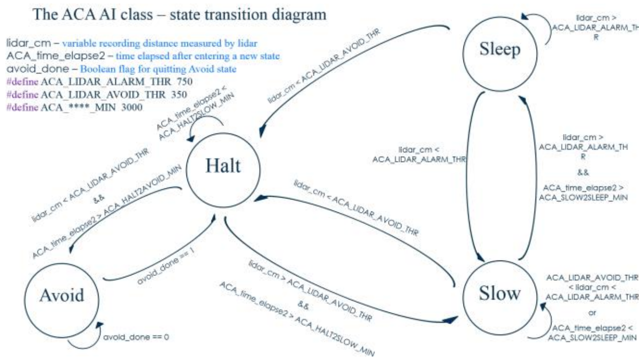


Figure 4 ACA AI state transition diagram

### 3.4 Avoidance scheme

After true object incidence is verified, ACA AI generates corresponding RC channel value to command the quadcopter in order to bypass the object. ACA AI starts with a leftward movement, followed by a rightward movement and ended with an upward movement. During the attempts, whenever the LIDAR distance no longer returns a short distance for a period of time (we set it to 3 seconds), meaning the object no longer stands in front of the quadcopter, ACA AI attempts a forward movement to complete the avoidance scheme. If the avoidance scheme is applied successfully, ACA AI goes back to Halt state and is ready to resume the previous mission. Otherwise, it keeps making avoid attempts.

Figure 5 illustrates the timing diagram of the avoid scheme in ACA AI

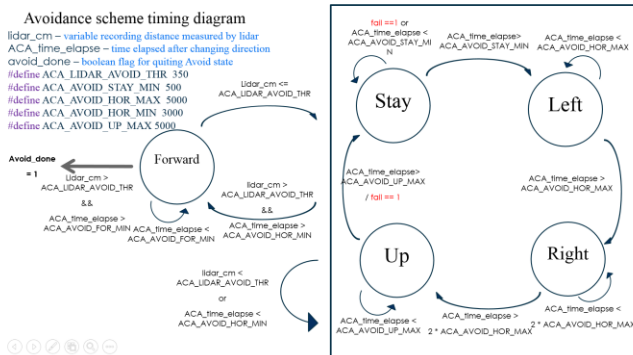


Figure 5 Avoidance scheme timing diagram

## 4. Results and Evaluations

The firmware we created is able to accomplish the autonomous collision avoidance feature as expected. However, the object is constrained to a flat surface to ensure successful avoidance. We have made two successful experiments that demonstrate the ability of our state transitions and avoidance scheme.

In the first experiment, the quadcopter was assigned to run a mission on a large field. We had a person carrying a piece of cardboard that simulates a large building. The quadcopter attempted to avoid the object by going left, right, and finally up. After the three attempts, it detected that there is no object and continues to its next waypoint. Log map is shown in Figure 6.

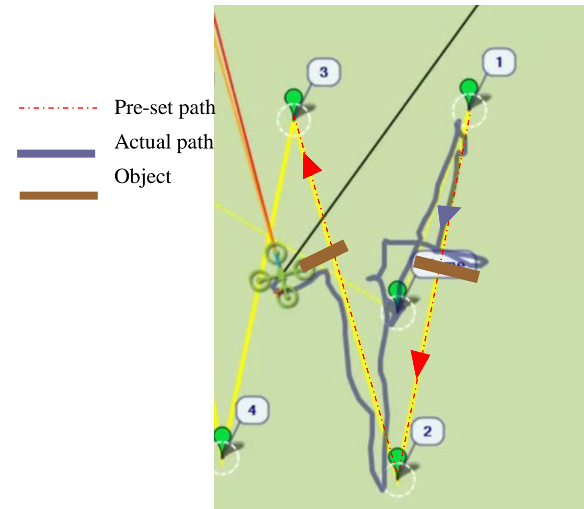


Figure 6 Flight log of experiment on cardboard

In the second experiment, the quadcopter was assigned to run a mission with a tree in the way. During the mission the quadcopter automatically went around the tree that blocked its way. Log map is shown in Figure 7.

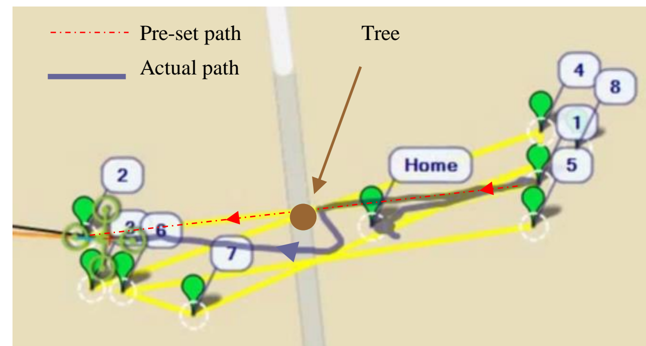


Figure 7 Flight log of experiment on Tree

The two experiments were recorded with live video records and screen capture of Mission Planner log. Links to the videos are given:

| experiment | Live video | MP log |
| --- | --- | --- |
| Card board | youtu.be/by2NV3cgJnc | youtu.be/sbxltKb1jWo |
| Tree | youtu.be/Y__QCXf_xPI | youtu.be/IeLikO9GvX0 |

## 5. Additional modifications

In addition to the ACA AI object we created, we also managed to explore some other features on the quadcopter firmware. The other two major modifications we made were creating a new flight mode and enabling a force landing RC command.

### 5.1 New flight mode – test mode

The test mode is for testing flight control algorithms and methods in addition to the existing 16 flight modes, without interfering other lower level functions. The algorithm we put in the test mode is an image guided control algorithm. An FPGA board process the image and sends corresponding commands to the quadcopter via general purpose digital pins. [6]

### 5.2 Force landing RC command

The force landing RC command serves as an emergency disarm switch for the motors. In the original firmware, the quadcopter relies on the altitude calculated from barometer readings to disarm the motors. In case of faulty barometer readings, the quadcopter motors cannot be disarmed properly. With the additional force landing RC command, quadcopter can disarm the motors whenever the user sends the force landing command.


## 6. ACKNOWLEDGMENTS

Our thanks to Professor Xiaoguang Liu, the proposer and supervisor for our project. Our sincere appreciation to Sean Headrick, who donates his retired Hugo quadcopters to us.


More information can be found at our project website:

minjiezhu.wordpress.com/quadcopter/

## REFERENCES


[1] Aerotestra Hugo quadcopter, www.aerotestra.com

[2] Ardupilot community www.ardupilot.com

[3] 3D robotics 3drobotics.com

[4] Pixhawk pixhawk.org

[5] Mission Planner software http://planner.ardupilot.com/

[6] UC Davis ECE 181 Senior Design group on FPGA image processing