Using a sonar ranging system to enable a quadrotor helicopter to follow a wall

Geoffrey Litt
Class of 2014
B.S. Candidate, EECS

December 21, 2012

EENG 471 - Independent Project
Group project with Caroline Jaffe '13
Prof. Roman Kuc
Yale University

Abstract

In this project, a sonar ranging sensor attached to a servo was mounted on a Parrot AR.Drone 2.0 quad-rotor helicopter, allowing the quadcopter to detect its distance from a wall and move back and forwards while maintaining a relatively constant distance from the wall. The final system was capable of tracking a wall within a tolerance band of approximately 1 meter, although the inherent instability of the flying system made it difficult to achieve completely reliable operation.

1 Introduction

1.1 Overview

The Parrot AR.Drone 2.0 is a technologically sophisticated quad-rotor helicopter (a.k.a. "quadcopter" or "drone"), with advanced capabilities like an internal gyroscopic stabilization system, altitude measurement using sonar, and two onboard cameras. This range of capabilities makes it potentially useful for a wide range of activities such as remote camera monitoring, transportation of light payloads, etc. However, one major limitation of the system is that it lacks an obstacle avoidance system. If movement commands are sent to the drone without a feedback loop, it could crash into obstacles at any time, defeating the usefulness of any autonomous operation. Therefore, a sensing system for the drone that detects objects in the environment and automatically applies corrections to movement controls could enable a whole host of other applications.

Figure 1: The completed drone system

We created a system to achieve a basic demonstration of this goal, by following a wall at a constant distance. The finished system is shown in Figure 1. we mounted a sonar sensor on a servo, and attached this sensing system to the drone. The system uses an Arduino Pro microcontroller to move the servo back and forth and drive the sonar at regular intervals, relaying back the sonar readings wirelessly to a computer. The computer processes the sonar echo time readings to determine the distance and orientation of the drone relative to the wall, and it issues appropriate movement commands to the drone so that the drone moves back and forth within several meters of the wall, without actually colliding with it. As discussed in more detail later, this capability could serve as a useful part of a more full-featured autonomous movement system.

1.2 Specifications

The main design goal for the project was to keep the drone at a fixed distance from the wall (within a certain tolerance band, as narrow as possible), so that it would neither collide with the wall nor move too far away. In the course of completing the project, we determined that a reasonably feasible goal was to keep the drone within 0 to 2 meters of the wall.

Achieving this goal required satisfying a number of prerequisite goals. First of all, the drone obviously had to be able to fly successfully. This proved to be a difficult specification to meet, since the drone's maximum payload weight was relatively small. Battery-powered operation was also a desired goal for the system so it could be used in any location, but this goal turned out to conflict with the weight reduction goal, and the final system was powered through a cable by a power supply.

It was also necessary to build a fast and accurate sensing system to enable the drone to follow the wall successfully. In testing our servo rotation code, we experienced a tradeoff between the speed of moving the servo and the accuracy with which the servo would move to a given angle – as we reduced the delay between servo positions, we observed increased discrepancies between readings taken as the servo was moving in one direction or the other. (We were unable to determine exactly the cause of this effect; it may have been related to the mechanical break frequency of the servo, or the way we attached the sonar to the servo.) We determined that 750ms was an appropriate length of time for a single scan in one direction – we would also send the drone a movement command after each of these single scans.

As for the accuracy of the sonar measurements, the sensors that we used were very precise, and we did not even require the full precision of the sensors for our project (upon receiving echo delay time readings in microseconds, we divided by 100 to discard the last two significant digits before processing, in order to make it easier for us to conceptualize readings without worrying about unimportant noise). However, although individual readings from the sensors were precise, we did have to consider noise that could arise from various errors in the servo's movement, or small features in the environment that could cause misleading sonar readings. It was necessary to smooth out this noise in software to some extent, to achieve stable operation.

Finally, reliable wireless communication with the drone was essential for successful operation of the system. Any dropping out of the wireless links used to send sensing data or control commands could lead to dangerous conditions, since it would prevent the drone from effectively avoiding the wall.

2 Methods

2.1 Equipment

- Parrot AR.Drone 2.0 quadcopter

- Dell Inspiron laptop running Ubuntu Linux
- Arduino Pro microcontroller
- micro servo, 150-degree range
- Senscomp 6500 ultrasonic ranging module
- Senscomp Instrument Grade transducer
- 2 x XBee Series 1 wireless communication module
- 2 x DC power supply
- USB-FTDI cable (for connecting to XBees, Arduino)

2.2 Design Philosophy

To preface a discussion of the detailed techniques used to design the system, there were a range of high-level design considerations we took into account when creating the system.

Originally, the plan for the project was to follow objects, processing images received from the cameras built into the drone. We decided to switch to the sonar sensing project we ended up completing for several reasons. First, it had proved technically difficult to access the video stream from the drone. In addition, we thought that the sonar approach could prove more extensible to a variety of applications related to autonomous movement and obstacle avoidance. Finally, adding our own sensing hardware to the drone presented a variety of interesting challenges from an electrical and mechnical engineering standpoint.

As a result, we generally made attempts to design the system in such a way that it could be easily extended to other applications. For example, we could have used a statically mounted vergence sensor, but we instead chose to mount a sonar sensor on a rotating servo to sweep over a wide angle and provide multiple scan data points. This added complexity to the system, but could enable the same hardware to be used for other applications with changes to the control software.

Another high level design consideration was designing for rapid iteration and failure. Because of the instability of the drone, small hardware changes could have large effects on the system's performance. We therefore avoided permanent hardware decisions as much as possible, allowing room for on-the-fly modifications to improve stability. This need was aggravated by the fact that the drone would often crash during testing, causing hardware elements to be dislodged. If we had had more time, it could have proved beneficial to design a robust enclosure and finalize a permanent hardware setup; however, we were changing components frequently enough that this was not feasible with our time constraints.

We took an empirically driven approach to the design process. We attempted to test components of the system independently and calibrate them based on

real observation, so that we would not end up building an entire system that worked hypothetically but not in practice.

2.3 Architecture

It may be helpful to provide an overview of the high-level architecture of the system before describing details. The system consists of two main sections – the custom hardware mounted on the drone, and a Linux laptop running software.

The custom hardware on the drone is controlled by an Arduino microcontroller, and activates the sonar sensor at various angles. It then receives echo time readings at various angles, and prints these readings in a standardized protocol over a serial connection to an XBee wireless device. The drone-mounted hardware is only there for sensing, and does not directly control the drone in any way.

The Linux laptop is connected to a WiFi network broadcasted by the AR.Drone, which enables it to send movement commands to the drone using an API provided by Parrot. It is also connected to an XBee wireless device with a USB FTDI cable, and receives sensor readings over this connection. The laptop runs a program written by us which takes in the sensor readings, computes the appropriate corresponding movement commands, and hooks into the AR.Drone API to send these commands.

2.4 Sensing Hardware

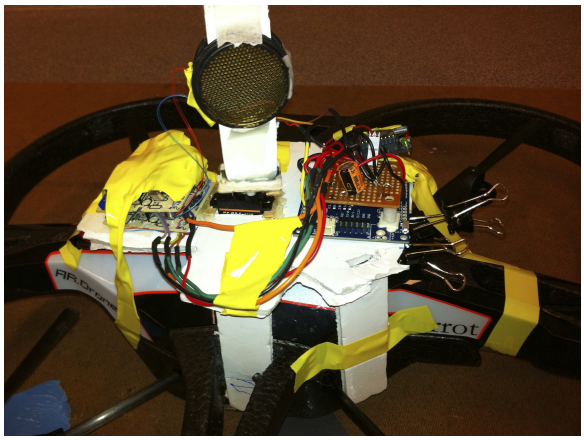


Figure 2: The sensing hardware

The sensing hardware setup we created is shown in Figure 2. The main components are the Arduino Pro microcontroller (at the right in Figure 2, two ultrasonic ranging modules (visible at the left of the image – one is a relic of an

older setup with two sonars, and is not used), a micro servo (in the center of the image, embedded in the white foamcore board), a sonar transducer mounted on a column on top of the servo, and an auxiliary board connecting all the components (mounted on top of the Arduino board on the right). The auxiliary board is also connected to the drone's 5 volt power cable.

Schematic

The electrical connections between the sensing components can be seen in the schematic, in Appendix A. As a summary: the auxiliary board distributes 5 volt power from the power cable to the Arduino, sonar module, servo, and XBee module. The 5 volt source also provides a pullup voltage source for the echo line from the sonar module. The Arduino has a digital output pin connected to the sonar module INIT, a digital input pin connected to the sonar module ECHO (with a pullup resistor), a digital output pin connected to the servo control pin, and serial Tx/Rx to the XBee module.

Hardware platform

All of the sensing components are mounted onto a foamcore board, which is attached to the drone with four vertical rectangular supports also made of foamcore, glued onto the side of the drone. Foamcore was chosen as the material for the supporting platform because it is extremely lightweight and weight was a major concern. The column supporting the sonar is made of foamcore and attached to the servo with a small balsa wood platform and glue; the servo is then mounted in a hole in the foamcore board and glued on. The other components were all attached to the board with tape and glue.

Arduino

The Arduino microcontroller used to control the sensing hardware is an Arduino Pro 328, which is a 3.3V, 8Mhz microcontroller based on the ATmega328 chip. We originally started out using an Arduino Uno, which is a more commonly used 5V board, but it was too heavy. The Arduino Pro is a significantly lighter board based on surface mount components, with no onboard USB connector or headers on pins.

The code executed on the Arduino can be seen in Appendix B. It is an extremely simple program. All the main program loop does is rotate the servo back and forth by a set increment (approx. 15 degrees), and send out a sonar pulse at each servo position. To send a sonar pulse, the sonar INIT line is set high for 20ms and low for 30ms. As a result, the servo was at each position in its rotation for 50ms, resulting in a sweep lasting approximately 750ms. Also, the time at which the sonar is initiated is recorded so that the time to echo can be calculated later.

To report sonar readings, an external interrupt was used. A handler function "handleEcho" was associated with a rising external interrupt on the sonar echo

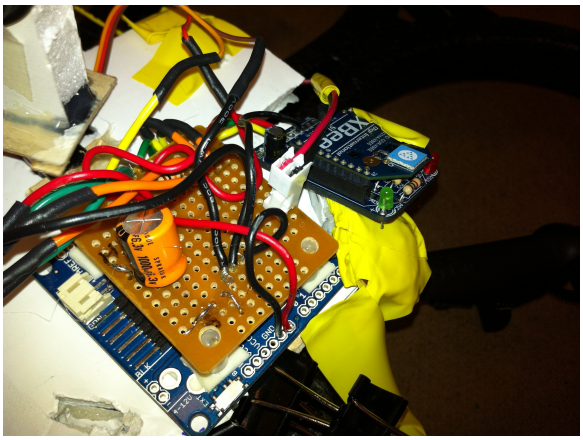


Figure 3: Arduino, auxiliary board and XBee module

pin, so that it would be triggered every time an echo was received. The handler function essentially records the time difference between the sonar init and echo, and prints this time in microseconds, along with the current servo angle, to the serial connection it has with the XBee (in the format "angle:time"). However, there is a slight degree of added complexity in the program which was added to prevent errors – if multiple different echo times are reported for a given servo angle, only the last one will be reported. This error correction was added to deal with an error where a very fast echo would be reported before the real echo.

Servo

The servo used on the final system is a generic "micro servo" with a total range of rotation of approximately 150 degrees. We experimentally determined the maximum and minimum values we could use as the duty cycle to the servo to be 750us and 2200us respectively. The Arduino Servo library handled setting the period of the communication to the servo, so we only had to set the high time per cycle.

After our original micro servo failed in a crash, we switched to the one used in our final system, which seemed to have some problems moving to various positions accurately.

Sonar

A Senscomp 6500 ultrasonic ranging module board was used to power the sonar transducer. This board receives a signal from the Arduino board, and creates a ping on the transducer by sending roughly 16 high-low transitions between +200 and -200 volts at 50kHz. The resulting sonar chirp travels outwards at the speed of sound, and the echo of the sound bounces off any objects and is detected by

the transducer. When the echo is received it pulls down a line connected to the Arduino, and the appropriate reading is recorded by the Arduino.

XBee wireless module

Wireless transmission of sonar readings to the computer base station was achieved using XBee Series 1 modules. The XBee modules use the IEEE 802.15.4 networking protocol to create a virtual serial link over a 2.4Ghz wireless connection. The XBees were configured to communicate at 9600 bps, with the 8N1 serial configuration – 8 data bits, no parity bit, and one stop bit. This speed proved more than fast enough to transfer all the data necessary, because we were only transmitting less than 2000bps (approx. 20 readings per second x approx. 8 chars per line x 8 bits per char = 1280bps). No parity was necessary because the XBee modules have built-in error correction, which was very useful for establishing a reliable serial link for our project.

3 Control software

3.1 Overview

The control software running on the Linux laptop serves a simple purpose at a high level – it takes sonar readings at various readings as an input, and based on these readings, it produces a movement command for the drone at regular intervals. The drone can be controlled by changing the front-back angle ("pitch"), left-right angle ("roll"), rotational angle about a vertical axis ("yaw"), and vertical speed. Because we were not concerned with vertical speed in this project, the program had to produce values for the first three parameters and send them to the drone. All the code is available in Appendix C

3.2 AR.Drone SDK

Our control software relied heavily on the AR.Drone SDK, provided by Parrot, the manufacturer of the quadcopter. The SDK, written in C, provides various functions which can be used to control the drone over a WiFi connection. To use the SDK, we created (using a special macro provided by the SDK) a thread named "mythread" in a file that previously contained a demo program, and registered the thread to be executed in parallel with other code in the SDK which manages communication with the drone. Thus, all the code that we wrote is contained in the "mythread" thread and a utility function called `ardrone_turn_tool` that we wrote.

We called two functions from the SDK in our program. The first, `ardrone_tool_set_ui_pad_start`, commands the drone to take off if a 1 is passed in, and commands it to land if a 0 is passed in. The next, `ardrone_at_set_progress_cmd` , takes 5 parameters. The first one is set to 1 to enable a command, and the next four values correspond to roll, pitch, vertical speed, and yaw values. Each of the last four parameters takes the form of a floating point value between 1.0

and -1.0, representing the maximum angle/speed that the drone can handle in opposite directions. As shown in Figure 4, we used macros to represent positive and negative values for each parameter in readable terms to make development easier.

```
56 //convert drone positive/negative angles to human-readable
       directions
57 #define CLOCKWISE 1
58 #define COUNTERCLOCKWISE -1
59 #define FORWARD -1
60 #define BACKWARD 1
61 #define LEFT -1
62 #define RIGHT 1
63 #define NO_TURN 0
```

Figure 4: Direction definitions (excerpt from Appendix C)

3.3 Control algorithm

The implementation of our control logic can be seen in Appendix C. This section will describe what the algorithm does at a conceptual level.

Sensor data processing

The first step of the algorithm is to determine the current distance from the wall, as well as the orientation relative to the wall, based on multiple sonar readings at different angles. Once the sonar finishes a full 180-degree scan in either direction, the algorithm processes all 15 readings for that scan, and finds the minimum echo time, which is assumed to correspond to the distance from the wall.

Finding orientation relative to the wall is slightly more involved – the scan angle where the minimum echo time was found might not actually be the angle that is perpendicular to the wall. This is because, rather than receive steadily decreasing and increasing echo delays as the sonar scans across the wall, the transducer always receives an echo from the perpendicular angle, so in fact the echo delay remains mostly constant for several scans until the wall is out of the angular range of the transducer. To process this effect, our algorithm works outwards from the minimum distance scan angle, finding scans where the echo delay was within a certain tolerance band ($\pm$ 1000 us) of the minimum delay. The range of scans that fit in this tolerance band is considered the "arc" of the wall that is detected, and the midpoint of this arc is considered the scan angle that is perpendicular to the wall.

This effect can be explained with some example diagnostic output from a test run (formatting edited to fit here), shown in Figure 5:

```
SCAN RESULTS:
FRONT [ 71  69 113  93  64  55  48  47  46  46  49  56  59 ] BACK
FRONT [                     *******MMMM***********          ] BACK
arc_start: 5, arc_end: 10, arc_mid: 7
```

Figure 5: Diagnostic output showing an arc

Here, the numbers are echo delay readings, with the last two digits of a microsecond value truncated (e.g. 71 means betwen 7100 and 7200 us). The minimum reading of 46 was detected on the scans with indices 8 and 9 (starting from the left with zero-indexing), but the middle of the arc was detected to be index 7, so the angle corresponding to index 7, not 8 or 9, is considered the angle which is perpendicular to the wall.

Computing roll angle

Two pieces of data are used to compute the roll angle to send to the drone: the current minimum distance to the wall, and the previous minimum distance, one scan ago. This information is used along with the calibration parameters in Figure 6 to calculate the roll angle.

```
#define LOW_THRESH 45   //*100 = minimum distance to wall in us
#define HIGH_THRESH 55 //*100 = maximum distance to wall in us
```

```
#define ROLL_MAX 0.08 //max roll angle
#define MAX_HORIZ_VELOCITY 20 //moving this *100 us towards/away
    from the wall in 1 scan
                              //will result in ROLL_CORRECT_MAX
                                  correction being applied
#define ROLL_CORRECT_MAX 0.08 //maximum correction by roll
    correction
```

Figure 6: Roll value calibration parameters (excerpt from Appendix C)

First, the algorithm uses the current distance from the wall to compensate for being too close to, or too far from, the wall. The high and low thresholds defined in the calibration parameters define a sort of band – inside this band, a linear control rule applies where the drone rolls more towards the center of the band if it is farther from the center; outside the band, the roll angle is limited to a maximum value. The maximum roll value is also defined in the calibration parameters. This control rule is visualized in Figure 7.

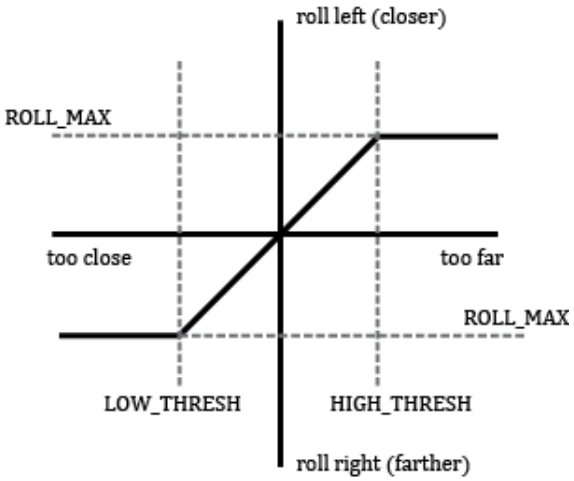


Figure 7: Roll angle control

To implement the control rule, the program calculates a coefficient that is essentially the slope of the middle part of the diagram – a number that multiplies a microsecond value to get a roll angle value, such that the angle value is plus or minus ROLL_MAX at LOW_THRESH and HIGH_THRESH. It multiplies this coefficient by the difference between the current distance from the wall and the ideal distance from the wall (halfway between the low and high thresholds), and then limits the value within ±ROLL_MAX. The code that implements this is shown in Figure 8.

```
410                         roll = (ideal_distance - min) *
                                roll_coefficient; //ideal_dist - min
                                gives a negative #, appropriately
411
412                         //limit roll within roll_max
413                         if(roll < (-1 * ROLL_MAX)){
414                                 roll = -1 * ROLL_MAX;
415                         }
416                         else if (roll > ROLL_MAX){
417                                 roll = ROLL_MAX;
418                         }
```

Figure 8: Roll value calculation part 1 (excerpt from Appendix C)

Computing roll correction

However, in testing, the roll angle control rule above was not sufficient to create stable behavior. The drone would gain momentum, and even if the roll angle was set to 0 once the drone was at the ideal distance from the wall, momentum from its previous roll angle would keep it moving. As a result, it was necessary to add some compensation for this effect, and have the drone roll slightly in the opposite direction of its movement. Figure 9 shows the calibration parameters and code used for this compensation.

```
70  #define MAX_HORIZ_VELOCITY 20 //moving this *100 us towards/away
        from the wall in 1 scan
71                                //will result in ROLL_CORRECT_MAX
                                      correction being applied
72  #define ROLL_CORRECT_MAX 0.08 //maximum correction by roll
        correction
```

```
424                                   roll_correct = (old_min - min) *
                                          roll_correct_coef;
425                                   if(roll_correct > (ROLL_CORRECT_MAX
                                          )) roll_correct =
                                          ROLL_CORRECT_MAX;
426                                   if(roll_correct < (-1 *
                                          ROLL_CORRECT_MAX)) roll_correct
                                           = -1 * ROLL_CORRECT_MAX;
427                                   fprintf(stderr, "roll correction: %
                                          f\n", roll_correct);
428                                   roll += roll_correct;
```

Figure 9: Roll value calculation part 2 (excerpt from Appendix C)

The code computes an approximation of the speed at which the drone is moving by subtracting the current distance from the wall from the distance one scan (approximately 750ms) ago. It then multiplies this speed by a coefficient, which is calculated to cause ROLL_CORRECT_MAX roll angle correction for a movement of (MAX_HORIZ_VELOCITY $\times$ 100) us in one scan. The roll angle correction is limited to ROLL_CORRECT_MAX, even for movement greater than that maximum horizontal velocity. Then, the roll angle correction is added to the roll angle computed in the previous step, to compute a final roll angle that is sent to the drone.

By having the drone attempt to slow its movement relative to the wall, the system achieves much more stable operation. Even if the drone is farther than the ideal distance to the wall, if it is moving very quickly towards the wall, it might not roll at all towards the wall (and might even roll away), because of the roll correction code. In our testing, we saw a noticeable improvement in performance with this additino. It was especially noticeable when the drone was approaching the wall rapidly – it was able to turn away from the wall aggressively

(a combination of the roll angle calculation and the movement correction both acting), in a way that would not have been appropriate if it was not moving towards the wall.

Computing rotational speed

The control algorithm also needs to consider the rotational angle of the drone relative to the wall. If the drone is not parallel to the wall, then any forward or back movement inherently causes it to move towards or away from the wall, so it is imperative that it remain parallel to the wall for stable operation.

We defined a range of indices in the array of scan readings (the "box"), which we considered to be acceptable values for the scan angle which is perpendicular to the wall (as explained previously, this angle is the midpoint of the "arc" that represents the wall). If the arc midpoint is in this range of scan readings (we defined it as index 5-9 based on testing), the rotational speed is set to zero. In other words, this is a deadband for the rotational speed of the drone. If the arc midpoint is outside the "box", then the appropriate correction is applied with a rotational speed. If the arc midpoint is too far towards the front of the drone, this means the nose of the drone is pointing towards the wall and it needs to turn clockwise. The opposite applies if the arc midpoint is too far towards the back of the drone. The rotational speed is a constant corresponding to a slow rotation – due to the relative imprecision of measuring the orientation of the drone, we did not find value in using any more sophisticated control rule. This control rule is visualized in Figure 10.

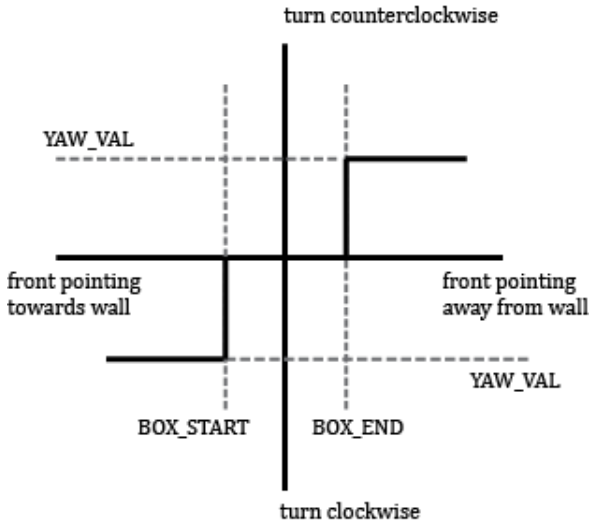


Figure 10: Rotational speed control

The code in Figure 11 shows the calibration parameter for rotational speed, as well as the code that implements the control rule.

```
53 #define BOX_START 5  //index in array where deadband for yaw begins
       -- compensate for asymmetric servo
54 #define BOX_END 9    //index in array where deadband for yaw ends
```

```
66 #define YAW_VAL 0.25  //turn speed
```

```
396                         if (arc_mid > BOX_END) { //front of drone
                                pointed away from wall
397                             fprintf(stderr, "OUTTA THE BOX YO,
                                    front pointed TOWARDS WALL!\n")
                                    ;
398                             yaw = CLOCKWISE;
399                         }
400                         else if (arc_mid < BOX_START) { //front of
                                drone pointed towards wall
401                             fprintf(stderr, "OUTTA THE BOX YO,
                                    front pointed AWAY FROM WALL!\n
                                    ");
402                             yaw = COUNTERCLOCKWISE;
403                         }
404                         else {
405                             fprintf(stderr, "GOOD ANGLE, DO NOT
                                     TURN\n");
406                             yaw = NO_TURN;
407                         }
```

Figure 11: Rotational speed calculation (excerpt from Appendix C)

Other simple control

There are several other control steps in our program unrelated to the main control algorithm. The drone takes off at the beginning of the program, and alternates between forwards and backwards movement regularly after a certain number of scans in each direction (defined by us as 10 scans, or about 8 seconds). It then lands after a certain number of direction switches (7 in our program). These parameters can be changed for different durations of demos, and different wall lengths.

4 Results

Our final system performs relatively well, achieving our goal of staying within 0 to 2 meters of the wall about 75% of the time. This is the result of extensive calibration to determine control parameters that would lead to stable operation. Due to weight constraints we could not meet our original goal of flying the drone

with its original battery, but the power supply solution works well, and has the added benefit of allowing unlimited operation.

A video of the drone successfully following the wall is included in the documentation that accompanies this paper.

Failures can be caused by a large number of factors, which explains why we consider 75% to be a relatively high success rate. We have experienced issues such as unreliable sonar readings, unreliable servo movement, and weight distribution problems with the drone, all of which can lead to unpredictable behavior and sometimes crashes into the wall. In fact, we had to significantly rebuild our system after one major crash.

5 Conclusion

5.1 Challenges and limitations

At the beginning of this project, we knew there would be significant challenges associated with it. Having a flying platform for a project instead of a more conventional ground-based one created a wide variety of difficulties.

Weight issues

The main problem we faced was dealing with weight – both the total weight of the payload on the drone, as well as its distribution. We initially thought that the drone was capable of carrying a normally constructed board of electronics, using an Arduino Uno and several auxiliary boards, but testing revealed that we were mistaken. We subsequently conducted controlled weight testing by putting bolts into a cup on top of the drone, and concluded that the maximum additional payload the drone could carry, with a battery and foam shell, was around 54 grams. Our electronics platform originally weighted approximately 150g, and we reduced the weight to less than 110g through various weight saving measures, but this was still far too heavy.

We tried approaching the problem by removing the 60g foam shell, but this proved too dangerous for the drone's propellers and led to a destructive crash (crashes are another problem that comes along with a flying platform). As a result, we were forced to remove the battery (which is heavier than the foam shell), and this allowed us to lighten the craft enough for it to fly. It also conveniently solved an issue we had experienced in our previous setup where our linear voltage regulator converting 11V from the battery to 5V for the electronics was overheating and failing. However, removing the battery also raised the center of gravity of the drone (since the battery was originally below our electronics), and caused increased instability. During testing we were forced to constantly adjust the location of electronics, and even add small weights to various parts of the drone, in order to even out its weight balance.

This challenge taught us to work in a resource constrained situation, which mirrors a real-life situation where engineers have to minimize weight, size, etc. We used a smaller microcontroller board and eliminated unnecessary boards and

wiring to significantly reduce weight. Creating a more robust hardware setup with less vulnerability to crashes may have improved our ability to conduct extensive tests on the drone.

Lack of position control

Unlike a ground-based robot with wheels, we lacked the ability to precisely control the drone's position and orientation. For example, even if we set all angles to zero, the drone could still drift significantly, based on its previous momentum, air currents, and other factors. This caused significant issues in stability which we had to deal with. Adding code to our control algorithm to compensate for the current direction of movement was the most effective thing we did to combat this problem. Any control algorithm for a quadcopter platform must take into account the lack of stable position control as a fundamental principle.

Noise

Our control algorithm had to be relatively resilient against random noise and sensor failures. We did not have to worry about a single sonar reading being slightly off, because we discarded some of the precision of the readings anyway. However, we did have to worry about a single reading being an outlier because of a small detail in the geometry of the wall and surrounding area. For example, if the drone was too far away from the wall and a single reading mistakenly reported it as close to the wall, it would roll away from the wall, causing a failure and even a crash.

To prevent this, we added some resistance to this type of problem in the code, where it would only accept an "arc" if the arc was more than just a single reading. In other words, if the minimum distance did not have a reading on either side of it that was approximately the same within a tolerance band (which should have never been the case for a correct reading from the wall), then that minimum distance reading would be discarded.

We also experienced an issue where the Arduino was reporting false echoes before reporting real echoes, and solved this by changing the Arduino code so it would only report the last echo it received at a given servo angle. We were unable to discover the root cause of this issue, but it probably had to do with hardware malfunctions caused by force from hitting the ground in crashes.

SDK limitations

We had to use the AR.Drone SDK to control the drone using high level commands– there is so much built in stabilization technology that we could not hope to control the drone's motors individually, even if we could discover a technical way to achieve that. Because of this dependency, we were limited by the capabilities of the SDK.

For example, at one point we could not get movement commands to work, and thought completing the project might be completely impossible. We eventually worked around the issue by downgrading to an older version of the SDK which did work, but we were unable to get a video feed from the drone using this older SDK. In addition, the SDK has relatively good documentation, but developer support from Parrot is sparse.

In general, the strict dependency on the SDK added risk to our project, and perhaps using an open-source quadcopter without this dependency could have been a safer choice.

5.2 Future work

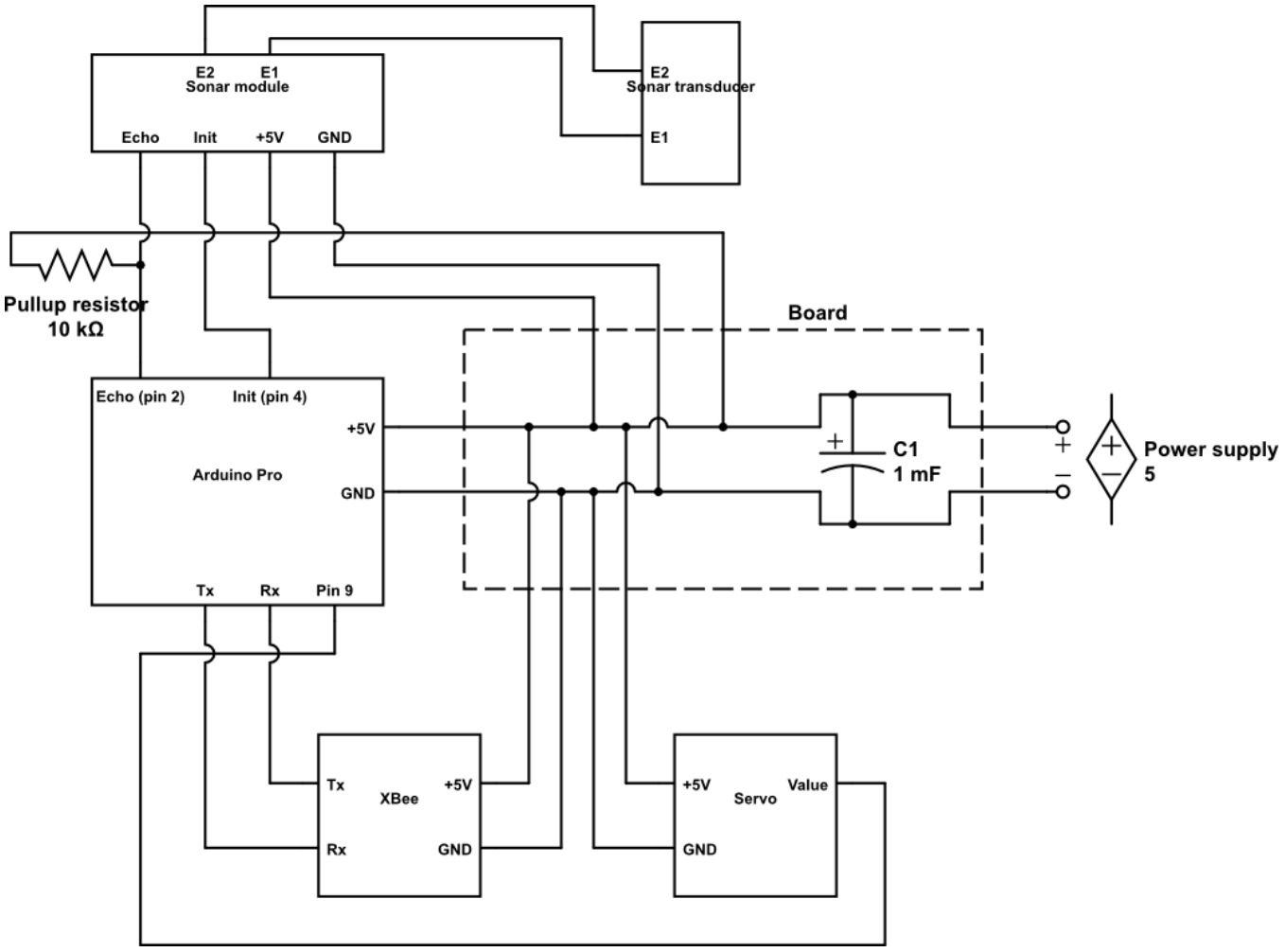
In general, the work conducted for this project could be continued in an effort to increase stability and reliability. A more robust platform and enclosure for the sensing hardware would have solved many of the issues we faced regarding reliability of sensors, and resilience to crashes. For example, a plastic 3D-printed platform which attached to the drone on the rods holding the propellers could maintain the light weight of the foam platform we used, but add extra structual support. It would also be useful to design a protective shell around the electronics that still exposed the sonar sensors appropriately.

In addition, the hardware setup we created could have a wide variety of potential applications. For example it could be feasible to write software that would move the drone along a pre-determined course, while dynamically avoiding obstacles along the way. This would essentially entail sending movement commands while never allowing the drone to get within a certain distance of an obstacle in any direction, and calcluating strategies to get around objects.

The drone could also potentially follow some sort of target, if that target was detectable by sonar. Another area to explore might be different kinds of sensors. Other types of range sensors like laser rangefinders could provide benefits over sonar (although weight is an important consideration to keep in mind). Also, the cameras onboard the drone, as well as additional cameras, might be used to perform image processing on the environment to discover information that sonars and range sensors would not be able to. Because the quadcopter can fly over objects, improvements like this might make the quadcopter useful for autonomous surveillance, delivery of light payloads, navigation of dangerous terrain, and other real-world applications. When a quadcopter can accurately detect and react to its environment, the possibilities are endless. Following a wall using sonar may be a good start.

Appendix

A Schematic for sensing hardware



B Arduino code

```
1  //Sonar reading code for EENG 471
2  //Caroline Jaffe and Geoffrey Litt
3
4  #include <Servo.h>
5
6  //Servo configuration parameters
7  const int SERVO_PIN = 9; //pin the servo is attached to
8  const int SERVO_MIN = 750; //minimum pulse length to servo (in us)
9  const int SERVO_MAX = 2200; //maximum pulse length to servo (in us)
10 const int SERVO_ANGLE_RANGE = 180; //total angle (deg) swept by
       servo
11 const int SERVO_INCR_ANGLE = 15; //deg to move servo between
       measurements
12 const int US_PER_DEGREE = (SERVO_MAX - SERVO_MIN)/SERVO_ANGLE_RANGE
       ;
13
14 //Sonar configuration parameters
15 const int ECHO_PIN = 2; //echo input pins for sonar 0 and 1
16                                 //MUST be 2 and 3 (ext. interrupt
                                       pins on Arduino Uno)
17 const int INIT_PIN = 4;
18
19 //Global variables
20 Servo myservo;  // create servo object to control a servo
21                 // a maximum of eight servo objects can be created
22 unsigned long init_time;
23 int servoAngle;
24 int prev_servo_angle = 0;
25 int prev_echo_delay = 0;
26
27 void setup()
28 {
29   //setup servo
30   myservo.attach(SERVO_PIN);
31
32   //setup sonar reading
33   Serial.begin(9600);
34   pinMode(INIT_PIN, OUTPUT);
35   pinMode(ECHO_PIN, INPUT);
36   attachInterrupt(1, handleEcho, RISING);
37 }
38
39
40 void loop()
41 {
42     int pos;
43     for(servoAngle = 0; servoAngle < SERVO_ANGLE_RANGE; servoAngle
           += SERVO_INCR_ANGLE){
44       pos = SERVO_MIN + (servoAngle * US_PER_DEGREE);
45       myservo.writeMicroseconds(pos);
46       sonarPulse();
47     }
48     for(servoAngle = SERVO_ANGLE_RANGE; servoAngle > 0; servoAngle
           -= SERVO_INCR_ANGLE){
49       pos = SERVO_MIN + (servoAngle * US_PER_DEGREE);
```

```
50        myservo.writeMicroseconds(pos);
51        sonarPulse();
52      }
53  }
54
55  void sonarPulse(){
56      digitalWrite(INIT_PIN, HIGH);
57      init_time = micros();
58      delay(20);
59      digitalWrite(INIT_PIN, LOW);
60      delay(30);
61  }
62
63  void handleEcho()
64  {
65    //only sends data once the next reading
66    //is taken at a different angle, avoids
67    //duplicate readings
68
69    long echo_delay = micros() - init_time;
70
71    if(prev_servo_angle != servoAngle){ //not a duplicate reading
72      Serial.print(prev_servo_angle);
73      Serial.print(":");
74      Serial.println(prev_echo_delay);
75    }
76    //remember these values to send
77    prev_servo_angle = servoAngle;
78    prev_echo_delay = echo_delay;
79  }
```

C Desktop code

```
/**
 * WALLFOLLOWING DEMO
 * EENG 471 PROJECT
 * CAROLINE JAFFE AND GEOFFREY LITT

 BASED ON SCAFFOLDING ARDRONE LIBRARY FILE BY:
 * @file main.c
 * @author sylvain.gaeremynck@parrot.com
 * @date 2009/07/01
 */

//ARDroneLib

#include <ardrone_tool/ardrone_time.h>
#include <ardrone_tool/Navdata/ardrone_navdata_client.h>
#include <ardrone_tool/Control/ardrone_control.h>
#include <ardrone_tool/UI/ardrone_input.h>

//Common
#include <config.h>
#include <ardrone_api.h>

//VP_SDK
#include <ATcodec/ATcodec_api.h>
#include <VP_Os/vp_os_print.h>
#include <VP_Api/vp_api_thread_helper.h>
#include <VP_Os/vp_os_signal.h>

//Local project
#include <UI/gamepad.h>
#include <Video/video_stage.h>

//our custom includes for wall-follow logic
#include <unistd.h>
#include <ardrone_testing_tool.h>
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <termios.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <limits.h>

#define error_message printf

//calibrate thresholds for yaw angle and distance from wall
#define ARC_BAND 10     //*100*2 = width of tolerance band for arc
    detection
#define LOW_THRESH 45   //*100 = minimum distance to wall in us
#define HIGH_THRESH 55 //*100 = maximum distance to wall in us
#define BOX_START 5  //index in array where deadband for yaw begins
     -- compensate for asymmetric servo
```

```
54 #define BOX_END 9    //index in array where deadband for yaw ends
55
56 //convert drone positive/negative angles to human-readable
       directions
57 #define CLOCKWISE 1
58 #define COUNTERCLOCKWISE -1
59 #define FORWARD -1
60 #define BACKWARD 1
61 #define LEFT -1
62 #define RIGHT 1
63 #define NO_TURN 0
64
65 //calibrate magnitude of drone movements
66 #define YAW_VAL 0.25  //turn speed
67 #define THETA_FORWARD_VAL -0.06// forward angle
68 #define THETA_BACK_VAL 0.06 //back angle
69 #define ROLL_MAX 0.08 //max roll angle
70 #define MAX_HORIZ_VELOCITY 20 //moving this *100 us towards/away
       from the wall in 1 scan
71                               //will result in ROLL_CORRECT_MAX
                                      correction being applied
72 #define ROLL_CORRECT_MAX 0.08 //maximum correction by roll
       correction
73
74 //calibrate general demo parameters
75 #define SLEEP_AFTER_TAKEOFF 5 //seconds to sleep after takeoff
76 #define SCANS_BEFORE_START 4 //number of scans to read before
       starting movement commands
77 #define SCANS_BEFORE_SWITCH 10 //number of 180-deg scans before
       forward/back dir switches
78 #define SWITCHES_BEFORE_LAND 7  //number of direction switches
       before landing
79 #define ARRAY_LEN 13 //number of sonar readings (13 = 180/15)
80
81 // set parameters for data smoothing after reading in from the
       sonars
82 #define SCAN_UP 1
83 #define SCAN_DOWN 0
84
85 //UTILITY FUNCTION TO CONTROL DRONE MOVEMENT
86 void ardrone_turn_tool(int dir, int yaw_type, float roll, int FLY)
87 {
88         float pitch;
89         if(dir == FORWARD){
90                 pitch = THETA_FORWARD_VAL;
91         }
92         else{
93                 pitch = THETA_BACK_VAL;
94         }
95
96         float yaw = yaw_type * YAW_VAL;
97
98         //debug output for forward angle + rotate speed
99         if(dir == FORWARD){
100                fprintf(stderr, "FORWARD, ");
101        }
102        else{
```

```
103                 fprintf(stderr, "BACKWARD, ");
104         }
105
106         if(roll == -0.1){
107                 fprintf(stderr, "LEFT ROLLMAX, ");
108         }
109         else if(roll == 0.1){
110                 fprintf(stderr, "RIGHT ROLLMAX, ");
111         }
112         else if(roll > 0){
113                 fprintf(stderr, "RIGHT ROLL, ");
114         }
115         else if(roll < 0){
116                 fprintf(stderr, "LEFT ROLL, ");
117         }
118         else{
119                 fprintf(stderr, "NO ROLL, ");
120         }
121
122         if(yaw_type == CLOCKWISE){
123                 fprintf(stderr, "CLOCKWISE => ");
124         }
125         else if(yaw_type == COUNTERCLOCKWISE){
126                 fprintf(stderr, "COUNTERCLOCKWISE => ");
127         }
128         else{
129                 fprintf(stderr, "NO YAW => ");
130         }
131
132         fprintf(stderr, "pitch = %f, roll = %f, yaw = %f\n", pitch,
              roll, yaw);
133
134         if(FLY){ //only send command if FLY enabled
135           ardrone_at_set_progress_cmd(1,roll,pitch,0,yaw); //
                command turn
136         }
137 }
138
139 //UTILITY FUNCTIONS TO SET UP SERIAL READING
140 //From external source: http://stackoverflow.com/questions/6947413/
        how-to-open-read-and-write-from-serial-port-in-c
141
142 int
143 set_interface_attribs (int fd, int speed, int parity)
144 {
145         struct termios tty;
146         vp_os_memset (&tty, 0, sizeof tty);
147         if (tcgetattr (fd, &tty) != 0)
148         {
149                 error_message ("error %d from tcgetattr", errno);
150                 return -1;
151         }
152
153         cfsetospeed (&tty, speed);
154         cfsetispeed (&tty, speed);
155
```

```
156         tty.c_cflag = (tty.c_cflag & ~CSIZE) | CS8;     // 8-bit
                chars
157         // disable IGNBRK for mismatched speed tests; otherwise
                receive break
158         // as \000 chars
159         tty.c_iflag &= ~IGNBRK;         // ignore break signal
160         tty.c_lflag = 0;                // no signaling chars, no
                echo,
161         // no canonical processing
162         tty.c_oflag = 0;                // no remapping, no delays
163         tty.c_cc[VMIN]  = 0;            // read doesn't block
164         tty.c_cc[VTIME] = 5;            // 0.5 seconds read timeout
165
166         tty.c_iflag &= ~(IXON | IXOFF | IXANY); // shut off xon/
                xoff ctrl
167
168         tty.c_cflag |= (CLOCAL | CREAD);// ignore modem controls,
169         // enable reading
170         tty.c_cflag &= ~(PARENB | PARODD);      // shut off parity
171         tty.c_cflag |= parity;
172         tty.c_cflag &= ~CSTOPB;
173         tty.c_cflag &= ~CRTSCTS;
174
175         if (tcsetattr (fd, TCSANOW, &tty) != 0)
176         {
177                 error_message ("error %d from tcsetattr", errno);
178                 return -1;
179         }
180         return 0;
181 }
182
183 void
184 set_blocking (int fd, int should_block)
185 {
186         struct termios tty;
187         vp_os_memset (&tty, 0, sizeof tty);
188         if (tcgetattr (fd, &tty) != 0)
189         {
190                 error_message ("error %d from tggetattr", errno);
191                 return;
192         }
193
194         tty.c_cc[VMIN]  = should_block ? 1 : 0;
195         tty.c_cc[VTIME] = 5;            // 0.5 seconds read timeout
196
197         if (tcsetattr (fd, TCSANOW, &tty) != 0)
198                 error_message ("error %d setting term attributes",
                        errno);
199 }
200
201 static int32_t exit_ihm_program = 1;
202 //END OF SERIAL UTILITY FUNCTIONS
203
204 //========================================
205 //DEFINE CUSTOM THREAD TO CONTROL THE DRONE
206 //(wall-following logic contained within)
207 //========================================
```

```
208
209 PROTO_THREAD_ROUTINE(mythread, nomParams);
210 DEFINE_THREAD_ROUTINE( mythread, nomParams)
211 {
212         //our thread writes all outputs to stderr.
213         //when the ar_drone program stdout is redirected to a file,
214         //this enables us to see just our thread's output in a
              terminal.
215         //
216         //example execution: sudo ./linux_sdk_demo > drone.log
217
218         //variables for wallfollow logic
219         int i, arc_start, arc_end, arc_mid;
220         int arr[ARRAY_LEN];
221         int scan_dir = SCAN_UP;
222         int angle, micros;
223         int min, min_ind;
224         int old_min = -1;
225         int FLY = 1; //change this value to disable/enable flying
226
227         //calculate ideal distance from the wall based on
              thresholds
228         const int ideal_distance = (LOW_THRESH + HIGH_THRESH) / 2;
229         //calculate roll coefficient: this * distance from ideal =
              roll value
230         const float roll_coefficient = ROLL_MAX / (HIGH_THRESH -
              ideal_distance);
231
232         //calculate roll correction coefficient:
233         //this * distance towards wall since last scan = amount to
              correct roll value in opposite dir.
234         const float roll_correct_coef = ROLL_CORRECT_MAX /
              MAX_HORIZ_VELOCITY;
235
236         fprintf(stderr, "ideal dist: %d, roll_coeff: %f\n",
              ideal_distance, roll_coefficient);
237         //initialize counters to zero
238         int switch_counter = 0; //counts # of direction switches so
               far
239         int scan_counter = 0; //counts # of 180-deg sonar scans in
              the current direction
240         int total_scan_count = 0; //number of 180-deg sonar scans
              so far this flight
241
242         //initialize drone directions to move straight forward
243         int dir = FORWARD;
244         int yaw = NO_TURN;
245         float roll = 0;
246         float roll_correct;
247
248         if(FLY){
249                 fprintf(stderr, "Flying commands ENABLED!\n");
250         }
251         else{
252                 fprintf(stderr, "Flying commands DISABLED.\n");
253         }
254
```

```
255         //---SERIAL CODE---
256         //try opening Xbee on USB1 first, then try USB0
257         char *portname = "/dev/ttyUSB1";
258         char buf[100];
259         int fd = open (portname, O_RDWR | O_NOCTTY | O_SYNC);
260
261         if (fd < 0)
262         {
263                 error_message ("error %d opening %s: %s\n", errno,
                        portname, strerror (errno));
264                 portname = "/dev/ttyUSB0";      //try the other USB
                        option
265                 fd = open (portname, O_RDWR | O_NOCTTY | O_SYNC);
266
267                 if (fd < 0)
268                 {
269                         error_message ("error %d opening %s: %s\n",
                                errno, portname, strerror (errno));
270                         THREAD_RETURN(1);        //neither USB port
                                worked
271                 }
272         }
273
274         fprintf(stderr, "USB open successful: %s\n", portname);
275
276         set_interface_attribs (fd, B9600, 0);  //set 9600bps,
                parity
277         set_blocking (fd, 1);                // set blocking
278         FILE *fp = fdopen(fd, "r");
279
280         //---END OF SERIAL CODE---
281
282
283         //---WALLFOLLOW LOGIC---
284
285         fprintf(stderr, "Taking off now...\n");
286         if(FLY){
287                 ardrone_tool_set_ui_pad_start(1);
288         }
289         fprintf(stderr, "sleeping for SLEEP_AFTER_TAKEOFF seconds
                ...\n");
290         sleep(SLEEP_AFTER_TAKEOFF); //tweak this value
291         fprintf(stderr, "takeoff routine complete, main logic
                beginning...\n");
292
293         fflush(fp); //flush the scans we got on the serial line
                while taking off
294
295         while(1){ //run this loop indefinitely
296
297                 //default movements
298                 yaw = NO_TURN;
299                 roll = 0;
300
301                 fprintf(stderr, "\nSCAN RESULTS:\n");
302                 while(fgets(buf, 100, fp) == NULL); //wait while
                        nothing to process from serial input
```

```
303                 while(fgets(buf, 100, fp) != NULL){
304                         sscanf(buf, "%d:%d\n", &angle, &micros);
305                         if(micros > 1000){ //discard erroneous
                                readings below 1000
306                             arr[angle/15] = micros/100; //store
                                 recorded sonar value in array
                                of readings

                                                        //(if a
                                 reading is missed, the
                                previously scanned value
                                remains)
307
308                             //process the scan data if we reach
                                 either endpoint of the scan
309                             if(angle == 0){
310                               scan_dir = SCAN_DOWN;
311                               break;
312                             }
313                             if (angle == 180){
314                               scan_dir = SCAN_UP;
315                               break;
316                             }
317                         }
318                 }
319
320                 arc_start = arc_end = arc_mid = 0;
321                 min = INT_MAX;
322
323                 min_ind = 0; //useless init value to stop compiler
                        warnings, should be overwritten always
324
325                 //Find min
326                 for (i = 0; i < ARRAY_LEN; i++) {
327                         if (arr[i] < min) {
328                                 min = arr[i];
329                                 min_ind = i;
330                         }
331                 }
332
333                 if(old_min == -1) old_min = min; //init old_min on
                        first scan
334
335                 //Find the arc start- and end- points
336                 arc_start = arc_end = min_ind; //in case the arc is
                         one wide
337                 i = min_ind-1;
338                 while (i >= 0 && abs(arr[i]-min) < ARC_BAND) {
339                         arc_start = i;
340                         i--;
341                 }
342                 i = min_ind+1;
343                 while (i < ARRAY_LEN && abs(arr[i]-min) < ARC_BAND)
                         {
344                         arc_end = i;
345                         i++;
346                 }
```

```
347                 arc_mid = (arc_start+arc_end)/2; //index which is
                        the center of the arc
348
349                 // adjust arc readings based on scan direction
350                 // this compensates for inaccuracies in scan data
                        depending on scan direction
351                 if (scan_dir == SCAN_DOWN && arc_mid < ARRAY_LEN){
352                         arc_start++;
353                         arc_end++;
354                         arc_mid++;
355                 }
356                 else if (scan_dir == SCAN_UP && arc_mid > 0){
357                         arc_start--;
358                         arc_end--;
359                         arc_mid--;
360                 }
361
362                 //print sonar readings to terminal
363                 fprintf(stderr, "BACK ranges: [");
364                 for(i = 0; i < 13; i++){
365                         fprintf(stderr, "%4d ", arr[i]);
366                 }
367                 fprintf(stderr, "] FRONT\n");
368
369                 //print arc calculation readings to terminal
370                 fprintf(stderr, "BACK arc:    [");
371                 for(i = 0; i < 13; i++){
372                         if(i == arc_mid){
373                                 fprintf(stderr, "MMMMM");
374                         }
375                         else if(i >= arc_start && i <= arc_end){ //
                                in arc
376                                 fprintf(stderr, "*****");
377                         }
378                         else{
379                                 fprintf(stderr, "     ");
380                         }
381                 }
382                 fprintf(stderr, "] FRONT\n");
383                 fprintf(stderr, "arc_start: %d, arc_end: %d,
                        arc_mid: %d\n", arc_start, arc_end, arc_mid);
384
385                 //ignore the first couple scans, which may not
                        contain full data
386                 total_scan_count++;
387                 if(total_scan_count <= SCANS_BEFORE_START){
388                         fprintf(stderr, "DISCARDING THIS INITIAL
                                SCAN DATA\n");
389                         continue;
390                 }
391
392
393                 if(arc_start < arc_end){ //only send a command if
                        arc more than 1 wide, prevents noise
394
395                         //control based on yaw angle
```

```
396                         if (arc_mid > BOX_END) { //front of drone
                                pointed away from wall
397                             fprintf(stderr, "OUTTA THE BOX YO,
                                    front pointed TOWARDS WALL!\n")
                                    ;
398                             yaw = CLOCKWISE;
399                         }
400                         else if (arc_mid < BOX_START) { //front of
                                drone pointed towards wall
401                             fprintf(stderr, "OUTTA THE BOX YO,
                                    front pointed AWAY FROM WALL!\n
                                    ");
402                             yaw = COUNTERCLOCKWISE;
403                         }
404                         else {
405                             fprintf(stderr, "GOOD ANGLE, DO NOT
                                     TURN\n");
406                             yaw = NO_TURN;
407                         }
408
409                         //control based on distance from wall
410                         roll = (ideal_distance - min) *
                                roll_coefficient; //ideal_dist - min
                                gives a negative #, appropriately
411
412                         //limit roll within roll_max
413                         if(roll < (-1 * ROLL_MAX)){
414                             roll = -1 * ROLL_MAX;
415                         }
416                         else if (roll > ROLL_MAX){
417                             roll = ROLL_MAX;
418                         }
419
420                         if(old_min != -1){ //make sure it's
                                initialized
421                             //add roll correction to roll value
422
423                             //compute correction based on
                                    difference between current min
                                    and previous min
424                             roll_correct = (old_min - min) *
                                    roll_correct_coef;
425                             if(roll_correct > (ROLL_CORRECT_MAX
                                    )) roll_correct =
                                    ROLL_CORRECT_MAX;
426                             if(roll_correct < (-1 *
                                    ROLL_CORRECT_MAX)) roll_correct
                                     = -1 * ROLL_CORRECT_MAX;
427                             fprintf(stderr, "roll correction: %
                                    f\n", roll_correct);
428                             roll += roll_correct;
429
430                         }
431                 }
432                 else{
433                         //if the arc is only one wide, don't send
                                any command
```

```
434                         fprintf(stderr, "NO SUFFICIENTLY WIDE ARC
                                FOUND. DO NOT TURN\n");
435                         yaw = NO_TURN;
436                 }
437
438                 if (scan_dir == SCAN_DOWN){
439                         fprintf(stderr, "scan_dir is SCAN_DOWN,
                                INCREMENTED VALUES\n");
440                 }
441                 else if (scan_dir == SCAN_UP){
442                         fprintf(stderr, "scan_dir is SCAN_UP,
                                DECREMENTED VALUES\n");
443                 }
444
445                 //SEND the movement commands that were calculated
446     ardrone_turn_tool(dir, yaw, roll, FLY); //will only turn if FLY
          is enabled
447
448     //switch directions after the specified number of scans
449                 scan_counter++;
450                 if (scan_counter >= SCANS_BEFORE_SWITCH) {
451                         scan_counter = 0; //reset scan counter
452                         switch_counter++; //keep track of direction
                              switch count
453                         dir *= -1; //switch directions
454                         fprintf(stderr,  "flight counter overflow,
                                switching directions to ");
455                         if(dir == FORWARD){
456                                 fprintf(stderr, "FORWARD\n");
457                         }
458                         else{
459                                 fprintf(stderr, "BACKWARD\n");
460                         }
461
462                         if(switch_counter >= SWITCHES_BEFORE_LAND){
463                                 //land the drone, end of flight
464                                 fprintf(stderr, "switch counter
                                        overflow, landing...\n");
465                                 break; //exit the while 1 loop
466                         }
467                 }
468
469                 old_min = min;
470         }
471
472         //always land before completing the program! we don't want
              a stranded drone.....
473   if(FLY){
474           ardrone_tool_set_ui_pad_start(0);
475   }
476
477   close(fd);
478   THREAD_RETURN(0);
479 }
480
481 //============================
482 //BOILERPLATE AR_DRONE API CODE
```

```
483 //============================
484
485 /* The delegate object calls this method during initialization of
        an ARDrone application */
486 C_RESULT ardrone_tool_init_custom(int argc, char **argv)
487 {
488   /* Registering for a new device of game controller */
489   ardrone_tool_input_add( &gamepad );
490
491   /* Start all threads of your application */
492   START_THREAD( mythread, NULL);
493   START_THREAD( video_stage, NULL );
494
495   return C_OK;
496 }
497
498 /* The delegate object calls this method when the event loop exit
        */
499 C_RESULT ardrone_tool_shutdown_custom()
500 {
501   /* Relinquish all threads of your application */
502   JOIN_THREAD( video_stage );
503
504   /* Unregistering for the current device */
505   ardrone_tool_input_remove( &gamepad );
506
507   return C_OK;
508 }
509
510 /* The event loop calls this method for the exit condition */
511 bool_t ardrone_tool_exit()
512 {
513   return exit_ihm_program == 0;
514 }
515
516 C_RESULT signal_exit()
517 {
518   exit_ihm_program = 0;
519
520   return C_OK;
521 }
522
523 /* Implementing thread table in which you add routines of your
        application and those provided by the SDK */
524 BEGIN_THREAD_TABLE
525   THREAD_TABLE_ENTRY( ardrone_control, 20 )
526   THREAD_TABLE_ENTRY( navdata_update, 20 )
527   THREAD_TABLE_ENTRY( video_stage, 20 )
528   THREAD_TABLE_ENTRY( mythread, 20)
529 END_THREAD_TABLE
```