Using a Sonar Ranging System to Develop Autonomous Wall-Following for a Quadcopter

Caroline Jaffe
Class of 2013
B.S. Candidate, EECS

December 22, 2012

EENG 471a
Professor Roman Kuc
Independent Project in collaboration with Geoffrey Litt '14
Yale University

Abstract

The purpose of this project was to develop a sensing system for a quadrotor helicopter so that it could follow and avoid a wall. This project integrated many different disciplines and techniques, ranging from software development and circuit design to mechanical assembly and control theory. We used our control algorithm to determine movement commands based on readings from a sonar ranging module mounted on top of the quadrotor. Our calibrations and testing produced a set of parameters for our control algorithm that allowed the quadrotor to successfully stay within 0.5 – 2m of a wall about 75% of the time. The main limitation we faced was the instability of the system due to the addition of our sensing hardware, which made it challenging to wield consistent control of the quadrotor.

1 Introduction

1.1 Overview

The purpose of this project was to develop a sensing system so that a quadrotor helicopter ("quadcopter" or "drone") could follow and avoid a wall. The Parrot AR.Drone 2.0 is a sophisticated consumer technology with onboard capabilities such as internal gyroscopic stabilization, altitude measurement using sonar, and two onboard cameras. These capabilities make it applicable to a wide range of monitoring and transport activities. However, the system lacks an obstacle avoidance system. If the quadcopter is sent movement commands without environmental feedback, it could crash into obstacles at any time, thereby limiting its effectiveness. Our project aimed to develop a sensing system that would detect objects in the environment and autonomously navigate away from them, making the drone suitable for a wide range of useful and interesting applications.

1.2 Background

This project built upon our shared background working with microcontrollers and writing low-level sensing software. We had experience working together on a similar type of "hacking" project with our previos work on Keepon robots. This project introduced the added complexity of mechanical design and control theory and challenged us to integrate our experiences in a collaborative and dynamic environment.

1.3 Specifications

In order to demonstrate the drone's obstacle avoidance capability, we decided to develop a system that could follow a wall at a constant distance. This demonstration would be a proof of concept that indicated the viability of a more advanced obstacle avoidance system. Throughout the course of the project, we determined that a reasonably objective was to keep the drone within 0.5 to 2 meters of the wall.

This undertaking required tackling a number of different engineering tradeoffs and integrating many different disciplines and techniques. First, the drone obviously needed to be able to fly. Although the drone could fly well unfettered by a sensing system, our additional sensors impeded its flying

ability because of their weight. Once weight reduction became a priority, we had to consider the value of including the battery versus the foam shell versus components of our sensing system. Eventually, we decided to sacrifice battery-powered operation in this tradeoff; our final setup was powered by a cable to a DC power supply.

We encountered another decision point in the development of a fast and accurate sensing system. We found that the speed of the servo and the accuracy of our sonar reading were inversely proportional; as we reduced the delay between setting the position of the servo, we found more irregularities in our sonar readings. Through testing, we found that a reasonable compromise between these two factors was allowing 750ms for a complete scan of 150 degrees, with about 50ms for each 15 degree increments.

Another area of consideration was in the accuracy and processing of the sonar measurements. These measurements were extremely precise. We eventually decided to divide each of the measurements by 100 both to improve readability, but also to smooth out noise from small features in the environment. In order to use these readings to determine the path of the drone, it was essential to have a reliable wireless connection to our Linux system. If this wireless communication failed, the obstacle avoidance ability of the drone would be severely hindered. This project required us to come to terms with a number of fundamental tradeoffs and design decisions in the course of the development of this basic demonstration. We are pleased that this process lay the groundwork for other projects that wish to use the drone as a sensing and control platform.

2 Design

Throughout the course of the project, we relied upon several high-level design objectives and development techniques. I will briefly discuss these here before delving into a detailed discussion of the project's individual components.

We originally proposed a project in which the drone would follow an object by applying image recognition techniques to the video stream from the drone's camera. However, after some initial difficultly accessing the video stream because of errors and inconsistencies in the latest version of the AR.Drone

SDK, we decided to pivot to a project that would be more viable with the hardware we had access to, and would better engage our electrical engineering skill set. The wall-following project fit these specifications, and would develop a platform that could eventually be extended to other applications like obstacle avoidance and object following.

In designing the hardware platform, we considered several different sensing options, such as vergence sensing with a pair of sonar sensors and the CMUcam, an open source programmable vision sensor. Our choice of using two sonars on a rotating servo was influenced by our desire to process sensing data for the entire horizontal plane, which would have been possible but more difficult with either of the other options because of their additional torque. We felt that obtaining a wide scan of data would make the platform more accessible to other applications. Additionally, we liked the idea of creating our own system instead of using one that was already developed, because it would be easier to integrate with the AR.Drone SDK, which is already a complex, built-up system.

In general, our project was driven by empirical experimentation. We tried to isolate different variables and test them individually through observation in the environment. We would integrate different tested components and calibrate the combined system, gradually building up to our final system. Another important design consideration of our system was its suitability for rapid iteration and failure. Much of our system was built with soft joints that were easily detachable so that the drone's physical integrity wouldn't be completely devastated if it crashed. We also made it easy to modify the system's weight distribution with on-the-fly changes, as our constantly evolving platform could change from run to run.

3 Equipment

- Sparkfun Arduino Pro 328 microcontroller – 3.3V/8MHz
- Parrot AR.Drone 2.0 Quadcopter with indoor hull
- AR.Drone 1.8 Software Development Kit (SDK)
- Set of XBee Series 1 wireless communication devices

- Generic "micro servo" with 150 degree range
- Senscomp 6500 ultrasonic sonar ranging module and instrument grade transducer
- 15" Dell Inspiron laptop running Ubuntu
- 5V and 11V DC power supply
- USB-FTDI cable (for connecting to Xbees, Arduino)

4 Methods

This project consisted of the integration of many different components and systems. Each component went through its own development process before it could be integrated into the final project. The final assembly consisted of two main parts: our custom hardware assembly mounted on the AR.Drone quadcopter, and the Linux laptop running Arduino and SDK software.

The electronics platform mounted on the AR.Drone quadcopter, which can be seen in Figures 1 and 9 and the schematic in Appendix A, consisted of an Arduino Pro that drove a servo and controlled a sonar sensing module. The sonar transducer was mounted on top of the rotating servo so that it could take readings from the entire horizontal plane. It would receive readings at various angles and print these readings in a standardized protocol over a serial connection to an XBee device. This custom hardware was only for the purpose of sensing, and did not directly control the drone's motions.

The Linux laptop is connected to the quadcopter by a WiFi network broadcast by the drone itself. The laptop would receive sensor readings from the XBee through an FTDI cable. We wrote a program that would receive these sensor readings, process them, and then use the Parrot API to send movement commands to the drone through its WiFi network. The drone itself and the electronics setup were powered individually by a tethered power supply, with 11V and 5V respectively. Here, I will describe the individual specifications each component in detail.

4.1 Hardware

4.1.1 Parrot AR.Drone 2.0 Quadcopter with indoor hull and Parrot AR.Drone SDK Version 1.8

We selected the Parrot AR.Drone quadcopter because it was a reasonably priced piece of hardware that had good documentation and a fairly extensive SDK. The drone has 4 brushless motors that spin specially designed high-efficiency propellers. It is equipped with a wide angle camera on the front that streams video over the drone's WiFi network and camera on the bottom that provides data for system stabalization. The drone has an ultrasound altimeter with a vertical range of 6m. The system has a 3 axis accelerometer, 2 axis gyrometer, and can fly at a top speed of 5m/s. Notably, the SDK abstracted communication with the drone's motors, a feature which was appropriate for our project because we hoped to focus more on sensing than on low-level control.

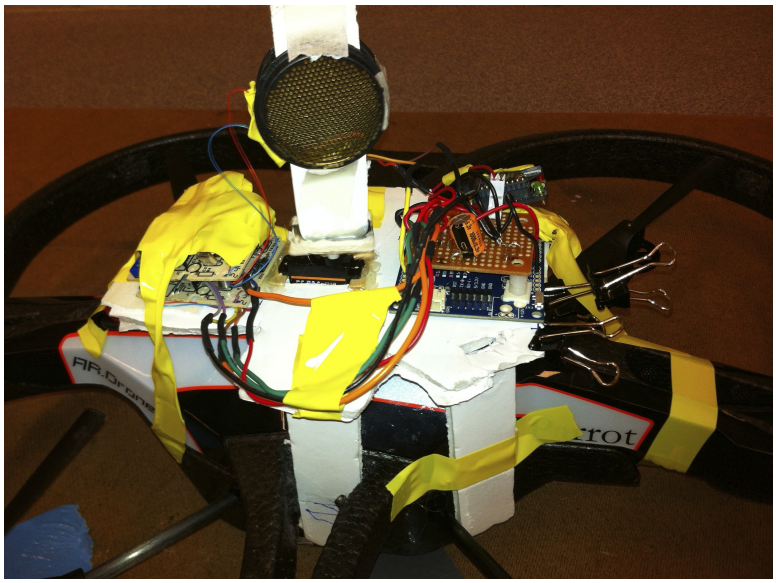


Figure 1: Closeup view of the sensing platform mounted on the drone.

4.1.2 Sensing platform and project setup

All of the custom sensing components were arranged on a foamcore board that rested above the body of the drone, as pictured in Figure 1. They were

arranged such that the servo was at the center, with the Arduino and XBee device on one side, and the sonar boards on the other. (Note that the second sonar board pictured in the image is a relic of an earlier system; our final system only used one sonar module.) We found that our initial setup was too heavy for the quadcopter to carry, so we pared down the elements on the platform, removing heavy wires and metal connectors, until we reached an allowable weight. The foamcore board rested on top of four rectangular supports that were attached to the side of the drone body with glue.

Another element of our setup was the foam shell that came with the drone and fit over the drone body to protect its propellers. We wrestled with whether or not to use the shell, because it added an additional 60g to our setup. However, initial testing proved that it was vital for the safe operation of the drone, so we made other weight adjustments to allow us to use the shell. A final notable element of our drone setup was the use of a string or wire tether. We used a tether to ensure that we could always maintain control of the location of the drone.

4.1.3 Weight payload testing

To determine the maximum weight payload of the drone, we attached a thin plastic cup (with negligible weight) to the top of the drone. We added small weights in increments of 14g to the cup, observed the drone's flight, and commented qualitatively on its ability to fly. Our data in Appendix B confirm that the maximum payload of the drone is about 113g = 60g foam shell + 53g weights.

4.1.4 Arduino and servo system

We used an Arduino Pro 328, which is a 3.3V, 8MHz microcontroller. We originally used an Arduino Uno board, but transitioned to using this Arduino Pro board, manufactured by Sparkfun, which is smaller and lighter than an Uno because it has no onboard USB connector. The Arduino microcontroller handles setting the position of the servo and operating the sonar ranging devices. Code for the Arduino can be seen in Appendix E and a schematic showing the wired connections between the Arduino and different hardware elements can be seen in Appendix A. The Arduino drove a generic "micro servo" that had approximately 150 degrees of rotation. The position of the servo could be set by writing the position of the servo (in microseconds) to

the servo using the Arduino's `writeMicroseconds(val)` command. In general, this command is only supposed to take values between 1000 and 2000, but through empirical testing, we found that our servo responded to values between 500 and 2100, where `writeMicroseconds(500)` set the servo to it's minimum position and `writeMicroseconds(2100)` set it to the maximum position. Our code was such that the servo changed position every 50 ms and the servo swept out 150 degrees in 15-degree increments; each sweep lasted approximately 750 ms.

4.1.5 Sonar ranging modules

The Arduino microcontroller also drove the Senscomp 6500 sonar ranging modules, which consisted of a board that abstracted the operation of the transducer and the transducer itself which sent and received the sonar "chirp". The sonar transducer was mounted on the servo so it could get data at a range of angles. The duty cycle of the sonars was 50 ms and a sonar reading was taken at every angle increment of the servo. The Arduino sent an "init" pulse to the board that lasted 20 ms, after which the init line was set low for 30 ms. After receiving this init pulse from the Arduino, the module board would create a ping on the transducer by sending 16 high-low transitions between -200 and +200V at 50kHz. The sonar "chirp" produced by this ping would travel outward at the speed of sound, and bounce off any objects in its path, creating an echo.

The Arduino handled the "echo" (or response) from the sonars with an interrupt. A handler function was associated with a rising external interrupt on the sonar echo line, so that this function was triggered every time an echo was received. When the echo was received, the interrupt routine would calculate the number of microseconds between the echo and the beginning of the init pulse and would print that delay value along with the associated angle where that reading was taken to the serial connection it had with the XBee in the format "angle:time". If multiple echo times were reported for a given servo angle, only the last one would be reported. This correction was added to account for the rare case when an errant fast echo would be reported before the real echo. This procedure can be seen in the code in Appendix E.

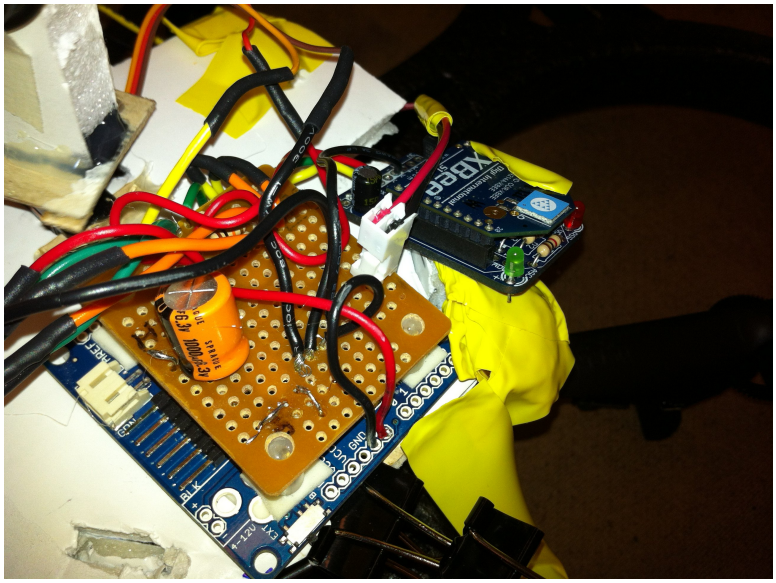


Figure 2: Closeup view of the connection between the Arduino Pro and the XBee device.

4.1.6 XBees and wireless communication

We used a pair of XBee Series 1 devices for wireless communication of the sonar data between the custom hardware mounted on the drone and the laptop. The Xbees use the IEEE 802.15.4 networking protocol to create a virtual serial link over a 2.4GHz wireless connection. They are abstracted to act as a basic wireless serial connection between the two devices. The XBees were configured with a baud rate of 9600 bps, with the 8N1 serial configuration: 8 data bits, no parity bit, and one stop bit. We did not need to include a parity bit for error connection because the XBees abstract a complicated wireless protocol that contains redundancy checking and error correcting. One XBee module was connected via wires to the Tx and Rx pins on the Arduino Pro, as seen in Figure 2, and the other was connected via FTDI cable to the USB port on the Linux laptop.

4.1.7 Power supply

During the primary stages of our project, we powered our entire sensing setup via the 11V battery that powered the quadcopter. We used a linear

regulator and a heat sink to isolate 5V to power the Arduino-servo-sonar system. However, we encountered some issues with this power supply setup because it would overheat and become erratic when the regulator got too hot. Attaching a heat sink mitigated some of these issues, but the system would still become overheated with long periods of use. Additionally, the battery was quite heavy (about 100g), and as the weight of our sensing platform became a primary consideration, we decided to switch to a different solution. Instead of using an on-board battery, we decided to use a DC power supply connected to the drone via a "tether" of four wires. We used two separate power supplies; one to supply 11V and 4A to the drone and another to supply 5V and .5A to the sensing platform. This ultimately proved to be a lighter and more flexible solution.

4.2 Software

On the software side, we wrote a program in C, `ardrone_testing_tool.c`, that integrated with the AR.Drone SDK in order to send appropriate movement commands to the drone. The program would take in sonar readings as input, process these according to our control algorithm, and output a movement command at regular intervals to be sent to the drone. The drone could be controlled by modulating the front-back angle ("pitch"), left-right angle ("roll"), vertical speed, and angular speed around a vertical axis (which changed the "yaw" angle). We were only concerned with movement in a single horizontal plane, so we only sent commands with values for pitch, roll and yaw. (N.B. We will refer to angular speed as the yaw value; we were never setting a particular yaw position, only rotating the drone towards a more favorable yaw position by changing its speed.)

4.2.1 Laptop and operating system

We determined that the AR.Drone SDK would be easiest to work with on a Linux system. Additionally, because we wanted to be able to leave a fully functional system for those who wished to build upon our work, we decided to purchase a dedicated machine for our project. We settled on a 15" Dell Inspiron laptop on which we installed the latest version of the Ubuntu Linux distribution.

4.2.2 AR.Drone SDK Version 1.8

Our software integrated tightly with the AR.Drone SDK, written in C, and provided by Parrot, the quadcopter manufacturer. We had initially planned to use Version 2.0, but after some initial testing, we found that the command we would be using most frequently to control the drone's movement did not work in this version because of a software bug. To account for this inconsistency, we downgraded to Version 1.8, which was still compatible with the 2.0 AR.Drone quadcopter.

To use the SDK, we made our own thread, called "mythread", in place of demo code that was available in one of the sample files. There is a boilerplate code from the SDK that links our procedures to the drone's other functionality. We wrote code to register and execute the instructions in our thread in parallel with other communication threads in the SDK. All of our code, available in Appendix D, is contained within "mythread", the `ardrone_turn_tool` utility function we wrote, and the `ardrone_tool_init_custom` function we modified to start our thread.

We relied primarily on two SDK functions. The first, `ardrone_tool_set_ui_pad_start`, would make the drone take off if called with 1 as its argument, and land if called with 0. The other was `ardrone_at_set_progress_cmd`, which took five arguments: a flags argument, roll value, pitch value, vertical speed, and angular speed. For our purposes, the flags argument was always set to 1 to enable commands. Our control algorithms dealt primarily with roll, pitch, and yaw. Each of these arguments took a floating point value between 1.0 and -1.0, representing the percent and direction of the maximum value for that parameter that would be sent to the drone. For example, if 0.5 was passed in for the roll value, the roll angle would be set to 50% of the system's maximum roll angle, which is a parameter defined by the SDK. In order to make these parameters more legible, we defined these argument values in terms of more legible terms, as seen in Figure 3.

```
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 3: Giving directional names to argument values.

4.3 Control Algorithm

At a high level, our algorithm attempts to send commands to the drone that change the pitch, roll and yaw values and will allow it to maintain a relatively constant distance from a wall. For pitch, we consistently alternated the polarity of the value we sent so that the drone would move backwards and forwards. This motion was simply to provide baseline motion so that the drone would traverse the wall. Beyond that, we tried to maintain an optimal distance between the wall and the drone by modulating the roll value, and an optimal angle relative to the wall by modulating the yaw value.

In order to calibrate our control algorithm, we tested each movement variable separately. First, we isolated the roll motion of the drone, only sending the drone commands that contained a non-zero value for the roll angle. We took videos of the drone's movements, matched those motions up to the output of our program, a sample of which can be seen in Appendix C, and then modified our control parameters accordingly. We repeated this process for the yaw variable until we were satisfied with movements related to that variable. Then, we combined the roll and yaw movements, continuing to make slight adjustments based on how our results compared to our expected outcome. Finally, we integrated all three variables (yaw, roll and pitch) for our final control algorithm.

4.3.1 Processing Sonar Readings

In order to implement this algorithm, we first needed to extract an estimation of distance to the wall, and angle relative to the wall, as these would be the primary data inputs to our algorithm. We used boilerplate code from an online source to set up and read from a serial port, so that we could get data

from the XBee's virtualized serial port into our C program. With this initialization code, we were able to give the serial port a file descriptor, which we could open and read from. We read 100 characters at a time, putting the received values, assumed to correspond proportionally to distance, in an array indexed by angle. That is, the reading for angle 0 was at index 0, the reading for angle 15 was at index 1, and so on.

After inputting values from a full 150 degree scan, the data processing would take place. The distance to the wall was simply assumed to be the minimum reading found among an entire scan. We used this metric because it was simple to implement and conservative, which was beneficial to our cause of obstacle avoidance. Finding the angle relative to the wall was a slightly more complicated process. Because the sonar transducer will receive readings from any perpendicular surface, instead of the point that is directly in front of it, we saw an "arc" pattern of mostly constant readings which indicated the range of angles that picked up an echo from the closest point on the wall. We identified this arc by finding the minimum reading, and then all adjacent readings that were within a tolerance band of 1000 microseconds. To find the index of the supposed true closest distance, we took the midpoint of the ends of the arc and determined the index of that point. As an additional smoothing measure, we incremented the indices of the arc if the drone took measurements from 150 degrees to 0 degrees ("scanning down"), and decremented the indices in the opposite case ("scanning up"). This correction was meant to correct inconsistencies in arc detection between the SCAN_UP and the SCAN_DOWN data.

```
1 SCAN RESULTS:
2 BACK ranges: [ 61  58  52  47  45  45  47  47  53  63  70  54  63 ] FRONT
3 BACK arc:    [             ************MMMM************           ] FRONT
4 arc_start: 3, arc_end: 9, arc_mid: 6
```

Figure 4: Example of diagnostic program output.

The diagnostic output (formatted to fit here) in Figure 4 illustrates this processing technique. The numbers represent the echo delays or distances, with their last two digits truncated to smooth our data and ease processing. In this case, the number 47 indicates an echo delay between 4700 and 4800 microseconds. Here, the minimum reading was 45, at indices 4 and 5, but the midpoint was calculated to be at index 6. By the logic of our algorithm,

the angle corresponding to index 6 is taken to be perpendicular to the wall.

4.3.2 Determining the correct roll value

The basic idea for this segment of the control algorithm was that if the drone was too close to the wall, it would adjust roll so that it moved away, and vice versa. This control function was made more complex by its dependence on two pieces of data: the current distance from the wall, and the distance from the wall in the previous scan. First, we empirically determined a threshold band that surrounds the "ideal distance" away from the wall. The high and low thresholds, defined within our program, can be seen in Figure 5. With these values, the ideal distance from the wall is 50, which corresponds to an echo delay of 5000 to 5100 microseconds.

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

Figure 5: Defining high and low thresholds for distance from the wall.

When the drone was within this threshold band, the drone's speed away from or towards the roll would scale linearly with its distance from the wall. Outside of this band, though, the drone's roll speed was limited by a maximum value, ROLL_MAX. This control rule is graphically represented in Figure 6. To implement this rule, the program would calculate a roll coefficient that essentially corresponds to the slope in Figure 6. It would multiply the differential between the current distance and the ideal distance by this coefficient, limiting the roll value by $\pm$ROLL_MAX at the high and low thresholds.

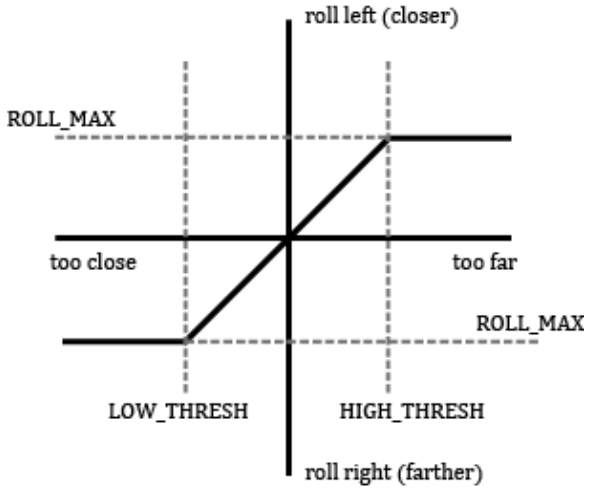


Figure 6: Roll angle control.

The second part of this control algorithm depended on the speed of the drone. Our system would correct the roll value based on the difference between the previous distance reading and the current one. The algorithm would multiply this differential by a correction coefficient, and add (or subtract) a correction, up to ±ROLL_CORRECT_MAX, from the roll value. For example, if the drone were moving very quickly towards the wall, this part of our algorithm would make the roll value larger so that it moved away from the wall with greater urgency. The correction calculation code can be seen in Figure 7.

```
424         //compute correction based on difference between current min and
                previous min
425         roll_correct = (old_min - min) * roll_correct_coef;
426         if(roll_correct > (ROLL_CORRECT_MAX)) roll_correct =
                ROLL_CORRECT_MAX;
427         if(roll_correct < (-1 * ROLL_CORRECT_MAX)) roll_correct = -1 *
                ROLL_CORRECT_MAX;
428         fprintf(stderr, "roll correction: %f\n", roll_correct);
429         roll += roll_correct;
```

Figure 7: Roll correction code.

4.3.3 Determining the correct yaw value

The high level idea for this segment of the algorithm was that was wanted to keep the drone's angle relative to the wall within a range of acceptable angles. We empirically determined the start- and end-points of this range, or "box", to be the indices 5 and 9. If the midpoint index of the arc was higher than that upper end of that range, it meant that front of the drone was pointing away from the wall. The higher indices corresponded to the back of the drone, so if the arc of close readings was centered on an index towards the back of the drone, it meant the back of the drone was close to the wall and the front was pointing away. Likewise, if the midpoint index was smaller than the lower end of the range, it meant the front of the drone was pointing towards the wall.

Based on this metric, the yaw value was either set to CLOCKWISE or COUNTERCLOCKWISE, which, multiplied by YAW_VAL in `ardrone_turn_tool`, would set the direction and magnitude of yaw correction for that scan. Within the "box" of acceptable angles was a yaw deadband; that is, no yaw commands were given within that range. This deadband was implemented to avoid jerky behavior that could results if the drone was sent consecutive, conflicting commands. Outside of the "box", only a single magnitude was given for the yaw value. It's direction would be set by the yaw direction, which was either clockwise or counterclockwise. A visual representation of this control rule can be seen in Figure 8.

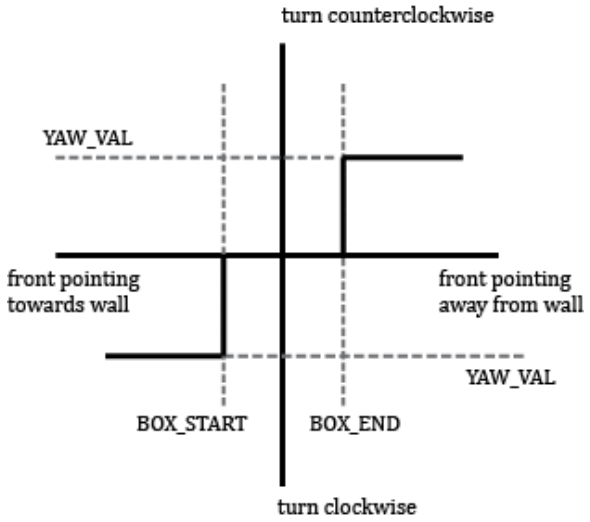


Figure 8: Yaw angle control.

5 Results

After much careful work and extensive calibration, these components were integrated to create a drone system that could successfully follow a wall with certain reliability. We were successfully able to get the drone to stay within 0.5 to 2m of a wall on 75% of our trials. A compilation video of some of our most successful trials can be seen in the accompanying folder and an image of our final assembled system can be seen in Figure 9.

Figure 9: The AR.Drone quadcopter with the sonar sensing platform mounted on top.

5.1 Values

The following control parameters were experimentally determined and were used in the final iteration of our quadcopter sensing system. These values, along with other calibration parameters, can also be found in our code in Appendix D.

- Drone payload weight: 113g = 60g foam shell + 53g weights
- Maximum angular speed or yaw value: 0.25
- Maximum forward or backward angle (absolute value): 0.06
- Maximum roll angle (absolute value): 0.08

5.2 Challenges and Limitations

We encountered a number of unexpected challenges throughout the course of this project. While we were able to adjust our system to account for some of these issues, others were persistent difficulties that ultimately limited the

capabilities and accuracy of our system. One limitation, for example, was that we only used one sonar transducer instead of two. This aspect of the system meant that we only got sonar readings for about half of the drone's surroundings, which limited the scope of the drone's avoidance capabilities. Because our control algorithms were crafted around this sensor setup, the current state of our project is dependent on the specific environmental setup we have. Another challenge we faced was getting the correct power to the drone and the sensing system. We went through several iterations of solutions – adding a capacitor between power and ground, adding a heat sink – before settling on using a tethered DC power supply as a solution that would deliver consistent power to our system.

Finally, the most restrictive challenge we encountered was the instability of the drone. While the drone sometimes exhibited unstable or unreliable flight when flying independently, this behavior was exaggerated by the addition of a sensing system mounted on top of the drone. The sensing system was constantly evolving, which made it extremely difficult to completely balance the drone and maintain the original center of mass. These balance issues strained the drone's built-in stabilization capabilities, and ultimately made it difficult for us to get reliable results with our control algorithms. We believe that a more stable system would have responded more dependably to our algorithms and would have displayed more consistently accurate behavior.

6 Conclusion

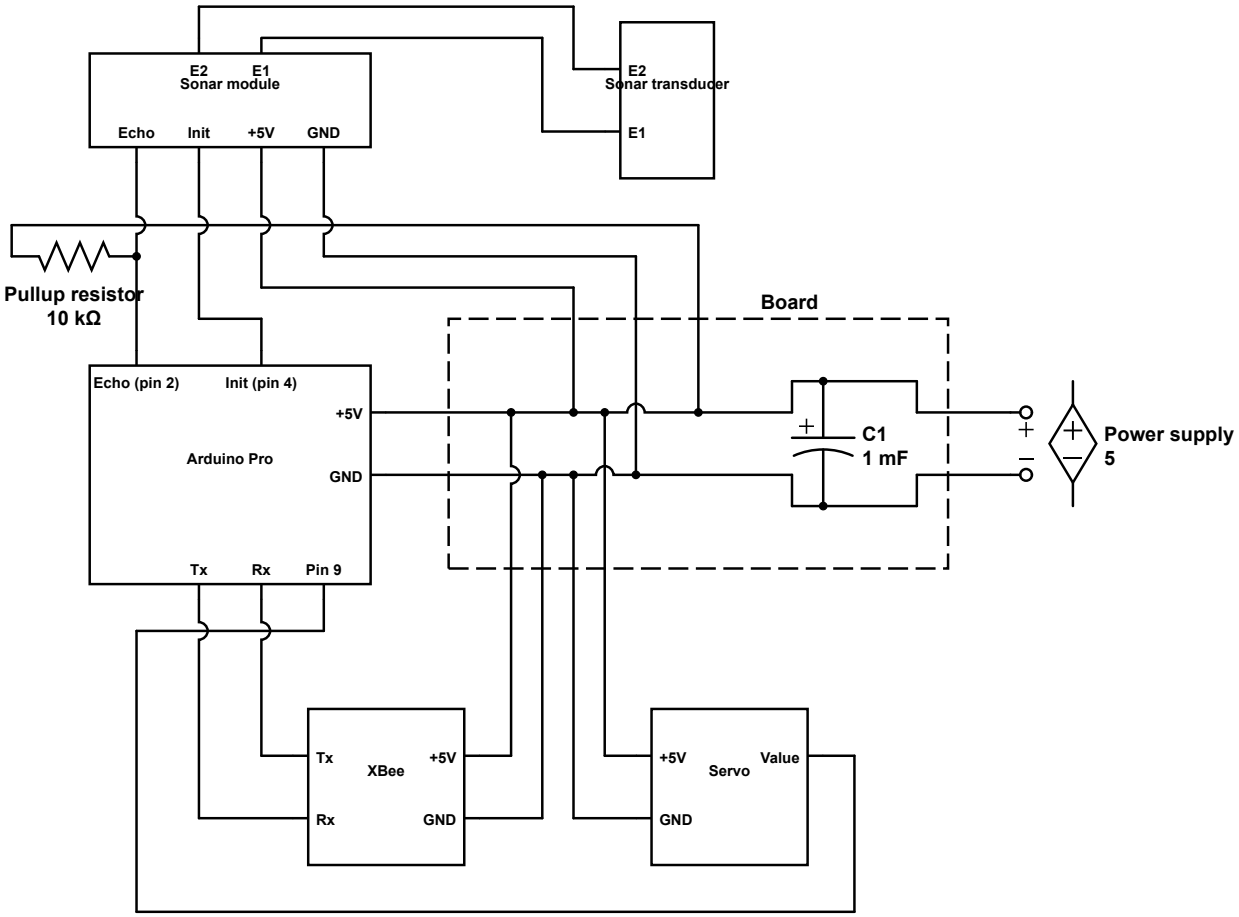
The purpose of this project was to develop a custom sonar ranging system for a quadcopter so that it could follow and avoid a wall. This project required the integration of many different disciplines and techniques, including hardware for the drone, sonars, and circuits and software for the drone's control algorithms. We used the built-in movement commands provided by the AR.Drone quadcopter SDK to give directions to the drone based on a control algorithm applied to readings received from the sonar sensing modules mounted on top. We used two XBee Series 1 devices to wirelessly communicate these readings between the flying quadcopter and the Linux system running our control algorithms.

Our calibrations and testing produced a set of parameters for our control algorithm that allowed the drone to successfully follow and avoid a wall, staying

0.5 to 2m away, about 75% of the time. The main limitation we faced was the instability of the drone due to the unbalanced addition of our sensing system and the inherent instability of the drone. We believe that our control algorithms would have run more smoothly and consistently if the drone had been more stable. Nonetheless, by introducing the drone and implementing some basic communication and control functionalities, this undertaking laid the groundwork for a variety of other useful and instructive projects that can involve the AR.Drone quadcopter.

Appendix

A Schematic



B Data from payload testing

Weight	Performance
0g	excellent performance
13.4g	excellent
27.2g	excellent
40.7g	excellent
53.8g	noticeably reduced but still very airworthy
67.5g	still flies, but more noticeably difficult for it
81.2g	not airworthy

C Sample output

```
SCAN RESULTS:
BACK ranges: [  61   58   52   47   45   45   46   47   53   64   70   54   63 ] FRONT
BACK arc:    [              **************MMMMM**************              ] FRONT
arc_start: 3, arc_end: 9, arc_mid: 6
GOOD ANGLE, DO NOT TURN
roll correction: -0.060000
scan_dir is SCAN_DOWN, INCREMENTED VALUES
FORWARD, RIGHT ROLL, NO YAW => pitch = -0.060000, roll = 0.010000, yaw = 0.000000

SCAN RESULTS:
BACK ranges: [  61   58   81   59   53   46   50   45   45   47   51   54   63 ] FRONT
BACK arc:    [              **************MMMMM******************          ] FRONT
arc_start: 3, arc_end: 10, arc_mid: 6
GOOD ANGLE, DO NOT TURN
roll correction: 0.000000
scan_dir is SCAN_UP, DECREMENTED VALUES
FORWARD, RIGHT ROLL, NO YAW => pitch = -0.060000, roll = 0.070000, yaw = 0.000000

SCAN RESULTS:
BACK ranges: [  61   58   52   47   45   45   47   47   53   63   70   54   63 ] FRONT
BACK arc:    [              **************MMMMM**************              ] FRONT
arc_start: 3, arc_end: 9, arc_mid: 6
GOOD ANGLE, DO NOT TURN
roll correction: 0.000000
scan_dir is SCAN_DOWN, INCREMENTED VALUES
FORWARD, RIGHT ROLL, NO YAW => pitch = -0.060000, roll = 0.070000, yaw = 0.000000

SCAN RESULTS:
BACK ranges: [  61   58   80   59   52   45   44   44   44   47   49   52   62 ] FRONT
BACK arc:    [              **************MMMMM******************          ] FRONT
arc_start: 3, arc_end: 10, arc_mid: 6
GOOD ANGLE, DO NOT TURN
roll correction: 0.003000
scan_dir is SCAN_UP, DECREMENTED VALUES
FORWARD, RIGHT ROLL, NO YAW => pitch = -0.060000, roll = 0.083000, yaw = 0.000000

SCAN RESULTS:
BACK ranges: [  57   53   48   42   42   42   43   51   60   68   69   52   62 ] FRONT
BACK arc:    [              **********MMMMM**************                  ] FRONT
arc_start: 3, arc_end: 8, arc_mid: 5
GOOD ANGLE, DO NOT TURN
roll correction: 0.006000
scan_dir is SCAN_DOWN, INCREMENTED VALUES
```

```
FORWARD, RIGHT ROLL, NO YAW => pitch = -0.060000, roll = 0.086000, yaw = 0.000000

SCAN RESULTS:
BACK ranges: [  57   53   79   54   47   43   44   43   43   44   46   48   54 ] FRONT
BACK arc:    [              **************MMMMM*******************          ] FRONT
arc_start: 3, arc_end: 10, arc_mid: 6
GOOD ANGLE, DO NOT TURN
roll correction: -0.003000
scan_dir is SCAN_UP, DECREMENTED VALUES
FORWARD, RIGHT ROLL, NO YAW => pitch = -0.060000, roll = 0.077000, yaw = 0.000000

SCAN RESULTS:
BACK ranges: [  69   59   56   50   48   47   49   49   55   76   82   48   54 ] FRONT
BACK arc:    [              **************MMMMM***************              ] FRONT
arc_start: 3, arc_end: 9, arc_mid: 6
GOOD ANGLE, DO NOT TURN
roll correction: -0.012000
scan_dir is SCAN_DOWN, INCREMENTED VALUES
FORWARD, RIGHT ROLL, NO YAW => pitch = -0.060000, roll = 0.038000, yaw = 0.000000

SCAN RESULTS:
BACK ranges: [  69   59   92   71   62   56   56   56   57   59   64   72   82 ] FRONT
BACK arc:    [              **************MMMMM***************              ] FRONT
arc_start: 3, arc_end: 9, arc_mid: 6
GOOD ANGLE, DO NOT TURN
roll correction: -0.027000
scan_dir is SCAN_UP, DECREMENTED VALUES
FORWARD, LEFT ROLL, NO YAW => pitch = -0.060000, roll = -0.067000, yaw = 0.000000

SCAN RESULTS:
BACK ranges: [  96   84   76   70   66   65   65   66   74   79  100   72   82 ] FRONT
BACK arc:    [                  **********MMMMM***************              ] FRONT
arc_start: 4, arc_end: 9, arc_mid: 6
GOOD ANGLE, DO NOT TURN
roll correction: -0.027000
scan_dir is SCAN_DOWN, INCREMENTED VALUES
FORWARD, LEFT ROLL, NO YAW => pitch = -0.060000, roll = -0.107000, yaw = 0.000000

SCAN RESULTS:
BACK ranges: [  96   84  170   98   80   71   71   70   70   70   74   84   85 ] FRONT
BACK arc:    [                  **********MMMMM***************              ] FRONT
arc_start: 4, arc_end: 9, arc_mid: 6
GOOD ANGLE, DO NOT TURN
roll correction: -0.015000
scan_dir is SCAN_UP, DECREMENTED VALUES
```

```
FORWARD, LEFT ROLL, NO YAW => pitch = -0.060000, roll = -0.095000, yaw = 0.000000
flight counter overflow, switching directions to BACKWARD

SCAN RESULTS:
BACK ranges: [  91   78   66   66   66   67   68   71   80   86   91   84   85 ] FRONT
BACK arc:    [              **********MMMMM**************                    ] FRONT
arc_start: 3, arc_end: 8, arc_mid: 5
GOOD ANGLE, DO NOT TURN
roll correction: 0.012000
scan_dir is SCAN_DOWN, INCREMENTED VALUES
BACKWARD, LEFT ROLL, NO YAW => pitch = 0.060000, roll = -0.068000, yaw = 0.000000

SCAN RESULTS:
BACK ranges: [  91   78  111   85   71   62   62   59   59   59   61   63   69 ] FRONT
BACK arc:    [                   **************MMMMM**************          ] FRONT
arc_start: 4, arc_end: 10, arc_mid: 7
GOOD ANGLE, DO NOT TURN
roll correction: 0.021000
scan_dir is SCAN_UP, DECREMENTED VALUES
BACKWARD, LEFT ROLL, NO YAW => pitch = 0.060000, roll = -0.049000, yaw = 0.000000

SCAN RESULTS:
BACK ranges: [  68   52   48   48   48   49   52   53   59   75   85   63   69 ] FRONT
BACK arc:    [          **************MMMMM**************                    ] FRONT
arc_start: 2, arc_end: 8, arc_mid: 5
GOOD ANGLE, DO NOT TURN
roll correction: 0.033000
scan_dir is SCAN_DOWN, INCREMENTED VALUES
BACKWARD, RIGHT ROLL, NO YAW => pitch = 0.060000, roll = 0.073000, yaw = 0.000000

SCAN RESULTS:
BACK ranges: [  68   52   79   58   47   41   39   38   38   40   40   47   51 ] FRONT
BACK arc:    [              **************MMMMM********************          ] FRONT
arc_start: 3, arc_end: 10, arc_mid: 6
GOOD ANGLE, DO NOT TURN
roll correction: 0.030000
scan_dir is SCAN_UP, DECREMENTED VALUES
BACKWARD, RIGHT ROLL, NO YAW => pitch = 0.060000, roll = 0.110000, yaw = 0.000000
```

D Drone code: ardrone_testing_tool.c

```
/**
 * 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
307         //(if a reading is missed, the previously scanned value remains)
308
309         //process the scan data if we reach either endpoint of the scan
310         if(angle == 0){
311           scan_dir = SCAN_DOWN;
312           break;
313         }
314         if (angle == 180){
315           scan_dir = SCAN_UP;
316           break;
317         }
318       }
319     }
320
321     arc_start = arc_end = arc_mid = 0;
322     min = INT_MAX;
323
```

```
324     min_ind = 0; //useless init value to stop compiler warnings, should be
            overwritten always
325
326     //Find min
327     for (i = 0; i < ARRAY_LEN; i++) {
328       if (arr[i] < min) {
329         min = arr[i];
330         min_ind = i;
331       }
332     }
333
334     if(old_min == -1) old_min = min; //init old_min on first scan
335
336     //Find the arc start- and end- points
337     arc_start = arc_end = min_ind; //in case the arc is one wide
338     i = min_ind-1;
339     while (i >= 0 && abs(arr[i]-min) < ARC_BAND) {
340       arc_start = i;
341       i--;
342     }
343     i = min_ind+1;
344     while (i < ARRAY_LEN && abs(arr[i]-min) < ARC_BAND) {
345       arc_end = i;
346       i++;
347     }
348     arc_mid = (arc_start+arc_end)/2; //index which is the center of the arc
349
350     // adjust arc readings based on scan direction
351     // this compensates for inaccuracies in scan data depending on scan
            direction
352     if (scan_dir == SCAN_DOWN && arc_mid < ARRAY_LEN){
353       arc_start++;
354       arc_end++;
355       arc_mid++;
356     }
357     else if (scan_dir == SCAN_UP && arc_mid > 0){
358       arc_start--;
359       arc_end--;
360       arc_mid--;
361     }
362
363     //print sonar readings to terminal
364     fprintf(stderr, "BACK ranges: [");
365     for(i = 0; i < 13; i++){
366       fprintf(stderr, "%4d ", arr[i]);
367     }
368     fprintf(stderr, "] FRONT\n");
369
370     //print arc calculation readings to terminal
371     fprintf(stderr, "BACK arc:    [");
372     for(i = 0; i < 13; i++){
373       if(i == arc_mid){
374         fprintf(stderr, "MMMMM");
375       }
376       else if(i >= arc_start && i <= arc_end){ //in arc
377         fprintf(stderr, "*****");
378       }
```

```
379       else{
380         fprintf(stderr, "     ");
381       }
382     }
383     fprintf(stderr, "] FRONT\n");
384     fprintf(stderr, "arc_start: %d, arc_end: %d, arc_mid: %d\n", arc_start,
            arc_end, arc_mid);
385
386     //ignore the first couple scans, which may not contain full data
387     total_scan_count++;
388     if(total_scan_count <= SCANS_BEFORE_START){
389       fprintf(stderr, "DISCARDING THIS INITIAL SCAN DATA\n");
390       continue;
391     }
392
393
394     if(arc_start < arc_end){ //only send a command if arc more than 1 wide,
            prevents noise
395
396       //control based on yaw angle
397       if (arc_mid > BOX_END) { //front of drone pointed away from wall
398         fprintf(stderr, "OUTTA THE BOX YO, front pointed TOWARDS WALL!\n");
399         yaw = CLOCKWISE;
400       }
401       else if (arc_mid < BOX_START) { //front of drone pointed towards wall
402         fprintf(stderr, "OUTTA THE BOX YO, front pointed AWAY FROM WALL!\n")
              ;
403         yaw = COUNTERCLOCKWISE;
404       }
405       else {
406         fprintf(stderr, "GOOD ANGLE, DO NOT TURN\n");
407         yaw = NO_TURN;
408       }
409
410       //control based on distance from wall
411       roll = (ideal_distance - min) * roll_coefficient; //ideal_dist - min
              gives a negative #, appropriately
412
413       //limit roll within roll_max
414       if(roll < (-1 * ROLL_MAX)){
415         roll = -1 * ROLL_MAX;
416       }
417       else if (roll > ROLL_MAX){
418         roll = ROLL_MAX;
419       }
420
421       if(old_min != -1){ //make sure it's initialized
422         //add roll correction to roll value
423
424         //compute correction based on difference between current min and
              previous min
425         roll_correct = (old_min - min) * roll_correct_coef;
426         if(roll_correct > (ROLL_CORRECT_MAX)) roll_correct =
              ROLL_CORRECT_MAX;
427         if(roll_correct < (-1 * ROLL_CORRECT_MAX)) roll_correct = -1 *
              ROLL_CORRECT_MAX;
428         fprintf(stderr, "roll correction: %f\n", roll_correct);
```

```
429         roll += roll_correct;
430
431       }
432     }
433     else{
434       //if the arc is only one wide, don't send any command
435       fprintf(stderr, "NO SUFFICIENTLY WIDE ARC FOUND. DO NOT TURN\n");
436       yaw = NO_TURN;
437     }
438
439     if (scan_dir == SCAN_DOWN){
440       fprintf(stderr, "scan_dir is SCAN_DOWN, INCREMENTED VALUES\n");
441     }
442     else if (scan_dir == SCAN_UP){
443       fprintf(stderr, "scan_dir is SCAN_UP, DECREMENTED VALUES\n");
444     }
445
446     //SEND the movement commands that were calculated
447     ardrone_turn_tool(dir, yaw, roll, FLY); //will only turn if FLY is
          enabled
448
449     //switch directions after the specified number of scans
450     scan_counter++;
451     if (scan_counter >= SCANS_BEFORE_SWITCH) {
452       scan_counter = 0; //reset scan counter
453       switch_counter++; //keep track of direction switch count
454       dir *= -1; //switch directions
455       fprintf(stderr,  "flight counter overflow, switching directions to ");
456       if(dir == FORWARD){
457         fprintf(stderr, "FORWARD\n");
458       }
459       else{
460         fprintf(stderr, "BACKWARD\n");
461       }
462
463       if(switch_counter >= SWITCHES_BEFORE_LAND){
464         //land the drone, end of flight
465         fprintf(stderr, "switch counter overflow, landing...\n");
466         break; //exit the while 1 loop
467       }
468     }
469
470     old_min = min;
471   }
472
473   //always land before completing the program! we don't want a stranded
        drone.....
474   if(FLY){
475     ardrone_tool_set_ui_pad_start(0);
476   }
477
478   close(fd);
479   THREAD_RETURN(0);
480 }
481
482 //============================
483 //BOILERPLATE AR_DRONE API CODE
```

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

E Arduino code: arduino.ino

```
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  }
```