Autonomous Unmanned Aerial Vehicle

CS492: Project2

Report

Prepared for

Dr. Furqan Muhammad Khan

Assistant Professor

National University of Computer and Emerging Sciences

Dr. Syed Tahir Qasim

Assistant Professor

National University of Computer and Emerging Sciences

Submitted By

Mahak Mohbat Ali (k11-2197)

Osman Ali Mian (k11-2123)

Hira Farman (k11-2205)

Mohsin Mansoor Kerai (k11-2219)

Students

National University of Computer and Emerging Sciences

Date

8th May 2015

Contents

Abstract

Unmanned Aerial Vehicles are nowadays routinely used in several applications where human interaction is difficult or dangerous. The project develops an autonomous unmanned aerial vehicle that is capable of flying, using visual navigation and an additionally equipped sonar sensor, without hitting into objects, identify potential hideouts and park itself there. This document explain the methodology to create such a vehicle, challenges, a proposed solution and future aspects. Open Source Computer Vision Library has been used for object identification; integrated together with Robot Operating System along with Raspberry PI mounted Sonar Sensor. This report paper discusses the Autonomous Unmanned Aerial Vehicle, A Final Year Project at National University of Computer and Emerging Sciences. It provides detailed overview of software and hardware, flight testing, experiments, estimation of hardware and software and future related work.

Introduction

Unmanned aircraft are especially useful in penetrating areas that may be too dangerous for manned aircraft. Military aspect is the most common application being used today through which aerial surveillance and drone attacks are taking place. Beyond the military applications of UAVs, numerous civil aviation uses have been developed. These include aerial surveying of crops over a wide area, search and rescue operations in natural disasters, network attacks, delivering medical supplies to inaccessible regions. The unmanned aircrafts are paving the way to a new future. [1]

The purpose of this project is to make AR. drone 2.0 flying autonomously, using visual navigation and sonar sensor. It should not hit any object coming on its way and reach a target point. The objective behind making AR. drone autonomous is to provide a platform for future developers to implement emergency navigation applications so that the drone can be used in areas where humans are unable to reach. Drone can be a life saver in search and rescue mission. For Instance, after an earthquake or any disaster, conditions of areas are difficult to search using human exploration, but it is easy for Autonomous Unmanned Aerial Vehicle to reach there, scan the targeted place, take images and videos of the place and tell us the internal situation of an area. This can be useful for the rescue team to decide that the area is safe to go or not. Other applications of UAV's include Map Making, Architecture Designing, Area Planning, Precision Agriculture and Bridge Inspection.

Autonomous drones can come in handy in navigating a damaged structure especially in search and rescue mission. An autonomous drone capable of finding a safe spot and parking it there could provide attack and surveillance opportunities including temporary security reinforcements by acting as portable surveillance cameras. With a limited capacity of carrying weight, drone can also be used as mobile and temporary lightening source at a given occasion. However these drones can't carry the heavy, power-hungry sensors that larger aircraft use to detect other planes.

Different theories exist to make drone autonomous for example using Arduino as a core controller, and there is a self-assembly kit that can get it started. We made parrot Ar. Drone autonomous using additional components including Robot Operating System (ROS), Raspberry Pi and Sonar sensor. This project use a different approach in making the drone autonomous by not just using the native sensors provided on the drone, but by building another external system that can detect obstacles, thereby simplifying the overall programming complexity. The two different systems running in parallel need to exchange their data only as they continue to work in their own respective capacities. Additional use of a built in trained model means that we are able to detect if a picture contains a safe spot "on the fly". Our approach is based on not making an overall complex system, but a communication of simple sub-systems, which work in their own capacity and keep exchanging data from each other, without knowing the underlying work of the other sub-system.

Related Work:

The idea of a pilotless aircraft is old concept it dates back to the mid-1800s, when Austrians sent off unmanned, bomb-filled balloons as a way to attack Venice. The drone seen today started innovation in the early 1900s, and was originally used for target practice to train military personnel. It continued to be developed during World War I, when the Dayton-Wright Airplane Company came up with a pilotless aerial torpedo that would drop and explode at a particular, preset time. [2]

A Hungarian team created the first drones that can fly as a coordinated flock. The researchers watched as the ten autonomous robots took to the air in a field outside Budapest, zipping through the open sky, flying in formation or even following a leader, all without any central control. Truly autonomous drone was created in 2011 by robotics researcher Dario Floreano at the Swiss Federal Institute of Technology in Lausanne. These machines were fixed-wing fliers that could move only at constant speeds and had to fly at different heights to avoid collisions

Sanjiv Singh, a professor and researcher at Carnegie Mellon University, has developed a new system to make drone autonomous. Most UAVs are fairly small and lightweight, they can't carry the heavy, power-hungry sensors that larger aircraft can use to detect other planes. So Singh and student Debadeepta Dey developed an algorithm that uses an ordinary camera and several software programs to detect potential obstacles. [3]

To navigate quad copter autonomously Jakob Engel, J¨urgen Sturm, Daniel Cremers used monocular SLAM system, an extended Kalman filter for data fusion and state estimation and a PID controller to generate steering commands. [4]

WITAS UAV deployed fully autonomous UAV over diverse geographical terrain that contains traffic and network. UAV can navigate autonomously, plan location, identify, track, monitor vehicles and identify complex patterns of vehicle overtaking, traversing of intersection and parking lot activities. [5]

Our project is heavily inspired by the presentations and researches of Technical University of Munich, Computer vision group [6]. They have previously worked on autonomous navigation of the same type of quad-copter that we intend to use for our project. Their researches are available online to assist us in proceeding and reaching our goal.

Project Requirements

Hardware Requirements:

- **Parrot AR-Drone 2.0**
  Radio controlled flying quad copter which is intended to be made autonomous. It has the ability to take photos and videos using IOS or Android device (already built apps). More Importantly there is an API available which can be used to program the drone, thereby making it able to carry out user defined roles, such as waypoint following, movie making and surveillance. Appendix A shows technical Specifications of AR-Drone.

- **Raspberry Pi**
  A small hand held processing board with a set of serial input pins. Raspberry Pi was needed to add an additional sensor to the drone to make it avoid obstacles. Raspberry Pi had to be configured with Robot Operating System to make it communicate with the main computer. Appendix B outlines the basic specifications of Raspberry Pi.

- **Sonar Sensor**
  A basic commonly used sonar sensor. This sensor was required to be mounted on Raspberry Pi. The final system comprising of the sensor and Raspberry Pi would then be mounted on the drone since the drone does not have an obstacle detection mechanism of its own. We used the HC SR 04 sonic sensor, details of which are available in Appendix C

Software Requirement:

- **Linux Operating System running Robot Operation System**
  The Robot Operating System (ROS) is a flexible framework for writing robot software. It is a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behavior across a wide variety of robotic platforms [4]. The main power of ROS comes from its concept of "Collaborative Robotics" where multiple robots (nodes) can be programmed to work as collaborative agents. Few of Its applications include. ROS follows a subscriber publisher mechanism, where multiple "nodes" can subscribe to a message being broadcasted by a certain "publisher". All the message passing goes through a central system, known as ROS_MASTER.

- **The Ardrone_autonomy open source driver**
  "ardrone_autonomy" is a ROS driver for Parrot AR-Drone. This driver is based on official AR-Drone SDK version 2.0 and supports both AR-Drone 1.0 and 2.0. This driver is required to connect with the drone. Since it is an open source driver it had a few noticeable shortcomings that we had to deal with during the course of this project. These were as follows,
    - Abnormal gyroscope/IMU readings while takeoff on random occasions

- Velocity along z-axis always measured zero
- Yaw angle of the drone keeps updating automatically even though the drone is parked on a stationary surface.

See Appendix D for further details.

- **Python / RosPy**

  Programming language used with ROS, Python has been used to program the commands and PD control that will be issued to drone. Python language was also used in writing the code for sonar sensor on Raspberry Pi

- **Open Source Computer Vision (Open CV)**

  Free open source library of programming functions used for object identification. It is written in C++ and runs on Linux, IOS, Windows and OS X.

Methodology

This project comprises of two main parts. One for making the drone fly autonomously which includes programming the PD control on the drone, this would be further complemented by writing an additional module involving Raspberry Pi with a mounted sonar sensor to prevent drone from colliding into obstacles. Second part comprises of Object recognition, which would focus on classifying things in our environment and identifying them on the run (These are doors and windows in our particular case).

The project can categorized into three modules:

- Start in a specific area and explore it, using one random co-ordinates initially (FYP1)
- Find a safe place by analyzing video feeds received from drone's camera and park itself there. (FYP2)
- Avoid obstacles while in transit from one point to the other (FYP1 and 2)

In order to achieve these goals, following steps were identified:

- **Setting up The Robot Operating System Environment**: This is the main interface to send and receive information from the drone. The ROS is available as fully documented and open source software on Linux Operating System

- **Use Raspberry Pi Equipped with a Depth Sensor:** Since drone does not have its own sonic sensor to detect distance along x axis, an additional system was required to help it detect obstacles. This system was mounted over the drone.

- **Process the Raw Image Data Being Received from Drone:** This was used in the SVM Classification part of this project. The raw image data was converted into bgr8 format of the OpenCV python library for further image processing and classification.

- **Integrate Drone Navigation with Classifier:** Eventually it was required that both navigational system as well as object detection systems be integrated together so that navigations decisions can be made using the feeds received from the drone's camera

Prototype Development

Once the basic tasks were identified to reach our final goal, we divided the overall project development into two sub-teams. These were

- **Visual Classification Team:** This team was required to build a machine learning system that would recognize potentially unsafe spots for the drone to land. We chose doors and windows as sample unsafe spots for our case

- **Navigation Team:** This team's main role was to implement algorithm that enabled the drone to move from 1 given point to another and take pictures of the destination points, these pictures would then be sent to the machine learning algorithm implemented by Visual Classification team for testing as a "safe spot". Additionally this teams responsibility was to introduce an obstacle detection mechanism in the drone

Overall System Diagram

The complete system built in our FYP-2 project is shown in thefigure1 given below. This final system was achieved only after going through various prototypes, which are discussed on following pages. A brief description of the final system is given as follows.

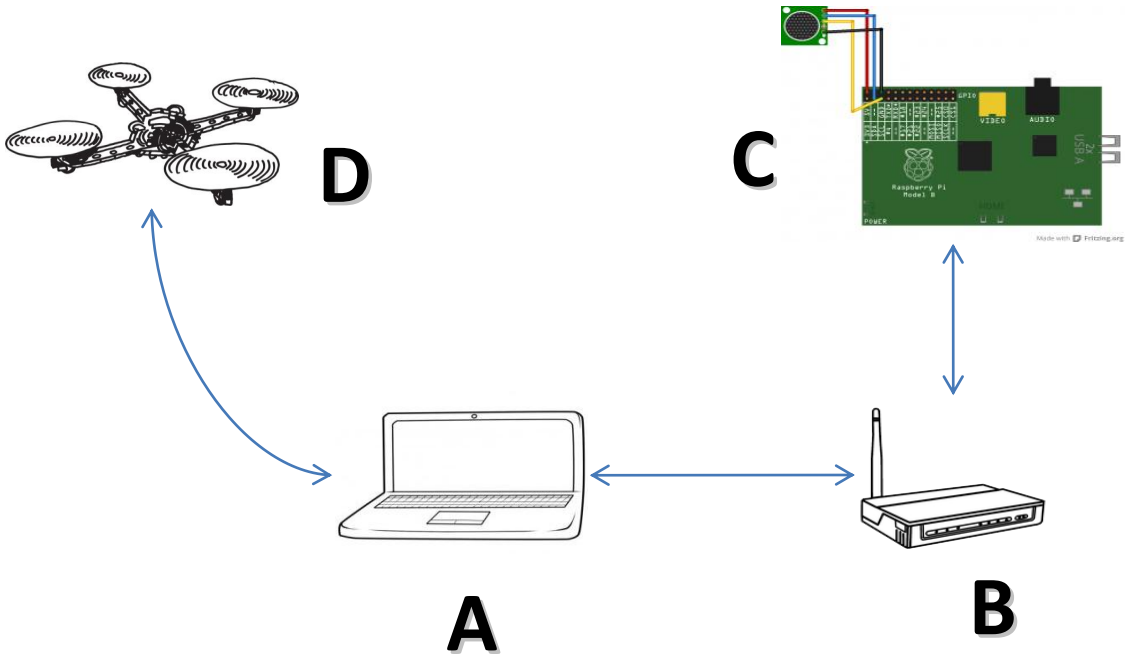


Figure1:Complete System diagram

Component Description

- **A:** The main computer/server. This computer has the original program that is designed to take input of readings from sonar sensor over Wi-Fi, take input of visual feeds from the drone, run the classifier over the visual feeds, and execute the next set of commands on the drone.

- **B:** The router that is used to connect to the Raspberry Pi which has the depth sensor mounted on it. The router communicates over the network 10.11.x.x. The main computer has an additional USB dongle attached to it to connect to two networks simultaneously.

- **C:** Raspberry Pi, with sonar sensor mounted. RPi has a low powered dongle in one of its USB ports. It is configured to automatically connect to the SSID of the router B. Once it is connected, an ssh connection is made from the main computer A via the router B and the proximity detection model is executed. Ideally, this node is supposed to be mounted right over the drone.

- **D:** The parrot AR drone, it sends visual feeds to the main computer, the main computer runs those feeds through the classifier to detect whether the drone is viewing a safe place or not. The main computer also issues set of movement commands to the drone. This drone is connected to main computer over the network 192.168.x.x.

Prototype Details

The following section discusses a number of prototypes that were made and eventually lead to the final design as shown in previous section. Some of the prototypes were useful to build upon, while some others were helpful in providing us an insight of how things are working within the drone. As discussed previously, the prototypes are also divided into two main categories, the navigation prototype and the visual classification prototypes.

Prototype 1: Creating Manually Controlled Drone

Team: Navigation
Duration: 2 Months

This prototype being just one part of the overall system diagram concentrates on creating a manual controller. The idea was to first getting a hands on experience with controlling the drone in some kind of manual way. This would enable us to view the perspective of the world with respect to the data being received by the drone and act accordingly when we finally decide to automate the system. Eventually we came up with a program that enabled us to control the drone using periodically issued commands from our main computer. A diagram of this prototype is shown in Figure2.

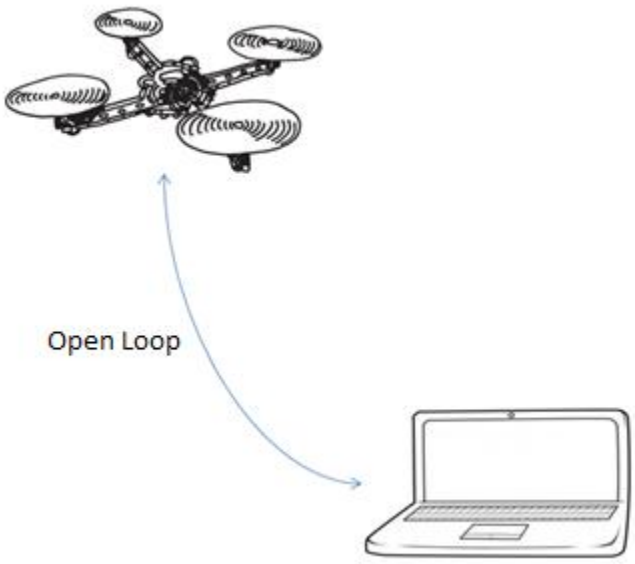


Figure 2: Prototype 1, partial system diagram

A manual controller is when the user specifies some movement commands to the main computer/ server which are forwarded to the drone. The drone responds based on the commands received from the main computer. In addition, the drone sends images back to the main computer. Based on these

images, the user can decide the future movements. This prototype came about after 3 experiments that are discussed as follows

Experiment 1a: Weight Constraints

This experiment was motivated by the need that the drone will require an additional raspberry Pi with a depth sensor to be mounted on it for obstacle detection reasons. Main purpose of this experiment was to get acquainted with how much weight the drone can carry so that the obstacle detection system can be made keeping in mind the weight limitations. 5 different test flights were made, the results are shown in table 1.0.

Weight	**Flight duration**	**Result**
10 Grams	5 minutes	Landed Successfully
50 Grams	5 minutes	Landed Successfully, Few jitters during flight.
90 Grams	2 minutes	Crashed. Flied up to 10 meters high
180 Grams	2 seconds	Crashed. Could not take off properly
180 Grams	1 minute	Weight was positioned differently. Crashed. Could not take off properly.

Table 1.0: Results of Weight Constraints Experiment on AR-Drone

Experiment 1b: Navigation Using Open Loop Commands

A predefined set of programs were written to test drone's movement in an outdoor location. These programs were mainly intended to make the drone trace a set of common shapes when seen from top. The outcome achieved from these experiments enabled us to judge the response drone gives on velocity commands and the variation in drone trajectory that may occur when we try to change its roll, pitch and yaw angles. The results of this experiment are shown in table 2.0.

Sr. No	Description	Type	Time (Seconds)	Result	Reason
1	Fly in A Straight Line	Outdoor	4	Crash	Wind Drift causing elongated movement
			3	Success	
2	Make A Square	Outdoor	10	Crash	Incorrect Code
			17	Crash	Wind Drift causing elongated movement
			40	Partial Success	Trajectory Effected by Strong Winds
			39	Partial Success	Trajectory Effected by Strong Winds
3	Make A Circle	Outdoor	5	Fail	Flight Duration Too Short
			10	Fail	Flight Duration Too Short
			15	Crash	Incorrect Code
			15	Partial Success	Robot Tends to lower itself when t turns simultaneously while moving

Table 2.0: Navigation commands with result and reason

Experiment 1c: Navigation Using Video Feed From Drone

This experiment was straightforward, a simple program was written to navigate the drone while viewing the world from drone's camera only. The pilot was not seeing the drone from the outside, rather was observing the world from the inside. Following observations were prominent during this experiment

- When the controlling device and the drone are not in line of sight, the delay between command and execution is around 2 seconds maximum.
- When the distance between controlling device and drone goes beyond 30 meters, the video feed faces lag.
- Below 20% battery charge, drone picture quality falls drastically.

Prototype 2: Drone with Autonomous Navigation Capabilities

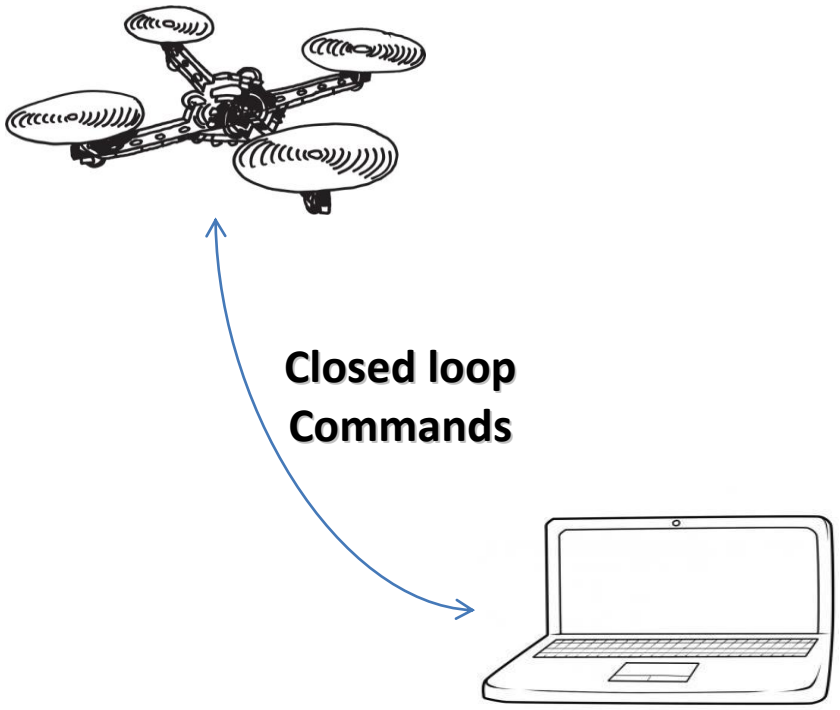


Figure 3: Prototype2 Diagram

Team: Navigation
Duration: 3 Months

This prototype constituted as main part of this project. A PD control was implemented on the drone to make it reach a desired co-ordinate. The open source driver for AR-Drone provides only a limited functionality for the drone's movement, a user can configure the speed with which drone travels but not the exact distance that the drone would cover, hence a mechanism was required to ensure that drone could navigate between waypoints. This mechanism would work by gradually decreasing speed of the drone based on drone's distance from the target location. This requirement was fulfilled using the PD control. A number of experiments were done to incorporate a working PD control in the drone. Since the open source API has its limitation while reading the drone's velocity along Z-axis, we had to try and create our own detection mechanism for it. This mechanism still remains in experimental phase. A detailed summary of the experiments for this prototype is as follows

Experiment 2a: The PD Controller

A proportional-integral-derivative controller (PID controller) is a control loop feedback mechanism (controller) widely used in industrial control systems. A PID controller calculates an *error* value as the difference between a measured process variable and a desired set point. The

controller attempts to minimize the *error* by adjusting the process through use of a manipulated variable. [5]

PID control (Figure 4) depends on measured process variable only. It does not use knowledge of process. [7] Controller of PID provide control action for specific process requirements. Response of controller is responsiveness of the controller to an error (degree to which the controller overshoots the set point and system oscillates). A PID controller can be called a PI, PD, P or I when respective control action is absent.

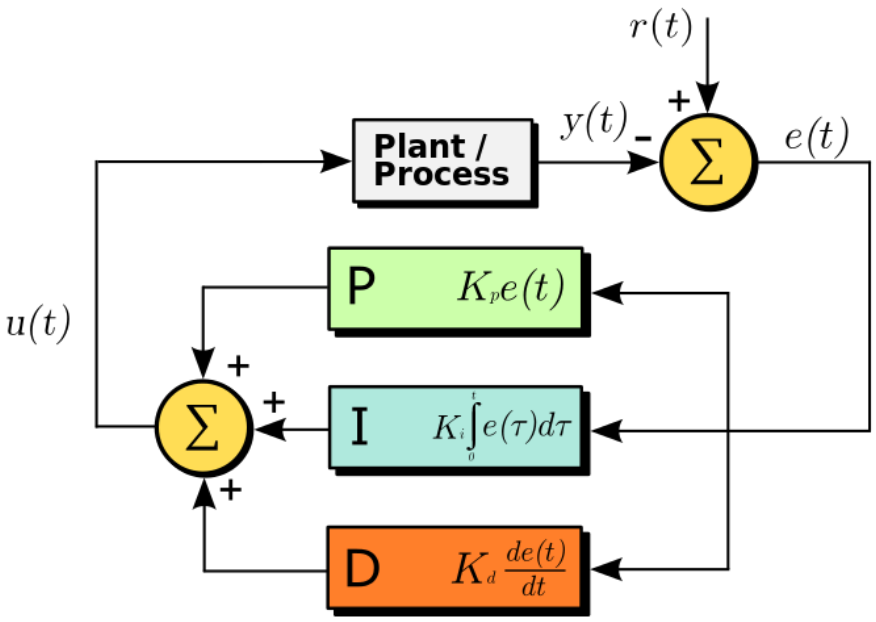


Figure 4: PID Control (Wikipedia)

For our experiment we decided to go only with the Proportional and Derivative part of the equation. Our proportional quantity was the distance moved by the drone, and our derivative part was the velocity. Our drone was repeatedly given instructions based on calculations from the following equation

next command = P (Current Position - Desired_Position)
+
D (Current_Velocity - Desired Velocity)

Where,

Current_Position: Present displacement of drone estimated from its take off point.
Desired _Position: The displacement currently set as target position for the drone
Current_Velocity: The instantaneous velocity of the drone at any given time *t*
Desired_Velocity: The velocity intended upon reaching the *Desired_Position*mark. This is almost always zero.

P: The proportional gain for the Proportional error. Similar to the driving force for a system. Setting this gain higher would imply more "driving force" for the drone. Setting it too high will cause over shooting in the system as shown in figure 5 and figure 6. The system goes way over its desired point (100cm) and returns back and goes below the mark and back up again several times before reaching the desired position.

D: The derivative gain, similar to causing a dampening effect in simple harmonic motion. The greater the value of this gain, the stronger "brakes" would be applied to the movement of the drone. Setting this parameter to a high value compared to the Proportion gain would result in system coming to the desired position at a very slow rate.

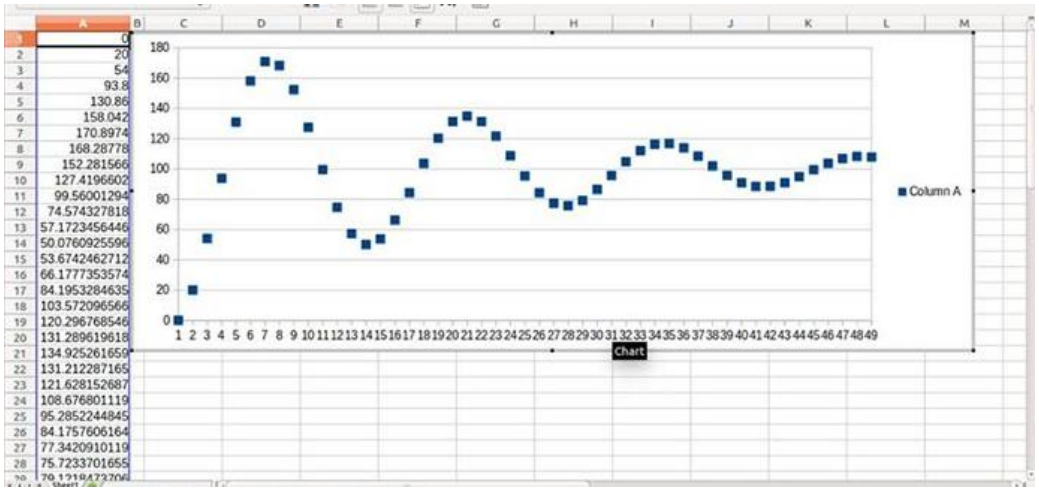


Figure 5: PD Control With Over Shooting

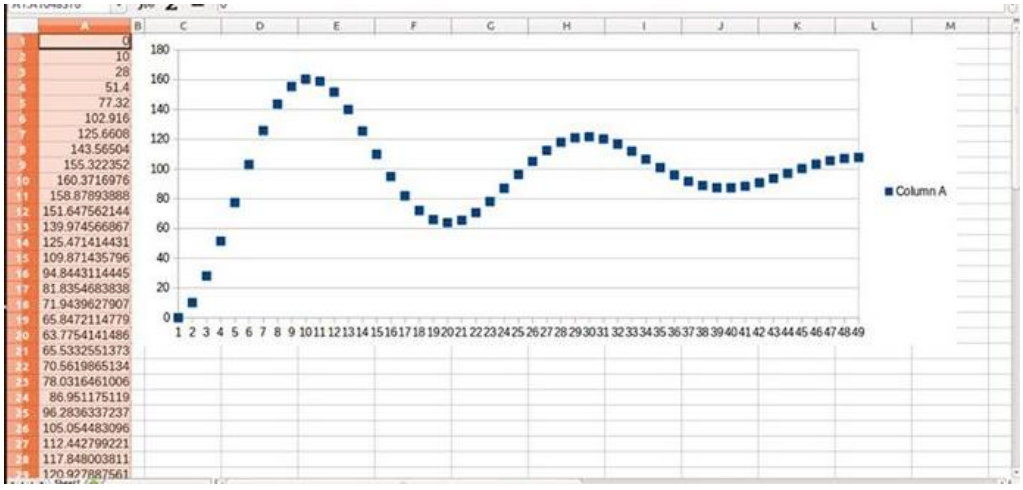


Figure 6: PD Control with Differential Term Enhanced. Systems comes to rest in almost same time, but lesser overshooting

When both the proportional gains and derivative gains are tuned in correct proportion, the system

a) Does not overshoot
b) Comes to rest in quickest possible time.

Figure 7 below shows the results of the experiment when PD control was configured in correct proportion. The system here comes to rest in almost 30 iterations of the PD control.

Further details and tables of these experiments are in the extra sheet provided with this report.

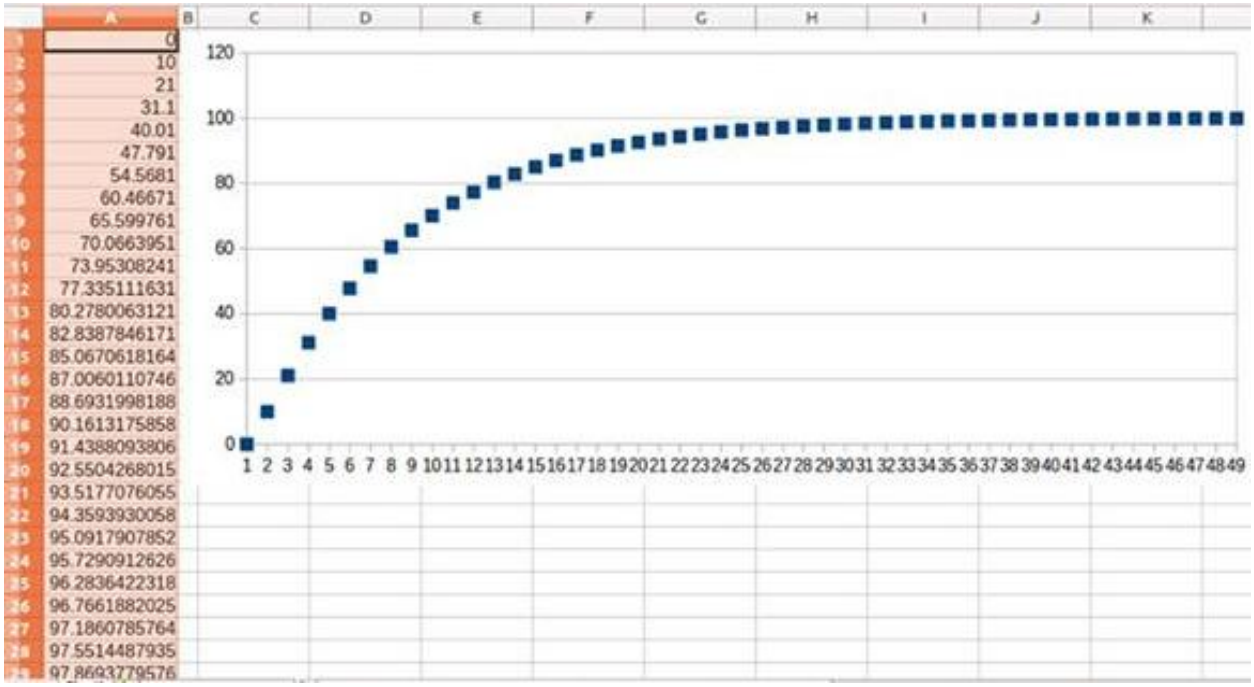


Figure 7: PD Control With Gains Tuned correctly

Experiment 2b: The waypoint queue

After the PD control was implemented, the drone was capable of accepting a single 3d coordinate. Once initiated, the drone would take off, try to reach the desired waypoint, reach the spot using the PD control and land. However in order to explore the area, one co-ordinate would not be enough, hence the concept of waypoint queue was brought in.

We wrote our custom data structure which was heavily inspired from a double ended queue. The data structure (we referred to as waypoint queue) would be populated with certain number of waypoints before the drone took off, the drone would follow those waypoints one by one, and take snapshot of the world as seen from that waypoint.

The waypoints could be added to this queue both from the front end as well as the back end. The backend was the conventional end to add way points as the drone would visit each point in the order specified, while adding from the front end was desirable in case any obstacle was encountered, which would mean the drone would need a temporary new waypoint to displace it from in front of the obstacle. Since the PD control calculated its next command using the front most coordinate in queue, it was advisable to add the displacement co-ordinate to the front end. This way the drone could displace itself away from obstruction and then resume its normal course.

Experiment 2c: Making a PD Controller for Z-axis (Experimental)

During a test flight, it was found out that PD controller was not performing correctly on waypoints around Z-Axis. Upon searching the forums it was discovered that this was a limitation inside the AR-Drone driver that we were using. The driver is open source and has a few lacking, one of which being its inability to detect drone's speed around Z-axis (this speed is always returned as 0mm/s by the sensor). Hence we developed our own method for velocity change around z-axis. The method is as follows

- Computer normal pd control along all axis.
- For the z-axis, store the PD control's z-axis command in a variable vz.
- Once the command is executed to the drone, assume that vz was carried out without any problems.
- Update vz with the new PD control calculation
- Repeat Until way point is reached.

However, despite our best efforts, this implementation still remains in experimental stage. The updating of vz without any check tends to cause unexplained movements of the drone along z-axis which keep increasing the height of the drone and eventually crash it when it hits the ceiling.

Prototype 3: Obstacle Detection System

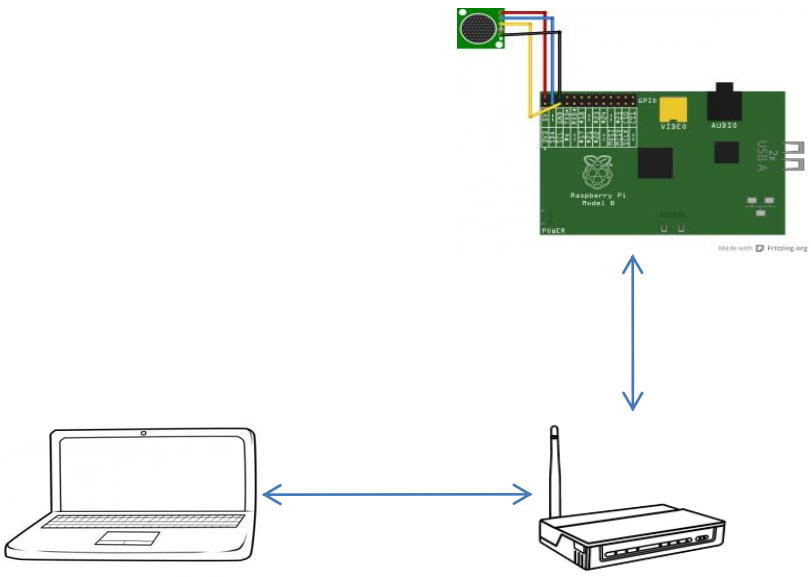


Figure 8

Team: Navigation
Duration: 2 months

This prototype was made so that it could be mounted over the drone for obstacle detection. An additional Wi-Fi dongle was required to be connected to the main computer in order to connect to Raspberry Pi since the built in connection was being used to connect to the drone (Figure 8). This prototype created comprised of a sonar sensor interfaced with Raspberry Pi and was broadcasting the readings received by the sensor to the main computer through the Wi-Fi network created using the additional router. A detailed overview of this prototype is as follows

3a: Installing the Robot Operating System

The first step was to initialize Robot Operating System environment on Raspberry Pi. This was done keeping in mind the requirement that we need our sonar sensor to communicate over same platform as our drone, hence both the main computer and raspberry Pi would communicate best if they were using Robot Operating System. The operating system version initially decided for this was ROS-Indigo, similar to the one being used on main computer. However, due to some package dependency issues we could not install the required version. Hence we shifted to an older distribution ROS-Fuerte, which worked fine on Raspberry Pi.

The communication was done by setting up a ROS_MASTER_URL inside raspberry pi, which directed the device to communicate with the main server at the address provided in the ROS_MASTER_URL. Also a hostname variable was to be set inside the console of RPi to ensure that the main server can recognize this node from its pre-defined name, instead of the IP's which keep changing.

Challenges Faced

- Initial package issues were very difficult to identify, let alone resolve, in ROS distributions for RPi.
- RPi's ROS was supposed to communicate remotely with the main computer over a Wi-Fi. However, the power supplied by RPi was not enough to power the USB dongle. Hence we initially had to use LAN wire to ensure connection has been established.

3b: Programming the Sonar Sensor.

Raspberry Pi provides us with a set of 24 pin output, which is shown in following figure. These pins each have a specific use assigned to them. We used 4 of these pins to connect our sonar sensor to the RPi device. These pins were

1. 5V, Pin2. For Providing VCC to the sensor
2. Ground, Pin6. For Providing a ground connection to the sensor
3. GPIO18 and GPIO23, Pin 12 and 16 respectively. These pins were attached to the Echo and Trig connections of the sonar sensor.

(See Appendix B for a detailed pin map)

After mounting the sensor to the RPi board, we used an open source code available online which enabled sonar sensor to measure distance from any obstruction with maximum frequency of 3Hz. Once the code was written, a Robot Operating System publisher was created with similar frequency as that of the Sensor. The function to measure the distance from the sensor was embedded inside the ROS publisher, which in turn broadcasted it to the ROS_MASTER. ROS_MASTER was our main laptop in this case.

Challenges Faced:

- The IP address initially was dynamic, it kept changing every time RPi was connected to LAN router. Similar case was for the main computer. Hence Raspberry Pi was given a static address both over LAN and Wi-Fi. (Wi-Fi for later use)
- Raspberry Pi was being powered using main's supply. To mount it on drone it would need its own source of power.

3c. Making Raspberry Pi an Independent system

Initially, our main aim was to somehow make sure that Raspberry Pi successfully communicated with the main computer so that distance readings can be used to make drone move about properly. Once this was completed, two major tasks were needed to make sure that RPi could become mountable on the drone.

1. The power to RPi should come from a portable source instead of the main's supply.
2. Raspberry Pi should communicate with main computer over a wireless network, instead of the Ethernet wire connected to the LAN router.
3. The weight of the final system should be inside weight limit specified previously for drone. (i.e. 70 grams)

This was by far the toughest time during the course of FYP as each of the above mentioned tasks took significant amount of time. Each one of them is described below.

- **The power to RPi should come from a portable source instead of the main's supply**: This problem was resolved using a rechargeable power bank available for as low as 250rs. The power bank is capable of providing 2600mAh, and can power Raspberry Pi sufficiently long for a 20 minutes of test flight. The power bank was eventually removed as it was too heavy for the flight and Raspberry Pi was powered using the built in USB port present in the drone.

- **Raspberry Pi should communicate with main computer over a wireless network:** The original hindrance to this problem was the fact that RPi could not generate sufficient power to power the dongle. First we tried to short the VCC and GND of the dongle and attach them directly to the power bank we used above and the remaining data wires could be used for communication. However we later managed to find a low power USB model. This model eliminated the need for any soldering or engineering and allowed us to keep the model simple.

- **The weight of the final system should be inside weight limit specified previously for drone:** This was and still is a persistent problem for the drone. The weight with only 1 sonic sensor easily crosses 70g and therefore causes issues for drone's flight. Drone's flight is greatly affected when the system is mounted on it. Interfacing works well enough in terms of the task, but the overall system creates lots of problem when it comes to mounting on drone. Results of test flight are shared below. The weight was constant at 81g for the experiment, until eventually we found out that we could also power the Raspberry Pi system using the USB port inside the drone, the flight results with this new setting were improved and the drone was able to navigate in a desirable manner. Results of the flights are shown in table 3.0.

Sr. No	Flight Duration	Description
1	3s	Abnormal Take off, crashed sideways
2	1s	Drone could not take off, kept hovering few cm above the ground
3	3s	Abnormal Take off, crashed sideways
4	5s	Weight was mounted after takeoff, steady while hovering, insufficient movement during navigations
5	10s	Weight was mounted after takeoff, steady while hovering, insufficient movement during navigations
6	60s	Raspberry Pi was powered using built in USB port of the drone, the attempt was successful as the weight of power bank was now taken out of the system

Component	Weight
Raspberry Pi	32 grams
Wi-Fi Dongle	5 grams
Sonar Sensor	8 grams
Wires	10grams
TOTAL	**55 grams**

Table 3.0: Final System Specifications

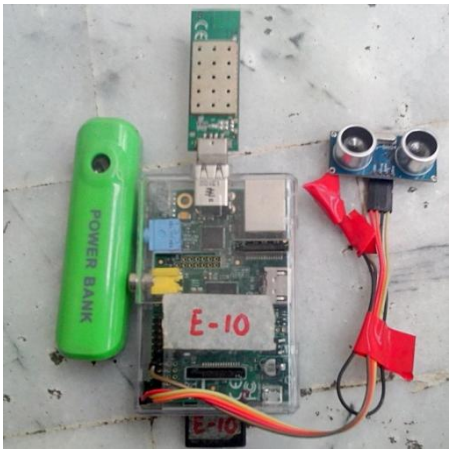


Figure 9: Final Raspberry Pi system with Sonar Sensor Mounted

3d: Mounting the Raspberry Pi on the Drone

This task was intended to mount the sonar system over the drone so it could detect obstacles as they come along the way in its direction of heading. However major weight constraints have so far prevented this part to be implemented in a desirable fashion. Despite best efforts the weight of the overall sensor package has kept going beyond the threshold. The main reason for latter being the weight of the portable power supply cell. The cell is used to power the sonar sensor system and cannot be reduced in weight except for its casing, which is negligible itself.

Without the power bank mounted, the drone takes off properly and is able to navigate around without any further issues, however once the power bank is mounted, the movement becomes highly unpredictable and jaggy.

The system was mounted over the drone as shown in figure below. The results of flight were not exceptional however.

Figure 10: System mounted on drone

Raspberry Pi was configured to automatically connect to a certain network that we were broadcasting through our router. Once the connection was established, we used ssh to enter the raspberry Pi and execute the sonar sensor module.

The sonar sensor communicated with the main computer over a Wi-Fi, which was on a different network mask (i.e. 10.11.x.x) compared to the drone (192.168.x.x).

A class was written in python to subscribe to the broadcast from raspberry Pi and update the proximity flag of the drone at a frequency of 3Hz. Once the class was written, its object was initialized inside the main function of the original program. The automatic callback feature provided by ROS automatically handled the function calls of proximity updates.

3e: Calibrating the field of view of drone's camera using the sonar sensor readings

This experiment was mostly inspired by the need to resize the images received from drone's camera for the sake of classification. Since the drone may be near to an unsafe spot or further away from it at any arbitrary time. We would need to resize the image based on the proximity to properly detect the unsafe object we are looking for.

This was done by plotting a graph of drone's field of view, against the reading on sonar sensor. The steps followed were

- Measured markings were made on a wall using insulation tape.
- Drone's position was adjusted so that it had the insulation tape markings on its either ends. (one insulation tape mark in left most side of picture and one on the right most side).
- The distance between tape markings was already measured, the corresponding reading being given by sonar sensor mounted over the drone was taken to complete the pair
- The process was repeated with a total of 8 different markers on the wall. The results of the graph are shown below

Using the following results, we could conclude a linear correspondance between the drone's field of view and the proximity readings on the sonar sensor. Note that the line overshoots approximately after 270 cm on sonar sensor, because of the limit of 3m on the range of sonar sensor. The sensor cannot detect obstacles beyond 3m in distance, hence beyond 270cm mark, the field of view could is undetermined. Below 270, the scatter plot can be modeled using a line of best fit and the relationship between the two variables can be considered linear.

True Width of View	Sonar Sensor Reading
inch	cm
26	56
36	77
57	120
72	140
107	256
96	210
82	168
350	270

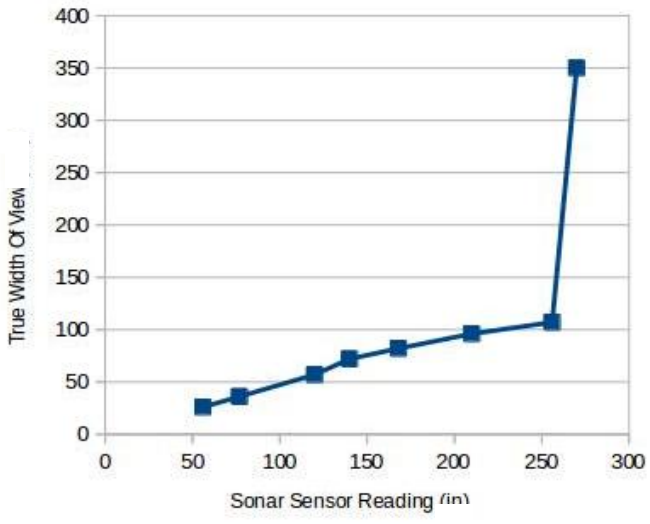


Figure 11: Sonar Sensor reading (left), Graph of Sonar Sensor reading (right)

Prototype 4: Training Model

Team: Visual Classification
Duration: 3 months

One of the objectives of our FYP is object detection. Based on the images received through drone, it is to classify each object on the image to their respective classes. This is done by first collect training dataset and then training our model on those examples. This prototype consists of 3 experiments.

First experiment is about collecting appropriate training dataset which includes outdoor doors as positive samples and outdoor scenes as negative samples. All of these training images, positive and negative samples were resized to specific size (64x128) before training. Similarly, whenever a test image was received, it was first resized to that specific size (64x128) and then used for prediction.

Second experiment is about extracting HOG features from these training dataset of fix size and then training linear SVM model on it. Since, the second experiment didn't produce good results we then shifted to our third and last experiment of this prototype which involves training non-linear SVM model. Non-linear SVM model is when it uses a non-linear kernel function and there exist many possibilities for this kernel function. We tried chi-square and RBF kernel function and faced various issues while doing so as discussed in the experiment.

Experiment 4a: Collecting Training Dataset

In order to collect training dataset, appropriate images of positive and negative samples were found. Positive images consisted of various outdoor doors. Some of these doors were tightly cropped whereas others included some background in it. Some of these positive images are shown in figure 12.

Figure 12: Positive Training Images

Negative samples consisted of outdoor scenes which were collected either online or by taking screenshots from different videos. Figure 13 shows examples of negative images.

Figure 13: Negative Training Images

Experiment 4b: Training Linear SVM Using HOG descriptors:

For implementation in OpenCV using C++, all the training images were first resized and HOG (Histogram of oriented gradients) features descriptors were extracted from these resized images. In HOG, an image is divided into many small squared cell of specific size, histogram of gradients is formed for each cell, results are normalized and then descriptor is formed for each cell.

HOG extracted features are then applied to a supervised learning algorithm known as SVM (support vector machine) for model training. Linear SVM classifiers used which analyzes the data and tries to recognize patterns between positive and negative samples to result an optimal decision boundary. After training the model, a test image is resized and then used for prediction.

Important parameters used are:

- Cell size: [8,8]
- Image size (resized): [40 80]

Testing our algorithm on the training set of 382 positive examples and 1091 negative examples gave the following results:

True Positives: 379
True Negatives: 220
False Positives: 61728
False Negatives: 383007

Ideally, the ratio between positive and negative examples must be around 1:5. Hence, increasing the negative samples to 1701 gave the following results:

True Positives: 378
True Negatives: 325
False Positives: 82117
False Negatives: 525489

Then trying to tightly crop our positive samples gave the following results:

True Positives: 370
True Negatives: 317
False Positives: 59926
False Negatives: 14992

Further changing the standard size of image to [64, 128] and the cell size to [16, 16] also results the same. Our results show false positive and false negative are way too high as compared to the true positives and true negatives which is not a good thing. Expected results for false positive and false negative are 0, since we are testing on the training data.

Experiment 4c: Comparison Between SVM Kernels

SVM linear classification didn't provide good results as mentioned in the last experiment. However, SVMs can efficiently perform a non-linear classification using kernels which implicitly mapping their inputs into high-dimensional feature spaces. These kernels include chi-square, radial basis function (RBF), etc. They give a modular way to learn nonlinear patterns using linear models which gives a non-linear decision boundary in the original space.

While training our SVM model using chi-square kernel, various problems occurred. Firstly, there were difficulties in finding the chi-square implementation in LibSVM, OpenCV 2.4 because kernel functions are available depending on the version of OpenCV used. In 2.4.9 version of OpenCV, SVM part of the opencv_ml library only supports three kernel types: radial basis function (RBF), poly and sigmoid. However, in the 3.0.0 version, OpenCV implements 5 kernel functions including previous three kernels along with chi-square (CHI2) and intersection (INTER). Though chi-square implementation was found in OpenCV 3.0 but then there also existed others issues regarding HOG and image handling. Secondly, we were unable to save the model in a file.

Due to the issues faced while using chi-square, we changed the kernel from chi-square to RBF and also implemented 5-fold cross validation. 5-fold cross validation is when the training data is divided into many equal parts and each part is considered as test data one by one while the others behave as train data. This technique resulted in quite good results as compared to our previous approaches. However, when tested on different dataset (not extracting test data from training data), the results went bad again.

Prototype 5: Techniques Used to Improve Results

Team: Visual Classification
Duration: 7 weeks

This prototype focuses on improving the results achieved using previous experiments because those results were not as good as expected. This prototype consists of three experiments so, three techniques were implemented for results improvement.

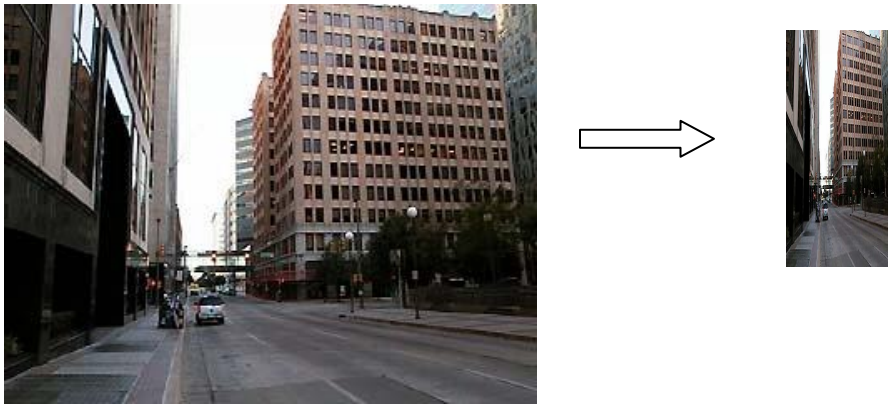
First experiment is about implementing a sliding window of size 64x128 on a test image along with 50% overlapping. When a test image is received from drone for prediction, then the sliding window starts sliding from top left corner of the image and also keeps predicting if a door or window exist in that frame. Hence, rather than resizing a test image which gives just one frame (size 64x128) and then using it for prediction, this experiment uses a sliding window on original size image which gives many frames (size 64x128) on which prediction occurs.

Second experiment is about updating our negative samples in the training dataset. Each negative image was taken and replaced with many smaller divided frames (of size 64x128) of itself. Hence, rather than resizing a negative image to 64x128 before training, negative image of original was divided into many smaller parts of size 64x128 and used for training dataset. In addition, some of those smaller frames which consists of doors or windows were moved from negative samples to positive samples.

Third experiment is to save the trained model in a file. Rather than training the model every time you need to do prediction, this approach helps to load the saved model whenever needed. This technique saves a lot time for prediction hence, results are computed faster.

Experiment 5a: Sliding Window Implementation

The reason for getting such poor results was found to be the resizing of the test image. Before prediction, a test image was being resized to 64x128 (which is the size of all trained images). This resizing resulted in a squeezed image as shown in Figure 14 which made it difficult for the classifier to do its work. It is difficult for the classifier to detect a door in an image if that image



has loosen its originality.

Figure 14: Original picture being squeezed

Sliding window was implemented on the test image. This sliding window slides over the original test image (without resizing it) frame by frame and classifies door in each frame as shown in Figure 10. The size of the sliding window is set to be 64x128which is the size of all positive training images used.

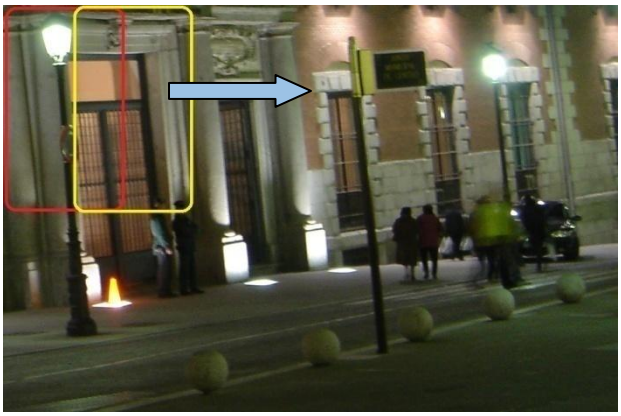


Figure 15: Original picture being squeezed

Sliding window starts from top left position (0, 0) and keeps on sliding until it reaches the end of the image which bottom right corner. The first frame is shown in red color followed by the next frame shown in yellow. If a door exist half at frame one and the other half at frame two while there is no overlapping of windows then the classifier may miss that door. To overcome that issue, we have overlapped windows. Both the frames shown in Figure 15, overlap by 32 while moving horizontally and by 64 while moving vertically.

Experiment 5b: Editing Negative Training Images

The results found were still not good enough due to the resizing of the negative images. As stated above, negative images include various outdoor scenes which were being resized to 64x128 before training. The main issue was that resizing image of an outdoor scene leads to squeezing or distorting of the image as shown in Figure 14.

Resizing the image loses its originality so, the classifier gets the wrong training negative images. For instance, an original image containing a building of ratio 1:10 (width: height) would change its shape after resizing resulting in the same building of ratio 1:30 (width: height). Hence, the classifier was unable to differentiate between positive and negative images and accurately predict on test images.

To overcome this problem, each negative image is broken down into many smaller images as shown in Figure 16.These many smaller images of size 64x128 are then used instead of the one original image. Using the updated negative training dataset, the model is trained and saved.

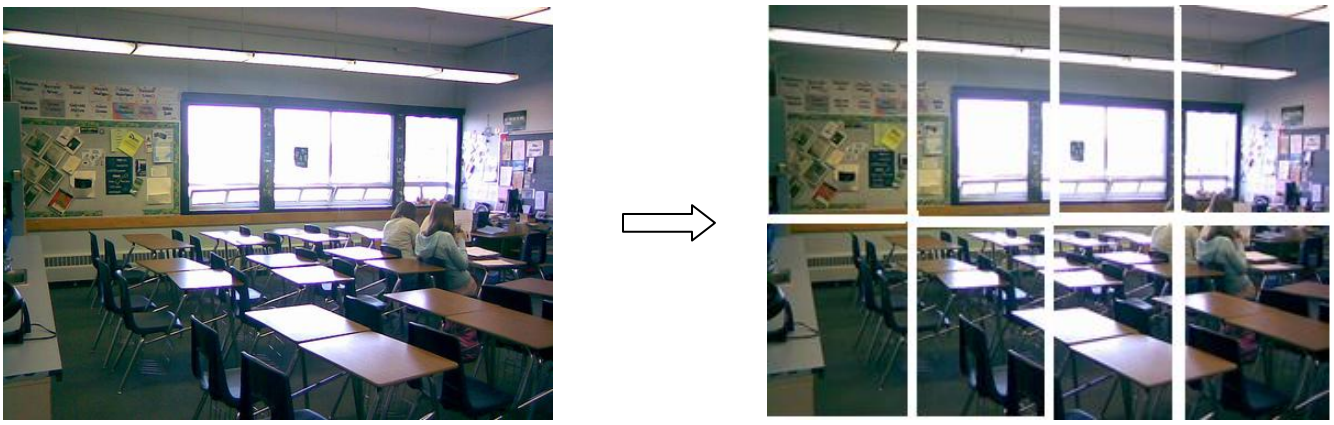


Figure 16: Original picture being broken into smaller images

After dividing each negative image into many smaller parts, it was noticed that some of those smaller images consisted (either partial or complete) doors or windows as shown in the figure below. So, all those smaller images (of size 64x128) were removed from negative samples and added to the positive samples.

Figure 17: Smaller images moved from negative to positive samples

Experiment 5c: Saving SVM Model using Python

In order to integrate with navigation part which is done using Python language, all the OpenCV code was needed to be in Python. There were two possible ways of doing so. We could have made a wrapper class which would easily call the functions written in C++. On the other hand, we could completely change the code from C++ to Python. We opted for the second choice and converted the whole code into Python.

After training the classifier, the SVM model needs to be saved. Saving the model helps to use it later on without the need of training all over again. While trying to save the model in C++, various issues were faced however, after changing the code to Python those issues were resolved. Here is a piece of code showing how the model is saved.

```
m = svms.svm_train(……)
svms.svm_save_model('myModel.xml',m)
```

As shown in the code, SVM train command returns a model m which is then saved into a file named as myModel.xml.

```
m = svms.svm_load_model('myModel.xml')
svms.svm_predict(……., m)
```

Later on, when we need to predict if the door has been detected or not on a test image, the saved SVM model is loaded back into m and used for prediction.

Prototype 6: Various Approaches Used for Prediction

Team: Visual Classification
Duration: 4 weeks

This prototype focuses on various approaches implemented on test image either before prediction or after prediction. This prototype consists of three experiments. First two experiments consist of approaches that are implemented before prediction and the last experiment consists of approach that is implemented after prediction is done.

First experiment is about ignoring some of the border areas of a test image before starting the sliding window on it for prediction. This approach helps to limit the search area for prediction which will also give faster results.

Second experiment is about forming a pyramid which resizes a test image into different scales and then start the sliding window of size (64x128) at each scaled image for prediction. Each scaled image means different level of the pyramid. However, the size of the sliding window being 64x128 remains same at each pyramid level. This approach is implemented to cater the situations when the door or window is either too near or too far away from the drone.

Third experiment is to display a bounding box in a test image at various locations wherever the door or window is detected. This approach is implemented after getting the prediction results.

Experiment 6a: Border Removal of Test Image

Before prediction, sliding window is implemented on a test image to extract features of each smaller window. This sliding window starts at top left corner (0,0) and then keeps on sliding. However, checking the borders of an image for the presence of door is unnecessary mainly due to two reasons.

Firstly, the main focus of the drone should be at the center of the image rather than the borders. So, there is no need to consider the borders for prediction and increase the search area. It's better to limit the search area which will also give the prediction results faster. Secondly, the borders will contain something that is incomplete hence, even if the classifier detects a door there is very less chances that the prediction is accurate.

To avoid the checking borders for door, sliding window starts at top left corner plus the border value (0+value, 0+value). The appropriate border value can be specified by the user. Moreover, the sliding window after sliding from one frame to frame, it may still overlap with previous window to avoid missing a door in between.

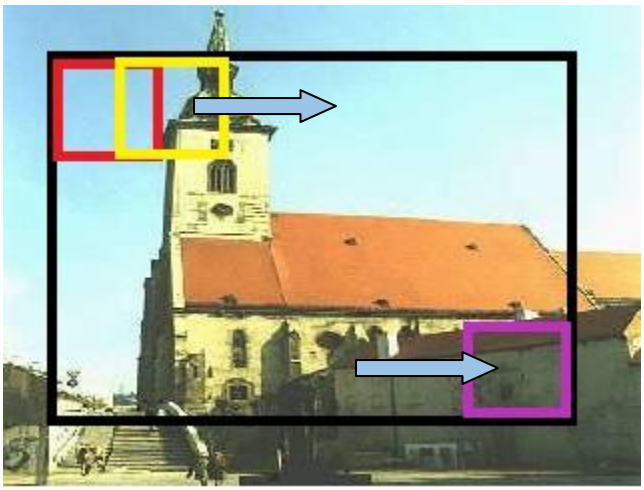


Figure 18: Sliding Window along with border removal

The black square shown in figure 18 is the border which needs to be ignored by sliding window. Hence, the 1st sliding window shown in red doesn't start at (0,0) but rather starts after the border. Sliding window then slides to next position shown in yellow. This process goes on until the sliding window reaches the last frame in the image as shown in purple.

Experiment 6b: Pyramid Formation to Scale Image Size

The sliding window implemented on a test image is of size 64x128 regardless of the size of the test image. If the test image contains a door of size greater than 64x128 then the sliding window will not be able to detect the door. This situation occurs when the distance between the door and drone is very less hence, the door is near to the drone. Similarly, test image containing a door of size quite smaller than 64x128 will also face detection issues. This situation occurs when the distance between the door and drone is too high hence, the door is far away from the drone.

To overcome these issues, a test image needs to be resized before implementing sliding window. If the door is too near to the drone then the door may appear quite larger in size. This image needs to be decreased in size to fit the door inside the sliding window of size 64x128. On the other hand, if the door is too far away from the drone then the door may appear quite smaller in size. This image needs to be increased in size so, the door size also increases to approximately 64x128.

We have formed a pyramid type structure to resize the test image. As you know, pyramid size decreases while going upwards. We adapt a similar approach of resizing the image to reduce its size and then classifying for door. For a test image, we first implement the sliding window on its original size and find out if the door is detected. Then we keep on decreasing the size of the image, and implementing sliding window on it to check for door. This process keeps on going.

This technique is used to map the sonar sensor readings to the size of the image. If the sensor reading specifies that the distance between door and drone is less, then image is zoomed out to

get good results. Similarly, if the sensor reading specifies that the distance between door and drone is high, then image is zoomed in size to get good results.

Figure 19 shows an example of how to resize an image which is too near to drone.

Figure 19: Scale image to fit sliding window

Experiment 6c: Bounding Box Formation

In order to identify the exact position where the door is detected on a test image, a bounding boxes needed to be displayed on that frame. A test image is divided into many frames as produced by the sliding window before prediction. So, those frames no. where the door is detected, are needed to be identified.

For prediction, SVM predict command is used which returns 1D array being size of the no. of frames in that test image. This array contains either 0 or 1 for each frame, 0 indicating “not detected” and 1 indicating “detected”.

After retrieving all the frame no. of detected door, x location of that frame can easily be calculated by taking the mod of frame no. by total possible frames on x-axis and multiplying with width of the frame. Similarly, the y location can also be calculated of that frame by dividing the frame no. by possible frames on y-axis and multiplying with height of the frame. Hence, once the location of the frame is known, the bounding box can be displayed on that position as shown in Figure 20.

Figure 20: Bounding Box (single door in an image)

It is possible for a test image to contain more than one door which means more than one frame will be indicated as “detected” leading to many bounding box in an image.(Figure 21)

Figure 21: Bounding Box (many doors in an image)

Prototype 7: Comparison between Different Models

Team: Visual Classification

Duration: 3 weeks

This prototype focuses on training new models. This prototype consists of two experiments.

First experiment describes the training of newer model. For that, a new dataset is collected. Training dataset includes indoor doors as positive samples and indoor scenes as negative samples. All of the positive training images, and not the negative images were resized to specific size (64x128) before training.

Second experiment describes the prediction results from both the models, old and new. The test images used for prediction were same for both the models so, that the comparison would be fair.

Experiment 7a: Training Various SVM Models

Firstly, our SVM model was trained for outdoor environment. The training dataset of positive examples consisted of exterior doors and the negative examples were of various outdoor scenes. This trained model worked well while flying the drone in an outdoor environment but not while flying it indoor environment.

To fly the drone in an indoor environment, a new SVM model was needed to be trained for that specific environment. So, a second SVM model was trained using new images for positive and negative training examples. Old positive examples were replaced by interior doors. These images included doors from various angles horizontally as shown in the figure 22.

Figure 22: New Positive Training Images

Old negative examples were replaced by all indoor things in the world rather than doors. These negative examples include indoor scenes containing table, chair, board, fan, TV, etc. Some of these negative images are shown in figure 23.

Figure 23: New Negative Training Images

These negative images were further divided into smaller images and those smaller images containing doors (partial or complete) were moved from negative to positive samples. To conclude, two different SVM models were trained each with different kind of dataset. After training the model, it was saved for later use of door prediction.

Experiment 7b: Results From Various SVM Models

Prediction results were saved for both models trained. One model for outdoor environment and one model for indoor environment.

A total of 135 test images (including 3 pyramid levels of an image) were used for prediction using both models so the result can be compared. For the model trained using outdoor training dataset, following results were calculated manually:

True Positives: 49
True Negatives: 1703
False Positives: 26
False Negatives: 22
False alarm rate: 0.015037594

For the same model, prediction results on some test images are shown below.

Figure 24: Prediction results for first model (outdoor training dataset)

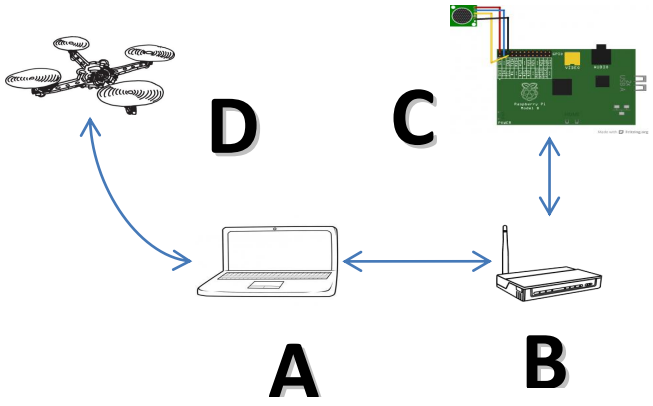
For the model trained using indoor training dataset, following results were calculated manually:

True Positives: 20
True Negatives: 1734
False Positives: 19
False Negatives: 27
False alarm rate: 0.0108385625

For the same model, prediction results on some test images are shown below.

Figure 25: Prediction results for second model (indoor training dataset)

Prototype 8: Adding SVM Classifier to Navigation Module



Team: Navigation and Visual Classification
Duration: 1 month

The overall nature of this task was straightforward and easy. Most of the time was taken to figure out LibSVM issues while running the integrated code. Eventually the source code had to be shifted inside the LibSVM python folder in order to make it recognize LibSVM specific commands. A class was written in python to manually load an image and classify whether it contained a door or not. Once this class was running satisfactory, we introduced a ROS specific message subscriber into it, which took input of a raw image coming from Drone, converted into an image recognizable by OpenCV and passed that image as a parameter to the classification model to detect the door.

Challenges Faced:

- The initial setup of the classifier kept returning an index out of bound error over the image. The problem was identified to be caused due to non-conforming window size over the image. Fixing the window size solved the issue.

After getting done with an ROS version of the trained classifier, we next moved on to integrate it with the existing PD control code that was being run by the drone. Due to ease of use created by object oriented programming paradigms, we only had to initiate an object of the previously mentioned class inside the main function of our original program. The automatic callback feature provided by ROS handled the function call to classifier on its own.

Challenges Faced:

- The continuous visual feeds from the drone were creating lots of load for the program. The program had problems processing 30 frames per second for its classification. This was resolved by introducing a delay of 100 function calls between image classifications. So after every 100 cycles, a picture from drones current feed would be sent to the classifier.

Discussion

We have eventually come up with a prototype of the drone that is capable of taking off at an unknown location, explore a certain number of randomly generated co-ordinates, and analyze the visual feeds at these co-ordinates to decide a safe spot and land there. The drone is also capable of avoiding any obstacles that come in the direction of its front camera. A few points of discussion that we had to come across and resolve are discussed in this section.

During the project AR Drone was found to be robust in its operation. Despite multiple crashes into bushes and trees, the drone managed to resume on its normal operation. However the two main limiting factors of the AR Drone were its not so efficient ability to carry weight, and the unavailability of a distance sensor.

This required consideration for mounting of Raspberry Pi on the drone and to equip it with a depth sensor. The trade-off in this case was the battery time, which got reduced further. The sonar sensor had its limitations over the maximum frequency (2Hz) and the Maximum range (3m) that it could detect. The limitation of frequency did cause undesirable behavior during the flight, so for most of the experiments the Sonar module was detached from the Drone and a hand was placed manually in front of it when drone got near to an obstacle. This was done to ensure that in case there is a delay in sending of commands to the drone, the crash would not affect the raspberry pi system mounted over it.

Under ideal conditions the drone would fly to its desired location using the implemented PD control. However the presence of wind hampered the flight of the drone and influenced the outcome of a number of flights during our experiments. During indoor flights however, the flight was stable, except for a few durations when the drone would fly near a curtain.

Few inherent issues that we had to face due to the problems in the open source driver "AR-Drone autonomy" also took a considerable chunk of time, first to figure out and then to fix. Three of the most notable ones included: random incorrect readings from the IMU sensor, inability of the drone to measure velocity around z axis and the gradual shifting in the Yaw angle of the drone despite being parked at the same place.

Resolving weight considerations also made up a significant part of the system because initially the sonar sensor prototype was exceeding desirable weight limits for the drone, causing it to sway before a proper take off. This was resolved using the USB cable provided inside the drone for "data transfer", however the current provided by it was found out to be sufficient enough to power the sonar prototype, thereby reducing the weight by almost 25grams.

While training indoor and outdoor SVM models for door detection, we initially used all tightly cropped images of doors and windows for positive training dataset. This technique made the classifier to detect all possible rectangles (even if it’s not a door) in a test image to be as door. To overcome this issue, we edited 491 of our positive training images of door with some background. To conclude, we included loosely coupled and tightly coupled doors in the positive training dataset to make sure only those

rectangles containing doors are detected and also to avoid missing a loosely coupled doors in a test image.

Suggested Future Work

This prototype is only a basic version of the overall idea of autonomous drones. We have managed to build a system at a very basic level and there are a number of options to improve this idea. Some of these are suggested below

- **Four Way Sonar Sensor**: Currently Drone has only 1 sonar sensor in its obstacle detection module, this was due to previous weight considerations which were resolved by the end of the project. This modification would enable drone to make more informed decision especially as it will have proximity data from more directions than just the front.

- **The PD control along z-axis:** Even though we have implemented a model that makes drone navigate to a desired position along z-axis, the overall performance is highly unpredictable, our model assumes that the z-axis commands that were issued were carried out religiously without any flaw, this assumption leads to elongated movement along z-axis in frequent cases.

- **A Probabilistic Model for Navigational Confidence:** Our prototype could use some kind of probabilistic model like a Kalman Filter to determine the probability of a drone actually being at the co-ordinate that it is claiming itself to be in. Currently our method of ensuring a reliable set of readings is to take an average of 5 previous readings of the drone and use the average in our calculations

- **A Bigger Definition of Un-Safe spots:** For this prototype, the only spots considered unsafe are the spots where we have a door or a window, this definition could be expanded to suite a bigger variety of locations that can be considered unsafe, for example benches at public places, or in front of surveillance cameras.

Conclusion

We have eventually come up with a prototype of the drone that is capable of taking off at an unknown location, explore a certain number of randomly generated co-ordinates, and analyze the visual feeds at these co-ordinates to decide a safe spot and land there. For the scope of this project, doors and windows quality as unsafe spots. The drone is also capable of avoiding any obstacles that come in the direction of its front camera. We accomplished this by writing two separate modules one each for object recognition/detection and for autonomous navigation. The autonomous navigation module comprised of two subsystems, one for developing the PD control to allow the drone to go from a given point A to point B, second one for mounting a sonar sensor prototype on the drone to make it avoid colliding into obstacles in its direction of heading. The classification subsystem was created using the HOG features trained over an SVM using an RBF kernel. We used a total of 2 models (one with indoor pictures and one with outdoor pictures), their results can be seen in prototype details of this technical report. Since this is a prototype, we leave a set of suggested improvements for anyone who would like to build this system further.

References

[1]Fung, Brian (16 August 2013). "Why drone makers have declared war on the word ‘drone’". The Washington Post. Archived from the original

[2] Sabine Hauert, Severin Leven, MajaVarga, Fabio Ruini Angelo Cangelosi, Jean-Christophe Zufferey, Dario Floreano , "Reynolds flocking in reality with fixed-wing robots: communication range vs. maximum turning rate"

[3] Robert Kanyike. "History of U.S. Drones"

[4] Jakob Engel, J¨urgen Sturm, Daniel Cremers , “Camera-Based Navigation of a Low-Cost Quadrocopter” , Intelligent Robots and Systems

[5] Patrick Doherty and G¨ostaGranlund and Krzystof Kuchcinski3 and Erik Sandewall and Klas Nordberg and Erik Skarman and Johan Wiklund , “The WITAS Unmanned Aerial Vehicle Project”

[6] TUM Visual Navigation for Flying Robots Course Website: http://vision.in.tum.de/research/quadcopter

[7] Bennett, Stuart (1993). A history of control engineering, 1930-1955

Appendix A: AR Drone Specifications and Observations

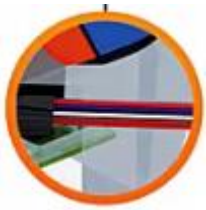
Technical specifications:

1. Dimensions: 77.7 x 38.3 x 12.5 mm
2. Weight: 380 g (With Outdoor hull)
3. Accuracy: +/- 2 meters
4. Frequency: 5Hz
5. Voltage: 5V TBC
6. Flash memory: 4 GB
7. Actuators
8. Ultrasound height sensor
9. Visual odometery sensor
10. Front camera
11. Linux embedded system

Schematic Diagram:

HD Video Recording

Electronic Assistance



Robust Structures

Motors

Initially Parrot AR-Drone 2.0 was flown manually to observe features and limitations of drone. Drone was flown and controlled by an Android Smartphone. Height limit, Wi-Fi limitation, Camera and Sensor limitation observed are mentioned in table below

Feature	**Observation**
Flight Time	16 minutes
Maximum Range	Tested safely until 40 meters (without any obstacles in line of sight)
Picture Dimension	1024 x 720 pixels
Frame Rate	30 fps
Command Delay	2 seconds max

Apart from the readings, the following observations were made

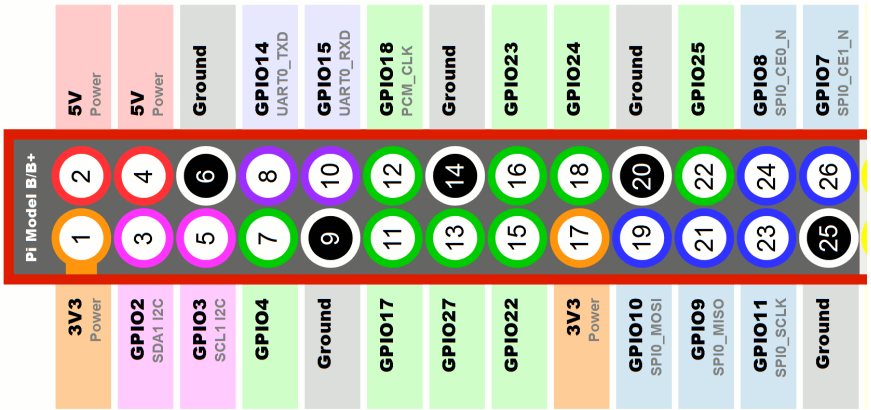
- Drone can fly up to the height of 40 meters but direction and speed of air can vary this range plus wind direction caused some problems to control the drone.
- A place where there is more than one Wi-Fi connection is also problematic for drone.
- It was concluded that ultrasonic sensor at front is required for depth sensing. For which Arduino can be used to solve the problem.

Appendix B: Raspberry Pi

Specifications:

Operating system	Linux (Raspbian Wheezy)
CPU	700 MHz single-core ARM1176JZF-S
Memory	512 MB
Storage	8GB SD Card
Graphics	Broadcom Video Core IV
Power	.5 W

Raspberry Pi provides us with a set of 24 pin output, which is shown in following figure. These pins each have a specific use assigned to them. We used 4 of these pins to connect our sonar sensor to the RPi device.



Appendix C: Ultrasonic Sensor Model HCSR04

Ultrasonic ranging module HC - SR04 provides 2cm - 400cm non-contact measurement function, the ranging accuracy can reach to 3mm. The modules includes ultrasonic transmitters, receiver and control circuit. The basic principle of work:

- Using IO trigger for at least 10us high level signal.
- The Module automatically sends eight 40 kHz and detect whether there is a pulse signal back.
- IF the signal back, through high level , time of high output IO duration is the time from sending ultrasonic to returning.
- Test distance = (high level time x velocity of sound (340M/S) / 2,

Pin Specifications:

- 5V Vcc
- Trigger Pulse Input
- Echo Pulse Output
- 0V Ground

Electric Parameter

Working Voltage	**DC 5 V**
Working Current	**15mA**
Working Frequency	**40Hz**
Max Range	**4m**
Min Range	**2cm**
MeasuringAngle	**15 degree**
Trigger Input Signal	**10uS TTL pulse**
Echo Output Signal	**Input TTL lever signal and the range in proportion**
Dimension	**45*20*15mm**

Appendix D: Limitations of AR-Drone Autonomy Driver

This package is a driver for the Parrot AR-Drone built for Robot Operating System. It is available for download as an open source software at https://github.com/tum-vision/ardrone_autonomy. We have used this driver to control the drone for our project. However, since this package is open source, there are a number of limitations that we encountered during our FYP. These driver specific problems are discussed in this section

Abnormal gyroscope/IMU readings while takeoff on random occasions: This problem was the hardest one to figure out, often the drone would be given a take-off command and it would instead sway in one direction and crash against a wall before a proper take-off. Our initial guess was that this happened due to Wi-Fi interference created by other networks in surrounding, however later we found out that the driver was providing the drone with erroneous readings of its orientation. Due to these readings, even though the drone took off from a flat surface, the sensor readings stated otherwise, hence it tried to change drone's orientation according to sensor data, resultantly the drone used to tilt in certain direction and resulting torque would make it sway away. This problem still persists and occurs on random occasions.

Velocity along z-axis always measured zero: This problem created the limitation that PD-Control along z-axis would not work properly for the drone, no matter how fast the drone was rising or descending, the driver shows a constant 0m/s speed being received from the sensor. We partially resolved it using a substitute variable in our code as mentioned in experiment 2c, however the solution remained experimental only.

Yaw angle of the drone keeps updating automatically even though the drone is parked on a stationary surface: If the drone was left parked at any location and it readings of yaw angle observed, the readings would keep changing gradually with time and give a difference of as much as 5 degrees after 2 to 3 minutes, this change would get bigger if the drone was heated up after a flight. This implied that the readings obtained from Yaw-angle sensor would be unreliable over time. Hence we took the initial yaw angle reading at the start of our code, and every time a reading came, we subtracted it from the initial reading to create a "False North", this way the drone's direction of heading would always be 0 at takeoff time.

Units of Measurement: The drone's API stated that the readings received from the sensors for the velocity along all 3 axis is in mm/s. However the documentation failed to mention that the commands are NOT issued in the same unit of mm/s. rather the commands were being taken in m/s unit. This created lots of problem initially since our PD control was calculating based on mm/s readings and issuing commands in same unit, however the drone was executing the same command with 1000 times more magnitude due to the unit conversion misinformation. The problem was identified when our gains for Kalman filter were to be reduced by 1000 times the normal values to make the system stable. This issue was resolved in our code once it was identified.