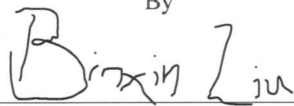



Heterogeneous Network of Autonomous Vehicles

A Major Qualifying Project Report
Submitted to the Faculty of the
WORCESTER POLYTECHNIC INSTITUTE
in Partial Fulfillment of the Requirements for the
Degree of Bachelor of Science
In Aerospace Engineering

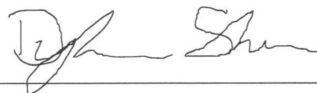
By



Binxin Liu



David Moore



Dylan Shields

Approved by:



Prof. Raghvendra V. Cowlagi, Advisor

Prof. Michael A. Demetriou, co-Advisor

Aerospace Engineering Program, Mechanical Engineering Department, WPI

This report represents the work of WPI undergraduate students submitted to the faculty as evidence of completion of a degree requirement. WPI routinely publishes these reports on its website without editorial or peer review. For more information about the projects program at WPI, please see <http://www.wpi.edu/academics/ugradstudies/project-learning.html>

Abstract

This project involves the design and implementation of a feedback control system to enable a quadrotor unmanned aerial vehicle (UAV) to detect and track the motion of a terrestrial wheeled mobile robot. The detection of the terrestrial robot is enabled by a camera carried onboard by the UAV, in conjunction with unique identification marks placed on the terrestrial robot. Raw camera data is processed by an algorithm developed using the open source software library OpenCV, which is executed on a Raspberry Pi embedded computer carried onboard by the UAV. To correct errors in the vision-based measurement of the position of the terrestrial robot, a Kalman filter is developed, which is also to be executed by the Raspberry Pi computer. Preliminary test results are reported.

Fair Use Disclaimer: This document may contain copyrighted material, such as photographs and diagrams, the use of which may not always have been specifically authorized by the copyright owner. The use of copyrighted material in this document is in accordance with the “fair use doctrine”, as incorporated in Title 17 USC S107 of the United States Copyright Act of 1976.

Table of Authorship

Section	Author	Project Work
Background		
Project Objective	BL	N/A
Ground Vehicle	BL	BL
Unmanned Arial Vehicle	DM	DM
Object Recognition	DS	DS
Kalman Filter	BL	BL
System Design		
Ground Vehicle	BL	BL
The IRIS Quad Copter	DM	DM
MAVLink	DM	DM
Code Construction	DS	DS
Kalman Filter	BL	BL
System Development		
Ground Vehicle	BL	BL
IRIS Quad Copter	DM	DM
Code Construction	DS	DS
Kalman Filter	BL	BL
Results and Discussion		
Ground Vehicle	BL	BL
IRIS Quad Copter	DM	DM
Code Construction	DS	DS
Kalman Filter	BL	BL

Contents

Abstract	2
Table of Authorship	4
Contents	5
List of Figures	7
1 Background	8
1.1 Project Objective	8
1.2 Ground Vehicle	8
1.3 Unmanned Aerial Vehicles	8
1.4 Object Recognition	9
1.5 Kalman Filter	10
2 System Design	11
2.1 Ground Vehicle	11
2.2 The IRIS Quad Copter	11
2.3 Battery	14
2.4 MAVLink	16
2.5 Code Construction	17
2.6 Object Detection	17
2.7 Navigation	19
2.8 Kalman Filter	20
3 System Development	22
3.1 Ground Vehicle	22
3.2 IRIS Quad Copter	22
3.3 Code Construction	23
3.4 Kalman Filter	24

4	Results and Discussion	25
4.1	Ground Vehicle.....	25
4.2	IRIS Quad Copter.....	25
4.3	Code Construction.....	26
4.4	Kalman Filter	27
5	Conclusions.....	28
6	Bibliography	29

List of Figures

Figure 1: Iris Quadrotor front view	13
Figure 2: Iris quadrotor bottom view	14
Figure 3: Planned code flow	17
Figure 4: Object detection locating the target in a scene	18
Figure 5: EZ-B v4 Wi-Fi Robot Controller (ezrobot.com)	22
Figure 6: Successful object detection.....	23
Figure 7: Matlab output of the matlab test of the kalman filter	27

1 Background

This project is about interfacing between vehicles of different types. The scope of the project is limited to two vehicles, a ground vehicle, and an aerial one, with a visual uplink between them. This section details extant information on the tools we use to achieve that goal: the two vehicles, the computer vision algorithms, and Kalman filtering.

1.1 Project Objective

The objective of this project is building a heterogeneous network between the ground vehicle and the autonomous UAV to enable the UAV to detect, track and follow the ground vehicle automatically. For achieving this objective, the video tracking, navigation and guidance system will be built on the autonomous UAV. This system consist of an onboard camera, a Raspberrry Pi miniature computer and a Pixhawk autopilot computer of the autonomous UAV. An object detection and navigation program will be built on the Raspberrry Pi miniature computer and the navigation data will be given to the autopilot guidance program on the Pixhawk computer to enable the UAV follow the ground vehicle automatically.

1.2 Ground Vehicle

iRobot Create is designed for robotics development. It is developed from the iRobot Roomba vacuuming cleaning robot. It replaces the vacuum cleaner with a cargo truck with a DB-25 port for serial communication, digital input and output, analog input and output and electrical power supply. It also has a 7-pin Mini-DIN serial port. This port can transfer the sensor data and the motor command that using the iRobot Roomba Open Interface (ROI) protocol.

1.3 Unmanned Aerial Vehicles

Unmanned aerial vehicles have been in use for several decades. Their first use was much longer ago than that where balloons were used to drop bombs on (hopefully) enemy targets. It was in World War I and more prominently later in World War II where UAVs gained some notoriety. The V1 Rocket, also known as the Buzz-Bomb, that were used against Britain were UAVs. The Nazi Germans would used them to fly without a pilot over England and when they ran out of fuel they returned to earth delivering their destructive ordinance. In more modern times, one of the most prolific is the Predator Drone used by the United States Military. These UAVs are piloted by soldiers remotely to perform a wide variety of reconnaissance and combat operations. With the continued miniaturization of electronics and advancements in computing

power on smaller and smaller devices. The UAV is now one of the fastest growing hobby markets in the US and is being explored as viable options for large corporations like Amazon as well.

The advantage of having a vehicle fly without a human pilot is appealing in these situations. A Military spends an enormous amount of time and energy to train its pilots, removing the risk of loss of life is an important effort. For companies like Amazon, UAV's are appealing for the purpose of delivering packages faster or in very remote delivery locations. For the consumer, UAVs are fun to fly and also when equipped with a camera, make for entertaining home videos.

Unmanned aerial vehicles have come to be colloquially named 'drones.' This is because of the extensive use of the Predator Drone, and also because a drone, by definition, is a low humming sound which is similar to the sound that many UAVs make. Unfortunately the term 'drone' has acquired the stigma of relating to the military operations more than the sound of low humming. In our work, we actively avoid using the term 'drone' as our work is not militarized in any way.

1.4 Object Recognition

The heart of object recognition software in most applications is a library of complex vision algorithms known as OpenCV. OpenCV was developed originally in 1998 at Intel, and is now used by professionals and amateurs around the world to create vision applications. The name stands for Open Source Computer Vision, and has grown into an extremely powerful tool. The software has been downloaded seven million times, and can be used in a myriad of situations. In their own words, "OpenCV's deployed uses span the range from stitching streetview images together, detecting intrusions in surveillance video in Israel, monitoring mine equipment in China, helping robots navigate and pick up objects at Willow Garage, detection of swimming pool drowning accidents in Europe, running interactive art in Spain and New York, checking runways for debris in Turkey, inspecting labels on products in factories around the world on to rapid face detection in Japan." [5]

The software contains hundreds of algorithms for different purposes. One such algorithm is SURF, Speeded Up Robust Features. It was presented by Herbert Bay in 2006 as an improvement upon existing detection programs. It uses simplified integral images and Hessian matrices to detect points of interest in a picture. These points of interest are points where the image changes suddenly, such as the edge of an object, or the corners of text. The area around those points is represented by a vector. Its primary purpose is to take two images of the same scene, and match the vectors between them. For example, if you were to use the algorithm on a close up image of a book and another image of that book from a distance, it would give you data that could be used to identify the book in the larger scene. SURF is more repeatable and, most importantly, much quicker than its predecessors. [2] That speed is what gives it its name, notwithstanding

the poor grammar. SURF uses very complicated mathematics that are outside the scope of this project, but we recommend anyone using vision detection to read Bay's article on the subject.

1.5 Kalman Filter

Kalman filtering is also known as linear quadratic estimation (LQE). It uses a series of measurements, which contains measurement noise and other noise to produce more precise estimation of unknown variables. It is named after the theory developer Rudolf E. Kálmán. It is widely used in Guidance, Navigation and Control of vehicles.

In our project, a Kalman filter is used to filter the measurement noise of tracking camera on UAV and produce a more precise estimation of the position of the ground vehicle and the guidance of UAV needs this estimation to generate a more precise route.

2 System Design

2.1 Ground Vehicle

The ground vehicle should be detected and followed by the UAV. For easy detection of the target by OpenCV object detection program, a significant visual feature should be placed on the top of the ground vehicle. After the tests with the solid spot, the solid square and the QR code, we selected QR code since it is easiest to be detected and recognized by our object detection program through the on board camera. The ground vehicle is required to achieve a remote control of its motion by using the ground station computer. So building wireless connection between the ground vehicle and the ground station is the most important element.

2.2 The IRIS Quad Copter

Our UAV is a small quad-copter called the IRIS. A quad-copter is a helicopter with 4 propellers mounted away from the center of the vehicle in a two by two pattern. You can think of them like tires on a car. One front-left, one front-right, one rear-left, and one rear-right. The control of the propellers is operated by a microcontroller inside of the quad-copter that changes the speed of the propeller motors accordingly. The microcontroller also takes input data from a variety of sources like the GPS receiver and IMU and also has the remote control receiver and other connectivity options such as for an antenna to transmit telemetry data to a ground station.

The IRIS components are controlled by its brain, the PixHawk. The PixHawk is a microcontroller specifically designed and built for controlling multi-copters of 2, 3, 4, 6, or 8 propellers. It is available for purchase separate from any UAV regardless of manufacturer or specific type. The PixHawk uses pulse-width modulation to control the rate at which the motors spin at. Pulse width modulation is a common electrical method to control motors by changing the electric wave sent to the motor, thereby changing how fast it will be induced to spin. At zero modulation, the signal is unchanged and the motor experiences full power from the controller.

Once we decided that we were going to run our commands to the PixHawk from onboard the IRIS Quad-rotor, this gives us where to start on developing the Air Vehicle. The IRIS is tasked with carrying the Raspberry Pi and a camera to image the marker on the Ground Vehicle. This requirement means that the solution must be lightweight and out of the way of any of the other components or the rotors.

The Raspberry Pi is mounted in a case on the nose of the IRIS. The case was purchased from a commercial vendor and is acrylic. It has holes for cables to connect to the GPIO header, the camera ribbon

connector, the video ribbon connector, in addition to the external port cluster. It is two pieces, a bottom, where the Pi board clips into place and has walls that make it deep enough to house the connection ports. The other part is a lid that snap-fits into place on the bottom.

The camera is on the bottom screwed to an acrylic sheet that is mounted on the bottom the IRIS. The sheet is larger than the camera board so a portion of material was removed from the bottom of the IRIS body. This was done to allow for the board to be located more up and inside the body of the vehicle while still easily allowing it to be connected to the Raspberry Pi on the exterior. The acrylic sheet is fixed in place with plastic ties that loop over it and through to the inside of the body. The ties form a loop and are tight but have some elasticity and the camera feels very secure and resists any attempts to move it by hand.

In the figure below you can see the position of the Raspberry Pi and its connections to the PixHawk inside the body of the IRIS. The rotor blades have been removed here for working in the lab. There are two methods available to connect data between the Pi and the IRIS, USB and UART Serial. The USB connection is the black cable plugged into the USB port on the Pi and that leads to the USB port on the PixHawk.

The UART connection travels over a cable that was made to splice together three wires salvaged from a junk device that have the correct end connection for the GPIO header. The wires carry signals for Transmit-Send, Transmit-Receive and Ground. These wires were soldered to a spare 6 pin connector that the PixHawk uses. The PixHawk has a port that has functionality for 2 serial ports. It is labeled “Serial 4/5” in documentation and on the PixHawk itself. The fabricated wire uses the serial 4 port and leaves the serial 5 port connected to nothing. The unused serial 5 wires were looped and taped to the bundle so they can be easily identified and used in the future if the need arises.

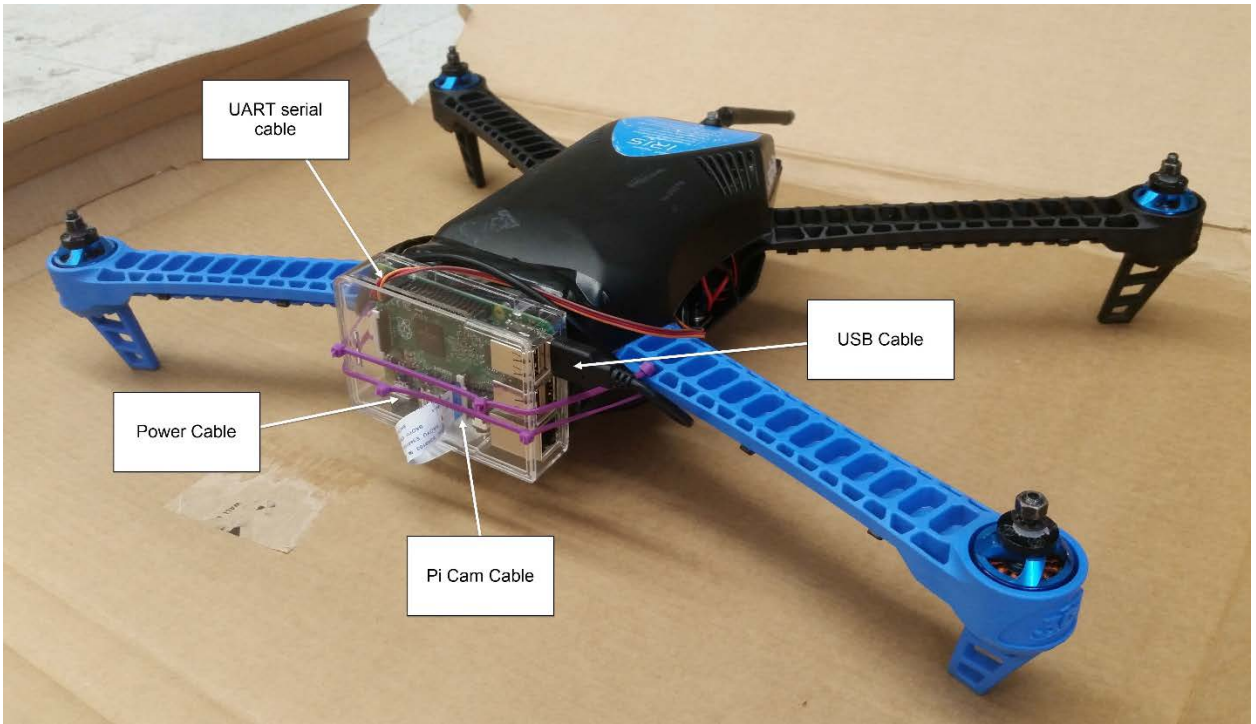


Figure 1: Iris Quadrotor front view

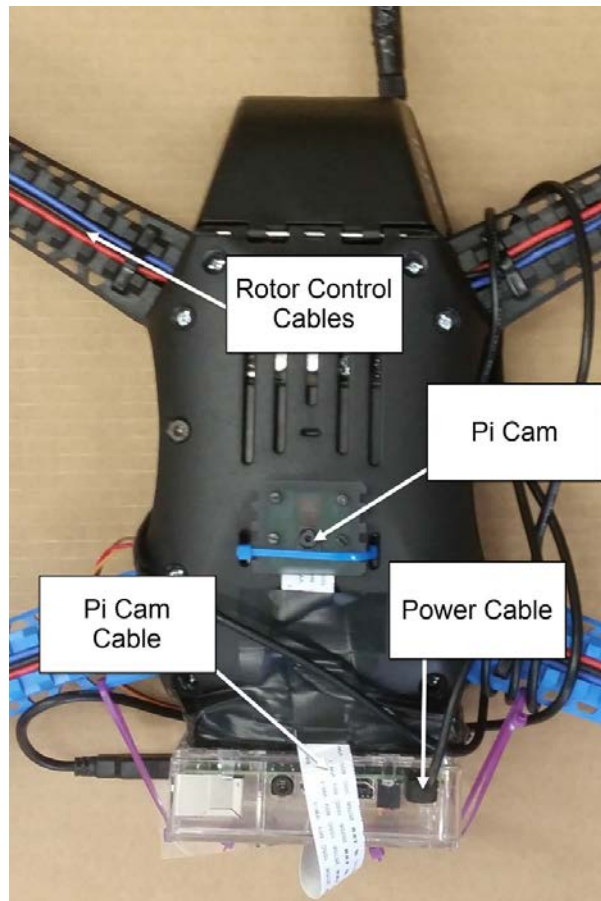


Figure 2: Iris quadrotor bottom view

2.3 Battery

Mounting and operating the Raspberry Pi onboard the IRIS presented a problem of where it is to be powered from. There are essentially two options to solve this issue. The first is to add its own power supply and mount this onboard as well. The other is to link it into the battery already in use by the IRIS.

We will first examine how much flight time we can expect to lose by powering the Raspberry Pi on the same battery as the IRIS. The Raspberry Pi, according to its manufacturer draws at maximum approximately 1200 mAh. [8] This includes any peripherals that may be connected to it while running and processing data. The PixHawk draws a maximum of 2250 mAh

On the IRIS is a six-cell lithium-polymer battery rated at 3500 mAh. The manufacturer expects flight time to last between 10 and 13 minutes for an unmodified IRIS. Amp-hours is a measure of how many

hours a power source can provide that amount of amperage at. We use this number divided by the total Amperage used, averaged over the course of an hour. The amperage drawn at any given instant can vary greatly depending on what the device is doing at that time. Climbing to a higher altitude consumes more power than just hovering. [1]

We could attach an ammeter to the IRIS and then fly it while sampling the current flow out of the battery. This is unnecessarily complicated and also would require a lot of testing to produce results. Instead we can use algebra and the expected flight time for the IRIS to determine what the expected average draw for a flight is. Solving for Ah drawn, the IRIS is expected to use approximately 17Ah of power over the course of an average flight lasting 10 to 13 minutes. For the predictions here the low value of 10 minutes is used to do the calculation and comparison. This also means that the motors on the IRIS are expected to draw around 3.7Ah per motor, again this value can be higher or lower depending on variation of the flight.

Factoring in the Raspberry Pi, the current drawn from the battery increases to around 18.25 Ah for a flight. In our calculations we also assume an 80% cut off time so that we do not excessively deplete the battery. Lithium-polymer batteries are well known to become unusable if they are completely discharged. The 80% cut-off is a common rule of thumb for micro air vehicle users. Below is a table of expected flight times based off of the previously explained methods. The range shown on the table goes from the ideal case of under usage, around 16 Ah of use, to higher use, 20 Ah, and an extreme case of very high usage, 25Ah.

Current Drawn (Ah)	Flight Time w/o Extra Pack (min)	Flight Time w/ Extra Pack (min)
16	10.8	11.4
17	9.9	10.6
18	9.4	9.3
19	8.9	10

20	8.4	8.9
25	6.7	7.1

Here we see that in the ideal case we gain less than a minute of flight time by adding a battery pack specifically for the Raspberry Pi. In the worst case, it could be lower than a half of a minute of added flight time.

The next element to consider in this decision is how the weight would affect the flight of the IRIS. Intuitively, adding more weight will mean more work needs to be done by the motors to do the same operation. Thus further reducing any gains in flight time by having an additional separate battery pack for the Raspberry Pi. The recommended safe payload for the IRIS is 425 grams. The Raspberry Pi, it's case, wiring and connected camera add about 235g of mass to the IRIS. Six AA alkaline batteries, a holder, and wiring would add a further 190 grams bringing the total added mass up to 425g. This is the limit of the recommended payload so any other mass addition would be expected to reduce flight time.

Given the tradeoff between a marginal amount of actual flight time added to any single flight for the cost of added equipment and payload usage, it was decided to keep it simpler and lighter and go without an extra battery pack and to link in the Raspberry Pi to the IRIS's extant power supply.

2.4 MAVLink

MAVLink is a Micro Air Vehicle Communication Protocol. This protocol is a header-only message marshalling library for micro-air vehicles. Functionally, this is sets of code that, when integrated into a program, allow that program to send and receive messages to the microcontroller on an UAV. This kind of program is commonly called a ground station as the most common are operated on the ground while the UAV is in light in the area. Some of these messages are sensor information from the UAV to the ground station and some others are navigation or waypoint commands to the microcontroller for the UAV. Some others are for configuring setting on the UAV microcontroller.

The list of specific messages is long and detailed which makes it a very versatile protocol. It is also developed closely with the community that uses it, the tech-savvy UAV hobbyist. It is also extensively used with common UAV microcontrollers, such as, P4X, PixHawk, APM and AR.Drone. These UAV controllers

have autopilot systems that can be updated by way of messages communicated with the MAVLink Protocol. [7]

2.5 Code Construction

The code as a whole follows a simple loop. The camera takes a photo; that photo is processed by OpenCV, as above; the program generates a waypoint; that waypoint is run through a Kalman filter to reduce noise; then it is sent to MAVLink onboard the Pixhawk; and then the process repeats. As of now, that process takes about a second, but will run slower on the RaspberryPi. We outline each step below

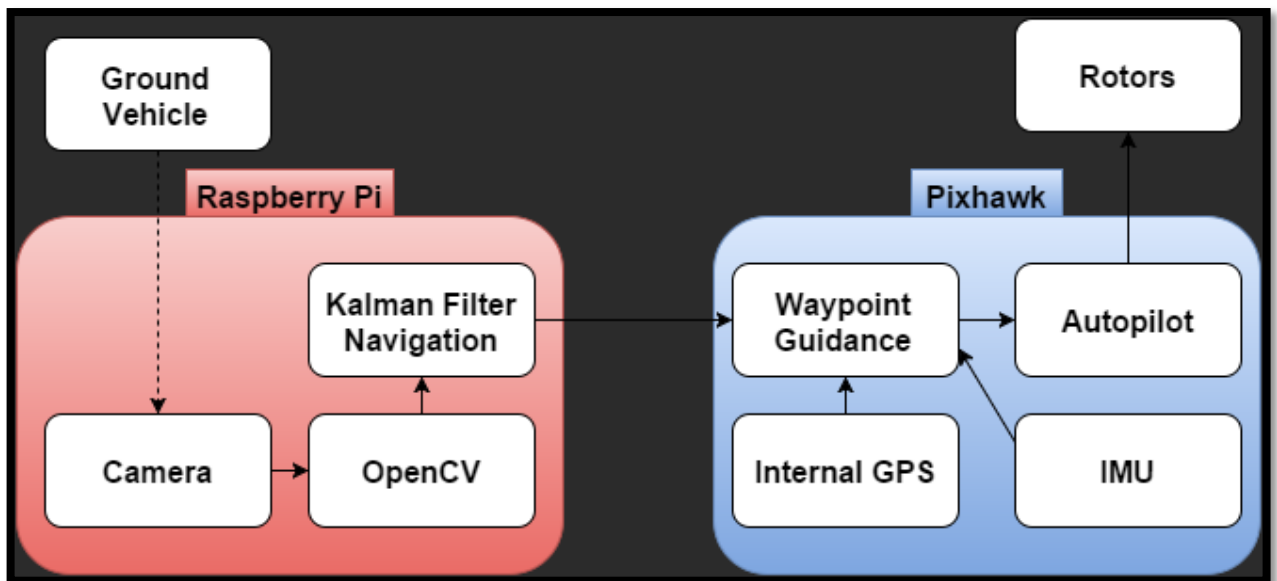


Figure 3: Planned code flow

2.6 Object Detection

We began working on the object detection software from the very beginning. Our object recognition is programmed in C++ using the OpenCV libraries. We also did the bulk of our programming on a Linux machine, as the Raspberry Pi runs a Linux based operating system. None of the project members had experience with programming, so our first step was learning C++. There are several good tutorials online

for learning the basics. Luckily, OpenCV programming is mostly self-contained, so only the basics of C++ are necessary, as long as you have an on hand reference.

The next step was installing the OpenCV libraries, and insuring they ran correctly. The instructions were fairly clear, so installation was not an issue. However, when compiling with CMake, we had some issues with paths. We attempted to run some basic sample code, that when run correctly would display a chosen image. All C++ code includes dependencies in the top of the document, and if these are pathed incorrectly the code will fail to compile.

We then did research into similar applications to our own. This is when we found the SURF detection algorithm. OpenCV's website has a large list of tutorials for different applications. We found a demonstration of finding a known object in a scene. [3] With some modifications, this became the basis for our code. This code, if given a target image and another image with that target in it, would identify the target in the scene, and highlight it. The dots in both images, with the lines drawn between them, are the points of interest as found by the SURF algorithm.

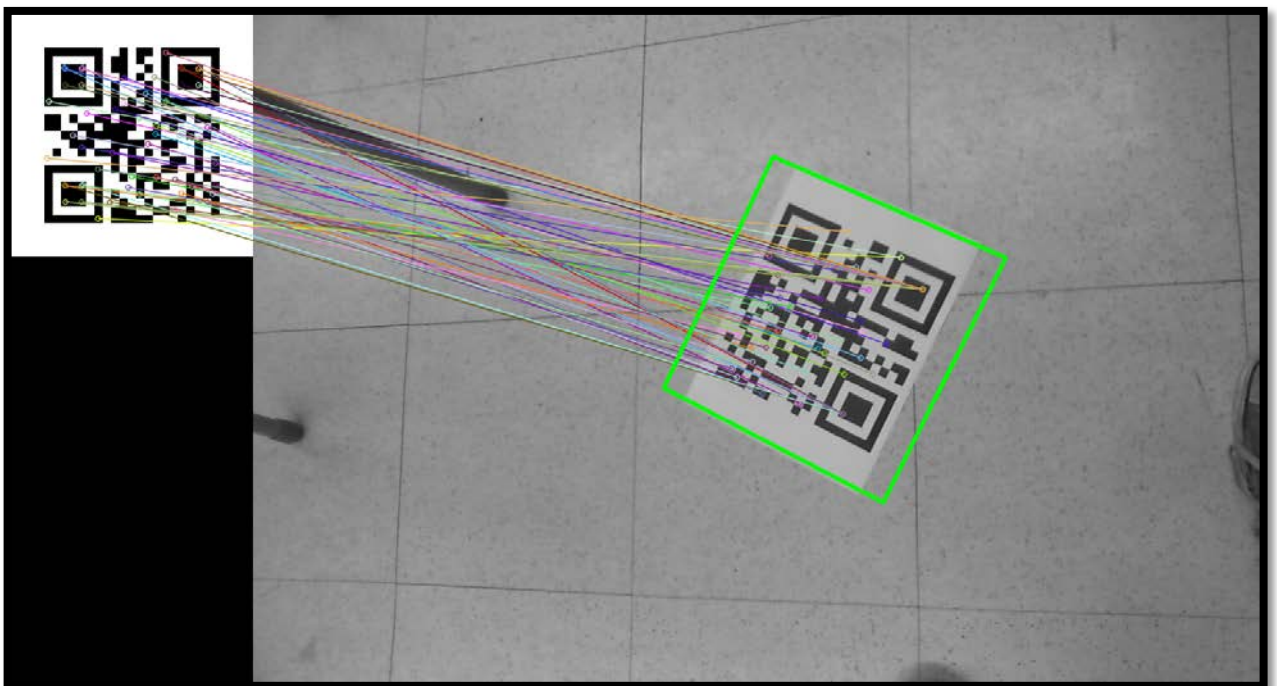


Figure 4: Object detection locating the target in a scene

The target here being the QR code displayed. We went through several different target images before we decided on the QR code. Some of our ideas included a target, or the symbol commonly seen on crash test dummies. We settled on a QR code because the SURF detector works best with corners, and with contrast between light and dark. A QR code has more corners, and a more distinct shape than our other options.

This code, as is, was obviously not going to function as part of our navigation system. We did not need the highlights, or indeed the final image at all. We needed the pixel locations of those points of interest, and very little else. We edited the code to output a matrix of those points, and we made the code delete the image as soon as the SURF was done running. A significant period of time was also spent making the code more controllable. In the end, we made the code run continuously, until given a command, so that it could feed navigation data more smoothly into the next part of the process.

The next step was optimization. This section of the code is very processor intensive, and so took a great deal of time for each iteration. We will list some of the steps we took to improve this. We had the RaspberryPi boot directly to a command line, to avoid eating up processing time with graphics. We also removed a great deal of superfluous code, much like the image displays we showed above. The search for areas we could optimize was ongoing for much of the project.

2.7 Navigation

The series of points needs to be converted into a single useable point. The code takes the mean of all the points, and that is passed on to the next step. It is important to note that the algorithm will still draw points even if the target is not onscreen. In this case, the points will be scattered and random, but it will always at least attempt a match. To this end, the program also takes the average deviation from the mean of the points. The more clumped they are, the more “sure” the algorithm is that the object is found. If that deviation goes above a certain point, the program assumes that it does not in fact see the target at all. In this case, it goes into search mode, as described in the code overview.

Our quad rotor, of course, cannot be given movement commands in pixels. We needed the output in meters, which could be converted geometrically. It is important to remember that cameras function with a pinhole system, so the distance from the center of an image correlates to an angle, not a length. Those conversions are easily found, because they are two of the most commonly quoted specifications of cameras, field of view and resolution. We would also know the distance to the plane the object is in, because of the onboard altimeters. The conversion is as follows:

$$x \equiv \text{horizontal distance from center (meters)}$$

$x_p \equiv$ horizontal distance from center (pixels)

$R \equiv$ horizontal resolution (pixels)

$\theta \equiv$ horizontal field of view (degrees)

$h \equiv$ distance from plane (meters)

$$x = h \times \tan\left(\frac{x_p}{R} \times \theta\right)$$

2.8 Kalman Filter

Since the onboard camera only can provide the position data of the ground vehicle over a unit time, a discrete-time motion simulation model is applied to simulate the object motion. This model assumes that the object executes a straight constant speed during the unit time of the model. And the average speed is calculate from the difference of the positions during the unit time. The Kalman filter is designed based on this discrete-time motion simulation model. [6] This Kalman Filter is designed to have four states: \mathbf{X}_x , \mathbf{X}_y , $\dot{\mathbf{X}}_x$, $\dot{\mathbf{X}}_y$. The first two state are the position data in x-axis and y-axis, and the last two states are the average speed data in x-axis and y-axis.

Here is the basic Kalman filter design:

The state estimation without Kalman Filter:

$$\hat{\mathbf{X}}(n^-) = A\hat{\mathbf{X}}(n)$$

Update state estimation (The state estimation with Kalman Filter):

$$\hat{\mathbf{X}}(n+1) = \hat{\mathbf{X}}(n^-) + K(n+1)(z(n+1) - C\hat{\mathbf{X}}(n^-))$$

In these two equations:

A and C are transfer matrix:

$$A = \begin{bmatrix} 1 & 0 & T_s & 0 \\ 0 & 1 & 0 & T_s \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$C = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

$T_s = 3.7$ seconds is the unit update time for every round that all programs runs in Raspberry Pi

z is the measurement of states come from onboard camera.

K is the Kalman Gain:

$$K(n+1) = P(n+1)C^T R(n+1)^{-1}$$

$P(n+1)$ is the update state covariance:

$$P(n+1) = [(AP(n)A^T + Q(n+1))^{-1} + C^T R(n+1)^{-1} C]^{-1}$$

Where $P(n)$ is the previous state covariance with the beginning that

$$P(0) = E((z(0) - \bar{z}(0))(z(0) - \bar{z}(0))^T)$$

For all equation listed, Q is the covariance of process noise and R is the covariance of measurement noise. Since the ground vehicle only can make the pivot turning and the acceleration is an impulse acceleration, most of the ground vehicle's motion is a straight constant speed motion. The process noise, the difference of real position and predict position, that results from the difference between the real time velocity and discrete time average velocity estimation, is negligible $Q(n+1) \approx 0$. And for the measurement noise:

$$R = \begin{bmatrix} R_1 & 0 \\ 0 & R_2 \end{bmatrix}$$

Where R_1 is the covariance of measurement noise on the x-axis and R_2 is the covariance of measurement noise on the y-axis. Both of them are constant and it is found by the a experiment measures the distance between the UAV and the ground vehicle in x-axis and y-axis by hands and take that value as the ture value $X(n)_{ture}$ and the distance measured by camera as the measurement value $\underline{X}(n)_{detect}$. Then the measurement noise $v(n)$ will be calculate as follow:

$$\underline{X}(n)_{detect} - X(n)_{ture} = v(n)$$

And we take the average measurement noise to calculate static covariance of measurement noise R_1 and R_2

$$R_1 = \bar{v}_x(n)^2$$

$$R_2 = \bar{v}_y(n)^2$$

3 System Development

3.1 Ground Vehicle

There are two options to build the connection between the ground vehicle and the ground station. One is building a Bluetooth connection between the ground vehicle and the ground station. A Bluetooth module is been put on the ground vehicle and achieve the wireless connection with the ground station computer. However, the remote range is limited since the small range that Bluetooth wireless connection can provide. Another method is building a Wi-Fi connection between the ground vehicle and the ground station. [4] We use EZ-B v4 Wi-Fi Robot Controller instead of the Bluetooth module and install it on the ground vehicle and remote the ground vehicle through a supplementary program EZ-builder. EZ-builder provides many options of control methods. We decide to use joystick to control the ground vehicle.

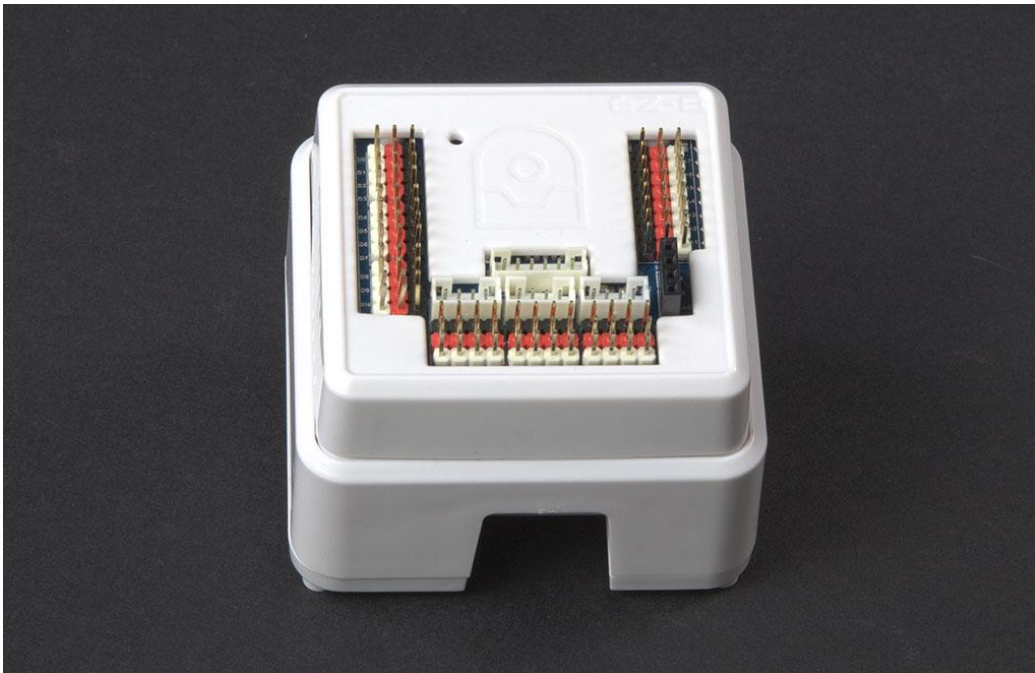


Figure 5: EZ-B v4 Wi-Fi Robot Controller (ezrobot.com)

3.2 IRIS Quad Copter

The hardware integration of the Raspberry Pi with the Iris is complete. What is meant by hardware is that the Pi is physically mounted onto the Iris in a way that is secure and is connected to the Iris or the

PixHawk in a way that is functional or should be functional, should someone attempt to use it in the future. The way that we know that features work is through testing in the lab with multi-meters and other instruments. This prodding of various wires produced no suspicious results so without symptoms of any remaining issues, we must conclude that it works as intended. The Iris flies with the Pi loaded onto it under manual control in similar ways to it unloaded.

3.3 Code Construction

The object detection code was very robust at the end. It detected objects, provided the relative locations of those objects, and had failsafe procedures for non-detection and for malfunctions. The premise of the design was simple, the IRIS was to be powered up, and then given a manual command to switch control to the autopilot. While we never achieved communication with the pixhawk, all of these functions were created.

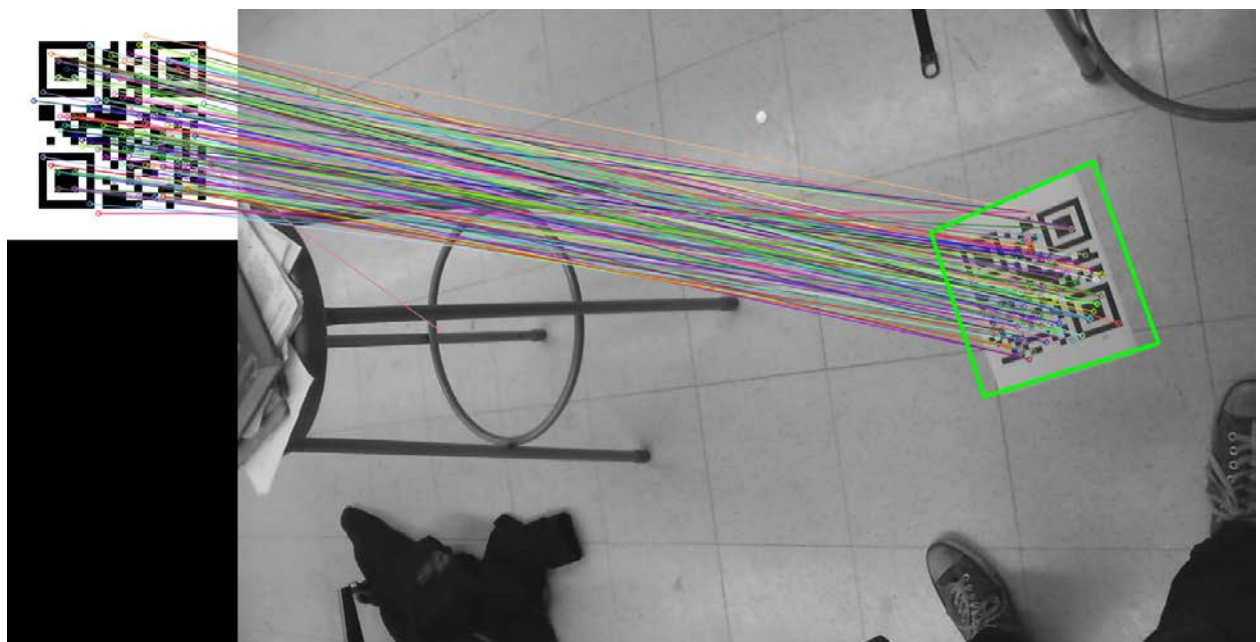


Figure 6: Successful object detection

The hurdle that prevented us from communicating with the pixhawk was incorrect references to the MavLink libraries or the incorrect installation of those libraries. No matter how we changed the code, one command or another could not be found. This occurred both with an installer and with manual installation of the libraries.

The detection feature was exactly as described in the methodology. OpenCV includes the SURF algorithms, which were used to identify the target objects. This part of the code outputs either two pixel coordinates, or a not found command.

The next part of the code was a translation between coordinate systems. The first system is the camera's view, with positive x being "up", positive y being "right", and z being ahead out of the camera. This reference frame rotates in 3 dimensions, so the translation is variable in time. The second reference frame is centered at the camera, but has z fixed straight down. In this frame of reference, the commands are simple 2D commands, as altitude is fixed.

The non-detection protocol outputs a simple series of 4 commands: [1, 0] [0, 1] [-1, 0] and then [0, -1]. This command has no override feature and so will be queued behind any other commands. The typical direction commands do feature an override. The effect of those two choices in tandem is that the quadroter, when lost, should fly to the last known location of the target, and then fly in a square until the target is found again.

The malfunction procedure was to set up a fence. When the quadroter leaves a specified boundary zone, this command gives control over to a program native to the pixhawk that returns the quadroter to its launch point. This of course only protects against failures in our own code, not in the pixhawk or IRIS themselves. In other emergency situations, manual control can be reinstated at any time from the remote controller.

The external shell code, as well as any parts of the code relying on mavlink only work in theory. Because we could not achieve communication between our raspberry pi and the pixhawk, many of those references caused halting (the code is designed to shut itself down if its inputs, such as the camera and the pixhawk, cannot be found). The shell code included commands that created and maintained the uplink, and navigated between functions of the code.

3.4 Kalman Filter

The simulation of Kalman filter is conducted on Matlab. It is designed to generate a series of coordinate data on x-axis and y-axis to form a series of coordinate of a straight line with constant time interval and constant distance interval on x-axis and y-axis. Then a normal distributed random with the limitation of maximum and minimum is added to the original data to produce the noise. And finally, the average absolute value of the random is taken as the static measurement noise and put that value into Kalman filter to test if the Kalman filter is functional.

4 Results and Discussion

4.1 Ground Vehicle

After we tested the remote control of the ground vehicle using both Bluetooth wireless connection and Wi-Fi wireless connection. The iRobot Create can achieve straight constant speed motion. But when it makes turn, it always stops first and two wheel with the motor runs at same speed with opposite direction to make a turning with respect to its own center, a pivot turning.

The reasons for the ground vehicle only can make pivot turning is that iRobot Create was preset that when it makes turn, it always stops and use the two wheel motor runs at same speed with opposite direct. It can only recognized the biggest movement of joystick from its center position. This limitation makes the iRobot create only can execute one order of movement direction.

4.2 IRIS Quad Copter

The construction of the Iris and its mounted Raspberry Pi went well. The challenges that we ran into were not ones we could not look up the information to solve it or figure out what to do in some other fashion. This was mostly when looking at the PixHawk and wiring of the unit.

Removing the previous group's hardware and mounting was easy but needed to be done carefully as to not rip wires or leave waste tape attached all over. We tried to leave parts in a configuration that allowed easy access to the interior of the body.

Stripping wire and soldering connections or ends was a basic process and took time to do carefully as some of the wires are quite small. This was greatly aided by a small vice in the lab. This vice was used with some alligator clips to hold connections together for soldering. All solder connections are wrapped in electrical tape rated for light use.

Assembling the Raspberry Pi was simple enough, if not time consuming. This entailed loading the Operating System, basic testing it, and then finally mounting it inside its case with all cables connected and then mounting the whole apparatus to the body of the Iris. It could be located anywhere on the body, our choice of the front kept it from impacting the ground upon landing and also kept the running of wires to the interior out of the way of any other parts, especially the rotor blades.

The last thing to comment on in regards to the Iris may not be necessary for this report but merits mentioning: We are happy to work alongside our colleagues in the lab space provided, but it was frustrating to go to work on the Iris and to find it missing parts a few different times. Sharing is good and we want to help our colleagues in other groups but when borrowed parts were returned, some important bits were

missing. The only things worth mentioning here are that when the bottom cover of the body was returned, none of the hardware to mount it back to the body was with it. All new screws of varying sizes and type had to be sourced and purchased because of this. And for some reason, the side portions of this bottom cover of the body were removed and presumed to be discarded.

4.3 Code Construction

The navigation code was the source of several interesting successes and several points of failure. While several key parts worked very well, the code had a number of bugs that prevented the project from being flight tested.

This part of the project represented a large part of our time on the project, which was compounded by our lack of background in computer science. While we did learn the basics of the C++ language, as well as the two extra libraries we needed, a lot of the “good practice” that is taught in CS courses was lost on us. Reviewing our build files, we found them to be poorly structured and organized, a fact to which we attribute the amount of time it took to overcome each new bug in the system. It did not help our progress that much of our progress had to be made with guess and check methods. While there was a lot of documentation on OpenCV and Mavlink, there was significantly less on their integration, and none on doing so on such a small processor. We were simply not prepared for the amount or the severity of bugs in our system. In the future, I would recommend at least one team member with a background in computer science, as this would make progress much faster.

In spite of that, the code is nearly in a final working state. We do not know if the section that communicates with the Pixhawk is written correctly, because of the libraries, but we feel confident that it is. It was carefully pieced together from existing communication code provided by the makers of MavLink, so we have no reason to suspect that it is incorrect. Though our skills with coding were lackluster, the theory behind our code was based on principles we were far more familiar with. The guidance laws, coordinate translations, and as we’ll talk about later, Kalman filtering, were all shown to be sound.

The camera processing code was one of the most interesting and successful parts of the process. While the SURF algorithm is difficult to conceptualize, its implementation went well. And after reading its documentation, we created an ideal target image. We had to go to great lengths to make the algorithm fail to find its target at that point. The processing speed of this algorithm however, left room to be desired. In the future, I would recommend a further parsing down of the program, as well as any superfluous functions running on the processor. We did a great deal in this respect, but more is necessary. Unfortunately, there is no faster alternative to SURF, as it is already a vast improvement over its predecessor in terms of processing speed. Storing images at lower resolution is also infeasible, because the resolution directly affects the efficacy of the image detection.

4.4 Kalman Filter

The plot of test result shows below:

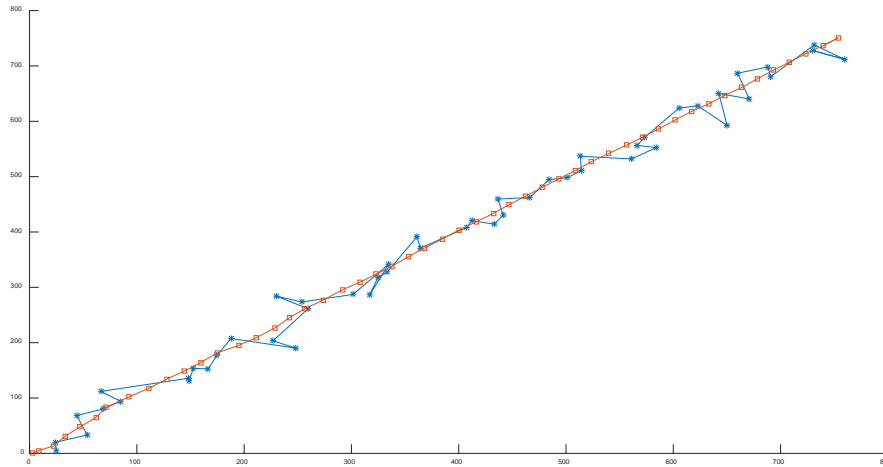


Figure 7: Matlab output of the matlab test of the kalman filter

The blue segment line is the plot of the path that detected with measurement noise and the red segment line is the plot of the path detected with measurement noise and processed by Kalman filter. This proves that the Kalman filter is functional on filtering the measurement noise.

This Kalman filter can only apply to the motion that execute the straight constant speed motion during the unit time. This limitation results from that our Kalman filter design cannot filter the process noise, which is the difference of discrete-time average speed estimation within the unit time and the real instant speed and its change within the unit time, since the lacking of detection of real instant speed of the ground vehicle. However, the process noise can be reduced significantly by increase the processing frequency of the pictures from the onboard camera and generate the position data of the ground vehicle faster.

5 Conclusions

Our project fulfilled many of our goals, though it did not make it to a final stage. We never had the UAV in flight following the ground vehicle. Because of this, we never truly had an opportunity to test the efficacy of what we had created.

The pieces that we could test functioned well. The physical integration of all of our mechanical parts was complete. Power breakouts, data linkups (both hard wired and wireless), and peripherals were all shown to be functional. Our solutions to these mechanical problems were simple and lightweight. The body was damaged somewhat while being transferred between project groups but this did not impede the attachment of the camera or the Raspberry Pi. The ground vehicle could be controlled wirelessly, as could the UAV in case of emergency. The power breakout to the Raspberry Pi allowed it to function fully with few modifications to the circuitry inside the IRIS. All in all, the mechanical aspects of the project were highly functional.

The digital aspects of our project were not as functional. The Kalman filter was completely coded and works well, but it was tied up in development for a long period of time. The uplink between the PixHawk and the Raspberry Pi never bore fruit, despite the fact that the link had been physically established. The object detection also ran far too slowly for accurate navigation to be achieved. Three and a half seconds is enough time for an object to leave the field of view entirely. Even if uplink had been achieved, this likely would have impeded the function of the system a great deal.

Even so, some aspects of the coding looked very promising. The functional object detection did give us a strong base to work off of, even if it did have some issues. The Kalman filter linked up to that code in a very intuitive way. We believe our design for the flow of the code was the most efficient possible. The coding certainly needs more time put into it, but is not conceptually flawed.

In sum, the project was very near a functional state at the end. The point of failure was the integration of MavLink in order to communicate commands, as well as a lack of efficiency in object processing times. Nearly every other system was functional and complete.

6 Bibliography

- [1] 3DRobotics. (2014). *2014 Iris*. Retrieved from 3DRobotics Homepage: <https://3drobotics.com/kb/2014-iris/>
- [2] Bay, H., Tuytelaars, T., & Van Gool, L. (2006). *Speeded-Up Robust Features*. Retrieved from http://www.vision.ee.ethz.ch/en/publications/papers/articles/eth_biwi_00517.pdf
- [3] Eruhimov, V. (2015, December 17). *Detection of planar objects*. Retrieved from Open CV tutorials: http://docs.opencv.org/2.4/doc/tutorials/features2d/detection_of_planar_objects/detection_of_planar_objects.html#detectionofplanarobjects
- [4] ezrobot. (2015). *iRobot Roomba*. Retrieved from ezrobot: <https://www.ez-robot.com/Tutorials/Hardware.aspx?id=21>
- [5] ItSeez. (2008). *About OpenCV*. Retrieved from ItSeez: <http://itseez.com/OpenCV/>
- [6] MathWorks. (2015). *State Estimation Using Time-Varying Kalman Filter*. Retrieved from MathWorks Documentation: <http://www.mathworks.com/help/control/examples/state-estimation-using-time-varying-kalman-filter.html>
- [7] QGroundControl. (1999). *MAVLink Getting Started*. Retrieved from QGroundControl: <http://www.qgroundcontrol.org/mavlink/start>
- [8] Raspberry Pi Foundation. (2008). *Raspberry Pi Homepage*. Retrieved from Raspberry Pi: <https://www.raspberrypi.org/>