# Autonomous Quadrotor Navigation by Detecting Vanishing Points in Indoor Environments

by

Nikhilesh Ravishankar

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved July 2017 by the
Graduate Supervisory Committee:

Armando A. Rodriguez, Chair
Konstantinos Tsakalis
Spring M. Berman

ARIZONA STATE UNIVERSITY

May 2018

## ABSTRACT

Toward the ambitious long-term goal of a fleet of cooperating *Flexible Autonomous Machines operating in an uncertain Environment (FAME)*, this thesis addresses various perception and control problems in autonomous aerial robotics. The objective of this thesis is to motivate the use of perspective cues in single images for the planning and control of quadrotors in indoor environments. In addition to providing empirical evidence for the abundance of such cues in indoor environments, the usefulness of these perspective cues is demonstrated by designing a control algorithm for navigating a quadrotor in indoor corridors. An Extended Kalman Filter (EKF), implemented on top of the vision algorithm, serves to improve the robustness of the algorithm to changing illumination.

In this thesis, vanishing points are the perspective cues used to control and navigate a quadrotor in an indoor corridor. Indoor corridors are an abundant source of parallel lines. As a consequence of perspective projection, parallel lines in the real world, that are not parallel to the plane of the camera, intersect at a point in the image. This point is called the vanishing point of the image. The vanishing point is sensitive to the lateral motion of the camera and hence the quadrotor. By tracking the position of the vanishing point in every image frame, the quadrotor can navigate along the center of the corridor.

Experiments are conducted using the Augmented Reality (AR) Drone 2.0. The drone is equipped with the following componenets: (1) 720p forward facing camera for vanishing point detection, (2) 240p downward facing camera, (3) Inertial Measurement Unit (IMU) for attitude control , (4) Ultrasonic sensor for estimating altitude, (5) On-board 1 GHz Processor for processing low level commands. The reliability of the vision algorithm is presented by flying the drone in indoor corridors.

*To my Parents and my Brother*

## ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. Armando A. Rodriguez, for his guidance and support throughout my graduate studies. I would also like to thank him for his kindness and tremendous patience during difficult times. I would like to thank Dr.Tsakalis and Dr.Berman for being a part of my thesis committee. I would also like to thank Zhichao Li, Jesus Aldaco, Karan Puttannaiah and Shubham Sonawani for their timely feedback on my research. Lastly, I would like to thank my parents and my brother for their constant support and patience.

TABLE OF CONTENTS

CHAPTER Page

# LIST OF TABLES

LIST OF FIGURES

# Chapter 1

# INTRODUCTION AND OVERVIEW OF WORK

## 1.1 Introduction and Motivation

Autonomous naviagation in indoor environments is an important problem in perception because of the absence of GPS information. The advent of Real Time Kinematic (RTK) and Differential Global Positioning Systems (DGPS) have made accurate outdoor localization of ground and aerial robots feasible. In order to accurately estimate the position and orientation of a robot indoors, a combimation of sensors like cameras, Light Detection and Ranging (LIDAR) sensor, gyroscopes and accelerometers, will have to be used.

In aerial robots, specifically quadrotors, it is important to select components that are light weight and consume minimal power in order to maximize flight time. For example, a miniature LIDAR typically weighs 270 grams and consumes 10W of power for operation and an additional 50-60W for mobility [43]. Comparing this with a camera, which consumes 1.5W of power for operation and 8W for mobility, it is clear why cameras are a popular choice among quadrotors. In this thesis, a single camera has been used as the primary sensor for the quadrotor's perception tasks.

The central objective of the thesis is to use the perspective cues in images of indoor environments, like corridors, to design a simple outer loop controller for the quadrotor, so that it can navigate from one end of the corridor to the other. The primary motivation for using these cues comes from nature as birds like Gannets track down fish by diving at speeds close to 24m/s by using simple visual features which allow them to estimate the time-to-contact with the water surface quickly. There is also evidence that house flies do not explicitly comprehend the 3D structure of their environment but rather use simple visual cues like optical flow for navigation [17].

The work presented here is a step toward the longer-term goal of achieving a fleet of *Flexible Autonomous Machines operating in an uncertain Environment (FAME)*. Such a fleet can involve multiple ground and air vehicles that work collaboratively to accomplish coordinated tasks. Such a fleet may be called a swarm [15]. Potential applications can include: remote sensing, mapping, intelligence gathering, intelligence-surveillance-reconnaissance (ISR), search and rescue, manufacturing, teleoperation and much more. It is this vast application arena as well as the ongoing technological revolution that continues to fuel robotic vehicle research.

The quadrotor used in the hardware demonstrations is the Augmented Reality (AR) drone 2.0. The drone proprietary software offers an inner attitude stabilization controller. The dynamics of the quadrotor are modeled using the Newton-Euler equations of motion.

This chapter attempts to provide a fairly comprehensive literature survey - one that summarizes relevant literature and how it has been used. This is then used as the basis for outlining the central contributions of the thesis.

## 1.2 Literature Survey: Robotics - State of the Field

In an effort to shed light on the state of vision based aerial robots operating in indoor environments, the following literature survey is offered.

- **Vision Based Navigation of Aerial Vehicles.**

  Cooper Bills et.al. in [18] provides a good introduction to the notion of perspective cues in single images and its subsequent benefit when used for navigating a co-axial helicopter and a toy quadrotor in indoor corridors and staircases. The image processing algorithm and the associated navigation strategy proposed in this paper provides the basic framework for designing an autonomous agent operating in indoor environments. Indoor environments are automatically classified into one of three categories (corridors, staircases and open rooms) using an image classifier tuned to differentiate among various perspective cues. Images in each of these categories has its own unique perspective cue. The MAV (Micro Aerial Vehicle) is programmed to have a unique control behavior for each of these categories. Parallel lines in the real world intersect at a point in an image due to perspective projection (an artifact when going from 3D to 2D). This point is called the Vanishing Point. One of the main contributions of this work is to show that such vanishing points are sensitive to the lateral (side to side) motion of the camera. This information was used to control the roll and the yaw of the quadrotor as it traversed indoor corridors and staircases at a constant forward speed and height.

Yong Woo-Seo in [27] offers a solution to one of the limitations of the work summarized above. Frame by Frame detection methods are a function of the number of lines detected every frame. As such, this method is sensitive to noise such as changes in the illumination and slight shakes in the camera. In order to improve the robustness of the algorithm and achieve smooth tracking, an Extended Kalman Filter (EKF) is proposed. This state estimation filter, with its non-linear observation model, is used to demonstrate the improved tracking of the vanishing point on highways. The author highlights the reliablity of the state estimation filter in varying illumination conditions. The vanishing points were subsequently used to estimate the instantaneous driving direction and also narrow the search for other vehicles driving on the road.

Davide Scaramuzza in [29],[30] provides a general framework for designing a visual odometry pipeline for autonomous systems. The techniques summarized and presented in this paper offer a glimpse of how a complete end-to-end vision based system (pixels to motor control) can be created. Some of the early accomplishments in the field of computer vision such as fast feature detection and outlier rejection along with some of the intial setbacks concerning featutre correspondence and computation burden are clearly outlined in this paper. These papers served as a guide book, offering tips and tricks to design custom image processing pipelines for specific applications.

- **Quadrotor Modeling.** Robert Mahony, Vijay Kumar and Peter Corke in [31] present a comprehensive literature on the model, design and control of quadrotors. They clearly explain the size, weight and power constraints of selecting the various components of the quadrotor. Starting with the Newton-Euler dynamic equations of motion, the paper describes how to perform frame of reference transformations, design simple hover controllers and generate minimum snap trajectories. The material presented in this paper provides a general framework for modeling quadrotors of various sizes and analyzing the effect of various components on the performance of the quadrotor.

- **Vision Algorithms.** The design and fine tuning of the vision algorithm to detect vanishing points in indoor corridors is one of the primary contributions of this thesis. The algorithm's parameters have been tuned to work with the images provided by the forward facing camera of the AR Drone. Thus the camera calibration matrix and the image resolution play an integral part in tuning the parameters of the vision algorithm. Once tuned, the algorithm is general enough to work in any indoor corridor as long the noise (illumination condition) is approximately constant. Relevant theory is presented within [9].

The basic image processsing pipeline is as follows: The RGB image from the camera is converted into a grayscale image. Horizontal and Vertical image gradients are extracted from the grayscale image. Image gradients represent edges in the image (regions where there is significant variation in intensity among neighboring pixels). Each pixel in the gradient is thresholded so as to extract only the pixels relevant for our application. Once the relevant binary image of the gradient directions has been formed, Hough transform is used to extract lines present in the image. Hough transform essentially takes the coordinates of each pixel in the binary image and draws hypothetical lines having different orientations with respect to the origin. The number of pixels that fall under a line of given orientation and distance from the origin is computed. A 2-D accumulator matrix is formed with the various distances from the origin as the rows of a matrix and the various line orientations as the columns of a matrix. Each pixel votes for a distance and orientation pair. All tuples greater than a specified threshold (minimum number of pixels) will be used generate the dominant lines in the image.The intersection points of these lines is calculated by taking a vector cross product of all non-collinear lines. The median of the various intersection points obtained is defined to be the vanishing point of the image. This process is repeated for every image frame.

The open source computer vision library OpenCV is greatly used in this thesis. Useful information for using this library is presented within [4].

- **State Estimation Algorithm.** In order to improve the robustness of the above mentioned vision algorithm, an Extended Kalman filter is used, essentially as a low pass filter, to eliminate noise and present reliable state estimates.

The filter assigns probabilistic weights to the previous state and the current measurement to generate the current state. If the current measurement is unreliable i.e. the covariance matrix is large, then the algorithm assigns a higer weight to the previous state in order to reflect the poor confindence in the current measurement. This scheme seems to work well for detecting vanishing points. Beacuse the lines detected in every frame are susceptible to changing light conditions, irrelevant lines frequently contribute to the calculation of the vanishing points and corrupt the estimate. The EKF resolves this by differentially weighing the quality of measurements obtained in the current frame compared to the previous frame.

- **Augmented Reality Drone 2.0.** The platform used for all the demonstrations is the Augmented Reality Drone (2.0). This drone is a popular research platform. It is affordable and encompasses a wide array of sensors. It includes a 30 FPS, 720p forward facing camera and a 60 FPS, 240p downward facing camera, an ultrasonic sensor for altitude measurement and a 3 axis gyroscope and a 3 axis accelerometer for measuring the quadrotor orientation and linear acceleration respectively. It can achieve a maximum vertical speed of 5m/s and a maximum horizontal speed of 1.66 m/s. It has an internal attitude stabilization controller and is not built for acrobatic/agile maneuvers.

## 1.3 Contributions of Work: Questions to be Addressed

Within this thesis, the following fundamental questions are addressed. The answers to these questions are important to move toward the longer-term *FAME* goal.

1. **How can visual information be extracted from the real world without modifying the environment?**

   In order to develop a general framework for visual perception, it is important to design algorithms that take advantage of the structural richness of the environment in which they are operating. Indoor environments are an abuntant source of line segments. Thus it is not necessay to implant colored and fiduciary markers in indoor environments for developing a navigation strategy. By knowing the camera calibration matrix and the coordinates of the vanishing point in indoor environments, the camera lateral dispalcement can be estimated. The entire vision algorithm is implemented in Python programming language using OpenCV library functions.

2. **How reliabile are the location estimates obtained from the visual input? Can they be improved?**

   The high sensitivity of visual features (points, lines or colored markers), used to compute location estimates, to even slight changes in illummination make any derived measurement unreliable. Although the reliability can be improved by further tuning of the parameters of the vision algorithm, it is laborious and unelegant. Designing a continous unimodal state estimation filter, like the Kalman filter, helps remove the high frequency noise thus allowing improved tracking and reliability.

3. **What are typical outer-loop objectives?**

   For the quadrotor considered in this thesis, the primary outerloop objective is to control the x, y and z position of the system so that the quadrotor can traverse an indoor corridor with a stable attitude and a constant forward velocity.

When taken collectively, the contributions of this thesis are of importance especially to those interested in conducting robotics/*FAME* research.

## 1.4 Organization of Thesis

The remainder of the thesis is organized as follows.

- Chapter 2 (page 11) presents an overview for a general *FAME* architecture describing candidate technologies (e.g. sensing, communications, computing, actuation).
- Chapter 3 (page 15) describes the image processing pipeline in detail and the algorithm used to compute the vanishing points.
- Chapter 4 (page 23) presents how to design the Extended Kalman filter, with a non-linear observation model, for state estimation.
- Chapter 5 (page 33) describes the AR Drone hardware, software and dynamic model. The design of the outer loop position controller is also detailed in this section.
- Chapter 6 (page 44) presents the AR Drone Gazebo Simulator and its features.
- Chapter 7 (page 52) describes the GPU programming framework and how they can be exploited for improving the run time of image processing algorithms.

- Chapter 8 (page 65) summarizes the thesis and presents directions for future robotics/*FAME* research. While much has been accomplished in this thesis, lots remains to be done.

- Appendix A (page 74) describes how to set up AR Drone TUM simulator from scratch. Appendix B (page 77) contains the Python files used to generate the results for this thesis

## 1.5 Summary and Conclusions

In this chapter, an overview of the work presented in this thesis and the major contributions have been provided. A central contibution of the thesis is the vast empirical evidence provided for the reliable detection of vanishing points in indoor environments and their use in developing robust navigation strategies for quadrotors.

Chapter 2

# OVERVIEW OF GENERAL FAME ARCHITECTURE AND $C^4S$ REQUIREMENTS

## 2.1 Introduction and Overview

In this chapter, a general architecture for the general *FAME* research is described. The architecture described attempts to shed light on command, control, communications, computing ($C^4$), and sensing ($S$) requirements needed to support a fleet of collaborating vehicles. Collectively, the $C^4S$ and $S$ requirements are referred to as ($C^4S$) requirements.

## 2.2 FAME Architecture and $C^4S$ Requirements

In this section, a candidate system-level architecture that can be used for a fleet of robotic vehicles[1] is described. The architecture can be visualized as shown in Figure 2.1. The architecture addresses global/central as well as local command, control, computing, communications ($C^4$), and sensing ($C^4S$) needs. Elements within the figure are now described.

- **Central Command: Global/Central Command, Control, Computing.** A global/central computer (or suite of computers) can be used to perform all of the very heavy computing requirements. This computer gathers information from a global/central (possibly distributed) suite of sensors (e.g. GPS, radar, cameras). The information gathered is used for many purposes. This includes temporal/spatial mission planning, objective adaptation, optimization, decision

[1]Here the term robotic vehicle can refer to a ground, air, space, sea or underwater vehicle.

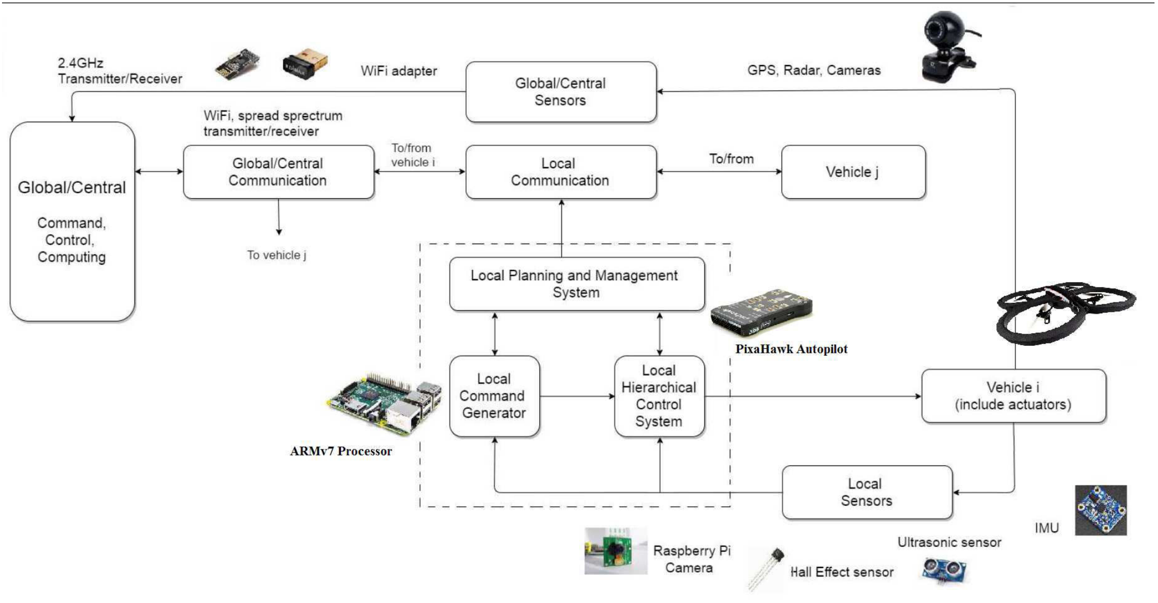


Figure 2.1: *FAME* Architecture to Accommodate Fleet of Cooperating Vehicles

making (control), information transmission/broadcasting and the generation of commands that can be issued to members of the fleet.

- **Global/Central Sensing.** In order to make global/central decisions, a suite of sensors should be available (e.g. GPS, radar, cameras). This suite provides information about the state of the fleet (or individual members) that can be used by central command.

- **Global/Central Communications.** In order to communicate with members of the fleet, a suite of communication devices must be available to central command. Such devices can include (wideband) spread spectrum transmitters/receivers, WiFi/Bluetooth adapters, etc.

- **Fleet of Vehicles.** The fleet of vehicles can consist of ground, air, space, sea or underwater vehicles. Ground vehicles can consist of semi-autonomous or autonomous robotic vehicles (e.g. differential-drive, rear-wheel drive, etc.). Here, autonomous implies that no human intervention is involved (a longer-term objective). Semi-autonomous implies that some human intervention is involved. Air vehicles can consist of quadrotors, micro/nano air vehicles, drones, other air vehicles and space vehicles. Sea vehicles can consist of a variety of surface and underwater vehicles. Within this thesis the focus is on ground vehicles (e.g. enhanced Thunder Tumbler differential-drive).

- **Local Computing.** Every vehicle in the fleet will (generally speaking) have some computing capability. Some vehicles may have more than others. Local computing here is used to address command, control, computing, planning and optimization needs for a single vehicle. The objective for the single vehicle, however, may (in general) involve multiple vehicles in the fleet (e.g. maintaining a specified formation, controlling the inter-vehicle spacing for a platoon of vehicles). Local computing can consist of a computer, microcontroller or suite of computers/microcontrollers. Within this thesis ARMv7, 1 Ghz processor is exploited for local computing. They are low-cost, well supported and easy to use.

- **Local Sensing.** Local sensing, in general, refers to sensors on individual vehicles. As such, this can involve a variety of sensors. These can include encoders, IMUs (containing accelerometers, gyroscopes, magnetometers), ultrasonic range sensors, Lidar, GPS, radar, and cameras. Within this thesis, the Quadrotor is equipped with the following sensors: 3 axis Accelerometer ( +/- 50 mg precision)

, 3 axis Gyroscope (2000 degress/sec precision), Pressure sensor (+/- 10 Pa), 40Khz Ultrasonic sensor (0 -3m range), 720p Forward facing camera (30 fps) and 240p Downward facing camera (60 fps). Lidar, GPS and radar are not used.

- **Local Communications.** Here, local communications refers to how fleet vehicles communicate with one another as well as with central command.

## 2.3 Summary and Conclusions

In this chapter, a general (candidate) *FAME* architecture for a fleet of cooperating robotic vehicles was described. Of critical importance to properly assess the utility of a *FAME* architecture is understanding the fundamental limitations imposed by its subsystems (e.g. bandwidth/dynamic, accuracy/static) [13].

Chapter 3

# IMAGE PROCESSING PIPELINE FOR VANISHING POINT DETECTION

## 3.1 Introduction and Overview

This chapter describes the image processing pipeline in detail. The variation in the quality of the computed features with image resolution is discussed. Empirical evidence for the detection of vanishing points in indoor corridors is provided. The changes in the location of the vanishing point with changes in the lateral motion of the camera i.e. the sensitivity of the vanishing point to lateral camera movement is analyzed.

## 3.2 Line Detection

Indoor corridors are an abundant source of parallel lines. Figure 3.1 depicts a typical indoor corridor. Parallel lines at the intersection of the walls and the ceilings are the lines most relevant to our application. It is the intersection of these lines that will define the vanishing point for the corridor [18] [27]. An artifact of perspective projection is that depth information is lost as the world is scaled down from 3 dimensions to 2 dimensions. In fact human vision works on the same principle and hence suffers from the same artifact [17]. It is common to observe that extremely long stretches of railway lines appear to intersect at the horizon. In a similar vein, these parallel lines in the indoor corridor intersect at a point.

We will now describe how lines are extracted from the image. The image is first smoothened by passing it through a Gaussian filter and then a Median filter (helps

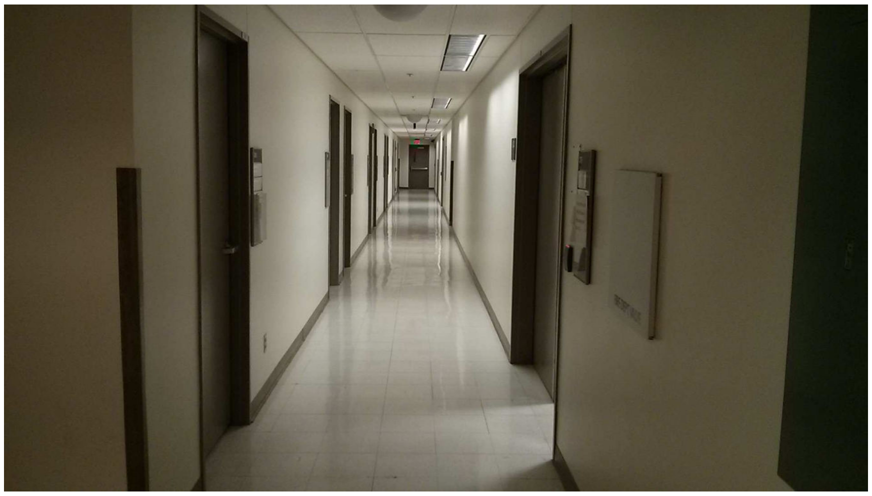


Figure 3.1: Indoor Corridor

remove salt and pepper noise). Often the size of these filters is empirically determined. The procedure [36] for finding lines in an image consists of:

1. Computing the Horizontal and Vertical Image Gradients
2. Thresholding
3. Voting in the Hough Space
4. Post-Processing in the Hough Space

In order to extract the lines of interest, the horizontal and the vertical image gradients have to be computed. This is accomplished by convolving the image with the Sobel filter. A 3 x 3 horizontal Sobel filter is given by:

$$S_X = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} \tag{3.1}$$

Similarly, a 3 x 3 vertical Sobel filter is given by:

$$S_Y = \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix} \tag{3.2}$$

The orientation of each pixel gradient is now computed by taking the inverse tangent of the ratio of the vertica image gradient and the horizontal image gradient for each pixel. Once direction of the gradient for each pixel is computed, it is thresholded so as to extract only the pixels relevant for our application. Intuitively this amounts to rejecting the responses below a certain threshold. Once the relevant binary image of the gradient directions has been formed, Hough transform is used to extract lines present in the image.

Hough transform is a popular technique for extracting shapes from images (predominantly lines and circles). We know that a line in the polar coordinate system can be expressed as

$$y = -\frac{\cos\theta}{\sin\theta} + \frac{r}{\sin\theta} \tag{3.3}$$

This can be rewritten as:

$$r = x\cos\theta + y\sin\theta \tag{3.4}$$

There are many lines that go through a given point $(x_i, y_i)$. Each line can be identified by the parameter $(r, \theta)$. Here $r$ represents the distance of the line from the origin and $\theta$ represents the orientation of the line. When we plot all these lines in the $r$ and $\theta$ plane, we get a sinusoid.This means that a point $(x_0, y_0)$ gets mapped to a sinusoid in the $(r, \theta)$ space (We consider only positive r and theta betweeen 0 and 2 $\pi$ ).This is depicted in the figure below for the point $(x_0, y_0) = (8, 6)$.

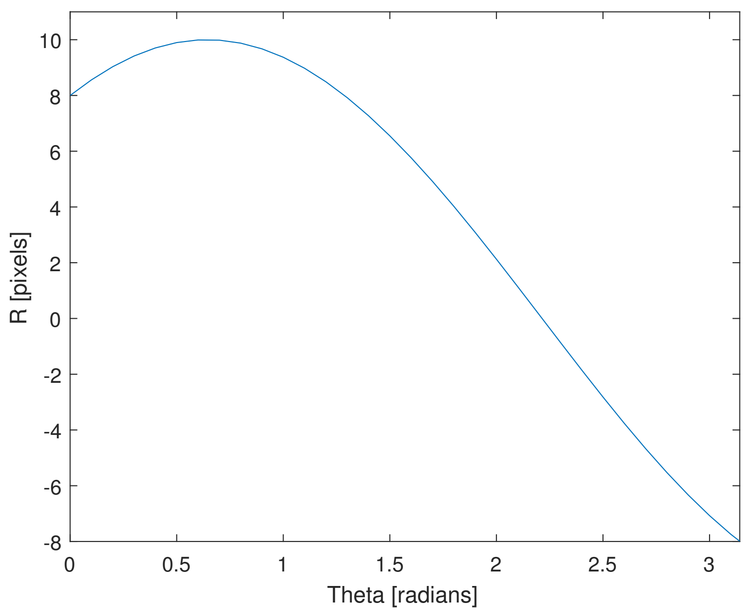


Figure 3.2: A Sinusoid in the $(r, \theta)$ Plane

If the sinusoids of two different points $(x_1, y_1)$ and $(\mathrm{x}_2, y_2)$ intersect in the $(r, \theta)$ plane, this means that the two points lie on a line given by the intersection point of the sinusoids $(r_{intersect}, \theta_{intersect})$. We can repeat this procedure for each and every pixel in the binary image. In general, the Hough transform keeps a tracks of these intersections, greater the number of intersecting sinusoids, greater is the number of points that belong to a line. We can define a threshold for the minimum number of intersections necessary to declare a line. The figure below depicts three sinusoids corresponding to three points ($(x_0, y_0) = (4, 9)$, $(x_0, y_0) = (12, 3)$, $(x_0, y_0) = (8, 6)$.)

The Hough accumulator array is a 2D matrix that is used to keep a record of all the intersecting sinusoids. The size of the matrix depends on the resolution with which the distance and the orientation of the lines in the image have to be determined. If two lines differing by 1 degree have to be resolved, then a 1 degree resolution Hough accumulator array is necessary (this translates to 360 rows in the matrix). If two

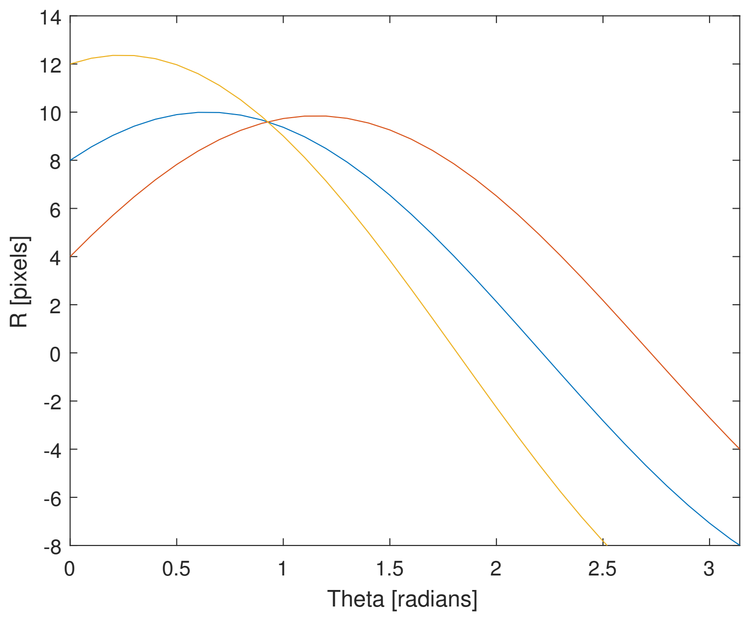


Figure 3.3: Intersection of Three Sinusoids in the $(r, \theta)$ Plane

lines differeing by 1 pixel distance from the origin have to be resolved, then a 1 pixel resolution Hough accumulator is necessary (This translates to $\sqrt{H^2 + W^2}$ columns in the matrix, where H and W are the height and width of the image respectively). The resolution of $r$ and $\theta$ must be chosen according to your application. If you can trade off accuracy for speed then you are better off choosing a lower resolution.

The domiant lines in the image correspond to the cells in the accumulator array with the most number of votes. Once the dominant lines have been identified, they are represented in the cartesian coordinate system. Lines that are collinear are discarded. The intersection points of all the non-collinear lines are determined. The median of the computed intersection points is defined to be the vanishing point of the image (median is more robust to large outliers when compared to the mean).This process is repeated for every image frame. Figure 3.4 depicts the vanishing point for an indoor corridor.

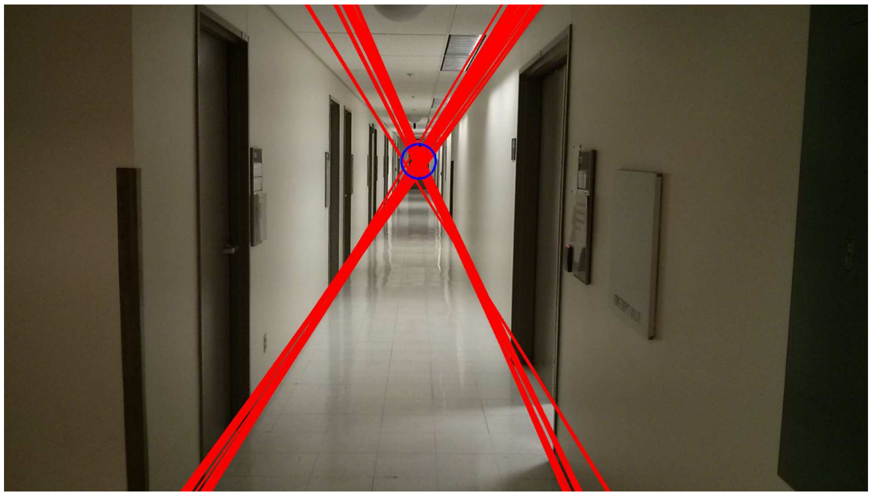


Figure 3.4: Indoor Corridor with Vanishing Point

Once the algorithm is tuned to detect a sufficient number of dominant lines, the camera can be displaced laterally to observe a proportional displacement in the location of vanishing point. The magnitude of the displacement observed is a function of the resolution of the image. For a 1.2 Mega Pixel image, we observed a **3 pixel/cm** displacement in the position of the vanishing point when the camera was moved along the width of an indoor corridor **150cm** wide.

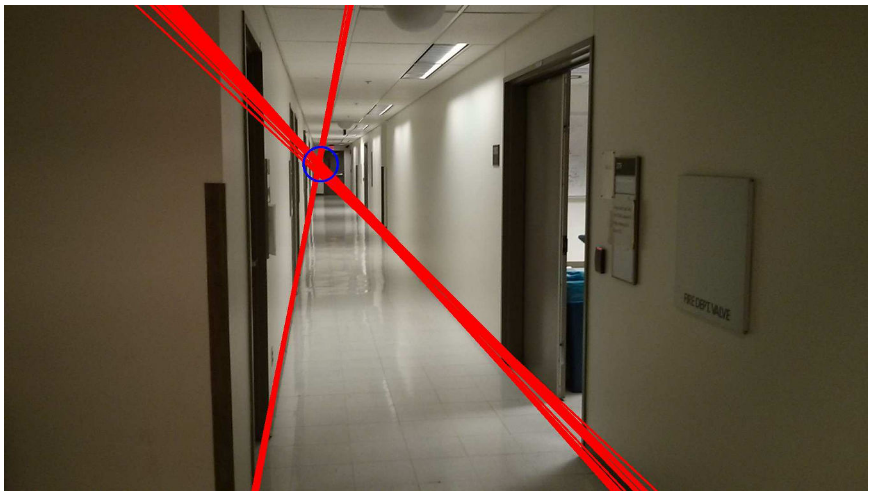


Figure 3.5: Vanishing Point Towards the Left

Note that in order to process the images quickly, the OpenCV implementation of the probabilistic Hough transform [37] randomly samples parts of the image to extract line segments. This approacah works for our application only because of the abundance of lines in indoor environments. Figures 3.5 and 3.6 depict the displacement of the vansihing point towards the left and toward the right of center of the corridor as the camera is moved towards the left and right of the center respectively.

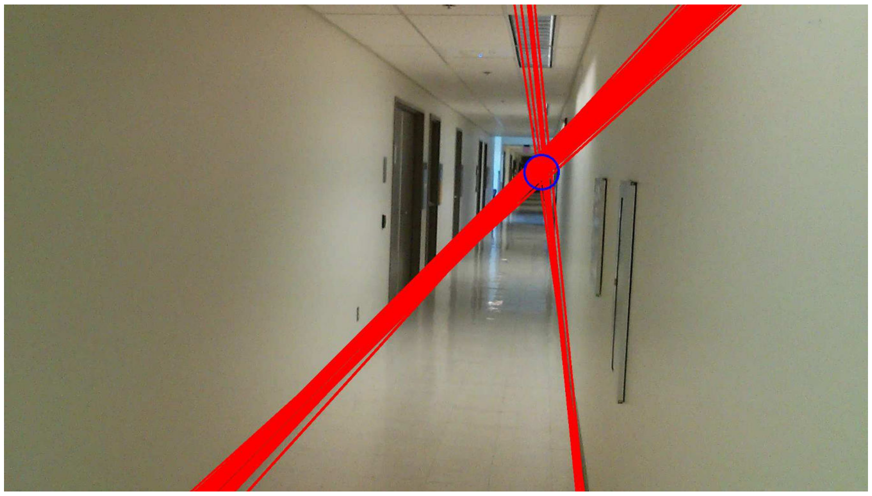


Figure 3.6: Vanishing Point Towards the Right

From the above discussion, we now know that the computation time of the Hough transorm is a function of the resolution of the input image. As the resolution of the image decreases, the computation time of the algorithm also decreases. But a decrease in image resolution will also cause a decrease in the resolution of the displacement of the vanishing point. Extremely small displacements to the left and to the right of center can no longer be differentiated. This will be discussed more thoroughly in the next chapter. Experimental tests have shown that at 0.5 Mega pixel resolution i.e. at a quarter of the full 2 Mega (1920 x 1080) pixel resolution, the displacement of the vanishing point to lateral camera movement is significant enough to be reliably detected ( 1 pixel/ cm displacement ). In order to convert this displacement in pixels to displacement in centimeters, we have to first determine the width of the corridor (in our case, the width was 150 cm). Then we would need to displace the camera laterally, along the width of the corridor. Using these reference measurements, we now have enough information to compute the displacement in centimeters. This displacement of the vanishing point from the center of the corridor helps us locate the position of the AR Drone along the Y axis. Intitally, the drone would take off from the center of the corridor and begin tracking the vanishing point. As it drifts sidewards, information aboout the displacement is fed to the position controller that reduces the drift and maintains the drone along the center of the corridor. The design of position controller will be described in Chapter 5.

## 3.3 Summary and Conclusion

This chapter has presented a description of the vision algorithm used within this thesis. The sensitivity of the vanishing point to lateral camera movement was analyzed. This algorithm is important since it is used in the subsequent chapters.

Chapter 4

# EXTENDED KALMAN FILTER DESIGN

## 4.1 Introduction and Overview

In this chapter, we are going to explore what Kalman filters are and describe how they can be used for the purpose of state estimation. We will look at the recursive algorithm for implementing a version of the Kalman filter known as Extended Kalman filte. With this state estimation technique, we will analyze the improvements in tracking the vanishing point at various image resolutions. We will also determine the algorithm run time at different image resolutions. With this information, we will be able to decide what image resolution to use for our application of navigating the drone along the center of the corridor.

## 4.2 Kalman Filters

Kalman filters are unimodal, continuous state estimation filters. They are very popular in the field of robotics. They are extensively used in tracking neighboring vehicles in self-driving cars. They are used in localization modules for the purpose of fusing global position data with local odometric data. They are one of the most popular implementations of the Bayes Filter.

Conceptually, the algorithm updates the state and uncertainty of the system probabilistically (adhering to Bayes Rule). Like all state estimation filters, the Kalman filter has a measurement cycle and a motion cycle. Under Gaussian noise assumptions and 1 dimensional state estimation, the mathematics of the state and covariance updates are easy to understand. As the number of dimensions is increased, the core intuition remains the same but the mathematics gets a little complicated. For a complete mathematical treatment of the Kalman filters, please refer [34]. Note that the Kalman filter assumes a linear measurement and a linear motion model.

An Extended Kalman filter is one in which the measurement model and/or the motion model is nonlinear.The purpose of designing an Extended Kalman filter (EKF) is to improve the robustess of the vanishing point tracking algorithm. In this thesis a 2D Extended Kalman filter with a nonlinear observation model is used to estimate the coordinates of the vansihing point from frame to frame as a quadrotor traverses from one end of the corridor to the other.

## 4.3 Measurement and Motion Model

Like traditional Kalman filters, the motion model of the Extended Kalman filter is linear in nature. The measurement models are usually designed ad hoc, to meet the requirements of the application. For the purpose of detecting vanishing points, a non-linear measurement model formulated in [27] is used. The equations below shows the prediction step and measurement step for EKF.

Prediction Step

$$\hat{x}_k = \hat{x}_{k-1}; x = [v_x, v_y] \tag{4.1}$$

$$\hat{P}_k = \hat{P}_{k-1} \tag{4.2}$$

Measurement Step

$$h(\hat{x}_k) = tan^{-1}\frac{(v_y - y_1)}{(v_x - x_1)} \tag{4.3}$$

$$\hat{y}_j = z_j - h(\hat{x}_k) \tag{4.4}$$

$$H_j = \frac{\partial(\hat{x}_k)}{\partial x} \tag{4.5}$$

$$S_j = H_j P_j H_j^T \tag{4.6}$$

$$K_j = H_j P_j H_j^T + R_j \tag{4.7}$$

$$x_k = \hat{x}_k + K_j y_j \tag{4.8}$$

$$P_j = (I - K_j H_j) P_{j-1} \tag{4.9}$$

Initially the state $x$ is assigned to be equal to the coordinates of the center of the image. The uncertainty matrix, represented by $P$, is set to a large value. In the Measurement step, the difference in the angle between the detected lines and the measured vanishing point is computed for each dominant line detected in the image. The same is done to compute the angle between the detected lines and the state of the system. The error of the measurement model is defined to be the difference between these two sets of angles. This is depicted in figure 4.1. It is this difference in the angles (the $\tan^{-1}$ computation) that introduces the non-linearity. The Jacobian of the measurement matrix is computed to linearize the system.

In order to analyze the functionality of the Extended Kalman filter, consider a video stream obtained by moving a camera sideways, from the center of the corridor to the left, and then back to the center of the corridor and then to the right. Figure 4.2 shows how the vanishing point of the corridor is being tracked by the image processing algorithm (frame by frame measurements) and by the Extended Kalman filter algorithm.

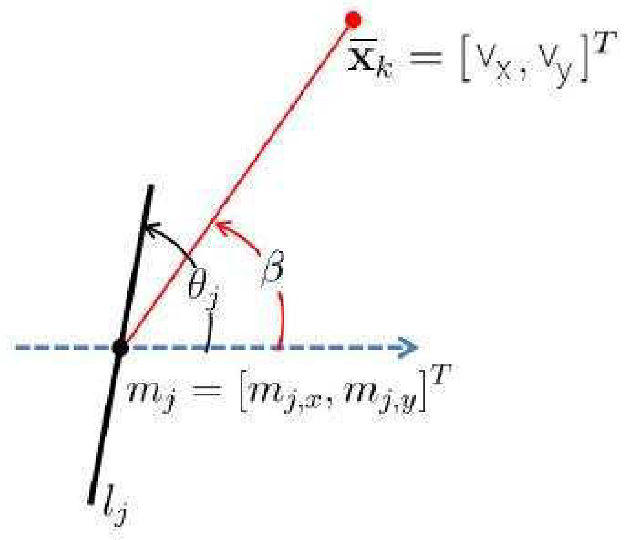

$\bar{\mathbf{x}}_k = [\mathrm{v_x}, \mathrm{v_y}]^T$
$\beta$
$\theta_j$
$m_j = [m_{j,x}, m_{j,y}]^T$
$l_j$

Figure 4.1: Angle Between Current State and Vanishing Point

The green line in the plot indicates the center of the corridor. Since the resolution of the image is high ( 1536 x 864 = 0.8 Mega pixels ), the performance of both the vision algorithm and the Extended Kalman filter algorithm are good. However, if the resolution is decreased, noise is expected to play a greater role in the frame by frame detection method. If the image resolution is decreased by a fourth, to 0.2 Mega pixels, then the behavior of the two algorithms in tracking the vanishing point is shown in Figure 4.3.

If the resolution is cut even further by a fourth, to 0.05 Mega pixels, the superiority of the Extended Kalman filter in tracking the vanishing point, as compared to the frame by frame line detection algorithm is clearly visible. Figure 4.4 clearly highlights this behavior. It is important to note that the tracking results are only as good as the long term quality of the image measurements [18]. At such low resolutions, the RMS measurement error observed in tracking the vanishing point is as high as 75 cm.

The benefit of using a state estimation filter on top of your vision algorithm

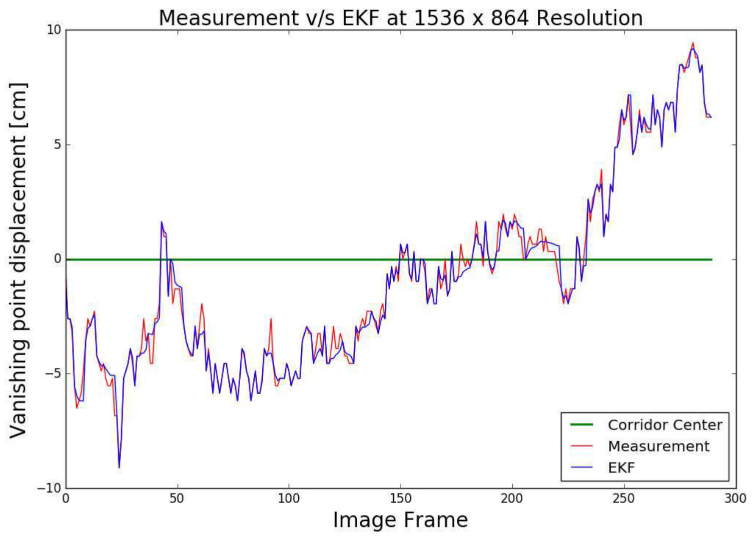


Figure 4.2: Measurement v/s EKF at 0.8 Mega Pixels

is depicted in the figures 4.6 and 4.7. The blue circle represents the vanishing point computed by the vision algorithm (frame by frame measurement) and the green circle represents the vanishing point being tracked by the state estimation filter. In situations were the number of dominant lines is below a certain threshold value, our frame by frame measurement is erratic. In these sitautions our confidence in the previous state of the vanishing point is high, hence the EKF algorithm correctly weighs the vanishing point estimate of the previous state more than the curren erratic measurement.

The variation in the measurement error and the algorithm update rate as a function of the image resolution is quantified in Table 4.1. Figure 4.8 highlights how the computation time of the algorithm varies as the image resolution is varied. The chosen image resolution depends on the operating requirements like, the speed with which the quadrotor has to travel, the width of the corridor (which determines the allowed margin for error) and constant/variable lighting conditions. In this thesis, the quadrotor is programmed to travel at a maximum speed of 70 cm/s, in a corridor that

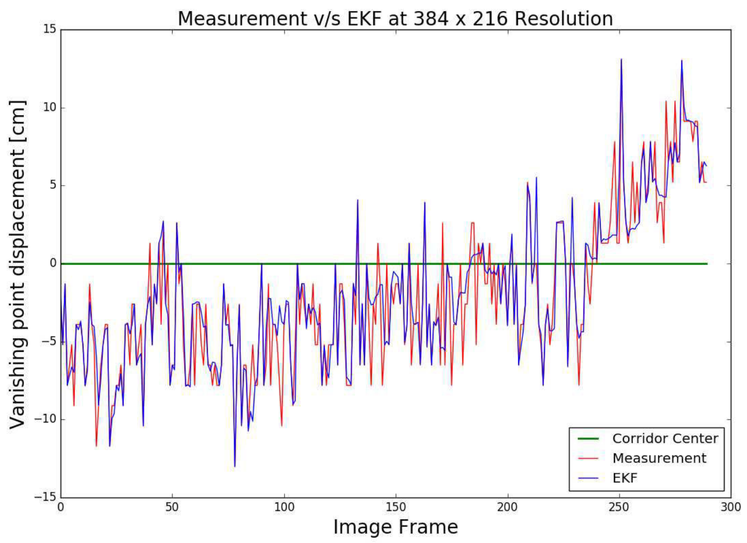


Figure 4.3: Measurement v/s EKF at 0.2 Mega Pixels

is 150 cm wide. Under such conditions, an image of 384 x 176 resolution resulted in a vanishing point measurement error of 5.1 cm. This proved to sufficient to navigate the quadrotor from one end of the corridor to the other.

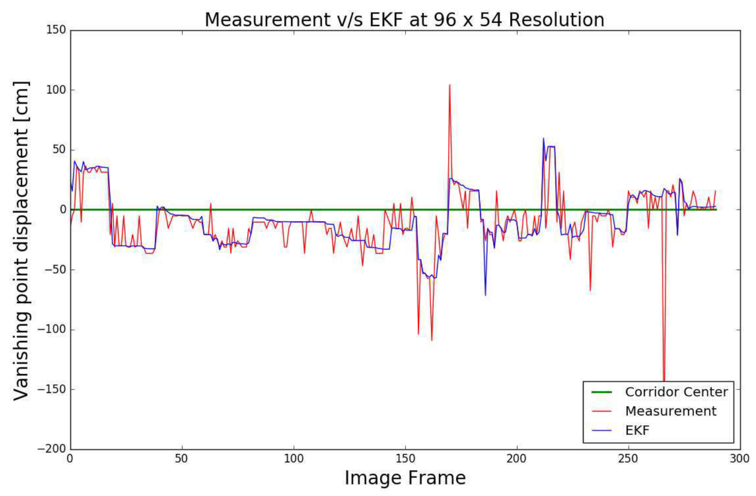


Figure 4.4: Measurement v/s EKF at 0.05 Mega Pixels

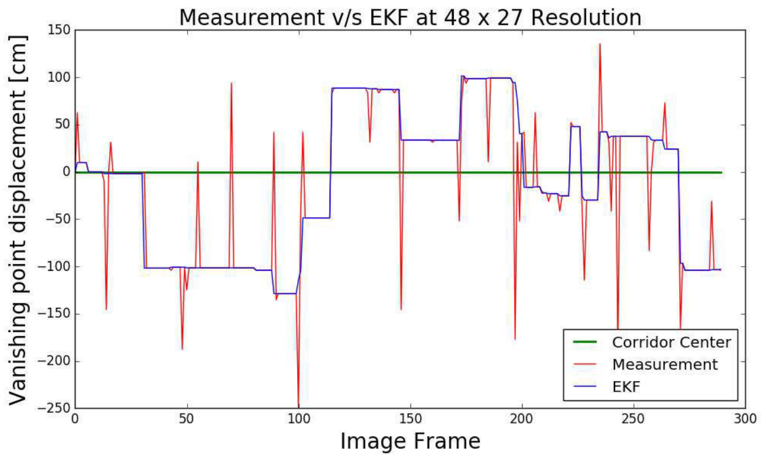


Figure 4.5: Measurement v/s EKF at 0.025 Mega Pixels

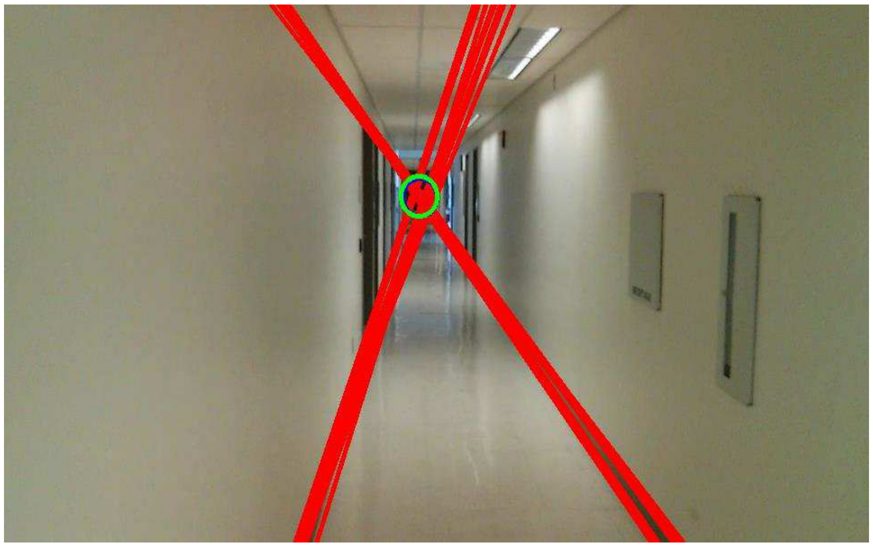


Figure 4.6: Measurement vs EKF Under Good Measurement Conditions (Blue and Green Circles Overlap)

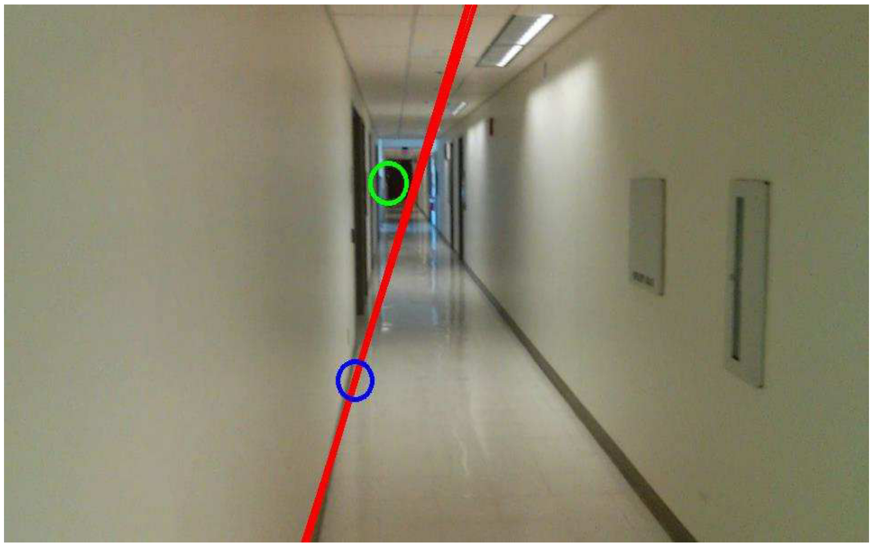


Figure 4.7: Measurement vs EKF Under Bad Measurement Conditions

| **Image Resolution (Mp)** | **Update Rate (Hz)** | **Error (cm) ($)** |
|---|---|---|
| 0.8 | 1.55 | 4.01 |
| 0.4 | 4.33 | 4.4 |
| 0.2 | 12.5 | 5.21 |
| 0.1 | 18.87 | 10.42 |
| 0.05 | 25.64 | 26.72 |
| 0.025 | 28.57 | 77.5 |

Table 4.1: Comparing Error and Update Rate at Different Resolutions

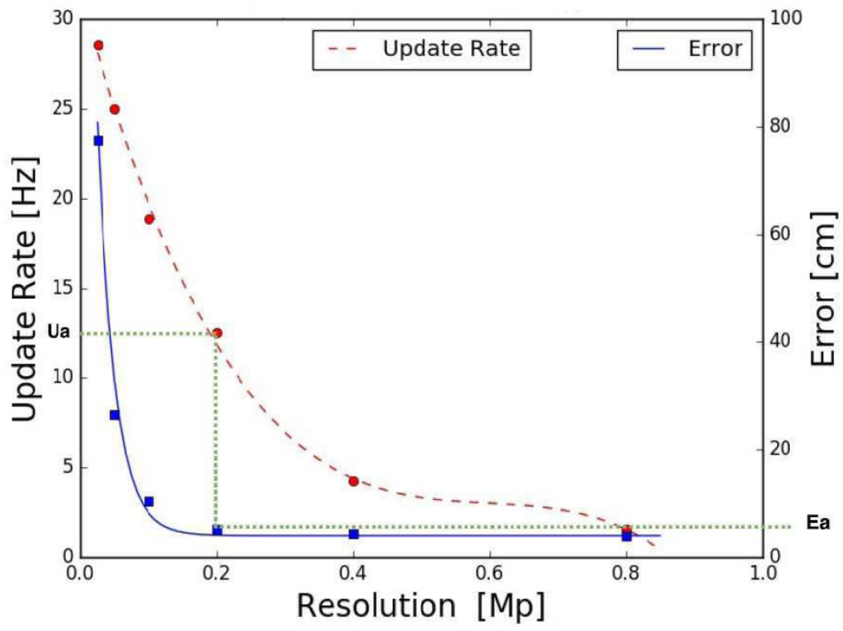


Figure 4.8: Image Resolution vs Update Rate

## 4.4 Summary and Conclusion

This chapter has explored the effectiveness of the Extended Kalman filter in tracking the vanishing point in an indoor corridor. The performamce of the filter and the vision algorithm under different image resolutions was analyzed. The improved tracking performace obtained under low image resolutions justify the inclusion of the Extended Kalman filter as part of the vanishing point tracking algorithm.

Chapter 5

# AR DRONE QUADROTOR MODEL AND CONTROL

## 5.1 Introduction and Overview

In this chapter, we will describe the AR Drone quadrotor platform, its dynamic model, hardware and software. We will describe how to incorporate the vanishing point coordinates along with the IMU measurements in order to get a more accurate estimate of the position of the AR Drone. Finally, we will summarize the literature describing the linear model of the AR Drone quadorotor and describe how to implement a closed loop position controller using this linear model.

## 5.2 AR Drone Platform

The quadrotor chosen for the experiments is the AR (Augmented Reality) Drone, from Parrot Inc., version 2.0. AR Drone is a commercialized, autonomous quadrotor that was primarily designed to be operated via smartphones or tablets, via Wi-Fi networks with specific communication protocols. Compared to other commercial quadrotors (AscTech Hummingbird III cost $4000) , the AR Drone is less expensive ($300). Its spare parts are easily available. In addition, Parrot provides a set of software tools to easily communicate with and control the drone platform [33]. These are some of the reasons why AR Drone 2.0 was chosen. Figure below shows the AR Drone with the adopted co-ordinate system. The drone is 55cm wide and 55cm long with the indoor hull. It weighs 450g with the indoor hull. The central cross of the drone is made of carbon tubes. A brushless motor is mounted at the end of each arm. The electronics are housed in a Poly Propylene basket at the center of the cross.

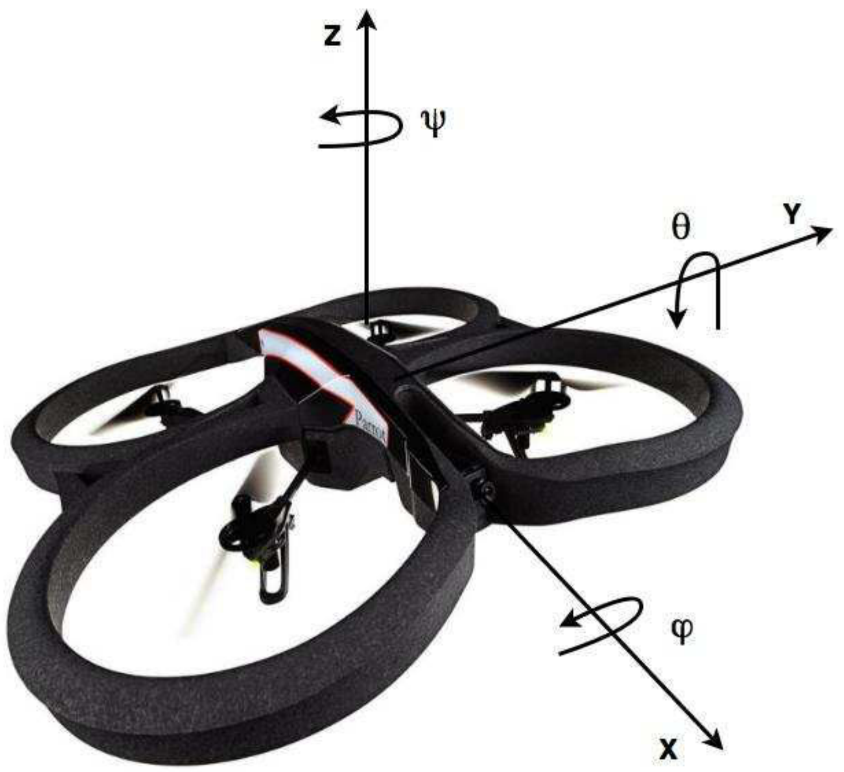


Figure 5.1: AR Drone with its Coordinate Axes in Body Fixed Frame

## 5.3 AR Drone Hardware, Software and Control Inputs

The AR Drone is equipped with two embedded boards. One of them is the sensor board and contains a 3 Axis Accelerometer (+/- 50mg precision, measures linear acceleration $a_x, a_y, a_x$), a 3 Axis Gyroscope(2000 degree/second precision, measures angular velocity $w_x, w_y, w_z$), an Ultrasonic Sensor( Range 0- 3m, for altitude measurement). The drone is also equipeed with a forward facing camera (120 x 720 resolution, 30 Hz frame rate), and a bottom facing camera (320 x 240 resolution, 60 Hz frame rate).

The second embedded board is called the principal board and contains the ARM Cortex A8 processor, with a 1GHz clock frequency, running an embedded Linux operating system. The principal board handles the data coming in from the sensor board, the video streams coming in from the front facing and the bottom facing cameras and the Wi-Fi network of the UAV system. The firmware installed on the quadrotor has the ability to perform tasks like take-off, landing, flight stabilization, besides responding to external commands fed by the user.

The drone, using its on-board sensors, delivers the following variables: $z, v_x, v_y, \phi, \theta, \psi$. The roll, pitch and yaw angles ($\phi$, $\theta$ and $\psi$ ) represent the orientation of the drone relative to the global coordinate system. z represents the height of the drone from the ground plane, measured by the downward facing ultrasonic sensor. $v_x$ and $v_y$ represent the linear velocities of the drone relative to the drones frame of reference (along $x_b$ and $y_b$)

Information about how these variables are generated can be found in [33]. Information about the algorithms used to generate these variables can be found in [24].

Despite these capabilities, the AR Drone is incapable of hovering over a point autonomously for a significant time period. It starts to drift (slide sideways). This is due to the accumulation of measurement errors that are integrated over time. Thus, the AR Drone must be continuously fed position information from an external reference source in order to negate the drift.

The command signals to control the motion of the drone are normalized to have values between -1 and 1. The commands that can be issued to the drone are:

1. $u_x$ - Inclination in the $x_w$(roll) axis which translates to velocity in the $y_b$ direction
2. $u_y$: Inclination in the $y_w$ (pitch) axis which translates to velocity in the $x_b$ direction
3. $\dot{z}$: Linear velocity which causes displacement along the $z_w$ axis
4. $\dot{\psi}$: Angular velocity which causes rotation (yaw) about the $z_w$ axis

In the following two sections, we will describe the approach suggested by [19], to develop the controller for the drone. Before that, we would like to make a quick note on the maximum velocity with which the drone can be controlled to fly in indoor corridors (without colliding into the walls). The maximum velocity is a function of the corridor width ($W_C = 150cm$), drone width ($W_D = 55cm$), measurement error ($\Delta Y_M = 5cm$), maximum drift rate ($\omega = 20$ degrees/second) and the time interval between command rate transmissions ($\Delta T = 10 * (\Delta T_{IMG} + \Delta T_{WiFi}) = 10 * (80ms + 50ms) = 1300ms$). Their realtionship can be described by the following equation:

$$v_{max} = \frac{W_C/2 - W_D/2 - Y_M}{\sin(\omega \Delta T)} \tag{5.1}$$

The computations were done off-board, using a 2.5 GHz Intel Ivy Bridge CPU ($\Delta T_{IMG} = 80ms$), the maximum velocity was computed to be equal to 1 m/s. When the image processing was performed off-board, on NVIDIA'S GeForce 970M GPU (1280 Cores, 1024 threads/core, 960 MHz), the time interval between control commands was $\Delta T = 10 * (\Delta T_{GPU1} + \Delta T_{WiFi}) = 10 * (11.43ms + 50ms) = 614.3ms$. The maximum velocity was computed to be approximately 2 m/s. When the image processing was performed on-board, using NVIDIA'S Jetson TX2 Embedded board (256 cores, 1024 threads/core, 1300MHz), the time interval between control commands was $\Delta T = 10 * (\Delta T_{GPU2}) = 10 * (30.21ms) = 300.21ms$. The maximum velocity was computed to be approximately 4 m/s.

## 5.4 AR Drone Dynamic Equations of Motion

The dynamic equations of motion of a quadrotor are given below [32].

$$m\ddot{x} = (\cos\psi \sin\phi + \cos\psi \cos\phi \sin\theta)u_1 \tag{5.2}$$

$$m\ddot{y} = (-\cos\psi \sin\phi + \sin\psi \cos\phi \sin\theta)u_1 \tag{5.3}$$

$$m\ddot{z} = (\cos\phi \cos\theta)u_1 - mg \tag{5.4}$$

$$I_{xx}\ddot{\phi} = u_2 - (I_{zz} - I_{yy})\dot{\theta}\dot{\psi} \tag{5.5}$$

$$I_{yy}\ddot{\theta} = u_3 - (I_{xx} - I_{zz})\dot{\theta}\dot{\psi} \tag{5.6}$$

$$I_{zz}\ddot{\psi} = u_4 \tag{5.7}$$

$\ddot{x}, \ddot{y}, \ddot{z}$ represent the linear accelerations along the x,y and z axis respectively. $\phi$,$\theta$ and $\psi$ represnt the orientation (roll, pitch and yaw) of the quadrotor with respect to the global x, y and z coordinate axis frames. $\dot{phi}$, $\dot{theta}$, $\dot{psi}$ and $\ddot{phi}$, $\ddot{theta}$, $\ddot{psi}$ represent the angular velocities and anlular accelerations with respect to the x, y and z axis respectively. $m$ represnts the mass of the quadrotor. $g$ is the acceleration due to gravity. $I_{xx}$ ,$I_{yy}$ and $I_{zz}$ represnt the Moment of Inertia (resistance to rotation) around the x, y and z axis respectively.

$u_1$ represents the vertical thrust input signal to the drone, $u_2$, $u_3$ and $u_4$ represent the input torques with respect to the three body fixed coordinate axis of the drone respectively.

Note that the model above does not take drag forces (resistance offered by the wind) into consideration. According to [2], a model similar to the one described above is used in the AR drone firmware, with the addition of other aerodynamic effects for flight stabilization. However, the firmware algorithm and the model parameters are restricted to developers. We will soon show how to take advantage of the drones internal processing results by modeling its dynamics in terms of the control action $u$.

To achieve this we rely on the observations of [28],[26],[20],[22],[23] and assume the behavior of the drone to be a linear function of the input command. Doing so, we can avoid the complex dynamics of the quadrotor. This linear model is given below.

$$\ddot{v}_x = K_1 u_x - K_2 \dot{v}_x \tag{5.8}$$

$$\ddot{v}_y = K_3 u_y - K_4 \dot{v}_y \tag{5.9}$$

$$\ddot{v}_z = K_5 u_z - K_6 \dot{v}_z \tag{5.10}$$

$$\ddot{\psi} = K_7 u_\psi - K_8 \dot{\psi} \tag{5.11}$$

Here, $v_x$, $v_y$ and $v_z$ represents the linear velocities along the drone coordinate axes $x_b$, $y_b$ and $z_b$ respectively. The same holds true for the accelerations $\ddot{v}_x$, $\ddot{v}_y$ and $\ddot{v}_z$. $\dot{\psi}$ and $\ddot{\psi}$ represents the angular velocity and angular acceleration with the respect to the $z_b$ axis. The constants K1 to K8 are experimentally determined by minimining the sum of squared differences between the actual values and the simulated values. Let's call this sum of squared difference function as the Energy function. We need to find constant K1 and K2 such that the energy $E(K_1, K_2)$ is minimum. Mathematically, the energy function for horizontal velocity is described below.

$$E(K_1, K_2) = argmin_{K_1,K_2}(\ddot{x}_{actual} - \ddot{x}_{simulated}) \tag{5.12}$$

$$\ddot{x}_{simulated} = K_1 u_X - K_2 \dot{x}_{actual} \tag{5.13}$$

Although, this model is not physically precise, it is sufficient for our application. The actual horizontal and vertical velocities of the drone are plotted against the ones generated using the linear model in the figures 5.2 and 5.3.

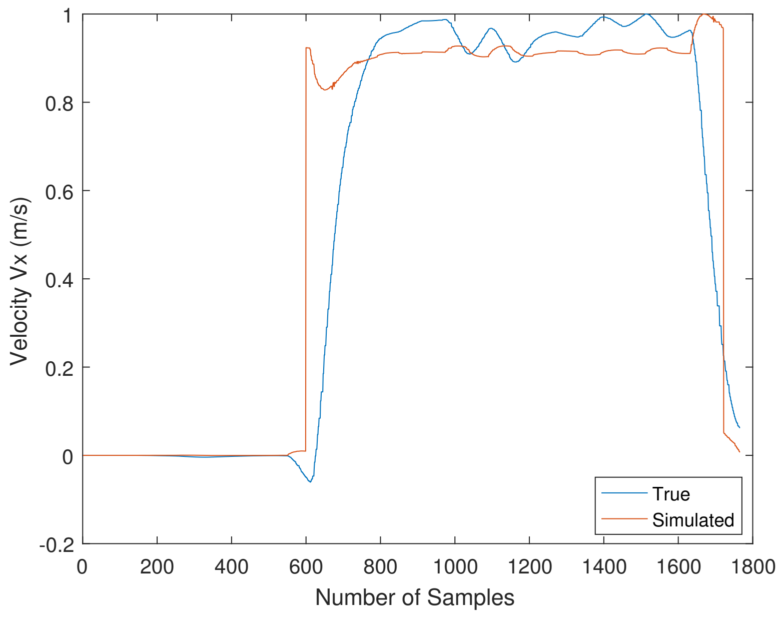


Figure 5.2: Plot of the True and Simulated Horizontal Velocities

## 5.5 AR Drone Position Controller

In order to design an outer loop position controller for the AR Drone, we need to know its state at every time interval. Here, the state of the drone refers to the 3 dimensional postion and velocity estimates along the x,y and z axes, along with the yaw angle and the yaw rate. The state vector is described below.

$$\mathbf{x} = [x, y, z, \psi, \dot{x}, \dot{y}, \dot{z}, \dot{\psi}] \tag{5.14}$$

We can integrate the velocity measurements along each coordinate axis of the drone to obtain the x, y and z position information of the drone. The yaw and

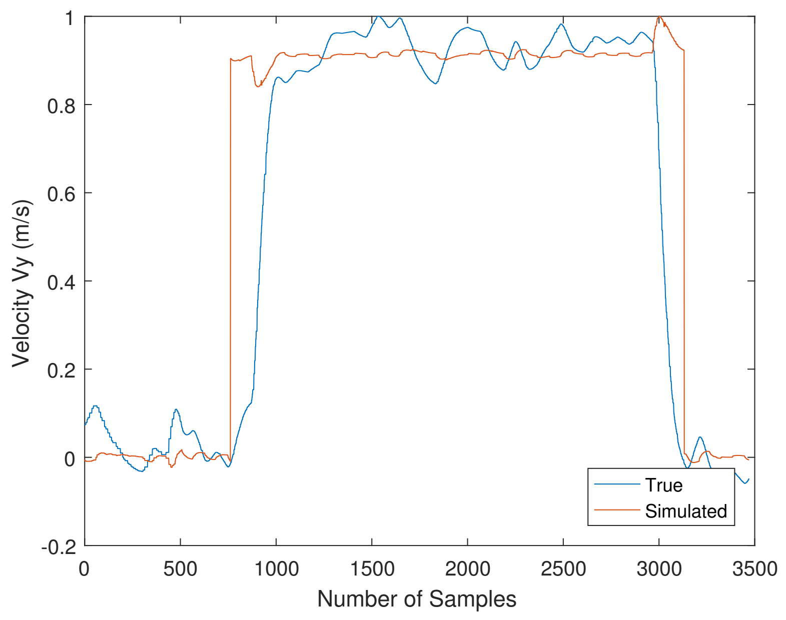


Figure 5.3: Plot of the True and Simulated Vertical Velocities

the yaw rate are directly provided by the AR Drone IMU measurements (For more information on how to subscribe to the necessary topics of the AR Drone, please refer to the ROS code at the Appendix and also to [2]). As soon as the vanishing point estimates are available, we update our state vector with a more accurate horizontal position ( represented by $y$) estimate. We do this by simply taking the weighted average of the two measurements (a weight of 0.1 was chosen by trial and error).

$$y_{accurate} = wy + (1 - w)y_{vp} \tag{5.15}$$

Here, $y_{vp}$ represents the horizontal position of the drone estimated by tracking the vanishing point in the indoor corridor. The linear model of the drone described in the previous section can be represented as:

$$\dot{X} = f_1 U - f_2 \dot{X} \tag{5.16}$$

Here, $X = [\dot{x}, \dot{y}, \dot{z}, \dot{\psi}]$ and $U = [u_{v_x}, u_{v_y, u_{\dot{z}}, u_{\dot{\psi}}}]$.

$$f_1 = \begin{bmatrix} K_1 \cos\phi & -K_3 \sin\phi & 0 & 0 \\ K_1 \sin\phi & K_2 \cos\phi & 0 & 0 \\ 0 & 0 & K_5 & 0 \\ 0 & 0 & 0 & K_7 \end{bmatrix}$$

$$f_2 = \begin{bmatrix} K_2 \cos\phi & -K_4 \sin\phi & 0 & 0 \\ K_2 \sin\phi & K_4 \cos\phi & 0 & 0 \\ 0 & 0 & K_6 & 0 \\ 0 & 0 & 0 & K_8 \end{bmatrix}$$

Given a desired trajectory $[x_{des}, y_{des}, z_{des}, \psi_{des}]$, our goal is to minimize the error $e = [x_{des} - x, y_{des} - y, z_{des} - z, \psi_{des} - \psi]$

From control theory, we know that:

$$\ddot{e} + K_p e + K_d \dot{e} = 0 \tag{5.19}$$

if $K_p > 0$ and $K_d >= 0$.

This means that if the two gains, $K_p$ and $K_d$( proportional and derivative gains) are positive, the error is gauranteed to asymptotically converge to zero. We can rewrite the above equation :

$$\ddot{x} = \ddot{x}_{des} + K_p e + K_d \dot{e} = 0 \tag{5.20}$$

Here $\ddot{x}$ (for the sake of notational convenience, let's call this $v$) is the commanded acceleration and $\ddot{x}_{des}$ is referred to as the feedforward term. We can rewrite equation 5.15 as:

$$U = f_1^{-1}(\ddot{X} - f_2 \dot{X}) \tag{5.21}$$

This now gives us:

$$U = f_1^{-1}(v - f_2\dot{X}) \tag{5.22}$$

Thus, we now know how to generate the control signals necessary to fly the AR Drone so that it follows a given trajectory.

## 5.6 Experimental Results

Here, we will present the position error resulting from flying the AR Drone (maximum speed is limited to 1.2 m/s) in a straight line along the center of the corridor. The desired trajecotry was set to [0.5t 0 1 0 ]. The figure 5.4 depicts the XY position of the drone (in red) relative to the center of the corridor. The maximum lateral position error (error along the Y axis) from the center of the corridor is 34 cm.

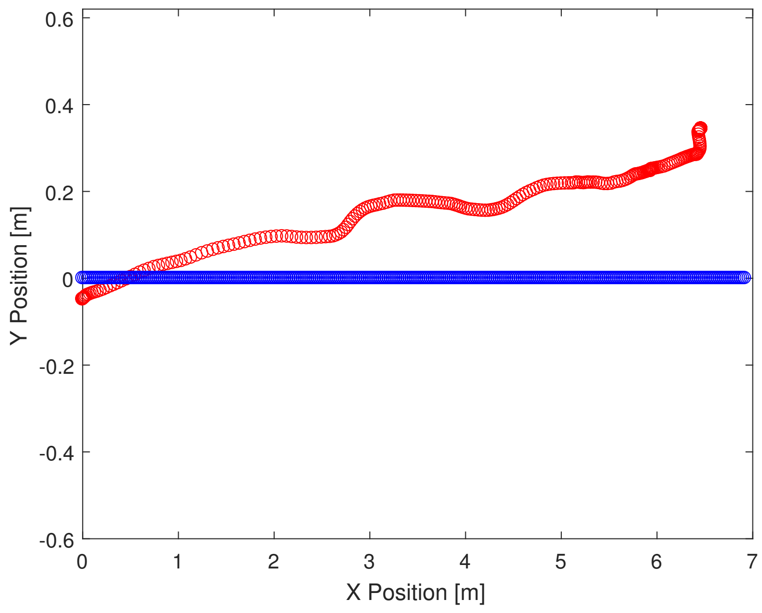


Figure 5.4: Plot of the Drone Trajectory Relative to the Center of the Corridor

## 5.7 Summary and Conclusion

In this chapter, we presented a brief description of the hardware and software features of the AR Drone quadrotor. We described the simplified linear model of the dynamics of the AR Drone and showed how to design a position controller using this model.

Chapter 6

# QUADROTOR SIMULATION ENVIRONMENT

## 6.1 Introduction and Overview

In this chapter, we will describe how the vision and the control algorithms described in the previous chapters can be implemented and tuned on a simulated AR Drone quadrotor using the Gazebo simulation software. The AR Drone simulator is a popular open source quadrotor simulator developed by the Technical University of Munich (TUM) [26]. The simulator model incoporates all the sensors on the Parrot AR Drone 2.0. The simulation model also follows the same communication protocols as the Parrot AR drone. The simulation world of an indoor corridor, is designed using a varielty of pre-existing templates available in Gazebo. The vision, control and simulation programs are implemented in the Robot Operating System (ROS) environment. The implementation pipeline, from intializing the AR drone simulator to running the vison and control algorithms on the drone, will be described in detail.

## 6.2 Robot Operating System

Robot operating system (ROS) is a useful software environment to program robots. Surgical robotic arms like *daVinc*i from Intuitive Surgical, and autonomous ground robots like the *Husky Unmanned Ground Vehicle* (UGV), use ROS beacuse of its simplicity and modular framework. ROS is different from a traditional operating system with respect to process management and scheduling. ROS is more a structured communication layer that sits on top of the host operating system. The primary motivation for developing ROS was to simplify software design for robots. Often

Figure 6.1: Da Vinci Surgical Robot

Figure 6.2: Husky UGV

hardware compatibility issues necessitates developing custom software for different hardware. As a result, a lot of time is spent reinventing the wheel and this slows down research in the field of robotics. ROS was designed to make sure large scale integrative robotics research will be possible [38]. .

The design goals for ROS can be summarized below:

1. **Peer to Peer connectivity**

   A number of different executable programs (called nodes in ROS), running on a number of different computers, can all be individually designed and loosely coupled during runtime.

2. **Language Agnostic**

   Framework is easy to implement in any programming language. ROS currently supports four languages: C++, Python, Octave and LISP.

3. **Modular Framework**

A lot of small modules are used to build and run various components in ROS. Such a modular approach tends to decrease efficiency but often reduces complexity and increases stability.

4. **Thin**

   By making sure all the complexity resides in standalone libraries, ROS is able to create small executables that allows for easier code extraction and reuse.

5. **Open-Source and Free**

   ROS is distributed under the terms of the BSD license and its source code is publicly available. The ROS community is active and regularly updates existing versions with new functionalities.

For an excellent introduction to ROS, please refer [39]

## 6.3 Gazebo Simulation Software

Gazebo is an open source, multi-robot, simulation environment that integrates easily with the Robot Operating System (ROS) framework. It has become the de facto standard in robotics research and allows contribution by other researchers to be easily integrated. Gazebo allows sensors like stereo cameras, LIDARs, sonar sensors, RGB-D cameras to be easily added to the robot. The simulation environment includes dynamics simulation, provided by the ODE (ordinary differential equations) or bullet physics engine. Although the simulator provides options for gravity and friction, it does not cover aerodynamics and propulsion systems which are integral to research in aerial robotics. For a comprehensive overview of a quadrotor simulation desgin using Gazebo and ROS please refer [40]. In this research, we have used the quadrotor simulator developed by the Technical University of Munich, called the *TUM AR*

*DRONE SIMULATOR* . Just like the actual AR Drone, the simulated drone accepts roll angle, pich angle, yaw rate and vertical velocity as input. It has a forward and a downward facing camera along with an ultrasonic sensor to measure the altitude. The simulated corridor environment along with the drone is shown in the figure below.

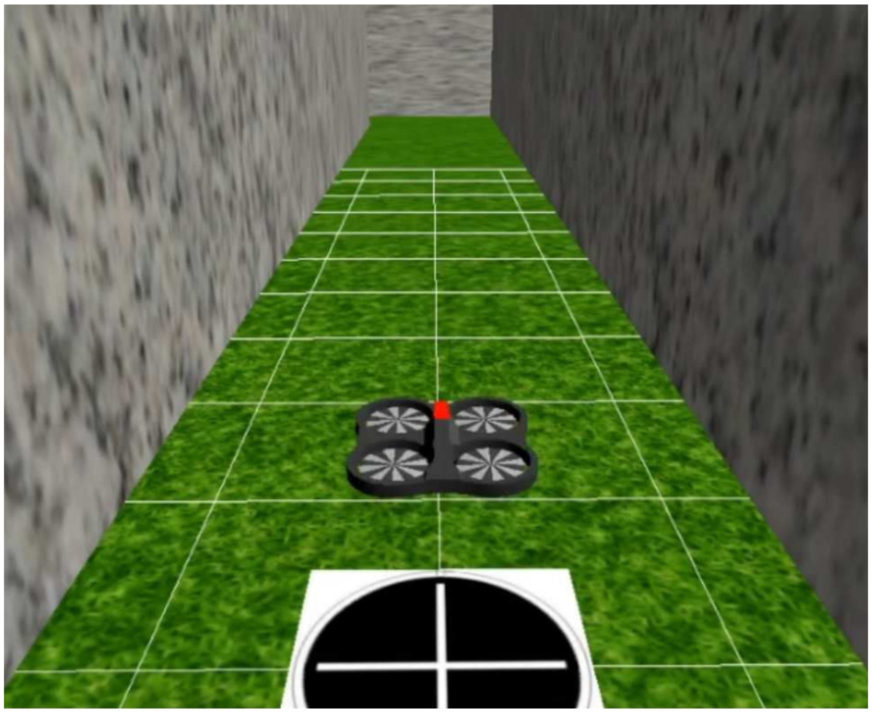


Figure 6.3: AR Drone Simulation in Gazebo

The view from the front camera as well as the vanishing point, can be observed in the figures below

Along with a simulation of the AR drone, the TUM AR Drone software package has a number of built in simulation worlds. With this basic framework in hand, creating a simulation environment with a ground plane, a roof and two enclosing walls is trivial. Gazebo also offers extensive tutorials on how to build simulation worlds from scratch.

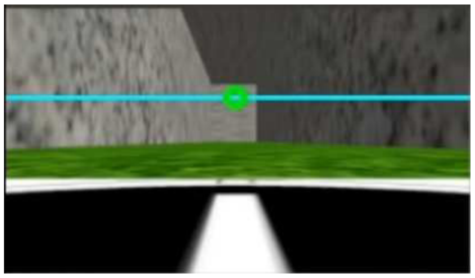


Figure 6.4: Image from the Drone Front Camera

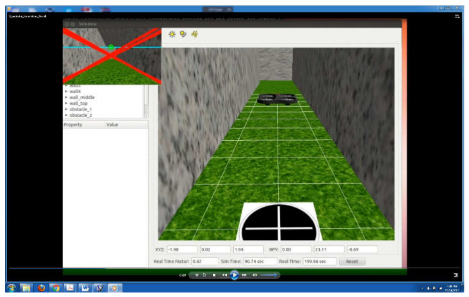


Figure 6.5: Vanishing Point Detected on the Front Camera Image

## 6.4 Implementation Pipeline

The setup guide in the appendix gives a detailed account of all the dependencies required for installing and running the AR Drone simulator. At the time of writing, the TUM AR Drone simulator was compatible only with *ROS Fuerte* and not with any of the more modern versions of ROS. So in order to use the simualtor without any glitches, it is necessary to make sure you are working with the *Fuerte* version of ROS. Along with the AR Drone simulator package, it is necessary to install the *Ardrone Autonomy* package which handles all the communication to and from the drone. Once all the necessary packages are installed, the information flow among the primary nodes (executable programs) is as shown in the figure below.

ROS allows users to tap into any sensor measurement via *ROS Topics*. A given topic can be queried to obtain the sensor mesasurement as well as the update frequency. An important topic published by the Ardrone Autonomy package is the *navdata* topic. This topic contains the measurements pertaining to the IMU, cameras and the ultrasonic sensor. The navdata topic contains the following information.

1. navdata.vx : Drone's velocity along the $x_b$ axis

2. navdata.vy : Drone's velocity along the $y_b$ axis

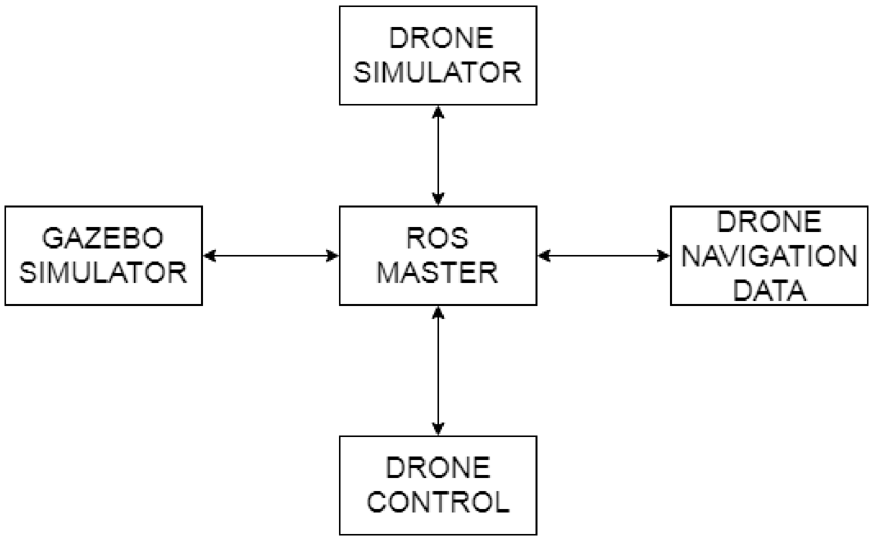


Figure 6.6: ROS Control Flow

3. navdata.vz: Drone's velocity along the $z_b$ axis
4. navdata.altd: Height of the drone above the ground plane
5. navdata.rotX: Rotation about the $x_b$ axis
6. navdata.rotY: Rotation about the $y_b$ axis
7. navdata.rotZ: Rotation about the $z_b$ axis
8. navdata.batteryPercent: Indicates the remaining charge in battery.
9. navdata.tm: Timestamp of the data returned by the drone.
10. navdata.state: State of the drone. It can be Hovering, Initiated, Flying, Landed, Landing or Take-off

We subscribe to this topic to get access to sensor measurements. Once this is done, the controller is designed using the procedure described in the previous chapter. The simulator can help provide approximate estimates for the proportional and derivative gains before fine tuning it during real world experiments. The TUM Simulator is light weight, intuitive and well documented. The procedure for creating your own simulation environment is explained in detail in the *Gazebo Tutorials Forum*.

## 6.5 Simulation Results

In this section, we present the results of programming the AR drone to navigate along the center of a simulated corridor. The trajectory of the drone, for a specified $[x_{des}, y_{des}, z_{des}, \psi_{des}]$ is shown in the figures below. The position error in each trial is tabulated below.

| **X desired (m)** | **Speed (m/s)** | **Max X Error (m) ($)** | **Max Y Error (m) ($)** |
|---|---|---|---|
| 15 | 0.3 | 0.84 | 0.42 |
| 25 | 0.3 | 0.94 | 0.51 |

Table 6.1: Position Error of the AR Drone for Different Distances

## 6.6 Summary and Conclusion

This chapter has presented the ROS framework for robotics research and briefly described the Gazebo simulation environment. The procedure for using open source drone simulators for the purpose of indoor navigation and vanishing point detection has been explored. Simulation results of the AR Drone navigating in a simulated world (with a structure similar to that of an indoor corridor) were presented.

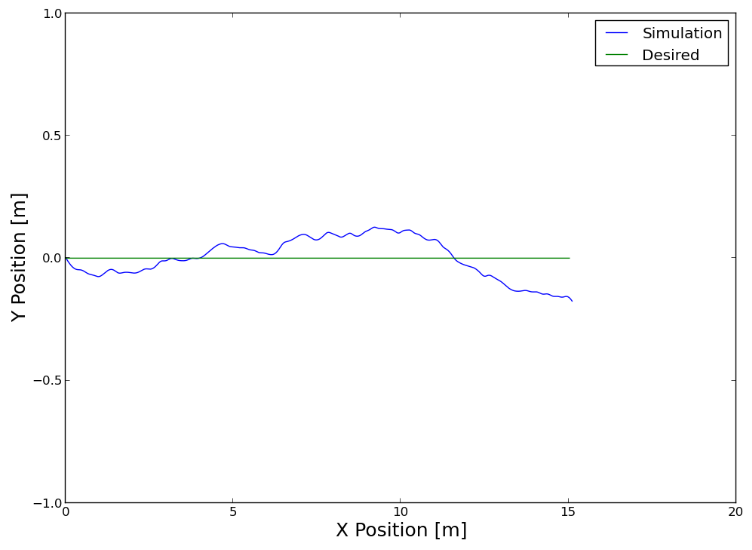


Figure 6.7: $[X_{des}, Y_{des}, Z_{des}, \psi_{des}] = [15, 0, 1, 0]$

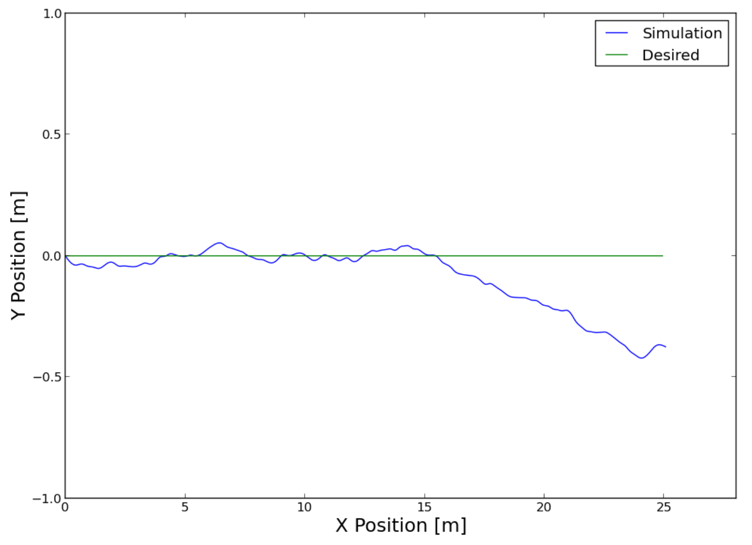


Figure 6.8: $[X_{des}, Y_{des}, Z_{des}, \psi_{des}] = [25, 0, 1, 0]$

# Chapter 7

# PARALLEL IMPLEMENTATION OF VISION ALGORITHMS USING GPU

## 7.1 Introduction and Overview

In Chapter 4, we described a single threaded implementation (single CPU core) of the line detection algorithm in the outer loop (A thread is a separate path of execution that may converge or diverge from the main program flow as and when needed). Current day GPU make use of millions of threads. The potential processing speed-up that a GPU offers for parallel implementation of computer vision algorithms is well documented [41].

In this chapter, we will provide an overview of NVIDIA GPU (Graphics Processing Unit) hardware and software (CUDA: Compute Unified Device Architecture) architecture. We will explain how the CUDA framework can be used to design a multi-threaded implementation of a common operation in image processing, convolution.We will summarize the approach taken towards designing a parallel implementation of the Hough transform [35]. We will compare and analyze the runtimes of the single threaded CPU and the multi-threaded GPU implementations. Lastly we will describe the features of a modern embedded GPU platform, NVIDIAs Jetson TK1 board and analyze its potential as a powerful outer loop processor on drones, taking into account the size, weight, and power constraints.

## 7.2 GPU Hardware

Modern CPUs are at a limit for how much instruction level parallelism (ILP) can be extracted per cycle. Even as transistors have become smaller and more numerous in number, the clock frequency has plateaued over the last few years because of the inability to manage the large amount of heat generated by billions of transistors. Power management has since become a crucial aspect of chip design. GPUs are designed to achieve the most bang for the buck in terms of computations for a fixed amount of power. They achieve this by having a much simpler control circuitry, compared to the CPU. This leaves more transistors in the data path for computation. But this causes GPUs to have a more restrictive programming model. Optimizing GPU code for speed comes at the cost of more complex code. Complexity not only in the number of source code lines but also in the degree with which parameters in the code (like image size) can be adjusted. GPU designers have traded fewer complex cores for many simpler cores. This means that unlike the CPU, the GPU is optimized for throughput, not latency.

Each GPU (depending on the generation) has a:

- Global Memory
- 8 to 32 Streaming Multiprocessors (SM)
- 1 to 16 Thread Blocks per SM
- 256 to 2048 threads per block.

A thread based view of the GPU is shown in the figure below.

The code executed on the GPU is called the kernel. Each thread in the GPU executes the same program not necessarily the same instructions. Each GPU can run thousands to millions of threads. Threads are arranged into thread blocks. Each thread block is part of a larger grid. All threads in a thread block are executed on the same processor (SM) and communicate via shared memory (on-chip memory). Threads are organized into a cluster of (at most) 32 warps. A more complete description of the GPU hardware is provided in [25].

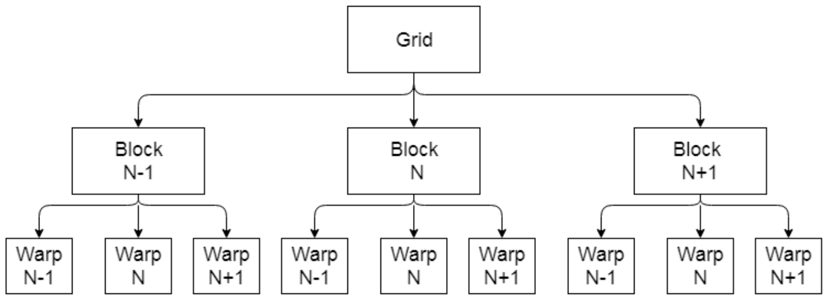


Figure 7.1: Organization of Threads in a GPU

## 7.3 GPU Software

In 2006, NVIDIA unveiled the GeForce 8800 GTX GPU. This was the first GPU to be built using NVIDIAs CUDA architecture. This was the first push towards making GPUs adept at general purpose programming. CUDA stands for Computer Unified Device Architecture. CUDA allows each and every arithmetic and logic unit (ALU) on the chip to be orchestrated by a program intended to perform general purpose (single precision floating point) arithmetic. Furthermore, the execution units are allowed arbitrary read and write access to the memory along with access to a software-managed cache known as shared memory [4]. These features allowed the GPU to excel at general purpose computation, in addition to the graphics related tasks. CUDA programming environment is very similar to C programming language. It conveniently allows us to program both the CPU (referred to as the host) and the GPU (referred to as the device) using a single program.

The CPU is typically responsible for the following tasks:

- Allocating memory on the GPU
- Copy data from CPU TO GPU
- Launch Kernels on the GPU

- Copy data from GPU to CPU

The GPU is responsible for expressing the computation on the kernel and executing it on a large number of threads. This will soon be made clear in the next section where we describe the kernel for convolving the image with a filter. But what is evident is that GPU follows the SPMD (Single Program Multiple data) model. Each thread performs the same computation but on different input data. To utilize the GPU effectively, you need to have a high ratio of computation (in GPU) to communication (between GPU and CPU).

## 7.4 Kernel for Performing Convolution

The goal of this section is to explain how the kernel function should be designed for implementing the convolution operation. We will also talk about the dynamics of launching multiple threads on the GPU. The two primary tasks to be executed prior to launching the kernel are:

- Allocating memory on the GPU for the input image, filter and the output.

- Copying the input image and filter from CPU to GPU.

We are going to convolve each channel of the RGB image separately with the given filter and then recombine them. Convolution entails sliding the filter across each pixel of the input image and taking a weighted sum. The operation is pictorially depicted for a 5 x 5 image and a 3 x3 filter below.

In image processing, we want the output of the convolution operation to be of the same size as the input. This is done by padding the input array. The two popular ways of padding are same padding and zero padding [42]. A good starting place to design the kernel is to map each thread to a pixel. Once we have done this, we can

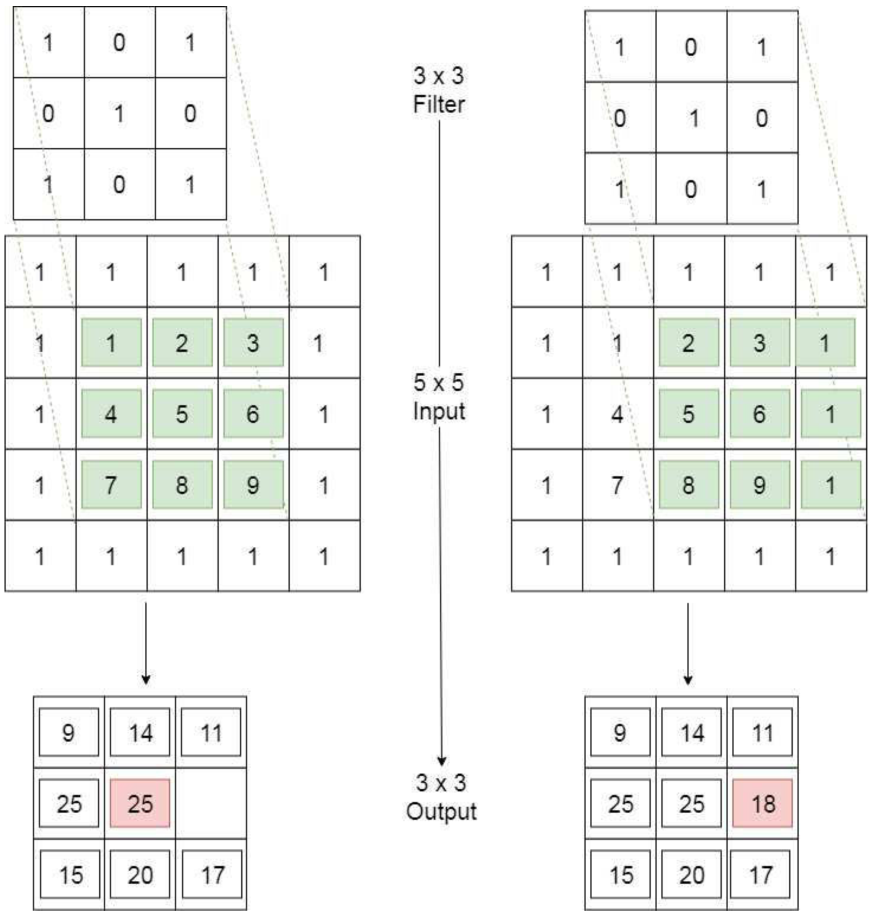


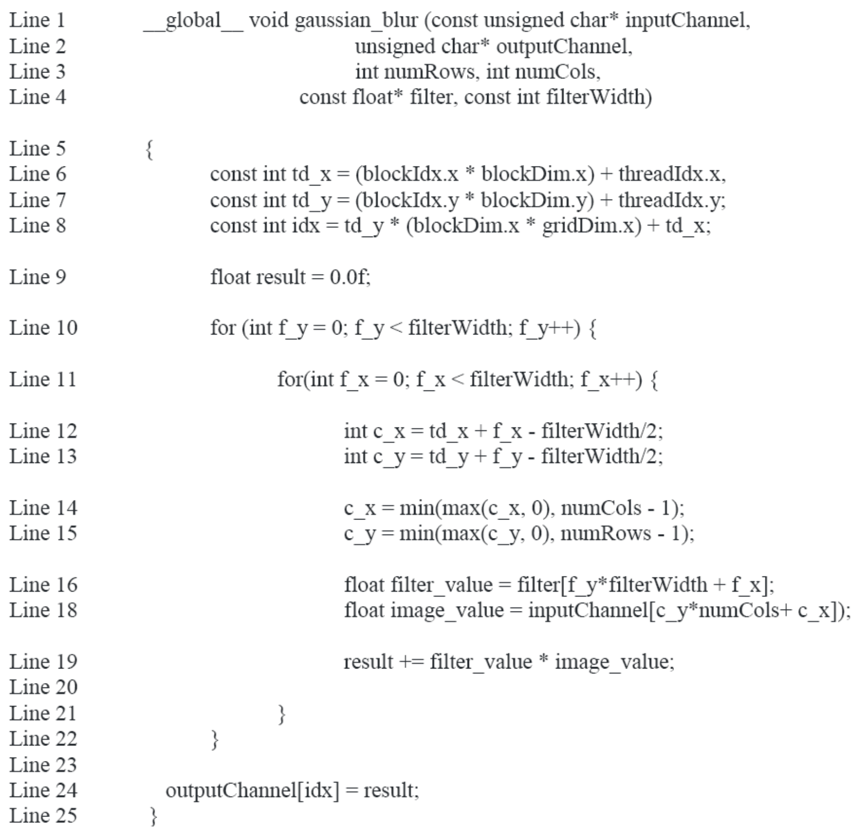
Figure 7.2: Convolving a 5 x 5 Input with a 3 x 3 Filter

use the filter to take a weighted average of the pixel and its neighbors. The CUDA code for the GPU kernel is shown in the figure below.

Each channel of the image is flattened from a 2-D to 1-D array. The filter is also flattened to a 1-D array. Let us walk through the important steps.

- In Line 1, the *global* prefix tells the compiler to generate GPU code and not CPU code. It also makes the GPU code globally visible from within the GPU code. We pass the necessary input and output pointers to the GPU memory space as arguments to the kernel.

- The *numRows* and *numCols* represent the height and width of the image re-



```
__global__ void gaussian_blur (const unsigned char* inputChannel,
                               unsigned char* outputChannel,
                               int numRows, int numCols,
                          const float* filter, const int filterWidth)

{
        const int td_x = (blockIdx.x * blockDim.x) + threadIdx.x,
        const int td_y = (blockIdx.y * blockDim.y) + threadIdx.y;
        const int idx = td_y * (blockDim.x * gridDim.x) + td_x;

        float result = 0.0f;

        for (int f_y = 0; f_y < filterWidth; f_y++) {

                for(int f_x = 0; f_x < filterWidth; f_x++) {

                        int c_x = td_x + f_x - filterWidth/2;
                        int c_y = td_y + f_y - filterWidth/2;

                        c_x = min(max(c_x, 0), numCols - 1);
                        c_y = min(max(c_y, 0), numRows - 1);

                        float filter_value = filter[f_y*filterWidth + f_x];
                        float image_value = inputChannel[c_y*numCols+ c_x]);

                        result += filter_value * image_value;

                }
        }

   outputChannel[idx] = result;
 }
```

Figure 7.3: CUDA Kernel Code for Convolution Operation

spectively.

- Thread Index and Block Index are provided as 3-D structures in CUDA: threadIdx.x, threadIdx.y and threadIdx.z ; blockIdx.x, blockIdx.y and blockIdx.z
- *threadIdx.x* represents the offset with a block of the X thread index
- *threadIdx.y* represents the offset with a block of the Y thread index.
- *blockDimx.x and blockDimx.y* represent the number of threads in the X and Y dimension of a thread block respectively.
- *gridDimx.x and gridDimx.y* represents the number of blocks along the X dimension and Y dimension of the grid respectively.

- For illustrative purposes, let the (number of columns) width of the image be 480 and the (number of rows) height be 360.
- Each processor, called Symmetric Multiprocessor (SM), in the GPU has a number of blocks and each block has a number of threads. To recap, a group of 32 threads is called a warp. Currently, the GPU hardware processes threads only in multiples of warps. All threads in a warp execute the same instruction in parallel (at the same clock cycle). Thus, it is good design practice to launch a multiple of 32 threads per block.
- We can launch the maximum allowed threads per block, 1024 (the maximum depends on the GPU generation). This is done by setting *blockDim.x* to 32 and *blockDimx.y* to 32. This gives us 1024 threads per block.
- In order to calculate the number of blocks required, we divide the number of pixels in the image by the number of threads per block and rounded it off to the highest integer. This gives 169 blocks. Now we shall set the *gridIdx.x* to 1 and *gridIdx.y* to 169. This gives us a total of 169 blocks in the grid.
- The entire thread layout for this example is shown in the figure below.
- Now its clear that *idx* in Line 8 runs from 0 to number of pixels (slightly larger than the number of pixels as the number of blocks is rounded off to the highest integer) in the image. Thus we have mapped each thread to a pixel.
- Lines 10 -22 now perform the trivial convolution operation.
- Now, since the kernel is launched by the CPU. The number of threads per block and the number of blocks per grid is also set by the CPU.

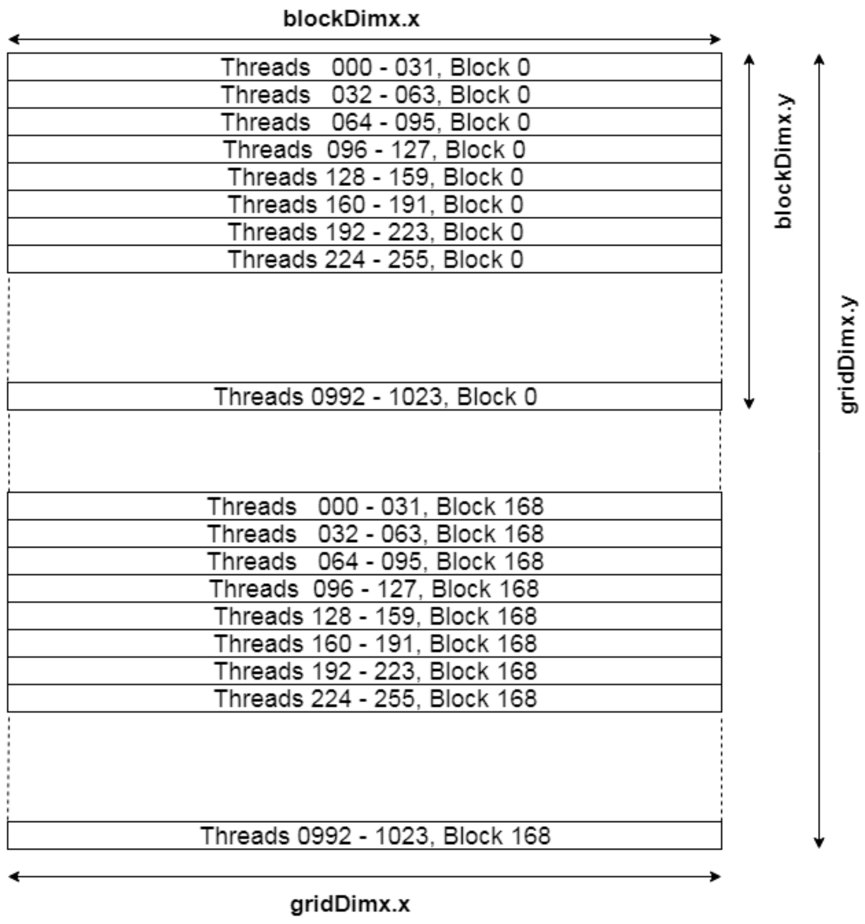

blockDimx.x
Threads 000 - 031, Block 0
Threads 032 - 063, Block 0
Threads 064 - 095, Block 0
Threads 096 - 127, Block 0
Threads 128 - 159, Block 0
Threads 160 - 191, Block 0
Threads 192 - 223, Block 0
Threads 224 - 255, Block 0
Threads 0992 - 1023, Block 0
Threads 000 - 031, Block 168
Threads 032 - 063, Block 168
Threads 064 - 095, Block 168
Threads 096 - 127, Block 168
Threads 128 - 159, Block 168
Threads 160 - 191, Block 168
Threads 192 - 223, Block 168
Threads 224 - 255, Block 168
Threads 0992 - 1023, Block 168
blockDimx.y
gridDimx.y
gridDimx.x

Figure 7.4: CUDA Thread Blocks for Convolution Operation

NVIDIA Kepler GK110b GPU, for example, has 15 SM units, and each unit has 16 blocks of threads. Thats a total of 240 blocks of thread that can be launched at a given instant (our application requires 169). Thus, CUDA allows the programmer to write the kernel in a serial fashion. Depending on the application, the programmer has to decide the number of threads and blocks that has to launched, which plays a critical part in determining the runtime of the algorithm.

## 7.5 Parallel Implementation of the Hough Transform

We have already described the Hough transform, in detail, in Chapter 4. To outline, the Hough transform is a popular technique to locate shapes (predominantly lines and circles) in images. The technique can be broken down into the following parts:

- Edge Detection
- Thresholding
- Voting in the Hough Space
- Hough Space Post Processing
- Drawing the Dominant Line

Edge detection entails convolving the image with a special kernel called the Sobel Kernel. We have already seen how to do this. The parallel implementation of thresholding an image is also well documented [6], the most popular one being Otsus binarization. Steps 3 and 4 are now handled by two separate kernels on the GPU.

The first kernel creates an array of pixels that have to be used in the voting process. If the pixel value is greater than a specified threshold, its coordinate value is stored in an array on the on-chip shared memory. Every thread in a warp reads a pixel. If the pixel value is less than the threshold, the corresponding thread does not remain idle but rather updates the location on the array where the valid pixel coordinates are to be stored. And since the actually number of pixels that count towards voting is significantly smaller (only 10 percent ) compared to the total number of pixels in the image, all of the valid pixels can be stored in the fast on-chip shared memory. This avoids the very costly step of having to store values in the off-chip global memory (usually consumes 400-600 GPU cycles).

The second kernel uses a single thread block to create a single line for every element in the array generated by the first kernel in the two Hough spaces (recall Cartesian parameterization has one space for lines with angles between -45 degrees to +45 degrees, and another space for angles between 45 degrees to 135 degrees). The number of lines depends on the accuracy of the angle parameterization. Each line is first put together in the shared memory and then transferred to the global memory.

The CUDA code and the array creating process are detailed in [35].

A comparison between the different processing times for different parts of the line detection algorithm is shown in the table below. The image sixe is 576 x 324 .The computational hardware on the CPU side involves an Intel i7, 1.73 Ghz processor with 8 GB of RAM. On the GPU side (GeForce GTX 970M), the specification included 16 Streaming Multiprocessors, 280 cores, 960MHz base clock frequency and 6GB of global memory. The total processing time in the CPU is 122 ms, while that on the GPU is 18 ms. Thus an effective speed up in processing time of 7x is observed when using the GPU.

| **Process** | **CPU (ms)** | **GPU (ms)** | **Speed Up** |
|---|---|---|---|
| Image Resizing | 12.35 | 1.58 | 8 |
| Noise Removal | 9.66 | 3.3 | 9 |
| Image Gradients | 16.67 | 5.6 | 9 |
| Thresholding | 15.54 | 5.04 | 3 |
| Hough Detection | 64.21 | 2.64 | 25 |

Table 7.1: Comparison between GPU and CPU Processing Times

## 7.6 Size, Weight and Power Constraints of Embedded GPU Boards

NVIDIA Jetson TX2 is a modern computing platform with a CPU and a GPU. Its features include.

- GPU : 256 Cores, 1024 Threads/Core
- CPU : ARM A57 processor + HMP Dual Denver architecture
- Memory : 8GB DDR4
- GPU Memory Bandwidth : 59.7 GB/s
- Video : 4K x 2K resolution, 60 FPS, H264 Encoder/Decoder
- Power Requirement: 19 Volts, 4.74 Amps (Max 90W)
- Weight: 1.58 Kg

Let us analyze how the addition of the Jetson board affects the performance of the drone. Typically an on-board embedded processor used on drones weigh 350g

The Jetson board is 4.5 times heavier than traditional embedded processors. This certainly places a limitation on the maximum payload capacity of the drone, if the thrust to weight ratio is to remain constant. The Thrust to Weight ratio can be used to assess the agility of a quadrotor. Although the required agility depends largely on the application of the quadrotor, a thrust to weight ratio of 2.7 can be used as a rule of thumb to categorize a quadrotor as agile. So if a quadrotor weighs 1.5 kg, the addition of the Jetson board increases its weight to 3kg. This means that the 4 motors must produce a combined thrust of (3000g x 2.7) 8100g to maintain the agility of the quadrotor.

The size of the Jetson board is 17cm x 17 cm. In addition to this, the central frame of the drone must also accommodate for the autopilot , battery and sensors (which would approximately require an additional 15 x 15 cm area).

If we assume that a typical quadrotor requires 200 W (each motor requires 50W of power) of power to lift 1 kg of mass, the Jetson board would consume 316W (200W/kg * 1.58 kg) of power for mobility and an additional 90W for operation. Lithium Polymer (Li-Po) batteries are popular among drones because of their high specific power, moderate specific energy, low cost, scalability, and high cycle life [43] In order to understand the effect of the Jetson board on the drone flying time, let us assume the battery energy capacity to be 200 Wh, the drone weight prior to the addition of the Jetson board be 1.5 kg, each motor consumes 50W of power, and the on board computer and sensors consume an additional 5W. Total Power consumed by the drone = (1 * 200W/kg) + (50*4) + 5 = 405W. Flying time = 200 Wh/405 30 min.

- Total = (1+1.58)*200W/kg + (50 * 4) + 5 + 90 = 811W
- Flying time = 200 Wh/811W = 15 min

Thus the flying time is cut down by half due to the power requirements of the Jetson board. The situation can be slightly improved if we are able to choose a battery with a higher discharge capacity while ensuring its weight doesnt increase significantly.

## 7.7 Summary and Conclusion

In this chapter, we have explored NVIDIA GPU's hardware and software architecture. We have described how to parallelize the 2-D convolution operation using CUDA C programming language. We have briefly described the mechanism by which the line detection algorithm (Hough Transform) can be parallelized. A *7X* effective speed up in processing time was observed when comparing the GPU implementation with that of a single threaded CPU implementation. And finally the potential of a modern embedded GPU processor as an outer loop controller on the drone was analyzed with respect to the drone's size, weight and power constraints. The addition of the GPU board cut down the flying time of the drone to half of it's original value.

Chapter 8

# SUMMARY AND FUTURE DIRECTIONS

## 8.1 Summary of Work

This thesis addressed perception issues that are important to achieve the longer-term *FAME* objective. The following summarizes key themes within the thesis.

1. **Literature Survey.** A brief literature survey of relevant work was presented.

2. **FAME Architecture.** A general *FAME* architecture has been described.

3. **Image processing.** A detailed description of the image processing pipeline was presented

4. **Extended Kalman filter.** The design and effectiveness of the Extended Kalman filter for tracking application was presented.

5. **GPU.** The potentail speed up that can be achieved by parallelizing our vision algorithm was explored.

## 8.2 Directions for Future Research

Future work will involve each of the following:

- **Onboard Sensing.** Addition of multiple onboard sensors; e.g. additional ultrasonics, camera, lidar, GPS, etc.

- **Multi-Vehicle Cooperation.** Cooperation between ground, air, and sea vehicles - including quadrotors, micro-air vehicles and nano-air vehicles [53] .

- **Parallel Onboard Computing.** Use of multiple processors on a robot for computationally intensive work; e.g. onboard optimization and decision making.

- **Modeling and Control.** More accurate dynamic models and control laws. This can include the development of multi-rate control laws that can significantly lower sampling requirements.

- **Guaranteed Performance.** Go beyond classical control techniques and use $\mathcal{H}^\infty$ methods to synthesize controllers to achieve stabilization with guaranteed performance [51; 54; 55; 56; 57; 58; 59; 60].

- **Control-Centric Vehicle Design.** Understanding when simple control laws are possible and when complex control laws are essential. This includes knowing how control-relevant specifications impact (or can drive) the design of a vehicle [52].

- **Simultaneous Localization and Mapping.** Concurrently estimate the pose of the robot and the map (sparse/dense) of the surrounding environment [45; 46; 47; 48; 49]

- **Reinforcement Learning.** Use the multiple hours of collected flying data to design an end to end (pixel to motor control) deep reinforcement system for navigating previously unseen indoor corridors [50].

## REFERENCES

[1] R. W. Brockett, “Asymptotic Stability and Feedback Stabilization,” in R. W. Brockett, R. S. Millman, and H. J. Sussmann, editors, *Differential Geometric Control Theory*, Birkhauser, Boston, MA, 1983

[2] P.I. Corke, “Robotics, Vision and Control,” *Springer*, 2011.

[3] J. Gao and Y. Zhang, “An Improved Iterative Solution to the PnP Problem,” *International Conference on Virtual Reality and Visualization*, 2013.

[4] G. Bradski and A. Kaehler, “Learning OpenCV: Computer vision with the OpenCV Library,” *O’Reilly Media, Inc.*, 2008.

[5] B. Jähne, *Practical Handbook on Image Processing for Scientific and Technical Applications*, CRC Press, Second Edition, 2004.

[6] R. Laganire, *OpenCV 2 Computer Vision Application Programming Cookbook*, Packt Publishing, 2011.

[7] L. Shapiro, G. Stockman, *Computer Vision*, 2000.

[8] J.E. Solem, *Programming Computer Vision with Python*, 2012.

[9] R. Szeliski, *Computer Vision Algorithms and Applications*, Springer, 2011.

[10] A.A. Rodriguez, *Analysis and Design of Feedback Control Systems*, Control3D,L.L.C., Tempe, AZ, 2002.

[11] A.A. Rodriguez, *Linear Systems: Analysis and Design*, Control3D,L.L.C., Tempe, AZ, 2002.

[12] I. Anvari, “Non-holonomic Differential-Drive Mobile Robot Control & Design: Critical Dynamics and Coupling Constraints”, Arizona State University, MS Thesis, 2013.

[13] Z. Lin, “Modeling, Design and Control of Multiple Low-Cost Robotic Ground Vehicles,” Arizona State University, MS Thesis, 2015.

[14] Z. Li, “Modeling and Control of a Longitudinal Platoon of Ground Robotic Vehicles,” Arizona State University, MS Thesis, 2016.

[15] M. Brambilla, E. Ferrante, M. Birattari, et al., “Swarm robotics: A review from the swarm engineering perspective,” Swarm Intelligence, 2013, 7(1): 1-41.

[16] B. Siciliano, L. Sciavicco, L. Villani, et al., Robotics: Modeling, Planning and Control, Springer Science & Business Media, 2009.

[17] D. Marr, S. Ullman, and T. Poggio, *Vision: A Computational Investigation into the human representation and processing of visual information.* W.H. Freeman, 1982.

[18] Cooper Bills, Joyce Chen, and Ashutosh Saxena. “Autonomous MAV Flight in Indoor Environments using Single Image Perspective Cues”, *In Robotics and Automation (ICRA), 2011 IEEE conference on*, pp. 5776-5783. IEEE, 2011.

[19] Lucas Vago Santana, Alexandre Santos Brandao, Mario Sarcinelli-Filho and Ricardo Carelli. “A Trajectory Tracking and 3D Positioning Controller for the AR.Drone Quadrotor, *Unmanned Aircraft Systems (ICUAS), 2014 International Conference on.* IEEE, 2014.

[20] Lugo, Jacobo Jimnez, and Andreas Zell. “Framework for autonomous on-board navigation with the AR. Drone.” *Journal of Intelligent Robotic Systems 73* no.1-4 (2014): pp. 401-412.

[21] Sanders, Jason, and Edward Kandrot. *CUDA by example: an introduction to general-purpose GPU programming.* Addison-Wesley Professional, 2010.

[22] Hernandez, Andres, et al. “Identification and path following control of an AR. Drone quadrotor.” *System Theory, Control and Computing (ICSTCC), 2013 17th International Conference.* IEEE, 2013.

[23] Hernandez, Andres, et al. “Model predictive path-following control of an AR. Drone quadrotor.” *Proceedings of the XVI Latin American Control Conference (CLCA14), Cancun, Quintana Roo, Mexico.* 2014.

[24] Bristeau, Pierre-Jean, et al. “The navigation and control technology inside the ar. drone micro uav.” *IFAC Proceedings Volumes* 44.1 (2011): 1477-1484.

[25] Cook, Shane. *CUDA programming: a developer’s guide to parallel computing with GPUs.* Newnes, 2012.

[26] Engel, Jakob, Jrgen Sturm, and Daniel Cremers. “Camera-based navigation of a low-cost quadrocopter.” *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on.* IEEE, 2012.

[27] Young Woo-Seo, “Detection and Tracking the Vanishing point on the Horizon”, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, May 2014.

[28] Krajnk, Tom, et al. “AR-drone as a platform for robotic research and education.” *International conference on research and education in robotics.* Springer, Berlin, Heidelberg, 2011.

[29] Scaramuzza Davide and Friedrich Fraundorfer, “Visual odometry Part I: The First 30 years and Fundamentals”, *IEEE robotics and automation magazine*, vol. 18, no. 4, pp. 80-92, 2011.

[30] Scaramuzza Davide and Friedrich Fraundorfer, “Visual odometry Part II: Matching, Robustness, Optimization, and Applications”, *IEEE robotics and automation magazine*,vol. 19, no. 2, pp. 78-90, 2012.

[31] Robert Mahony, Vijay Kumar and Peter Corke, “Multirotor aerial vehicles”, *IEEE Robotics and Automation magazine*,vol. 20, no. 32, 2012.

[32] Daniel Warren Mellinger, “Trajectory Generation and Control for Quadrotors,” University of Pennsylvania PhD Thesis, 2012

[33] Piskorski, Stephane, et al. “Ar. drone developer guide.” *Parrot, sdk* 1 (2012).

[34] Thrun, Sebastian, Wolfram Burgard, and Dieter Fox. *Probabilistic robotics.* MIT press, 2005.

[35] Van den Braak, Gert-Jan, et al. “Fast hough transform on GPUs: Exploration of algorithm trade-offs.” *International Conference on Advanced Concepts for Intelligent Vision Systems.* Springer, Berlin, Heidelberg, 2011.

[36] Ballard, Dana H. “Generalizing the Hough transform to detect arbitrary shapes.” *Readings in computer vision.*1987. 714-725.

[37] Kiryati, Nahum, Yuval Eldar, and Alfred M. Bruckstein. “A probabilistic Hough transform.” *Pattern recognition* 24.4 (1991): 303-316.

[38] Quigley, Morgan, et al. “ROS: an open-source Robot Operating System.” *ICRA workshop on open source software.* Vol. 3. No. 3.2. 2009.

[39] OKane, Jason M. “A gentle introduction to ROS, independently published (2013).” *Electronic copies freely available from the authors website.*

[40] Meyer, Johannes, et al. “Comprehensive simulation of quadrotor uavs using ros and gazebo.” *International Conference on Simulation, Modeling, and Programming for Autonomous Robots.* Springer, Berlin, Heidelberg, 2012.

[41] Allusse, Yannick, et al. “Gpucv: an opensource gpu-accelerated framework forimage processing and computer vision.” Proceedings of the 16th ACM international conference on Multimedia. ACM, 2008.

[42] Dumoulin, Vincent, and Francesco Visin. “A guide to convolution arithmetic for deep learning.” *arXiv preprint arXiv:1603.07285* (2016).

[43] Mulgaonkar, Yash, et al. “Power and weight considerations in small, agile quadrotors.” *Micro-and Nanotechnology Sensors, Systems, and Applications VI.* Vol. 9083. International Society for Optics and Photonics, 2014.

[44] Singh, Brij Mohan, et al. “Parallel implementation of Otsus binarization approach on GPU.” *Int J Comput Appl* 32.2 (2011): 16.

[45] H. Durrant-Whyte and T. Bailey. "Simultaneous localization and mapping: part I," *in IEEE Robotics  Automation Magazine*, vol. 13, no. 2, pp. 99-110, June 2006.

[46] Bailey, Tim, and Hugh Durrant-Whyte. "Simultaneous localization and mapping (SLAM): Part II." *IEEE Robotics  Automation Magazine* 13.3 (2006): 108-117.

[47] Klein, Georg, and David Murray. "Parallel tracking and mapping for small AR workspaces." *Mixed and Augmented Reality, 2007. ISMAR 2007. 6th IEEE and ACM International Symposium on.* IEEE, 2007.

[48] Mur-Artal, Raul, Jose Maria Martinez Montiel, and Juan D. Tardos. "ORB-SLAM: a versatile and accurate monocular SLAM system." *IEEE Transactions on Robotics* 31.5 (2015): 1147-1163.

[49] Engel, Jakob, Thomas Schps, and Daniel Cremers. "LSD-SLAM: Large-scale direct monocular SLAM." *European Conference on Computer Vision.* Springer, Cham, 2014.

[50] Giusti, Alessandro, et al. "A machine learning approach to visual perception of forest trails for mobile robots." *IEEE Robotics and Automation Letters* 1.2 (2016): 661-667.

[51] Puttannaiah, Karan, et al. "Analysis and use of several generalized $\mathcal{H}^\infty$ mixed sensitivity frameworks for stable multivariable plants subject to simultaneous output and input loop breaking specifications."*Decision and Control (CDC), 2015 IEEE 54th Annual Conference on.* IEEE, 2015.

[52] Rodriguez, Armando A., et al. "Modeling, design and control of low-cost differential-drive robotic ground vehicles: Part I: Single vehicle study." *Control Technology and Applications (CCTA), 2017 IEEE Conference on.* IEEE, 2017.

[53] Rodriguez, Armando A., et al. "Modeling, design and control of low-cost differential-drive robotic ground vehicles: Part II: Multiple vehicle study." *Control Technology and Applications (CCTA), 2017 IEEE Conference on.* IEEE, 2017.

[54] Nandola, Naresh N., and Karan Puttannaiah. "Modeling and predictive control of nonlinear hybrid systems using disaggregation of variables-A convex formulation." *Control Conference (ECC), 2013 European.* IEEE, 2013.

[55] Puttannaiah, K. *$\mathcal{H}^\infty$ control design via convex optimization: Toward a comprehensive design environment.* Diss. MS Thesis, Arizona State University, Tempe, AZ, 2013.

[56] Mondal, Kaustav. *Multivariable control of fixed wing aircrafts.* Arizona State University, 2015.

[57] Justin A. Echols, et al. “Fundamental control system design issues for scramjet-powered hypersonic vehicles.” *AIAA Guidance, Navigation, and Control Conference.* 2015.

[58] K.Puttannaiah, Justin A. Echols, and Armando A. Rodriguez. “A generalized $\mathcal{H}^\infty$ control design framework for stable multivariable plants subject to simultaneous output and input loop breaking specifications.” *American Control Conference (ACC), 2015.* IEEE, 2015.

[59] K.Puttannaiah, Armando A. Rodriguezet, et al. “A generalized mixed-sensitivity convex approach to hierarchical multivariable inner-outer loop control design subject to simultaneous input and output loop breaking specifications.” *American Control Conference (ACC), 2016.* IEEE, 2016.

[60] A. Sarkar and K. Puttannaiah and A. A. Rodriguez “Inner-Outer Loop based Robust Active Damping for LCL Resonance in Grid-Connected Inverters using Grid Current Feedback”. *American Control Conference (ACC), 2018.* IEEE,2018

# APPENDIX A

# SIMULATOR SETUP INSTRUCTIONS

## A.1 Instructions for Setting up the Ubuntu Environment

In order to play with all the features of the AR Drone simulator, it is recommended that you install the necessary packages from the source as several files were modified for the purposes of this project (vanishing point detection and tracking).The instructions below were implemented in Ubuntu 14.04, and should work for 16.04.

(a) The Primary goal is to install tum_simulator package, the open source ROS AR Drone simulator package. Unfortunately, at the time of this writing, the simulator is only compatible with ROS Fuerte.

(b) So we need to go back to rosbuild and rosws instructions instead of catkin (This is used in ROS Indigo and later versions)

(c) Install ROS-Fuerte desktop version `http://wiki.ros.org/fuerte/Installation/Ubuntu`

(d) Add source /opt/ros/fuerte/setup.bash. Add this statement to the end of .bashrc

(e) Install Gazebo (4x series is the Indigo compatible one (seems to work with Fuerte too), Fuerte compatible (1x series) was no longer available

(f) Create and Initialize ROS workspace
  - Rosws init $\tilde{/}$fuerte_ws optrosfuerte
  - Instructions in:`http://wiki.ros.org/fuerte/Installation/Overlays`

(g) Download fuerte branch of ardrone_autonomy
  - `https://github.com/AutonomyLab/ardrone_autonomy/tree/fuerte-devel`.
  - Follow instructions in README.md in the above repository.

(h) After inserting a package into the fuerte_ws (workspace), use rosws set package name (Note: package name is a placeholder, insert the name of the package in its place), and then **source /fuerte_ws/setup.bash** (The ROS workspace was named fuerte_ws). This will update ROS_PACKAGE_PATH. Lets call this step [*]

(i) Download tum_simulator fuerte branch
  - `https://github.com/tum-vision/tum_simulator/tree/fuerte`
  - Rosmake necessary packages using instructions in `http://wiki.ros.org/tum\_simulator`
  - If you are only interested in the simulations used in the thesis, then instead of the github repository, copy the local tum_simulator package (in the project folder given to you).

(j) Copy ros2cv package (custom package with vision modules, created for the purpose of detection vanishing points, can be found in the project folder) in your workspace. Repeat step[*]

(k) Run roslaunch cvg_sim_gazebo test.launch. Have fun playing the simulator!

## A.2 Project File Structure

The ROS workspace in which all the programs are implemented is named fuerte_ws. Your workspace must contain the following packages:

ardrone_ autonomyadrone_tutorials ros2cv tum_simulator

Executable python/C++ nodes are found under the *src* directory in each package.

## A.3 Instructions for Launching the Vanishing Point Detection, AR Drone Simulator and AR Drone Controller Nodes.

(a) Run roscore

(b) Run **roslaunch cvg_sim_test competition.launch**

(c) Run the vision module **rosrun ros2cv ros_to_cv.py**

(d) Run the module that publishes the altitude **rosrun ros2cv test_navdata.py**

(e) Run the drone controller module **rosrun ardrone_tutorials height_control.py**

## A.4 Creating Custom ROS messages

(a) Custom ROS messages were created in ros2cv package for publishing vanishing point coordinated

(b) For more information on how to create custom messages, please refer `http://wiki.ros.org/ROS/Tutorials/CreatingMsgAndSrv\#Creating\_a\_msg` and `http://wiki.ros.org/ROS/Tutorials/CustomMessagePublisherSubscriber\%28python\%29`

# APPENDIX B

# PYTHON CODE

```
# Program to extract lines in an image using probabilistic Hough transform

import cv2
import numpy as np
import imutils
from Kalman_Filter import matrix
from Kalman_Filter import EKF
import intersection_library

def HoughDetect_May_13(img,x,P,resolution = 1):

    # Read state information
    v_x = x.value[0][0]
    v_y = x.value[1][0]
    i_x = x.value[0][0]
    i_y = x.value[1][0]
    i_w = int(img.shape[1] * resolution)
    img = imutils.resize(img, width = i_w)
    i_h = int(img.shape[0])
    img_gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    img_gray = cv2.medianBlur(img_gray,7)
    img_gray = cv2.GaussianBlur(img_gray, (9,9), 0)

    # Find image gradient along the y axis
    sobely_64f = cv2.Sobel(img_gray, cv2.CV_64F, 0, 1 , ksize = 3)
    abs_sobelx64F = np.absolute(sobely_64f)
    abs_sobelx64F = abs_sobelx64F/ abs_sobelx64F.max() * 255
    sobely_8u = np.uint8(abs_sobelx64F)
    y_grad_threshold = sobely_8u.copy()


    # Find image gradient along the x axis
    sobelx_64f = cv2.Sobel(img_gray, cv2.CV_64F,1,0,ksize = 3)
    abs_sobelx = np.absolute(sobelx_64f)
    abs_sobelx = abs_sobelx/abs_sobelx.max() *  255
    sobelx_8u = np.uint8(abs_sobelx)

    # Find gradient direction
    #grad_magnitude = cv2.add(sobely_8u,sobelx_8u)
    grad_directions = np.arctan2(sobely_8u, sobelx_8u)


    # Fine tuning for AR drone front camera
    #y_grad_threshold[y_grad_threshold < 20] = 0
    y_grad_threshold[y_grad_threshold > 30] = 255


    # Threshold gradient image to process essential pixels only
    grad_directions[y_grad_threshold < 10] = -np.pi

    grad_directions_normalized = ((grad_directions*180/np.pi) \
                                 + 180)/360
    grad_directions_8u = np.uint8(grad_directions_normalized \
                                 * 255)


    # Hough Transform
    maxLineGap = 2
    adaptive = int(100*resolution)
    minLineLength = adaptive
    lines = cv2.HoughLinesP(grad_directions_8u,1,\
                 np.pi/180,250,minLineLength,maxLineGap)

    anys_1 = []
    anys_2 = []
    l_r = []
    l_l = []
    G_x = []
    G_y = []
    j=0
    m1 = 0
    m2 = 1000
    c1 = 0
    Grid_locations = []
    weight = []
    w_l = []
    w_r = []
    ka = i_h
    kb = (i_w)/2.0 + 1
    lines_new = []
    lines_EKF = []
    lines_inter = []
    lines_r_l = []
    lines_length = []

    # Yellow line to visualize vp change with
    # lateral camera movement
    cv2.line(img,(0,int(i_h/(3))),\
            (i_w,int(i_h/(3))),(255,255,0),2)



    # Select lines that satisfy application constraints
    if lines is not None:

        for feature in range(len(lines)):
            for x1,y1,x2,y2 in lines[feature]:
                if (x1- x2)!=0:
                    theta = np.arctan2((y2-y1),(x2-x1))
                    m2= np.tan(theta)
                    intercept_c1 = y1 - m2*x1
                    d_v = abs((v_y - m2*v_x) - \
                    (y1 - m2*x1))/\
                    np.sqrt(1+np.square(m2))
                    h_d = min(y1,y2) - v_y
                    const = 20*resolution

                    if True :
                        print ("Pass")
                        dist = np.sqrt(max((np.square(x1-v_x)+\
                        np.square(y1 - v_y)),(np.square(x2 - v_x)\
                        + np.square(y2 - v_y))))
                        l_mag = np.sqrt(np.square(x1 - x2)\
                        + np.square(y1 - y2))
                        lines_length.append(l_mag)
                        mid_1 = int((x1+x2)/2.0)
                        mid_2 = int((y1+y2)/2.0)
                        c2 = y1 - m2*x1
                        x3 = int(1000 + x1)
                        y3 = int(m2*x3 + c2)
                        x4 = int(x1 - 1000)
                        y4 = int(m2*x4 + c2)
                        k = mid_2 + (ka*mid_1) - (ka*kb)
                        k_new = mid_1 - (i_w/2)

                        if abs(theta) > 0.2 and l_mag > 20 \
                        and abs(theta) < 1.3:

                            lines_EKF.append((x1,y1,m2,c2,dist,l_mag))
                            lines_r_l.append((x3,y3,x4,y4))
                            cv2.line(img,(x3,y3),(x4,y4),(0,0,255),3)


    else:
        return img,x,P,x

    if len(lines_r_l)!=0:
        lines_all = intersection_library.\
                    lines_from_points(lines_r_l)

        for j in range(len(lines_all) - 1):
            lines_inter.append((lines_all[j],\
                    lines_all[j+1]))


    if len(lines_inter) != 0:
        i_x,i_y = intersection_library \
        .points_from_lines(lines_inter , x)




    # Line from intersection library
    cv2.circle(img,(i_x,i_y) ,\
            int(i_w/50),(0,255,0),3)


    lines_new = np.array(lines_EKF)

    measurements = matrix([[i_x],[i_y]])


    # Blue circle for EKF tracked vanishing points
    x,P = EKF(x,P,lines_new,measurements,resolution)
    v_1 = int(x.value[0][0])
    v_2 = int(x.value[1][0])

    cv2.circle(img,(v_1,int(i_h/3)) \
                ,int(i_w/48),(255,0,0),3)

    return img,x,P,measurements
```

```
# Module to compute the intersection point

import numpy as np


def cross_product(x,y):
    res = np.dot(np.matrix([[0,-x[2],x[1]],\
    [x[2],0,-x[0]],[-x[1],x[0],0]]),\
    np.array(y).T)
    return np.array(res)

def lines_from_points(points):
    lines = []
    left_limit = min(min(zip(*points)[0]), \
                min(zip(*points)[2]))
    right_limit = max(max(zip(*points)[0]),\
                max(zip(*points)[2]))

    for x1,y1,x2,y2 in points:
        point_1 = np.array([x1,y1,1])
        point_2 = np.array([x2,y2,1])
        line = cross_product(point_1 \
              , point_2)
        lines.append(line)

    return lines

def points_from_lines(lines, state):
    intersections = []

    for line_right, line_left in lines:
        right_slope = - (line_right[0][0]/\
                         line_right[0][1])
        left_slope = - (line_left[0][0]/\
                         line_left[0][1])
        intersection = cross_product(line_right[0]\
                                     ,line_left[0])
        if intersection[0][2] != 0  \
          and (left_slope != right_slope):
          intersection = intersection/\
                            intersection[0][2]
          intersection = np.array(intersection)
              if intersection[0][0] >= 0 and\
                       intersection[0][1] >= 0:
            intersections.append(intersection[0])

    if len(intersections) != 0:
        x_cordinates = zip(*intersections)[0]
        y_coordinates = zip(*intersections)[1]


        v_x = int(np.median(x_cordinates))
        v_y = int(np.median(y_coordinates))

    else:
        v_x = int(state.value[0][0])
        v_y = int(state.value[1][0])

    return v_x,v_y
```

```
# Write a function that implements a multi-
# dimensional Kalman Filter
import numpy as np

class matrix:

    # Implements basic operations of a matrix class

    def __init__(self, value):
        self.value = value
        self.dimx = len(value)
        self.dimy = len(value[0])
        if value == [[]]:
            self.dimx = 0

    def zero(self, dimx, dimy):
        # check if valid dimensions
        if dimx < 1 or dimy < 1:
            raise ValueError, "Invalid size of matrix"
        else:
            self.dimx = dimx
            self.dimy = dimy
            self.value = [[0 for row in range(dimy)]\
            for col in range(dimx)]

    def identity(self, dim):
        # check if valid dimension
        if dim < 1:
            raise ValueError, "Invalid size of matrix"
        else:
            self.dimx = dim
            self.dimy = dim
            self.value = [[0 for row in range(dim)]\
            for col in range(dim)]
            for i in range(dim):
                self.value[i][i] = 1

    def show(self):
        for i in range(self.dimx):
            print self.value[i]
        print ' '

    def __add__(self, other):
        # check if correct dimensions
        if self.dimx != other.dimx or \
         self.dimy != other.dimy: \
            raise ValueError, "Matrices must be of \
                            equal dimensions to add"
        else:
            # add if correct dimensions
            res = matrix([[]])
            res.zero(self.dimx, self.dimy)
            for i in range(self.dimx):
                for j in range(self.dimy):
                    res.value[i][j] = self.value[i][j] \
                    + other.value[i][j]
            return res

    def __sub__(self, other):
        # check if correct dimensions
        if self.dimx != other.dimx or self.dimy != other.dimy:
            raise ValueError, "Matrices must be of equal//
            //dimensions to subtract"
        else:
            # subtract if correct dimensions
            res = matrix([[]])
            res.zero(self.dimx, self.dimy)
            for i in range(self.dimx):
                for j in range(self.dimy):
                    res.value[i][j] = self.value[i][j] \
                                - other.value[i][j]
            return res

    def __mul__(self, other):
        # check if correct dimensions
        if self.dimy != other.dimx:
            raise ValueError, "Matrices must be m*n //
            //and n*p to multiply"
        else:
            # multiply if correct dimensions
            res = matrix([[]])
            res.zero(self.dimx, other.dimy)
            for i in range(self.dimx):
                for j in range(other.dimy):
                    for k in range(self.dimy):
                        res.value[i][j] += self.value[i][k] \
                                    * other.value[k][j]
            return res

    def transpose(self):
        # compute transpose
        res = matrix([[]])
        res.zero(self.dimy, self.dimx)
        for i in range(self.dimx):
            for j in range(self.dimy):
                res.value[j][i] = self.value[i][j]
        return res

    def Cholesky(self, ztol=1.0e-5):
        # Computes the upper triangular Cholesky factorization of
        # a positive definite matrix.
```

```python
        res = matrix([[]])
        res.zero(self.dimx, self.dimx)

        for i in range(self.dimx):
            S = sum([(res.value[k][i])**2 for k in range(i)])
            d = self.value[i][i] - S
            if abs(d) < ztol:
                res.value[i][i] = 0.0
            else:
                if d < 0.0:
                    raise ValueError, "Matrix not//
                    //positive-definite"
                res.value[i][i] = np.sqrt(d)
            for j in range(i+1, self.dimx):
                S = sum([res.value[k][i] * res.value[k][j] \
                for k in range(self.dimx)])
                if abs(S) < ztol:
                    S = 0.0
                res.value[i][j] = (self.value[i][j] - S)\
                        /res.value[i][i]
        return res

    def CholeskyInverse(self):
        res = matrix([[]])
        res.zero(self.dimx, self.dimx)

        # Backward step for inverse.
        for j in reversed(range(self.dimx)):
            tjj = self.value[j][j]
            S = sum([self.value[j][k]*res.value[j][k] \
                    for k in range(j+1, self.dimx)])
            res.value[j][j] = 1.0/tjj**2 - S/tjj
            for i in reversed(range(j)):
                res.value[j][i] = res.value[i][j] = \
                -sum([self.value[i][k]*res.value[k][j] \
                for k in range(i+1, self.dimx)])/self.value[i][i]
        return res

    def inverse(self):
        aux = self.Cholesky()
        res = aux.CholeskyInverse()
        return res

    def __repr__(self):
        return repr(self.value)

# Extended Kalman filter function
def EKF(x,P,lines,measurements,resolution,count=0):
count = 0
I = matrix([[1., 0.], [0., 1.]])
for x1,y1,m1,c1,d1,l1 in lines:
    theta = np.arctan2(measurements.value[1][0] - y1,\
            measurements.value[0][0] - x1)
    h_theta = np.arctan2((x.value[1][0] - y1),\
               (x.value[0][0] - x1))
    y = theta - h_theta
    d_square = np.square(x.value[0][0] - x1) \
                + np.square(x.value[1][0] - y1)
    H = matrix([[-(x.value[1][0] - y1)/d_square,\
            (x.value[0][0] - x1)/d_square]])
    omega = abs(measurements.value[0][0]\
            - x.value[0][0])

    if omega > 3:
        factor = (omega/3.0 * np.pi) % (np.pi - 0.01)
        R = matrix([[factor**2 + 0.01]])
    else:
        R = matrix([[0.01]])

    if count < 5:
        if abs(y) < (1):
            count+= 1
            # Measurement Model
            S = H*P*H.transpose() +  R
            K = P*H.transpose()*S.inverse()
            y = matrix([[y]])
            x = x + K*y
            P = (I - K*H)*P
        else:
            P = matrix([[1250., 0.], \
                    [0., 1250.]])
    else:
        if abs(y) < (.2):
            count+= 1
            # Measurement Model
            S = H*P*H.transpose() +  R
            K = P*H.transpose()*S.inverse()
            y = matrix([[y]])
            x = x + K*y
            P = (I - K*H)*P
        else:
            P = matrix([[1250., 0.], [0., 1250.]])
return x,P
```

```
# Program to Convert ROS Image to OpenCV image and detect Vanishing Points

#!/usr/bin/env python
from __future__ import print_function
import rospy
from sensor_msgs.msg import Image, CameraInfo
import numpy as np
import cv2
import sys
from cv_bridge import CvBridge, CvBridgeError
from Robust_detection import HoughDetect_May_13
from Kalman_Filter import matrix


class cvBridgeDemo():
    def __init__(self):
        self.node_name = "ros_to_opencv"

        # Initialize ROS node
        rospy.init_node(self.node_name)

        # During Shutdown
        rospy.on_shutdown(self.cleanup)

        # Modifying the code to update state and uncertainty
        self.res = 1
        self.state = matrix([[320. * self.res],[160. * self.res]])
        self.uncertainty = matrix([[1000., 0.], [0., 1000.]])
        self.initial_vx = 320


        # Create the cv_bridge object
        self.bridge = CvBridge()

        # Subscribe to camera image and set the appropriate callbacks
        self.image_sub = rospy.Subscriber("/ardrone/image_raw", \
                                Image, self.image_callback)


        rospy.loginfo("Waiting for image topics...")




    def image_callback(self, ros_image):

    # Use cv_bridge() to convert the ROS image to OpenCV format
        try:
            frame = self.bridge.imgmsg_to_cv2(ros_image,"bgr8")
        except CvBridgeError as e:
            print (e)

        #Convert image to numpy array
        frame = np.array(frame, dtype = np.uint8)

        #Process the frame
        temp = matrix([[1.0],[1.0]])
        display_image,x,P = self.process_image(frame)
        self.state = x
        self.uncertainty = P
        print ("Vanishing Point coordinates, \
              from EKF = {0}".format(x))

        print ("Intial horizontal vanishing Point \
              coordinate = {0}".format(self.initial_vx))

        time_present = rospy.get_time()

        #Display the image

        cv2.namedWindow('Window', 1)
        cv2.setWindowProperty('Window',\
                  cv2.WND_PROP_FULLSCREEN, 1)
        cv2.imshow("Window",display_image)

        #Process keyboard command
        self.keystroke = cv2.waitKey(1)
        if 32 <= self.keystroke and self.keystroke < 128:
            cc = chr(self.keystroke).lower()

            if (cc == 'q'):
                rospy.signal_shutdown("User has,\
                            hit q to exit")
            if (cc == 'r'):
                self.initial_vx = x.value[0][0]

def process_image(self,frame):
    img_result,x,P,_= HoughDetect_May_13(frame,\
    self.state,self.uncertainty,\
    resolution = self.res)
    return img_result,x,P


def cleanup(self):
    print ("Shutting down ,\
    vision node")
    cv2.destroyAllWindows()


def main(args):
    try:
        cvBridgeDemo()
        rospy.spin()
    except KeyboardInterrupt:
        print ("Shutting down,\
                vision node”)
        cv2.destroyAllWindows()

if __name__ == '__main__':
    main(sys.argv)
```

```
# Program to fly the drone in the TUM Simualtor

#!/usr/bin/env python

# Import the ROS libraries, and load the manifest
# file through which <depend package=... /> \
# will give us access to the project dependencies
import roslib; roslib.load_manifest('ardrone_tutorials')
import rospy
import numpy as np
import cv2
from ros2cv.msg import AR_ALTD
from ros2cv.msg import Vpoint
import sys
# Load the DroneController class, which handles
# interactions with the drone
from drone_controller import BasicDroneController
import time
from Pan_tilt import retrieve_angle



def main(args):
    global altd
    global past_altd
    global vx
    global vy
    global vx_past
    global past_pan
    global num_lines
    global key
    global count
    global past_num_lines
    try:
       rospy.init_node('Height_controller')
       subNavdata = rospy.Subscriber('Drone_Altitude'\
                    ,AR_ALTD, ReadHeight)
    ##########################################################
       subVpoint = rospy.Subscriber('vanishing_point',\
                    Vpoint, ReadVpoint)
    ##########################################################
       controller = BasicDroneController()
       rospy.sleep(1)
       controller.SendTakeoff()
       print('Takeoff')
       rospy.sleep(2)
       while not rospy.is_shutdown():
        vanish_point = np.array([[vx],[180],[1]])
        pan, tilt = retrieve_angle(vanish_point, res = 0.5)
        pan = pan * 180./np.pi
        tilt = tilt * 180./np.pi
        # working param kp = 0.02 and desired pan = 0
        cmd_yaw =  0.02 * (-0.5 - pan) + 0.0 * (past_pan - pan)
        cd_roll = 0
        cd_pitch = 0.1
        R_pan = np.matrix([[np.cos(pan * np.pi/180),\
              -np.sin(pan * np.pi/180)], [np.sin(pan * np.pi/180) \
              , np.cos(pan * np.pi/180)]])
        cd_rp = np.array([[cd_roll],[cd_pitch]])

        tf_cmds = np.array(np.dot(R_pan, cd_rp))
        cmd_roll = tf_cmds[0][0]
        cmd_pitch = tf_cmds[1][0]

                # Good parameters 0.05 and 0.9
        cmd_z = 0.05 * (1000 - altd)/100  + 0.9 \
                  * (past_altd - altd)/100

        cmd_z = max(min(cmd_z,0.2),-0.2)

        print ('Command yaw = {0}'.format(cmd_yaw))
        controller.SetCommand(roll = cmd_roll,\
                  pitch = cmd_pitch,yaw_velocity = \
                  cmd_yaw, z_velocity = cmd_z)


        if (num_lines) < 10 and count > 1000:
            rospy.sleep(1)
            controller.SendLand()
            rospy.sleep(1)
            rospy.signal_shutdown\
                     ("Landing Requested")
            rospy.signal_shutdown('Great Flying!')

            count+=1
            past_altd = altd
            past_pan = pan
            vx_past = vx
            past_num_lines = num_lines


    except KeyboardInterrupt:
        rospy.signal_shutdown\
              ("Landing Requested")
        rospy.sleep(1)
        rospy.SendLand()
        rospy.signal_shutdown\
              ('Great Flying!')




def ReadHeight(data):
    global altd
    altd = data.height

def ReadVpoint(img_data):
    global vx
    global vy
    global key
    global num_lines
    vx = img_data.vpoint1
    vy = img_data.vpoint2
    key = img_data.key
    num_lines = img_data.nlines


# Setup the application
if __name__=='__main__':
    altd = 0
    past_altd = 0
    past_pan = 0
    vx = 160
    vy = 100
    vx_past = 160
    num_lines = 10
    past_num_lines = 0
    count = 0
    try:
        main(sys.argv)
    except rospy.ROSInterruptException:
        pass
```