TÉCNICO LISBOA

Distributed Architecture for Sensorial Implementation

Jorge Tiago da Rocha Afonso

Thesis to obtain the Master of Science Degree in

Aerospace Engineering

Supervisor(s): Prof. Paulo Oliveira
Prof. Carlos Cardeira

Examination Committee

Chairperson: Prof. José Fernando Silva
Supervisor: Prof. José Eduardo Sanguino
Member of the Committee: Prof. Carlos Cardeira

July 2019

Acknowledgments

To Professor Paulo Oliveira and Professor Carlos Cardeira for all the knowledge, time and support, without whom this work would have been impossible.

To my family for all their support, and to my dear girlfriend for all her patience and dedication.

Resumo

Nesta dissertação aborda-se o design, implementação e avaliação de um sistema distribuído, destinado a suportar o desenvolvimento de algoritmos avançados de navegação e controlo. Este sistema foi desenvolvido por forma a ser instalado na arena robótica do laboratório de Automação e Robótica do Instituto Superior Técnico. É composto por vários componentes, de onde se destacam um sistema de localização com capacidade para detetar e seguir pontos num referencial tridimensional, uma interface gráfica intuitiva e de utilização simples, uma frota de veículos aéreos e terrestres adaptados de forma a permitir a modificação dos seus algoritmos de navegação e controlo.

O sistema de localização passou por vários estados de desenvolvimento diferente. Culminaram numa arquitetura composta por várias câmaras remotas equipadas com sensores de profundidade Kinect. Foram utilizados marcadores coloridos para identificar os pontos de interesse que o sistema deve seguir. Todos os métodos e algoritmos utilizados nas fases de deteção e seguimento são apresentados ao pormenor e discutidos. Apesar da frota de veículos ser composta por dois tipos de veículos, terrestres e aéreos, apenas o segundo tipo foi abordado no âmbito desta tese. O desenvolvimento dos veículos terrestres foi elaborado numa outra tese de mestrado, Nuno Martins[55]. Os veículos aéreos utilizados foram drones AR.Drone 2.0. Estes foram alterados de forma a permitirem ser comandados por algoritmos a correr externa ou internamente. O processo de análise de todo o hardware e software do drone encontra-se documentado, assim como o seu funcionamento, e o porquê das várias alterações feitas ao seu sistema operativo original. São apresentados todos os aspetos relativamente à comunicação e partilha de dados , incluindo uma descrição pormenorizada de todos os protocolos e mensagens usados.

Finalmente, foi feita uma avaliação ao desempenho final do sistema com destaque à exactidão do sistema de localização. Apesar de alguns problemas de performance terem sido identificados, o sistema apresentou bons resultados, concluindo-se que pode ser utilizado para o desenvolvimento e teste de algoritmos de navegação e controlo com confiança. Para os problemas encontrados, foram sugeridas soluções.

Abstract

This dissertation addresses the design, implementation, and evaluation of a distributed system conceived to support the development of advanced navigation and control algorithms. It features a tracking system capable of detecting and follow specialized markers in three-dimensional space, a simple graphical user interface to visualize real-time data and configure system parameters, and a fleet of customized airborne and grounded vehicles. The final system was installed on the robotics arena of the Automation and Robotics Laboratory of Instituto Superior Técnico.

The location system detects specialized markers, using classic machine vision techniques. These techniques are applied on images and depth maps collected by several remote cameras equipped with depth sensors. Colored markers were used to identify the points of interest to be tracked by the system. All methods and algorithms used for detection and tracking are presented and discussed in detail. Despite the fact that two types of vehicles were used, grounded and airborne, only the second type was addressed in the context of this thesis. The development of land vehicles was address in another master's thesis by Nuno Martins[55]. The aircraft used was the AR.Drone 2.0 drone by Parrot. The internal operating system of these drones had to be customized to fulfill complete system integration, allowing for third-party control and navigation algorithms to be installed and executed. The drone's hardware and software exploration process was documented, as well as all of the changes to its factory operating system. All aspects, regarding communication and data sharing, are explained, including a detailed description of all the messages and protocols used.

Finally, a performance evaluation was made, particularly focused on the localization system accuracy. Although some performance problems were identified, the system showed good results in fulfilling its main goal of being a platform to develop and test advanced navigation and control techniques. To the problems encountered, solutions were also suggested and discussed.

Contents

List of Figures

List of Algorithms

List of Tables

Nomenclature

2D	Two-dimensional
3D	Three-dimensional
ACK	Acknowledgement
ADC	Analog to Digital Converter
ADIS	Arquitectura Distribuída para Implementação Sensoria
API	Application Programming Interfaces
bash	Unix shell
CAMShift	Continuously Adaptive Mean Shift
DC	Direct Current
DOF	Degrees of Freedom
DSP	Digital Signal Processor
EOM	End of Message
ESSID	Extended Service Set Identification
FAT32	File Allocation Table 32 bit
FPS	Frames per Second
FTP	File Transfer Protocol
GB	Giga-Bytes
GHz	Giga-Hertz
GLONASS	Global Navigation Satellite System
GPS	Global Positioning System
HD	High Definition
HOG	Histogram of Oriented Gradients)
HSV	Hue, Saturation and Value color space
HTML	HyperText Markup Language
I/O	Input/Output
IDE	Integrated Development Environment

IMU	Inertial Measuring Unit
IP	Internet Protocol
LAN	Local Area Network
LED	Light-emitting Diode
mAh	Milliampere hour
MCU	Microcontroller Unit
MHz	Mega Hertz
MIPs	Microprocessor without interlocked pipeline stages
mm	millimeters
PC	Personal Computer
QVGA	Quarter Video Graphics Array
RAM	Random Access Memory
RANSAC	Random Sample Consensus
RGB	Red, Green and Blue color space
rootfs	Root File System
RS-232	Standard for serial communication and transmission of data
SDK	Software Development Kit
SIFT	Scale-invariant feature transform
SSD	Solid-state Drive
SSH	Secure Shell
SURF	Speeded-Up Robust Features
TCP	Transmission Control Protocol
TTL	Transistor–Transistor Logic
UART	Universal Asynchronous Receiver-Transmitter
UDP	User Datagram Protocol
UEFI	Unified Extensible Firmware Interface
USB	Universal Serial Bus
V	Volts

Wi-Fi	Family of radio technologies for wireless local area networking based around IEEE 802.11 standards
WPA-EAP	Wi-Fi Protected Access - Extensible Authentication Protocol
WPA2	Wi-Fi Protected Access 2

1 Introduction

There is no doubt that scientific and technological advances have transformed our world and shaped our lives and the impact of technology in our world is undeniable.

Mobility has always been one of the areas of technology that changed our life the most. The invention of the automobile and its mass production ended an era where animal strength was the heart of the transportation industry and started a revolution where everyone could travel with no effort and transport goods efficiently. After the automobile, the new revolution on mobility was the airplane. It connected countries and reduced distances, making trips that used to last months to last just a few hours. Currently, the next step in mobility technology could be autonomous driving and navigation. Most vehicles still require at least one human driver, which represents not just an annoyance to the driver, but also a security risk. A complete autonomous vehicle could not only improve security and reduce crashes but also increase efficiency and speed in transportation. Autonomous or self-driving vehicles can also be used in many industrial applications like herbicide spraying, harvest cropping, automated package delivery, private security or even in fields where human lives are at risk like firefighting or military applications.

This kind of vehicles relies on multiple types of navigation and control algorithms to cruise, maintain stability and to make decisions. These algorithms need to be able to execute tasks like sense and avoid obstacles, optimize routes according to multiple factors such as energy efficiency or traffic, cooperate with other vehicles to share information like position and trajectory planning, prioritize tasks and actions to avoid imminent risks or to guarantee driving comfort, among others.

This thesis is not focused on navigation and control algorithms but rather on creating tools and infrastructure to support their development, laboratory implementation, and testing. When the goal is real-world use of navigation and control techniques, implementation, validation, and refinement beyond simulation are essential to go further than the initial theoretical construct. A controlled laboratory environment becomes then the natural next step in the development process. However, this requires specialized equipment or infrastructure capable of accommodating multiple controller types to be deployed and its behavior measured. With this in mind, the emphasis of this thesis is to create a development platform, for laboratory implementation and testing of advanced control algorithms aimed at grounded and airborne vehicles.

1.1 State of the Art

Typically a system to implement and test control and navigation algorithms has a certain degree of complexity and it's composed by multiple subsystems that fulfill different functions. The approach used to analyze the current technologies was to divide the State of the Art exploration into the following topics: Vehicles, Indoor Localization Systems, and relevant software techniques.

1.1.1 Vehicles

These last five years a considerable growth of hobbyist communities around embedded electronics and the commoditization of low cost, low power ARM-based processors, led to the appearance of many development platforms geared towards robotics and moving vehicles. These platforms commonly follow one of two approaches, they either come as a single development board format with a set of input and output interfaces that can be connected to sensors and actuators, or they come as fully assembled vehicles.

A wide range of options for development boards is available on the market, designed to be implemented in all kinds of applications and environments. Using processing power and functionality as differentiators, the simplest tier of boards are hobbyist grade 8-bit/16-bit micro-controller boards. These type of boards come in different shapes and sizes equipped with a wide range of microcontrollers. These are simple general purpose boards, ideal for less demanding tasks that require low power consumption. Usually with simple I/O interfaces, composed mainly by Digital I/O pins, to connect to digital interfaces like DC motor speed controllers, and Analogue I/O pins connected to ADC's to be used with analog interfaces, like analog speed sensors, these boards provide a reasonable amount of features at a lower cost. A multitude of models from many suppliers exist but one of the most popular for simple applications are the Arduino boards. These boards come mostly equipped with different micro-controllers from the AVR family, like the ATmega328P in the Arduino Uno, and are programmed using their own Arduino Software (IDE). Due to their popularity, there is a generous amount of resources available to support development in the shape of documentation and libraries. Also, many add-on modules exist to expand the board capabilities. These can be connected to multiple sensors and fitted with motors and wheels to create cheap moving robots and even add Wi-Fi connectivity.

The tier above the 8-bit/16-bit micro-controller is the 32-bit micro-controller boards. These although similar to the 8-bit boards, they come equipped with more powerful hardware and interfaces. The increase in computational power allows for more demanding tasks to be executed, albeit at the expense of higher power consumption. In this tier, licensed ARM designs like the ARM Cortex-M in the STM32 family of microcontrollers from STMicroelectronics, are very popular.

Even more capable options in terms of processing power are available. Single board computers such as the Raspberry Pi 3 Model B+ with its quad-core 64-bit ARMv8 based SoC, 1GB of LPDDR2 SDRAM and even Wi-Fi dual-band support can accommodate fairly powerful jobs while also being affordable. Control algorithms coupled with image processing techniques to steer simple moving platforms can be easily implemented taking advantage of the integrated CSI camera port and the 40-pin GPIO header. Unlike the previous micro-controller boards, these single board computers support the installations of a Linux based operating system. This comes with many advantages, due to the fact that the Linux kernel provides access to robust TCP/IP and Ethernet networking stacks for connectivity and wireless communication, and USB protocol support for easy peripheral integration. However running above Linux, where a scheduler handles load balancing, and not executing bare-metal applications, means that extra care is needed when time critical operations are required.

The highest tier of development boards for vehicles is aimed at the automotive industry. One example

of this is the NVIDIA DRIVE AGX platform. Built upon the NVIDIA Xavier architecture and TensorCore GPUs, the two available boards, the Xavier and the Pegasus, deliver high levels of computational power geared towards AI applications. They also provide standard automotive interfaces to fuse data from sensors like cameras, radar, and lidar. However, despite being considerably more powerful than the previously analyzed solutions, these boards are also much more expensive.

Although microcontrollers and single board computers are flexible platforms for development, they share the common disadvantage of not being off-the-shelf solutions for implementing control and navigation algorithms in a laboratory environment. They require the extra development cost of building a moving platform and sensor hub for them to interact with. As an off-the-shelf alternative that avoids this cost, radio-controlled drones represent an interesting alternative both in terms of implementation freedom and challenges.

The term drone most commonly describes flying vehicles with small dimensions and multiple vertical oriented rotors, capable of vertical takeoff and landing. Quadcopters are the most common type of drones available on the market. Most of these vehicles follow the same basic formulaic approach to their design. An onboard, low power, flight controller, interfaces with a set of brushless motors, to control rotor speed. Stable flight is achieved orchestrating the multiple propeller speeds according to data acquired from an onboard sensor hub. Inertial measuring units are mandatory for aircraft attitude detection. Other common sensors are high-definition cameras, altimeters, and GPS/GLONASS transceivers.

1.1.2 Localization Systems

When building a system to test control and navigation algorithms, the ability to locate a vehicle in space is essential. Localization systems using time-of-flight of communications with satellites, like GPS, are the most common solution. Unfortunately, GPS based systems do not work well in indoor environments, because microwave signals are easily scattered or absorbed by walls and roofs. Indoor localization solutions for the common public are not widely available yet. Some rough estimations can be done by analyzing the Wi-Fi signal strength of known hotspots, however, this method is not accurate enough to be used by a navigation controller to steer a moving vehicle. Wi-Fi signals work at 2.4 and 5 GHz frequencies making them highly susceptible to attenuation and distortion from nearby obstacles. This cripples the signal strength to distance relation and makes it highly environment dependent. Other radio-based solutions exist[2][3][4] or even acoustic[5]. Other popular localization methods are based on stereo camera techniques. This kind of solutions is commonly known as motion capture systems. The automation and robotics laboratory of the Mechanical Engineering Department of IST is equipped with one of these systems produced by Qualisys. This system is capable of producing measurements with extreme accuracy and precision. It uses custom cameras capable of reaching very high frame rates, 300 to 1 100 frames per second (depending on the resolution), with very low latency, around 4ms, to detect specialized markers that reflect infrared light. Then using stereoscopic vision techniques, it calculates the position of multiple markers in space. For these systems to work multiple cameras need to capture the same marker simultaneously. The biggest downside to these multi-camera motion capture systems is their purchasing and maintenance costs. Making them not suited to be used by

large groups of students or multiple researches, because damage can be done due to inexperience or careless handling, resulting in large repair costs.

1.1.3 Feature Detection

Feature detection techniques are used to extract characteristics of interest of an image. These characteristics depend on the final application of the algorithm. Commonly they are corners, edges or blobs. These characteristics are usually the starting point for image processing algorithms.

SIFT (Scale-invariant feature transform) Introduced in 2004 by David Lowe[6], SIFT is an image descriptor with applications in areas such as 3D mapping, gesture recognition, and object recognition. It generates a set of unique features that can be used to recognize unique shapes in images, independent of scale, rotation, illumination, and viewpoint. To generate these feature sets, SIFT explores multiple scale representation of an image by constructing what is called a scale space. A collection of multiple representations of the original image with progressively smaller sizes and intensities of Gaussian Blur applied. From the scale, space key points are evidenced using the second order derivative a Difference of Gaussian images (DoG) and identified as the maxima or minima in the DoG images. Low contrast key points are removed as well as edges. To achieve invariance to image rotation, an orientation is assigned to each key-point based on their neighborhood. The final set of key-points can than be stored in an object database and used as a “fingerprint” to identify the original object in the different images.

SURF (Speeded-Up Robust Features) SURF is an improved version of SWIFT introduced in 2006 by Bay, H., Tuytelaars, T. and Van Gool [7]. To build a scale-space, SWIFT approximates the Laplacian of Gaussians with a Difference of Gaussians. SURF goes further and approximates LoG with Box Filer. Convolution with box filter can be calculated with integral images in parallel for different scales, speeding up the scale-space creation. For scale and location relies on the determinant of the Hessian matrix. The final feature description of SURF is based on the sum of the Haar wavelet response around the point of interest which can be computed with the aid of the integral images.

HOG (Histogram of Oriented Gradients) Popularized in 2005 by Navneet Dalal and Bill Triggs [8], Histogram of Oriented Gradients assumes that the object appearance and shape can be described by the distribution of intensity gradients or edge directions. The descriptor is a concatenation of histograms compiled by dividing the image into small cells, and for the pixels within each cell, calculate a histogram of gradient directions. The histogram from each cell can be contrast-normalized against a measure of intensity across a larger region of the image, called block. This improves accuracy and invariance to changes in illumination and shadowing.

The HOG descriptor main application is human detection in images because as Dalal and Triggs discovered, coarse spatial sampling, fine orientation sampling, and strong local photometric normalization permits the individual body movement of pedestrians to be ignored so long as they maintain a roughly

upright position. HOG is also invariant to photometric and geometric transformations, except for object orientation.

1.1.4 Object Tracking

CAMShift (Continuously Adaptive Mean Shift) CAMShift [9] is an improved version of Mean Shift. Mean Shift works by moving a small window iteratively until the encapsulates the area with maximum pixel density. This iterative process detects the centroid of the encapsulated points and moves the window towards this centroid until they both align. CAMShift adds to the aforementioned principal by rescaling and reorienting the window after aligning its center and the points centroids. This improves the tracking performance when objects approach or distance them self's from the image plane.

Kanade–Lucas This method assumes that all pixels in the vicinity of a point of interest have similar motion [10]. Taking a small three by three patch around a point of interest produces nine over-determined equations of motion to solve. Typically Kanade-Lucas solves these equations with least square fit method, resulting in an optical flow vector (motion vector) per point of interest, that can be used to track each point through time. Because the image is divided into small patches big jumps from frame to frame become impossible to track. Kanade-Lukas deals with this problem with a pyramid approach. By reducing the resolution of the image consecutively, small motions disappear and large motions become small motions that can be tracked.

Kalman Filter and Hungarian Algorithm This method uses Kalman filters [56]as estimators of future positions and the Hungarian Algorithm[54] to assign these positions to tracked points. Each point has his own Kalman Filter that, each frame, will estimate its location using a state-transition model. This state-transition model is a mathematical representation of the point's motion. Then the Hungarian Algorithm is used to assign the estimated positions to observable points. This assignment allows tracking points from frame to frame because each Kalman is supervising on point and one point only.

1.2 Contributions

These thesis contributions came from all the challenges faced during the development process, and from the finalized system as well. In regards to the localization system, the following was created:

- an algorithm to detect specialized markers through three-dimensional space using a low-cost Kinect sensor;
- a simple calibration process using both Singular Value Decomposition and the RANSAC algorithm to calculate the adequate homographic transformations and floor positioning in relation to the Kinect and the inertial plane;
- a remote processing unit was created capable of handling all of the processing requirements to drive the Kinect;

- a custom software to handle all of the remote cameras tasks and communications, equipped with a text-based interface for debugging purposes;
- a custom software tool with a graphical user interface to manage all of the cameras, and process all of the acquired location data, with real-time preview of said data, video feed, depth information, and image processing results.

In regards to the vehicles, the following was created:

- a detailed description of how the AR.Drone 2.0 from Parrot works and the processes used to discover it;
- how to create an alternative Linux based root file system compatible with the AR.Drone 2.0.

The final system also provides a way to perform research about navigation and control algorithms applied to grounded and aerial vehicles. It enables many other scientific projects to be made, for example, while this thesis was being developed many groups of Mechanical Engineering students already used the system for experimental projects. These were made within the Optimum Control class. Some groups developed algorithms based on optimum control theory to keep a drone in a stable hovering state, while others were focused on optimizing a car's trajectory to take as little time as possible. As of the time of writing, there are multiple theses undergoing, that fully utilize the finalized system and both grounded and airborne vehicles.

1.3 Thesis Outline

This thesis is divided into nine chapters:

- **Chapter 1** is an introductory section where a short summary of this thesis is made and a state of the art is presented.
- **Chapter 2** starts by presenting a general overview of the complete system. The basic system operation is described, as well as all of its possible architectures and necessary components.
- **Chapter 3** address the localization system. It starts by presenting all of the major theoretical concepts applied and then explains all of the functional underpinnings and processes utilized for this system.
- **Chapter 4** details all of the information regarding the vehicle fleet. The drone's characteristics are listed along with a detailed exposition on how and why the drones were modified.
- **Chapter 5** documents the network topology and all of the necessary communication protocols.
- **Chapter 6** explores important details about software and tools created
- **Chapter 7** is a presentation and analysis of the tests made to the overall systems performance with special emphasis on the localization system.

- **Chapter 8** is a conclusion about all the work done and the results of this thesis.
- **Chapter 9** describes possible future work that can be done to improve the final result of this thesis.

2 System Overview

2.1 Name and Iconography

The name of the system appeared of the need to create an identifiable image for the final product of this thesis. Not just because a name and an identifiable image adds value to the final system but also because it is intended to be used by a lot of people. The name chosen was ADIS, and it is an acronym of this thesis title in Portuguese, **A**rquitectura **D**istribuída para **I**mplementação **S**ensorial, and it is pronounced ‘adi:ss. To go along with the name, an icon was created inspired by the letter A.

(a) ADIS Icon (b) ADIS splash screen

Figure 1: ADIS iconography

2.2 Basic Operation

The ADIS system is made to provide a wide framework and set of tools, for development and testing of navigation algorithms. To achieve this, some basic requirements were set in the beginning, to ensure that the user has at his disposal everything he needs for its development work. The basic requirements were:

- Access to reliable location data
- Access to telemetry vehicle data
- Easy to use tooling and interface

The first requirement dictates the need to know the controlled vehicle location in space. The location coordinates need to be measured in a user-defined inertial frame. This requirement is a must for a system of this nature since navigating a vehicle through space is a prime objective for any navigation controller. Defining if a 2D or a 3D inertial frame is used is also important. A 3D inertial frame implies a more complex system, both in terms of the necessary hardware and in terms of mathematical models to use. This requirement affected greatly the ADIS system final form and complexity.

The second requirement addresses the need to know the orientation and controlled vehicle’s inertial state. To ensure that a route or path is navigated correctly, the navigation system must control the attitude and inertial dynamics of the vehicle. For gathering this telemetry data, each vehicle must be equipped with a wide range of onboard motion sensors. Multiple options and sensor sets were available,

depending which vehicles were selected to be integrated into the ADIS system, however, when it came to the grounded vehicles capabilities and characteristics, most of the work was done and integrated by another colleague that resulted in his master thesis[55].

Finally, the third requirement is related to the ease of use of the final system. Since the ultimate goal of the ADIS system is to be used for academic development, it is imperative that it provides a simple and user-friendly workflow. It must grant the adequate abstraction layers to its theoretical and functional principals, as well as, accessible programming languages and structure. It is equally important that the resulting system is robust and reliable, to provide a cohesive experience.

Taking these requirements as guidelines, the finalized system can be divided into three different components:

- Localization System
- Vehicle Integration Layer
- Communication Layer

These components work together, like in figure 2.

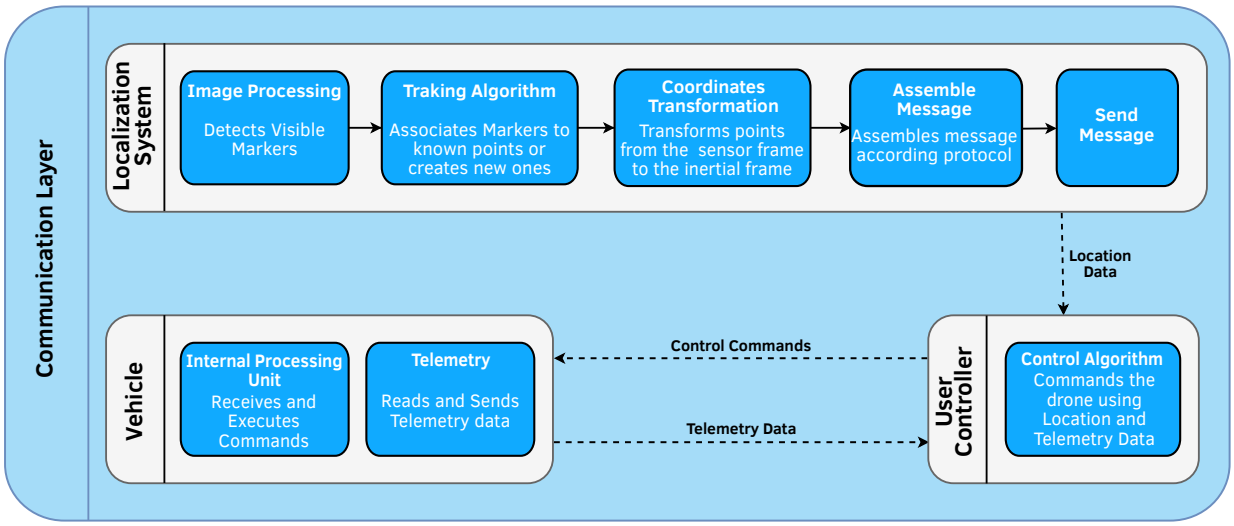
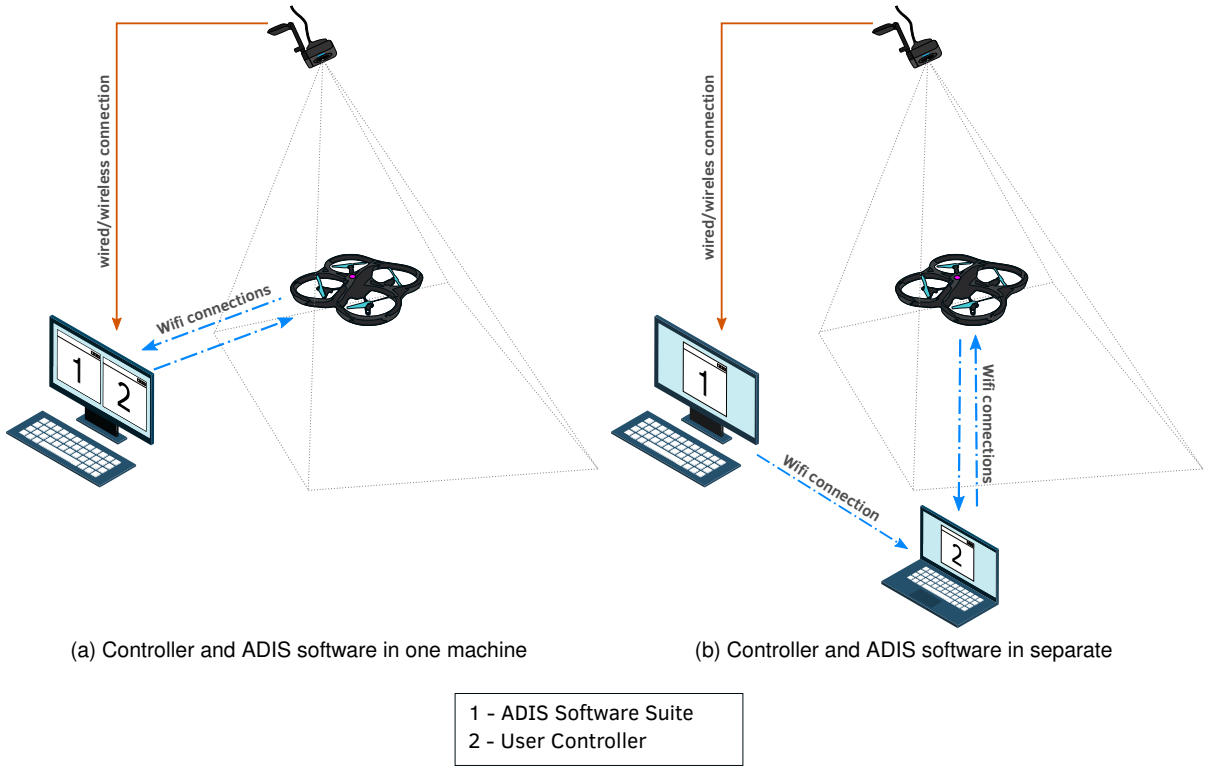


Figure 2: Basic Operation Diagram

The localization system is responsible for acquiring location data. This is accomplished using image processing and tracking algorithms to identify and track specialized markers in the inertial frame defined by the user. These markers identify relevant points to the experiment, like landmarks or vehicle position. Since the tracking algorithm ensures that any marker has the same identification number at any given time, it is possible to follow these relevant points during the experiment. Meanwhile, the vehicles are entirely controlled by the user. To connect every component to each other, there is a communication layer that handles all traded messages and defines every message format.

2.3 System Architectures

The ADIS system has a defined architecture that allows its components to work together. However, it also allows some flexibility in its architecture, providing the user with a choice on how and where his controller will be integrated. The most basic configuration is described in figure 3.



(a) Controller and ADIS software in one machine

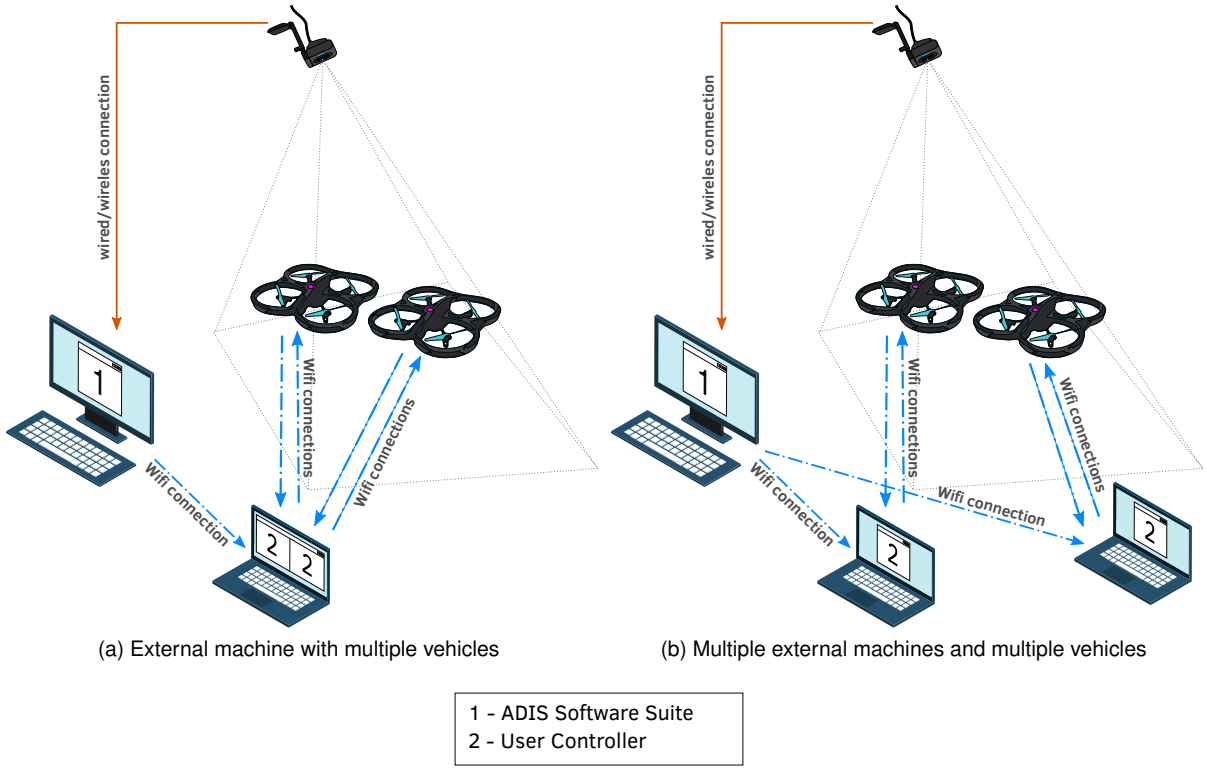
(b) Controller and ADIS software in separate

Figure 3: ADIS basic architectures

The architecture shown has two main components, a desktop, and a camera. This configuration and many others are thoroughly explained in section 3. These architectures diagrams do not represent the real network topology used, they represent how every component is connected to the others within the network. More about this matter in subsection 5.2.

In figure 3a, the localization system and the user custom controller are running on the same machine. The localization system sends the location data directly to the controller without any external connection. Finally, the controller interprets the received data and controls the drone accordingly. All control commands are sent wirelessly from the computer to the drone through a local Wi-Fi network. The drone is also sending to the same network its telemetry data to be read by the controller. This simple network architecture has the advantage that only two wireless connections per drone are required, making it the simplest configuration for a new user to work with. However, running the ADIS localization system and the user's controller on the same machine means that these two processes are competing for the same resources. Adding more and more vehicles or cameras to the system will increase greatly the resource requirements of the overall system, resources which are provided by only one computer. This limits greatly the system's scalability. In order to overcome this scalability problem, the architecture in figure 3b was developed.

In figure 3b, the ADIS localization software is still running in the same computer as before, but now the user controller is running in a separate machine. The localization system sends the location data through the Wi-Fi network to the second computer which controls the drone. The telemetry from the drone is sent back to the second computer. This kind of configuration adds some complexity but overcomes some scalability problems from the last architecture, mainly problems related to the growing number of vehicles. By adding a separate machine to control the vehicles, some processing power is unloaded from the computer that runs the ADIS localization software to the new computer. This adds resources to the overall system, allowing it to accommodate more vehicles and more cameras.



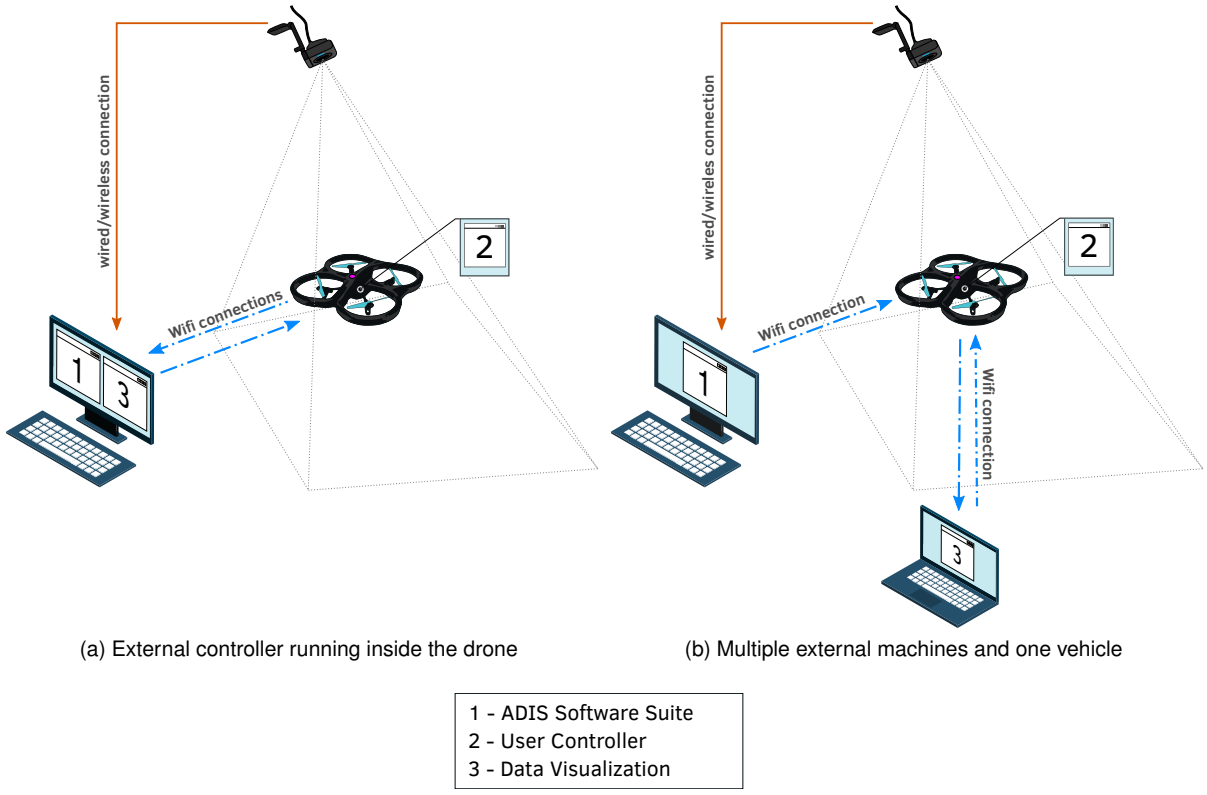
(a) External machine with multiple vehicles

(b) Multiple external machines and multiple vehicles

Figure 4: ADIS multi-vehicle architectures

As shown in figure 4, this architecture can be expanded to accommodate more computers running their own control algorithms to control even more drones. Figure 4a shows how one computer is able to control multiple vehicles, by running more than one controller at the same time. Each controller communicates to its corresponding drone and uses the location data sent by the localization system. It is important to note that these controllers must run within the same process (see section 5). There is also a possibility to run these multiple controllers alongside the localization software in the main computer, but it is not recommended due to the possible lack of resources, which could lead to the program crashes and controller unresponsiveness. Figure 4b, describes how even more computers can be integrated into the system. Each computer runs one control algorithm and steers his corresponding drone. The location data is sent to multiple connections around the network. This is done by setting the ADIS localization software to broadcast it's location data over the Wi-Fi network, allowing all of the connected computers to receive this data. This is the default configuration of the ADIS localization system.

Up until this point, all controllers ran in an external machine. This kind of algorithm controls the drone in a similar manner as a human controller would, by sending directional commands to it. These commands are then interpreted by a stability controller running inside an onboard flight controller. This controller guarantees drone stability when flying. The user's external controllers are not able to bypass this onboard control process, so, it is not possible to write external controllers that can, for example, address directly the thrust levels for each individual rotors. Fortunately, as described in section 4.1 it is possible to write and compile algorithms that run directly inside the drone's onboard processing unit. When using the onboard computer it is also possible to visualize the drone's attitude data, as well as send start and stop commands. This is done through an available data visualization interface. Figure 5 describes two different architectures using this technique.



(a) External controller running inside the drone

(b) Multiple external machines and one vehicle

Figure 5: ADIS onboard algorithm architectures

Using the main computer for hosting both the localization software and the data visualization interface, like in figure 5a, it is possible to observe the drone behavior and the location data in one machine. In this case, the drone handles the receiving and processing tasks of the location data as well as running the user custom control algorithm that directly controls its attitude and behavior. Once again, like in figure 3a, there are scalability issues, however, the resources required to run the figure 5a configuration are much lower, since the processing load is distributed between the computer and the drone.

Like in previous configurations it is possible to add external machines to alleviate the main computer. Just like figure 5b demonstrates, hosting the data visualization interface into another computer frees the main computer to deal only with localization-related tasks. Once again, the localization data is sent directly to the drone, and not to an external computer. This last approach improves upon the scalability

factor greatly, distributing the processing load between a wider set of devices than the other architectures presented until now.

The ADIS system provides a fair amount of flexibility to its user, however, it is still possible to join some of the configurations presented previously to get a more complex, yet, more capable system. One example of this is shown in figure 6.

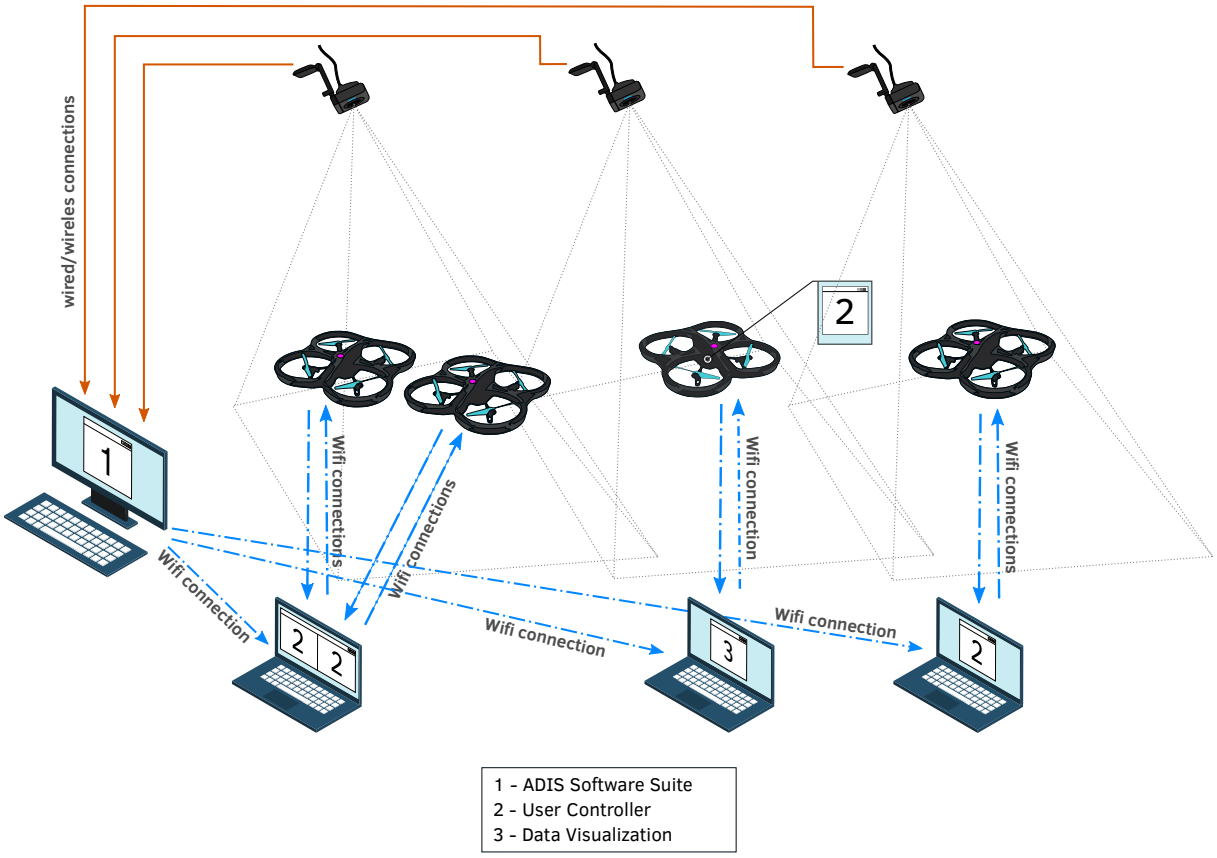


Figure 6: Extended Architecture

This extended configuration is composed of multiple external computers controlling the vehicles in various different ways. In this case, the architectures from figures 3b, 4a and 5b, are coupled together. This is targeted for more complex test scenarios and development environments, allowing themes like formation flight, dynamic obstacle avoidance, and cooperative vehicle control to be explored.

2.4 Components

In table 1 the full list of components used in the ADIS system is presented, with a small description.

Name	Description	Quantity
Windows 10 64-bit Desktop	The main server for the Localization system	1
Logitech C615	Camera used for the 2D tracking	2
Xbox 360 Kinect	Camera and depth sensor used for 3D tracking	1
ASRock Q1900M	Motherboard with integrated CPU for the Remote Camera Processing Unit	1
MAXPOWER FSFX400MP	Power supply for the Remote Camera Processing Unit	1
SSD 2.5P ADATA SP550 120GB	Storage for the Remote Camera Processing Unit	1
Bi-Sonic SP501512H	50x50X15 mm 12V Fan for the Remote Camera Processing Unit	2
TP-Link TL-WN722N	Wireless Wi-Fi card for the Remote Camera Processing Unit	1
Parrot AR Drone 2	Wi-Fi controlled commercial drone	3
8 GB Flash Drive	Additional storage for the drone	3
LaTrax SST	Remote Control car	5

Table 1: List of Components

3 Localization System

It is essential to any navigation algorithm to have access to location data relative to the vehicle it controls. This makes the localization system one of the core parts of the ADIS architecture. It is responsible for acquiring reliable location data and provide access to it whenever needed.

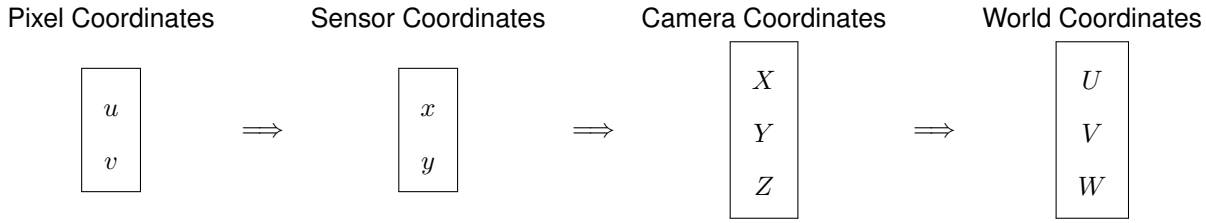
In order to be used for development and testing of navigation algorithms, some requirements were taken into account while creating the localization system. The system must have a big enough coverage area that allows an aerial or grounded vehicle to move freely. If the controlled vehicle is out of sight of the localization system, its location is unknown, therefore, the navigation algorithm is unaware of the vehicle location, unless it uses other sensors to estimate its position. Second, the localization system must follow and track objects through the entire experiment duration. This is essential because when more than one point is being detected, it is necessary to be able to distinguish which point corresponds to each vehicle or landmark reliably. When working with more than one vehicle like doing formation maneuvers, or landing on a moving target, the detected points must be associated correctly, otherwise, the whole experiment can fail, because the controller algorithm is suddenly trying to maneuver with wrong position inputs. Finally, the location system must have a sampling rate that is both fast enough and stable. The sample rate determines how fast a controller is going to be able to observe the vehicle behavior if the sampling rate is too low, the ADIS system is not going to be able to handle inherently fast-moving vehicles, like drones. The sample rate must also be stable because fluctuations on the sampling time can affect discrete controllers responses, that are projected with a specific sampling time.

The ADIS localization system characteristics are going to be explained next, however, it is important to note that depending on the experiment goal, the localization system can have slightly different architectures. This feature was not planned in the initial system design, which aimed to build a localization system only capable of tracking objects in two dimensions. But during the development work, arose the necessity for the ADIS system to be used by students for academic assignments. At that time the three-dimensional tracking feature was not fully developed, so the students based their work on the two-dimensional version. In order not to lose their work due to changes in the localization system, backward compatibility was ensured across the entire software suite, which led to multiple architectures being supported too. These multiple architectures also support multiple hardware configurations.

3.1 Theoretical Overview

In optical localization technologies, one of the main tasks is to associate pixel coordinates, measured from a digital sensor, to real-world coordinates from a visible scene. This association should be described by a well defined mathematical model, which allows a computerized system to interpret the real

world and understanding its surroundings.



The pinhole model is one of these models that handle the conversion between Sensor coordinates and Camera coordinates and comes from one of the oldest photography techniques, the pinhole camera. They do not have a lens and are built from an opaque box with a small side drilled hole. Light enters the dark box through this hole and projects an inverted image into the opposite wall.

3.1.1 Pinhole Model

The pinhole model describes how the three-dimensional coordinates of a point in space, are mathematically related to its projection onto the image plane of an ideal camera. The image plane corresponds to the plane in the world from which the scene is viewed, and in the case of a digital camera, it's the plane of the photosensitive sensor.

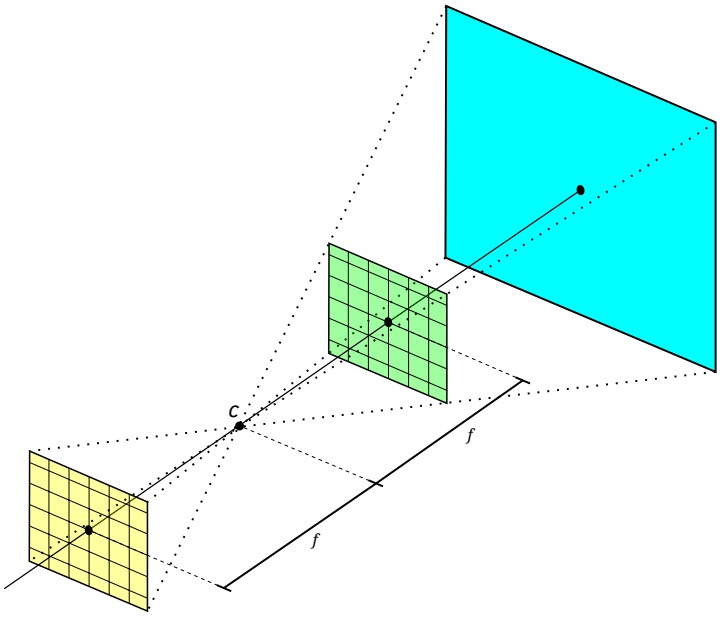


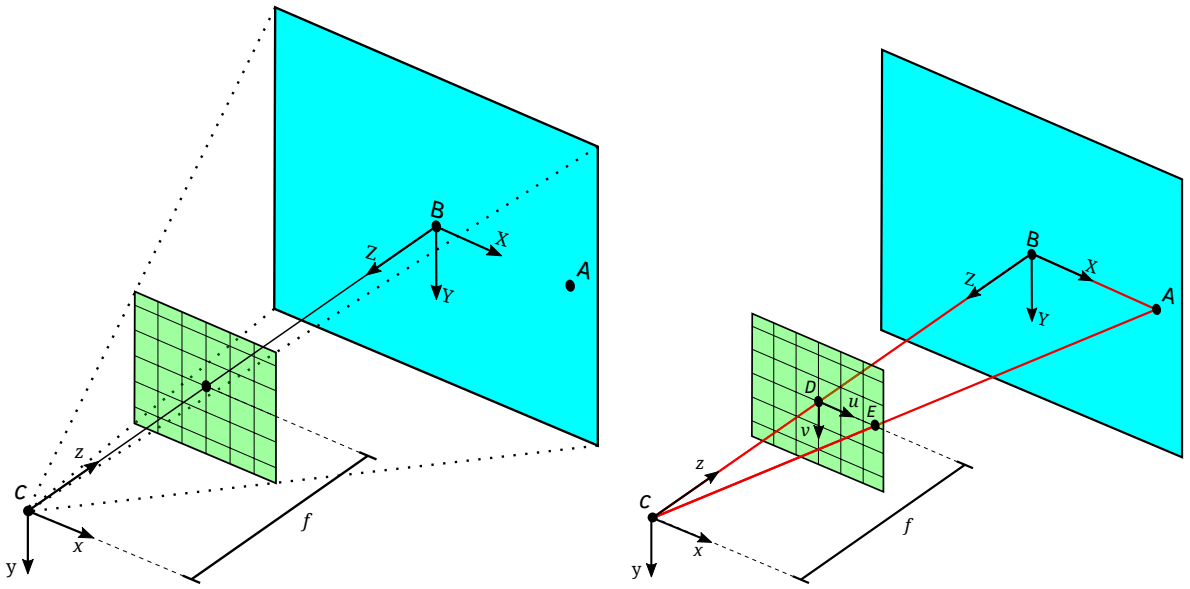
Figure 7: Pinhole camera geometry

Figure 8 shows the pinhole camera geometry. Point C marks the hole of the camera: here the light enters the opaque box, the blue surface corresponds to the captured image, and the yellow plane to the wall where the captured image is projected in reverse. This image can be permanently saved, using specialized photosensitive materials. The distance between the side hole and the opposite wall, represented by f, is called focal length.

Applying this model to the modern counterpart of the pinhole camera, the digital camera, can be done

with some reservations. Pinhole cameras do not have lenses like the digital cameras do, and these lenses introduce some distortions that are not taken into to account by the pinhole model. Because lenses are not perfect, some geometric distortions caused by deformations in the lenses structure are introduced. This means that this model validity depends on the lens quality, and it's reduced the further away from the center of the lens the point of interest is. The pinhole model also does not predict errors introduced by blurring effects of unfocused objects.

Nonetheless, the pinhole model can be used to obtain useful mathematical relations. Looking again at figure 7, it is possible to simplify this model assuming that the captured image is not inverted. This simplification is done automatically in most digital cameras. The plane that provides a noninverted image is the plane that is at the same distance from the point C as the yellow plane but in the exterior of the camera box. This plane is represented by the green surface.



(a) Camera Sensor plane (green) and Ground plane (blue)

(b) Defining a triangle $\triangle ABC$

Figure 8: Pinhole Geometry

In figure 8a the yellow plane was removed and a point of interest A added in its place, along with two three dimensional orthogonal coordinate axes. One with its origin at point C, and pointing to the viewing direction of the camera. Line $\overline{CB}$ is called the optical axis. The other axis has its origin at point B, where the optical axis intercepts the arbitrary blue plane. Where the optical axis intercepts the green plane, point D, was also added a two-dimensional orthogonal coordinate axis. Using points A, B, and C it is possible to define the triangle $\triangle ABC$, contoured red in figure 8b. Figure 9 shows a top perspective of this triangle. From the resulting triangle, two similar triangles can be found,

$$\triangle ABC \sim \triangle CDE$$

which leads to

$$\frac{\overline{AB}}{\overline{CB}} = \frac{\overline{ED}}{\overline{DC}} \Leftrightarrow \overline{AB} = \overline{CB} \times \frac{\overline{ED}}{\overline{DC}}$$

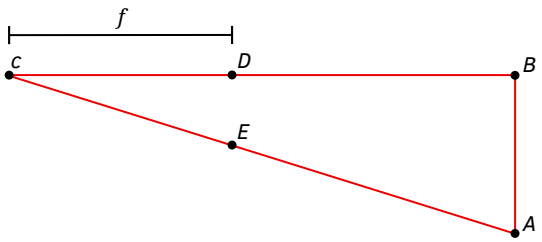


Figure 9: $\triangle ABC$

using the reference orthogonal axes and the focal length

$$X = \frac{Z \times u}{f} \qquad Y = \frac{Z \times v}{f} \tag{1}$$

To represent this as a matrix transformation, the concept of "homogeneous coordinates" must be introduced.

Homogeneous Coordinates This concept allows representing a two-dimensional point as three-dimensional point by using a "fictitious" coordinate. By convention, from a point in homogeneous coordinates (x', y', z') is always possible to recover (x, y) using:

$$x = \frac{x'}{z'} \qquad y = \frac{y'}{z'} \tag{2}$$

These coordinates are called homogeneous because the overall scaling of the coordinates is not important. In other words:

$$(x', y', z') = (k.x', k.y', k.z'), \qquad \forall k \neq 0$$

because

$$x = \frac{x'}{z'} = \frac{kx'}{kz'} \qquad\qquad y = \frac{y'}{z'} = \frac{ky'}{kz'}$$

This means that a representation of a Cartesian point (x, y) is not unique but independent of scaling. So, the pinhole model as a matrix equation using homogeneous coordinates is given by:

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \tag{3}$$

3.1.2 World to Camera Transformation

Figure 10 has a representation of the Camera frame located at point C, the Inertial frame with its origin in point B, and point A. A_B and A_C represent the same point but seen from different frames.

The transformation from the inertial frame to the camera frame is given by a translation, T, of the

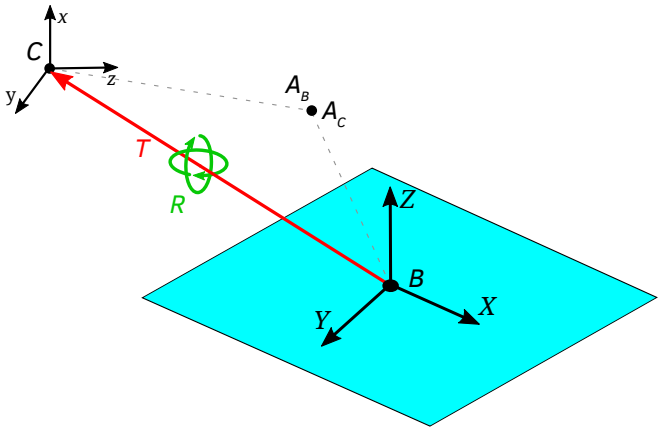


Figure 10: Camera extrinsic geometry

origin of the inertial frame, B, accompanied by a rotation R.

$$A_c = RA_B + T$$

The parameters T and R are called extrinsic parameters. Using homogeneous coordinates, this transformation is given by

$$\begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} = \left[\begin{array}{ccc|c} & R & & T \\ \hline 0 & 0 & 0 & 1 \end{array}\right] \begin{bmatrix} U \\ V \\ W \\ 1 \end{bmatrix}$$

which leads to

$$\begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} U \\ V \\ W \\ 1 \end{bmatrix} \tag{4}$$

3.1.3 Sensor Plane to Pixel Coordinates Transformation

This transformation describes the coordinate transform between an image projected on to the photosensitive sensor, and the digital pixel array captured by it. This transformation is needed because the sensor captures light with a finite set of photosensitive cells. These cells return a discretized projected image in the shape of an array of colored pixels. Each one of the pixel arrays is identified with a pair of indexes that do not correspond to the Cartesian coordinates measured in the sensor frame. As seen in figure 11, it's necessary to correct the origins offsets.

$$u = x + O_x \qquad v = y + O_y$$

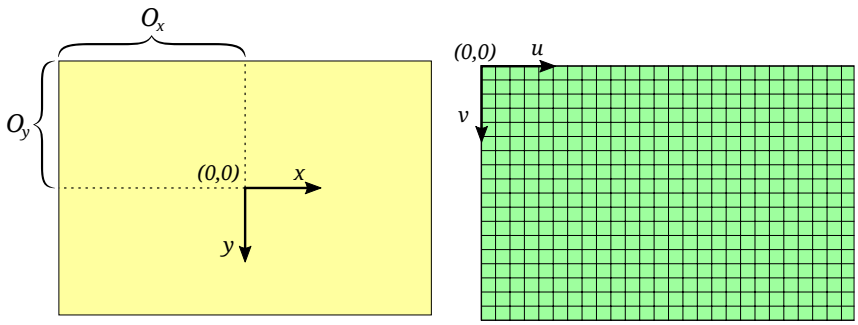


Figure 11: Sensor to Pixels

It is also necessary to convert from the metric units of the sensor plane to pixel units from the pixel array. For this, a scaling is used factor that corresponds to the size of each cell of the sensor.

$$u = \frac{x}{s_x} + O_x \qquad v = \frac{y}{s_y} + O_y$$

Now using homogeneous coordinates in a matrix representation:

$$\begin{bmatrix} u' \\ v' \\ w' \end{bmatrix} = \begin{bmatrix} \frac{1}{s_x} & 0 & O_x \\ 0 & \frac{1}{s_y} & O_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} \tag{5}$$

3.1.4 Planar Homography

Finally, using the equations 3, 4 and 5 :

$$\begin{bmatrix} u' \\ v' \\ w' \end{bmatrix} = \begin{bmatrix} \frac{1}{s_x} & 0 & O_x \\ 0 & \frac{1}{s_y} & O_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} U \\ V \\ W \\ 1 \end{bmatrix} \tag{6}$$

Considering point A in an inertial frame, as shown in figure 12. The coordinates of the point A are

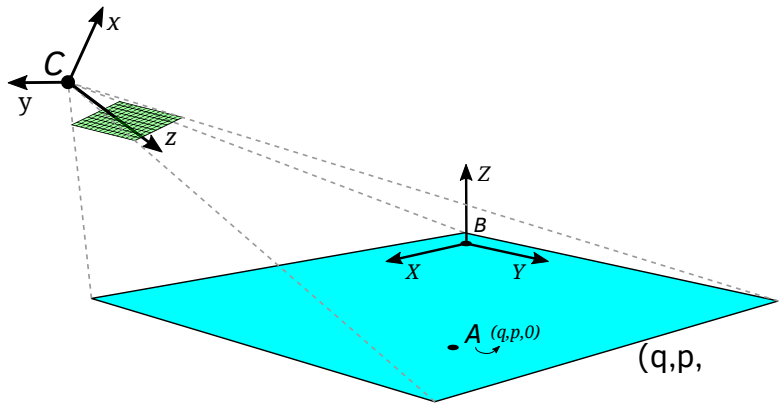


Figure 12

$(q, p, 0)$. Converting them to homogeneous coordinates yields $(q, p, 0, 1)$. Using these coordinates in the equation 6, allows the following simplifications.

$$\begin{bmatrix} u' \\ v' \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{1}{s_x} & 0 & O_x \\ 0 & \frac{1}{s_y} & O_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p \\ q \\ 0 \\ 1 \end{bmatrix} \Leftrightarrow$$

$$\begin{bmatrix} u' \\ v' \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{1}{s_x} & 0 & O_x \\ 0 & \frac{1}{s_y} & O_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} f & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & t_x \\ r_{21} & r_{22} & t_y \\ r_{31} & r_{32} & t_z \end{bmatrix} \begin{bmatrix} p \\ q \\ 1 \end{bmatrix} \Leftrightarrow$$

$$\begin{bmatrix} u' \\ v' \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{fr_{11}}{s_x} + r_{31}O_x & \frac{fr_{12}}{s_x} + r_{32}O_x & \frac{t_x}{s_x} + t_zO_x \\ \frac{fr_{21}}{s_y} + r_{31}O_y & \frac{fr_{22}}{s_y} + r_{32}O_y & \frac{t_y}{s_y} + t_zO_y \\ r_{31} & r_{32} & t_z \end{bmatrix} \begin{bmatrix} p \\ q \\ 1 \end{bmatrix} \Leftrightarrow$$

$$\begin{bmatrix} u' \\ v' \\ 1 \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \begin{bmatrix} p \\ q \\ 1 \end{bmatrix}$$

These simplifications convert a three-dimensional to two-dimensional projection into a two-dimensional to two-dimensional projection and originate matrix H, called the homography matrix. This matrix converts points from the inertial plane to the camera sensor plane in pixels. The homography matrix, as presented by Hartley and Zisserman [1], is an invertible mapping from points in $\mathbb{P}^2$ (that is homogeneous 3- vectors) to points in $\mathbb{P}^2$ that maps lines to lines. In other words, is an invertible mapping h from $\mathbb{P}^2$ to itself such that three points x_1 , x_2 and x_3 lie on the same line if and only if $h(x_1)$, $h(x_2)$ and $h(x_3)$ do. The fact that H has an inverse is useful because it is possible to map points from the inertial plane to the pixel array from the sensor plane but also from the pixel array to the inertial plane, like so:

$$\begin{bmatrix} p \\ q \\ 1 \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix}^{-1} \begin{bmatrix} u' \\ v' \\ 1 \end{bmatrix} \tag{7}$$

With this mapping is possible to use a correctly calibrated camera to measure points from a plane in an inertial frame from the real world.

3.1.5 Projection of a point onto a plane

The projection of a point onto a plane, as in figure 13, can be found by first considering the plane defined by a point $P_0 = (x_0, y_0, z_0)$ and a normal vector $\vec{n} = (a, b, c)$. Then the strategy is to find the R, so that the dot product between vectors $\overrightarrow{RP}$ and $\overrightarrow{RQ}$ is equal to zero, in other words, form a right angle between

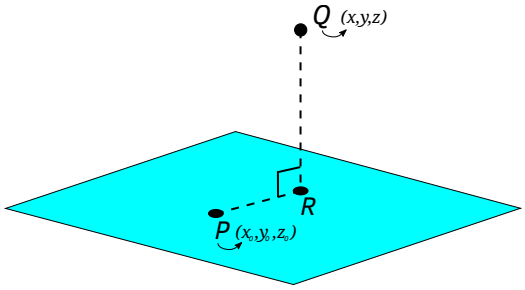


Figure 13: Geometry from a projection of a point onto to a plane

them. So for R, to become normal to the plane:

$$R = (x + ka\,,\, y + kb\,,\, z + kc)$$

making

$$\overrightarrow{RP} = [-x + ka + x_0\,,\, -y + kb + y_0\,,\, -z + kc + z_0]$$
$$\overrightarrow{RQ} = [-x + ka + x\,,\, -y + kb + y\,,\, -z + kc + z] = [ka\,,\, kb\,,\, kc]$$

applying the dot product

$$\begin{aligned} 0 &= \overrightarrow{RP}.\overrightarrow{RQ} \\ &= [-x + ka + x_0\,,\, -y + kb + y_0\,,\, -z + kc + z_0]\,.\,[ka\,,\, kb\,,\, kc] \\ &= k(a^2 + b^2 + c^2) + x_0a + y_0b + z_0c - xa - yb - zc \end{aligned}$$

isolating the missing variable k,

$$k = \frac{-xa - yb - zc + x_0a + y_0b + z_0c}{a^2 + b^2 + c^2} \tag{8}$$

which completely defines the projection point R. This concept is used in section 3.3.1.

3.1.6 Random Sample Consensus (RANSAC)

The RANSAC algorithm is an estimation algorithm introduced by Fischler and Bolles[18], as a technique to estimate parameters of a certain model using a data set with a large number of outliers, and is used in section 3.3.1. An outlier is a data point that does not fit the “true” model specified by the ”true” set of estimated parameters, because it is not within some error threshold which defines the maximum deviation associated to the effect of noise. It is an iterative algorithm that will search randomly for the best solution that fits the given data set.

Algorithm 1 Random Sample Consensus algorithm

Require: $data\_set$, $maximum\_iterations$, $decision$, $threshold$

while $iteration < maximum\_iterations$ **do**
  Select randomly n_values from $data\_set$
  Estimate the $model$ using the n_values

  for every $point$ from $data\_set$ not in n_values **do**
    if $point$ fits $model$ with an $error$ < $threshold$ **then**
      Add $point$ to n_values
    end if
  end for

  if the number of n_values is > than $decision$ **then**
    Measure how well $model$ fits by calculating a $model\_error$
    if $model\_error < best\_model\_error$ **then**
      $best\_model = model$
      $best\_n\_values = n\_values$
      $best\_model\_error = model\_error$
    end if
  end if

  Increment $iteration$
end while

3.1.7 Singular Value Decomposition (SVD)

The singular value decomposition (SVD) is a factorization of a real or complex matrix, with many applications in technological areas like in signal processing and statistics. Formally the singular value decomposition theorem states that:

Theorem. *Any matrix $A \in \mathbb{R}^{m\times n}$, can be decomposed into two orthogonal matrices $U \in \mathbb{R}^{m\times m}$, $V \in \mathbb{R}^{n\times n}$ and a diagonal matrix $\sum \in \mathbb{R}^{m\times n}$, i.e.,*

$$\Sigma = \begin{bmatrix} \sigma_1 & & & 0 & \cdots & 0 \\ & \ddots & & & & \\ & & \sigma_r & & & \\ & & & 0 & & \\ & & & & \ddots & \\ & & & 0 & \cdots & 0 \end{bmatrix} \quad \textit{for} \quad m \leq n$$

with diagonal entries

$$\sigma_1 \geq \ldots \geq \sigma_r > \sigma_{r+1} = \ldots = \sigma_{min}\{m, n\} = 0$$

such that

$$A = U\Sigma V^T$$

The diagonal entries σ_i, of Σ are called singular values of A. The columns of V are called right singular vectors and the columns of U are called left singular vectors.

This technique has many applications, one of them is the ability to solve plane fitting problems as described in “Solving Problems in Scientific Computing using Matlab and Maple”, by W. Gander and J. Hrebicek[53].

3.2 2D Localization

As referred before, initially the localization system was only capable to track two-dimensional coordinates. Although a three-dimensional version was explored later, this first version is still relevant, because it maintains compatibility with already developed work and it offers some advantages due to differences in hardware.

3.2.1 Basic Operation

The 2D implementation of the localization system uses image processing to detect and track specialized markers, placed on points of interest defined by the user. To detect these markers, digital cameras are used, in tandem with specialized image processing algorithms.

The basic operation concept of the localization system is fairly simple. As seen before in figure 2, the localization system workflow can be divided into five phases. First, it detects specialized markers using digital cameras and image processing techniques. Then associates each detected marker with others detected in previous frames. If no match is found, assumes that a new marker appeared. After that, transforms the pixel coordinates of each marker to inertial frame coordinates using equation 7. Finally assembles and sends the message to the navigation controller.

With this procedure is possible to detect and locate vehicles only in one plane. Equation 7, is only useful to transform points from one plane to another and does not take into account that the vehicle might not be moving on the plane. This denies the possibility to integrate any kind of flying vehicle to the system because it is assumed that every detected marker is placed on the inertial plane. If the airborne vehicle is able to measure its height, it’s possible to correct the equation 7 and calculate the correct location coordinates of the vehicle.

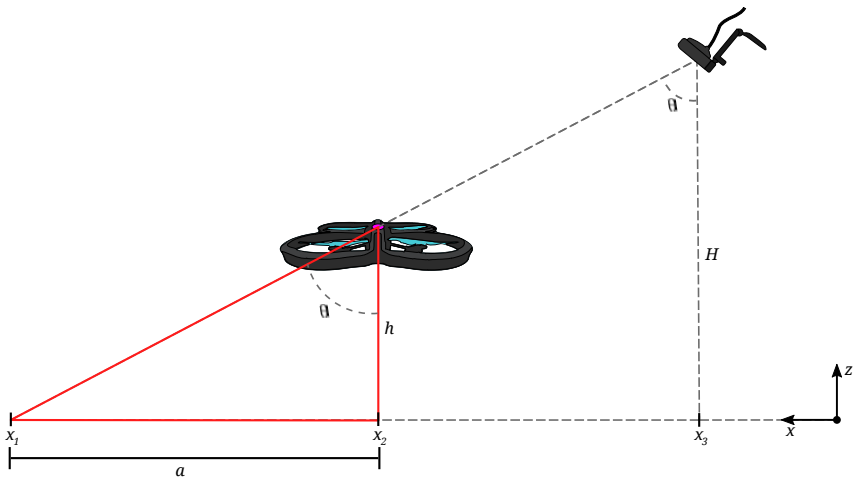


Figure 14

Because the drone as height h, the localization system perceives its location as x_1 instead of x_2.

$$\begin{cases} x_1 - x_2 = a \\ tan\,\theta = \frac{a}{h} \\ tan\,\theta = \frac{x_1 - x_3}{H} \end{cases} \Leftrightarrow \begin{cases} x_2 = x_1 - a \\ a = \frac{(x_1 - x_3)h}{H} \end{cases} \Leftrightarrow x_2 = x_1 - \frac{(x_1 - x_3)h}{H}$$

the same is valid for the y axis, leading to:

$$x_2 = x_1 - \frac{(x_1 - x_3)h}{H} \qquad y_2 = y_1 - \frac{(y_1 - y_3)h}{H} \tag{9}$$

With equation 9, using the camera height and position, along with the drone's height, is possible to correct the coordinates given by the localization system. The problem with this solution is that it requires the localization system to establish a connection with each drone to access its height. This is not a good solution because the localization system would need to manage which connection corresponds to which marker, adding more processing overhead to the system and increasing the overall complexity with more connections. To avoid these, this workaround needs to be implemented by the user in the controller's side. Because the user already has a connection established to access the onboard sensors data, the navigation algorithm would use the drone's height directly and correct the location coordinates. To do this the user still needs the camera height and location in the inertial frame. Implementing an automated way to calculate these values is possible, but since a complete 3D localization system was developed, (described in section 3.3) if this method is used, the camera height and location in the inertial frame would have to be measured manually.

3.2.2 Architecture

For the 2D localization system, two possible architectures were implemented. These architectures were developed with two different goals in mind and represent multiple stages of development of the ADIS system. The first developed architecture is presented in figure 15.

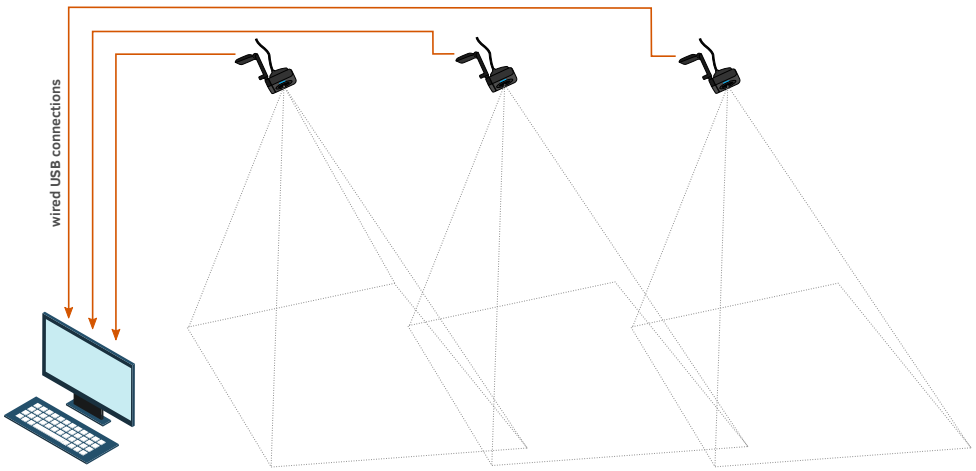


Figure 15: Wired Architecture

Each camera is connected directly to the computer via a USB connection. This requires the use of USB extensions with embedded signal repeaters, to avoid signal losses due to their length since each camera is fixed to the ceiling of the laboratory. Every camera sends the captured frames directly to a computer to be correctly processed with no additional hardware in between. This allowed to test and develop software tools and image processing algorithms without complicated setups or development environments. Unfortunately, this architecture also has setbacks that led to other more complex solutions. The biggest problem is that all of the necessary processing work is unloaded in one single machine. This greatly limits the image processing algorithm implemented, because it needs to be fast enough to be executed once for each connected camera during the time that the quickest camera takes to capture a frame. If the desired frame rate is 30 frames per second then all of the processing work for all cameras needs to execute in less than 33 milliseconds. If the image processing algorithm is not fast enough, the sampling time of the localization system may be too slow to control rapid moving robots. This limits the amount of area that can be covered because the more cameras are added, the more information the main computer needs to process per second. This unavoidably results in scalability problems. A solution to this problem is, for example, lower the camera's resolution. But this will make it harder for the image processing algorithms to detect markers because of their necessarily small size. This results in the need to implement more complex algorithms that take more time and processing power to execute, which doesn't solve the problem.

To find a solution, it was necessary to implement a way to increase the available processing power. Improving computer components was not a viable option due to the high costs of high-end computer parts. So a different architecture, that unloads the processing work from the main computer, elsewhere was necessary, effectively increasing the overall processing power available. With this in mind, the following architecture was developed.

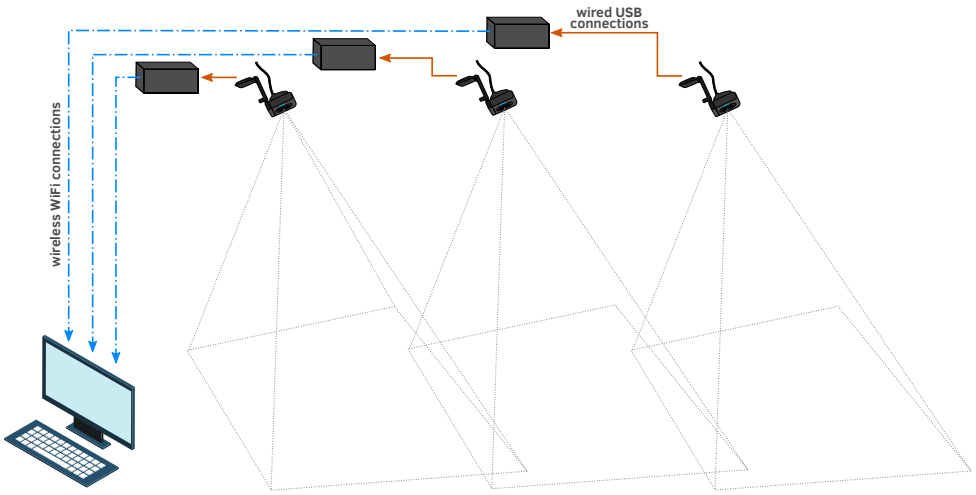


Figure 16: Distributed architecture

Instead of connecting every camera to the main computer, now the camera is part of a module composed of the camera itself and a dedicated processing unit. This processing unit gathers the image information captured by the camera, processes it, and sends it to the main computer. The connection between the main computer and the camera module is wireless through a Wi-Fi connection, removing

the need to use the previously required USB extensions. Despite this, the dedicated processing units need to be powered externally through mains power. This processing load is effectively distributed between each camera processing unit and the main computer. First, the camera processing unit acquires the frame captured by the camera and detects every visible marker. Then, it sends this information back to the main computer to track markers through time, which sends the results to their final destination, like a navigation algorithm running in another computer. This method alleviates the processing costs of adding extra cameras to the main computer, in exchange for added complexity to the localization system.

The processing unit is also responsible for handling all communications to the main computer, not only regarding marker detection but also regarding system setup and organization. This means that they need to communicate with the main computer to organize the network in a sensible manner. The main computer is still the system's hub, from which the final location data is obtained. When a user uses this type of system topology, gets access to the location data the same way as using the previous architecture, but the main computer also controls the remote cameras using a custom made protocol, described in section 5.

3.2.3 Marker Detection Process

The cameras used are common web-cameras available in most retail stores. The way that the marker detection process works went through an evolution process that started by exploring if the cameras worked only with infrared light. Working in the infrared spectrum provides multiple advantages. Because infrared cameras only capture the infrared light intensity, they return monochrome (black and white) pictures, which are easier to process. Also, because in a normal indoor situation infrared light is not as abundant as visible light, working with infrared wavelengths avoids many sources of noise, reducing the amount image processing necessary. However, this method requires specialized cameras that only detect infrared light, that is not easily available. To avoid using expensive equipment, which would go against this thesis objective of building a low-cost system, regular web-cams were modified to only detect infrared light.

Regular digital web-cameras can be converted to infrared only cameras by removing the infrared filter from the lens assembly and adding a crossed polarizer filter to block visible light. To do this, multiple HP HD 2300 webcams, figure 17, were modified as previously described.

Unfortunately, this method proved not to be ideal, because the web-cameras were not sensitive enough to capture reliably infrared light reflected by passive markers, or emitted by active markers. The alternative was to use regular web-cameras and color segmentation algorithms instead of infrared cameras.

The specific web-camera model used was the Logitech C615. This camera was chosen due to being equipped with glass lenses. Good quality lenses are important because they are less prone to having deformities due to bad manufacturing which, as said in section 3.1.1, the pinhole-model does not take into account. This web-camera is capable of capturing images with a maximum rated resolution[11] of 1080p (1920x1080 pixels), although with a low frame rate around 10 frames per second. In order to keep a stable frame rate of 30 frames per second, the maximum sampling rate for this model, the camera

Figure 17: HP HD 2300

Figure 18: Logitech C615

must capture images at 720p (1280x720 pixels) resolution. These cameras connect to a computer via USB 2.0 interface. They also come equipped with auto-focus and auto-exposure features.

For the image processing algorithm that detects the markers, image color segmentation was used. All algorithms were implemented either using Matlab or Python plus OpenCV (more information in sections 3.4 and 6). Color segmentation is a process that separates the individual color channels of an image into individual monochrome pixel matrices. Each element of each pixel matrix has a corresponding value. This corresponds to the color intensity of the channel color that the pixel had in the original image. For example, digital cameras can return images with colors coded using an 8-bit RGB color space. In this color space, every color is the sum of three color values, red, green and blue. Separating the color channels from the original image results in three matrices, one for each channel. Any element of these matrices is a value with a depth of $2^{8\,bit} = 256$, meaning that is between 0 and 255. To actually identify the markers thresholding was used. If a marker is a red circle, after separating the three RGB color channels, the blue and green channels will not show the marker, but the red channel will have a bright spot where the marker is located. So to identify the marker, every pixel in the red channel is compared against a limit value, a threshold value, and converted to a boolean. If it is higher than the threshold limit is set to 1, otherwise, it is set to 0. This creates a binary matrix, where, if the threshold value is set correctly the marker is visible as a set of “1’s” grouped together.

Unfortunately, this approach has problems because every red colored object inside the frame will be visible and will originate false positives. Furthermore, every bright white object like light reflections on the floor will originate false positives as well. Because of the white color on the RGB color space being represented by the triplet $(255, 255, 255)$ it is present on every channel. This color segmentation method is also vulnerable to changes in lighting across the frame. For example, if the marker goes to a less illuminated area inside the frame, its color would appear to be darker than before. This means that the values for the marker area, in the red channel will be lower. If these values drop below the threshold level the marker is invisible to the segmentation algorithm.

To deal with other red objects in the frame, a basic background subtraction technique was added. Background subtraction methods remove the background of captured frames, to extract only the foreground. There are numerous background subtraction techniques, ranging from temporal median filters to more complex techniques like Gaussian Mixtures and others. For this system, it was used a very simple technique that only works because the cameras are stationary during the length of the experiment,

and the laboratory is illuminated using constant artificial light. Making the background approximately constant throughout the day. To remove the background from the original image, the mean background image is generated before the experiment starts and is subtracted to every captured frame. Meaning that every vehicle needs to be removed from the camera's view before the mean background image for each camera is created.

To avoid false positive detections where white colored objects appear in the frame is possible to threshold the green and blue color channels as well. For example, if after applying the threshold to the red channel, any pixel marked with a "1" as a high value on the green channel, lets say "210", this color can be a shade of yellow, or if it has a high value in the blue channel, it can be a shade of grey. Unfortunately, this process is computational heavy, since for every pixel it is necessary to check multiple color channels. To avoid this, another approach was used. It consists of subtracting to the red channel of the current frame a black and white image of the current frame. This eliminates the white spots in the frame because the black and white frame is generated by calculating a mean between every channel of the RGB image for each frame. For example, a red pixel RGB representation is given by the triplet $(220, 10, 10)$, has a mean value of 80. Subtracting this value to the red channel gives $220 - 80 = 140$. On the other hand, a white pixel represented by the triplet $(255, 255, 255)$ has a mean value of 255, subtracting that to the red channel value $255 - 255 = 0$, which eliminates the white spots on the image. The final version of the image processing algorithm 2, implemented in the architecture from figure 15 is:

Algorithm 2 Marker Detection Algorithm with RGB color space

Require: $Background$ for each camera

for each camera **do**
  Generate $Black\_White\_Red\_Background$ from $Background$
  Extract $Red\_Background$ from $Background$
  $Tracking\_Red\_Backgound \leftarrow Black\_White\_Red\_Background - Red\_Background$
end for

while running **do**
  for each camera **do**
    Get $Frame$ from camera
    Generate $Black\_White\_Frame$ from $Frame$
    Extract Red_Frame from $Frame$
    $Thresh\_Frame \leftarrow (\, Red\_Frame - Black\_White\_Frame\,) - Tracking\_Red\_Background$
    $Binary\_Frame \leftarrow threshold(\, Thresh\_Frame\,)$
    $Centroids \leftarrow blob\_analisys(\, Binary\_Frame\,)$
  end for
end while

This algorithm was also sped up using Regions Of Interest (ROI). Every cycle that the algorithm runs, the full set of pixels that compose the captured image is processed, even though, the markers occupy small areas. Using a tracking algorithm, described later in section 3.7, it is possible to predict an estimation of the current position of the marker. This creates a binary mask, that shrinks the area of pixels to process into smaller areas around each estimated position, while still including the desired marker. This technique greatly decreases the number of pixels processed in each iteration, ultimately

increasing execution speed. Obviously, it's necessary, from time to time, to process the full image to detect new markers that may have appeared in the camera's field of view. This process occurs once every 30 frames, meaning that once per second new markers can be detected.

The last mentioned problem requires a more drastic change to the marker detection algorithm. When using an RGB color space, color is defined by adding three basic colors together in different quantities. The more of each color is added the brighter the final color becomes. This means that there is no way to define how bright a color is, without the need to look at all three color channels. To identify a color independently of how bright or how dark that color is, a different color space representation called HSV is used. HSV stands for Hue, Saturation, and Value. Hue is the channel that stores the actual color information. Typically it ranges from $0°$ to $360°$. Saturation is the value that describes how intense the color is. And the Value channel describes how bright the color is. This color space expresses color information independently of brightness and saturation, meaning that a color is the same independently of lighting conditions. This a more precise color detection algorithm. The main disadvantage of using HSV color representation comes from the fact that digital cameras output images using RGB color representation, that needs to be converted to HSV at runtime. This requires a per pixel conversion using[13], that cannot run fast enough in a single machine with multiple cameras while maintaining an acceptable sampling rate for the localization system. For this reason, the HSV approach is only implemented when using a distributed architecture described in figure 16 with remote cameras.

Using an HSV color representation, where the color information is defined by one single channel, creates more robust and simple color detection algorithms. After thresholding, the hue, saturation, and value channels a binary image where the marker is visible as a set of "1's" grouped together appears. This algorithm does not require techniques like background subtraction, or any correction for bright white spots, because it effectively detects only the desired color in the captured image.

Algorithm 3 Marker Detection Algorithm HSV Color Space

while running **do**
  for each camera **do**
    Get $Frame$ from camera
    Convert HSV_Frame from $Frame$
    $Binary\_Frame \leftarrow threshold(\, HSV\_Frame\,)$
    $Centroids \leftarrow blob\_analisys(\, Binary\_Frame\,)$
  end for
end while

The $blob\_analysis$ function used in the algorithms 2 and 3, returns the coordinates of the centroids in pixels, from every marker detected. This function was implemented using the Matlab function $blod\_analysis$ [15] from the "Image Processing Toolbox". This function is coded in a pcode file, which is an encrypted file format to protect the source code inside. However, for the purpose of this thesis, an alternative algorithm is presented.

The algorithm must execute two different tasks. Identify and label every blob of "1's" from the binary image, and calculate the centroid for each labeled blob. Analyzing the available documentation for the "Image Processing Toolbox" [14], the labeling task can be achieved using one of the algorithms for

connected component labeling from *Computer and Robot Vision*[16] or *Algorithms in C*[17]. Then it calculates the centroid of each labeled blob using:

$$x = \frac{1}{n}\sum_{k=1}^{n} x_k \qquad y = \frac{1}{n}\sum_{k=1}^{n} y_k$$

3.3 3D Localization

For grounded vehicles, it is not as important to be able to measure height as it is for aerial vehicles. When locating a flying vehicle in three-dimensional space, it is paramount to be able to measure height values. To measure height directly, the most common method consists in equipping the flying vehicle with an altimeter. However, when the goal is to measure height with an external system, options ranging from stereo vision techniques with multiple cameras to the use of dedicated hardware depth sensors exist. For the ADIS system, a depth from structured light sensor, the Kinect 360, was chosen, since it requires less complex algorithms and hardware already available in the laboratory.

3.3.1 Basic Operation

The depth sensor used is a Kinect 360. This sensor shipped with the Xbox 360 gaming console and comes equipped with two cameras. One that captures normal RGB video, and another infrared camera that captures depth information. The equipped depth sensor, captures the distance from the camera plane, to the object, represented by the letter d in figure 19, and not the distance from the camera to the object. The joint operation of these two cameras allows expanding some theoretical concepts applied in the two-dimensional version of the system, to create a three-dimensional version.

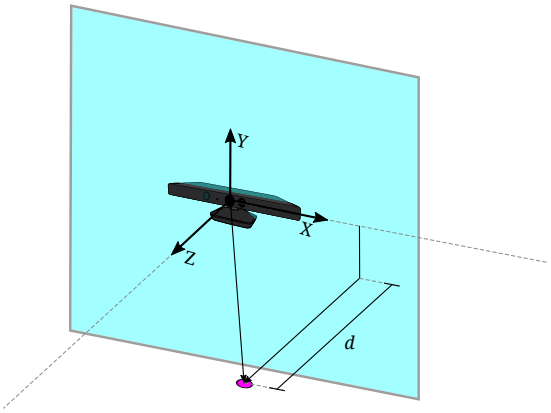


Figure 19: Kinect depth measurements

The basic operation of the 3D localization system is still divided into five different phases. The marker detection phase is the same as the one described in algorithm 3, using the HSV color space to detect specialized markers. Tracking phase, message assembly and sending process continue to be the same as in the two-dimensional versions. The changes are in the conversion from pixel coordinates to real-world coordinates because extra depth information needs to be taken into account. The planar homography model given by equation 7, is not enough to compute the height of an airborne vehicle,

because it does not take into account the existence of depth.

Like any other sensor, the depth measurements taken with the Kinect have noise. When pointing it to the floor, the captured depth information has errors that deform, ever so slightly, the planar surface. To correct this, it's assumed that the floor is a perfect plane,and that it can be described by the equation 10.

$$Ax + By + Cz + D = 0 \tag{10}$$

Thus, before any other calculation, the floor planar equation must be estimated. To do this, the Random Sample Consensus algorithm also known as RANSAC is used as described in section 3.1.6. The RANSAC algorithm outputs a good estimation for the parameters A, B, C, and D, that serves as a reference for any other calculations made relative to the ground plane. The floor planar equation is determined using a reference frame with the origin located in the center of the Kinect and not in the inertial frame.

With a good estimation of the floor position, it is possible to add depth information to the location coordinates according to the following:

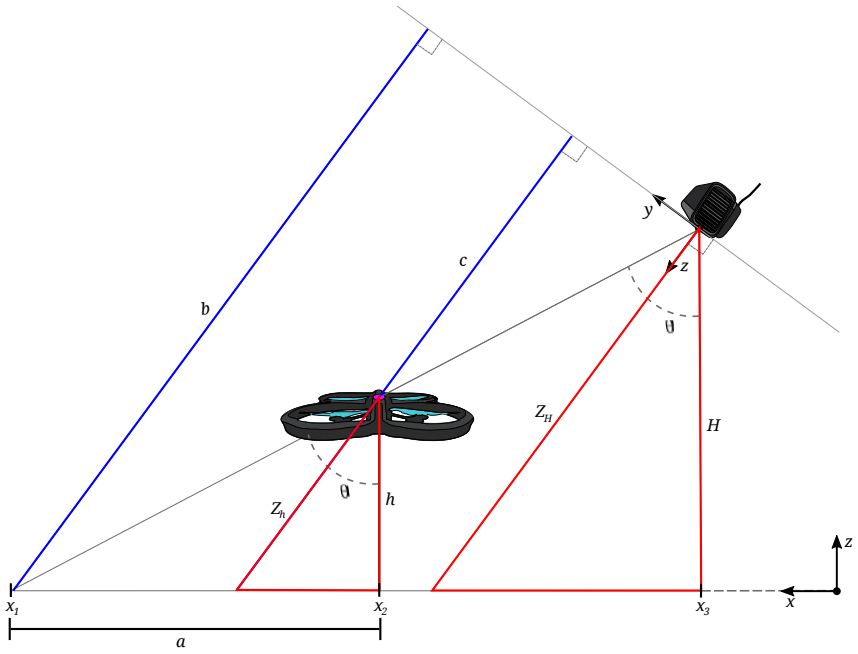


Figure 20: Geometric construct for height calculation

Analyzing the geometric construct from figure 20, we can use the two similar red triangles to calculate the height of the drone relative to the ground. Naming the smaller triangle $\triangle_1$ and the bigger triangle $\triangle_2$, yields:

$$\triangle_1 \sim \triangle_2$$

which leads to

$$\frac{H}{Z_H} = \frac{h}{Z_h} \Leftrightarrow h = \frac{Z_h \times H}{Z_H} \Leftrightarrow h = \frac{(b-c) \times H}{Z_H} \tag{11}$$

Equation 11, determines the height of the drone without the need to use an onboard altimeter as suggested before. It remains the problem that, due to its height, the drone is detected as a marker located

in a point further away than it actually is. Earlier in this thesis, it was presented a way to correct these incorrect marker coordinates, using quantities measured prior to the experiment for each camera. This is not a viable implementation, because every time the position of any camera in the system changes, new measurements are required. To avoid this, an automated method was developed.

Leveraging the geometric properties evidenced in figure 11 and the depth information from the Kinect, three-dimensional coordinates of the marker on top of the drone can be obtained using the pinhole model equations 3.1.1. These coordinates are represented in the camera frame and not in the inertial frame. To convert them to the inertial frame, equation 4 could be used, nonetheless, this would break backward compatibility with the two-dimensional system. The 2D implementation uses one homography matrix per camera to transform points from the sensor plane to the ground plane. To use this already implemented algorithm, it is necessary to calculate the pixel coordinates that the point defined by the marker would have if it was sitting on the ground. In other words, the projection of the point onto the estimated ground plane. Looking at figure 20, the abscissa of this projected point is represented by x_2. This is given by the mathematical method described in section 3.1.5, using the parameters A, B, and C generated with the RANSAC algorithm. Assuming the arbitrary point in the ground plane P_0, as $P_0 = (0, 0, z_0)$, where z is given by the estimated ground plane definition,

$$\begin{aligned} 0 =& A \times 0 + B \times 0 + C \times z_0 + D \\ z_0 =& -\frac{D}{C} \end{aligned}$$

and the normal vector $\vec{n}$ is given by $\vec{n} = [A, B, C]$. So if the detected marker point is $Q = (x, y, z)$, the projected point onto the ground plane is

$$R = (x + kA, y + kB, z + kC)$$

with

$$k = \frac{-xA - yB - zC - \frac{D}{C}C}{A^2 + B^2 + C^2}$$

After this correction, the point R is converted back to pixel coordinates that can be converted into inertial frame coordinates with homography matrices used by the 2D version of the localization system.

To summarize, the 3D localization process starts by generating an estimated mathematical expression to represent the ground plane in the camera frame, followed by detecting the specialized markers with algorithm 3. Next, uses the pinhole model with the depth information, to convert 2D pixel coordinates into 3D coordinates in the camera reference frame, which allows it to get the relative height from the estimated ground plane. Finally, the detected point projection is calculated as previously explained, and using the pinhole model along with the conversion from section 3.1.3, the correct coordinates in pixels are calculated. These last coordinates are then converted to real world ones using the equation 7.

The process described is straightforward and provides the ability to integrate with previous work done for the 2D version of the localization system.

3.3.2 Architecture

All efforts described in section 3.2.2 aimed to create a simple, scalable, and efficient system. These efforts are still valid for the 3D implementation. However, the existence of two different architectures came from the necessity to maintain backward compatibility with previous assignments done by students using iterations of the localization system. But, since they only used the 2D version, there is no backward compatibility to maintain a 3D version of the system. So, the architecture selected to implement the algorithms and methods discussed in the last section is based on the one shown in figure 16, where a distributed topology is used along with remote processing units for each camera with wireless connections to the main computer. This decision provides more processing power available for each Kinect, facilitating the development and integration work. However, as discussed in section 3.4, this led to extra changes regarding software drivers for the Kinect. Nonetheless, in figure 21 is represented the final architecture for the 3D localization system.

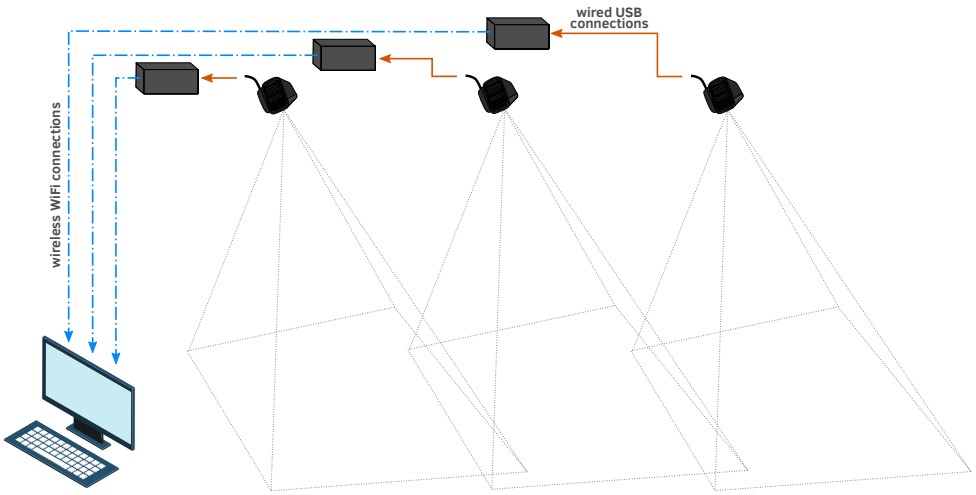


Figure 21: Distributed Architecture for the 3D system

3.4 Remote Cameras

The necessity to dilute and decentralize the processing requirements of the localization system led to the development of a distributed architecture composed by multiple remotely controlled cameras, each equipped with its own processing unit. Excluding the Kinect, the cameras are the same as the ones used in a non distributed architecture. Because of this, the challenges of creating a distributed architecture for the ADIS system were mostly related to the integration of the dedicated processing units and Kinect sensors.

3.4.1 Processing Unit

Each camera has its own dedicated processing unit, responsible for handling all image processing tasks necessary for marker detection. It also handles all of the outgoing and incoming communications with the rest of the architecture components. Every processing unit must be equipped with hardware capable

of connecting to the ADIS network, to control the connected camera or Kinect and process the captured data. It is also loaded with the appropriate software to manage all referred tasks.

Hardware

Hardware wise, the processing unit had different configurations during the development process. These changes were dictated either by software incompatibilities or by hardware availability.

The first iteration of the processing unit was developed using the popular Raspberry Pi 3. This is a widely available, inexpensive single board computer, which offers a compelling set of features. Equipped with an onboard Wi-Fi chip, 1 GB of RAM and with a Broadcom BCM2837 Arm v7 quad-core processor running at 1,2 GHz, it is a development platform easy to customize and with sufficient processing power to handle the required image processing tasks. Its small size (85x56mm), makes it an ideal option for the processing unit for each remote camera. Unfortunately, when connected to the Kinect sensor, some incompatibilities were found regarding the USB connection with the Raspberry Pi, that sporadically failed. After some debugging work, it was discovered that the Kinect connection was stable as long as the USB bus was already under load at the time of the Kinect initialization, using, for example, a USB Wi-Fi dongle. To avoid future problems due to this bug, the Raspberry Pi option was rejected.

Has a replacement, considering budget, availability and time constraints for this thesis execution, it was used a small form factor, consumer-oriented, computer motherboard with an embedded processor. The specific model was an ASRock Q1900M equipped with an Intel J1900 quadcore processor running at 2.42 GHz. It's a Micro-ATX motherboard with dimensions of 185x225x60 mm. Because this is not a fully equipped solution like a single board computer, it was also necessary to add a 2GB stick of ram, a 120 GB SSD for storage, a power supply, and a Wi-Fi network card. The acquired hardware was fitted into a custom made acrylic case, along with two 4 mm fans to provide sufficient airflow for cooling. The final result can be seen in the figure 22.

(a) Remote Camera front view (b) Remote Camera back view

Figure 22: Remote Camera

This alternative has a much larger footprint than a Raspberry Pi, however, it's still acceptable for a

remote processing unit. It also has the added advantage that due to the power supply being integrated inside the custom acrylic case, the Kinect can be powered directly from it, eliminating the need to use another external power supply. The processor embedded in this solution is also more powerful than the Raspberry Pi[23]. The J1900 is also based on an x86 architecture and not on ARM architecture, this provides, for the moment, broader compatibility with other 3D sensing instruments that may be integrated in the future.

Software

The operating system for each camera is based on the Linux distribution, Debian. Debian is a free and open source operating system, which provides a platform with the necessary customization to be fully integrated into a system like the ADIS system. It is also a widely supported distribution with extensive documentation and community support. A minimal installation of the version 8.0 Jessie, was used and the processing unit was configured with the following characteristics:

- Boot automatically when plugged to a power socket
- Connect automatically to the ADIS system Wi-Fi network
- Automatically login as root and run the camera controller software with elevated privileges

The automatic boot is activated in the UEFI setup utility included with the motherboard. The automatic Wi-Fi connection was done according to the official Debian Wiki[25], by editing the network interfaces configuration options. To automatic login as root, *systemd's* capability bundled with the operating system was used. *systemd*[26] is a suite of building blocks for Linux systems. It provides a system and service manager that runs as the first awaken process at boot that starts the rest of the system. It controls and manages a set of units that can be services, mount points, devices, and sockets. To login automatically as root, a special service was created, which runs when the system starts up. To run the camera controller software a starting command is added to the rc.local file for the root user. This rc.local file is used by the system administrators to run special commands after all the normal system services are running. This file is ideal to run the camera controller software after the boot and login processes are complete.

The camera controller software runs on top of the aforementioned operating system and it is responsible for:

- Treating the captured information from cameras and Kinect sensors using the image processing algorithms and transformations previously mentioned
- Controlling the behavior of the camera in the overall system, answering to multiple commands, enabling its remote control and integration
- Sending back all retrieved information to the central computer for further processing.

To implement the image processing algorithms it was used the OpenCV library. As stated on the official website, "OpenCV (Open Source Computer Vision Library) is an open source computer vision and

machine learning software library“[24] and, as of now, is the de facto standard for image processing applications in academic environments. The version used was version 2.4.13. OpenCV is written mainly in C++, but it possesses bindings for Python, Java and MatLab/Octave. The selected language for the main camera controller process was Python. To help with debugging and usability a command line interface was created. This interface uses the *curses* library. This library provides the ability to screen-

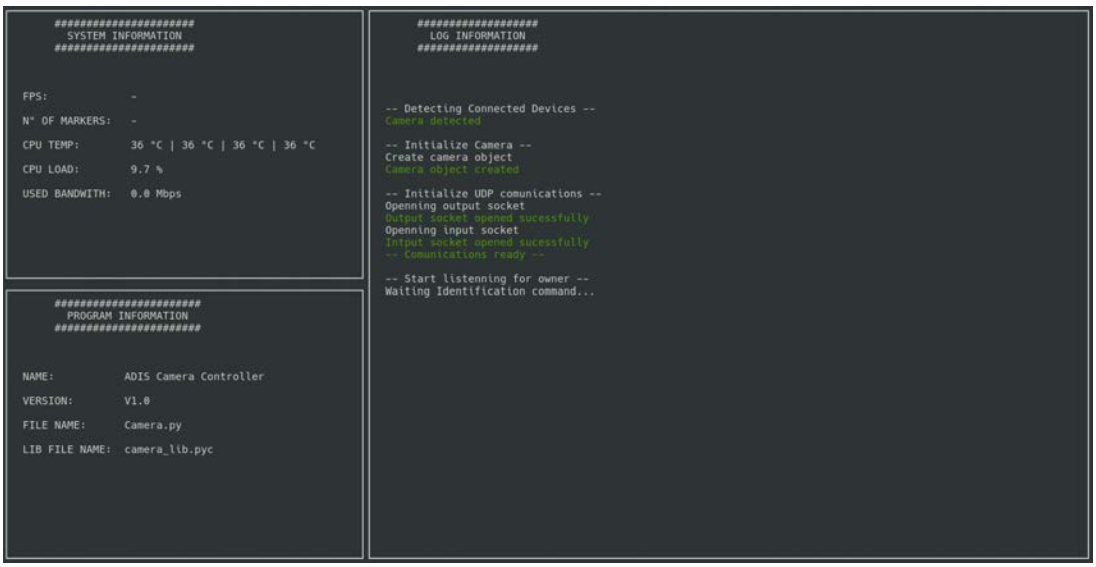


Figure 23: Camera Controller Software Interface

paint a terminal window and create command-line based interfaces. The created interface is composed of three different regions delimited by rectangles. The two smallest regions sit at the left of the screen and on top of each other. The top one displays real-time system information. It is refreshed every 2 seconds and provides details regarding the number of frames per second that are processed and the number of markers detected when running the marker detection algorithm (algorithm 3). It also displays the current temperature for each CPU core, the overall CPU load and the used bandwidth for the Wi-Fi connection. The bottom region displays basic program information like the program's name and version, the current file name and main camera library being used. This library is an integral part of the camera controller software. It contains all functions used to interact and control the Kinect and the cameras. The third and biggest region displays a log recording all events that occur at run-time. This region adds a new line with relevant information every time a new event is logged. When no free lines are available, the region automatically scrolls up and adds the most recent line on the bottom. The displayed information ranges from the received commands, success messages, warning messages, errors and current process being executed. Every message is color coded. Success messages are green, warning messages are yellow and errors are red. Other information is presented with the default white color. It is possible to present messages with a solid color background for other desired purposes. This interface can be accessed by connecting a monitor directly to the remote camera or, when there is no physical access to the camera, remotely using an SSH connection. SSH stands for Secure Shell and it is a cryptographic network protocol to control devices remotely over a network.

Although the communication protocol used to communicate between the remote cameras and the

central computer is described under section 5.1, some relevant functionalities should be mentioned. Every remote camera is fitted with the capability to stream the captured image feed to any connected device. This functionality is not suited for sending the image feed elsewhere for processing and vehicle control. because the feed is slightly delayed compared to reality. However, this image feed is ideal to monitor what is being captured by the camera and how it is processed. To achieve this functionality the camera acts as a server, that upon request from a client, starts capturing data and streaming it. To implement this functionality a framework called Flask was used. Flask is a framework geared towards building web applications. With Flask, a minimalistic web-server is running in the remote camera listening for user requests. When a user connects to the correct IP address and port, this web-server responds with an HTML web-page containing the video feed. When the HTML video container is clicked, the feed cycles through a series of visualization modes. If a Kinect is connected, the video modes are the normal video captured by the camera, a real-time colored depth map from the depth sensor, and a binary image that shows what the image processing algorithm is detecting. If a web camera is connected, every visualization mode is available except the colored depth map. This method allows the video feed to be visible from any device that has web browser capabilities, for example, a smartphone. Besides allowing to see what is captured by the camera in the localization system software interface, it is also vital when installing or moving a camera, to guarantee the correct alignment and coverage.

3.4.2 Kinect

Kinect is a product line, trademarked by Microsoft, of motion sensing devices launched in November 2010[19] along with the Xbox 360, made from licensed technologies from a company called PrimeSense. It's a peripheral for interacting with Xbox games, using the human body instead of a game controller. However, this sensor started to be used for more than just interacting with games. Due to its ability to sense and process spatial data, it quickly became an inexpensive way to add three-dimensional awareness to robotic related projects.

For the ADIS system was used the first version of this sensor called Kinect V1.

Kinect Basic Operation

Official information regarding the Kinect functionality is not publicly available. Nonetheless, it is possible to infer the basic principles by analyzing its hardware components and existing patents from PrimeSense[20][21][22].

As seen in figure 24, the Kinect main components are a regular RGB camera, an infrared sensor, and an infra-red projector. To compute the depth from a scene, the Kinect uses three different computer vision techniques. First, uses a technique called structured light. This technique projects a known pattern onto the scene, which is deformed by the various objects, in view. By analyzing the deformations in the known pattern, depth information can be inferred. The second and third techniques are called depth from focus and depth from stereo. Depth from focus uses the principle that the more out of focus an object is, the further away it is. The Kinect uses a special "astigmatic" lens with different focal lengths for the x and y axes. This type of lens transforms the circles from the projected pattern into ellipsis,

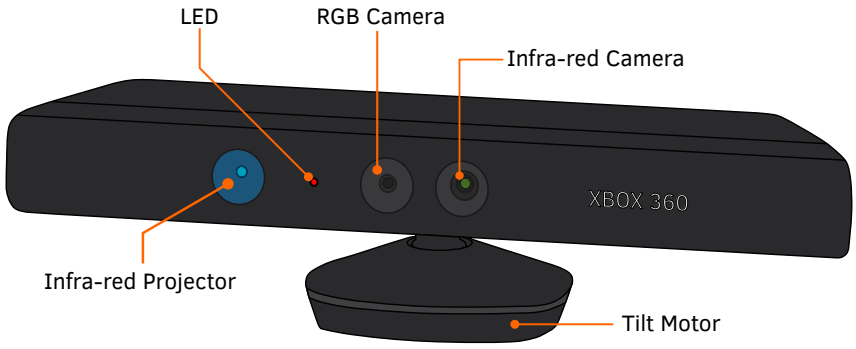


Figure 24: Kinect Hardware Scheme

which are oriented in different directions depending on how far away they are. Depth from stereo uses the parallax effect, that says that when looking at the scene from different angles, the objects closer to the camera are shifted to the side more than objects far away. So, depth is measured by analyzing the perceived shift from the circles of the projected pattern. The second and third techniques are used to improve the accuracy from the depth determined using the structured light. It is important to note that to measure depth, the Kinect uses the built-in infrared projector in tandem with the infrared sensor, and not the regular RGB camera. The measured depth also is not given directly in metric units, but rather raw values with 11 bits of precision, meaning values ranging from 0 to $2^{11} = 2048$. When it is not possible to acquire depth information because the object is to close, the raw value is 0. On the contrary, when the object is too far away the raw value is 2048. To check how these values vary with the object distance, a flat surface was placed in front of the Kinect at different distances. The results are shown in the graph from figure 25. These results can be fitted using a mathematical equation. By consulting

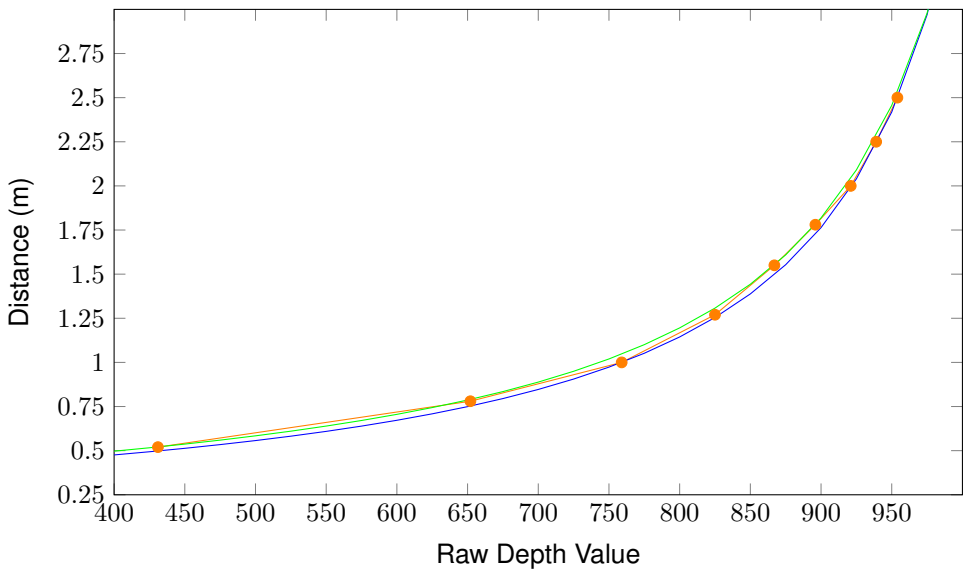


Figure 25: Variation from the Kinect raw depth values with the distance

available documentation regarding the Kinect operation[30] (this documentation refers to the driver used

for the Kinect system integration explored later), there are two already estimated expression

$$depth = \frac{1}{(-0.00307 \times raw\,depth + 3.33)} \tag{12}$$

or

$$depth = 0.1236 \times tan\left(\frac{180}{\pi}(\frac{raw\,depth}{2842.5} + 1.1863)\right) \tag{13}$$

Equation 12 is represented with a blue line in figure 19 and the equation 13 in green. Both fit the acquired measurements, even though the second equation provides more accurate results at longer distances. With these two expressions, it's possible to covert the raw depth information provided by the Kinect to metric units. This, along with the methodologies described in section 3.3, make possible to use a Kinect as a 3D localization sensor.

Kinect Driver Integration

As mentioned before, every remote camera runs an operating system based on Linux. To operate the Kinect, a Linux compatible device driver is required. Because the Kinect is made by Microsoft, officially only a Windows SDK exists, however, there are other unofficial drivers available which are Linux compatible. The OpenKinect[27] community aims to develop a set of free and open-source libraries that make the use of the Kinect hardware with Windows, Linux, and Mac possible. They develop and supply an open-source driver for the Kinect V1 hardware called libfreenect[28]. This drive also includes wrappers for multiple program languages including Python. The driver itself is written in C and the python wrapper uses Cython which is a "superset of the Python language that supports calling C functions"[29]. This driver allows to control the tilt motor, the LED indicator, read from the included accelerometers and microphone array, and capture RGB and depth images. Relevant to this thesis is the ability to control the LED indicator and capture RGB and depth images. Tilt motor control would be interesting to remotely align the Kinect, unfortunately, the tilt motor only actuates if the Kinect is in a horizontal upright position.

This driver can gather data from the Kinect sensors, either synchronously or asynchronously. When using asynchronous functions, the driver executes a certain routine every time newly captured information is available. During testing, this method caused two problems. First, even though the depth sensor and the camera are rated to work at 30 frames per second, the callback routine that runs every time new information is available was being called around 500 times per second. Secondly, there were also some sporadic tearing problems in the depth map information which resulted in the detected markers having no height information. Most likely, this last problem was not the driver's fault, but rather a python's multi-threading bug, because the tearing only appeared when separated threads were used to gather the captured RGB images and depth maps. To avoid these problems a synchronous approach was used. The synchronous functions that come implemented, work by creating a separate C thread that handles all of the captured information and saves it into dedicated buffers. Then, when the image processing algorithm runs the command to get the image and depth map, the driver accesses these dedicated buffers and returns the stored data.

This driver also comes with a "registration mode" that maps the depth information to the image

captured by the Kinect. Because the integrated RGB camera and the depth sensor are not in the same place, but rather side by side, the information is captured in slightly different points of view. So, there is not a direct relation between pixels from the RGB images captured and the depth maps, without any manipulation. Setting the libfreenect driver depth as “REGISTERED” deforms the depth map, matching every pixel from it to the corresponding pixel in the RGB image. The “REGISTERED” mode also handles the conversion from raw depth values to usable metric values expressed in millimeters.

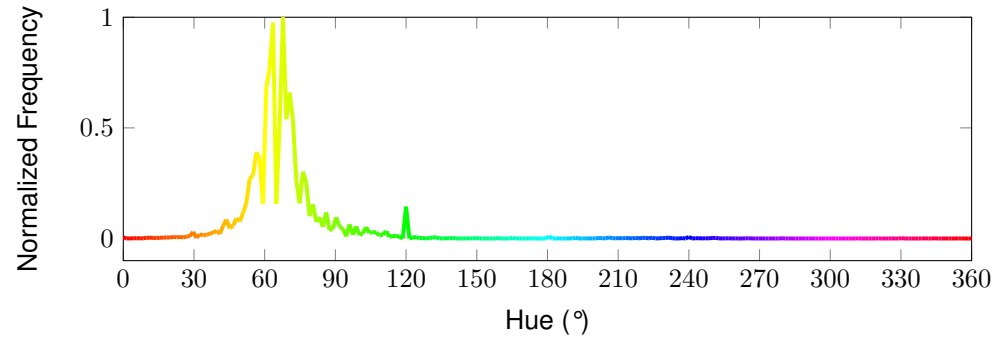
For all the functionality the driver already provides, it needed to be customized to fulfill every function required to use the Kinect in the ADIS 3D localization system. There were some functions added to the main driver source code and some changes were also made to the Python wrapper. A list of these changes can be seen in table 3.

Changes Location	Function Name	Description
python wrapper	write_cmos_register	exposed the ability to write directly to the Kinect CMOS sensor registers
python wrapper	set_ir_brightness	exposed the ability set the infrared projector intensity
python wrapper	set_flag	exposed the ability to set numerous internal flags defined by the driver
python wrapper	camera_to_world	exposed the ability to convert the detected pixel coordinates to coordinates in the camera frame
python wrapper/driver source code	world_to_camera	adds the ability to convert from camera frame coordinates back to pixel coordinates
python wrapper/driver source code	sync_camera_to_world	added the ability to convert the detected pixel coordinates to coordinates in the camera frame when using the synchronous driver API
python wrapper/driver source code	sync_write_cmos_register	added the ability to write to the CMOS sensor registers when using the synchronous driver API
python wrapper/driver source code	sync_set_led	added the ability to control the information LED when using the synchronous driver API
python wrapper/driver source code	sync_set_flag	added the ability to set numerous internal flags defined by the driver when using the synchronous driver API
python wrapper/driver source code	sync_set_ir_brightness	added the ability to set the infrared projector intensity when using the synchronous driver API
python wrapper/driver source code	sync_world_to_camera	added the ability to convert the camera frame coordinates back to pixel coordinates when using the synchronous driver API

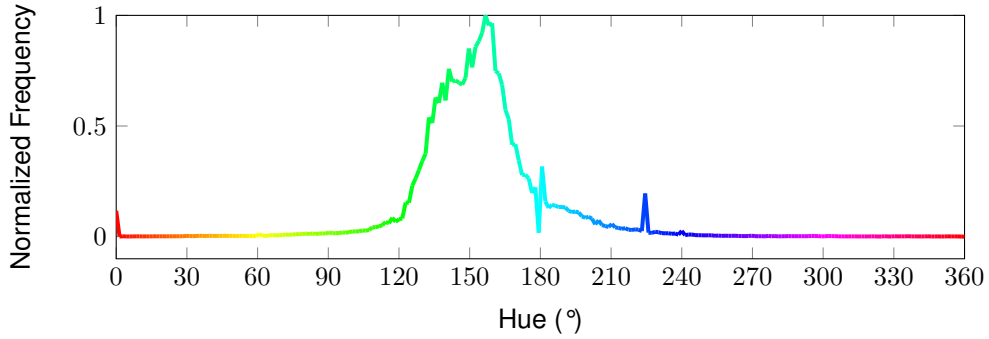
Table 3: Changes and additions to the libfreenect driver

3.5 Markers

The markers used are simple colored circles with five centimeters in diameter. The material used for the markers must be as non-reflective as possible. Reflections from lights in the ceiling can turn the marker completely white. This changes the perceived color of the marker, turning him invisible to the marker detection algorithm. Many materials were tested ranging from 3D printed plastic to synthetic fabrics. The best results were obtained with markers made out of paperboard. To make the marker detectable and avoid false positives, its color must be unique in the frame captured by the camera. To find this color, the histogram in figure 26 was made, from a picture taken with the RGB camera and the Kinect, in the laboratory where the system was installed.



(a) Histogram from picture taken with the Logitech C615 camera



(b) Histogram from picture taken with the Kinect

Figure 26: Laboratory picture histogram

In these histograms, the rarest colors are the ones with a hue between $270°$ and $360°$. These hues represent violet, pink and red. Markers with these three colors were tested to see how well they were detected by the RGB camera and the Kinect. Using simple visual inspection of the images captured by each camera for the three colors, the color pink with a hue around $300°$, was selected for the markers.

3.6 Overlap Zones

The ADIS system is capable of using multiple cameras to cover an area larger than it would be possible to cover using only one camera. In order to do this without blind spots, either the cameras are perfectly aligned with each other or overlap zones occur, where more than one camera covers the same area. In these overlap zones, one marker is the field of view of multiple cameras, meaning that is perceived as

more than one point. This would be easy to correct if the localization system was perfectly aligned, and one marker spawned multiple detections with exactly the same coordinates in the user-defined inertial frame. However, the system has inherent errors that make different cameras detect the same marker in slightly different positions. To solve this problem, it is possible to analyze the shape "drawn" by the markers instead of its location, assuming that the perceived shape does not completely change with the camera point of view.

The Procrustes analysis was used to interpret the shape "drawn" by multiple points. The Procrustes analysis determines a linear transformation (translation, reflection, rotation, and scaling) that best conforms a given set of points into another. The methodology used is described in algorithm 4.

Algorithm 4 Overlap Zone Correction Algorithm

Require: $Overlap\ Zones$, $Detected\ Points$w

for every $Overlap\ Zones$ **do**
  if any $Detected\ Points$ is in $Overlap\ Zones$ **then**
    $Points\ Inside \leftarrow Detected\ Points$ that are inside $Overlap\ Zones$
    if the $Points\ Inside$ were captured by diferent camera **then**
      $Points\ Inside\ Camera\ 1 \leftarrow$ every point from $Points\ Inside$ captured by the first camera
      for every $Camera$ that captured points excluding the frist one **do**
        $Points\ Inside\ Camera \leftarrow$ every point from $Points\ Inside$ captured by $Camera$
        $Transformed\ Points \leftarrow$ used $procrustes$ to transform $Points\ Inside\ Camera$ into $Points\ Inside\ Camera\ 1$
        if the $distance(Transformed\ Points, Points\ Inside\ Camera\ 1)$<$Limit$ **then**
          Clear from $Detected\ Points$ the $Points\ Inside\ Camera$
          Replace from $Detected\ Points$ the $Points\ Inside\ Camera\ 1$ by the mean between $Points\ Inside\ Camera\ 1$ and $Points\ Inside\ Camera$
        end if
      end for
    end if
  end if
end for

Algorithm 4 identifies the repeated detected markers using one camera as reference, eliminates these extra points and replaces the remaining correct points by the mean between the remaining point and the extra removed point. To decide if the points are actually repeated pairs, the distance between them needs to be below a certain limit (variable $Limit$ in algorithm 4). This limit was determined by trial and error, doing multiple experiments and choosing a sane value that makes the algorithm work as intended. Replacing the repeated point pairs by arithmetic means is done mainly to aid the tracking algorithm described in section 3.7. Because the cameras are not optically perfect, when a marker goes from one camera point of view to another, it suffers a small translation in its inertial frame coordinates. When using markers placed too close from each other, an abrupt change in position can be challenging for the tracking algorithm to follow. Using a mean smooths this translation, allowing the transition from one camera to another to be less abrupt.

3.7 Tracking Algorithm

The sole purpose of this algorithm is to interpret the detected marker coordinates and associate these coordinates to a fixed identity through time. Discussing different approaches for tracking algorithms and evaluate their advantages and disadvantages, is outside of the scope of this thesis.

The chosen tracking technique is motion-based tracking using Kalman filters [56]. The implemented algorithm is heavily based on the "Motion-Based Multiple Object Tracking" example contained in the Computer Vision System Toolbox[52] for MatLab from MathWorks. The basic algorithm for this tracker is as follows:

Algorithm 5 Motion Based Tracking Algorithm

$Tracks \leftarrow IniatlizeTracks()$

while running **do**
$Frame \leftarrow Get\_Frame()$
$Points \leftarrow Detect\_Markers(Frame)$

$Tracks \leftarrow Predict\_New\_Locations\_Of\_Tracks(Tracks)$
$Matched, Unmatched\_Tracks, Unmatched\_Points \leftarrow Match\_Points\_To\_Tracks(Points, Tracks)$
$Tracks \leftarrow Update\_Matched\_Tracks(Tracks)$
$Tracks \leftarrow Update\_UnMatched\_Tracks(Tracks)$
$Tracks \leftarrow Delete\_Lost\_Tracks(Tracks)$
$Tracks \leftarrow Create\_New\_Tracks(Unmatched\_Points)$
end while

Before the tracking loop starts, a variable called $Tracks$ is initialized. This variable is a set of tracks objects which contain, among other properties, an identification number, an age count, total visibility count, and a Kalman filter. This Kalman filter object is already implemented in the computer vision toolbox from MatLab[52].

To explore Kalman filters and their properties is also outside the scope of this thesis. Kalman filters are self-updating algorithms, in their simplest form used on linear systems, which use a series of measurements over time, containing Gaussian noise, to produce estimates of unknown variables. They start by describing the current system state, called state space, with a $posteriori$ state estimate matrix, $\hat{x}_{k-1}$, and $posteriori$ error covariance matrix, P_{k-1}, at time $k-1$. This assumes that the current state can be described by an n-dimensional Gaussian with mean $\hat{x}_{k-1}$ and covariance P_{k-1}. To predict the next state of the system a state-transition model in the shape of a prediction matrix F_{k-1} is used. This matrix describes how the linear system should evolve between each iteration of the Kalman filter. Applying this matrix to the current state space, to move forward in time to k, gives:

$$\hat{x}_k = F_k \hat{x}_k$$
$$P_k = F_k P_{k-1} F_k^T$$

To take into account the uncertainty caused by unknown variables not expressed by the state-transition model, the Kalman filter adds a control matrix $B_k$and a control vector $\vec{u}_k$, along with a process noise

covariance matrix Q_k.

$$\hat{x}_k = F_k\hat{x}_{k-1} + B_k\vec{u}_k \tag{14}$$

$$P_k = F_kP_{k-1}F_k^T + Q_k \tag{15}$$

The equations 14 and 15 are called the prediction step of a Kalman Filter. This step produces a fuzzy state of the entire linear system. To add measurements from a sensor which evaluates the system at time k, it is assumed that each measurement $\vec{z}_k$, returns with Gaussian noise $v_k$and covariance matrix R_k, deformed according to the observation model matrix $H_k$characteristic of the sensor.

$$\hat{z}_k = H_k\hat{x}_k + v_k \tag{16}$$

$$P_k = H_kP_kH_k^T + R_k \tag{17}$$

And that the value for the expected measurements is given by,

$$\hat{\mu}_{expected} = H_k\hat{x}_k \tag{18}$$

$$\Sigma_{expected} = H_kP_kH_k^T \tag{19}$$

The optimum estimation for the measurement at time k is given by the Gaussian mixture of the two Gaussian blobs defined by equations 16 and 17 and equations 18 and 19, which gives the final set of equations for the Kalman filter:

$$\hat{x}'_k = \hat{x}_k + K'(\vec{z}_k - H_k\hat{x}_k) \tag{20}$$

$$P'_k = P_k + K'H_kP_k \tag{21}$$

$$K' = P_kH_k^T(H_kP_kH_k^T + R_k)^{-1} \tag{22}$$

These three equations are known as the update step of the Kalman filter were $\hat{x}'_k$ and P'_k are the best estimate of the system state at time k, and the quantity K' is called Kalman Gain.

Summing up, Kalman filters work in a two-step process, composed by a prediction step and an update step. In the first step, the Kalman filter produces an estimate for the next variable state, and once the actual measurements for that state are gathered, the Kalman filter is updated by comparing the prediction with the real measurements.

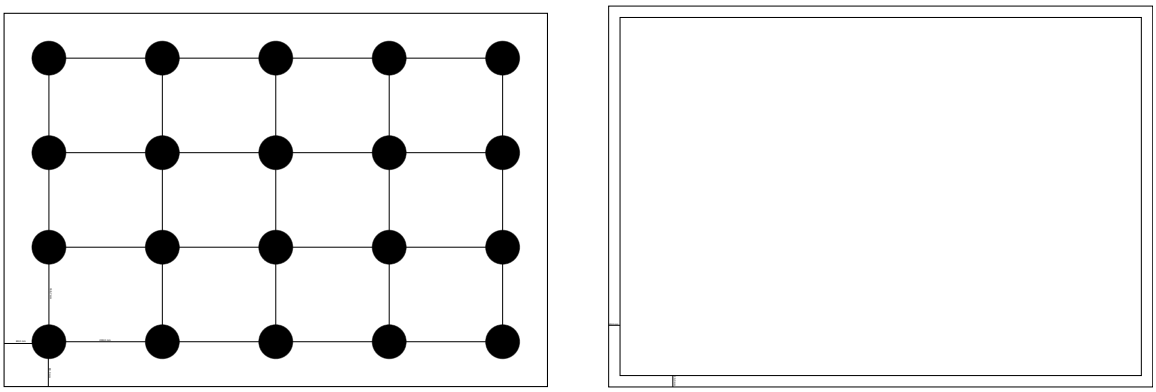
In algorithm 5, after reading the current frame and calculate the real world coordinates for each detected marker, the tracking algorithm tries to predict the next location for each track. This task is only executed if there are already markers being tracked. This corresponds to the prediction step of the Kalman filter. After which, the algorithm tries to assign the detected points to an already existing track. This assignment process involves two steps: first, the cost of assigning every detected point to each track is computed using the distance method provided by the Kalman filter object from the Computer Vision toolbox[52]. This cost accounts for the Euclidian distance between the detected point and the

track predicted location. The second step solves the track to point assignment problem. The included Matlab function $assignDetectionsToTracks$ is used and calls the Munkres version of the Hungarian algorithm[54] to compute an assignment - which minimizes the total cost. Next, the tracking algorithm updates the assigned tracks. This corresponds to the update step for the Kalman filter of each track. It uses the newly matched point to update the Kalman filter of the corresponding track calling the correct method provided by the Kalman filter object. The $Update\_Unmatched\_Tracks$ function just increases the age of the unsigned tracks by 1. The next function call deletes tracks that are considered lost. When they have been invisible for too many frames, or it was recently detected but has been invisible for too many frames overall. The parameters that indicate these two states are the age parameter and the total visibility count. Finally, the algorithm handles points that have not been deleted or assigned. These points are considered new tracks and are added to the $Tracks$ variable with a unique identification number.

3.8 Calibration Process

Any measurement instrument needs to undergo a calibration process to provide accurate readings. The ADIS localization system is not different and it has its own calibration process, that aims to be simple to use and to provide accurate results. This calibration process has some variations depending on the architecture being used.

The first step in the calibration process is common to all architectures. Its objective is to estimate the planar homography matrix for each camera, and it needs to be repeated as many times as the number of cameras in use. This step provides the system with a set of points on the ground, measured relative to the final inertial reference frame, that is then used to estimate the planar homography matrix for each camera. Measuring each point individually every time the system needs to be calibrated, would be tedious and time-consuming. Thus, in order to automate this process, a black and white calibration pattern is used, with twenty points organized in a 5x4 grid. Because the pattern dimensions are known, it



(a) Top Calibration Pattern (b) Bottom Calibration Pattern

Figure 27: ADIS Calibration Pattern

is only required for the user to provide two points. These two points indicate the position and orientation of the calibration pattern. This is enough information to automatically generate the rest of the pattern

points positions. With all calibrations points the system then estimates the homography matrix using the singular value decomposition technique described in 3.1.7. The calibration pattern is located near the middle, where the lens distortions are less effective. This decision was tested against placing points uniformly spaced across the entire frame. For this test, all points were measured with a measuring tape and compared to the ones given by the calibrated localization system. The final results are in table 4. The mean error when the calibration pattern is roughly in the middle of the frame is almost two

Pattern Position	Error	
	mean (cm)	standard deviation (cm)
Centered	4,1	4,7
Scattered	5,9	7,5

Table 4: Calibration Pattern Location Tests Results

centimeters bigger than when the calibration points are placed across the entire frame. The same is true for the standard deviation. This error increase is most likely related to lens distortions around the edges. The estimation algorithm fits these distortions in the homography matrix, increasing the overall error. In reality, these differences are not that pronounced when compared to the size of the vehicles in use, but they justify the choice of a centered calibration pattern instead. A centered calibration pattern also has the advantage of being easier to handle, than a pattern placed across the entire field of view. Although these measurements were not taken using any camera currently in the ADIS system, but rather taken using a Logitech QuickCam 3000, they are still valid because the cameras currently implemented (the Logitech C615 and the Kinect) use similar technologies and lens assemblies to capture RGB images.

The full calibration algorithm is given by algorithm 6. To start the calibration process it's necessary

Algorithm 6 Marker Detection Algorithm with RGB color space

Require: $Frame\_Top$ with both calibration patterns, $Frame\_Bottom$ with just the bottom calibration pattern

Generate $Complement\_Frame\_1$ and $Complement\_Frame\_2$
$Auxiliary\_Frame \leftarrow Complement\_Frame\_1 - Complement\_Frame\_2$

$threshold\_value = 0.1$
while $threshold\_value < 1$ **do**
  $Binary\_Frame \leftarrow threshold(Auxiliary\_Frame)$
  $Centroids \leftarrow blob\_analisys(Binary\_Frame)$
  if number of $Centroids = 20$ **then**
    break while cycle
  end if
  $threshold\_value = threshold\_value + 0.01$
end while

Order $Centroids$ according to the calibration pattern
Ask for two points in the user inertial frame
Generate $RealWorldPoints$ from the given pair of points
Esimate the $Hommagraphy\_Matrix$ using the Singular Values Decomposition (SVD) technique

to acquire two different images, one with both the top (figure 27a) and bottom parts of the calibration pattern (figure 27b), and another with just the bottom part. This requires that the bottom calibration

pattern stays in the same place between images. From each image, a complement image is generated. A complement image, also called a negative image, is obtained by replacing every pixel by the difference between the maximum value that the pixel can have and the current value. Considering the 8-bit RGB color space in use, the complement pixel of $px = (r, g, b)$ is given by $(2^8 - r, 2^8 - g, 2^8 - b)$. Because the top part of the calibration pattern has a white background with black dots, the complement image gives a pattern with a black background and white dots. When subtracting both images, the pixels that are not part of the pattern are removed, and the pixels that are part of the calibration pattern, show the pattern dots in white. The two parts of the pattern ensure that the complement image without the calibration dots always has a black rectangle where the calibrations dots are. This guarantees that the dots are not removed by dark areas on the floor, after subtracting both images. However, due to small variations in lighting and captured noise, the subtraction does not fully remove every pixel that isn't a calibration pattern dot. To remove this remaining noise, an increasingly higher $threshold\_value$ is applied to the subtraction result. This removes more and more noise until only the calibration dots remain visible. Then the centroids for each calibration point are identifying and matched with points in the inertial frame defined by the user for the experiment. This task starts by ordering all centroids in a predetermined sequence, to facilitate the matching process. Then the user inserts the coordinates of two points in the inertial frame. Since the pattern dimensions are known, the rest of the calibrations points can be generated with the same order as the one used to organize the previously detected centroids. The last step is to estimate the homography matrix that transforms the centroids into the generated points in the user-defined frame, using Singular Values Decomposition.

This calibration process creates a homography matrix for every camera that is added to the ADIS system. This matrix can be saved for later use, removing the need to calibrate every camera when it's added to the system, as long as the camera does not move.

4 Vehicles

Considering that the ADIS system aims to provide a complete platform to develop and test advanced control and navigation algorithms, it would not be complete without a fleet of vehicles. These are based on commercially available solutions that can easily be acquired physically in model building stores or online via the official web-stores. Every airborne vehicle was prepared to integrate the ADIS communication network and expose interfaces for control and onboard sensor access. The integration was made to provide access to every aspect of the integration and components for the user to explore and change, without the fear of irreparably breaking the vehicle. The reason for this concern along with a description of the investigative work related to the flying vehicle is presented in this chapter.

4.1 Airborne Vehicles

As far as flying vehicles are concerned, there was no shortage of options available in the current market. Multiple incremental advances in technologies around cost and weight reduction of good kinetic sensors, batteries and low energy processors, lead to a rather wide selection of remote controlled low-cost vehicles with multiple different sets of capabilities to be available. Commonly known as “drones”, these flying machines come in a vast range of configuration and prices, depending on their target mission and consumers. The majority of them come in a quad-copter configuration. This configuration is ideal for indoor environments due to its stability and capability of vertical takeoff and landing. A quad-copter is composed of four rotors connected to four independently controlled motors. As mentioned before, the ADIS system communication network relies on Wi-Fi technologies. To keep every system component using similar technologies and avoid unnecessary compatibility problems, the selected vehicle should also rely on Wi-Fi communications. This requirement narrowed the drone selection process and led to the final decision of using an AR.Drone 2.0 from Parrot.

4.1.1 AR.Drone 2.0 Characteristics

The AR.Drone 2.0 is the second iteration of the medium range offering from Parrot. It is a Wi-Fi controlled aircraft, equipped with auto stabilization capabilities, and a wide range of sensors. It has a range of 50 meters and a maximum speed of 11.1 meters per second. As product targets the amateur market, and can fly both in outside and inside environments. It can capture video using two onboard cameras and is controlled via a dedicated application running on a smartphone. The full specifications are mentioned below.

Structure and Body

The AR.Drone 2.0 is built around a central cross made of carbon fiber arms connected by plastic fittings. The onboard electronics and motors are fixed to this central cross, which is integrated with the interior hull. This interior hull is made out of expanded polypropylene and it keeps the battery, the front camera, the sonic sensor, and the USB connector in place. Clipped to it, using magnets, there is an exterior hull. Its main function is to protect the drone. Included in the box come two different exterior

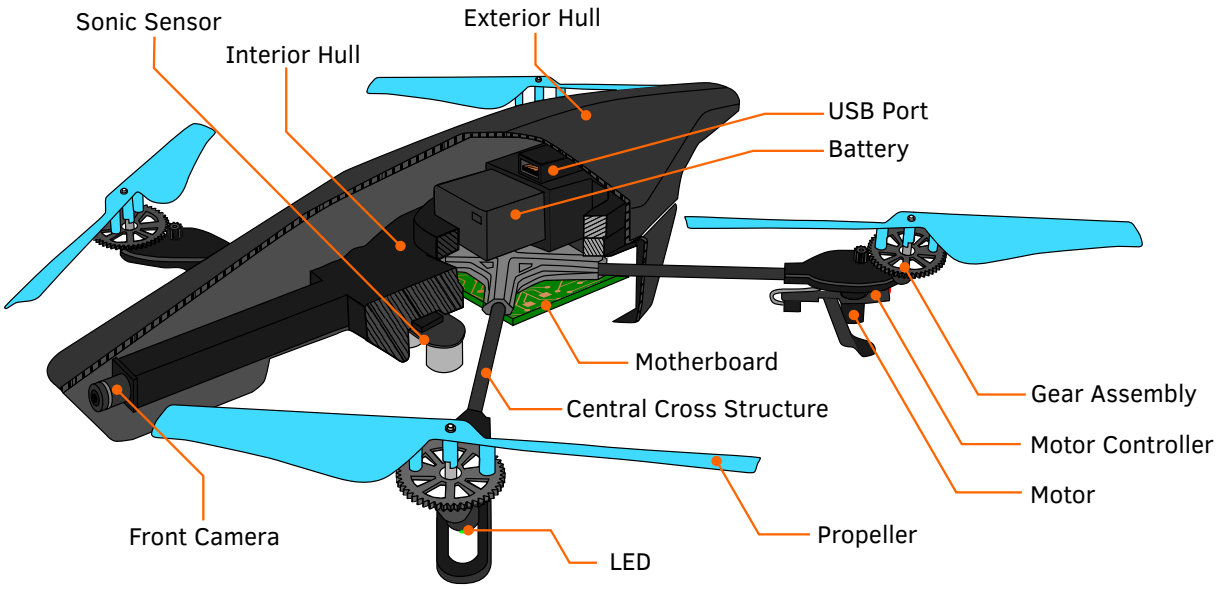


Figure 28: AR.Drone 2.0 cut-out

hulls. One for outdoor flights that is lightweight but only protects the body of the drone. The other is for indoor flights and it offers more protection against collisions by wrapping around the propellers, at the expense of adding more weight compared to the outdoor hull. Both hulls are made out of expanded polypropylene like the interior hull.

Motors and Propellers

The AR.Drone 2 is propelled by four 15 Watts three-phase brushless DC motors equipped with plastic propellers, specially designed for this drone. Each motor operates at 28 000 rpm, that is converted to 3 000 rpm for the propeller, through a small low noise Nylatron gear assembly. Each motor is controlled by 8-bit MIPs AVR MCU and 10-bit ADC [35, 36] to manage the rotation speed.

On start-up the drone automatically detects and tests each motor individually. When one rotor is blocked mid flight, the drone stops all engines immediately. This prevents extended damages to the plastic rotor, and injuries if the drone hits person.

Batteries

The AR.Drone 2.0 is powered by one 1000 mAh or 1500 mAh, 11.1 V, Lithium-Polymer battery. The battery provides 12 minutes or 18 minutes of flight time depending on the battery version used. The battery discharges from 12.5 V when fully charged, to 9 V.[35, 37]

The drones calculates the available battery percentage from the battery's voltage. Mid flight the battery life can be red directly from the drone's telemetry stream, but it is also indicated trough the LED's located below every motor. When fully charged, all four LED's are green, and turn red as the battery discharges. When the battery hits 20%, the drone does not allow take-off. If the drone is flying and the battery reaches a critical level, the drone automatically lands to avoid any unexpected behavior.

onboard Electronics

Inside the AR.Drone 2.0 there is a set of sensors and embedded electronics responsible for stabilization and control. Below the central hull, there is a motherboard equipped with:

- 1 GHz ARM Cortex A8 microprocessor (ARM v7 architecture),
- 8 GHz video DSP TMS320DMC64x,
- 1 GB of 200 MHz DDR2 RAM
- Atheros Wi-Fi 802.11 b/g/n chip
- Pressure sensor with +/- 10 Pa precision
- 30 frames per second forward facing HD camera
- 60 frames per second downward QVGA camera
- USB-A 2.0 port
- Molex 2-pin Female connector
- Reset Button

Connected to this motherboard, there is a navigation card containing all the Kinetic sensors. This card features:

- a 3 axis gyroscope with 2000°/second precision
- a 3 axis accelerometer with +/- 50mg precision
- a 3 axis magnetometer with 6° precision
- Ultrasonic sensor

The motherboard stores and process the operating system that runs the program responsible for controlling the drone. It has a dedicated DSP (digital video processor) to handle video encoding tasks, a Wi-Fi chip for LAN connection, a pressure sensor to determine height above 3 meters of altitude and a down facing camera for ground speed measurements. The navigation card contains all of the motion sensors. The gyroscope, accelerometer, and magnetometer compose one IMU (inertial measuring unit) with 9 DOF (degrees of freedom), that is used for automatic pitch, roll and yaw stabilization and assisted tilting control. The ultrasonic sensor uses ultrasound reflections to detect the height of the drone below 3 meters of altitude, for height and vertical speed control. The USB port supports USB 2.0 protocol and can only read USB keys with a grounded USB connector to the casing and formatted using the FAT32 file format. The battery connects to the Molex 2-pin connector to provide power.

onboard Cameras and Video Stream

This drone comes equipped with two cameras, one is front facing and mounted on the “nose” of the interior hull, the other is facing the ground and it is attached to the motherboard. The front-facing camera is capable of recording in native 360p (640x360) or 720p (1280x720). The down-facing camera captures at native 240p (320x240) and 360p or at 720p with image up-scaling.

The video stream frame rate from both cameras can be adjusted for 15 or 30 FPS, however, the bottom camera can record at 60 FPS at 240p. Using the included 8-bit DSP the video can be encoded using MPEG4 and H.264 video codecs. There is an option to save the recorded video directly to a flash drive using the USB 2.0 port.

4.1.2 Basic Operation

Being a quadcopter the AR.Drone 2.0 uses four motors equipped with propellers, to fly. Each pair of opposite motors turns in the same direction. One pair turns clockwise, while the other turns counter-clockwise.

4.1.3 AR. Drone 2.0 Integration

The objective of the integration process was to examine, explore and eventually modify the drone to allow its use in the ADIS system. This required extensive research and the use of appropriate scanning tools, supported by the AR.Drone Developer Guide version 2.0 [38]. This guide describes the official communication protocols used by the drone for remote control, data and video streaming, and remote configuration. However, this information does not disclose how the operating system works. This lead to an investigatory phase to explore customization options to integrate the drone with the rest of the system.

Drone Examination

The first step taken to explore the drone was simply a visual analysis of its hardware to look for any kind of communication interfaces that could provide a way to analyze the AR.Drone 2.0 software. The only hardware interface found is located under the drone, soldered to the motherboard (figure 29). This connector is a serial port that can provide access to the operating system. This serial port cannot be connected directly to a serial port from a computer, doing so would potentially damage the motherboard or some of its components because the drone’s serial port uses TTL voltage levels ($\pm$5V) and not common RS-232 voltage levels ($\pm$13V is the most common for PC’s). An RS-232 to TTL converter is required to connect a computer to the drone safely. A USB to UART (TTL level) converter can also be used. Because none of these adapters were readily available another approach taken.

When the drone is connected to the battery, it turns on and starts a startup sequence. First, the motors LEDs turn red and, after a few seconds, every motor vibrates slightly, one at a time. The drone creates his own Wi-Fi hotspot and the four motor LEDs turn green. This marks the end of the startup sequence and the drone is ready to fly. The hotspot created does not have a password and its ESSID is

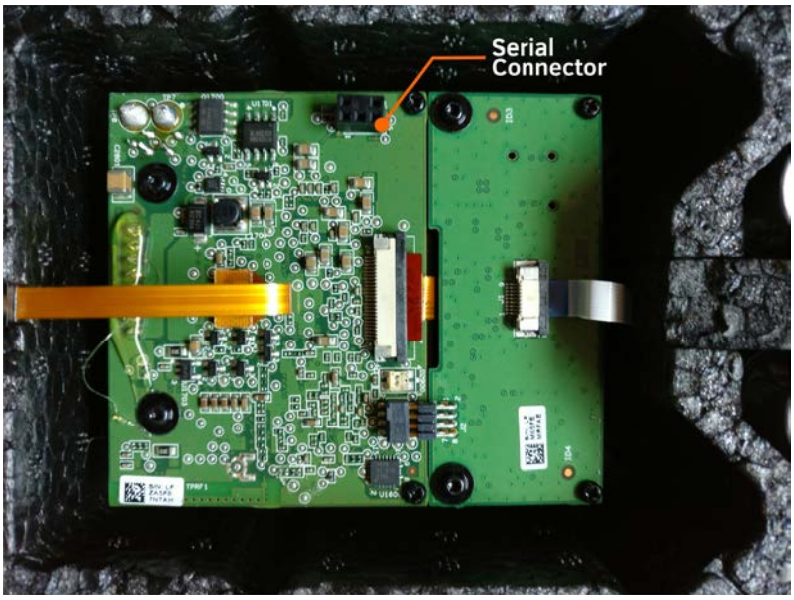


Figure 29: AR.Drone 2.0 bottom mounted serial port

called “ardrone2_xxxx”. As per the SDK[38] the drone identifies itself on the network using a specific IP address, typically “192.168.1.1”. The SDK also reveals that communication with the drone is done using three main services through three different ports:

- Using the UDP port 5556, the drone can be controlled and configured using a set of *AT commands*, these commands are interpreted by the drone up to 30 times per second.
- Using the UDP port 5554 the drone sends to the client information regarding its status, position, speed, orientation, etc. This information is called *navdata*, and it is sent 15 times per second when running in a mode called “demo” mode, or 200 times per second when running in “debug” mode.
- The video stream is sent through UDP using the port 5555.

There is also a fourth communication channel, called *control port*. This connection runs using a TCP protocol on the 5559 port. This mode is used to send critical information that cannot be lost, like retrieve or set configuration data.

Since the network has no password any device can connect freely to the hotspot. Usually, a smartphone running the remote control application would connect, recognize the drone and start its remote control. However, to examine the drone, a laptop running a Linux based operating system was connected to the network and a series of tests executed.

First to be sure that the drone is located at the IP “192.168.1.1” a simple *ping* command was sent. This command had an immediate response confirming the drone’s presence. To see if there were any remote interfaces available to connect and possibly modify the drone, a scanning tool called *Nmap* was used. According to its website “Nmap ("Network Mapper") is a free and open source (license) utility for network discovery and security auditing.”. This application can scan large networks and analyze “what hosts are available on the network, what services (application name and version) those hosts are offering, what operating systems (and OS versions) they are running, what type of packet filters/firewalls

are in use, and dozens of other characteristics". *Nmap* can scan large networks or be used in single clients as well. So, pointing *Nmap* to the drone's IP address revealed the results presented in table 5. The results shown are only relevant results. More ports were detected, but are not relevant for this analysis.

Discovered Open Ports	Protocol Type	Possible Service
21	TCP	ftp
23	TCP	telnet
5555	UDP/TCP	freeciv?

(a) Port summary

Discovered Operating System	Version
Linux	2.6.x

(b) Operating System summary

Table 5: *Nmap* relevant results summary

Three relevant ports were discovered. Port 5555, as seen before, corresponds to the video streaming port. The *Nmap* tool tried to label it as a port used for a strategy game called Freeciv. The two remaining ports, however, reveal that the drone is running a ftp and a telnet service. This reveals a way to remotely interact with the drone, telnet, and how to send files to the drone, ftp. Connecting to the drone is them achieved by running $telnet192.168.1.1$ when connected to the drone's hotspot.

After connecting to the drone using the telnet command, a Linux terminal session starts, as detected by the *Nmap* tool. This terminal session is running as root. The Linux kernel version that the drone is running is version 2.6.32.9. With this terminal access, the drone root file system was explored.

The AR.Drone 2.0 is running a lean Linux operating system, that provides a very basic set of tools. Using the *df* command, revealed that the total system memory is 213,2 MB for this particular drone, with 157 MB free, which leads to the conclusion that the AR.Drone 2.0 model probably comes equipped with a 254 MB internal memory.

Further exploration of the root file system led to the discovery of the two important files: the "config.ini" and the "rcS". The "config.ini" file is located under the "/data" directory, and it is a configuration file containing useful information and settings regarding multiple aspects of the AR.Drone 2.0 state. This file is organized in multiple sections, preceded by an identifying title. Table 6 is a list summarizing the available settings and information in the "config.ini" file organized by section:

Table 6: File "config.ini" information and settings

Section Name	Information/Settings	Data type	Data Attributes
General	total flight time	number	Read Only
	Drone's hardware and software versions	number	Read Only
	Drone serial number	number	Read Only
	Motors hardware and software number	number	Read Only
	Drone's name	string	Read and Write
	NavData demo mode	boolean	Read and Write
	Video enable	boolean	Read and Write

	Vision enable	boolean	Read and Write
	Minimum Battery voltage	number (in millivolts)	Read and Write
	GPS software and hardware versions	number	Read Only
	Timezone	number	Read and Write
	Battery Type	number	Read and Write
Network	SSID single player	string	Read and Write
	SSID multi player	string	Read and Write
	Wi-Fi mode	number	Read and Write
	Owner Mac address	number pattern	Read and Write
Control	Accelerometer offset and gains	vector	Read Only
	Gyroscope offset and gains	vector	Read Only
	Magnetometer offset and radius	vector	Read Only
	PWM reference for Gyroscope	number	Read Only
	Maximum and Minimum Altitude	number	Read and Write
	Outdoor Mode	boolean	Read and Write
	Flight without shell	boolean	Read and Write
	Indoor maximum Euler angle, upwards velocity and yaw	number	Read and Write
	Outdoor maximum Euler angle, upwards velocity and yaw	number	Read and Write
Pic	Ultra-sound frequency	number	Read and Write
	Ultra-sound version	number	Read and Write
Video	Camera FPS	number	Read Only
	Camera Buffers	number	Read Only
	Number of Trackers	number	Read Only
	Video on USB mode	boolean	Read and Write
	Codec FPS	number	Read and Write
	Exposure Mode	number pattern	Read and Write
	Saturation Mode	number pattern	Read and Write
	White Balance Mode	number pattern	Read and Write
Detect	Enemy Colors	number	Read and Write
	Enemy without shell	number	Read and Write
	Ground stripe colors	number	Read and Write
	Detect Type	number	Read and Write
Syslog	Output	number	Read and Write

GPS	Latitude	number	Read Only
	Longitude	number	Read Only
	Altitude	number	Read Only
	Accuracy	number	Read Only
Flightplan	Settings and information about automatic flight using GPS navigation		
Custom	Application description	string	Read Only
	Profile description	string	Read Only
	Session description	string	Read Only

In Table 6, some information and settings were omitted, because they are not relevant to this thesis scope. Comparing the available settings above with the Developer Guide[38], revealed that many of these settings and information can be set and read via already available commands. However, some capabilities are not implemented in the drone's protocol, which makes the method of manually change the "config.ini" file, the only way to access some capabilities. As shown in table 6, there is a way to change the network settings of the AR.Drone. The network mode option is described in the Developer Guide[38]. There is no remote command implemented to change this option, but it can be manually set to three different values:

- "0" is the default value and configures the drone to create its own Wi-Fi hotspot
- "1" configures the drone to create or connect to a network in Ad-Hoc mode
- "2" forces the drone to connect to a network defined by the SSID described by "SSID single player" configuration parameter

Since the ADIS system has its own Wi-Fi network, the option number "2" is the one that configures the drone as desired. Unfortunately, during testing, the drone was only able to connect to networks that were not password protected. This presents some problems discussed later. Reading the second file called "rcS" contents revealed that it is an initiation file that contains the necessary commands for the drone to boot properly. This is also corroborated by the fact that this file is located at the "/etc/init.d/" directory, which is the directory that traditionally contains the scripts used by System V initiation tools. Inside this file bash routines are called to initiate the Wi-Fi network setup, to initiate the motors and more. There is one particular routine to decide which IP address the drone will have inside the network. This routine is interesting because it can be easily changed to place the drone in a predetermined IP address.

Executing the command *ps* returns a snapshot of the current processes running inside the drone. This, revealed two interesting processes called "/bin/sh /bin/program.elf.respawner.sh" and "/bin/program.elf ". The first process is a "respawner" process, responsible for kipping the second process alive if, for some reason, it terminates. To observe what the second process does, first, the "respawner" process must be terminated and then the "program.elf" process can be terminated too without respawning. When this happens, the drone immediately stops even if it is mid-flight, falling directly to the ground, changing the motor LEDs to red. If the "program.elf" is then initiated using the telnet terminal, and

some text is printed to the terminal highlighting the main operations being executed. This text shows that the second process is responsible for controlling and flying the drone. It is the main process of the drone, that receives and executes the user commands, analyzes the onboard sensors data and keeps the drone flying in a stable manner. Because it has direct access to the onboard sensors and it controls directly the motors, if any other navigation algorithm wants to run inside the drone, this main process must be killed first, to grant unrestricted access to these resources.

Besides some core discoveries that can be leveraged to integrate the drone with the ADIS system, some flaws were also uncovered. The two main problems are:

- the lack of an easy method to repair/reflash the drone's software in case any change to its system goes wrong
- the drone's Wi-Fi hotspot is not password protected and the drone cannot connect to secure Wi-Fi networks

The first problem was a main concern during all the examination process, because if any mistakes were made, like unknowingly adding an error to a script needed for the drone initialization process, it could lead to a drone that would not create its own Wi-Fi hotspot or finalize the boot process, making its shell inaccessible by telnet. The only way to fix the drone would be through its serial port. But this requires a specialized converter and the knowledge to connect correctly to the drone's serial port. This problem can easily originate many "bricked" drones when the system is being used by many users, that will lack the knowledge about the ADIS system operation to fix this issue.

The second problem is a major security concern. Although the ADIS system is not going to be installed in a public environment, it needs to be designed to work properly in a laboratory that is attended by a large number of people doing different tasks, many of them requiring Wi-Fi networks and internet access. Unavoidably, an open Wi-Fi access would attract unwanted users, leading to extra traffic through the network that could impact the ADIS system performance. To avoid this, extra security measures are required.

Drone Implementation

The AR.Drone 2.0 examination revealed a drone with a relatively open Linux based operating system, that has its strengths and shortcomings, but that can be fully integrated into the ADIS system. Some issues needed to be resolved like the lack of security and the high repairability difficulty.

The first issue tackled was the lack of security. As mentioned before, the drone comes configured to create its own open Wi-Fi hotspot, but it can be reconfigured to connect to an external network by editing its "config.ini" file. Unfortunately, during testing, the drone was not able to connect to password protected networks. The Wi-Fi networks tested used WPA2 security since it is the current standard security algorithm for personal network protection. There was no need to use more secure protocols like WPA-EAP, since it would only introduce more complexity and the added security is not necessary.

This inability to connect to secure WPA2 networks comes from the complete lack of support from the installed operating system and not from a software error or lack of capable hardware. In a traditional

Linux operating system, the process to connect to a WPA2 secured network involves the creation of a master key (passphrase), by combining the ESSID and a pre-shared key (commonly known as the Wi-Fi password). This process involves three binaries called: *wpa_cli*, *wpa_passphrase*, and *wpa_supplicant*. These three binaries are missing from the installed OS. The solution is to compile these files from source, using the correct compiler for the AR.Dorne 2.0 processor, or use pre-compiled versions from online repositories.

The reparability problem required a different approach to how the drone was being used. This problem resides in the fact that the drone has no easy method to repair eventual mistakes that affect its Wi-Fi connection. This is a problem because if the user wants to develop a navigation algorithm that runs inside the drone, and not in an external computer, it is going to need to access the drone's internal memory, exposing internal critical files. These files are not protected by a permission system. Most Linux based operating systems have file system permissions set to every file, to grant or prohibit access depending on which user is logged in. But because the telnet connection established with the drone grants root (super-user) access to the root file system, every file is readable and writable by default. The only interface method available if the drone's Wi-Fi capabilities are impaired is the bottom serial port, which is not trivial to connect and use. One solution would be to change the telnet connection properties not to provide super-user access by default. But to avoid deep changes to the main operating system, another solution was developed.

In a very simplistic manner, a Linux based operating system can be seen as modular and composed of two distinct components: the applications and the kernel. The applications are all high-level processes that execute any necessary task, while the kernel is the software responsible for low-level operations like connecting applications to the hardware. Ideally, the integration process should keep these modules as close to their factory form as possible, and provide a secure separation between them and the ADIS user, while keeping critical tasks like kernel specific tasks available to custom developed algorithms. To achieve this separation, a special mechanism was used called *chroot*. A *chroot* is an operation that changes the root directory of the current running process and its children. Meaning that when the *chroot* system call is invoked the current process and any other process that it spawns is contained inside a virtual environment known as chroot jail. This method can effectively isolate processes while maintaining their ability to access kernel features like controlling hardware. Figure 30 shows how the *chroot* mechanism was used.

First, the drone boots normally executing its initialization sequence and start all of its normal services and applications. When the boot process is complete the *chroot* command is invoked and from that point forward, every spawned process is confined to the *chroot* jail, and isolated from the rest of the system, while retaining the capability to use the drone's hardware, since the kernel is still accessible. The chroot jail directory, however, is not contained inside the drone's internal memory. It is rather located in a purposely created root file system contained inside a USB flash drive. This method completely separates the drone's original root file system from the ADIS system integration files and completely isolates the user from the original operating system processes.

The use of this *chroot* method also keeps the necessary changes to the drone to a minimum. It

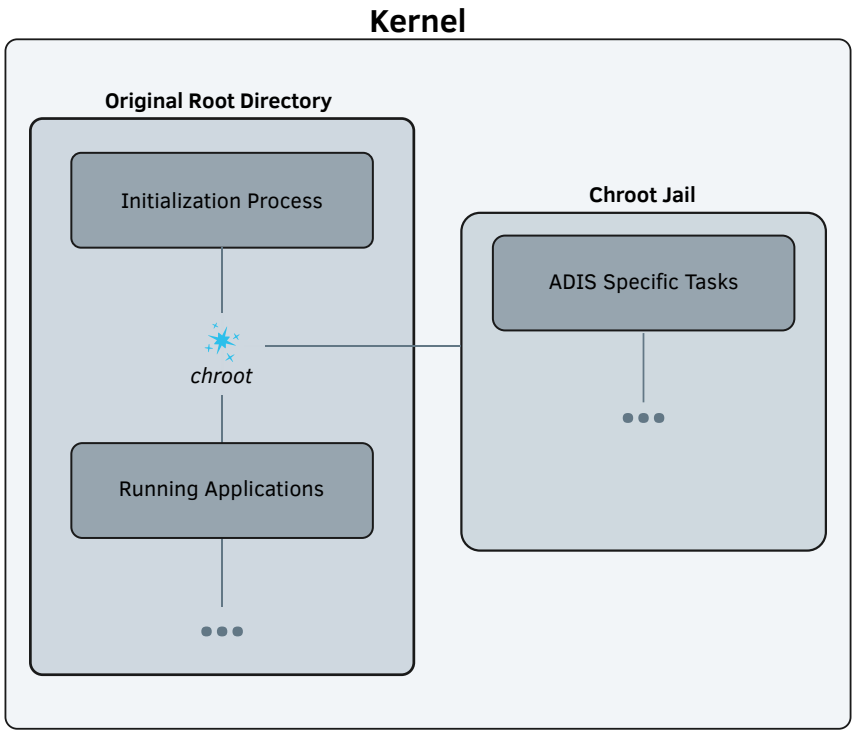


Figure 30: Chroot Operation Diagram

only requires the addition of one line to an original file and the creation of another file. The new line is added into the “rcS” under “/etc/init.d/” to run the new file named “start_USB_OS”, located in the same directory. This new file is a shell script that verifies if a USB flash drive is inserted into the drone’s USB port. If this is the case, mounts the flash drive and executes an initialization file located inside. This initialization file is where the actual *chroot* invocation is done. If no flash drive is detected, no action is taken, living the drone in its original factory state.

The USB flash drive as mentioned before contains a root Linux file system that will run inside the drone, once the chroot jail is created. For this, the USB flash drive must be prepared accordingly. It was decided that the USB flash drive should contain two distinct partitions inside, the USER partition that must be accessible by any operating system and the SYSTEM partition that contains the Linux root file system.To make the USER partition accessible using any operating system, it was formatted using the fat32 file system. This file system has a file size limited to 4GB, but it is natively supported by most current operating systems. For the SYSTEM partition ,the ext4 file system was chosen, which is the current default file system for Linux based distributions. To hold these partitions a “msdos” type partition table was used. Once again, it’s a widely supported type of partition table which will guarantee good operating system compatibility. The actual location in memory of these two partitions is illustrated in figure 31.

The USER partition is located in memory first to avoid issues when an operating system that is not compatible with the EXT4 file system tries to access the flash drive. The SYSTEM partition contains a lean Linux root file system based once again in the Debian distribution. This distribution is the same used in the remote cameras processing unit and it was chosen to keep consistency. Only the root file system is required and not the entire distribution with all of its components since these other components are all

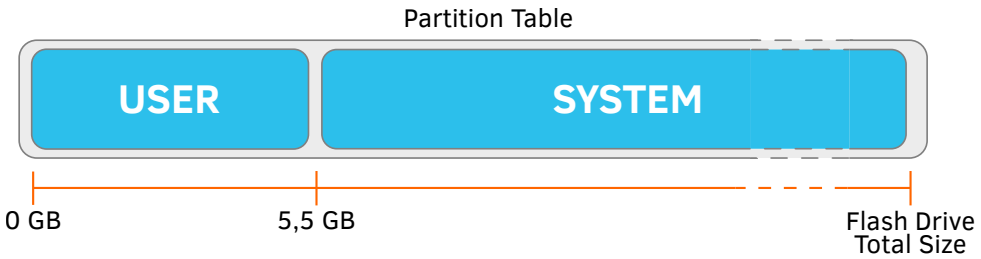


Figure 31: USB flash drive partition mapping

provided by the factory operating system. The flash drive root file system must be compatible with the drone's hardware, meaning that it must be compiled to support the ARM v7 architecture from the onboard CPU. The process used to create such root file system is commonly called "cross-debootstrapping". As mentioned in the Debian online documentation regarding this method[43] "Debootstrap can be used to install Debian in a system without using an installation disk but can also be used to run a different Debian flavor in a chroot environment. This way you can create a full (minimal) Debian installation which can be used for testing purposes". The "cross-debootstrapping" is when the debootstrap method is used to create a root file system for a machine using a machine with a different architecture. In this case, an x86-64 computer running Linux was used to create a rootfs for an ARM v7 machine.

The debootstrap process involves four different steps:

1. Download the necessary .deb packages from a repository.
2. Unpack them into the target directory.
3. Chroot into the target directory.
4. Run the installation and configuration scripts from each package, finishing the setup.

The steps 3 and 4 mentioned above need to be executed inside the target machine because the installation and configuration scripts for each package are architecture dependent. Unfortunately, the last step is very computationally intensive, and it is slow when running using the drone's hardware, to the point where the drone's battery life ends before the fourth step has time to finish. To solve this, the third and fourth steps were executed inside the arm64 computer but using a processor emulator to emulate the ARM v7 architecture of the drone. The emulator used is named "QEMU", and according to the projects home page[44], is a "generic and open source machine emulator and virtualizer." Running this emulator the fourth is still a long process. However, in the end, a minimal root file system was created in the USB flash drive that provides a completely separated environment inside the drone which the user can access without changing the factory operating system by mistake. There are still some modifications required to finalize the root file system. Starting by the initialization file that actually invokes *chroot*, when the drone starts and a flash drive is detected. This initialization file runs the following task in order:

1. Mounts the flash drive temporary directory "/'flash drive'/tmp" in RAM.
2. Binds the original operating system relevant directories to their correspondents inside the flash drive's root file system

3. Mounts the USR partition

4. Connects to the ADIS system Wi-Fi network

5. Invokes the chroot system call to the SYSTEM partition

The first step creates a directory used by the Linux operating system to store temporary files. This is necessary for the processes running inside the chroot jail. Then it binds required directories to allow the process spawned inside chroot jail to run correctly. The bind action executes a bind mount onto the file system. A bind mount is an alternative view of the directory tree. Classically, a mount creates a new view of a storage device as a directory tree. A bind mount, instead, takes an existing directory tree and replicates it under a different point. Meaning that the existing directory is accessible through two different points and that any change made to one point is visible in the other. The required directories are “/dev”, “/proc”, “/sys” and “/dev/pts” , that are binded to “/’flash drive’/dev”, “/’flash drive’/proc”, “/’flash drive’/sys” and “/’flash drive’/dev/pts”. The “/dev” directory is where special or device files are located. The “/proc” directory is referred to as a process information pseudo-file system. It does not contain any ’real’ files but rather runtime information for running processes like system memory, devices mounted, hardware configuration, etc. The “/sys” in essence contains kernel related information about the system and its components, mostly regarding attached or installed hardware. The files under the “/dev/pts” directory are console devices, which provide command line access to the system. In the third step, the script mounts a specific user directory that is going to be explained in detail later. The fourth step is self-explanatory: the drone connects to the ADIS system WPA2 protected network. The network information is stored in a file “’flash drive’/data/config.ini” that can easily be edited by the user. This file contains the network SSID and password as well as the drone’s network IP address. This file as a simple syntax and can be edited by the user. Bellow is a sample that shows how this configuration file looks like: This file name and

```
IP:             192.168.1.3
ESSID:          ADIS_Wi-Fi_network
PASSWORD:       this_is_the_password
```

Figure 32: Example of the config.ini file for the USB flash drive

location were based in the “config.ini” file that comes with the drone, which contains information relative to the drone’s state and configuration parameters. The fifth and last step invokes the *chroot* system call. As mentioned before, this changes the root directory for the calling process and for every process that spawns from it. After adding this last initialization script into the SYSTEM partition with the name “init”, another “init” file is added to the USER partition. This last initialization file is called by the chroot caller process and it runs inside the chroot jail. It executes two different tasks:

1. Creates a new telnet server.

2. Sets the gateway IP address and DNS.

The first task creates a new telnet server, but this time it’s running inside the chroot jail. Connecting to this telnet server opens a Linux terminal inside the chroot jail with the newly created root file sys-

tem. Inside this terminal session, more tools are available compared to the original telnet session. Including the package manager “Advanced Package Tool” or “apt”. This tool allows the user to install packages directly from the Debian repositories. The “dpkg” tool also allows to install other third party “.deb” packages not provided by the Debian repositories. Some interesting packages available are for example the ROS meta-operating system, which consists of a set of libraries and tools that are used for building robot applications. The second task sets the gateway IP address of the network and the DNS. These are both the router IP address from which the drone is able to connect to the internet. After adding this second initialization file, the Debian repositories sources list must be added to the “’chroot jail’/etc/apt/sources.list”. The sources list indicate which Debian repositories the “Advanced Package Tool” should access to retrieve packages and updates.

With this, the root file system is finalized. This manual process of creating the rootfs in a USB flash drive and modify the drone to use this new *rootfs* can be considered too long and complex. And it needs to be executed every time a drone’s USB flash drive needs to be repaired or a new drone is added to the ADIS system. In order to reduce the root file system creation time, an automated process was created.

The most time-consuming task is the fourth step of “cross-debootstrapping”, where the installation and configuration scripts from each package are executed. This process can be bypassed using a raw disk image of the root file system. This raw image is a “.img” file that contains all of the binary data from the SYSTEM partition excluding empty space. This raw image can then be copied to the SYSTEM partition of any flash drive. This change reduces the root file system creation process from two hours to around three minutes, depending on the computer and flash drive performances. To further decrease the root file system creation and drone setup time, a bash script was developed that completely automates both of these processes. This script is capable of preparing and formatting both partitions to a pre-selected flash drive, creating the root file system in the SYSTEM partition either by burning the raw binary image previously created or by running the slow “cross-debootstrapping” process automatically and also places all necessary files in their correct directories. It also has the capability to verify if a drone has already been altered to operate in the ADIS system. All of these operations are done automatically. Only two inputs from the user are required, one to choose from the menu which task he desires to execute, and another to select the correct USB flash drive to format and install the file system. If the user desires to adapt a brand new drone, he must connect to the drone’s Wi-Fi hotspot before proceeding with the integration process.

The only disadvantage of this approach is that the “cross-debootstrapping” process used can only be done under Linux, the script is not compatible with any other operating system. Since this problem is not easily solvable, other native Linux functions were used to facilitate the script development, this further decreased the compatibility with other operating systems other than Linux based ones. This incompatibility is a problem because the ADIS localization software was developed to run on Windows. To avoid installing other operating systems into the main computer the following workaround was used.

Once again, for consistency, a minimal Debian installation was purposefully created to run on a virtual machine on the main computer. However, this installation has only the single purpose of running the automated drone setup script. Leveraging once again the system resources, a specialized service was

created to run the script when the virtual machine boots up. When the user decides to exit, the script itself turns the virtual machine off, closing its window. Using the free VirtualBox[45] program for the virtualization, it is possible to create a shortcut that automatically boots-up a defined virtual machine. With this, the user clicks the shortcut icon and the script executes under a Linux environment almost immediately.

Up until this point all necessary steps to modify and integrate the AR.Drone 2.0 were explored, as well as, intermediate steps and solutions. Next, to clarify the final implementation process, a summary of all relevant steps is presented.

Integration Process Out of the box, the drone creates its own Wi-Fi hotspot. When connected to this hotspot there is available one ftp server and one telnet server. Connecting to the telnet server provides access to the drone's original root file system. To integrating the drone in the ADIS system, new USB flash drive containing a new file system must be set up, and the necessary changes need to be applied to the drone. To prepare the flash drive a specialized integration script is executed, by double-clicking the correct shortcut. After clicking, a terminal window opens up running the script in a virtual machine. This script provides a menu from which the user selects either to modify the drone or to create the USB flash drive.

If create a new USB flash drive is selected, the script asks which storage device to prepare. After the drive is selected, the script prepares the flash drive by wiping it and creating two partitions called SYSTEM and USER. The SYSTEM partition can only be accessed through an operating system compatible with the ext4 file system. The new root file system raw image is burned to this partition. If the user desires to use the "cross-debootstrapping", the script also provides this option with all steps automated. The USER partition is a fat32 partition that can be accessed through any computer. The script executes some finalization steps and notifies the user that the flash drive is ready to be used in the drone with the message "DONE!". Throughout the script's execution, messages are printed in the command line that informs the user about what task is being done, and if any error occurred.

To modify the drone, the script first asks the user to connect to the drone's Wi-Fi hotspot. After establishing the connection, verifies if the drone is in its factory state, or not. If the drone is in its factory state, the script will proceed to create the "start_USB_OS" file and add a line to the "rcS" file under "/etc/init.d/". After these changes, the drone is ready to be used with the ADIS system.

New startup sequence The modified drone has a new startup sequence, fully described in figure 33. When the drone is connected to a power source, the original initialization process is executed. This process was already described before and when it finalizes, the drone is in its factory intended state, with its own Wi-Fi hotspot turned on. Then the extra line added to the "rcS" executes the "start_USB_OS" file. This file checks if the USB flash drive is plugged in. If this verification returns false, the file execution ends and the drone is left untouched in its factory state. However, if the verification returns true, the drone mounts the flash drive to the "/mnt/" directory, binds required directories, and connects to the ADIS Wi-Fi network with a predetermined IP address in the "config.ini" file from the new file system.

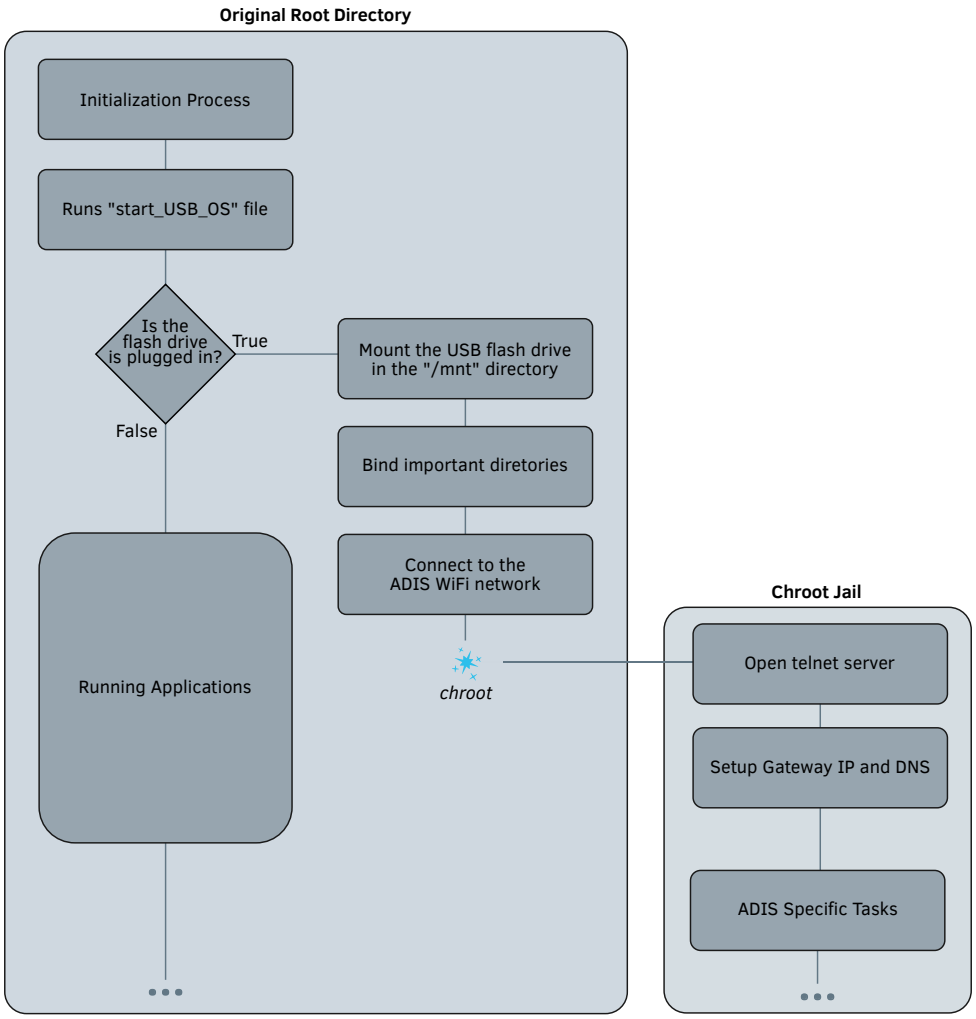


Figure 33: Integrated AR.Drone 2.0 startup sequence diagram

After these three steps, the drone invokes the *chroot* system call imprisoning the following processes to the chroot jail. The chroot jail confines all children processes to the new file system created previously inside the USB flash drive. Finally, the drone opens a new telnet server that provides access exclusively inside the chroot jail and sets up the gateway IP address and DNS server, allowing the drone to connect to the internet. After these tasks complete, the AR.Drone 2.0 is ready to be used within the ADIS system. The user knows that the startup sequence is finished when the motor LED's color turns solid green.

Drone's Interfaces To access the integrated drone, multiple interfaces are available. The original interfaces are still available, meaning that the ftp server, and telnet server that connect to the original file system with superuser rights, are up. However, when the drone is working with the new purposely created file system, a new telnet access is available directly to the chroot jail. The user should choose to use this last telnet server instead. With it, a Debian base system is available with all of its resources and tools available. As mentioned before external ".deb" packages can be installed to augment the drone's capabilities, as well as a vast set of officially supported packages. To load files directly to the drone, like navigation controller binaries, the USB flash drive can be plugged in directly to any computer. Because the USER partition is formatted using fat32, compatibility with most operating systems is assured.

Flight Controller Implementations

Although the drone's integration is at this point complete, there is still no direct way for controlling the drone. For this to be possible, a framework or toolset should be provided along with the rest of the developed software.

Besides the already mentioned SDK the AR.Drone 2.0 drone has multiple API's available, written for multiple programming languages. Three different alternatives are going to be briefly mentioned in this thesis. The first two are both API's developed two work with MatLab within Simulink.

The AR Drone Simulink Development-Kit V1.1[47], is a toolbox of Simulink blocks, developed in the context of the 2013 MathWorks Summer Research Internship project. This toolbox is compromised by two main blocks, one that handles all communications with drone via Wi-Fi, and another block that mimics the drone dynamics for simulation purposes. Inside the Wi-Fi control block, other blocks handle all outgoing and incoming communications as well as all protocols for coding and decoding tasks. There is also included a set of examples on how to use this toolbox, with simplistic controllers for hovering and way-point tracking maneuvers. This toolbox is simple to use, although it has some limitations.

This toolbox for controlling the drone uses the protocol described by the AR.Drone Developer Guide. This protocol communicates with the stabilization and control process running inside the drone called "program.elf". In other words, this toolbox sends commands which are interpreted by the "program.elf" process, and not commands that change the propellers actuation directly. This means that any controller developed using this toolbox is going to work around the factory implemented controller. Effectively, any control loop will be an exterior control loop with an unknown control loop running inside. Depending on the final objective of the controller, this can be an advantage. For example, if the objective is to build a navigation algorithm to maneuver the drone, and not to control its stability, to have the drone attitude already stabilized is an advantage.

The case where designing controllers that are able to actuate directly the drone's propellers also exist. The user might want to create his own custom stabilization algorithms and test them, to achieve results that are not possible with the factory provided stabilization controller. To provide such functionality, the Simulink automatic code generation can be used. Simulink provides support for embedded hardware and enables the creation of software solutions that can later be deployed to the specific target hardware, allowing them to run natively. This requires a set of files and configurations that define the final target hardware. Since the AR.Drone 2.0 is supported by Mathworks, all of the necessary files and configurations are already provided[48]. These target files allow to automatically generate code for the AR.Drone 2.0 directly from a Simulink model, use the "external mode" to monitor and tune the deployed algorithms in real-time, and support for camera interfacing for onboard image processing is also available. Basic template models and controllers are also provided that can be used as a base for new projects.

The third and last alternative mentioned in this thesis is the PS-Drone API[49]. This is a python based API, that provides a straight forward set of tools as well as an easy to follow set of examples and documentation. This API is ideal for any user that does not want to rely on Simulink to develop his project. It is free and open-source which means that it can be used and changed freely. It is not

a solution to deploy native code like the previous one, but rather issue commands that are interpreted by the “program.elf” process like the AR Drone Simulink Development-Kit. The main difference when comparing it to the AR Drone Simulink Development-Kit is video support. With the PS-Drone API it is possible to decode and use the video streamed by the drone in real-time. It uses resources from OpenCV, which allow the implementation of advanced image processing techniques. This API, besides video support, also provides the ability to calibrate the drone, move it, configure and decode NavData information, and even configure the drone remotely.

It was previously mentioned that it is not possible to actuate directly the drone propellers using the factory provided controller software. However, the PS-Drone API implements an undocumented pwm instruction that can changes the motor actuation directly. But because the drone receives commands only 30 times a second with its factory controller, it may not be sufficient to stabilize the drone by controlling directly its motors rotation rate. There are also implemented commands that execute pre-choreographed animations. These animations are defined in the drone controller and can make the drone spin, execute small “dances” from side to side, or even do a back-flip.

5 Communications

This chapter focuses on the network related details of the ADIS system. The ADIS system communication relies almost on its entirety in a Wi-Fi network. Every component of the system is connected to this network and uses standard internet protocols to communicate with other components within the system. This provides the ADIS system with flexibility, allowing to freely arrange its remote cameras, computers, and vehicles.

5.1 Protocol Descriptions

To send information from one computer to another, the system relies on two different protocols, TCP and UDP. These two protocols are part of the Internet Protocol Suite, will be mentioned as part of the "Transport Layer" of the ADIS system. They allow all processes to exchange information between them. TCP and UDP are not the only protocols used. Others are used to define how the data sent through the "Transport Layer" is formatted.

5.1.1 Remote Camera to Central Server

The messages exchanged between the remote cameras, from chapter 3.4 to the central computer can be classified into two types: commands and location data. Commands are the messages that describe information regarding actions that the remote camera needs to execute, while location data contains information regarding detected trackers locations in pixels. Both types of messages use different transport protocols and encoding formats. These encoding formats refer to the way that the information is arranged by the software before being sent through the Wi-Fi network using the "Transport Layer" defined by TCP and UDP protocols.

Command messages are the ones that tell the remote camera how to act depending on what the localization software needs from it. Because the success of some commands is dependent on the execution of previously issued commands, it is undesired that some commands are lost, not reaching the target. For example, to start streaming the video feed from a remote camera to the localization software, it's necessary that the remote camera isn't running the tracking routine, so a "stop tracking" command must be issued before a "start stream command" can be executed with success. Thus, to guarantee that every issued command reaches its destiny, the TCP transport protocol is used. This protocol by definition guarantees that any packet sent reaches its destination unless the connection between the source and destination devices is broken. This delivery guarantee has the disadvantage of slowing down communications, along with the congestion avoidance algorithms that TCP implements that limit the number of packets sent to the network in order not to overload it. However, because the command messages are not time critical, the TCP protocol can be used.

The messages besides being sent using the TCP protocol, are also encoded using a custom protocol described below. It's important to note that for every message described in the protocol below, either command or ACK, the EOM character is the newline character, $/n$.

Identification Command: This command is used to ask for the remote camera identification information, meaning that the remote camera returns what type of device is using, either a Kinect or an RGB camera.

Data Type	Format		Notes
	Command	ACK	
string	ID	device	The *device* can either be *kinect* or *camera*

Get Resolution Command: This command asks what the camera resolution in pixels is.

Data Type	Format		Notes
	Command	ACK	
string	GET_RESOLUTION	height␣width	The response is composed by the video *height* and *width* seperated by one space

Get Parameters Command: When this command is received, the remote camera returns the camera parameters related to brightness, contrast, saturation, and exposure. The parameters are the minimum, the maximum, the minimum step value and the current value. This command is only available when an RGB camera is connected to the remote processing unit. If a Kinect is connected nothing is returned.

The information is organized in sets of four values for four camera parameters separated by spaces. The values are the minimum, maximum, smaller step and current value for each parameter in this order. The parameters are brightness, contrast, saturation, and exposure in this order.

Data Type	Format		Notes
	Command	ACK	
string	GET_PARAMS	Bm␣BM␣Bs␣Bv␣ Cm␣CM␣Cs␣Cv␣ Sm␣SM␣Ss␣Sv␣ Em␣EM␣Es␣Ev␣	B for brighness values C for contrast values S for saturation values E for exposure values m for minimum M for maximum s for minimum step v for current value

Set Parameters Command: This command sets the current value for a specific video parameter for the remote camera. The available parameters are brightness, contrast, saturation, and exposure. This command is not available when a Kinect is connected to the remote camera. This command returns nothing when is received.

Data Type	Format		Notes
	Command	ACK	
string	SET_PARAMS␣*param*␣value		The *param* parameter in the command can be BRIGHTNESS, CONTRAST, SATURATION and EXPOSURE.

Calibrate Camera Command: This command sets in motion the calibration sequence. Until this command finishes the progress of the calibration process is sent as an ACK. If something goes wrong, the ACK is a CALIBRATE_FAILED message. If the camera calibration is successful, the final ACK is 100. This command can fail if an error occurs during the calibration process, if because the video stream is active, or even if the camera is not initiated by the start camera command.

Data Type	Format		Notes
	Command	ACK	
string	CALIBRATE	CALIBRATE_FAILED or values between 0 and 100	This command as multiple ACKs to indicate the progress of the calibration process

Get Camera Height Command: This command asks for the camera height relative to the ground. It is only available when a Kinect is connected to the remote processing unit. This command fails if executed before running the calibration process, returning GET_HEIGHT_FAILED.

Data Type	Format		Notes
	Command	ACK	
string	GET_HEIGHT	GET_HEIGHT_FAILED or height	height is the vertical related to the ground measured with the Kinect in the calibration process.

Start Camera Command: The start camera command is used before using any task that requires image processing. It allocates the camera device (Kinect or webcam) and initiates it for image acquisition. This command can fail in if an error occurs, if the camera is already initiated, or even if the video stream is already running. If any of the previous conditions occur, the ACK sent is CAM_FAILED.

Data Type	Format		Notes
	Command	ACK	
string	START_CAM␣height␣width	CAM_FAILED or CAM_OK	The height and width are always required, but only take effect when using a webcam.

Start Stream Command: The start stream command starts a real-time video stream. This command may fail if the camera is initiated by the start camera command, if a video stream as already started or if an unexpected error occurs.

Data Type	Format		Notes
	Command	ACK	
string	START_STREAM	STREAM_FAILED or STREAM_OK	The resolution for the video feed is hardcoded and cannot be changed by this command

Stop Camera Command: When this command is sent, the remote camera tries to return the image capture device to its state before being initiated by the start camera command. This command must be sent if, for example, the software wants to start a video stream, but the camera is initiated for image processing tasks. This command can fail only if an unexpected error occurs or if a video stream is running because if the camera is not initiated, the camera state is the same after its initialization, so the command succeeds.

Data Type	Format		Notes
	Command	ACK	
string	STOP_CAM	CAM_STOPPED_FAILED or CAM_STOPPED_OK	If this command fails, the unexpected error may lead the remote camera to an unrecoverable state, which can be fixed by hard rebooting the remote camera

Stop Stream Command: This command stops the video stream. It fails if the camera is initiated or if an unexpected error occurs. If this command is issued when the stream is not running, the command succeeds.

Data Type	Format		Notes
	Command	ACK	
string	STOP_STREAM	STREAM_KILL_FAILED or STREAM_KILLED	If this command fails, the unexpected error may lead the remote camera to an unrecoverable state, which can be fixed by hard rebooting the remote camera

Get Image Command: The get image command, asks for an uncompressed frame captured by the camera. This command is used to transfer an uncompressed image to an external device for posterior

image processing. This command sends the images in chunks of twenty RGB pixels.

Data Type	Format		Notes
	Command	ACK	
string	GET_IMG	GET_IMG_FAILED or partial_image_frame	Each partial_image_frame is composed by the RGB values separated by spaces for each pixel. All of the RGB triples are also separated by spaces.

Reboot Command: This command reboots the remote camera. It is never used by the ADIS system, and it was added for debugging and development purposes.

Data Type	Format		Notes
	Command	ACK	
string	REBOOT		This command does not have an ACK

Shutdown Command: This command shuts down the remote camera. It is never used by the ADIS system, and it was added for debugging and development purposes.

Data Type	Format		Notes
	Command	ACK	
string	SHUTDOWN		This command does not have an ACK

Start Tracking Command: This command starts the marker tracking process. This command fails if the camera is not initiated, if a video stream is running or if an unexpected error occurs. If the command succeeds, a stream of information containing the location information starts flowing to the localization software.

Data Type	Format		Notes
	Command	ACK	
string	START_TRACKING␣port	START_TRACKING_FAILED or location_information	The location_information format is: xcorr␣ycorr␣z␣x␣y The port parameter refers to the UDP port on the client side

The location information sent to the localization server corresponds to the location data messages previously mentioned. These messages are exchanged using the UDP protocol. This protocol, unlike TCP, does not guarantee that the message is delivered, however, this makes the UDP communication much faster than TCP. Because the location data must reach its destination as fast as possible, the UDP

protocol is the best alternative to use. By not guaranteeing the message delivery, the UDP protocol can suffer from interference and loss of information. This problem is explored in more detail in chapter 7.

The information sent depends on what video capture device is connected to the processing unit, however, to avoid different data structures, the same codification is used either if a Kinect is connected or an RBG camera. The xcorr and ycorr parameters refer to the x and y coordinates in pixels, corrected for the drone height relative to the ground z, in case a Kinect is being used. If an RGB camera is connected to the processing unit the message format is maintained, but the xcorr and ycorr parameters are the same as the x and y coordinates. This allows using the same code in the localization software for both devices.

5.1.2 Central Server to Clients

The messages traded between the main computer to other clients refer to the location data that the localization software sends to an external computer running a control algorithm. This information can be compared to the location information shared from one remote camera to the central server, in the sense that it needs to reach its destination as fast as possible. So, UDP was used. The message sent by the localization software contains information other than location data. It also has information regarding tracking identification and capture time. Once again, the EOM character is the newline character, $/n$.

Data Type	Format		Notes
	Command	ACK	
string	TIME,ID_1,X_1,Y_1,Z_1,..,ID_n,X_n,Y_n,Z_n|		This command does not have an ACK

The message format presented above describes the way that the information is sent to a client computer. Every message is started by a timestamp, relative to the simulation duration in minutes measured by the server computer when the message is sent and can be used for synchronization purposes. This time stamp is followed by all necessary information to describe the detected points. The ID is a unique identification integer selected by the tracker algorithm that identifies the same marker throughout the whole experiment. The X, Y and Z parameters are the coordinates of the marker in the user-defined inertial frame. If an RGB camera is being used instead of a Kinect, the Z coordinate is always zero.

5.2 Network Topology

When the ADIS system architectures were addressed in chapter 2.3, it was assumed that every system component communicates directly to any other component of the system. This would form an ad-hoc network. This type of network is decentralized and every system component has to manage its won connections to other network entities, in a peer-to-peer mesh style topology. The diagram from figure 34 illustrates this concept. Every network node has its own connection, to any other node that it desires to communicate with. The messages can be sent directly from the original node to the destination node or

Figure 34: Peer-to-peer mesh network topology diagram

can be passed along the by other nodes until reaches its destination, in case the mesh network is not fully connected. A fully connected mesh network means that any node from the network is connected to other nodes, by one direct connection.

However, the ADIS system does not use a mesh peer-to-peer network topology, because it's message delivery speed is inherently dependent on all of its nodes network speeds. And they are complex networks to manage due to the big number of connections. The diagrams in chapter 2.3, are drawn in a simplified way to better describe how the communications inside the network are organized. To illustrate that a drone communicates with the user's computer, a direct connection is drawn. In reality, the ADIS system uses a star topology, like the one from figure 35, in which every network node is connected to a central hub with a point to point connection. In this type of centralized network, every message is

Figure 35: Star network topology diagram

sent to the central node and then routed to its final destination. This central node is commonly known as a router. Start networks are the most common network typologies used in home internet networks. Because every message is relayed in the router to its final destination, the networks speed depends mainly on the router network performance. This topology also requires a low number of connections to be managed by the router, keeping network complexity to a minimum.

5.3 Network Scanning

During the ADIS system development arose the necessity to build a way to scan and find known devices connected to the Wi-Fi network. This feature is important for to reasons:

- The localization software needs to be able to find all of the connect remote cameras.
- A tool to find connected vehicles and set up Simulink template files with all of the correct network configurations was created.

Basically, this feature serves to further automate the basic workflow of the ADIS system. The user could be prompted with a window to insert the remote camera current IP address or to set up his Simulink project, but this would require a manual network scan or knowledge on how to access the router interface to identify the correct IP address between all of the other attributed addresses. To avoid this a network scanning tool was included.

The network scan method used relies on the Address Resolution Protocol or ARP. The ARP protocol was defined by the RFC 826[50] in 1982, and it is used to map network addresses (IP addresses) to psychical address like a MAC address. A MAC address is a hardware identifier that is attached to a network connected device. This address is defined by the network card manufacturer, and it is different for any device.

Without going into to much detail the ARP protocol uses the following workflow: when a computer named computer 1 wants to communicate with another computer named computer 2, inside a local Ethernet network, it requires the MAC address for computer 2. For example, the computer 1 knows that the IP address from computer 2 is 192.168.1.25, and wants to send a message. To send a message through a local network the MAC address from computer 2 is required. The computer 1 first looks up to cached ARP table and sees if it already has the MAC address for the IP address 192.168.1.25. If this is the case, computer 1 can send a message to computer 2. However, if the MAC address for 192.168.1.25 is not cached, the computer 1 sends a broadcast ARP message that is accepted by all computers in the network, asking who is the computer with the address 192.168.1.25. Upon receiving the broadcast message, the computer 2 answers with its IP and MAC address. This answer is then stored in the ARP cache by computer 1, which is now able to send the message to computer 2.

The example above gives some leads on how the network scan process is executed. First, a *ping* command is sent across a predetermined interval of IP addresses. This triggers the process exemplified above, that finishes with an ARP cache filled with IP addresses and the respective MAC addresses from devices connected to the network. Then these cached MAC addresses are compared with known MAC addresses from ADIS system components or vehicles, that were collected in advance. If any known MAC address is found, the ARP cache provides the correspondent IP address. This simple process allows the localization system and other software tools to find any desired device connected to the Wi-Fi local network. However, two files are required for this process to succeed. One that stores the MAC address of ADIS network devices, and another that indicates which IP addresses to scan.

The first file is called “Devices_List.txt”. This file is composed of three columns: one that indicates the device name, another with the MAC address and another that indicates what type of device the device

is. Bellow an example of how this file looks. The “Camera_wireless” and “Camera_wired” indicate if the

```
Camera_1  11-11-11-aa-aa-aa  Camera_wireless
Camera_1  22-22-22-bb-bb-bb  Camera_wired
Drone_1   33-33-33-cc-cc-cc  Drone
Drone_2   44-44-44-dd-dd-dd  Drone
```

Figure 36: Example of the Device_List.txt

remote camera is connected to the network via Wi-Fi or Ethernet cable.

The second file required is called “NetworkMap.txt”. This file contains a description of how the ADIS system components are organized within the network. This organization is not required for the scan process to work, but scanning the discrete intervals of the network is more time effective than to scan the complete network. A predetermined network map was defined that describes the interval of IP addresses that the ADIS network connected devices must have. The correct interval is selected according to the device type. Bellow is an example of a network map file. Two columns compose this file.

```
Drone_setup     1-1
Main_PC         2-2
Vehichles       3-19
Remote_Cameras  20-25
Gateway         255-255
```

Figure 37: Example of the Device_List.txt

The first column describes what type of devices are within the IP addresses interval described in column two. The second column does not contain the full IP addresses that describe the scan interval, but rather the last triple of the IP address. For simplicity, it is assumed that all of the IP address starts with the same three triples, “192.168.1”. The first interval contains only one IP address, the 192.168.1.1. This IP address is only used when a drone needs to be repaired or configured. The IP address 192.168.1.2 is reserved for the main PC that runs the localization software. The IP addresses between 192.168.1.3 and 192.168.1.19 are reserved for any vehicle inside the ADIS system, either cars or drones. The 192.168.1.20 to 192.168.1.25 address are allocated exclusively for remote cameras. This interval is the one that the localization software scans every time it starts. This scan is compromised of 5 IP address instead of 255 if the whole 192.168.1.x IP address range. This decreases greatly the network scan process, as mentioned before. The last IP address described in the file is the 192.168.1.255 that represents the network gateway node. This gateway is the local IP from which any device can communicate to the rest of the Internet, and represents the router’s IP address.

This file can be edited to reorganize the network when needed, granted that every device must then be reconfigured manually to obey to a new network map.

6 Software Suite and Tooling

Up to this point, most of the tools that compose the ADIS system were already mentioned. Due to being required to explain what the system does and how it does it, functional parts like algorithms and mathematical relations were already addressed. Nonetheless, some features like the GUI (graphical user interface) and basic workflow, are still lacking explanation. This chapter is entirely dedicated to briefly introduce these features.

One of the main goals of this thesis was to provide a good set of tools to use. The first approach to this could be to create a simple script with no interface, that handled all necessary tasks by himself only requiring its initialization. Unfortunately, due to the high complexity of the localization system, which requires calibration routines, real-time video feeds for camera setup procedures and more, a bare-bones script would quickly become to complex. To overcome this, extra work was done to build a simple GUI, that provides the necessary functionality while maintaining being easy to use.

ADIS Localization Software GUI

The localization system is the main piece of software developed. All of the system functionality, except vehicle communications, hinges in the localization system software and is managed by it.

The graphical user interface for the localization system is encapsulated in a single window. This window is divided into two zones. The first one is a group of two tabs and the other one is a toolbar. The two tabs are titled “CAMERA FEED” and “TRACKING”.

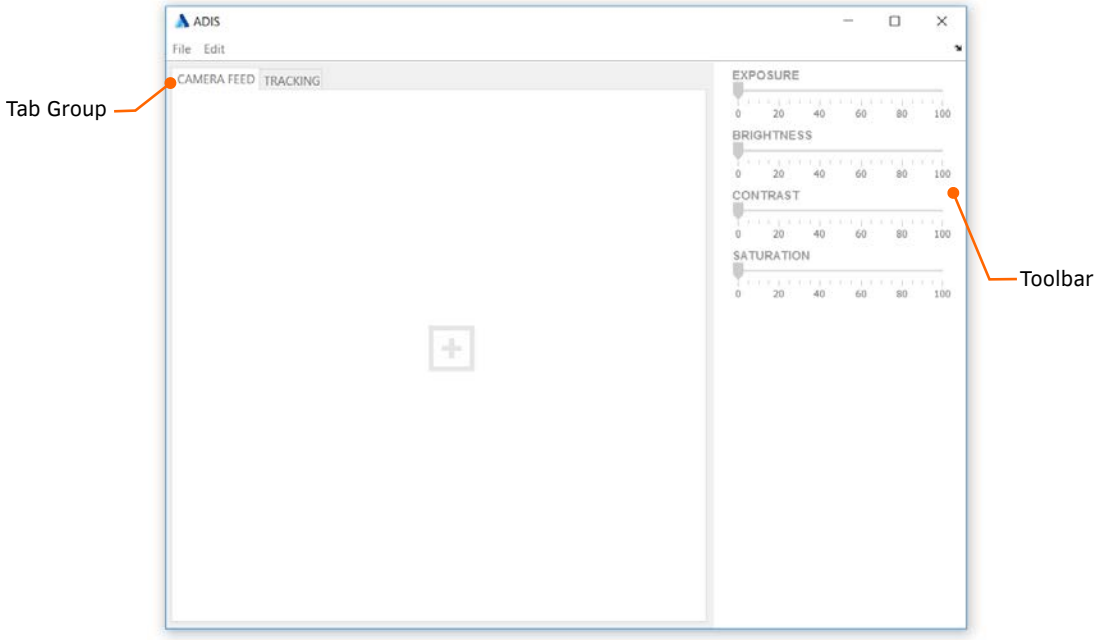


Figure 38: Localization system software main window

The first tab handles previewing, in real-time, the connected camera video feeds as well as provide tools to manage them. The number of video feeds is software limited to six. This is just a software imposed precaution, to prevent unexpected interface behavior due to, too many video feeds being added.

However, the software can be altered to support more video feeds. From this first tab, any compatible camera can be controlled in real time by changing its parameters, using the toolbar. Due to driver limitations, the Kinect image parameters are not altered consistently, so the toolbar access is limited when a remote camera is connected to a Kinect. The video feeds can be freely resized. The plus sign in the middle of the tab adds a new camera to the system. Clicking it opens a smaller window in which the correct camera can be installed and added to further use. The “IMAGE CAPTURE INTERFACE” drop-

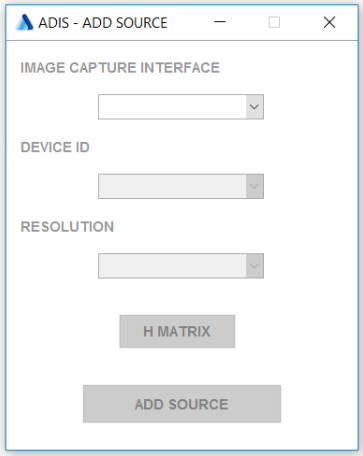


Figure 39: Add new camera window

down menu, selects the interface used to communicate with the cameras. On Windows, to use USB connect cameras, the interface is called “winvideo” and is fully provided by Matlab. To use the remote cameras the “Remote CAM” option must be selected. The software only allows the use of one video interface at a time, meaning that it is impossible to use remote cameras, and USB connected cameras at the same time. The “DEVICE ID” drop-down menu is used to select the correct camera to use. If no camera is available, a warning message pops up when selecting the image interface in the previous step. The “H MATRIX” button opens a new dialogue window. This window provides the option to load a homography matrix previously created, or to create a new one with the previously mentioned calibration process. This calibration process provides clear instructions with images indicating all necessary steps. When every option is set correctly, the “ADD SOURCE” button is enabled, and the user can add a new camera by clicking on it.

The second tab is composed of all the necessary elements to monitor the tracking process. The most prominent element of this tab is the “3D Live Graph” that draws in real-time all trackers detected by the system. This graph can be edited using the “3D Live Graph Tools”. This toolbar provides three different functions: rotate, zoom and pan. For performance reasons, these tools are disabled while the tracking process is running, so, any desired changes to the “3D Live Graph” must be done before initiating the tracking process. The three indicators on the left side display useful information regarding the tracking process. Frames per second and the number of detected markers indicators are useful to monitor the localization system performance. The two switches are used to turn on or off the UDP data stream to the user’s computer, or the “3D Live Graph”. This last option is useful to increase system

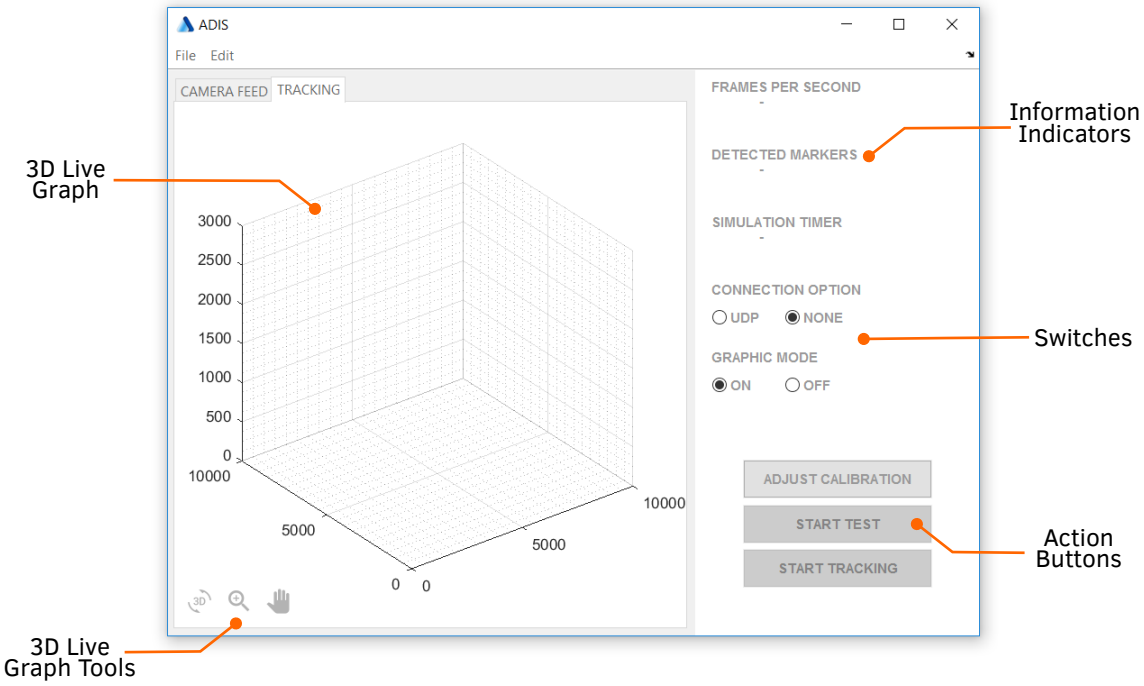


Figure 40: Localization Software TRACKING tab

performance in case a big number of trackers are being detected, and the frame rate drops considerably. Finally, there are three buttons on the bottom right corner. The first one is used to adjust the calibration of the background subtraction used for marker detection with USB connected cameras. This button is necessary because, although the system is installed indoors, the lighting condition still change during the day, which requires reacquisition of the camera backgrounds.

Simulink Project Creator

This tools can be described as an automated Simulink project creator. The need for this tool aids users to configure the Simulink blocks to communicate with the drone, by creating a blank project only with the required and pre-configure blocks to communicate with the drone.

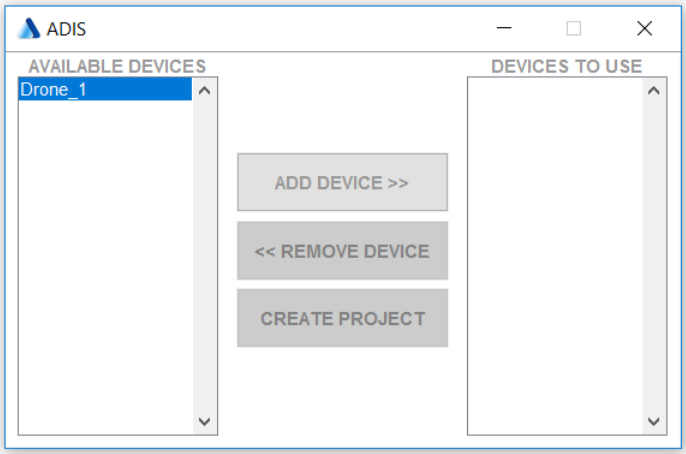


Figure 41: Tool for creating Drone Simulink projects

This tool has a minimal interface, as shown in figure 41. It has three columns, two of them are a list of vehicles and another is a set of buttons. The first column shows devices available in the network, and the third column shows the devices that the user wants to use, while the second column is a set of buttons which allow selection actions.

At launch, the program scans the network, according to the network map file as described in section 5.3, and displays all of the connected vehicles. Then the user can select which vehicles wants to use for his project, by clicking the “ADD DEVICE” button. Devices can also be deselected from the project clicking the “REMOVE DEVICE”. Finally, after clicking the “CREATE PROJECT” button the tool creates a Simulink project with all of the necessary blocks already configured to work with each vehicle. Every IP address is set to the right block according to the detected drone. The user can directly use this project to build his controller without needing any further modifications. Currently, this tool only supports the AR.Drone 2.0 and the Simulink blocks from the AR Drone Simulink Development-Kit V1.1[47].

7 Performance Analysis

Until now, all of the system components were presented and their individual operation was explained in detail. How they connect to each other and work in unison to achieve a cohesive system was addressed as well. This chapter evaluates the complete system performance and a comprehensive evaluation of its measurements. This performance analysis is mainly focused on the ADIS localization system since it is the main component of the complete ADIS system.

To begging the performance analysis a baseline for success must be set. Every image capture device, either the USB cameras or the Kinect works with a maximum frame rate of 30 frames per second. As mentioned before the drone interprets commands at a rate of 30 Hz. So ideally the ADIS system should also work at a frequency of 30 Hz. However, due to processing times, this frequency may not always be fulfilled. So to ensure that the working frequency is always enough, a minimum sample rate of 20 Hz was set. Below this limit, the system is considered unfit to control a drone or a grounded vehicle. The drones dimensions are approximately 38x29x12 centimeters. The smallest vehicle dimension is then the drone height at 12 centimeters. Empirically this was defined as the maximum absolute position error for the localization system.

Figure 42 shows the setup used to evaluate the ADIS localization system performance. The setup

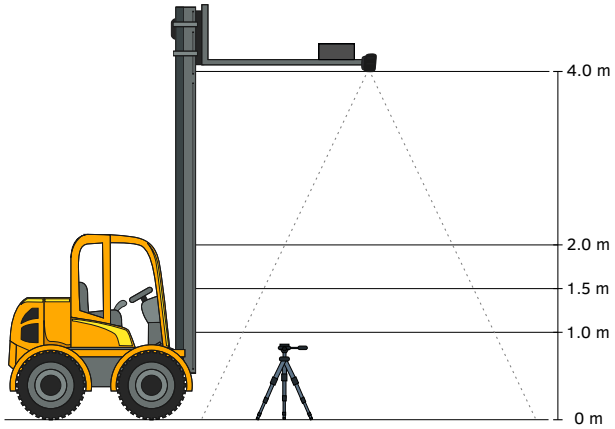


Figure 42: Setup for the localization system tests

consisted of one forklift with a remote camera mounted on the tip of its forks. The forks were elevated until the remote camera was at 4 meters from the ground. A tripod was used to hold a 5 centimeters marker at a fixed height. A grid of 40x40cm squares was drawn on the ground occupying the entire field of view of the camera. The intersection points of the grid were used as the locations for all of the measurements during the tests. For each point, 30 consecutive measures were taken, which corresponds approximately to 1 second of acquisition. From these 30 measurements, the mode was assumed as the measured value. To analyze not just the system accuracy but the data quality as well, other statistical quantities were computed, like the average and the standard deviation.

To test the USB cameras accuracy just one test was needed with the marker placed on the ground, making its height 0 meters. On the other hand, to test the Kinect accuracy, four tests were made with different heights, 0 meters, 1 meter, 1.5 meters, and 2 meters. Tests with a height above 2 meters

were not made, because the area viewed by the camera starts to get significantly smaller. At 2 meters from the ground, the measured area that the camera could see was approximately a 1.5 meters by 2 meters rectangle, roughly amounting to an area of 3 square meters. With such a small area if the control algorithm faces some stability problems and the drone becomes unstable, the drone rapidly goes out of the field of view of the camera. Tests at a height of 0.5 meters, were also not made because when the drone hovers so close to the ground it suffers instability problems due to ground effects.

When the test results are presented, the grid points are ordered from top to bottom, left to right. This is always true unless stated otherwise. So the bottom left point of the grid is point number 1, the point immediately to the right is point number 2 and so on, until reaching the last point that will be the top right point.

7.1 USB Camera

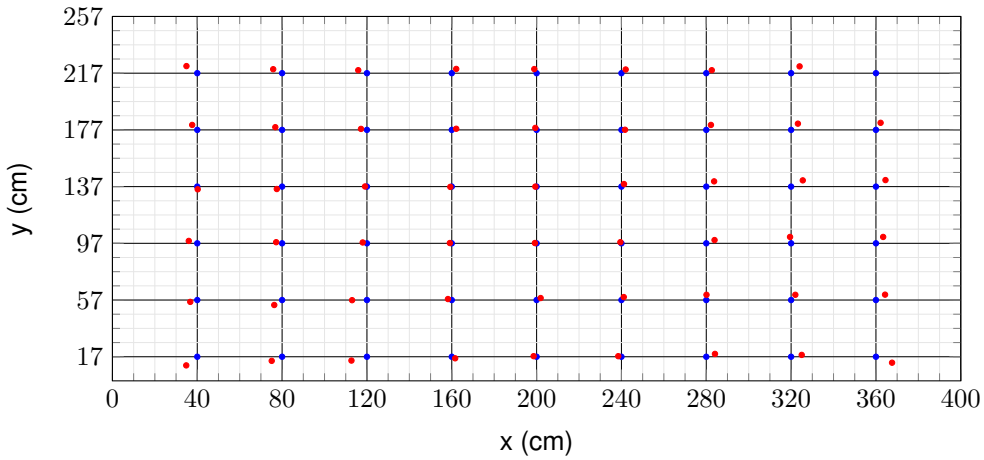
The measurements made with the USB camera are shown in figure 43.

Looking first to figure 43a, we can see the measured x and y coordinates for the 0 meters test. No Z coordinate data is present because when a USB camera connected the system is only capable of two-dimensional measurements. Visually, all measured points are close to their reference values, and the closer the measurements get to the center of the image, the more accurate the measurements are. This is to be expected because no steps were done to correct lens distortions that are not predicted by the pin-hole model. These lens distortions are more pronounced in the edges of the lens and less pronounced close to the center.

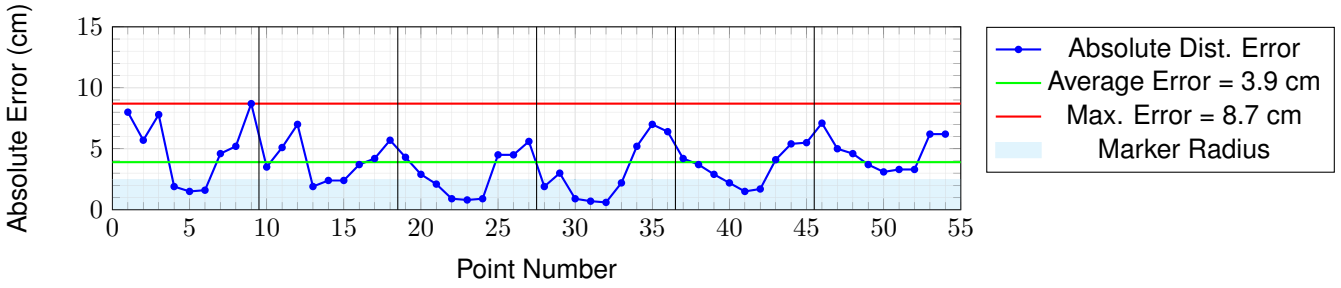
In figure 43b it's possible to better evaluate the accuracy of the measurements. This graph contains information regarding the absolute error of each measurement for each point. This absolute error is the two-dimensional Euclidean distance between the reference point and the measurement, given by:

$$Euclidian\,Distance = \sqrt{(X_{measured} - X_{ref})^2 + (Y_{measured} - Y_{ref})^2}$$

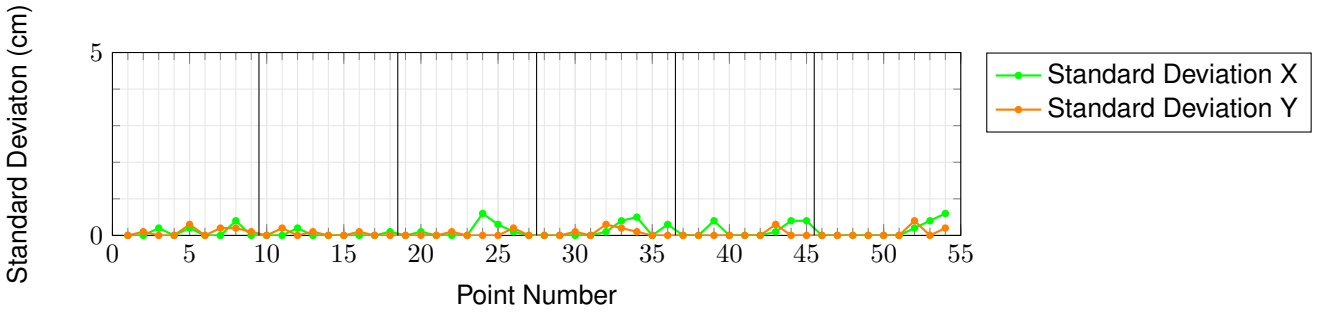
This graph also contains two horizontal lines that correspond to the absolute error mean and the maximum absolute error. The five black vertical lines are there for visual aid, and they separate the measured points by which reference grid row line they belong to. An analysis of this graph corroborates the last statements. The maximum error is 8.7 centimeters which is under the baseline value for success defined earlier of 12 centimeters. Furthermore, the average error is just 3.9 centimeters, meaning that most of the individual points have small errors. In this graph is even more clear the effects of lens distortion. In the middle of the vertical lines, the error dips considerable reaching errors below the 2.5 centimeters. The graph in figure 43c has the standard deviations relative to the 30 individual measurements of x and y coordinates acquired for each individual point. These standard deviations show how disperse these 30 measurements were. A large standard deviation reveals a system with a noisy output, that would require extra steps, like filtering, to extract reliable information. Fortunately, the standard deviations reveal very low noise levels. This is a direct consequence of the clear image produced by the camera. The errors in the location measures increase evenly along the image length, meaning while a marker is being tracked,



(a) X and Y measured coordinates



(b) Absolute distance error for each point



(c) Standard Deviation for each coordinate

Figure 43: USB location data test at 0 meters high

no abrupt changes in position will occur derived from lens distortions or calibration errors. This not only benefits the tracking algorithm performance but also guarantees that relative positions between points are maintained across the captured area.

This test reveals that the use of a USB webcam is within the ADIS system goals.

7.2 Kinect

The procedure to test the performance of the system with a Kinect is similar to the one involving a USB camera. The only difference is that a third coordinate must be taken into account, due to the three-dimensional sensing capabilities of the Kinect.

The graph from figure 44a shows the acquired x and y measurements against the reference grid. This

time it's also necessary to take into account the graph from figure 44b, which contains measurements relative to the height of the markers. From figure 44a, it's clear that the measured points are close to the reference grid points. Lens distortion is still noticeable in the edges of the image and increases the location error locally, however, these distortions appear gradually without abrupt changes that could confuse the tracking algorithm.

The graph from figure 44b, reveals the quality with which the Kinect detects depth. Ideally, the measured height should always be 0 meters, predictably this is not the case. Some spikes are noticeable along that graph which should contribute to an increase of the absolute error. Some values also seem to be floored artificially to the 0 meters level. This is caused by the localization system which never returns a negative height. This rule imposed by the remote camera software. As a consequence of this, when the Kinect senses a depth with enough error to be below the ground plane, the software corrects the measurement to zero. From figure 44c the actual accuracy of the system in this test can be objectively assessed. The maximum absolute error is 9.7 centimeters and the average error is 4.1 centimeters. These results are very similar to the results from the USB camera. Further analysis of the absolute error graph, once again, show the effects of lens distortion that produces lower errors the closer to the image center the marker is. Finally looking at the graph from figure 44d, it can be concluded that all of the three coordinates measurements have low noise levels, although, the depth data is more dispersed than the rest. This shows that the depth sensor has more inherent noise and inaccuracies than the camera sensor. All of the relevant baseline criteria for success were fulfilled by this test.

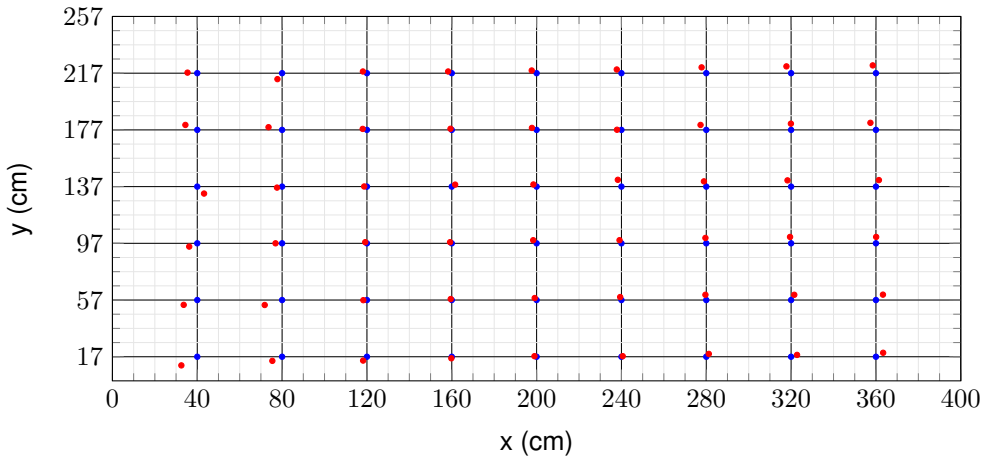
The results from the test with the marker one meter above the ground are in figure 45. The measured x and y coordinates continue to be close to the reference points, as seen in figure 45a. In this figure, some points do not have measurements. These points are missing because they were not seen by the Kinect. Due to the height increase, some points are out of the Kinect's field of view and get cropped. This will be evident on each subsequent test at higher heights. The z coordinate is the one that has higher error values. Looking to the absolute error graph, figure 45c, both the average error and the absolute error increased to 5.1 centimeters and 10.2 centimeters respectively. The maximum absolute error is still below 12 centimeters. This error increase is caused by the x and y coordinate correction based on the height of the drone. With a height of 0 meters, the Kinect behaves similarly to a USB camera, but when the height increases, the x and y corrections, equation 9, start to have accuracy errors introduced from the depth measurement, equation 11. These errors are not accounted for by the calibration process. Looking at the standard deviations, the height measurements have the most noise, however, the standard deviation reaches lower values than the 0 meters test. This is to be expected since the closer the marker is to the Kinect the better the depth measurements are. Yet, the three coordinates measurements have low noise levels.

The test with the marker at 1.5 meters from the ground is displayed in figure 46. This test has more cropped points than before due to the height increase. Analyzing the figure 46a, it is evident that the X and Y coordinates start to deviate from the reference points. The Z coordinate also displays errors, similar to the previous test. The absolute error only has marginal improvements compared to the 1 meter test. These improvements are due to the fact that most of the points are in the center of the lens. The

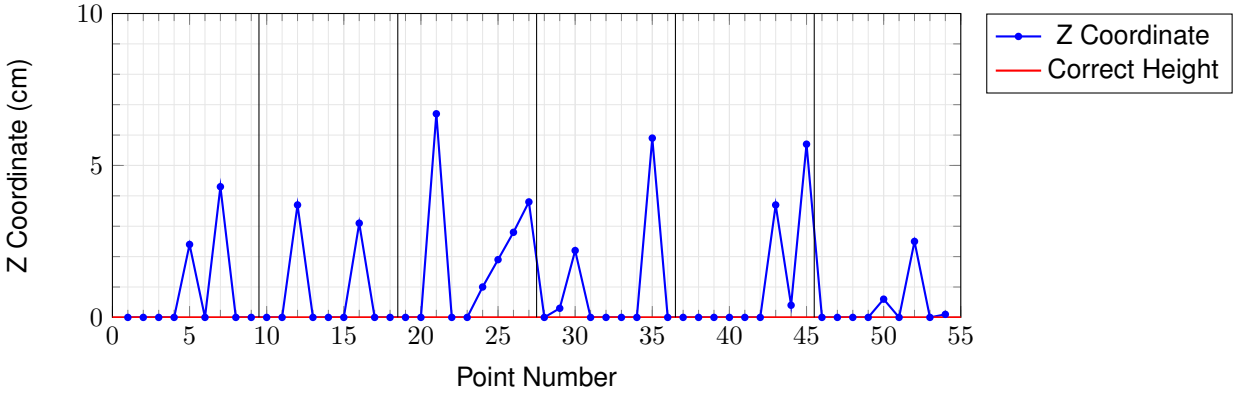
effects of the x and y coordinate correction are still the same as the ones from the one meter above the ground test. The average error decreased just 3 millimeters and the maximum absolute error decreased 8 millimeters. The standard deviations are also low as the last test, the standard deviations relative to the Z coordinate are a little slightly lower than before because the Kinect is closer to the marker.

The last test, where the marker was placed with a height of two meters relative to the ground is displayed in figure 47. Now only 23 points were measured comparing to the 54 points in the initial test. Looking at figure 47a, the measurements are still close to the reference points. The figure 47b shows that the height measurement contains most of the total error. This is confirmed by the absolute error graph (figure 47c) that accompanies the fluctuations of the height measurement error. None of the measurements falls inside the 2.5 centimeters interval, however, the average error is still low at 5.7 centimeters and the maximum error is 10.9, which is lower than the defined baseline for success of 12. This time the standard deviations for the three coordinates are very similar. Because the marker is closer to the Kinect the measurements have less noise and are more stable.

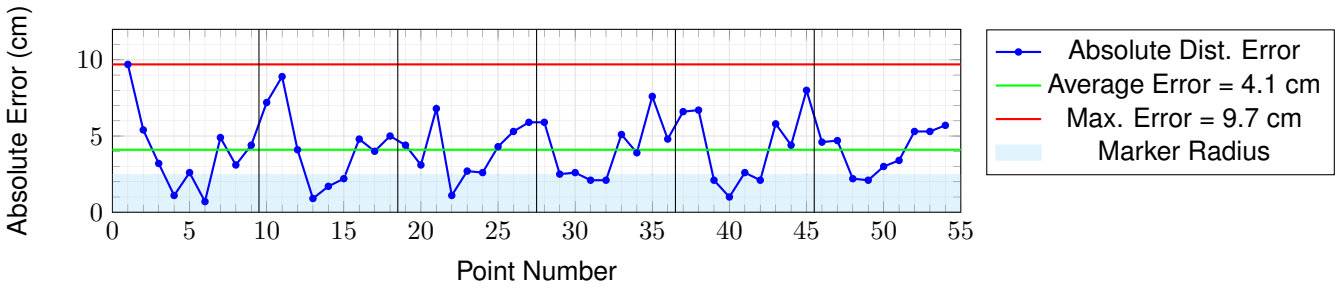
These four tests had very good results. The measurements have very low absolute errors and noise levels. Given that the marker is correctly detected, the localization data is reliable and can be used to navigate or control airborne or grounded vehicles. However, during the execution of these tests, two details were discovered. The Kinect depth sensor doesn't work well with very dark colors. These colors absorb the infrared pattern used by the Kinect and if the dark area is too large, there is not enough information available to calculate depth data. For this reason, vehicles with very dark colors don't work well with the ADIS system and must be painted. There is an additional problem regarding the marker surface area. Even with a diameter of 5 centimeters, depending on the distance to the Kinect, marker might not be big enough to produce reliable depth data. Usually, this is not a problem because the surface of the vehicle itself adds the missing area to calculate depth information. However, if the vehicle is too small or painted with dark colors (which reduces the usable surface area), this problem may appear and decrease the localization system performance.



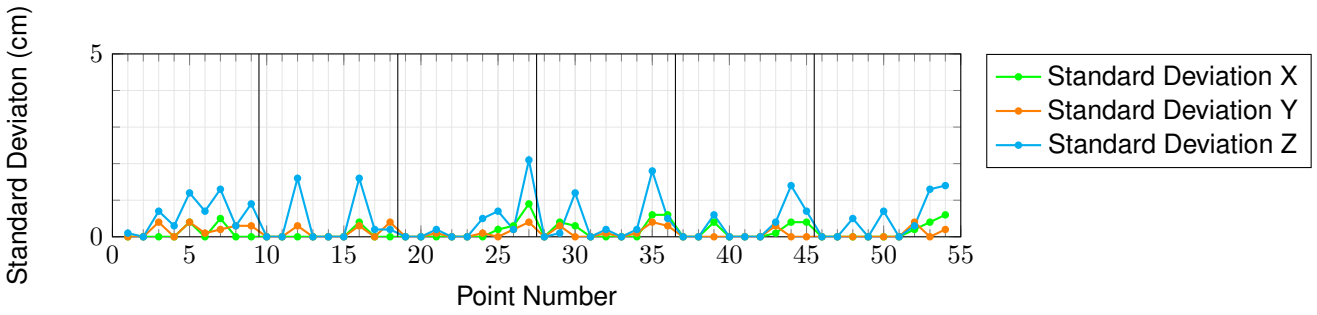
(a) X and Y measured coordinates



(b) Z measured coordinates

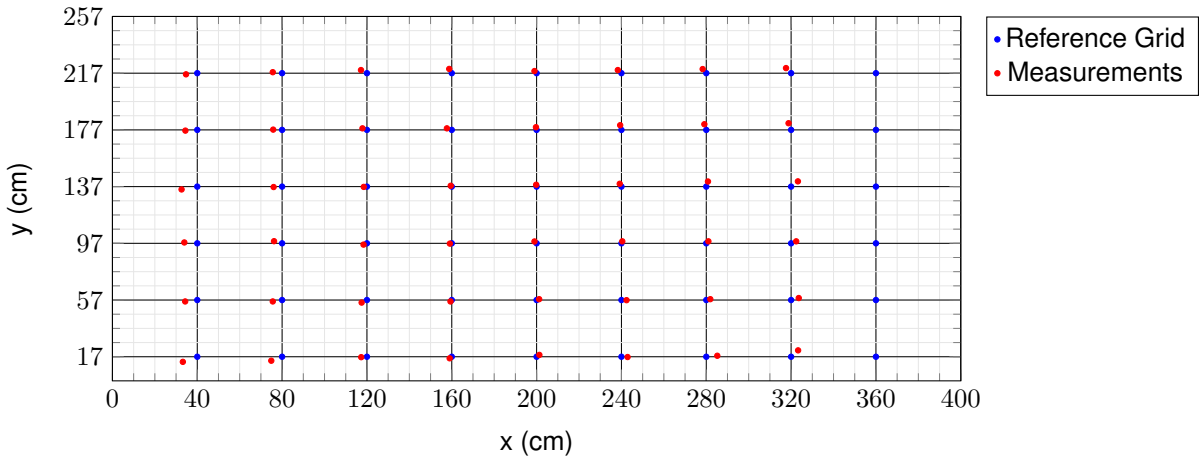


(c) Absolute distance error for each point

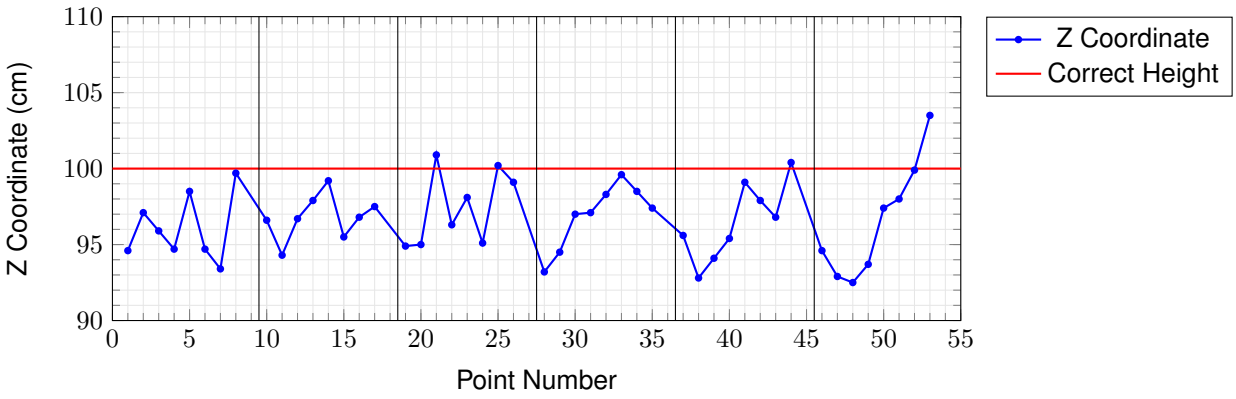


(d) Standard Deviation for each coordinate

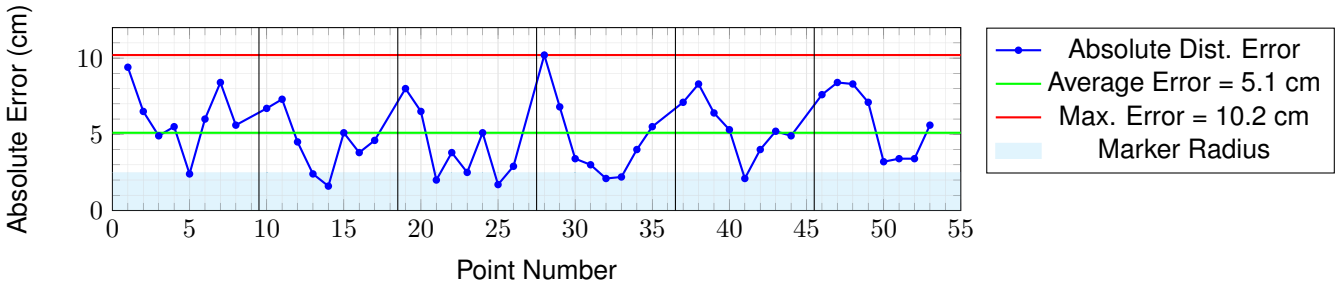
Figure 44: Kinect location data test at 0 meters from the ground



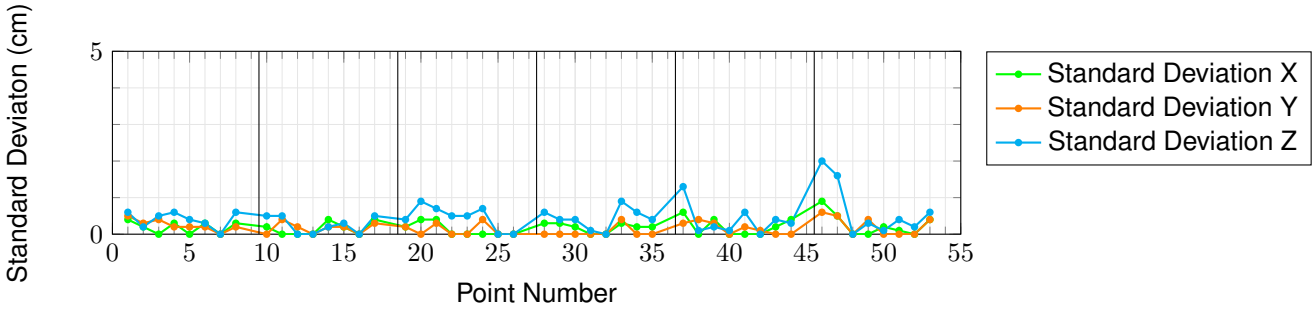
(a) X and Y measured coordinates



(b) Z measured coordinates

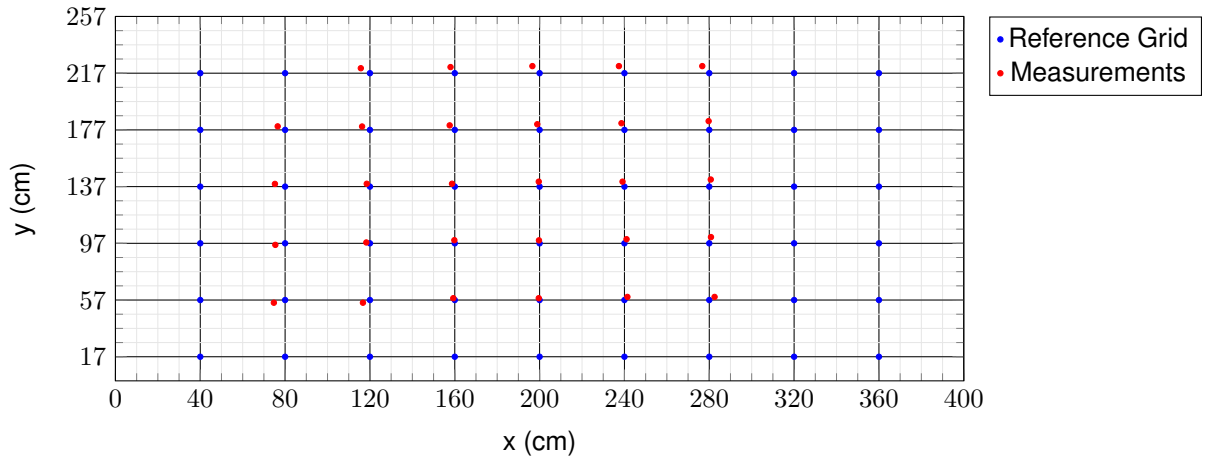


(c) Absolute distance error for each point

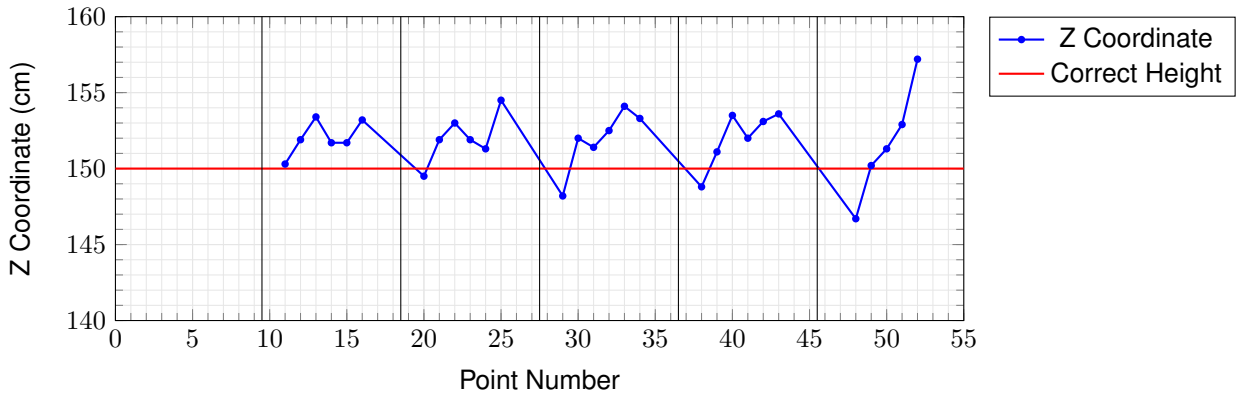


(d) Standard Deviation for each coordinate

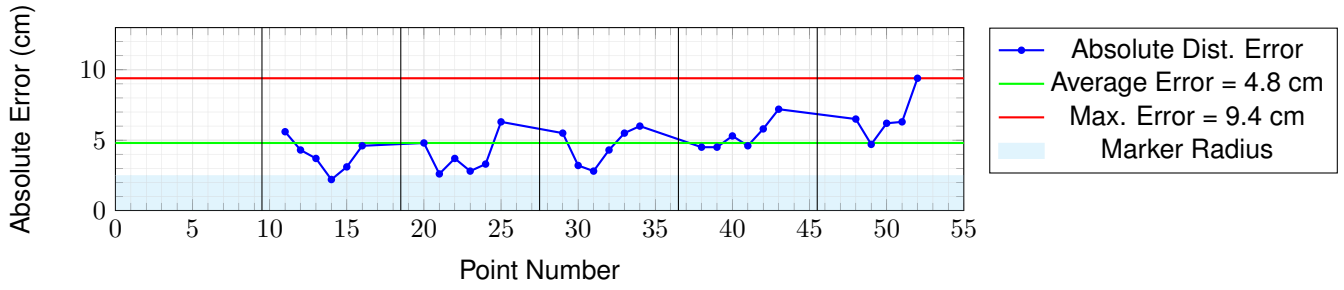
Figure 45: Kinect location data test at 1 meters from the ground



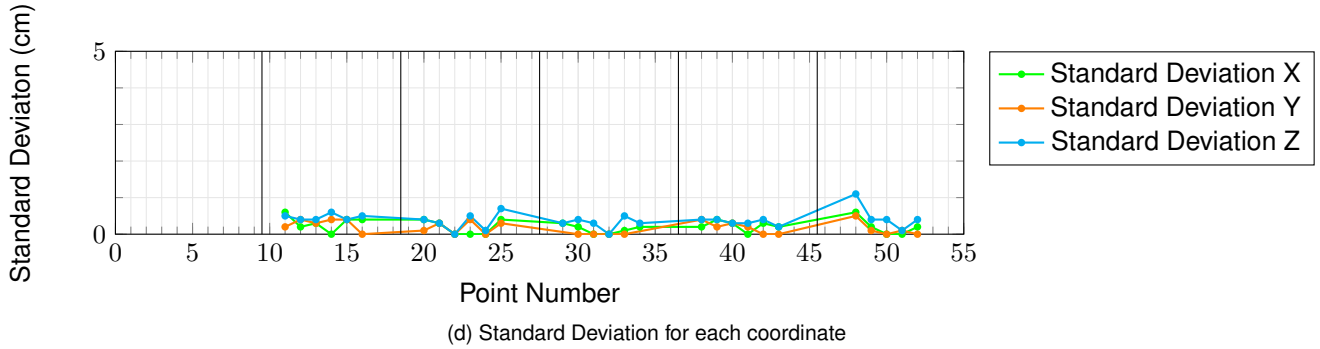
(a) X and Y measured coordinates



(b) Z measured coordinates

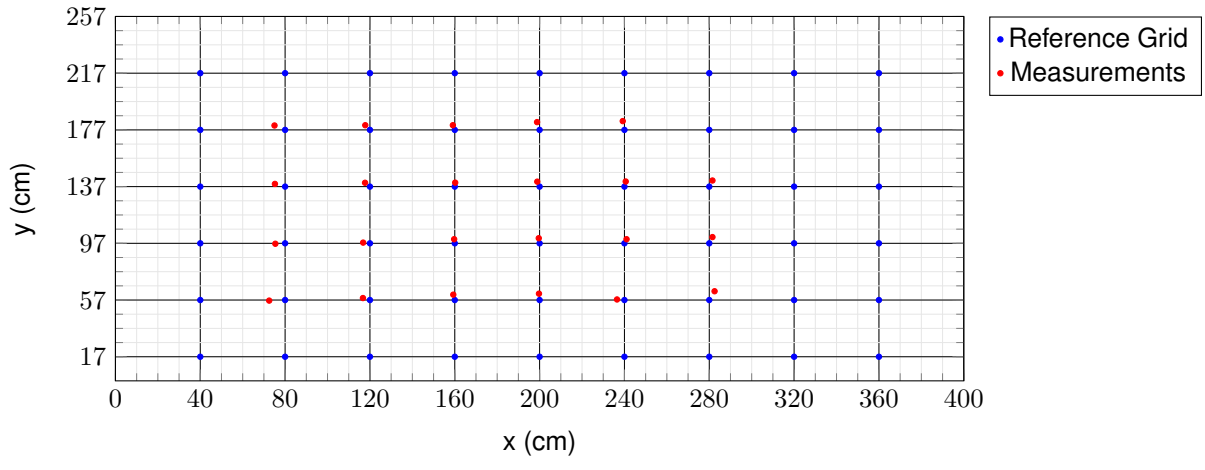


(c) Absolute distance error for each point

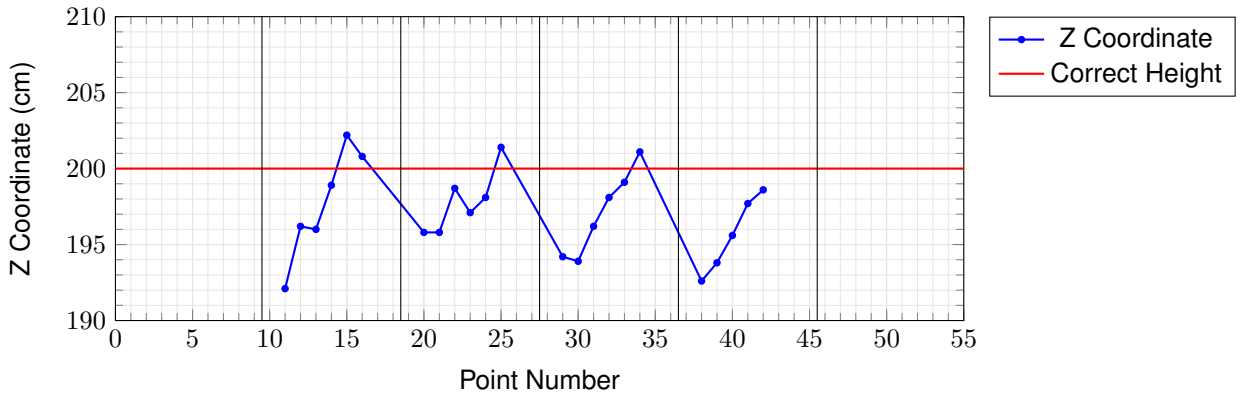


(d) Standard Deviation for each coordinate

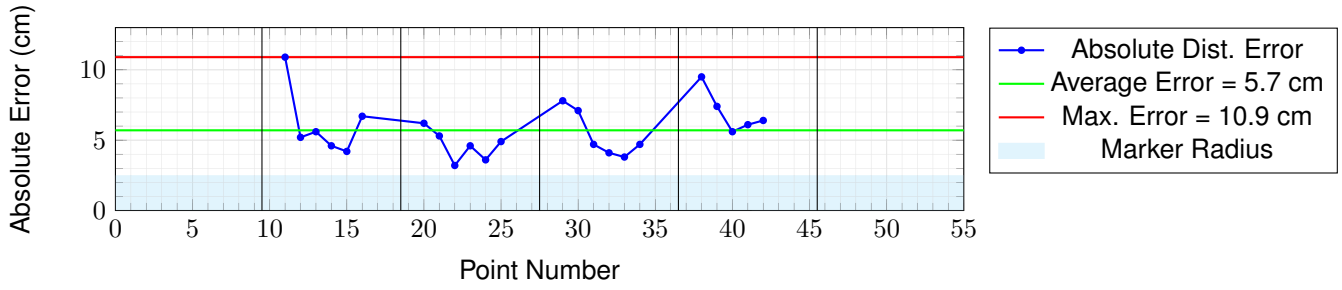
Figure 46: Kinect location data test at 1.5 meters from the ground



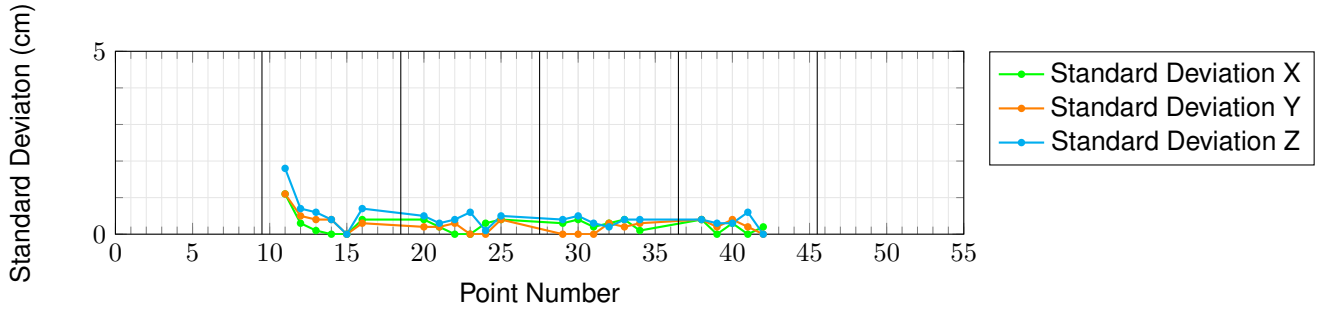
(a) X and Y measured coordinates



(b) Z measured coordinates



(c) Absolute distance error for each point



(d) Standard Deviation for each coordinate

Figure 47: Kinect location data test at 2 meters from the ground

7.3 Sampling Rate

The sampling rate of the localization system is an important aspect because most discrete controllers work relying on a fix sampling rate. Significant changes to the sampling rate while the controller is running can lead to critical changes in the controller output, resulting in poor stability and vehicle control. As mentioned before, the sampling rate for the ADIS system must be as close to 30 Hz as possible, but never go below 20 Hz. The biggest obstacle regarding a stable sampling rate is that the localization system software execution speed depends on multiple factors. It depends on the number of cameras being used, the number of markers being detected and processed, how many markers are in the overlap zones, on the sampling rate of the cameras or Kinect, on the load of the computer on which the software is running on, etc.

To control the sampling rate both the ADIS localization system software and the remote camera's software were programmed to never go over the 30 Hz. Their detection and processing loops are controlled by precise timers that slow down the execution speed when the loop will complete faster than 0.033 seconds. However, the real problem exists when the execution speed is bigger than 1/30 seconds. To observe how the system behaves under normal load, a simple test was made, that consisted of recording the frame rate of the system while detecting five markers with one Kinect. As seen in figure 48, the frame rate rarely drops below 30 frames per second and never reached any FPS number below 27 frames per second. Occasionally the sampling rates of 31 frames per second were measured. These are caused by rounding errors when calculating the sampling time.

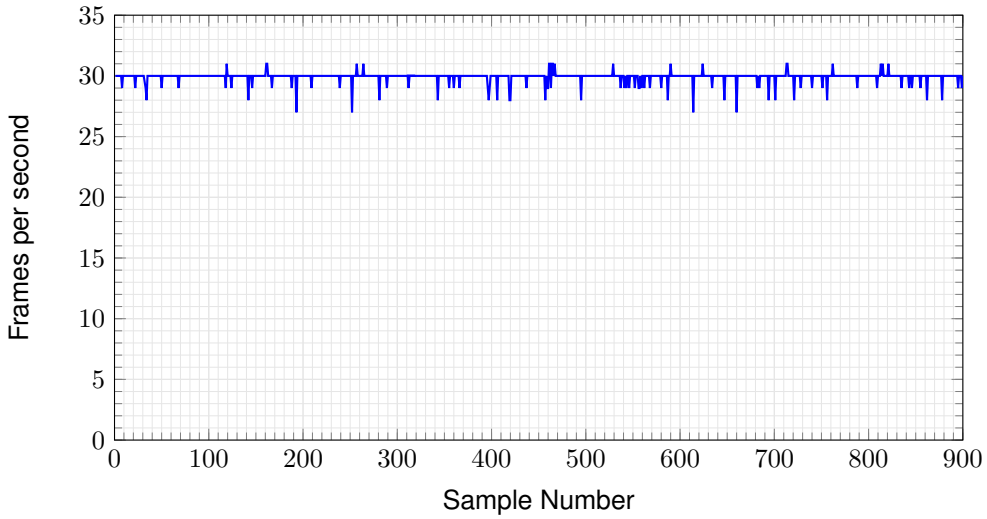


Figure 48: Frames per second test results

These last results are good and provide robust location data to work with. Yet when the current version of the ADIS system was installed in the laboratory, occasionally results like the ones in figure 49 were found.

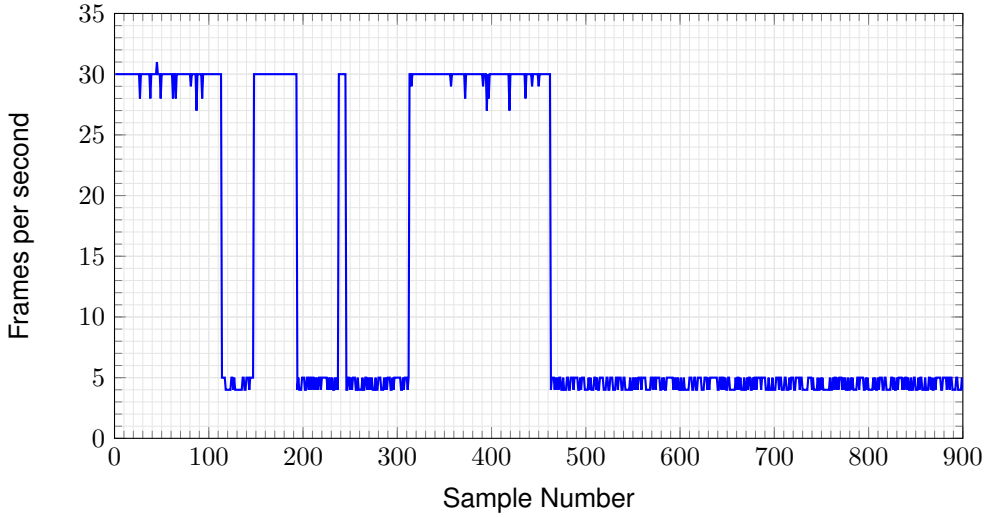


Figure 49: Frames per second test with bad connection results

These results were unexpected and initially, it was thought that the computer was the culprit. However, this frame rate drops didn't occur consistently and seemed independent of the computational load. After some debugging, it was discovered that this behavior appeared only when remote cameras were being used. This was still not consistent with the behavior localization system during its accuracy tests were it ran for hours with a stable frame rate around 30 frames per second. In the end, this behavior was caused by packet drops, when information is being sent from the remote camera to the localization system software. The UDP protocol used, does not guarantee packet delivery, which makes it extremely fast. However, this also leads to situations were due to interference some packets are lost through the network. These situations were happening because there were a lot of people in the laboratory where the system is installed. They were not connected to the ADIS Wi-Fi network, but because the 2.4 GHz Wi-Fi space is overloaded with other networks sharing the same Wi-Fi channels. This amount of Wi-Fi traffic interferes with the UDP communications causing lost packets. A solution for this problem would be either acquire a new router with stronger antennas that can receive packets even through heavy concurrent traffic. Another solution could be to switch all the Wi-Fi communications to the 5 GHz space, that is empty. Unfortunately, hardware that supports this frequency is very scarce at the moment. Because the 5 GHz frequency is more easily attenuated than the 2.4 GHz, it is rare for a drone to support the 5 GHz Wi-Fi frequency because the range would be affected, making this an unreliable option.

7.4 Communication Delays

The elapsed time between the detection of a marker and the arrival of that information to the controller is also an important factor that needs to be taken into account. Unfortunately to measure precisely communications delays, at least tone synchronized machines would be required. Solutions for this exist like the Network Time Protocol or NTP[51], which is able to synchronize a network of computers within a range of a few milliseconds. Unfortunately, there was not enough time to execute this kind of complex setup and quantify how large the delay is, instead a much more empirical test was used.

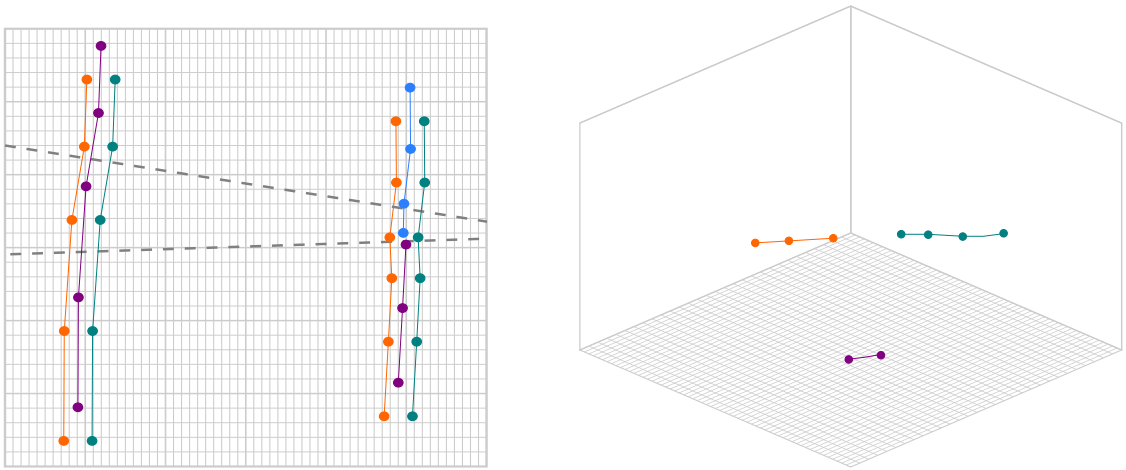
Because all of the vehicles compatible with the ADIS system are originally made to be operated

by humans, it is easy to conclude that human-like reflexes are enough to operate and navigate these vehicles. So, if a person with a marker on his hands cannot detect any delay between his movements and the data sent from the localization system, the communication delay can be considered acceptable. Although not objective at all, it was a simple test to verify if the system could handle the current vehicle fleet. This kind of test is completely unacceptable if a user wants to develop a controller that relies on very low latency communicates, but then again, the ADIS system was not developed with does kind of demands in mind. The distributed architecture of the ADIS system impairs this kind of applications because the information needs to travel through multiple devices, using wireless technologies before reaching the controller.

In short, the ADIS system was not designed for very low latency applications, but rather to cater to less demanding situations where an easy to use system is more beneficial than a complex but high-performance system. This empirical test, in the context of the ADIS system applications, is enough to provide a good degree of certainty that the system has an acceptable latency to control and navigate its compatible vehicles.

7.5 Tracking and Overlap Zones

The performance of the tracking algorithm must also be analyzed. To test how the tracking algorithm performs, two similar tests were executed. One consisted of using three cameras to cover a large area, and then drive a car with three markers on top and see if any marker was lost or switched. This test evaluated the performance if the tracking algorithm in a 2D setup. The second test aimed to evaluate the tracking algorithm in a 3D setup. It used one remote camera equipped with a Kinect to track three markers on top of a drone. Both tests attested to the reliability of the tracking algorithm, however, each revealed two problems.



(a) Overlap Zones problem illustration

(b) Lost of depth problem illustration

Figure 50: Tracking Algorithm Performance Issues

Figure 50a illustrates one problem related to 2D tracking. When the car passes over a wide section of an overlap zone, the tracking algorithm performance was good and never missed a point. Yet when

the car passed through a thinner section of the overlap zone one marker was consistently lost and identified has a new one. Analyzing frame by frame the behavior of the localization software while the car passed over the thinner section of overlap zone revealed when the car reaches the neighborhood of the thin overlap zone, a new track appears because one marker is being detected by two cameras at the same time. This should only happen inside the overlap zone, where the algorithm4 would eliminate this extra point. Unfortunately, small errors in the calculation of the boundaries of the overlap zone placed this thin section slightly to the side. This leads to the algorithm4 not taking action at the right time, leaving the extra point. Because this overlap zone is so thin, the real world boundaries and the calculated boundaries do not overlap, meaning that the localization software never eliminates the new marker. This stresses that good camera placement and careful calibration are always required. When the overlap zones are wider, the real boundaries and calculated boundaries will overlap, which will trigger the algorithm4, eliminating the extra points regardless. Another problem still related to these overlap zones is that algorithm4 affects the overall system performance. Even after optimization, the sampling time inside these zones may be slower than desired. This should be taken into account and the vehicle should spend as little time as possible in these zones.

The last two problems encountered are related to Kinect depth performance. Usually, in dark-colored areas, the Kinect cannot measure depth information. As illustrated by figure 50b, this could lead to abrupt changes in the marker location that the tracking algorithm assumes as new markers appearing into the frame. This problem was solved by feeding into the tracking algorithm the X an Y coordinates still not corrected for the Z coordinate. This means that the tracking algorithm is the same as the one used for 2D tracking and during testing it displayed the same performance. The final encountered problem is related to overlap zones with the Kinect. Unfortunately, these zones are not possible to implement when a Kinect is used. When to Kinects are pointing at the same area, their infrared projected patterns interfere with each other. This interference prevents either Kinect to acquire depth information. This problem could be solved by synchronizing the depth capture process in such a way that each Kinects projected their infrared pattern alternately. However, this would require a way to rapidly switch the infrared project on and off, which is not currently supported by the Kinect driver used.

8 Conclusion

This thesis aimed to create a comprehensive system for the development of navigation and control algorithms targeted at mobile vehicles. The main idea behind a system like this is to provide a stable development environment, capable of accurate and reliable data collection for educational or research purposes. The vehicles used were both commercially available drones and radio controlled cars.

A system like this presents itself with three main challenges, how to locate and track more than one test vehicle in an indoor environment, how to communicate and control these vehicles and how to provide a simple, but yet useful user interface. The solution to the first problem was to build an indoor localization system. Because solutions based on GPS signals are often not suited for indoor applications, another approach using cameras and depth sensors was used. Various iterations of this concept lead to a final architecture, composed by remote cameras connected to Kinect sensors. These are able to detect specific markers in three-dimensional space. After the tests presented in section 7, it is safe to say that the localization system has enough accuracy to locate either a car or a flying drone. Nevertheless, problems still exist regarding overlap zones, because two Kinects interfere with each other when their field of view intercepts. Because Kinects, among other techniques, use a projected infrared pattern to perceive depth, when two of these patterns overlap, both Kinects are unable to output depth information. This problem prevents the use of multiple Kinects to cover an area uninterruptedly. The current solution is to replace the Kinects with USB cameras, which provides similar accuracy, but location data in two dimensions.

To follow the test vehicles through time, a tracking algorithm was implemented. It assigns a Kalman filter to every point detected. This Kalman filter predicts the marker future locations using previously recorded location data. These estimated positions allow matching points detected in the next frame with the ones detected in the current frame. This tracking technique works well in the right conditions, unfortunately, the overlap zones are still a problem. To handle these overlap zones, a specialized algorithm was integrated that uses the Procrustes method to find and delete duplicated points, however, this algorithm only starts when a point is detected as being within an overlap zone. Unfortunately, due to small inconsistencies between calibrated cameras, these overlap zones are calculated with small errors in their boundaries locations. This, in certain conditions, originates duplicated points that confuse the tracking algorithm and cause point matching errors.

To send and retrieve information from vehicles to fulfill any kind of communication with a remote device, a Wi-Fi based communication network was created. Depending on the type of information sent through the network, different transport layer protocols were implemented. Most of the time the TCP protocol is used. TCP guarantees the delivery of a message to its destination, making it ideal to send messages containing control commands or any type of information that does not require to be delivered with the least amount of latency possible. Latency can be an issue for example, when a remote camera is sending location data to the computer running the localization software. In this case, all messages are sent using UDP. UDP is faster than TCP because it does not guarantee that a packet is delivered to its destination. This causes issues, in situations where there is a lot of Wi-Fi traffic. This excessive traffic

does not necessarily need to be on the ADIS Wi-Fi network. Because the laboratory has a very high density of Wi-Fi signals, interference coming from separate networks impacts the system performance greatly. This impact can lead to large amounts of dropped UDP packets, slowing down the localization system. Thankfully, this problem is not always present, because it requires a big amount of Wi-Fi traffic.

Regarding vehicle integration, this thesis only addressed flying vehicles, more specifically drones. The drone model selected was the AR.Drone 2.0 from Parrot. In the end, this choice turned out to be a success. This drone model is durable, and it endured testing, without breaking. Another relevant characteristic was the official and unofficial community support around it. Simulink controllers are available for it, directly provided or promoted by MathWorks. This helped the overall system integration considerably. Nonetheless, the USB port and embedded Linux based operating system were the keys for a successful integration within the ADIS system. The lack of WPA2 support and no easy repair method were definitely challenges that took a time to overcome correctly, but the use of the $chroot$ method to change to a purposely made file system contained inside an external USB flash drive, provided the desired degree of security and flexibility.

The software workflow and final user interface can also be considered a success. This conclusion is quite subjective, but if we consider how simple the user interface appears to be and the amount of functionality it provides, it is hard not to consider this as a good and necessary addition to this thesis final product. No script or text-based tool can provide the amount of information provided by this simple GUI, which contains real-time video and depth streaming, location data visualization capabilities, as well as visual cues to guide the user throughout the mandated workflow.

9 Future Work

For as much as this thesis has created from scratch, there still is a big amount of work that can be done to further improve and expand the feature set of the ADIS system. Algorithm optimizations can definitely be done to not only improve general performance but also to solve problems mentioned previously. The overlap zones issue can be easily solved by artificially expand the calculated boundaries. This would force the algorithm that identifies and deletes duplicated points to start earlier, removing the extra points that confuse the tracking system. However, further performance optimizations to this algorithm are also needed. The remote cameras can also be enhanced. A smaller form factor would definitely be a big addition to the system flexibility. In terms of size, the ideal processing unit for a remote camera would be a single board computer. Usually, these kind of computers are not powerful, but with enough optimization, they have enough processing power to run the necessary vision algorithms. One of these optimizations could be a complete rewrite from Python to C/C++ of the image processing algorithms used by the remote cameras. Because C/C++, support real multi-threading, true multi-core algorithms can be written. Many image processing techniques can be highly parallelized to fully use all multiple cores available, greatly improving performance. The Kinect driver is also a point that could be improved in the future. If the ability to turn the infrared projected pattern rapidly on and off was added, the overlap zones interference between Kinects could be resolved. Together with a synchronization system, Kinects with overlap zones could be coordinated to only acquired depth data and project its infrared pattern one at a time, effectively eliminating interference.

Regarding network performance, a better router is undoubtedly necessary. Excessive Wi-Fi traffic can occasionally slow down the ADIS system performance rendering it unusable. A higher quality router can withstand noisier environments and prevent UDP packet drops. Change from UDP to TCP would eliminate packet drops, but would probably not solve the low sample rate problem, since TCP would equally suffer from interference slowing down the communications as well.

Future perspectives include the use of ADIS system as the position sensor for navigation and control or mobile robots, exploiting its dynamics and convergence to solutions with proved stability using ADIS for the update of state variables phases.

References

[1] R. Hartley and A. Zisserman. *Multiple View Geomerty in Computer Vision*. Cambridge University Press, second edition, 2003.

[2] Reza AW, Geok TK. 2009. Investigation of indoor location sensing via RFID reader network utilizing grid covering algorithm. Wireless Personal Communications 49(1):67-80.

[3] Bahl, P., & Padmanabhan, V. N. (2000). RADAR: An in building RF-based user location and tracking system. In Proceedings of the IEEE Conference on Computer Communications (INFOCOM), Vol. 2, pp. 775–784, 2000.

[4] Chang N, Rashidzadeh R, Ahmadi M. 2010. *Robust indoor positioning using differential Wi-Fi access points*. IEEE Transactions on Consumer Electronics 56(3):1860-7.

[5] Schweinzer H, Kaniak G. 2010. *Ultrasonic device localization and its potential for wireless sensor network security*. Control Engineering Practice 18(8):852-62.

[6] Lowe, David G. (2004). "Distinctive Image Features from Scale-Invariant Keypoints". International Journal of Computer Vision. 60 (2): 91–110.

[7] HHerbert Bay, Andreas Ess, Tinne Tuytelaars, and Luc Van Gool, "Speeded Up Robust Features", ETH Zurich, Katholieke Universiteit Leuven.

[8] N. Dalal, B. Triggs, "Histograms of Oriented Gradients for Human Detection", Proc. IEEE Conf. Computer Vision and Pattern Recognition, 2005.

[9] Gary Bradski´ "Computer Vision Face Tracking For Use in a Perceptual User Interface", Intel Technology Journal, No. Q2, 1998.

[10] B. D. Lucas and T. Kanade, "An iterative image registration technique with an application to stereo vision", Proceedings of Imaging Understanding Workshop, pages 121-130, 1981.

[11] Logitech C615 “http://www.logitech.com/en-us/product/hd-webcam-c615”

[12] HP HD 2300 ”http://www8.hp.com/pt/pt/products/oas/product-detail.html?oid=5190035”

[13] RGB to HSV algorithm “https://www.mathworks.com/help/matlab/ref/rgb2hsv.html”

[14] Image Processing User’s Guide “https://www.mathworks.com/help/pdf_doc/images/images_tb.pdf”

[15] $blob\_analysis$ function “https://www.mathworks.com/help/vision/ref/vision.blobanalysis-class.html#bsftx88”

[16] Haralick, Robert M., and Linda G. Shapiro, *Computer and Robot Vision*, Volume I, Addison-Wesley, 1992, pp. 28-48.

[17] Sedgewick, Robert, *Algorithms in C*, 3rd Ed., Addison-Wesley, 1998, pp. 11-20.

[18] M. A. Fischler and R. C. Bolles, “Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography,” Communications of the ACM, vol. 24, no. 6, pp. 381–395, 1981.

[19] Xbox 360 release date ”http://gizmodo.com/5563148/microsoft-xbox-360-kinect-launches-november-4”

[20] PrimeSense patent, Depth Mapping Using Projected Patterns, “http://www.patentsencyclopedia.com/app/20100118123”

[21] PrimeSense patent, Depth Mapping Using Multi-beam Illumination, “http://www.patentsencyclopedia.com/app/20100020078”

[22] PrimeSense patent, Integrated Processor For 3D Mapping, “http://www.patentsencyclopedia.com/app/20100007717”

[23] Intel J1900 Specifications “https://ark.intel.com/products/78867/Intel-Celeron-Processor-J1900-2M-Cache-up-to-2_42-GHz”

[24] About OpenCV, “http://opencv.org/about.html”

[25] Debian Wiki Wi-Fi “https://wiki.debian.org/Wi-Fi/HowToUse#Command_Line”

[26] Debian Wiki *systemd, “https://wiki.debian.org/systemd”*

[27] OpenKinect community page, “https://openkinect.org”

[28] Libfreenect GitHub, “https://github.com/OpenKinect/libfreenect”

[29] Cython about page, “http://cython.org”

[30] Libfreenect imaging documentation, ”https://openkinect.org/wiki/Imaging_Information”

[31] Kendall, David G. "A Survey of the Statistical Theory of Shape." Statistical Science. Vol. 4, No. 2, 1989, pp. 87–99

[32] Bookstein, Fred L. Morphometric Tools for Landmark Data. Cambridge, UK: Cambridge University Press, 1991

[33] Seber, G. A. F. Multivariate Observations. Hoboken, NJ: John Wiley & Sons, Inc., 1984.

[34] Parrot AR.Drone 2 Official Page, “https://www.parrot.com/us/drones/parrot-ardrone-20-elite-edition#parrot-ardrone-20-elite-edition”

[35] AR.Drone 2 Offical Global Page, “http://global.parrot.com/au/products/ardrone-2/”

[36] AR.Drone 2.0 motor Specifications, “https://www.parrot.com/uk/spareparts/drones/parrot-ardrone-20-brushless-motor#parrot-ardrone-20-brushless-motor-details”

[37] AR.Drone 2.0 battery Specifications, “https://www.parrot.com/uk/spareparts/drones/hd-battery-ardrone-20#undefined”

[38] Stephane Piskorsk, Nicolas Brulez, Pierre Eline, Frederic D’Haeyer. *AR.Drone Developer* Guide. Version 2.0.

[39] Wasim Ahmad Bhat and S.M.K. Quadri. *Review of FAT data structure of FAT32 file system*, Oriental Journal of Computer Science & Technology. Vol. 3.

[40] Erez Zadok, Ion Badulescu, and Alex Shender. *Extending File Systems Using Stackable Templates*. Proceedings of the USENIX Annual Technical Conference, June 6-11, 1999

[41] Russon, Richard; Fledel, Yuval. *NTFS Documentation*.

[42] Nmap official website “https://nmap.org”

[43] Debian online documentation for debootstrap, “https://wiki.debian.org/EmDebian/CrossDebootstrap”

[44] QEMU project home page, “http://www.qemu-project.org/”

[45] VirtualBox home page, “https://www.virtualbox.org/”

[46] Windows 10 Insider Preview Build 14951 release blog post, “https://blogs.windows.com/windowsexperience/2016/10/19/announcing-windows-10-insider-preview-build-14951-for-mobile-and-pc/#3XcLATCiID0qGG84.97”

[47] AR Drone Simulink Development-Kit V1.1 ,”https://www.mathworks.com/matlabcentral/fileexchange/43719-ar-drone-simulink-development-kit-v1-1?requestedDomain=www.mathworks.com”

[48] AR.Drone 2.0 Support from Embedded Coder,”https://www.mathworks.com/matlabcentral/fileexchange/48558-ar-drone-2-0-support-from-embedded-coder”

[49] PS-Drone API download page, “http://www.playsheep.de/drone/downloads.html”

[50] David C. Plummer (November 1982). "RFC 826, An Ethernet Address Resolution Protocol -- or -- Converting Network Protocol Addresses to 48.bit Ethernet Address for Transmission on Ethernet Hardware". Internet Engineering Task Force, Network Working Group.

[51] David L. Mills (12 December 2010). Computer Network Time Synchronization: The Network Time Protocol. Taylor & Francis. pp. 12–. ISBN 978-0-8493-5805-0.

[52] MathWorks, Computer Vision System Toolbox User’s Guide, March 2017

[53] W. Gander and J. Hrebicek. Solving Problems in Scientific Computing using Matlab and Maple . Springer Verlag, 1993.

[54] R. A. Pilgrim, Munkres’ Assignment Algorithm. Modified for Rectangular Matrices, Course notes, Murray State University.

[55] M. Nuno. Integration of RC Vehicles in a Robotic Arena. Master Thesis, Instituto Superior Técnico, Universidade de Lisboa, November 2016

[56] R. E. Kalman, A New Approach to Linear Filtering and Prediction Problems,1960. R. E. Kalman, A New Approach to Linear Filtering and Prediction Problems,1960